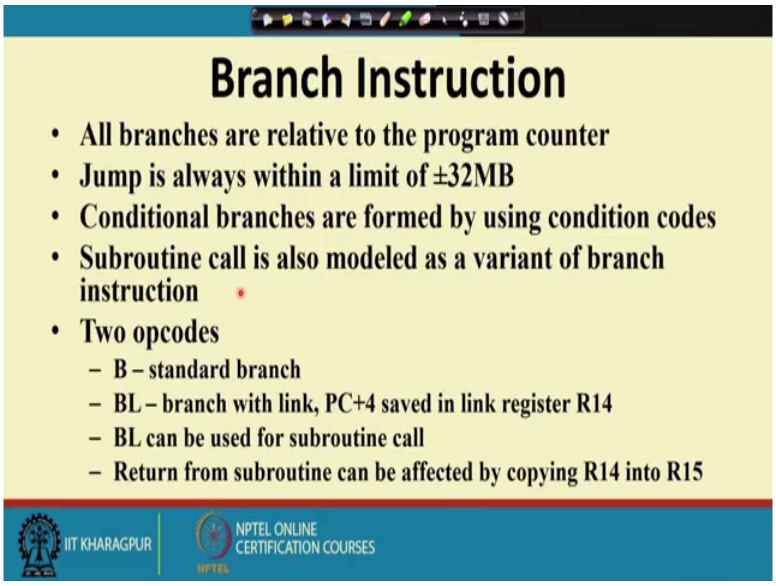
Microprocessor and Microcontrollers

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

Lecture – 47
ARM (Contd.)

So, we have branch instruction. Now, all branches in case of ARM processor so, they are relative to the program counter. So, previously we have seen that there are different types of branch. So, one classification was absolute versus relative branch. So, in absolute branch so, the target address was specified directly, so, the branching was to that particular address and the relative branch it is with respect to the program counter.

(Refer Slide Time: 00:20)



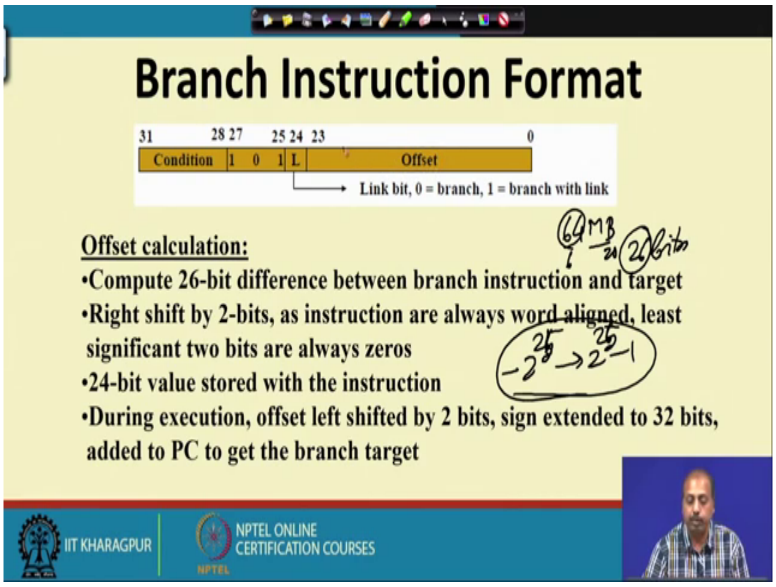
Branch Instruction

- All branches are relative to the program counter
- Jump is always within a limit of ±32MB
- Conditional branches are formed by using condition codes
- Subroutine call is also modeled as a variant of branch instruction
- Two opcodes
  - B – standard branch
  - BL – branch with link, PC+4 saved in link register R14
  - BL can be used for subroutine call
  - Return from subroutine can be affected by copying R14 into R15

So, instruction will have an offset value that offset value will be added with the program counter and whatever be the result so, the control will jump to that particular address. So, this is useful because it will help us in program relocation. So, that we have seen previously and this jump is always within a limit of plus minus 32 MB. So, so, it is not the entire address range, but plus minus 32 MB. Conditional branches you can of a make by having this condition codes like say branch is the simple branch is B now you can have say EQB like say for branch and equality like that. So, you can have the condition codes part of it.

Subroutine call is also a variant of branch instructions. So, we will have opcodes one is the B which is the standard branch instruction and the other one is BL which is the branch with link. So, subroutine call the difference is that the PC plus 4 value should be saved in the link register. So, if you use this BL instruction this BL instruction will save PC plus 4 into this register R14 and so, BL can also so, this can be used for subroutine call and return for sub subroutine. So, there is no return instruction so, you can just have this R14 copied back into R15. So, it is coming back from the subroutine. So, we can use that.

(Refer Slide Time: 02:11)



Branch Instruction Format

Offset calculation:

- Compute 26-bit difference between branch instruction and target
- Right shift by 2-bits, as instruction are always word aligned, least significant two bits are always zeros
- 24-bit value stored with the instruction
- During execution, offset left shifted by 2 bits, sign extended to 32 bits, added to PC to get the branch target

$-2^{25} \to 2^{25}-1$

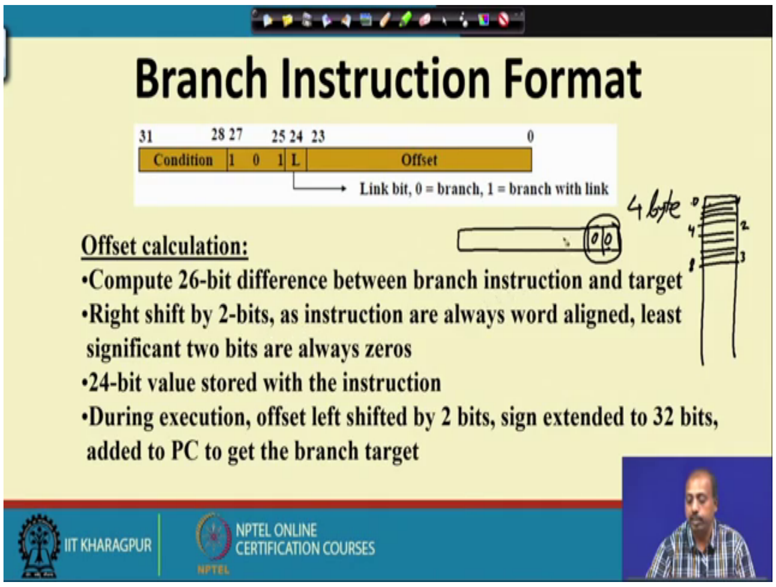
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, the branch instruction format is interesting. So, we have got this first most significant 4 bits. So, they will correspond to the condition code then this 1 0 1, so, this is the opcode for branch and then this L bit. So, L is the link bit. So, it is 0 for branch and one for branch with link. So, this is this 0 and 1 so, the branch and branch with B and BL instructions and in the offset part so, this will tell the offset by which you want to jump.

Now, you see that I have said that in the previous instruction a previous slide that this is sorry we have said that this branch is plus minus 32 MB. So, plus minus 32 MB if you want to go so, how much total space is total is 64 MB. So, 64 MB means so, this is MB is 20 bits and this 64 is 6 bits. So, total 26 bits of offset will be necessary.

So, in 26 bits of offset so, you can jump from minus 2 power 26 to 2 power 26 minus 1, so, so many space so, much of space that is plus minus 32 MB. Now, if you are doing that so, you can sorry this is 25 to 25 minus 1. So, if you if you want to do that then this 26 bit has to be stored now this offset part we have got space for only 24 bits. So, bit 0 to 23.

(Refer Slide Time: 04:21)



Branch Instruction Format

31–28	27	26	25	24	23–0
Condition	1	0	1	L	Offset

Link bit, 0 = branch, 1 = branch with link

Offset calculation:

- Compute 26-bit difference between branch instruction and target
- Right shift by 2-bits, as instruction are always word aligned, least significant two bits are always zeros
- 24-bit value stored with the instruction
- During execution, offset left shifted by 2 bits, sign extended to 32 bits, added to PC to get the branch target

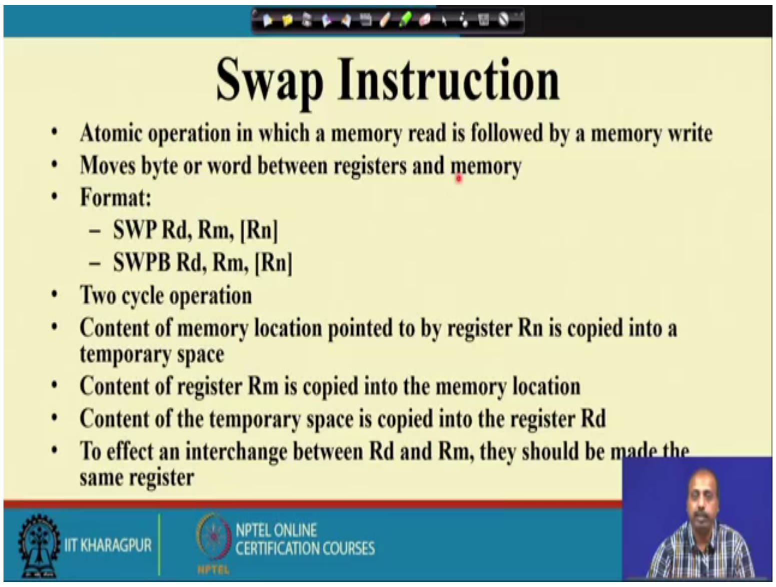
So, for that matter so, this 26 bit difference will be calculated, but the offset value that we have is only 24 bit. Now, why is it like this? So, you see that if you look into this ARM instruction so this ARM instructions are all 32 bit wide. So, it is 4 byte wide. So, if you look into the memory map the instructions when it is put into the memory. So, it will be like this the first instruction will be at location 0, 0, 1, 2 and 3. So, they will have the first instruction then the location 4 will have the second instruction 4, 5, 6 and 7 they will have the four second instruction then location 8 will have the third instruction. So, it will go like this. So, this is the first instruction, this is the second instruction, this is the third instruction. So, it goes like that.

So, what I want to mean is that you do not need to jump in between, ok. So, whenever you are jumping on to a particular location. So, you are jumping at the beginning of an instruction and all these addresses 0, 4, 8 etcetera if you there if you look into their binary representation then the most the least significant 2-bits are always0. So, these 2-bits are always 0. So, there is no point when you are taking the difference between the PC and this the jump target so, this least significant 2 bit results are always 0 so, there is no point storing those zeros separately.

So, what is done for this for this offset calculation purpose we first compute the 26 bit difference between the branch instruction and target and then we right shift the value by 2-bits. So, that these two zeros will go out and this least significant 2-bits are they are always zeros they will go out. As a result from this 26 bit you will get the 24 bit and this 24 bit will be stored with the instruction and during execution. So, you will get this 24 bit offset value and that 24 bit offset value will be left shifted by 2-bits to get 26 bit offset value and that will be sign extended to 32 bits and then it will be added to the program counter for branch target.

So, this way this branch operation will take place and it will be taking care of this so, this space optimized version you can say. So, in one instruction so, if entire effort is made to feed the instruction in a in one memory word 32 bit word. So, the it is done like that.

(Refer Slide Time: 06:46)

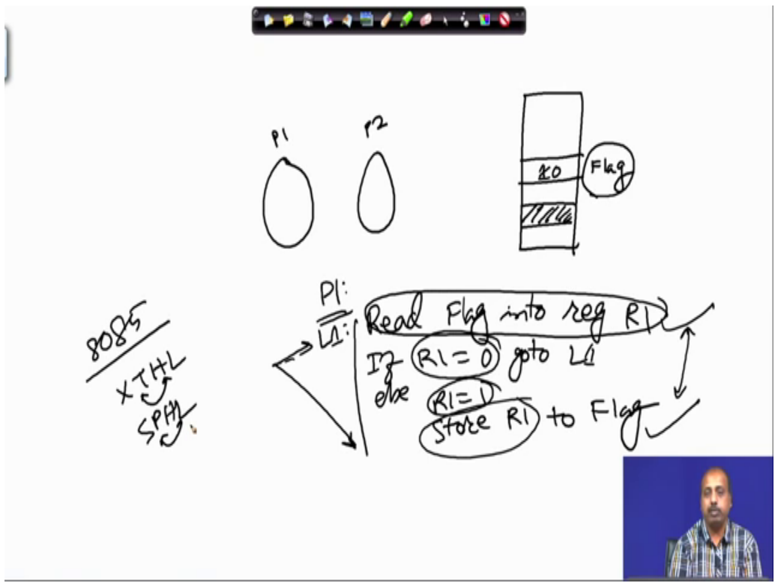


Swap Instruction

- Atomic operation in which a memory read is followed by a memory write
- Moves byte or word between registers and memory
- Format:
  - SWP Rd, Rm, [Rn]
  - SWPB Rd, Rm, [Rn]
- Two cycle operation
- Content of memory location pointed to by register Rn is copied into a temporary space
- Content of register Rm is copied into the memory location
- Content of the temporary space is copied into the register Rd
- To effect an interchange between Rd and Rm, they should be made the same register

So, next we look into another important instruction which is known as the swap instruction. So, swap instruction is an atomic operation in which memory read is followed by a memory write. So, if you look into other operations of any of this ARM processor memory operation so, it is either a memory read operation or a memory write operation, but it is never the same, but as far as the OS design is concerned so, whenever we are going for implementing some synchronization primitives like semaphore, monitor, mutex etcetera or some locking mechanism we need to gate the would check the content of a memory location and set it to some value in the same instruction.

(Refer Slide Time: 07:48)



So, what is required is it is like this suppose I have in a program so, there are so, the there is a memory location whose name is say flag and there are two programs. So, program P1 and program P2 both of them want to modify a portion of the memory. So, it wants to modify they want to modify this portion of the memory. Now, naturally if both of them modify simultaneously then the content will be lost. So, what this OS designers provide is they you can have some sort of flag like this. So, you can set this flag to 1 and when this processor 1 comes, so, it check process 1 comes the program 1 comes so, it checks the flag, finds that it is 1, 1 maybe it is allowed.

So, what this program will do so, it will may put it to 0 and go into this modification and after modification it will revert it back to one and p when the value is 0 if P2 comes so, it will find that the value is 0. So, P2 will know that it is not free so, it will wait until the value becomes 1 and when the value becomes 1 then possibly P2 will put it to 0 again and then proceed. So, this is a very standard protocol that is followed in the synchronization between processes.

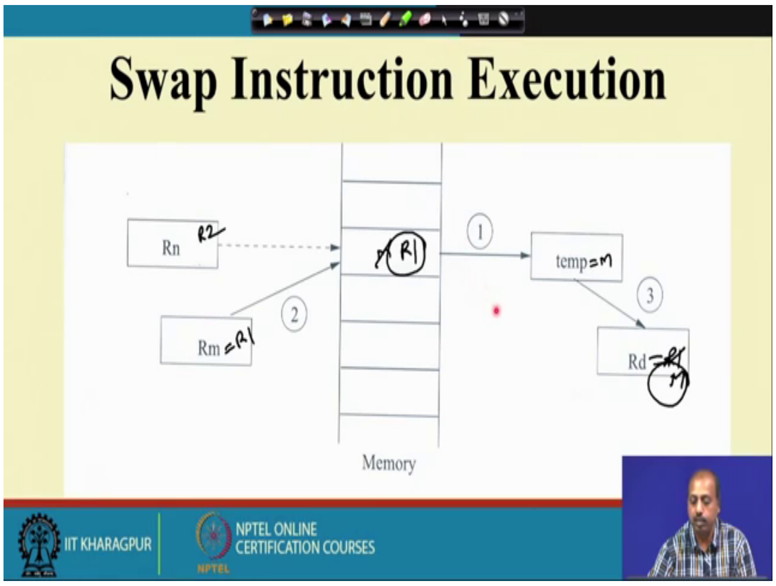
Now, the difficulty is that when P1 is checking the value of flag so, is so, it has to do it at a very in a within a very short amount of time and it should not be interrupted in between. So, I can the P1's code if I say if I write like this it may be like this that read flag into register R1 if R1 equal to 0 go to L1, where L1 is this instruction itself reading the again the reading the register content else you set R1 equal to 1 and store R1 to flag.

So, if I say that my arithmetic operations in an arithmetic operation I cannot save the cont[ent]- you cannot use memory location so, then you have to write the code like this where this reading is one instruction, then this condition check is another instruction this setting is another instruction and this saving is another instruction. So, total four instructions will be there. So, what can happen is that in number of read write operations one memory read and one memory write so, they are needed for doing this whole operation and since they are two different operations so, interrupt can come in between and that interrupting routine so, it can do some modification to the flag. So, that that way it may be the content may become inconsistent.

So, somehow I need a technique by which this read and write they are done simultaneously, ok. So, almost all the processors you will find some technique by which you can do it. For example, if you look into say the 8085 processor you remember that there is an instruction like XTHL where the stacked up content is exchanged with the HL register pair in is. So, that is so, these two contents are exchanged. So, in a single instruction so, we are exchanging the x in the stack top with HL or so, that way it or say SPHL instruction.

So, in a single instruction we are exchanging this pair. Why is it so important to exchange in a single instruction? Because that will avoid inconsistency into the values of those registers, but ARM so, far we have seen that it is not doing that it is it is having a load store architecture. So, in an instruction you can either load from memory or you can store into the memory. So, this swap instruction is the only instruction by which you can you can do this loads and load and store simultaneously in a in a single instruction. So, the format is like this is the swap read swap then read Rd, Rm and Rn. So, I so, I will just this diagram explains is it better.

(Refer Slide Time: 12:16)



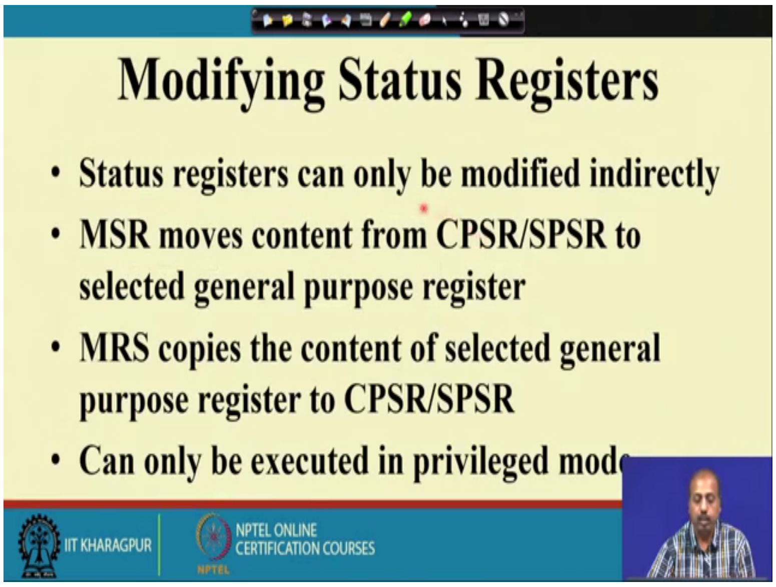
Swap Instruction Execution

So, this Rd, Rm and Rn in these registers are mentioned. So, first the first operation is from the memory location which is pointed to by the Rm register the value will be moving to the temp[orary]- a temporary register. Then secondly, this value of this Rm register will be put into this memory location and finally, content of this temporary register will go to Rd. Now, if the Rm and Rd they happen to be the same register. So, if this Rm is say Rm equal to R1 and the Rd is also equal to R1 and this memory location content is say M, and Rn is some other register suppose this is the R2 register which is just a pointer.

So, in the after the first operation this temp will be equal to M after the second operation this m will get the content of R1 and after the third operation this Rd will get the content of m equal to Rd will get the value of temp. So, that is M. So, what has happened effectively is that this R1 register and this R1 register and this memory location content they have got exchanged. After this whole thing is over this memory location has got the content of R1 and R1 has got the content of memory location. So, this way in a single instruction I can swap between the two inst[ruction]- to memory location and register provided these two are same.

So, this is this is written here so, we can. So, we can in a two cycle operation content of memory location pointed to by register Rn is copied into temporary space then the register Rm is copied onto the memory location and finally, content of temporary space is copied onto register Rd. So, if Rd and Rm are same in that case there will be an interchange of the content, ok. So, this is the utility of swap instruction and this is very much useful for the OS designers.

(Refer Slide Time: 14:25)



Modifying Status Registers

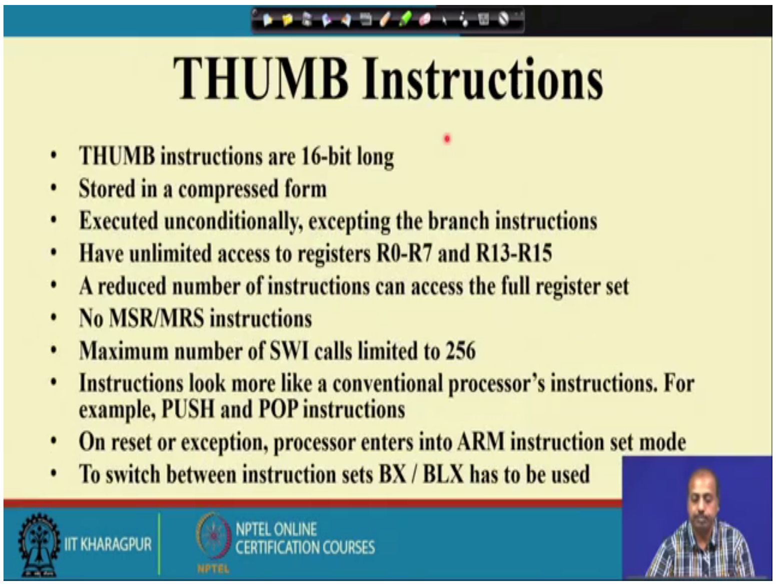
- Status registers can only be modified indirectly
- MSR moves content from CPSR/SPSR to selected general purpose register
- MRS copies the content of selected general purpose register to CPSR/SPSR
- Can only be executed in privileged mod

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Then modification of the status registers like the CPSR and SPSR register. So, this can be done only in the privileged mode not in the user mode. So, this there is an instruction like MSR that moves content from the CPSR or SPSR are registered to selected general purpose register. So, you can you can get it onto a general purpose register and then you can access that general purpose register to check the content and MRS it is it is doing just the reverse. So, the content of the general purpose register is copied on to CPSR or SPSR register and it can be executed only in the privileged mode.

So, this way we can modify the status registers in the privileged mode of operation.

(Refer Slide Time: 15:08)



THUMB Instructions

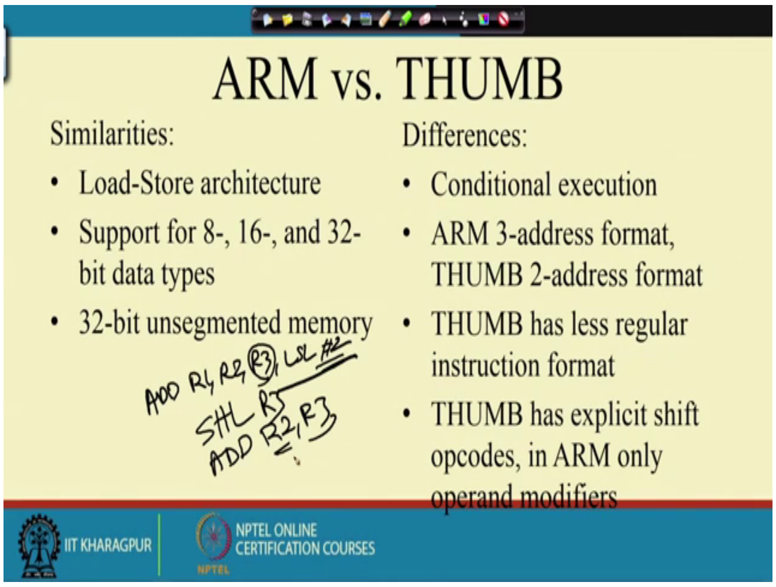
- THUMB instructions are 16-bit long
- Stored in a compressed form
- Executed unconditionally, excepting the branch instructions
- Have unlimited access to registers R0-R7 and R13-R15
- A reduced number of instructions can access the full register set
- No MSR/MRS instructions
- Maximum number of SWI calls limited to 256
- Instructions look more like a conventional processor's instructions. For example, PUSH and POP instructions
- On reset or exception, processor enters into ARM instruction set mode
- To switch between instruction sets BX / BLX has to be used

So, far whatever instructions we have discussed about so, they are all ARM instructions. So, we can have THUMB instructions as we said that THUMB is 16 bit instruction set. So, unlike ARM which is that 32 bit so, this 16 bit so, they are stored in a compressed form. So, in per memory word so, in memory what is 32 bit. So, per memory word, so, you have got two such THUMB instructions. So, I should say that this THUMB instruction set is more close to the standard microcontroller instructions. So, they are executed unconditionally. So, there is nothing like EQ add or say LE sub like that. So, you can have only add sub type of instructions.

And, the only the branch instructions are conditional, but excepting that all other instructions they are unconditional then they have got unlimited access to registers R0 to R7 only and R13 to R15. So, R8 to R14, so, they are not R R8 to R12. So, those registers are not accessible in the user mode. The reduced number of instructions will access the full register set MRS and MSR type of instructions are not available. Maximum SWI calls limited to 256. So, we are back to the convention that the Intel processors have they have got so, INT the software interrupts number 256. So, when you go to the THUMB mode so, it has got only 256 SWI calls then the instructions look more like conventional processors instruction like PUSH, POP instructions are back now, ok.

So, you can have this PUSH and POP instructions and on reset or exception this ARM the instruction set activated is the ARM instruction set and if you want to switch from R mode to THUMB mode so, you can use this BX or BLX type of instructions. So, BX will be branch with exchange of instruction set and BLX is basically the call with exchange of instruction set. Similarly, when you are executing the thump instruction set there you can put a BX to come back to the ARM mode of execution so, instruction set is ARM instruction set in that case. So, this BX and BLX these instructions can be used.

(Refer Slide Time: 17:35)



ARM vs. THUMB

Similarities:

- Load-Store architecture
- Support for 8-, 16-, and 32-bit data types
- 32-bit unsegmented memory

Differences:

- Conditional execution
- ARM 3-address format, THUMB 2-address format
- THUMB has less regular instruction format
- THUMB has explicit shift opcodes, in ARM only operand modifiers

ADD R4, R2, R3, LSL #2

SHL R3

ADD R2, R3

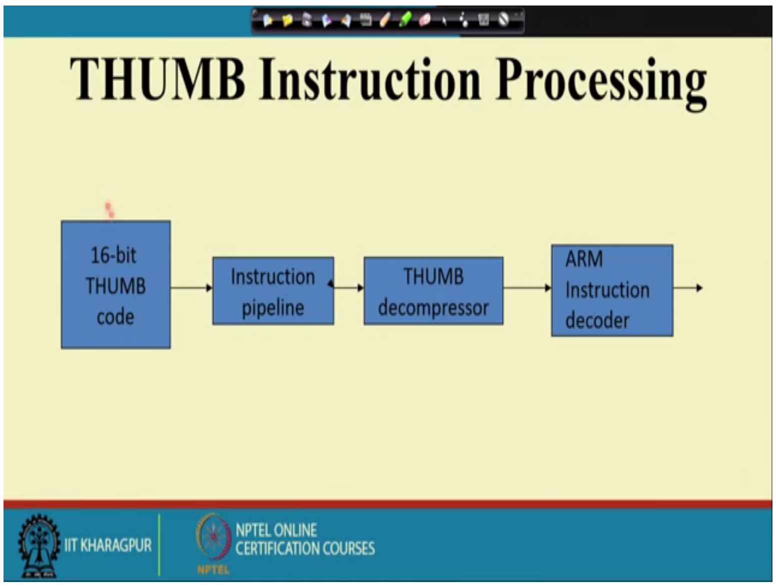
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, if you just want to compare between this ARM and thumb. So, it is like the similarity wise both ARM and thumb so, they are load store architecture they will support both will support 8, 16 and 32 bits of data and the memory module that is used is 32 bit ARM segmented memory. So, entire memory space is similar,. So, we do not have differentiation like this is true for some Intel processors where the memory is divided into code segment, data segment, stack segment like that, but in case of both ARM and THUMB it is taken as a un unsegmented memory. So, it is the responsibility lies with the programmer to have the conveyor have the understanding as a program memory or data memory like that.

As far as differences are concerned conditional execution is supported in arm, but it is not there in thumb. So, ARM in general has 3 address format whereas, THUMB has two address format, again that is the destination and source first operand they are same. So, it will be going back to the conventional instruction style. Then THUMB has less regular instruction format. So, there instruction format will be not that regular like ARM and it has explicit shift of course, like that in case of ARM you remember that we have got this the LSL in a part, but their modifier was put along with the opcode like say we have got say ADD we had instructions like say add R1, R2, R3 and then we said the LSL by 2 bits.

So, it was that R3 was left shifted by 2 bits. So, that shifting was part of the instruction itself, but in case of thumb. So, there is a this shifting is separate. So, first you first of all you have to do a shift left R3 and then only you can say ADD R2, R3 telling that R2 gets content of R2 plus R3. So, the two operand instructions mostly so, that will be the format.

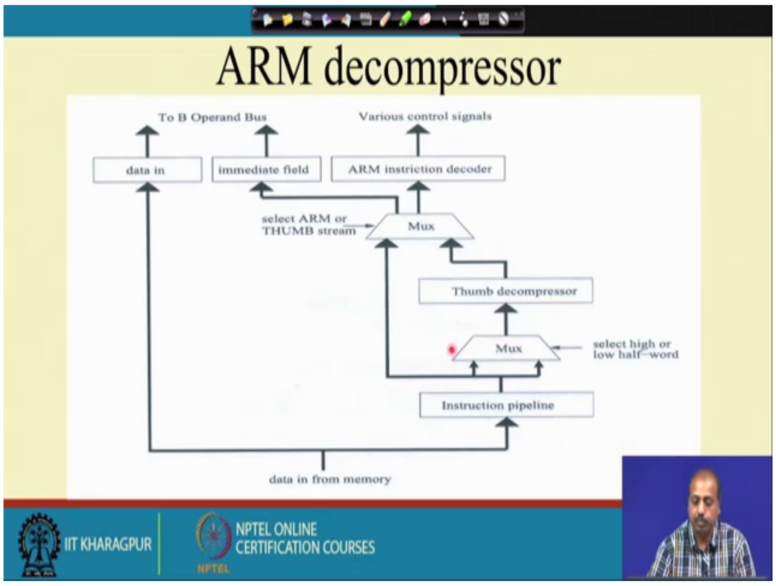
(Refer Slide Time: 19:54)



THUMB Instruction Processing

Now, other differences and similarities are like this. So, this is the instruction execution, ARM execution is faster. So, if you have got this 16 bit THUMB code so, it goes to the instruction pipeline and then it goes to a THUMB decompressor, ok. So, this decompressor will convert this ARM instruction a THUMB instruction into ARM instruction and that goes to the ARM instruction decoder. So, the processor does not execute THUMB instruction directly, it executes ARM instruction only. So, that way this requires some extra over write the THUMB instruction execution will be slower compared to the ARM instruction.

(Refer Slide Time: 21:31)

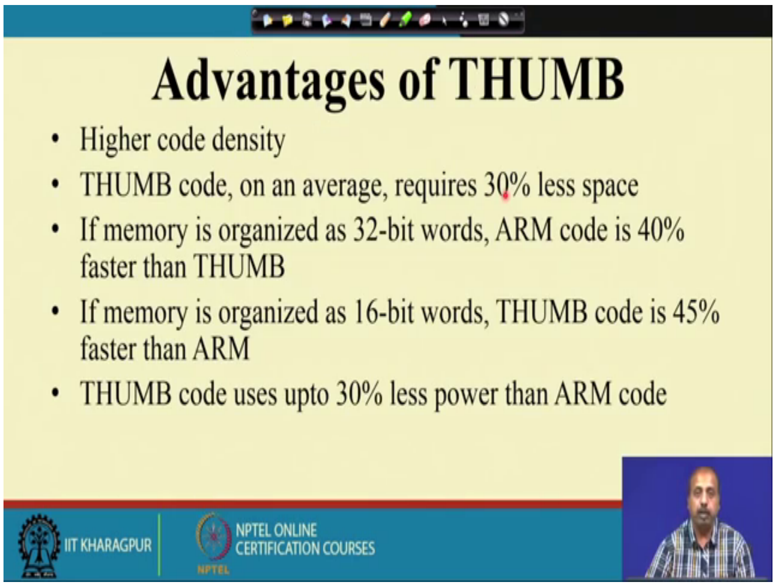

ARM decompressor

So, this is the way conceptually you can see the from that data from the data memory it comes. So, if it is an instruction so, it will enter into the instruction pipeline, if it is a data it will go to the da[ta]- data it will come as data in and it will go to the B operand bus B operand bus you remember that that was a part of the ALU ALU was getting a bus and B bus, so, this is the B bus.

Now, if it is an instruction it goes to the instruction pipeline. Now, so, we can have this select high or low half-word. So, 16 bits 16 bit is selected. So, it is going to the THUMB decompressor. So, is for the see the are for THUMB instruction. So, this is the this is consisting of two instructions now. So, one instruction is the higher order 16 bit and other one is the lower order 16 bit.

So, this multiplexer through this multiplexer so, with the so, it will select the instruction that will go to the THUMB decompressor or if it is a simple ARM mode then this part does not have any meaning. So, the instruction 32 bit instruction comes to this multiplexer and then it will be going to the ARM instruction decoder or it is from the THUMB decompressor so that the THUMB instruction that you have got here is converted into 32 bit ARM instruction and that goes to this multiplexer and ultimately the 32 bit is coming as output and out of the 32 bit some portion is the opcode so that will go to the instruction decoder part and some portion is may be the immediate data, so, immediate field. So, that immediate field will go to the B bus finally. So, that if the immediate part is there then it will be so, it will get that immediate field will get that value, if the immediate part is not there then of course, this instruction decoder will generate various control signals for doing the operations.

(Refer Slide Time: 22:20)



Advantages of THUMB

- Higher code density
- THUMB code, on an average, requires 30% less space
- If memory is organized as 32-bit words, ARM code is 40% faster than THUMB
- If memory is organized as 16-bit words, THUMB code is 45% faster than ARM
- THUMB code uses upto 30% less power than ARM code

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

So, why should we go THUMB like apparently it seems that, it is slower than arm, but why should we still go for the THUMB instruction set. First of all the code density is higher. Code density we have said that if you have a computation a function then the enough power, so, if you write down the corresponding program then how many bytes are needed for holding that program.

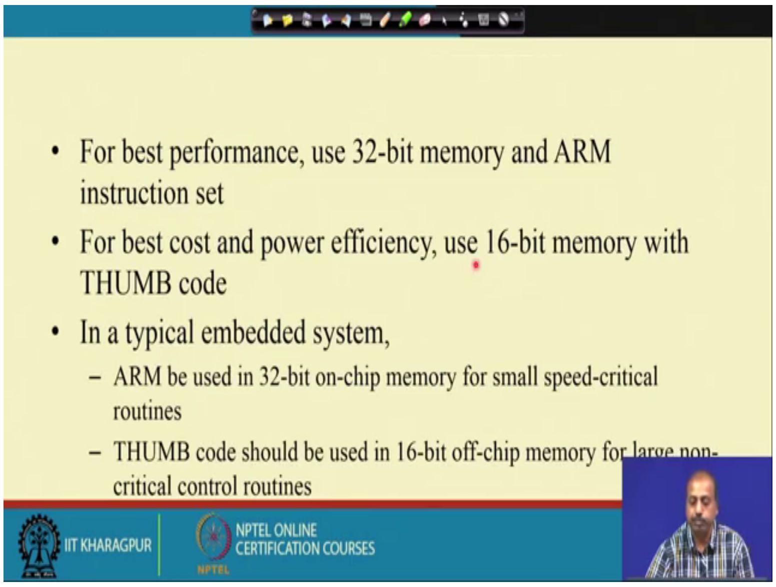
So, that is a measure of the code density the THUMB code on an average requires 30 percent less space. So, these are all statistical in data, so, it is not true in general, ok. So, it is so, if you take any arbitrary program so, this will not hold this may not hold, but in the statistically it has been seen that it requires 30 percent less space and if memory is organized that 32 bit word then the ARM code is 40 percent faster than thumb. So, that is a huge speed improvement. So, ARM will be running much faster.

And, as far as THUMB is concerned, so, THUMB is if it you organize a 16 bit word then the THUMB code is 45 percent faster than ARM. Why this thing happens? Because when the memory is organized as 32 bit word in each memory access the processor gets one ARM instruction. So, that way it is executing whereas, it gets a two THUMB instructions, but they are they are to be passed through that decompressor, so, that way it will be slowing down.

On the other hand if you are having 16 bit memory word then for getting a 32 bit instruction ARM instruction so, you need to access memory twice. So, that way the memory access time and particularly the memory is off chip then this memory access time will increase, so, this THUMB code will now be faster because every access will give it one instruction and that it can go for execution so, THUMB code will be faster. But, but the major advantage of THUMB is that it consumes 30 percent less power than ARM code, ok.

So, that is a huge advantage because this ARM processors they are meant for this power saving and they you can see that the using THUMB instruction so, you can have 30 percent less power consumption. So, that is one thing that we will look for.

(Refer Slide Time: 24:36)



- For best performance, use 32-bit memory and ARM instruction set
- For best cost and power efficiency, use 16-bit memory with THUMB code
- In a typical embedded system,
  - ARM be used in 32-bit on-chip memory for small speed-critical routines
  - THUMB code should be used in 16-bit off-chip memory for large non-critical control routines

So, this is the suggestion like for best performance we should use 32-bit memory and ARM instruction sets. So, whenever we are looking for performance of the system that is that delay of the system will be minimum. So, in that case we should go for 32-bit memory organization and go for ARM instruction set because the that will make the system fast.

On the other hand if you are looking for best cost and power efficiency then we should go for 16 bit memory with THUMB code because, now this power will be less power consumption will be less and this 16 bit memory so, this memory bus size and all those things will be less. So, this power so, cost of the system will come down, ok. So, cost of the memory space required will be less, so, the cost of the system will come down.

So, if you are looking into a into an embedded application then in a embed embedded application, so, all tasks are not equally critical. So, there are certain tasks which are very critical and certain tasks which are not so. So for the critical tasks so, the major emphasis there is to optimize the execution time and for that purpose we should use 32-bit on chip memory for speed critical routines, ok. So, as you know that ARM code is there so, along with that we can put some memory core and synthesize the whole thing together, fine.

So, the memory if I since the it is made on-chip so, I cannot make that memory very large, but I can put some small that is the most important jobs of the application, most important tasks of the application into that memory on-chip memory and use them I use the ARM instruction set for their coding so that it will be executed faster. On the other end this THUMB code so, there are less important operations of the system, so, I can use off-chip memory and so, the and store the program for that non-critical tasks onto that external memory and this they can be coded in the THUMB code because now, that will take less space, ok. Now, though it is the program may be large, but because of this coding the THUMB coding the program size will be small, ok.

So, that way this for embedded applications we need to use both of these ARM and THUMB instruction sets. ARM instruction sets will be for critical operations and THUMB instruction set will be for non-critical operations.