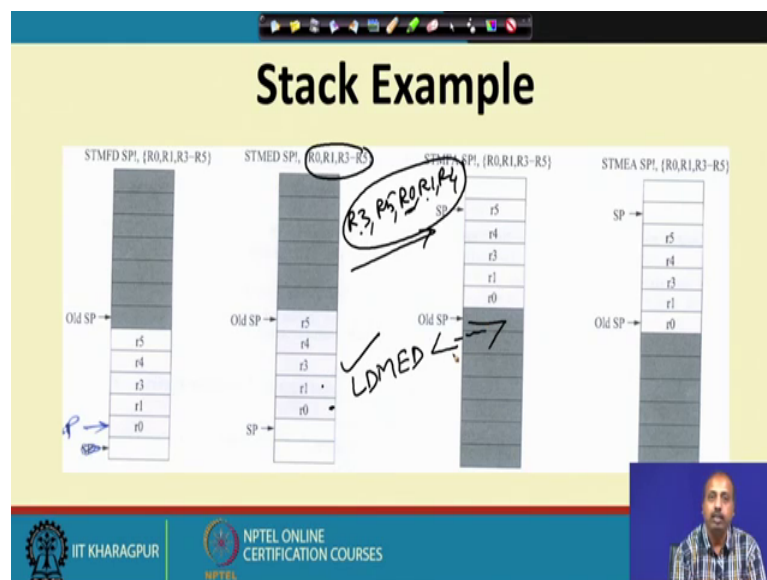


Microprocessors and Microcontrollers
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture – 46
ARM (Contd.)

So, if we look into an example of this stack, like The first example is STMFD stack pointer, then this exclamatory mark is there means it will be following that auto increment, auto decrement mode either of them.

(Refer Slide Time: 00:20)



Then here I can specify the registers that I want to say ok, so this R0, R1, R3 to R5. So, if this is the previous stack pointer, now, if it is, it is empty descending stacks. So, I assume that the memory address decreases when I go towards the lower direction. So, the our 0 is here, the 0 instead, address 0 is on this side and this side I have got the highest address.

So, the stack pointer value now, is decremented and the values are stored here. So, this is a full stack. So, it was pointing to the location till which it was having the value and now, the new before putting the next value the stack pointer is decremented and the value is saved here. So, that way it contains. So, new stack pointer will be pointing to this because the up to this much the stack is full now. So, it will be pointing to this.

Now, if we are having say the STMED then the previous stack pointer was pointing to the next empty slot. So, from that point onwards the values of the registers will be saved and this stack pointer will be pointing to the next empty slot after doing this operation. Then STMFA, this is it is it is also full ascending stack. Now, the previous stack pointer was here. So, it is ascending stack.

So, stack pointer value will be incremented. So, it will be going towards the higher addresses and then this it is full. So, this stack pointer will point to the last field entry. And this one STMEA, this stack pointer value will be decremented it is going the ascending direction. So, this is, this is this is the stack pointer value will be incremented it is going the ascending direction this side I have got highest address this side I have got lowest address. Now, these values will be saved in the corresponding locations.

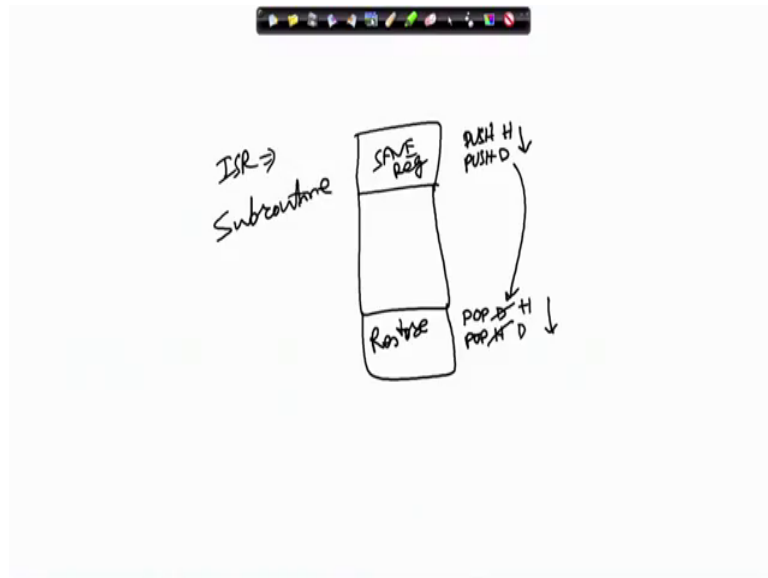
Now, you see that in all the cases if you look into the pattern in which they are saved. Now, everywhere I am writing like R0, R1, R3 to R5, but they are not saved in the same order. So, you see that the in this if you look into this particular example then this the this is the lower address and you see the R0 register has been saved at the lowest address, then the R1 then R3 R 4 and R. Similarly here, R0 register has gone to the lowest address and the R3, R 4, R 5. Whereas, in this case R0 register has gone to the lowest address and this R the R0 it has gone to the lowest address and R5 has gone to the highest address. So, in all the saving, it is following that convention that the higher order register is going to higher order memory address.

And one beautiful thing about this processor ARM processor is that it does not matter in which order you specify these registers. So, even if you write like say R3, R5, R0, R1, R 4. So, if you see if you simply write like this also then also the saving will be same as this one because it will always follow that whatever be the lowest register. So, that should go to the lowest address. Similarly whatever is this R0 is the lowest one. So, R0 will come here then R1 is the lowest R1 will come here. So, whatever be the order in which you have specified here. So, it does not matter. So, it will be it will be taking it will be say saving in that order

Similarly, whenever you are loading the value. So, if you are using this LDM instruction, so LDMED instruction then whatever be the order in which you specify this registers it does not matter the lowest order location content will be retrieved in to the lowest

register. That way the highest order content will be retrieved in to the highest register. So, this is very much useful. I will take an example and try to explain like previously in 8085 or 8051 when we discussed about one ISR structure.

(Refer Slide Time: 04:42)



So, we said that if this is the body of the ISR. So, before that you need to save the registers you need to save registers and after the body you need to restore registers or for that matter if it is in any other procedure also any other subroutine also and that you need to save registers and restore. So, we have to do it like this. And we have to be careful, like if you have written here like say PUSH, PUSH H then PUSH D then while retrieving. So, you have to first do a POP D and then a POP H.

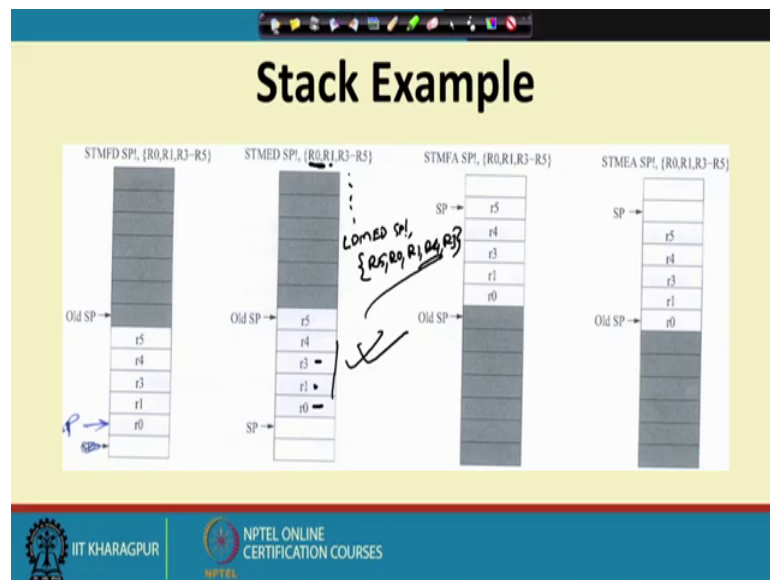
So, you have to do it in the other directions of, so you have to follow the order in which you have pushed it while popping. So, you have to POP in the other direction otherwise the contents will be mixed up. So, if you do it do by mistake if you do here POP D, POP H and POP D then the register values will be corrupted.

So, but many times what happens is that the programmers while writing their programs. So, they do this type of mistakes and it they may not be following the order in which the registers have been pushed and they forget to retrieve in the reverse order ok. So, this is this is a very common mistake that has been detected for the assembly language programmers.

So, what these ARM people have done? They have just got rid of this problem. How? It does not matter. So, as we have seen here it does not matter in which order you have specified this PUSH POP or these registers in this saving whatever be the order, it does not matter. So, it will always take R0 as the lowest register that is specified here. So, it will put it in the lowest address.

So, then it will find that R3 is the next lowest register. So, it will put R1, R 1 is the next lowest register. So, it will put R1 as the next lowest address. Then it will find it will analyze all this register and find R3. So, R3 it will be putting here, at the lower next lowest address, this way it will do. So, it is irrespective of the order in which you have specified it in this here. So, it does not matter.

(Refer Slide Time: 07:07)

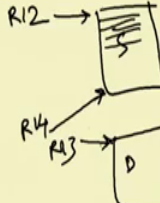


So, after doing this, so if you are after some after this is STMFD, ED sometime later if you have written like LDMED stack pointer and then in the register if you have written like R5, R0, R1, R 4, R3. So, then also it does not matter it will be retrieving these values properly onto those registers ok. So, that is the good thing about this ARM processor.

(Refer Slide Time: 07:41)

Moving Large Data Block

- **Instructions –**
 - STMIA/LDMIA: Increment after
 - STMIB/LDMIB: Increment before
 - STMDA/LMDA: Decrement after
 - STMDB/LMDB: 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



IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, next we will be looking into that large block of data transfer like that moves instruction that we have seen in case of that we know about if this moves instruction 80-86 type of processors. So, here the instruction that will be useful are STMIA and LDMIA which is increment after, then STMIB and LDMIB for decrement, for implement before then STMDA and LDMDA decrement after and this is decrement before.

So, this is one example that I have taken suppose I have got a block of data and this block of data is pointed to by R 12. So, R 12 is pointing to the block of data that we have and source data. So, this is the source and we want to put it to this destination which is pointed to by R 14. So, R 14 points to this destination and R sorry not here, R 14 a points to the end of the block. So, R 14 is this one, R 14 is pointing to the end of the block and R 13 is the beginning of the destination. So, this is R 13.

So, in this case, we have got this. So, this R0 to R 11 these 12 registers are free for this data transfer operation. So, this LDMIA instruction, what I am doing? So, R 12 estimated remark R0 to R 11. So, it will be copying the content from this locations pointed to by R2 l into the registers R0 to R 11. So, each of these registers are 4 bite white. So, total 48 bytes of data will be transferred by this instruction from memory to the R0 to R 11 registers. After that it will be using one STMIA instruction.

Now, R 13 was pointing to the destination. So, it will be transferring the 48 bytes of data from R0 to R 11 to the memory locations pointed to by R 13. And after every transfer

this value will be implemented R 13 value will be implemented because the discriminatory mark is there.

Then we compare R21 with R14 to see whether we have reached the destination, if they are not equal if the destination is not reached then it will go back to this loop and it will do the next 48 bytes of transfer. So, this way it will go on. So, but this is a very preliminary type of routine you can see that it does not check for any overlapping of the blocks, it does not check for any the data bytes that you are transferring the data block, may not be an integer multiple of 48. So, all those problems are there, but still. So, this gives us some idea about how this large block of data can be transferred.

(Refer Slide Time: 10:34)

Multiplication Instruction

- Several versions
 - Integer multiplication (32-bit result)
 - Long integer multiplication (64-bit result)
 - Multiply accumulate instruction
- Instructions
 - MUL - 32-bit multiply
 - MULA - 32-bit multiply accumulate
 - UMULL - 64-bit unsigned multiply
 - UMLAL - 64-bit unsigned multiply accumulate
 - SMULL - 64-bit signed multiply
 - SMLAL - 64-bit signed multiply accumulate
- Example
 - MUL R0, R1, R2; R0 = R1 * R2
 - MULA R0, R1, R2, R3; R0 = R1 * R2 + R3

$y = \sum a_i x_i$

The slide includes logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES. A small video inset shows a man speaking.

So, next we look into the multiplication instruction. So, multiplication instruction in ARM processor that is a very very powerful you have seen that 8085 does not have any multiplication operation. In case of 8051 we have got multiplication, but it is 8 bit multiplication only and the instruction format is fixed. So, a and b registers they should have the two operands and this MUL a b instruction. So, it was just multiplying the numbers a and b numbers in a and b.

Now, in ARM processor this multiplication has got many variants like we have got integer multiplication, which is 32-bit which will give you a 32-bit result, then there is a long integer multiplication that gives 64-bit result and there is a multiply accumulate instruction, for this multiplication addition, multiplication over arrays and all that. So,

the instructions that we have is MUL for 32-bit multiplication MULA for 32-bit multiply accumulate and then this a UMULL. So, it is 64-bit unsigned multiply then UMULL, the this second L is for long and this U is for unsigned and so this is the multiply accumulate. So, typical examples are like this. So, multiply MUL R0, R1, R2. So, R0 gets R1 into R2, MULA R0 R1 R2 R3. So, this is an exception. So, is a 4 operand instruction.

Whereas, in ARM in most of the instruction it is a 3 operand instruction this is a 4 operand instruction where R0 will get R1 into R2 plus R3. So, this is good because this MULA instruction. So, this can be used for doing that operations like if I have if I have got this y equal to $\sum a_i \times x_i$. So, what I can do I can allocate this y to say R3 and then this a_i to these R1 registers and x_i to R2 registers and then I can do this MULA and then after that we move this R0 register value to R3 register.

So, that way I can do this multiply accumulator I can utilize this multiply accumulate to for doing this multiplication and addition simultaneously.

(Refer Slide Time: 13:00)

Multiplication Instruction (Contd.)

- Destination and source cannot be the same register
- PC (R15) cannot be used for multiplication
- Uses Booth's algorithm
- For each pair of bits, it takes 1 cycle
- One more cycle needed to start the instruction
- Multiplication continues till source register has some 1's left. Otherwise it early-terminates
- To multiply 18 by -1, if 18 is in source, it takes 4 cycles, whereas, if -1 is the source, it needs 17 cycles

The slide includes a diagram of a register with bits and a small video inset of a speaker in the bottom right corner. Logos for IIT Kharagpur and NPTEL are at the bottom.

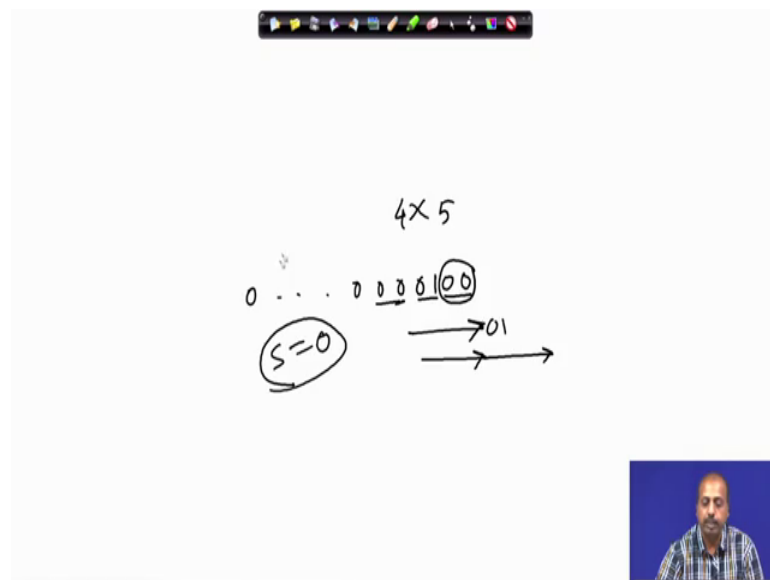
So, the next instruction that we will have is this some restriction on the multiplication instruction like destination and source they cannot be the same register. So, this is something very interesting because, may be that this destination register. So, the source register will get modified during the operation. So, as a result if you are using it as destination also then the register may content will get corrupted.

So, we will not get the correct value. So, this destination and source cannot be same register then this are 15 register the program counter it cannot be used for multiplication purpose ok, the R say the program counter cannot be used and it uses Booth's algorithms. So, Booth's algorithm is a well known multiplication algorithm. So, you can get it into computer architecture books. So, this is a multiplication algorithm. So, that is quite efficient.

And here for each pair of bits it takes one cycle. So, and one more cycle is needed to start the instruction. So, it is like this that if I have got say one 32-bit number, if I have got say one 32-bit number two 32-bit numbers to be multiplied then for each pair of bits. So, it will take one cycle. So, it is, this is these are the pairs of bits. So, we will have 16 such pairs there will 16 such pairs.

So, that will take 16 cycles plus one cycle will be needed for starting the instruction so that way this multiplication will take 17 cycles. But there is another interesting thing which is known as early termination, so this is the early termination. So, what is it? It is like this that whenever the content source register content becomes 0.

(Refer Slide Time: 15:02)



Like say if we are say multiplying, if you are multiplying two numbers and if you see that the numbers are say 4 multiplied by say 5 something like that. So, 4 is in the source register. So, it is 1 0 0 and these bits are all 0. So, when I do grouping, I do a grouping of these two bits, these two bits then after that I find that, after these two bits have been

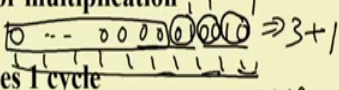
processed. So, this whole thing is shifted and I see that the register is the source register that contained 4 initially is no more is not yet is 0 ok. So, it is still having the value one. So, next time, this will be shifted next time. So, the once this shifting is done. So, 0 1 comes to the least significant two positions.

Now, I will be. So, I will do that multiplication with this 0 1 of that 5 and then it will be shifted further. So, this 0 1 will go out of the register and the source register content will become 0 and when the source register content becomes 0 there is no point trying to do multiplication with 5 and generate this partial multi, partial sums to be added finally, ok. So, that way the algorithm will terminate. So, it will not continue once that source register content becomes 0 and so that leads to the condition called early terminate and that helps in speeding of the multiplication operation.

So, this is a typical example say suppose we want to multiply 18 by minus 1. So, if we take 18 as the source then it will take 4 cycles because 18 is so this is say, this is the number 18. So, this is the first 4 is the next 4. So, it is like that. So, rest of the bits are all 0.

(Refer Slide Time: 16:39)

Multiplication Instruction (Contd.)

- Destination and source cannot be the same register
- PC (R15) cannot be used for multiplication
- Uses Booth's algorithm 
- For each pair of bits, it takes 1 cycle
- One more cycle needed to start the instruction
- Multiplication continues till source register has some 1's left. Otherwise it *early-terminates*
- To multiply 18 by -1, if 18 is in source, it takes 4 cycles, whereas, if -1 is the source, it needs 17 cycles

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

And I am trying to multiply it by minus 1. So, for minus one all these bits are one all the bits are one. Now, if you are using 18 as the source register then first for these two bits. So, it will take one cycle then for this one it will take one cycle and for these two bits it will take one cycle. After that the source register will become 0. So, this takes 3 cycles

plus 1 cycle to start the instruction. So, total 4 cycles will be needed if 18 is kept as the source.

On the other hand if you take minus 1 at the source that is you try to do this multiplication minus 1 multiplied by 18 then for this minus 1 all the bits are one. So, it will take total 17 cycles as we had calculated previously for every pair it will take one cycle the 16 such pairs plus one cycle to start the instruction total 17 cycles will be needed for doing that multiplication operation if minus 1 is taken as the source.

So, here the compiler has got a role to play. So, whenever it will find that it is multiplying by some constant. So, it can try to see how many how many cycles will be needed if that constant is taken as the first operand, and if that number is profitable then it will be putting that constant as the first operand otherwise it will put the variable as the first of all or if it is multiplying two constants. So, it can do that way. So, it can it can put one of them as the first operand whichever will require less number of cycles.

(Refer Slide Time: 18:36)

Software Interrupt

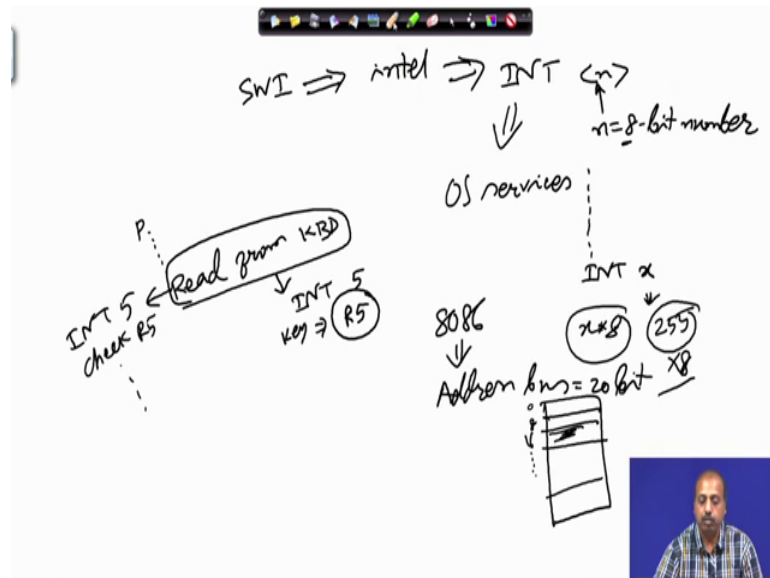
- Forces CPU to supervisor mode
- Instruction format: SWI # n
- Causes an exception trap to the SWI hardware vector, exception handler is called
- Exception handler analyzes the value of n to determine the action
- Processor completely ignores n
- Used to implement *system calls*
- Value of n is 24-bit, allowing 2^{24} different system calls

(Handwritten note: 26 bits / 2 = 764 MP)

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Next we will look into the software interrupts instructions. So, it forces the CPU to the supervisor mode. Now, in case of software interrupts the instruction format is SWI hash n and it causes an exception trap to the SWI hardware vector. So, we will see what is this vector address ok. So, before going into this, we need to look into this software interrupt concept a bit more ok.

(Refer Slide Time: 19:13)



So, as I said that this software interrupts, so they are generally available as they are the software interrupts in case of say Intel family of processors. So, the corresponding instruction is INT n. And the format is here is INT n and the number in where this n is a 8 bit number, is a 8 bit number ok. So, in a program, in a program, if you are writing a program somewhere you can put like. So, INT x where x is an 8 bit number and then this x will be multiplied by 8.

So, this x will be multiplied by some 8 and accordingly it will be jumping to one particular address where this service routine will be there. So, that is the interrupt service routine. So, the for the software like this INT instructions. So, they are useful for waste services as I said that it is normally useful for operating system services. So, in a program whenever you are willing to get service from the system, so you have to use this INT x instruction.

So, typically for example, suppose I am I have a program P and after executing for some time it needs to read from keyboard read from the keyboard device. Now, how will it read. So, the OS designer they have they will tell you that for reading for this keyboard it is available as INT say 5, ok. So, if you put INT 5. So, it will invoke that read from keyboard service and at the end of this the keyboard the value that the user has pressed that key value will be available in some particular register say R5, R5 the R5 register will have the key.

So, what you can do? In your program, when you when you have got to read from keyboard, you can translate it into one INT 5 instruction and then on getting when you return from this interrupt you know that the R5 register will have the key value. So, you can check the R5 register to see which key has been pressed by the user. So, that way you can have it. So, that whenever you need any OS services normally the operand operating system designer. So, they have will provide you a list of such services available from the operating system and they will be specified as this INT in this address will be specified there and you can you can use it in your program.

Now, in case of ARM processor, we have the similar thing and these services. So, as I said that n is an 8 bit number and this in case of processor in case of say 8086 type of processor then the address bus. So, address bus is twenty bit and then this 8 bit number. So, it can point to some fixed memory locations. So, it will it is it is not arbitrarily distributed.

So, it is multiplied by 8 and there those locations will be this. So, INT 0 it will be coming to the location 0, INT 1 will come to the location 8, INT INT 2 will come to the location 16. So, that way it will go. So, there are fixed memory addresses where this location. So, these are actually the vectored interrupts, so whatever be the value, it will be coming to that location and it will be taking it from there.

So, the maximum value that I can have with this x is 255 because it is 8 bit number. So, it is 255. So, 255 multiplied by 8, it can you can have up to that much address dedicated for this in vector table. Now, in case of ARM processor what has happened is that we have increased this range ok. So, this range is increased. So, this SWI n, this n can be any of the 24 bits. So, n is a 24 bit value. So, instead of that 8 bit value; now, n is a 24 bit value. So, if it is a 24 bit value.

So, we can have 2^{24} different OS services each of this service is called a system call. So, we have got 2^{24} different services. So, that is good because I now, I can implement large number of system calls in myself in my operating system, but that difficulty is that for each of these 2^{24} system calls we have to have the corresponding memory address if the address bus is assumed to be even say 32-bit address bus. So, in that case also, so the 32-bit address has to be specified. So, total I will

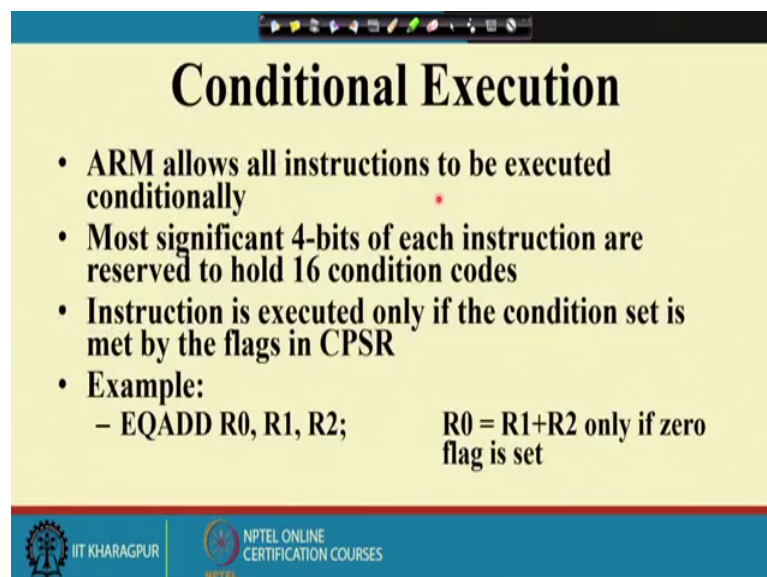
require. So, for each of these I will require 4 bytes of memory space to hold the corresponding vector address.

So, to power 24 multiplied by 4. So, 2 power 26 memory locations 2 power 26 bytes of memory location will be needed to just hold the vector table just to hold the vector table. So, 2 power 26 memory locations will be needed and this is a huge number ok. So, this is your this is a 64 MB of space so that will be required just to hold the just to hold that memory address. So, that is not that interrupt vector address. So, that is not advisable.

So, what this ARM people have done? They have done they do it like this that, they will they will not go to different addresses whatever be the value of n that you have specified here it will not go to different address it will go to the same address for all these 2 power 24 determinants system calls the hardware will come to the same address and in your software you need to differentiate between those values of n and from there you call the routine.

So, the processor gets interrupt and it comes to the same address from that point it is the interrupt service routine the it is responsibility is to analyze the value of n and accordingly decide to which call which procedure should be called. So, based on the service request the value of n is used to differentiate between the service procedure to be called. So, processor will completely ignore the value of n because it cannot handle such a big thing. So, that is about the software interrupt.

(Refer Slide Time: 26:15)



Conditional Execution

- ARM allows all instructions to be executed conditionally
- Most significant 4-bits of each instruction are reserved to hold 16 condition codes
- Instruction is executed only if the condition set is met by the flags in CPSR
- Example:
 - EQADD R0, R1, R2; R0 = R1+R2 only if zero flag is set

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Conditional execution, so I have already introduced it. So, that ARM it will allow instructions to be executed conditionally, most significant 4 bits they can used of each instructions they are deserved as condition code. So, we have got this say R0 equal to R1 plus R2 only if the 0 flag is set. So, the EQADD, so EQADD this EQ part, this will tell that addition will be done only if the EQ the 0 flag was set. So, this way this conditional execution it is helpful. Like if we have got a number of instructions and if you have got a number of condition checks and conditional execution.

So, you can use it and that way we have seen that it can be utilized to reduce unnecessary jumping in the program. So, that this conditional execution leads to straight line type of code which helps in the pipeline execution of programs.