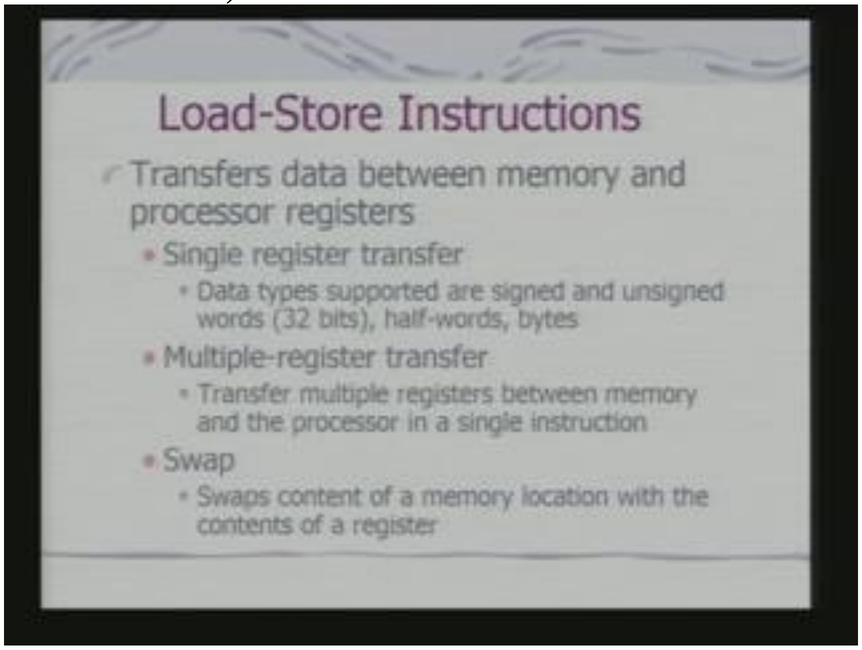
Embedded Systems

Dr.Santanu Chaudhury
Department of Electrical Engineering
IIT Delhi

Lecture 6
More ARM Instructions

In the last class we have discussed some of the ARM instructions. We shall continue with this instruction set of ARM processors and look at the different modes in which ARM Architecture can be enhanced and corresponding to that how the instructions set changes. We shall first look at load store instructions.

(Refer Slide Time 01:48 min)

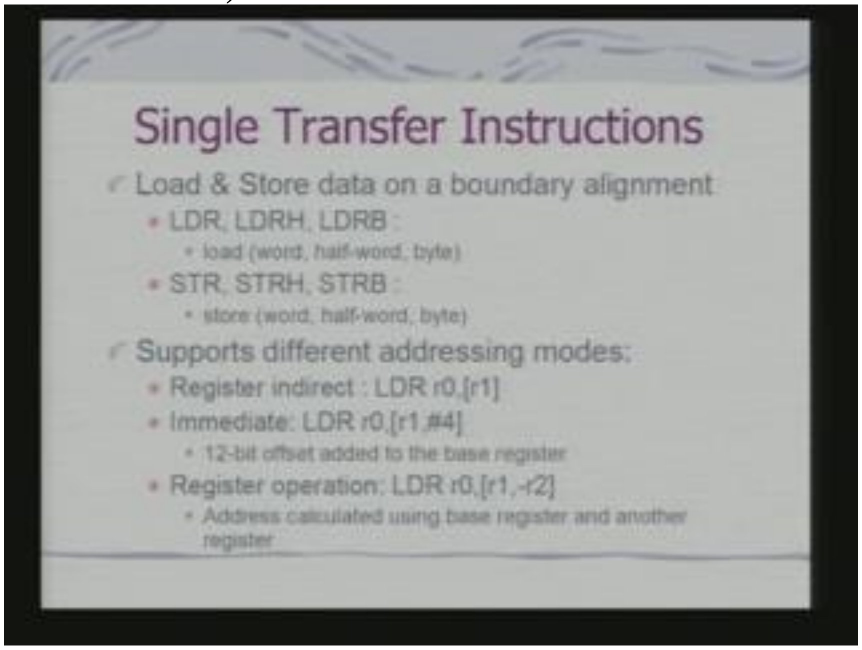


Load-Store Instructions

- Transfers data between memory and processor registers
  - Single register transfer
    - Data types supported are signed and unsigned words (32 bits), half-words, bytes
  - Multiple-register transfer
    - Transfer multiple registers between memory and the processor in a single instruction
  - Swap
    - Swaps content of a memory location with the contents of a register

Now, ARM is an example of RISC architecture. So, basically memory access is through load and store. And this load store instructions and therefore for data transfer between memory and processor registers. There are 3 basic types of load store instructions single register transfer, multiple register transfer and swap. In fact multiple register transfer in case of ARM is a significant departure from classical RISC model or RISC instructions sets. The single register transfer supports signed and unsigned 32 bit transfer half-words transfer as well as byte transfer. Let us look at single transfer instructions.

(Refer Slide Time 02:53 min)

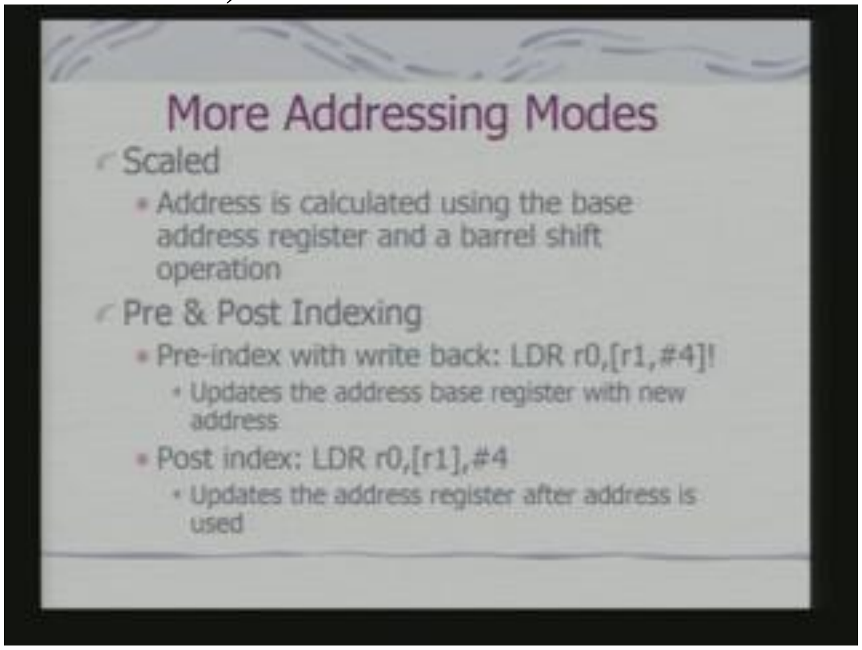


Single Transfer Instructions

- Load & Store data on a boundary alignment
  - LDR, LDRH, LDRB :
    - load (word, half-word, byte)
  - STR, STRH, STRB :
    - store (word, half-word, byte)
- Supports different addressing modes:
  - Register indirect : LDR r0,[r1]
  - Immediate: LDR r0,[r1,#4]
    - 12-bit offset added to the base register
  - Register operation: LDR r0,[r1,-r2]
    - Address calculated using base register and another register

These load and stored instructions are particularly for transferring data at a boundary alignment. What do we mean by boundary alignment? It means that the data is expected to be aligned at the correct memory address. So, when we load or store a word, the address should be at the 32 bit boundary; when we load or store or half-word it should be at the 16 bit boundary. In fact these load store instructions support a variety of addressing modes. The simplest addressing mode is register indirect. Here the memory address is specified in a register. Variation of this will be found in this case where I have specified an immediate mode offset. This offset is added to the base register to get the memory address.
We can even have register operation specified in this case I have specified a second register and I have given a minus sign here. It means that there would be an arithmetic operation performed between the content of r1 and r2 for obtaining the memory address. This is true for store instructions as well. There are more addressing modes one of them is scaled addressing mode. Now, I hope you remember that there is a barrel shifter in the data path of ARM. In scaled addressing mode the barrel shifter is used for calculation of address. So, address is calculated using the base address register and a barrel shift operation. The same set of operations that we had discussed in the context of data processing instructions are applicable here also. Then we have pre and post indexing this is also a very interesting scheme of addressing memory locations.

(Refer Slide Time 04:43 min)

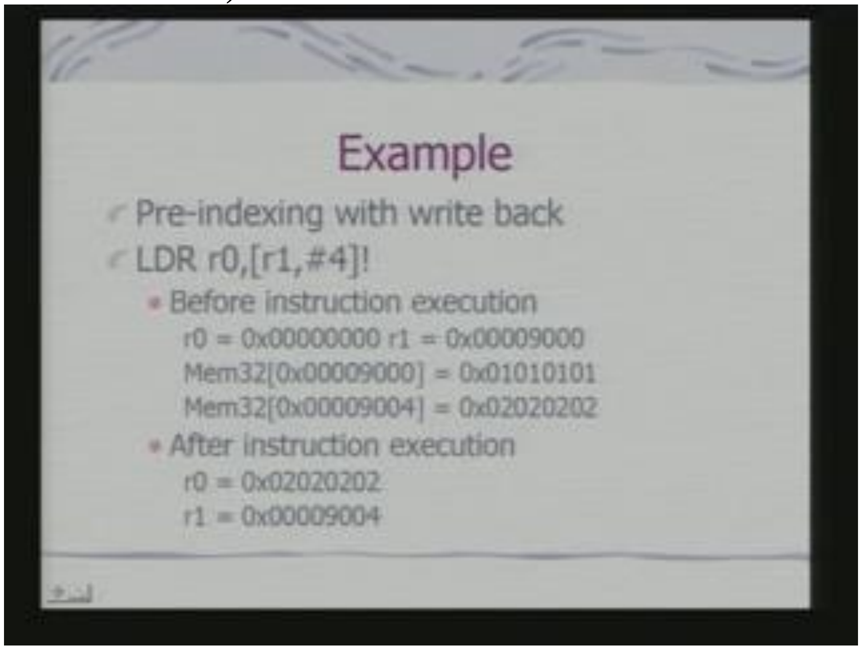


More Addressing Modes

- Scaled
  - Address is calculated using the base address register and a barrel shift operation
- Pre & Post Indexing
  - Pre-index with write back: LDR r0,[r1,#4]!
    - Updates the address base register with new address
  - Post index: LDR r0,[r1],#4
    - Updates the address register after address is used

The two distinct modes are pre indexed with write pack and post indexed. The normal mode that we have seen so far is typical pre index addressing mode. When we have pre index with write back the interesting is that when we calculate this address of the memory location what we have done. We have added the offset immediate mode offset with the content of r1 and then what we do,
We update the address base register with the new address. Now, why it is a pre index because when we fetch the data we use the address of r1 plus 4, okay r1 plus four in this case four is an immediate value and then this r1 plus 4 value is loaded on to r1. In contrast to this we can look at post index. In the post index you will find that syntax of the instruction is slightly different. Here, what happens is that we update the address register after address it used. What does that mean that means we use r1 for referring to the memory and after we have accessed the memory. We modify r1 and load the modified value of r1 back to r1. So, this is post indexing. So, the pre index means that this offset is added to the base for accessing memory. In post index the base value is used for accessing memory and then base value is added with the offset and this new value is stored back to r1. See, if I again use this instruction in a loop then in that case the modified r1 value would be used for accessing next memory location. Let us take an example to understand it better. Here I am illustrating a case of pre indexing with write back. So, we have got r0 as the target register and if you look here what we are telling is that we have got this memory location. Let us say this memory location is has got some value and then this memory location has got some other value. Now this register r1 which is my base is initially loaded with 9000, okay. Now, after I execute the instruction what happens? My r0 is loaded with content of this memory location because I have added 4 to r1 to access the memory location.

(Refer Slide Time 07:52 min)

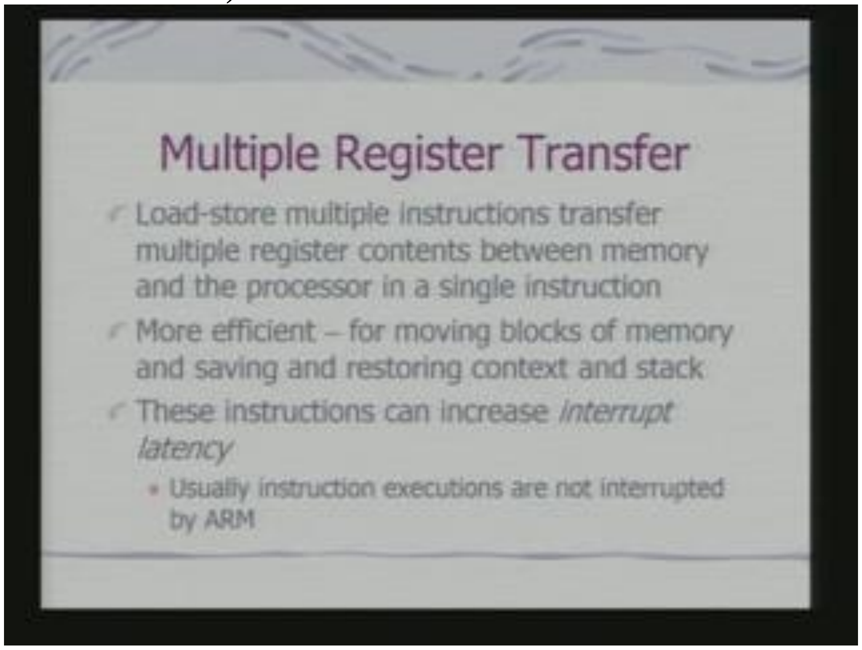


Example

- Pre-indexing with write back
- LDR r0,[r1,#4]!
  - Before instruction execution
    r0 = 0x00000000 r1 = 0x00009000
    Mem32[0x00009000] = 0x01010101
    Mem32[0x00009004] = 0x02020202
  - After instruction execution
    r0 = 0x02020202
    r1 = 0x00009004

So, my r0 is now loaded with this memory location and then r1 content is also changed, r1 is content is change to 9004. This is the basic operation that means what you can find out that incase of pre indexing the content of two registers in this case. Since I am using a load instruction gets modified the base register as well as that of the target register. Next, we have got multiple register transfers. In this case we transfer contents of multiple registers to the memory or content of multiple memory locations to multiple registers using a single instruction. Obviously this provides and efficient way of moving blocks of data between memory and the set of registers.

(Refer Slide Time 09:26 min)

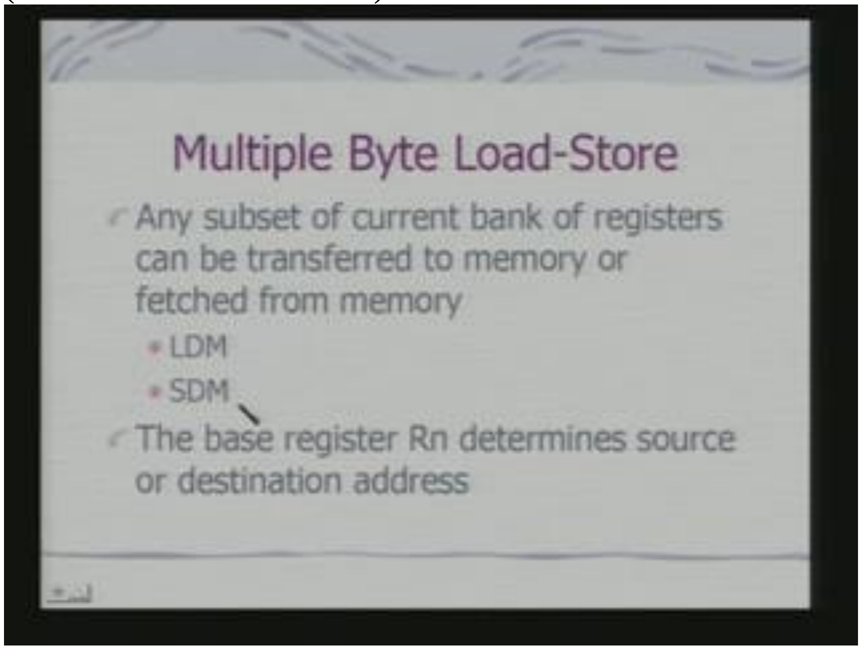


Multiple Register Transfer

- Load-store multiple instructions transfer multiple register contents between memory and the processor in a single instruction
- More efficient – for moving blocks of memory and saving and restoring context and stack
- These instructions can increase *interrupt latency*
  - Usually instruction executions are not interrupted by ARM

And since I am using this multiple byte transfer or word transfer in a single instruction, what happens is that this instruction cannot be interrupted under normal circumstances. So, this instruction may increase interrupt latency. That means if there is an interrupt pending that interrupt can only be serviced if and only if I have completed transfer of all this, all this transfers completed all this transfers from memory or register bank. So, how, what is the mnemonic used for this multiple byte load store. In this case I use LDM or SDM.

(Refer Slide Time 11:01 min)

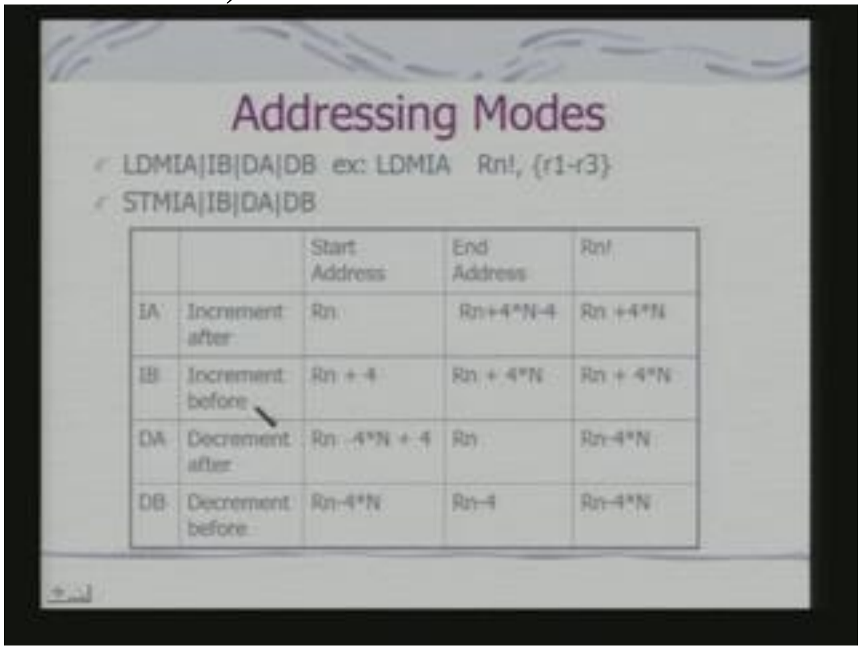


Multiple Byte Load-Store

- Any subset of current bank of registers can be transferred to memory or fetched from memory
  - LDM
  - SDM
- The base register Rn determines source or destination address

And the base register Rn determines the source or destination address depending on the instruction whether it is a load instruction or whether it is a store instruction. Now, there are also number of addressing modes which are supported here .So, I have listed this addressing modes. We can have increment after, we can have before, we can have decrement after, we can have decrement before. Now, what happens in this case? If you look it, look at the other columns we have illustrated these operations.

(Refer Slide Time 11:32 min)



Addressing Modes

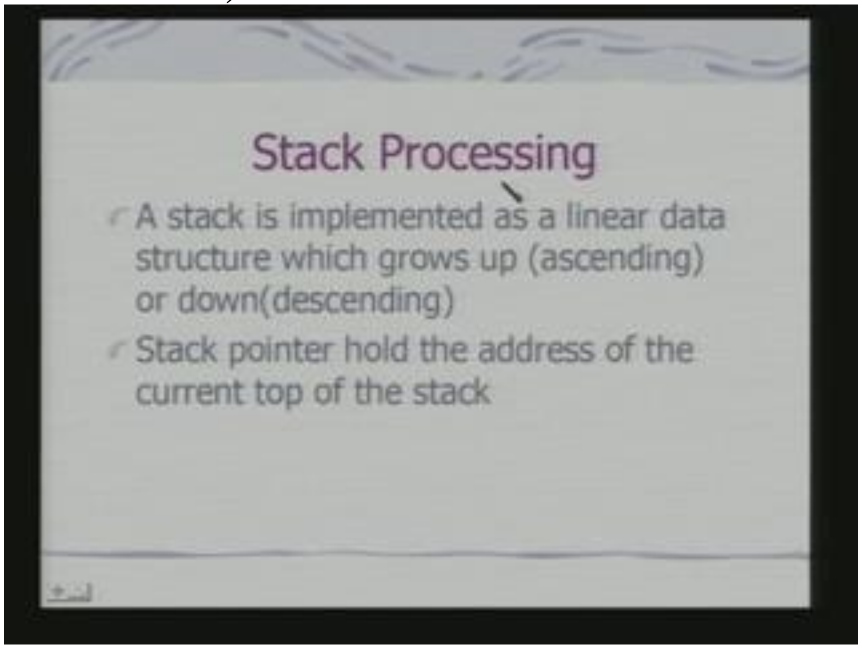
- LDMIA|IB|DA|DB ex: LDMIA Rn!, {r1-r3}
- STMIA|IB|DA|DB

		Start Address	End Address	Rn!
IA	Increment after	Rn	Rn+4*N-4	Rn +4*N
IB	Increment before	Rn + 4	Rn + 4*N	Rn + 4*N
DA	Decrement after	Rn -4*N + 4	Rn	Rn-4*N
DB	Decrement before	Rn-4*N	Rn-4	Rn-4*N

The start address when I am using addressing mode IA; so what I shall have, I shall have the one instruction LDMIA Rn and I have got the set of registers. This Rn is the base address is stored in this Rn register. And what I am trying to transfer? I am trying to transfer the content of registers r1 r2 r3. So, the start address is Rn, the end address is this. Why I am having this value because I am transferring what- the content of this registers to memory. Since I am transferring content of registers to the memory, so what I shall have? This address has to be incremented and they have to be incremented by 4. If there are N N registers content is to be transferred. Then it has to be multiplied by 4 because ARM has got byte addressing because each byte has got a unique address and I am subtracting minus 4 because it is a increment after.

So, final value of Rn is this. Similarly when I have got an increment before, the start address is Rn the last 4. So, base address is added with the offset 4 and then what we get the first register value is transferred. Here the end address would become Rn plus 4 into N, okay and my Rn here also that is the final value of Rn will be Rn plus 4 into N. Similarly, I can have decrement after and the decrement before modes. In this case instead of increment I have got decrement as the basic operation. So, what we have therefore seen is that using a single instruction by specifying the base register and by specifying in this case the set of registers here I can load; here what is happening. This is a load instruction, so what I shall be doing, this memory location pointed to by the base will be loaded on to this registers. If instead of that if I got store I shall be storing the content of the registers on to the corresponding memory locations. And the addresses of the memory locations will be given by the content of Rn. Now, this multiple register instructions in various ways facilitate stack processing.

(Refer Slide Time 14:57 min)

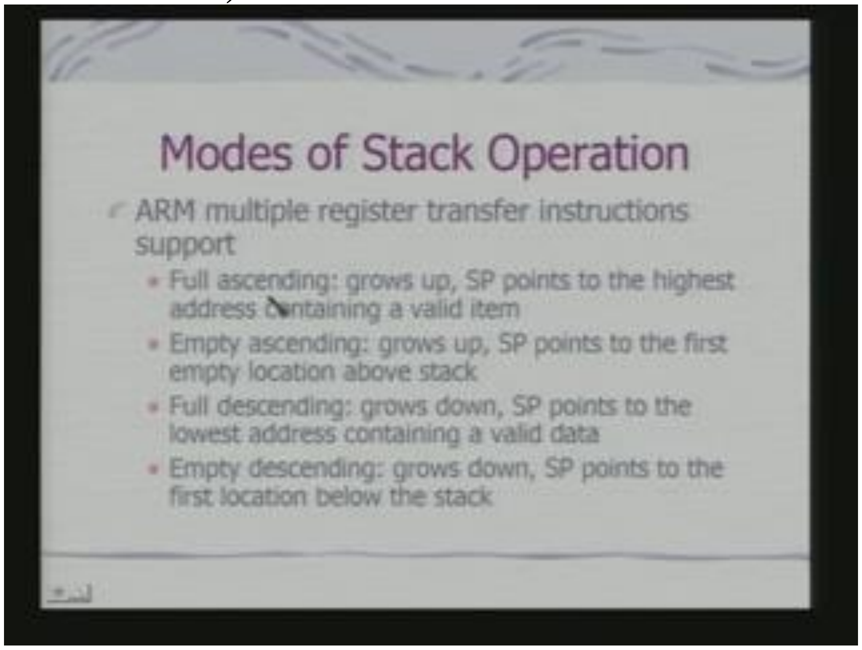


Stack Processing

- A stack is implemented as a linear data structure which grows up (ascending) or down(descending)
- Stack pointer hold the address of the current top of the stack

Now, what is the stack? Stack is implemented as a linear data structure which grows up or down. So, I can have growing up stack or I can have a growing down or a descending stack and stack pointer register hold the address of the current top of the stack. Now, how shall I use these different addressing modes that I have discussed in the contest of stack? Do you really need special push and pop instructions? Strictly speaking I do not need why because I have got a symmetric organization with any of the register as space registers I can use these addressing modes. If I am using these addressing modes then using my stack pointer register I can implement either ascending or descending stack.
Hence we shall have in ARM all this possible modes of stack operations. We have full ascending, empty ascending, full descending, empty descending. In case of full descending, the stack grows up but the stack pointer points to the highest address containing a valid item.

(Refer Slide Time 16:33 min)



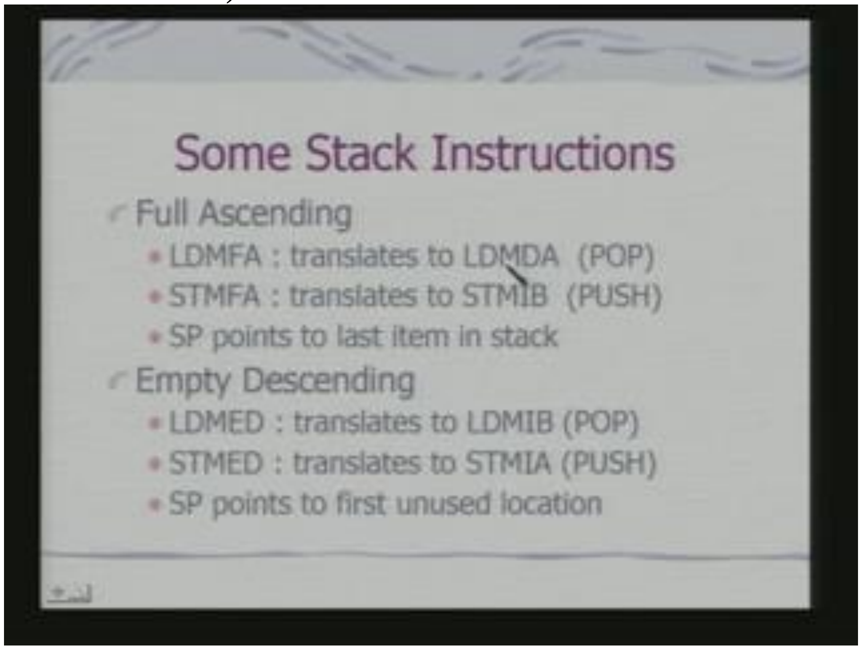
Modes of Stack Operation

- ARM multiple register transfer instructions support
  - Full ascending: grows up, SP points to the highest address containing a valid item
  - Empty ascending: grows up, SP points to the first empty location above stack
  - Full descending: grows down, SP points to the lowest address containing a valid data
  - Empty descending: grows down, SP points to the first location below the stack

In case of empty ascending the stack grows up but SP points to the first empty location above stack. So, that is why it is empty ascending .Similarly I can have a full descending whether the stack grows down and SP points to the lowest address containing a valid data. In case of empty descending the stack grows down and SP points to the first location below the stack. Now, you can realize that if I used stack pointer as my base registers with addressing modes for multiple byte transfer that I have discussed that is increment after increment before decrement after and decrement before. If I use these addressing modes, using these addressing modes I can implement stack in any of these modes that I have listed, okay. And this is a flexibility that this ARM architecture provides.

If I compare this with a typical processor like your 8085 or 8086 depending on the operations defined for push or pop instructions, the nature or mode of the stack gets defined. And that is defined by the architecture itself, but here the mode of the stack operation is programmer defined. So, we can now talk about stack instructions which can implement full ascending or empty descending.

(Refer Slide Time 18:53 min)

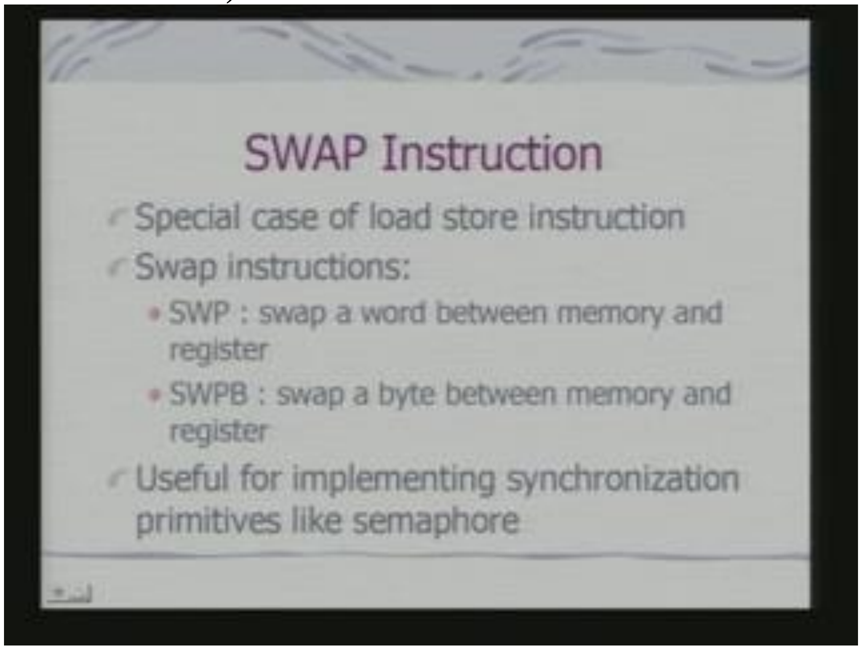


Some Stack Instructions

- Full Ascending
  - LDMFA : translates to LDMDA (POP)
  - STMFA : translates to STMIB (PUSH)
  - SP points to last item in stack
- Empty Descending
  - LDMED : translates to LDMIB (POP)
  - STMED : translates to STMIA (PUSH)
  - SP points to first unused location

Similarly I can have the other counter parts as well. So, we talk about instructions LDMFA. LDMFA actually translates to this LDMDA and STNFA translates to STMIB and SP points to the last item in the stack. When I am using empty descending this translates to IB that is increment before and STMIA is increment after, okay and SP used to the first unused location in the stack, okay. Now, in this case the interesting feature is if you look into it, that these are not real instructions. In many case the ARM assembler provides this instructions, okay and these instructions translates to one of these instruction modes. Next, we have got swap instruction. In case of swap instruction what happens- a word is swapped between memory and register.

(Refer Slide Time 20:12 min)

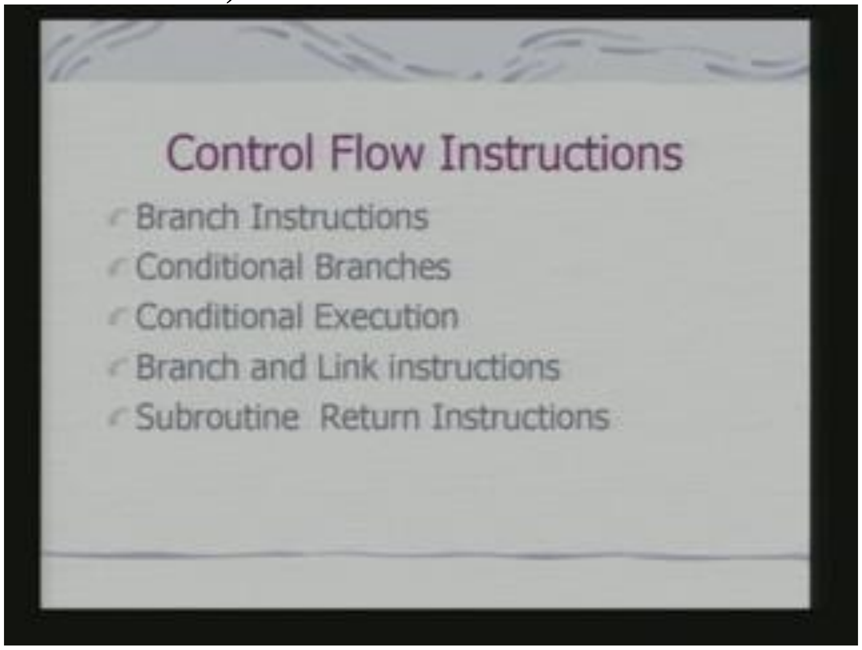


SWAP Instruction

- Special case of load store instruction
- Swap instructions:
  - SWP : swap a word between memory and register
  - SWPB : swap a byte between memory and register
- Useful for implementing synchronization primitives like semaphore

In case of SWPB, you swap a byte between memory and register and this is useful for implementing synchronization primitives like semaphore. If you have already done a course on OS, you know what a semaphore is and we have also discussed in the context of PIC that is if you want to prevent access to a common memory location by concurrent threads, that we want to restrict the access to a single thread when a memory location is shared between two concurrent threads, we would like to use semaphores. To implement such operations swap is a hardware supports; swap instruction is a hardware support. Next, we shall look at control flow instructions.

(Refer Slide Time 21:15 min)

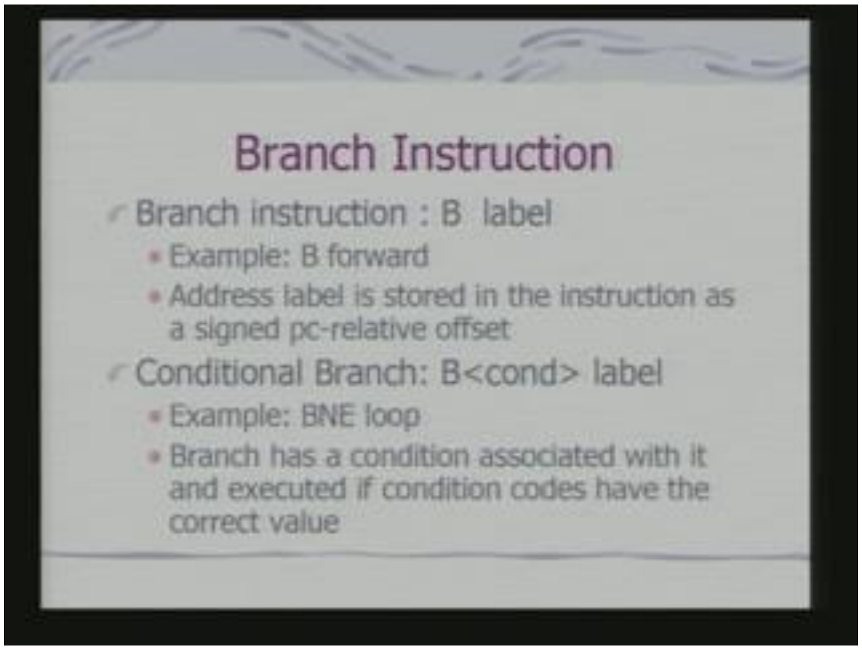


Control Flow Instructions

- Branch Instructions
- Conditional Branches
- Conditional Execution
- Branch and Link instructions
- Subroutine Return Instructions

You have got branch instructions, conditional branches, conditional executions, in fact this is the very interesting feature of ARM. ARM enables execution of each instruction conditionally. Then you have got branch and link instructions as well as subroutine return instructions. And these instructions are all used for controlling your program flow. Typically branch instruction has got 2 variance; one is branch which is actually an unconditional branch or jump.

(Refer Slide Time 21:46 min)

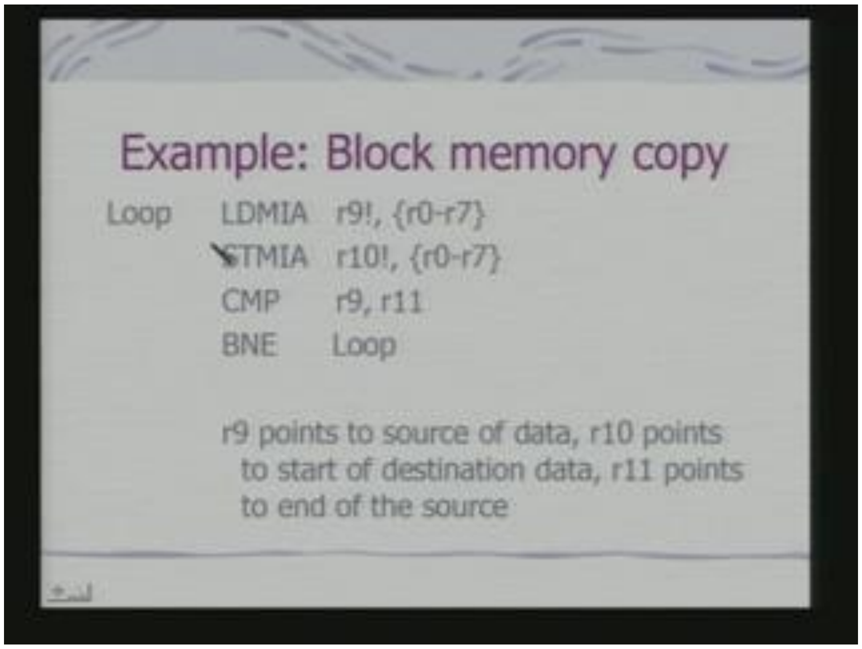


Branch Instruction

- Branch instruction : B label
  - Example: B forward
  - Address label is stored in the instruction as a signed pc-relative offset
- Conditional Branch: B<cond> label
  - Example: BNE loop
  - Branch has a condition associated with it and executed if condition codes have the correct value

The other one is conditional branch that is you branch on certain condition. In both the cases you have address label which is part of the instruction and is a signed pc-relative offset. So, you jump to the location whose address is calculated with reference to the current value of PC. Now, we can look at an example how to use this conditional jump instruction. In this case if you look at here, that we have used multiple transfers, multiple byte transfer instructions. This is load and this is store and then we have compared and then we have jumped. So, I can have r9 pointing to source of data and r10 can point to the start of destination data; r11 in fact r0 and r11 points to end of the source. So, what we are doing here? I am loading this r9 is points to the source of data; so data is loaded onto this registers and then I am storing them back. So, effectively these two instructions are doing block memory copy. If you look into it I am copying what- a set of memory locations from one base address, starting from one base address to another base address because I have got the source in r9 and destination base in r10. So, these are two distinct values so I can actually do a block memory copy using just 2 instructions. In fact here I am checking whether my r9 that is it is end of the source because whether I have actually reached the end of the source and if it is not again I am going back and I am actually doing the loop. So, this is the simple code snippet, okay, of ARM instruction by which we have shown how this load store compare as well as conditional branch can be used.

(Refer Slide Time 24:24 min)



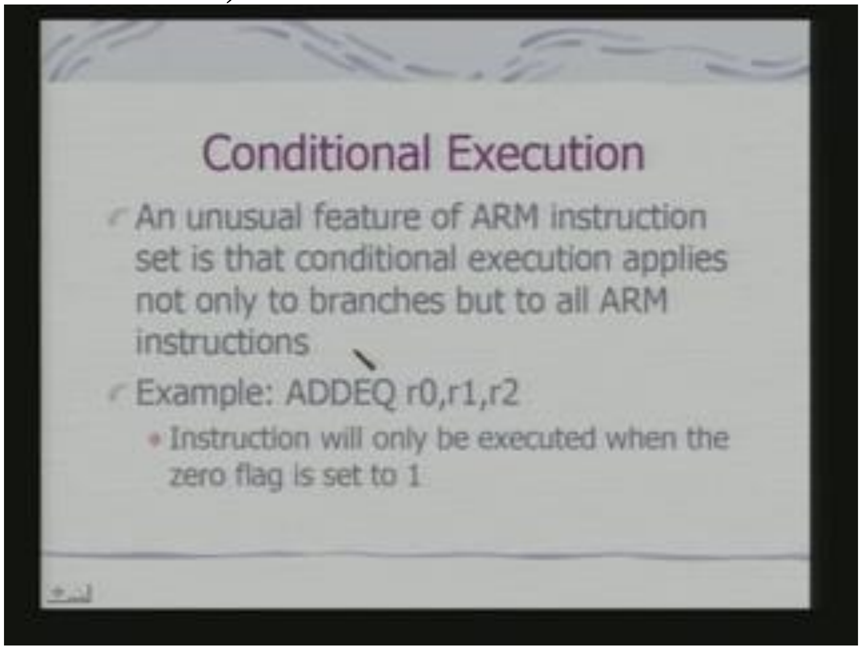
Example: Block memory copy

```
Loop    LDMIA   r9!, {r0-r7}
        STMIA   r10!, {r0-r7}
        CMP     r9, r11
        BNE     Loop
```

r9 points to source of data, r10 points to start of destination data, r11 points to end of the source

This loop is typically a label, okay which will be obviously used by that assembler to calculate the address. This address will be replacing these loop a symbolic reference and it will be calculated with respect to what, the current value of PC. The assembler will calculate the current value of the PC and with respect to the PC, the offset will be provided here for doing the actual jump. An unusual feature of ARM instruction set is conditional execution of each and every instruction. We have already shown that I can have branch instruction with condition code but it is not that you can use condition code only with branch instruction. You can use condition code with other instructions as well. Here is an example where we have shown that this addition, okay; so this addition instruction is associated with the condition code. Now, what does that mean? That is addition instruction will only be executed when the 0 flag is set to 1. That is exactly the condition code that I am referring to here. Now, if it is not so what will happen- this addition instruction will be skipped and next instruction will be executed.

(Refer Slide Time 25:27 min)

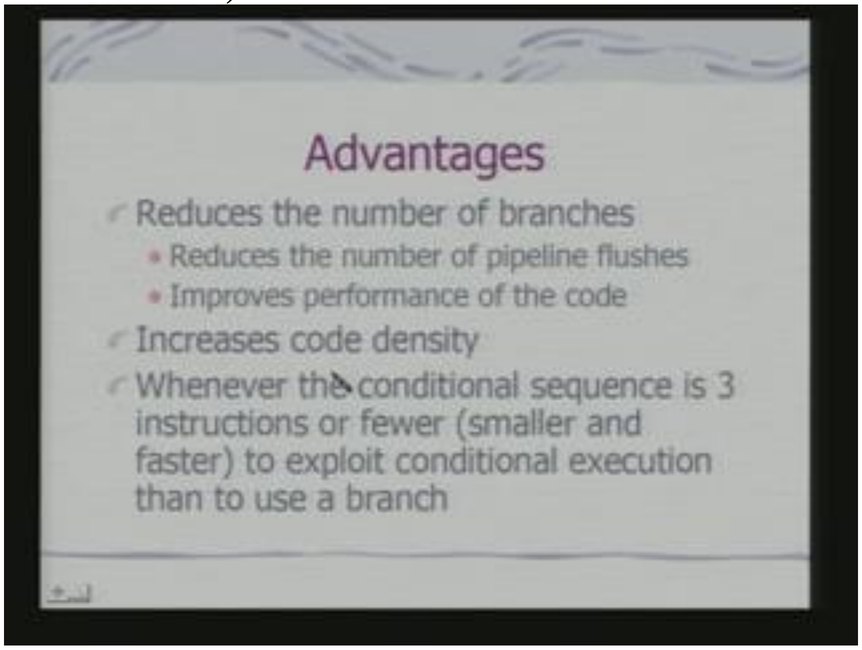


Conditional Execution

- An unusual feature of ARM instruction set is that conditional execution applies not only to branches but to all ARM instructions
- Example: ADDEQ r0,r1,r2
  - Instruction will only be executed when the zero flag is set to 1

So, effectively this ADDEQ will be converted to a no operation instruction. So, what are the advantages? Why is that, that in ARM instruction set these kind of instructions have been provided.

(Refer Slide Time 26:45 min)



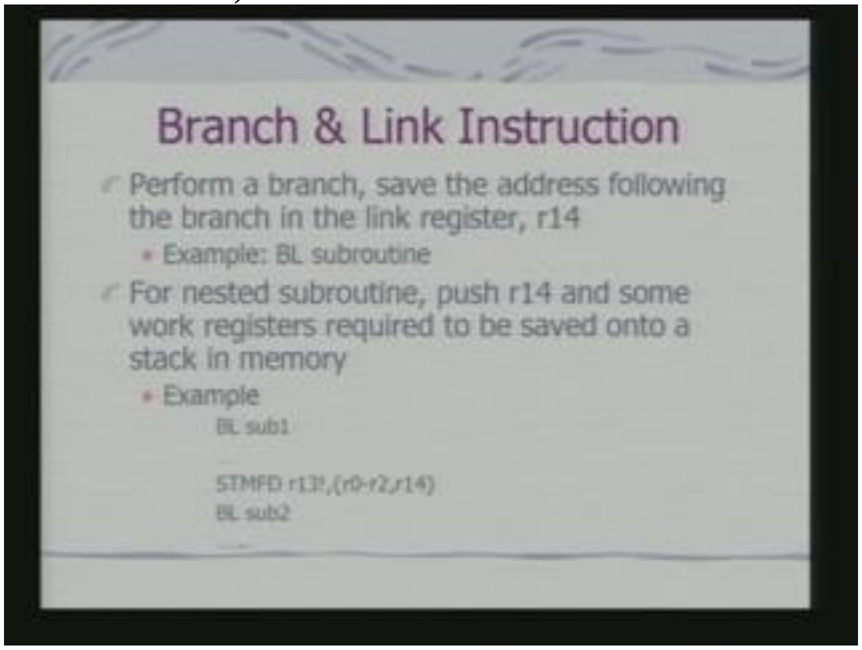
Advantages

- Reduces the number of branches
  - Reduces the number of pipeline flushes
  - Improves performance of the code
- Increases code density
- Whenever the conditional sequence is 3 instructions or fewer (smaller and faster) to exploit conditional execution than to use a branch

Obviously it reduces the number of possible branches, okay. As I reduce branching if I am implementing a pipelined architecture, then the number of pipelined flushed reduces because if I have a pipelined architecture then what happens; I need to pre-fetch the

instructions and when there is a branch the pre-fetch instructions have to be removed from the pipeline. Now, when I have a conditional instruction and I am not using branching then the conditional instruction remains in pipeline only that this instruction is not really executed and I replace that by nop, no operation. As a result, since I am reducing the number of pipeline flushes I have got an improvement in the performance. Also it increases code density. Why? Because a branch could essentially mean that I have to actually used a branch instruction if I think in terms of the previous example I have to used a branch on the condition code, branch on equality and then I have to used the add instruction.
So, the instruction count becomes 2. If I use a conditional instruction in this case instruction count is 1 and obviously my code density increases. So, a thumb rule says that whenever the conditional sequence is 3 instructions or fewer it is useful to exploit conditional execution than to use a branch. But if it is really a number of instructions that is to be executed on a particular condition is bigger than this, what will happen. If you use conditional instructions, your pipeline will only have effectively nops. So, when the condition is not getting satisfied, the pipeline will effectively execute nop. So, your CPU becomes under utilized, okay. So, when the branching that is branching say branch set of instruction is small enough, you should use conditional instructions rather than branch. That would increase code density and at the same time increase efficiency of your code.

(Refer Slide Time 29:54 min)



Branch & Link Instruction

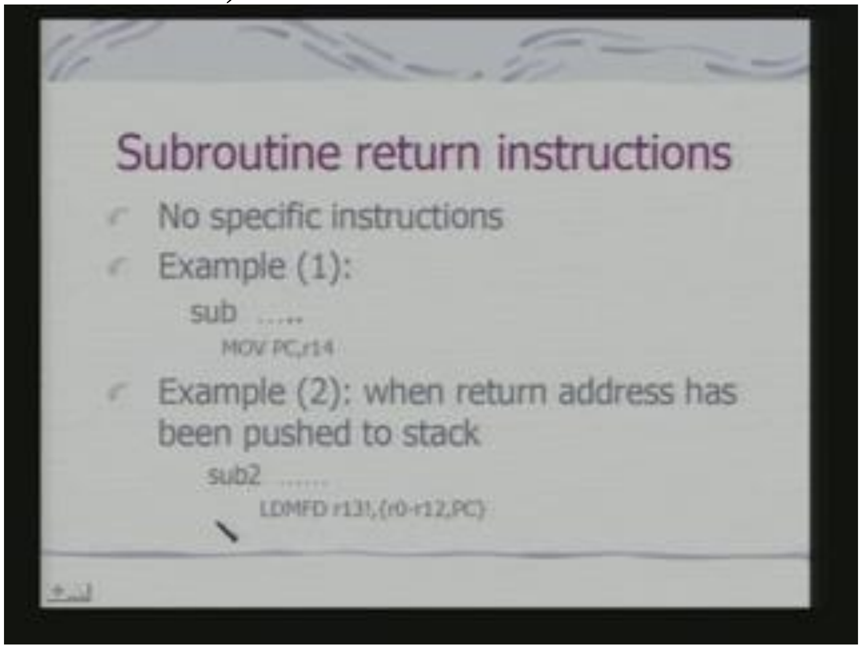
- Perform a branch, save the address following the branch in the link register, r14
  - Example: BL subroutine
- For nested subroutine, push r14 and some work registers required to be saved onto a stack in memory
  - Example

```
BL sub1
...
STMFD r13!,{r0-r2,r14}
BL sub2
...
```

Next, is branch and link instruction. This branch and link instruction is primarily used for subroutine call. So, it performs the branch; using this instruction you can perform a branch. But along with branching what happens? The address following the branch is saved in the link register, the next value that is the return address is saved it in the link register, okay. So, the basic different between ordinary branch and branch and link is the use of the link register. In case of a branch the next value of PC is not saved in the link register. In this case the next value is saved in the link register. So, here we are showing

an example that is when I do a subroutine call, I use branch and link subroutine. So, here I am branching to the beginning of the subroutine and the return address which could be the instruction following this branch will be stored in the link register. Now, I have got only 1 link register, okay and there maybe nested subroutine calls. What is to be done under that condition? For nested subroutine, you will be pushing r14 that is the link register and some work registers in the stack and stack will be set up in the memory. So, here I am just showing you how it is to be done. Say you are now in the first, inside the first subroutine. You have called the first subroutine, okay using BL sub1. So, the return address is stored in a link register. So, from this subroutine okay from inside this subroutine you would like to call this, okay. You would like to call this.
So, then what you will be doing, I need to save what, I need to save the link register as well as the current working registers. So, I use a multiple transfer instruction, okay multiple store instruction, multiple byte store instruction. So, where I am storing, I am storing to the location pointed to by r13 which is my stack pointer. What I am storing? I am storing the work register as well as the link, okay. So, the link register is link of the previous subroutine call. Now, when I execute this BL sub2, the return address from this will be stored in the r14, the current value of r14 and the previous r14 is now saved in the stack. So, this is how the nested subroutine call is to be managed ion ARM processors. Then how do you return from subroutine. Now, there are no specific instructions like return because the moment I can load my PC with the value of the link register I have returned to the main flow from where the subroutine was called. So, the simplest thing would be this that is move you move r14 to the PC which is your the r15 that is the register which is your program counter.

(Refer Slide Time 33:50 min)



Subroutine return instructions

- No specific instructions
- Example (1):

```
sub .....
    MOV PC,r14
```

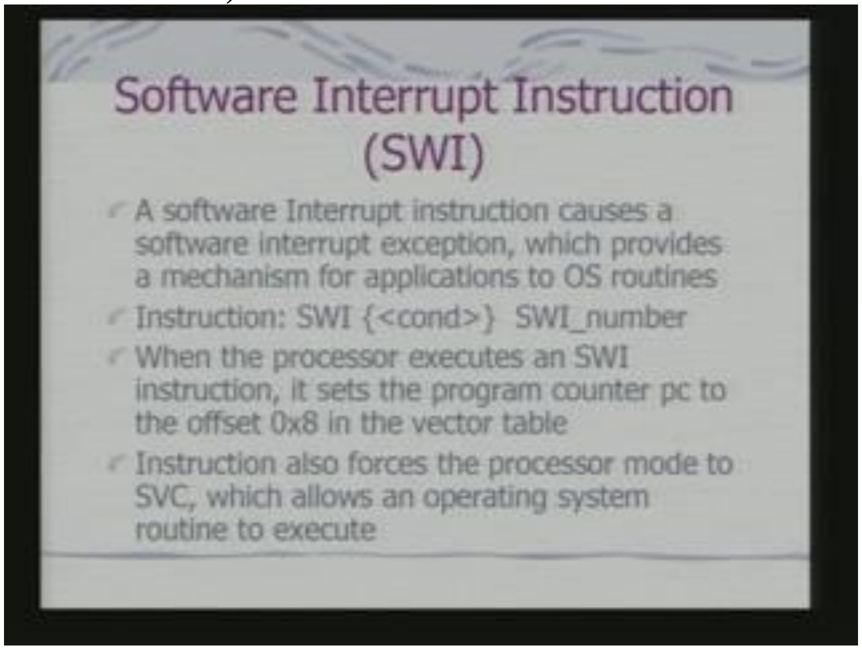
- Example (2): when return address has been pushed to stack

```
sub2 ......
    LDMFD r13!,{r0-r12,PC}
```

But when the return address has been pushed into the stack then you can use what, this kind of instruction okay a load instruction which uses the stack pointer register r13 and you load the value onto the set of target registers. Now, what is interesting in both these

cases you will find that when I am really returning from the subroutine, if I am using this multiple word or multiple byte transfer instructions, what it ensures, it ensures that you registers are always correctly loaded because this register transfer cannot be interrupted. If I am using, say for example, you do not have this multiple byte or word transfer instructions or you are not using this instructions you are using single register transfer instruction for loading the parameter pack onto the registers while returning from the subroutine what can happen. If an interrupt occurs in between you will jump possibly to an interrupt service routine, okay.
And these register will be lost and the state of the computation will not be correctly restored when you come back and in fact typically when I need to return from this kind of subroutines I would like to do what- disable interrupt. So, that the status of the registers are correctly saved, okay. If I am using this multiple data transfer instructions, I make sure that the state of the computation can not be corrupted by an interrupt .But what is the consequence, I already told you that interrupt latency increases. So, when you are writing a software these has to be kept in mind and your timing calculations have to be appropriately done. Next, we shall look at software interrupt instruction.

(Refer Slide Time 35:56 min)



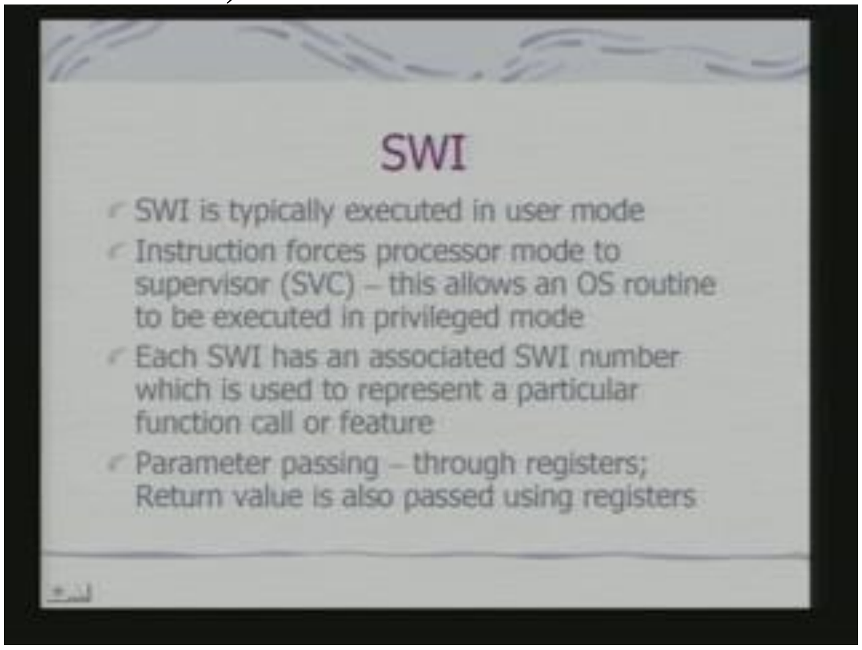
Software Interrupt Instruction (SWI)

- A software Interrupt instruction causes a software interrupt exception, which provides a mechanism for applications to OS routines
- Instruction: SWI {<cond>} SWI_number
- When the processor executes an SWI instruction, it sets the program counter pc to the offset 0x8 in the vector table
- Instruction also forces the processor mode to SVC, which allows an operating system routine to execute

A software interrupt instructions causes what we call a software interrupt exception and these provides a mechanism for applications to call OS routines.
Now, typically if you have this instruction that SWI which is software interrupt instruction; now just like any other ARM instruction I can have a associated with it a condition code and I have a software interrupt number associated with it. In fact in a way you can realize that software interrupt is what, you are actually calling a routine, okay. That means you are calling a routing which is part of the operating system and not part of your program that is user code. You are calling a routine which is part of your operating system and not part of your own set of code. Now, what is the different between a software interrupt and subroutine call; this is the basic different that in case of a

subroutine you actually call a subroutine which maybe part of your code and the subroutine can be located anywhere in the memory.
But when you are actually using a software interrupt, the software interrupt servicing has to start from fixed locations, fixed vector locations and that is why you can established a kind of a universal protocol for accessing OS utilities from the applications of the users because if the OS locates the utilities at different memory locations and if you have to use a subroutine call to do that, then it becomes an unmanageable situation; you have to remember and you have to be notified and told about the location of all these OS routines. So, that you can use them through a subroutine call when I using a software interrupt the protocol gets fixed, you exactly know where the interrupt handler is located. And there is another real advantage of using software interrupts in case of ARM there is a mode switch because I have already told you that your application program will run in user mode, but OS routines will run in supervisor mode.
So, when you have to actually call the OS routines, you have to switch mode from user to supervisor mode and supervisor mode is a privileged mode. So, software interrupt enables this switching of mode as well. In this case, in case of a ARM it sets the program counter PC to the offset 08 in the vector table. In fact I am not going into the details; this can be a different address as well. So as I have already told you, these software interrupt instruction is typically part of user program.

(Refer Slide Time 39:52 min)



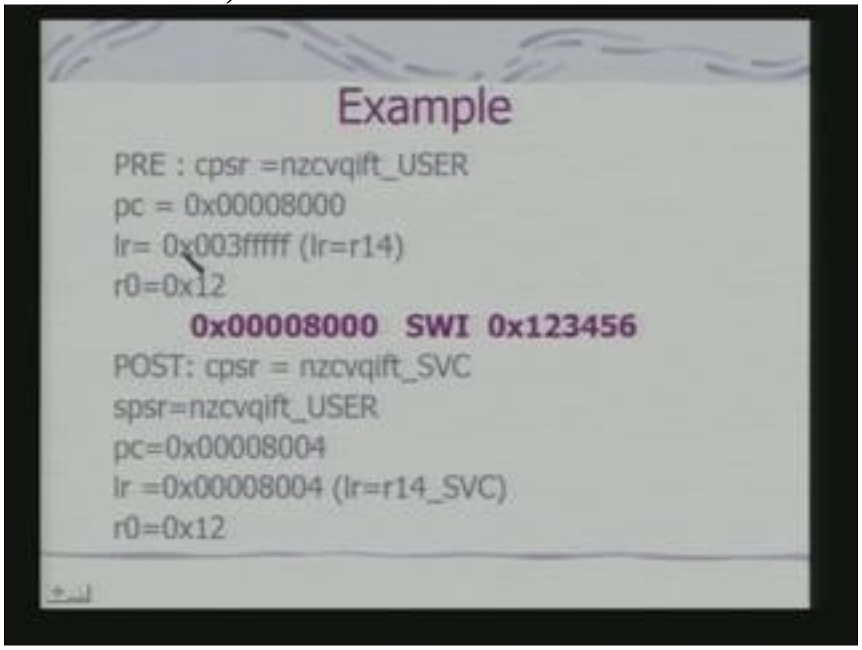
SWI

- SWI is typically executed in user mode
- Instruction forces processor mode to supervisor (SVC) – this allows an OS routine to be executed in privileged mode
- Each SWI has an associated SWI number which is used to represent a particular function call or feature
- Parameter passing – through registers; Return value is also passed using registers

So, it is executed in the user mode and instruction forces the processor mode to become supervisor and this allows the OS routine to be executed in privileged mode. Each software interrupt instruction has an associated number which is used to represent the particular function caller feature. But this number is not directly used by this instruction; please keep this in mind. This number is not directly used by this instruction. In fact what happens is, the software interrupt handler routine or the exception handler can use this number for identifying the service to be provided. You need to pass parameters, so use

typically registers for passing the parameters. In fact return value is also passed using registers. So, let us take an example; this is an example of a software interrupt instruction and in this case you have got, this is your CPSR which is the program status register I am showing you these are the flags- condition flags. These are your interrupt enable disable flags, this is the thumb mode flag and this is the mode bit which is now user mode.
This is currently of this value. So, currently this is the instruction which is to be executed and this is the software interrupt instruction. This lr is the link register value, some value okay which is not really consign right now in this context of discussion and this lr is what the r14 value.

(Refer Slide Time 41:11 min)

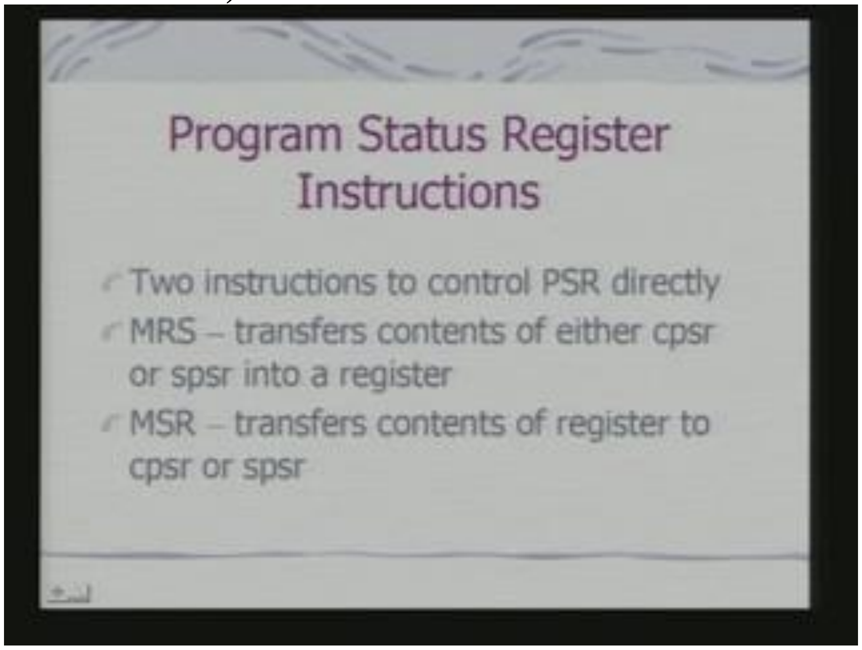


Example

PRE : cpsr =nzcvqift_USER
pc = 0x00008000
lr= 0x003fffff (lr=r14)
r0=0x12
0x00008000 SWI 0x123456
POST: cpsr = nzcvqift_SVC
spsr=nzcvqift_USER
pc=0x00008004
lr =0x00008004 (lr=r14_SVC)
r0=0x12

And you can have this register r0, okay. You may use r0 for passing parameters; so I am just showing one some value or 0 having 12. So, what happens when the software interrupt instruction is executed? These kinds of changes take place. Obviously the mode now switches to the system. So, it is SVC and this is what, this is saved program status register. I hope you remember this register that I had talked about in the last class; this is the saved program status register. This register will have the previous value, the previous value the mode was user, so that is saved, but in this case you will find that these bits remain unchanged. The other bits remain unchanged. Now, the PC value, okay has been changed to the desired location and now this lr is what, this lr value if you see, lr value is will be this location okay and what is the interesting feature. The interesting feature here is that your lr is what; now linked register is r14 is SVC. r14 SVC is what the copy of r14 which becomes available in SVC, okay. Now, we shall look at some of the program status register instructions and there are typically two instructions to control PSR directly. One is MRS another is MSR.

(Refer Slide Time 43:33 min)

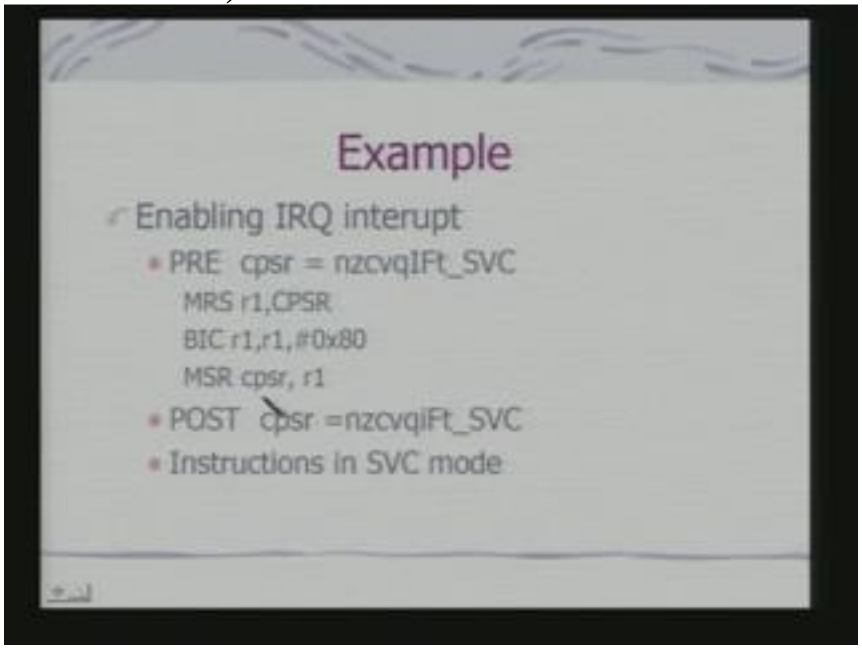


Program Status Register Instructions

- Two instructions to control PSR directly
- MRS – transfers contents of either cpsr or spsr into a register
- MSR – transfers contents of register to cpsr or spsr

MRS transfers contents of either cpsr or spsr into a register and MSR transfers contents of register to cpsr or spsr. Now, there are this example, example is that of enabling IRQ interrupt. So, how will you do it? So, the code uses these instructions, okay; this is for accessing cpsr, so I get the cpsr. Then I use the bit manipulation, the PIC instruction and then I load it back to cpsr. So, the PRE in this case I was not set and in this case I was set I am showing in a small the change.

(Refer Slide Time 44:04 min)



Example

- Enabling IRQ interupt
  - PRE cpsr = nzcvqIFt_SVC

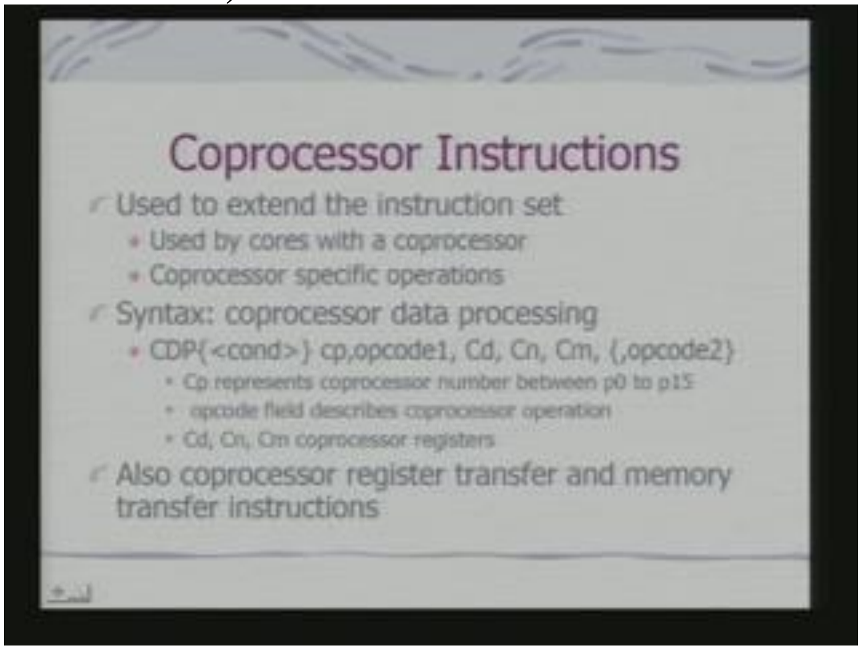
```
MRS r1,CPSR
BIC r1,r1,#0x80
MSR cpsr, r1
```

  - POST cpsr =nzcvqiFt_SVC
  - Instructions in SVC mode

So, now this is the modified status of the IRQ flag that is this is the masking bit. So, what I have, these instructions are typically executed in SVC mode. So, I told you that SVC is the privileged mode, so in that case you can actually modify this, these bits, okay. So, that is why you have got these instructions which are available in your privileged mode. In your user mode you can only change the flag bits and not the status and mode bits. Next, we shall look at co processor instructions. In fact this is again another interesting feature of ARM because ARM architecture as such is an extensible architecture. That means what we have discussed so far is basically the instruction set which is supported by the core ARM processor.

Now, I can have add on coprocessor to that core. These core processors can be targeted for specific applications; a very common is memory management application. Now, when we have got a coprocessor, what does the coprocessor mean, it effectively means that I am having another processor working with my original processor and what is interesting. Whatever instruction I actually fetch from my program memory that becomes visible or available in a sense to the coprocessor. And coprocessor can execute those instructions if those instructions are meant for coprocessor.

(Refer Slide Time 45:07 min)



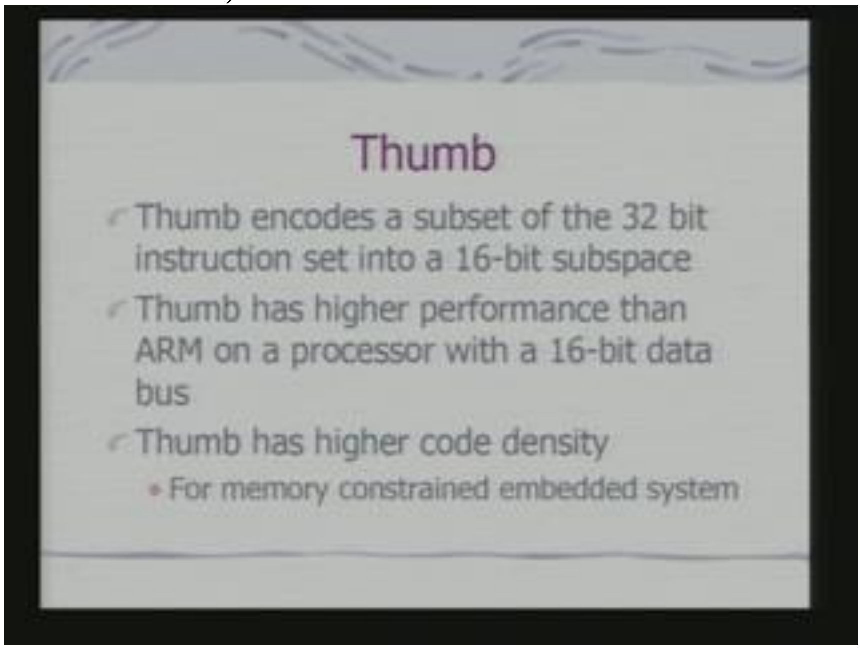
Coprocessor Instructions

- Used to extend the instruction set
  - Used by cores with a coprocessor
  - Coprocessor specific operations
- Syntax: coprocessor data processing
  - CDP{<cond>} cp,opcode1, Cd, Cn, Cm, {,opcode2}
    - Cp represents coprocessor number between p0 to p15
    - opcode field describes coprocessor operation
    - Cd, Cn, Cm coprocessor registers
- Also coprocessor register transfer and memory transfer instructions

So, that is why in the instruction set of ARM you have got what are called coprocessor instructions. In fact the pneumonic for coprocessor instruction is one of the coprocessor instructions that is coprocessor data processing instruction is CDP. Now, this CDP instruction will be useful if and only if there is a coprocessor core present in the actual chip- ARM chip that way you see. And these instructions will have as part of its parts of its operands the specification of the operation that coprocessor is expected to execute, okay. So, it will be for coprocessor specific instructions. The only thing is that when I have a CDP, so CDP will stand for certain binary code and by looking at the code ARM would know that this instruction of the coprocessor and coprocessor would know this is an instruction for the core processor to execute.

Now, ARM has a provision for having up to 15 to 16 coprocessors, okay. So, what you have got as part of the instruction, this filled with specifies the coprocessor number. Then you have got opcode, these opcode describes the operation for the coprocessor that means this opcode is expected to be recognized by the coprocessor and not ARM. And these could be the registers of the core processor for doing the operation; in fact optionally you can specify additional opcode for the target coprocessor. And this is a typical syntax for coprocessor data processing instruction. You have got similarly coprocessor register transfer and memory transfer instruction because if I have a coprocessor that is again another processor, it will have registers I need to a instructions to transfer data between the registers between register as well as that of memory. Now, we look at thumb. Thumb is what? A16 embedded 16 bit variant of ARM.
So, what we say in case of thumb, a subset of the 32 bit instruction set is encoded into a 16 bit subspace for thumb. In fact thumb has a higher performance than ARM on a processor with 16 bit data bus. What does it mean? It means that if I now build, ARM is basically a 32 bit processor, if I build a processor with 16 bit data bus then if I am using a 32 bit processor I have a degradation in performance, so if I use the 16 bit variant I shall have a much better performance because I shall be getting the 16 bit word and each memory. You can understand very simply your memory transfer becomes much more efficient. But actually what I have you got? You have got a thumb embedded into a 32 bit processor. So, when you use thumb, you use thumb when you actually require 16 bit operations and not really 32 bit operations. And in many cases the 16 bit is good enough to specify 16 bit operations and even 32 bit operations can be also specified by 16 bit instructions. So, thumb is good for specifying 16 bit, 32 bit operations using 16 bit instructions. If I am doing that effectively what happens? My code density increases and that is the prime motivational factor for enhancing ARM architecture with thumb mode.

(Refer Slide Time 48:48 min)

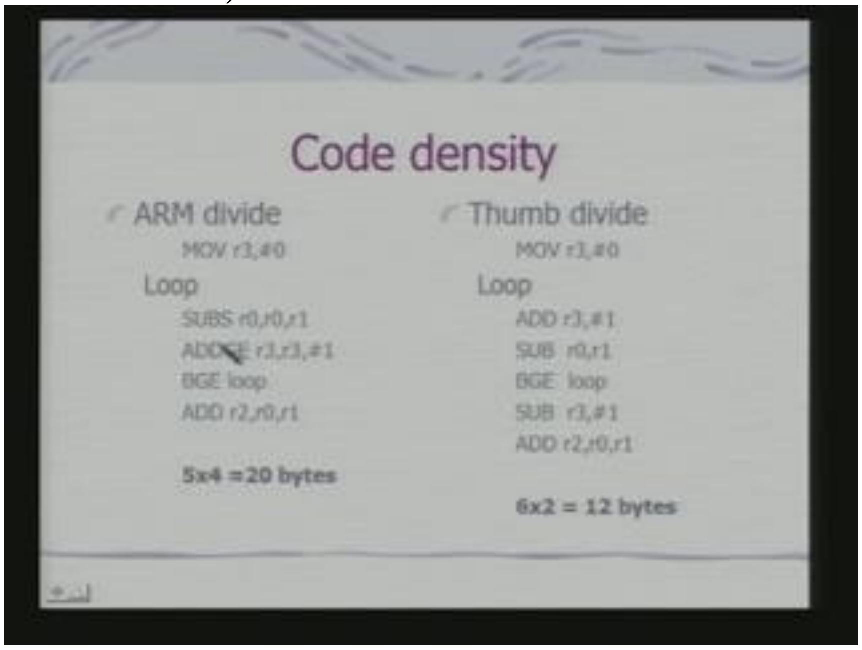


Thumb

- Thumb encodes a subset of the 32 bit instruction set into a 16-bit subspace
- Thumb has higher performance than ARM on a processor with a 16-bit data bus
- Thumb has higher code density
  - For memory constrained embedded system

So, thumb is targeted for what we call memory constraint embedded system. Let us take a simple example to understand this code density. This is a code for divide operation. This is ARM code when it is not operating in thumb mode and this is the code when it is operating in the thumb mode. Now, you will find here what I have done is obviously since it is a subset, the simplest thing is, you will not find this kind of conditional instruction in case of thumb because it cannot be accumulated in a 16 bit instruction word. 16 bit processor means a 16 bit instruction word and also you do not have this kind of variants sub. SUBs is what? The instruction when it FX is the flags. Now, it is a division operation implemented by I hope you understand repeated subtraction, okay

(Refer Slide Time 50:54 min)



Code density

- ARM divide

```
    MOV r3,#0
Loop
    SUBS r0,r0,r1
    ADDGE r3,r3,#1
    BGE loop
    ADD r2,r0,r1
```

5x4 =20 bytes

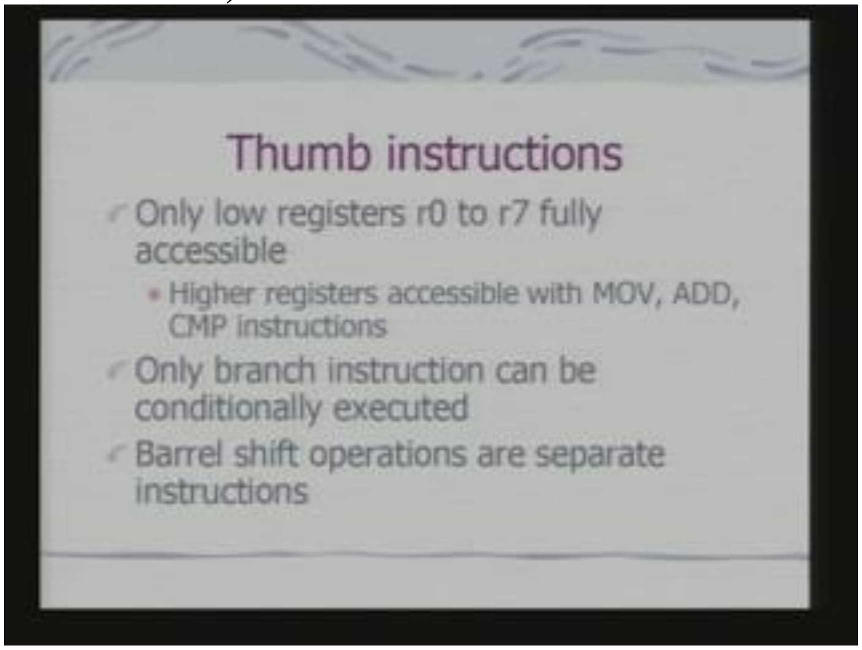
- Thumb divide

```
    MOV r3,#0
Loop
    ADD r3,#1
    SUB  r0,r1
    BGE  loop
    SUB  r3,#1
    ADD r2,r0,r1
```

6x2 = 12 bytes

Now, here the number of instruction is 5. Each instruction occupies 4 bytes. So, it is 20 bytes is the total memory requirements. Here, I am doing the same operation because if you look into it I have got my, I am doing operations involving same registers, okay, r0 r.1 And in this case what is interesting? The total number of bytes required for coding the same division operation is only 12, okay. And this increases effectively my code density. This is the reason why you have got 16 bit thumb more embedded into ARM, okay. In this case I really do not need to use 32 bit instructions. So, I can use 16 bit instructions, the instruction count can be more but the memory usage is less and hence I can write code which can be accumulated in less memory. So, typically thumb instructions are subset. So, there are some restrictions, only low registers r0 to r7 is fully accessible in all operations. High registers accessible with only MOVE ADD and compare instructions. Only branch instruction can be conditionally executed that is I do not use condition code with each and every instructions.

(Refer Slide Time 52:44 min)

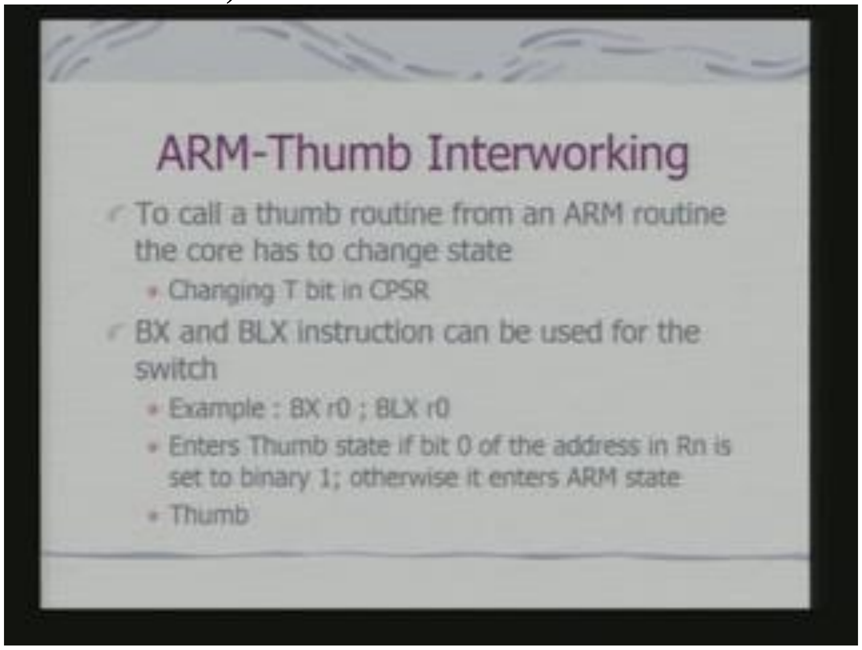


Thumb instructions

- Only low registers r0 to r7 fully accessible
  - Higher registers accessible with MOV, ADD, CMP instructions
- Only branch instruction can be conditionally executed
- Barrel shift operations are separate instructions

And Barrel shift operations are separate instructions. We do not provide it as part again the same issue is that the coding instruction within 16 bit word. And the next interesting thing is how do you switch from ARM to thumb. So, what we call ARM thumb inter working. In fact for this purpose we typically use the instruction BX and BLX.

(Refer Slide Time 53:00 min)

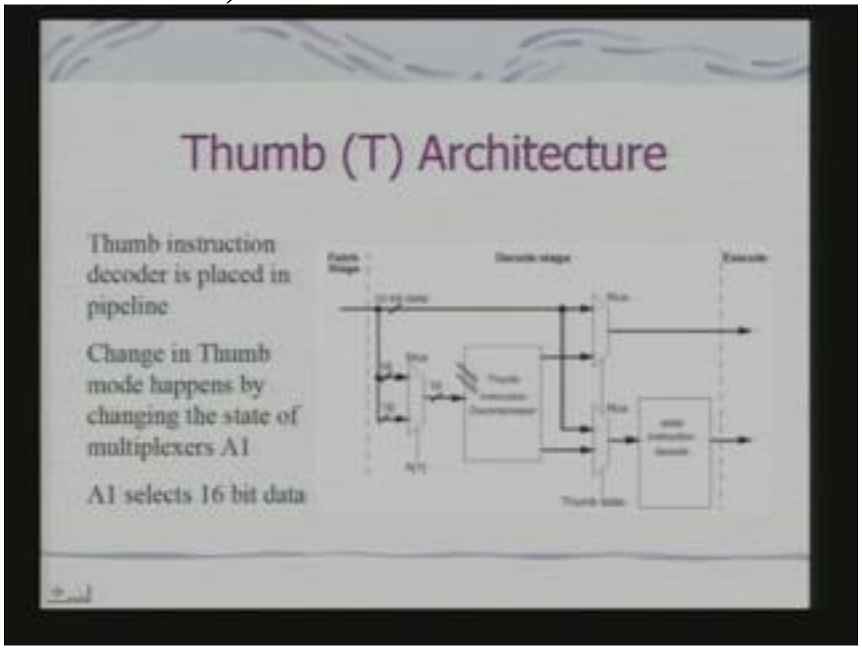


ARM-Thumb Interworking

- To call a thumb routine from an ARM routine the core has to change state
  - Changing T bit in CPSR
- BX and BLX instruction can be used for the switch
  - Example : BX r0 ; BLX r0
  - Enters Thumb state if bit 0 of the address in Rn is set to binary 1; otherwise it enters ARM state
  - Thumb

Now, BX is used, the typical syntax is BX r0 and BLX is r0. The BLX is similar to branch and link and this is typically branch kind of a thing, branch kind of an instruction. So, when I execute BX or BLX in ARM mode that I am currently executing in ARM mode and execute this instruction, it enters thumb state if bit 0 of the address in Rn because this register can be any of the registers. Here as an example I have shown r0. So, effectively bit 0 of the address which is specified in Rn is set to binary 1, okay. So, bits 0 if it is one of the address because this address is what the branch address, if 1 then this is interpreted as switch to thumb mode, similar thing true is BLX. If it is 0 send from thumb it can enter ARM mode. So, you can see that very easily I can do a switch, okay. Now, the interesting feature here you can under stand why this bit is used because if you typically look at that an address in case of a thumb mode will have last 2 bits 0 0 because it will be at the 32 bit boundary, okay. So, that if I am using 1, so I can tell the processor that I am now switching. Now, the architecturally if you see the thumb instruction decoder is actually placed in the same instruction data path.

(Refer Slide Time 55:06 min)



Thumb (T) Architecture

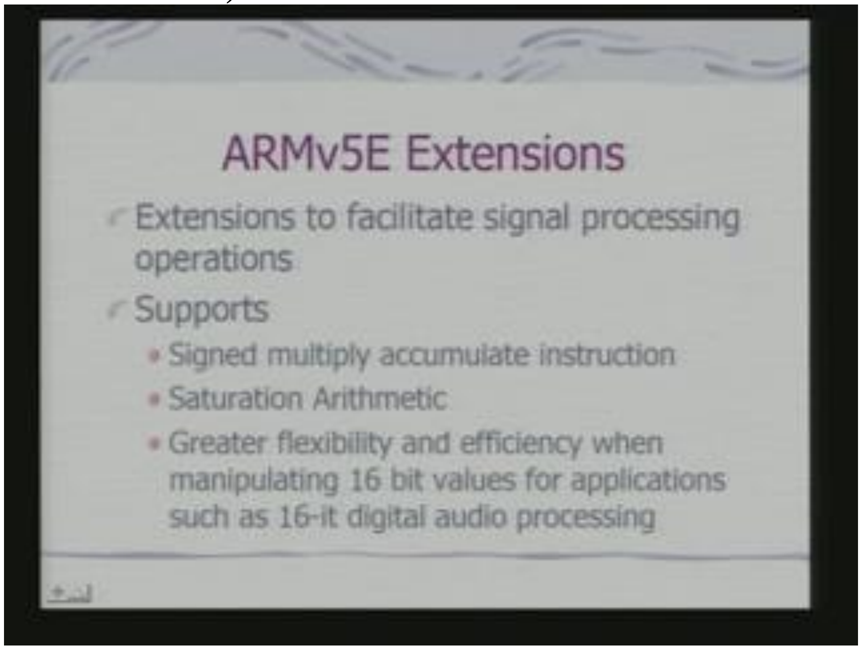
Thumb instruction decoder is placed in pipeline

Change in Thumb mode happens by changing the state of multiplexers A1

A1 selects 16 bit data

What we call instruction pipeline path of the decoder and this thumb instruction decoder is nothing but actually a thumb instruction de-compressor. That means it de-compresses the 16 bit compress instruction, okay to a 32 bit value which is actually decoded by the actual ARM instruction decoder, okay. So, now this multiplexer, okay you have a 32 bit data and then this multiplexer is enabled by the appropriate bit. If this multiplexer is enabled, then only this de-compressor the instruction will go through the de-compressor and de-compressor means what, the thumb instruction is actually translated to a 32 instruction internally so that it can use the same instruction decoder. Is it clear?
Externally you are using a 16 bit instruction; internally it is getting decompressed to a 32 bit instruction so that it can use the same instruction decoder. We shall now briefly look at this architecture 5E extensions. In this case, you have got what, this extension is targeted typically for your signal processing operations.

(Refer Slide Time 56:24 min)

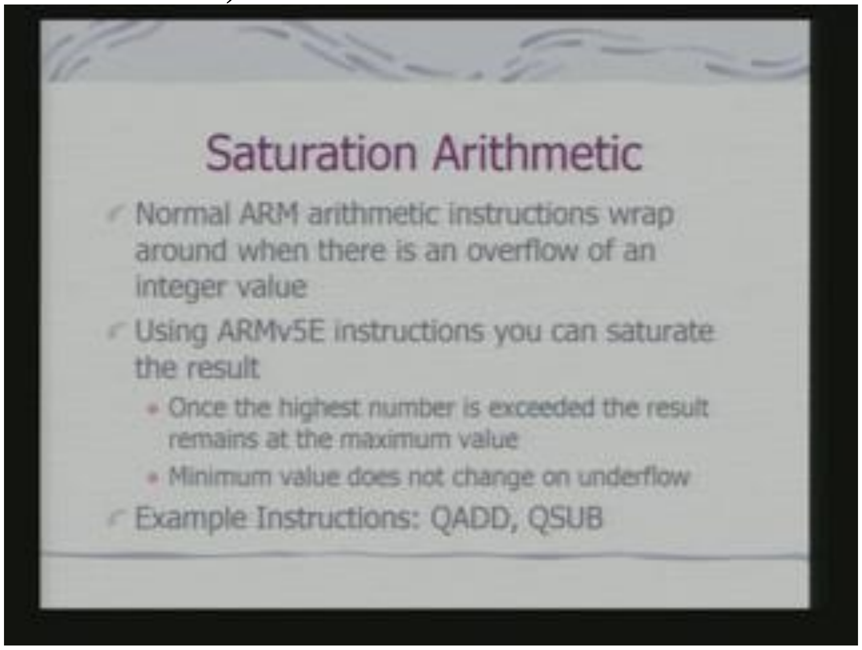


ARMv5E Extensions

- Extensions to facilitate signal processing operations
- Supports
  - Signed multiply accumulate instruction
  - Saturation Arithmetic
  - Greater flexibility and efficiency when manipulating 16 bit values for applications such as 16-it digital audio processing

So, you will find in the instruction set of ARMv5E, okay ARM version 5E- signed multiply accumulate instruction. We have already studied multiply accumulate but in those cases we say that multiply accumulator does not have a signed version. So, it has got a signed version, it supports saturation arithmetic and it has greater flexibility in dealing with 16 bit data so that it can be used for 16 bit audio processing in the ARM mode itself, okay. So, what is really saturation arithmetic? So, normal ARM arithmetic instructions typically wrap around when there is an overflow of an integer value. I hope you know all that; that is when it is all 1, 32 bit all 1 and if I try to add 1 its value will become 0. The similar thing, if there is an underflow B 1 0 0, it will come all 1. So, that is basically the under plan overflow wrap arounds. Now, using ARMv5E instructions you can saturate the result that means the result will be stuck at the maximum or the minimum value.

(Refer Slide Time 57:11 min)

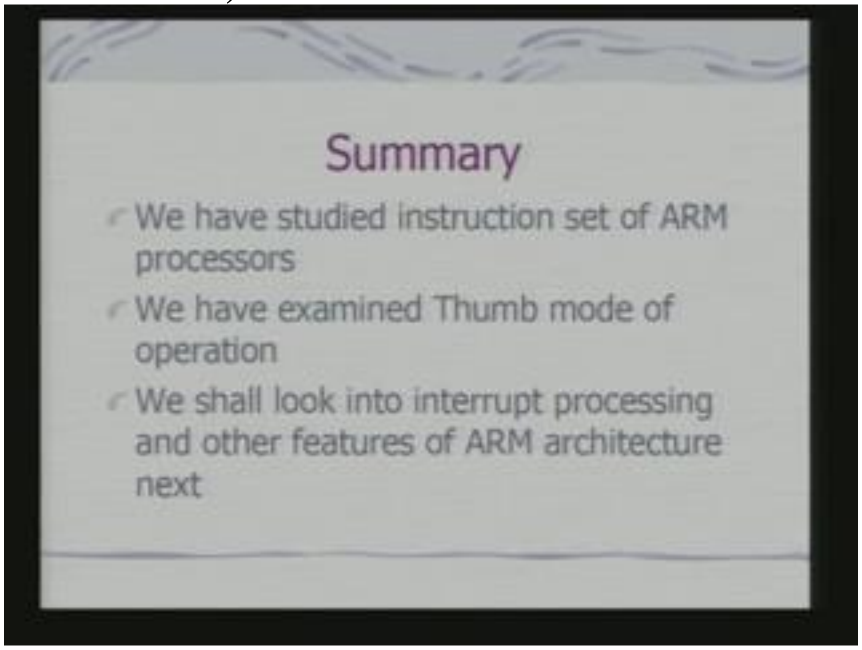


Saturation Arithmetic

- Normal ARM arithmetic instructions wrap around when there is an overflow of an integer value
- Using ARMv5E instructions you can saturate the result
  - Once the highest number is exceeded the result remains at the maximum value
  - Minimum value does not change on underflow
- Example Instructions: QADD, QSUB

So, the result remains that maximum or minimum value, okay. So, this is saturation arithmetic and for that you have got additional instructions. These instructions typically are indicated by Q as a first letter; I have given two examples QADD and QSUB. So, in this case overflow or underflow will keep the value of the corresponding result register at maximum or minimum.

(Refer Slide Time 58:14 min)



Summary

- We have studied instruction set of ARM processors
- We have examined Thumb mode of operation
- We shall look into interrupt processing and other features of ARM architecture next

So, this finishes our discussion on ARM instruction set. We have also looked at today thumb mode of ARM. And the other aspects of this ARM architecture, in particular, we

have not discussed in detail intra processing because you know intra processing is critical for any embedded applications. As well as other features of ARM architecture we shall take up next. Any questions?
See it is, the question is- what is the motivation for different mode of stack operations. This different mode of stack operation is decided by the programmer, what I have trying to illustrate is that these different addressing modes facilitate stack implementation in all these variants. It is not architecture defined. So, this, the flexibility that these different addressing modes provide you that enables a programmer to implement stack in any of these modes that suites his application. Any other question, okay then we shall meet in the next class and discuss remaining aspects of ARM architecture.