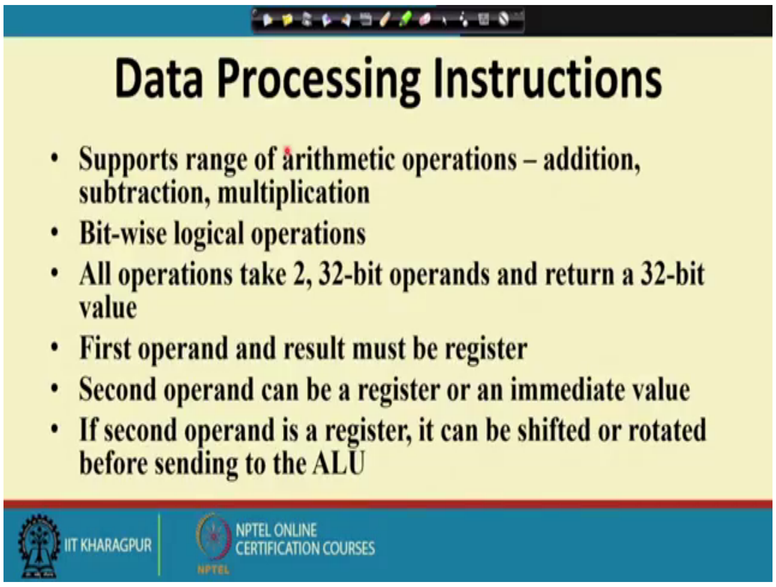
Microprocessors and Microcontrollers

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

Lecture – 45
ARM (Contd.)

So, the data processing instruction that is a first category of instructions that we have for ARM; So, we it supports arithmetic large number of arithmetic operations, like addition subtraction and multiplication.

(Refer Slide Time: 00:25)



Data Processing Instructions

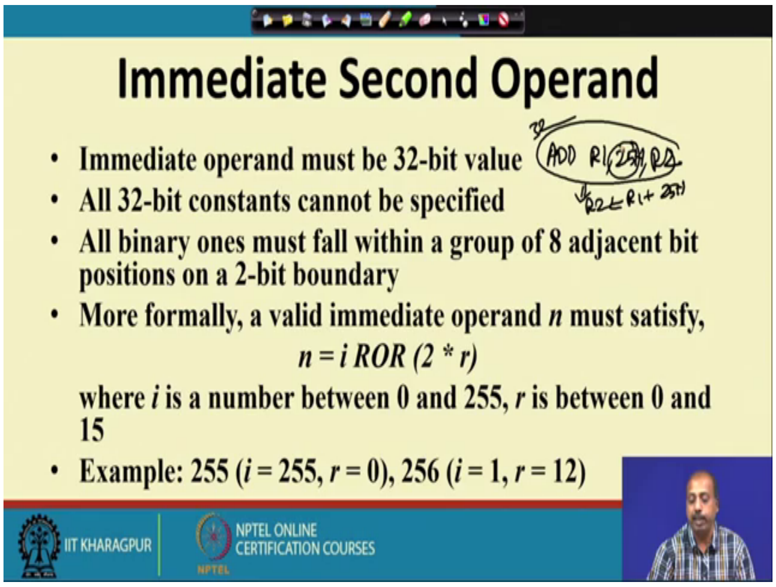
- Supports range of arithmetic operations – addition, subtraction, multiplication
- Bit-wise logical operations
- All operations take 2, 32-bit operands and return a 32-bit value
- First operand and result must be register
- Second operand can be a register or an immediate value
- If second operand is a register, it can be shifted or rotated before sending to the ALU

So, one thing that you note here that division is not supported. So, that may be taken as a disadvantage, but the problem is that supporting division. So, the division algorithm has to be implemented and that way the architecture will become complex. So, if you really need division then you can possibly do it using software and many cases we may not need it. So, multi this was addition, subtraction and multiplication. So, these are the 3 instructions that are supported and the multiplication has been made every powerful so that we will see that. And that that takes care of this division operation a to some extent.

Then the bit wise logical operation, that is there. So, we can do logical operation then all operations they take 2, 32-bit operands and return one 32-bit value. So, the internal phase is 32 processor, so it all the operations around 32-bit data. First operand and the result must be register. So, this is the important thing that to the first operand that must be a register and the result must also be a register. So, that is why it is a register type of architecture. Then the second operand it can be a register or it can be an immediate value and if the second operand is a register it can be shifted or rotated before sending to the ALU. So, the first operand is register, and the second operand can be register or immediate. So, if it is a register the value will be taken from the register directly if it is an immediate operand. So, that way the value will be available in the instruction opcode; in the instruction itself. So, the value will be taken from there and if for registers, so you can do this shifting rotating like that

So, that you remember that there is a barrel shifter and it is connected to the register file. So, from there it will be when it is coming. So, it will be taking that register, the register value will come to that barrel shifter and that barrel shifter will be shifting the value for coming in the from coming from the register. So, that way we can have this shift operation as a part of it.

(Refer Slide Time: 02:34)



Immediate Second Operand

- Immediate operand must be 32-bit value
- All 32-bit constants cannot be specified
- All binary ones must fall within a group of 8 adjacent bit positions on a 2-bit boundary
- More formally, a valid immediate operand *n* must satisfy,

  $$n = i \; ROR \; (2 * r)$$

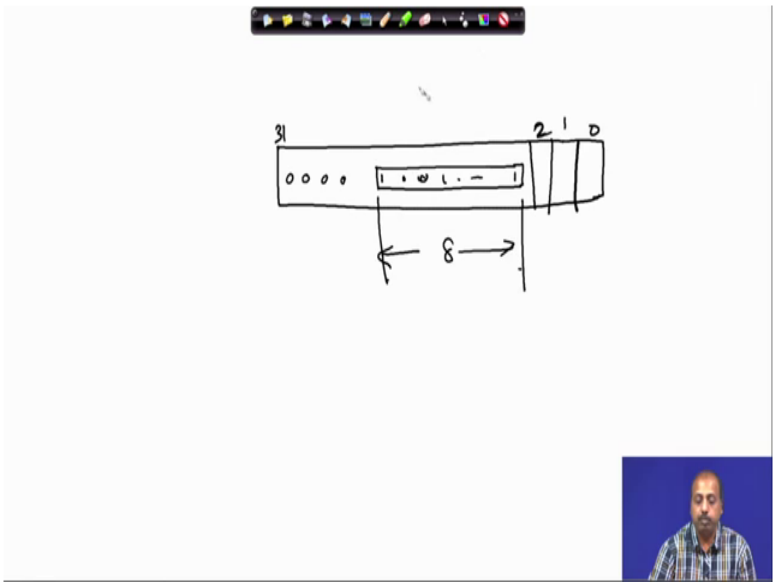
  where *i* is a number between 0 and 255, *r* is between 0 and 15
- Example: 255 (*i* = 255, *r* = 0), 256 (*i* = 1, *r* = 12)

But if this second operand is an immediate operand then the immediate operand must be 32-bit value. So, they must have a 32-bit value. And naturally we have a catch here because the instruction itself is 32-bit and then I am telling that the immediate operand should also be 32-bit. So, it is like this suppose I have got an add instruction. So, I say that add R1 comma say some 32-bit value say 25 hex, and the result be stored in R2, meaning R2 will get R1 plus 25 hex.

Now, you see that whole instruction is 32-bit the coding of this whole instruction is 32-bit and it says that this immediate operand is 25 hex. So, this should also be coded as a 32-bit instruction, 32-bit numbers and that is a problem ok. So, it naturally you cannot represent all the numbers in this 32-bit value and it what it what this arm people have done. So, they have allowed only a category of 32, 32-bit numbers that can be used as the second operand, how let us see.

So, all 32-bit constants cannot be specified. So, the numbers which are allowed then all binary ones must fall within a group of 8 adjacent bit positions and on a 2-bit boundary. So, what does it mean? Is like this, we have got this thing.

(Refer Slide Time: 04:06)



Suppose this is the 32 number that I have got and in that number this is bit number 0 this is bit number 31. So, all this bits sets as 0 in between you have got a block ok. So, if you mark the left most occurrence of 1 and the right most occurrence of 1 ok, in between the bits may be 0 or 1 whatever. So, this should be 8 bit this should be 8 bit.

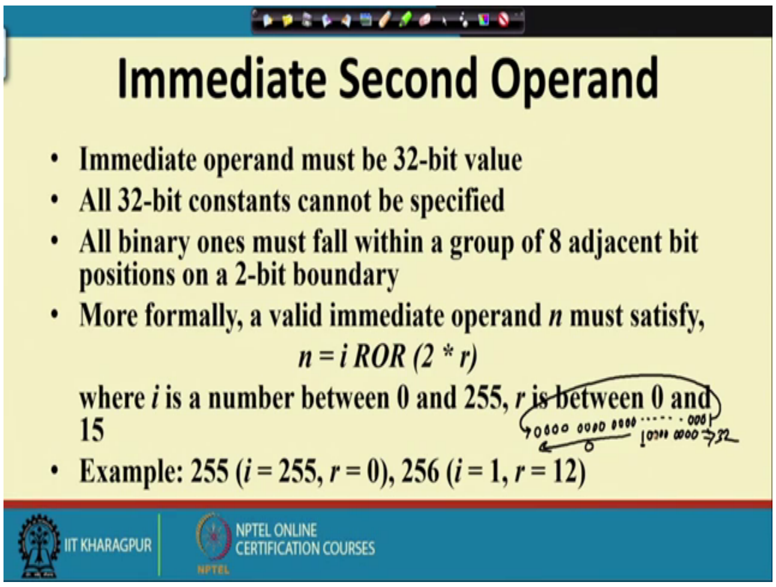
So, the number the 1's should be distributed in a block of 8 bits and this block it cannot start at arbitrary point. So, it has to be at 2-bit boundaries. Like it can start at location 0, but it cannot start at location 1. So, it can start at location 2. So, like that it can go. So, this start point and end point. So, they are they should be some multiple of 2.

So, only those type of numbers. So, will be able to represent and how this can be done. So, if we just look back it says that a valid immediate operand n. So, it is like this i rotate right 2 into r, where i is a number between 0 and 255. So, it is an 8 bit number. So, this is between 0 and 255 this is between 0 and 255, and this r is between 0 and 15. So, if it is 0 and 15. So, it is rotate right by up to it can be 0 bit to 30 bits.

So, up so many positions it can be shifted, but this r value r can be equal to 0. So, in that case no rotation r equal to 1, it is rotation by 2 bits r equal to 2, so rotation by 4 bits. So, you see that it is getting shifted by the even boundaries. So, it you cannot have the number in odd boundaries.

So, this for example, the number 255, so 255 you can represent in this format because I you can take i equal to 255 and r equal to 0 that is fine. Similarly you can also represent 256 because I can take i equal to 1 and r equal to 12. So, if I take i equal to 1 and r equal to 12 then what will happen is that. So, this i equal to 1, number is so, if I if I take the number, it is 0 0 0 0, 0 0 0 0, 0 0 0 0. So, this way ultimately 0 0 0 1

(Refer Slide Time: 06:34)



Immediate Second Operand

- Immediate operand must be 32-bit value
- All 32-bit constants cannot be specified
- All binary ones must fall within a group of 8 adjacent bit positions on a 2-bit boundary
- More formally, a valid immediate operand *n* must satisfy,

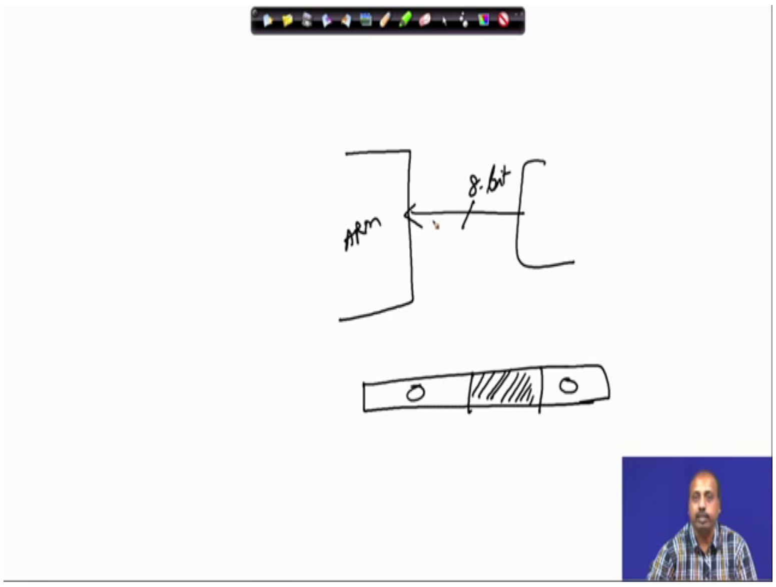
  $n = i \ ROR \ (2 * r)$

  where *i* is a number between 0 and 255, *r* is between 0 and 15
- Example: 255 ($i = 255, r = 0$), 256 ($i = 1, r = 12$)

0000 0000 0000 ····· 0001
0
1000 0000 → 32
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

And I am rotating it right by 24 bits. So, it is 2 into r, 24 bits. So, I will be rotating it like this. So, if I rotate it then this 1 will come to the position. So, where it is 1 then there will be 0 0 0 0, 0 0 0 0. So, this is the number 256 and all this bits will be 0. So, you do the rotation and see that it will be a number like this ok. So, this will be this is the 32-bit number and here this bit number 9 will be 1 and all other bits are 0. So, this way we can represent the number 256 in the in this particular format fine. So, all those numbers that can be feted into this particular representation, they will be allowed as immediate operand others are not. Now, why is it why is it useful?

(Refer Slide Time: 07:44)

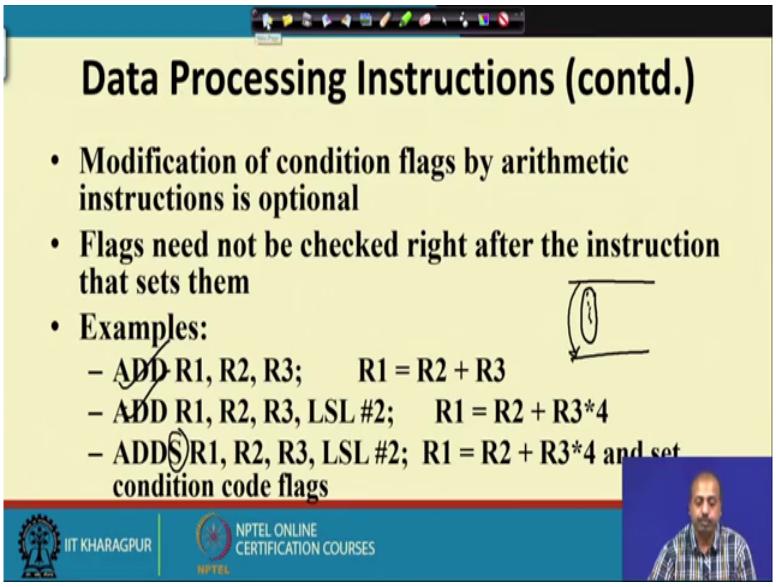


So, it is like this that, many times what happens is that for embedded application. So, we have got say the ARM processor and we are getting some signal value from outside walled. And this outside walled value, so many of the cases they are only 8 bit values, though internally I am doing some 16-bit processing, and 32-bit processing and all that. So, when we are taking values from the outside walled in terms of say this a d c analogue digital converter or values of some digital switches and all that. So, if I taking the value, so in many cases the values are only 8 bit wise

So, it is sufficient that I put, so in this in this 32 notation these 8 bits will be located in a in a in a fixed position ok. So, this 1's a rest of the positions are not meaningful when I am getting this 8 bit data from the outside wall. So, in many cases, this is sufficient, this is this is good enough ok. So, you do not need to have 32-bit constants and so this is so far it is done like this that we have got this notation; and if you badly need more number of bits like the numbers that are not fitted into this category.

So, for that purpose you have to somehow load that number into some register you can you can you have to keep that constant in a memory location from the memory location you have to load into a register and after that you do the register operation as it is. So, there is no bar in that. So, you can always do that. But if you are willing to use as immediate operand then we have to follow this convention ok.

(Refer Slide Time: 09:23)



Data Processing Instructions (contd.)

- Modification of condition flags by arithmetic instructions is optional
- Flags need not be checked right after the instruction that sets them
- Examples:
  - ADD R1, R2, R3; R1 = R2 + R3
  - ADD R1, R2, R3, LSL #2; R1 = R2 + R3*4
  - ADDS R1, R2, R3, LSL #2; R1 = R2 + R3*4 and set condition code flags

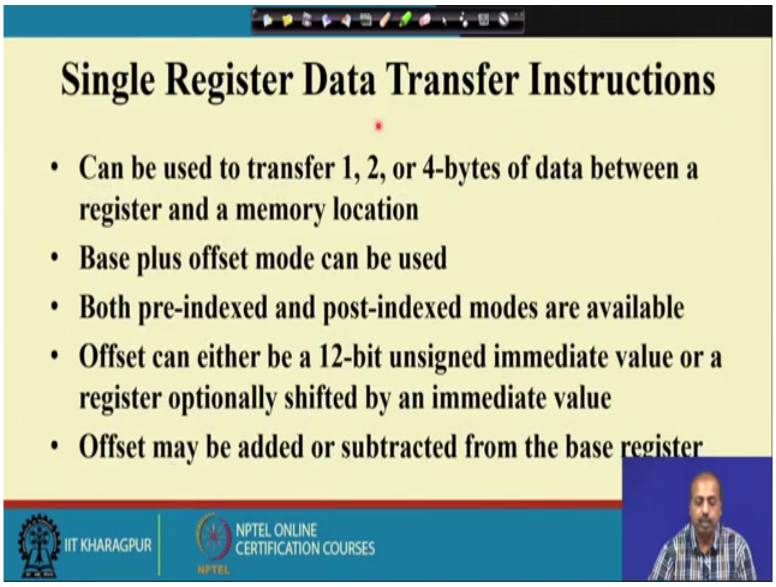
So, next we see that they the modification other thing that we have the modification of the condition flag by the arithmetic instructions that is optional. Like the ADD instruction we have got two variants ADD and ADDS; So, this ADD instruction, ADD R1 R2 R3. So, this will ADD R1 equal to R2 plus R3. So, R2 and R3 are added and this is coming to R1. So, the operand format we have is after the opcode we have got the destination then the first operand then the second operand. So, all instructions will follow this particular format.

Then the next instruction say ADD R1 R2 R3 LSL hash 2. So, this is basically the case when where the second operand happens to be a register and it is left shifted by 2 bits ok. So, this the overall execution is R1 equal to R2 plus R3 into 4. So, R3 will be left shifted by 2 bits, so it will be multiplied by 4. So, R1 gets R2 plus R3 into 4. So, these two instructions, these two add instructions they do not affect the status flags or the condition flags.

Whereas, this ADDS instruction, this will affect the condition flags like ADDS R1 R2 R3 LSL hash 2. So, this will be doing this same operation as R1 equal to R2 plus R3 into 4, but at the same time it will set the condition flags also. So, depending upon the result if the result becomes 0 then the 0 flag is set, so N Z C V. So, those flags were there. So, those flags will get affected.

So, since this is since this setting is optional. So, I have the flexibility like if some instruction sets the flag suppose this instruction sets the flag. So, I do not need to check the flag here. So, I can check this flag setting in some other instruction provided the intermediate instructions they are all of this category they are of type add type of instruction. So, they are not having this S in them, they are not affecting the condition flag. So, these are the data processing instructions.

(Refer Slide Time: 11:33)



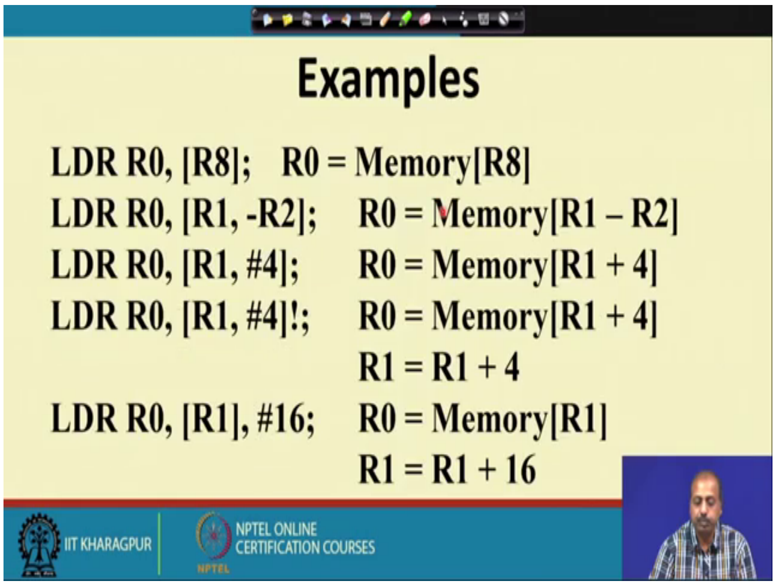
Single Register Data Transfer Instructions

- Can be used to transfer 1, 2, or 4-bytes of data between a register and a memory location
- Base plus offset mode can be used
- Both pre-indexed and post-indexed modes are available
- Offset can either be a 12-bit unsigned immediate value or a register optionally shifted by an immediate value
- Offset may be added or subtracted from the base register

Then we have got the data transfer instructions. So, the first category that we have is the data transfer instructions for transferring 1, 2 or 4 bytes of data between a register and a memory location. So, we can have 1 by data transfer, 2 by data transfer or 4 by data transfer. So, total words size is 32-bit. So, you can have this 4 byte data transfer. So, base plus offset mode can also be used.

So, you can specify a base and then offset and then that way you can specify the address pre indexed post indexed modes are also there. So, you can have we will take some example. Now, this offset can be 12 bit unsigned immediate value or a register shifted by another immediate value, I will I will take some examples.

(Refer Slide Time: 12:21)



Examples

LDR R0, [R8]; R0 = Memory[R8]
LDR R0, [R1, -R2]; R0 = Memory[R1 – R2]
LDR R0, [R1, #4]; R0 = Memory[R1 + 4]
LDR R0, [R1, #4]!; R0 = Memory[R1 + 4]
R1 = R1 + 4
LDR R0, [R1], #16; R0 = Memory[R1]
R1 = R1 + 16

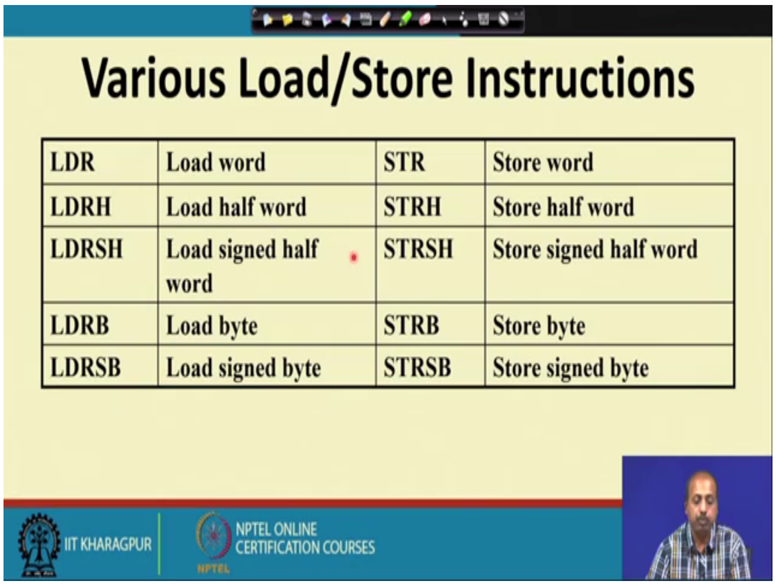
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

So, like this LDR, so LDR instruction is the load register LDR stands for load register load register R0 within square bracket R8 means this register 0 will get the content from memory location R8 ok. Then we have got this is the this is the this is the indirect addressing, then we can have this base plus offset type of addressing like LDR R0, R1 comma minus R2. So, R1 is the base address R2 minus R2 is the offset. So, this offset is added and then this is content of that memory location will come to the register R0.

Then we can have this offset as some immediate value also say R1 hash 4. So, R0 gets content of memory location R1 plus 4. Then we can have post increments, so this exclamatory mark, this identifies it as it as post increment. So, R0 gets may content of memory location R1 plus 4 and simultaneously R1 will be incremented by 4. So, this is the post increment mode ok. So, it is base offset plus post increment.

So, you can have something like this that you can have this LDR R0 within bracket R1 comma hash 16 this is another way of writing this. So, here R0 gets content of memory location R1 after that R1 is incremented by 16 ok. So, this these are the different instruction formats that are available in arm for the LDR instruction.

(Refer Slide Time: 13:54)



Various Load/Store Instructions

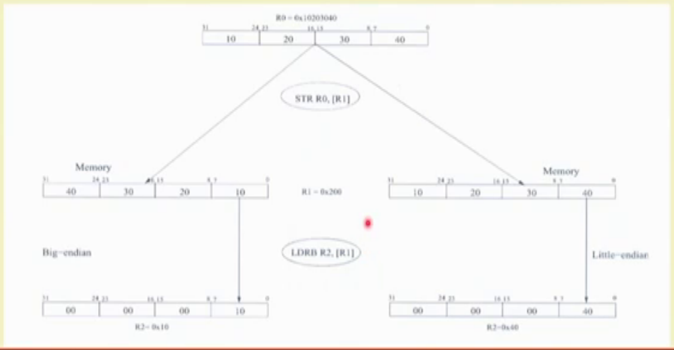
LDR	Load word	STR	Store word
LDRH	Load half word	STRH	Store half word
LDRSH	Load signed half word	STRSH	Store signed half word
LDRB	Load byte	STRB	Store byte
LDRSB	Load signed byte	STRSB	Store signed byte

Next we will look into the other instruction like STR. So, these are the variants. So, LDR is for loading and similarly STR is for storing. So, just the other way, SO you can say the STR some register to some memory location. So, content of that register will be stored in the memory location.

Then the variants like LDRH load half word then STRH store half word, then LDRSH load signed half word, then store signed half word LDRB load byte, LDR STRB store byte. So, these are the various instructions that we have in load and store variants.

(Refer Slide Time: 14:33)

Little-endian vs. Big-endian



So, this slides, this will explain the difference between this big-endian and little-endian suppose we have got this register R0 it Is a 32-bit register. suppose its content is a hexadecimal 10, 20, 30, 40. So, they are distributed the bytes are distributed like this. So, 4 0 is in bits 0 to 7 and 10 is in bits 24 to 31. So, we have got this highest order byte in highest order portion of the register and the lowest order byte in the lowest order position of the register.

Now, suppose we are executing this instruction STR are store R0 comma within bracket R1 where R1 value is 0 x 2 0 0. So, in this 0 x 2 0 0 that location, this is again a 32-bit location then this if it is big-endian format then this higher order byte will come to the lowest address then 2 0 goes to the next one. So, this is the first byte. So, it goes to this location 2 0 goes to the next location.

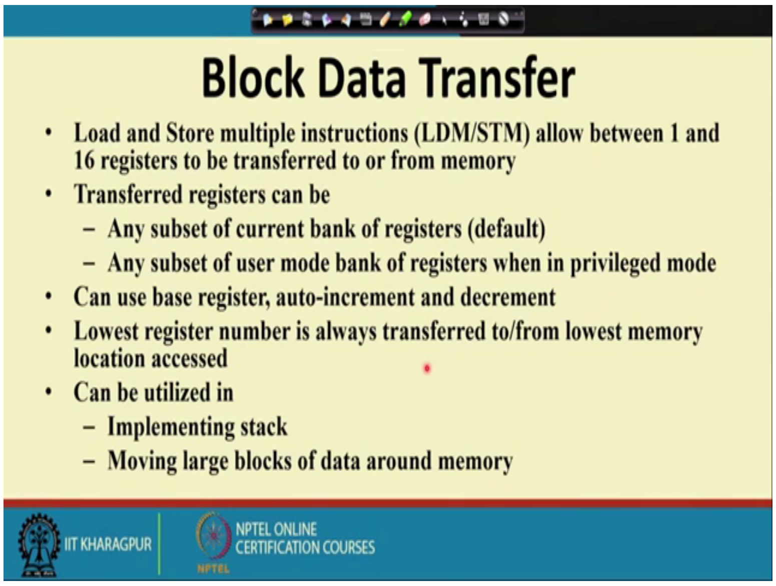
So, you see ultimately this lowest order byte. So, it has come to the highest order portion of the memory. So, if your memory is organised as 8 bit word then this is the this is the lowest address word and this is the highest address words. So, that way the lowest order byte has gone to the highest order memory location. So, this is this is the big-endian format.

On the other hand if it is little-endian format then this lowest order byte will go to the lowest order address and this highest order byte will go to the highest order address like again the same thing if it is byte organised memory then this 4 0 will go to the lowest address 3 0 to the next one, 2 0 to the next one, and 1 0 to the highest address. So, that way it will go.

So, as such there is no difference like as long as you are reading and writing as words 32-bit words. So, there is no difference between the two, but if you are doing a byte loading like say this instruction LDRB load byte then R2 register I will I will be loading the content of the memory location pointed to by R1.

So, in this case R1 is pointing to the byte containing the value 10. So, 10 will be loaded and in the little-endian format. So, this R1 will be pointing this location two hundred will have the value 40. So, the 40 will be loaded into R2. The difference will come at this position, otherwise if you are reading writing word at word level then there is no difference.

(Refer Slide Time: 17:12)



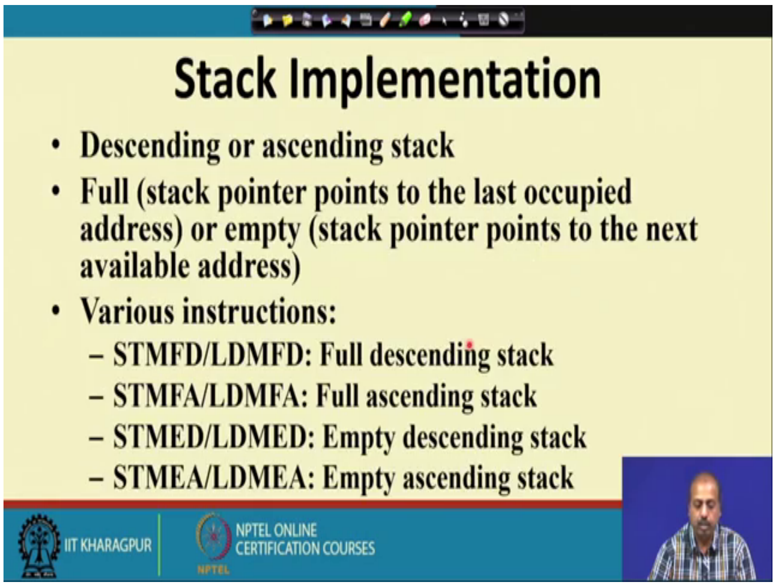
Block Data Transfer

- Load and Store multiple instructions (LDM/STM) allow between 1 and 16 registers to be transferred to or from memory
- Transferred registers can be
  - Any subset of current bank of registers (default)
  - Any subset of user mode bank of registers when in privileged mode
- Can use base register, auto-increment and decrement
- Lowest register number is always transferred to/from lowest memory location accessed
- Can be utilized in
  - Implementing stack
  - Moving large blocks of data around memory

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Next we look into this block data transfer type of instruction and they are known as load store multiple instruction LDM, STM. So, this LDM, STM instruction, they are actually the variant of that MOVs instruction that we have in 8086 type of processor as I was telling. So, you can transfer between 1 to 16 registers to or from memory. So, I will take an example then I will come back to this ok.

(Refer Slide Time: 17:41)

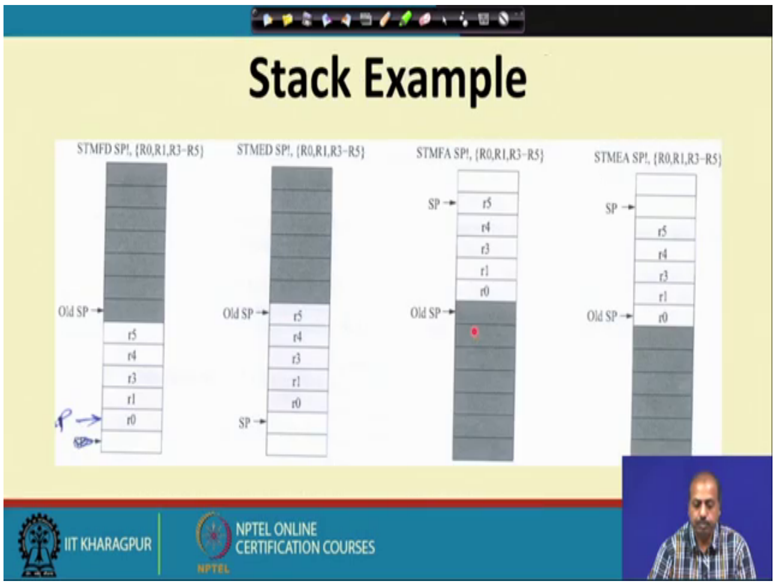


Stack Implementation

- Descending or ascending stack
- Full (stack pointer points to the last occupied address) or empty (stack pointer points to the next available address)
- Various instructions:
  - STMFD/LDMFD: Full descending stack
  - STMFA/LDMFA: Full ascending stack
  - STMED/LDMED: Empty descending stack
  - STMEA/LDMEA: Empty ascending stack

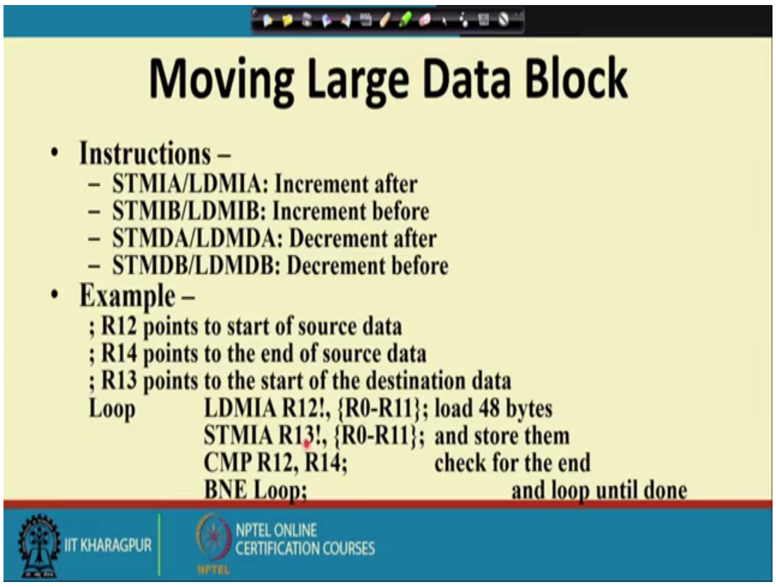
IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

(Refer Slide Time: 17:43)



Stack Example

(Refer Slide Time: 17:48)



Moving Large Data Block

- **Instructions –**
  - STMIA/LDMIA: Increment after
  - STMIB/LDMIB: Increment before
  - STMDA/LDMDA: Decrement after
  - STMDB/LDMDB: Decrement before
- **Example –**

```
; R12 points to start of source data
; R14 points to the end of source data
; R13 points to the start of the destination data
Loop       LDMIA R12!, {R0-R11}; load 48 bytes
           STMIA R13!, {R0-R11};  and store them
           CMP R12, R14;            check for the end
           BNE Loop;                       and loop until done
```

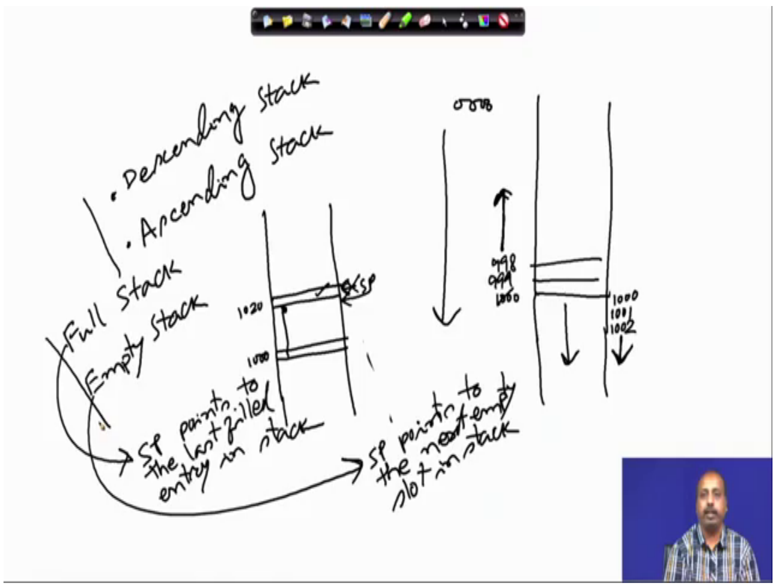
So, say this one say suppose I want to transfer say for some data. So, this instruction this LDMIA, it will transfer 48 bytes of data because R12 base pointing to the block where the source data starts. So, this they will be loaded into this R0 to R11 registers total 48 bytes, them this STMIA instruction.

So, it will be storing the values on to the location pointed to by R13. So, this way I can load store higher sized blocks on to this transfer between CPU register and the memory blocks.

So, this transfer registers they are any subset of current bank of registers or any subset user bank of registers in a privileged mode of operation you can use base register auto increment auto decrement etcetera. So, this is the thing that is it can be utilised for implementing stack and moving large blocks of data around memory. So, in arm processor we have I have told that the stack is not there by default.

So, if you are willing to implement a stack, so you can use this load multiple and store multiple type of instruction for implementing a stack. So, we will see that. So, if you as per as the stack circumstance, so there can be different variants of the stack, like say like this. So, if this is the memory on to which I am trying to implement a stack so I can say, so if this is the lowest address. So, this is the address with 0 0 0 0 and the address increases in this direction fine.

(Refer Slide Time: 19:27)



And I have got this particular address from where the stack will start and these address is say 1000, fine. Now, my stack may be when I am doing a push operation. So, it grows in this direction. So, stack pointer value after the first push the stack pointer value become 999, then after the second push it becomes 998. So, it grows in this direction fine. So, it grows towards the lower address ok.

So, it grows towards the lower address. Other possibility is that it grows towards the higher address. So, may be after my stack grows in these directions. So, initially the stack pointer was 1000, after the first push the stack pointer will become 1001, after that the stack pointer will become 1000. So, it will grow in this direction.

So, in this case I can say that it is a descending stack this stack is a descending stack because the stack pointer values are getting decremented. So, it was thousand to 999 to 998 etcetera and for the pop operation the stack pointer value will be incremented. So, this type of implementation is a descending stack implementation.

On the other hand here the stack pointer values are increasing. So, this will be called an ascending stack. So, the stack pointer values are incrementing with successive stack operation successive push operation and with the pop operation this stack pointer value will decrease. So, we have got descending stack we have got ascending stack, both are correct. So, there is no doubt like which one is correct which one is not. So, there is nothing like that. So, both are correct. So, any implementation can follow either of the conventions.

Another convention that we can have is say suppose I say that my stack starts at this point it is the location 1000, it is a location 1000. Now, at some point of time the stack has grown up to this much. So, it has gone to the location, 1020. Now, what does it mean? My stack pointer is here my stack pointer is here. So, if the stack pointer is there. So, there I can have two possibilities like. So, it may have two meaning, one meaning is that that the stack is actually full up to this point up to this much it is full and the next push operation will be filling up this location.

So, this is one possible implementation. So, I can say that wherever the stack pointer points to ease and empty location. So, that location is empty and the next push operation will put the data there. And other convention may be the stack pointer instead of pointing to this location it points to the previous location. So, in that case for doing a push operation I should first implement the stack pointer and then put the data on to the stack. So, this way I can have another classification one is called full stack full stack and another is called another version is called empty stack.

So, they do not mean that the stack is full or stack is empty like that. So, full stack means that the stack pointer points up to points to the last field entry last field entry in the stack. Whereas, this empty stack will mean that it is the stack pointer points to points to the next empty slot, it points to the next empty slot where the data can be put ok. So, overall, I have got two more variants one is full stack another is empty stack. So, if I take these two and these two into consideration. So, I can get four different types of stacks implemented, full descending stack, full ascending stack, empty descending stack and empty ascending stack. So, 4 variants can be there. So, ARM processor they do not restrict you to have any of this variants. So, you can have all the 4 variants of stack, but you can, so you can you have to use your own instructions for doing it. So, you have to use you have to implement it separately. So, that is the only requirement, but this arm processor will not stop you to that.

On the other hand if you remember 8085 or 8051 there is a specific saying like the stack pointer it points to the empty the slot and the push operation should first increment the stack pointer and then that value should be copied. That means, it is the actually having some sort of a full stack implementation and the stack pointer was always getting decremented. So, it is a full descending stack that we had there. But arm processor will allow you to have any of the variants of the stack.

So, coming back to our discussion; So, we have got a 4 different variants, descending or ascending and full stack pointer points to the last occupied address or empty stack pointer points to the next available address. So, we have got different variants of this instruction like store multiple full descending and LDMFM, load multiple full descending.

So, they are used for implementing a full descending stack similarly STMFA and LDMFA full ascending stack implementation. So, store multiple full ascending LDMFA load multiple full ascending, then STMED and LDMED empty descending stack and these STMEA and LDMEA for empty ascending stack. So, we can have a any variant. So, you should not mix up this instruction.

Like, if you are use STMFD while for storing the values on to stack. So, while loading you should not use LDMFA, ok. So, then the values that you get will be corrupted. But otherwise there is no problem. So, if you are using the pair properly then there is no problem. So, you will be getting the same values popped out from the stack.