

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

Lecture – 43
ARM (Contd.)

In 8-stage pipeline so, this is another advanced version over that 6 stage pipeline.

(Refer Slide Time: 00:19)

8-stage Pipeline

- Two new features introduced in ARM11 core
- Shift operation has been separated into a separate pipeline stage
- Both instruction and data accesses are distributed across two pipeline stages
- Execution unit is split into three different pipelines that can concurrently operate and commit instructions out-of-order also

$x = x + y \ll 2$

The slide includes logos for IIT Kharagpur and NPTEL Online Certification Courses, and a small video inset of the professor in the bottom right corner.

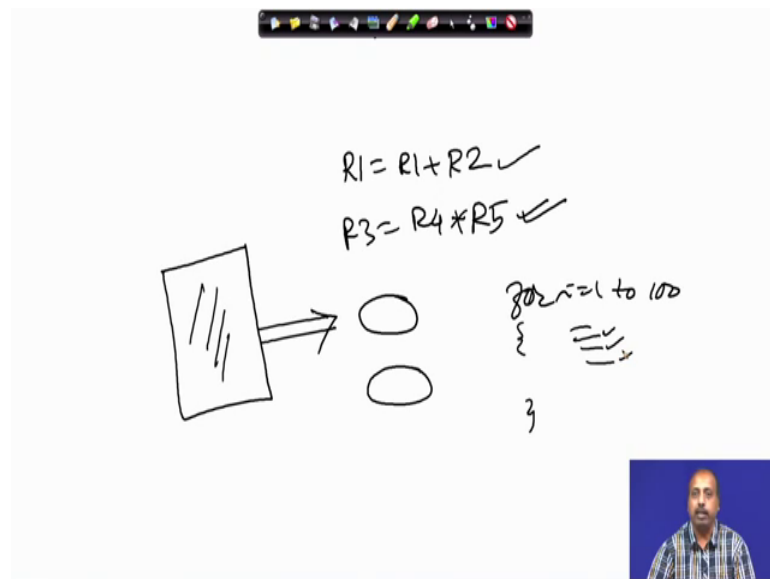
So, this is available in the ARM11 core. So, they are 2 new features have been introduced first one is the shift operation. So, shift operation it has been separated into a separate pipeline stage. So, you remember that in the ARM architecture we have shown that or before the ALU there is a barrel shifter. So, we can always do the shift operation in a separate pipeline stage ok. So, that way so, that has been exploited in ARM11 architecture.

So, it is a 8 stage pipeline so, the shift operation is constituting one stage. So, if you have got instruction like this so, where we are if you have got instruction like say x equal to x plus y left shifted by 2. So, while doing this operation this y left shifted by 2 so, this is done by a separate pipeline stage and this addition is done by a separate pipeline stage previously so, this was these two are done together.

So, that way it was taking more time now it is divided into 2 pipeline stages so, the operation will be faster ok. So, that way it helps in the that way it help in the pipelining this reducing the speed of reducing that delay of operation and both instruction and data accesses are distributed into 2 pipeline stages. So, the fetch part so, that is also divided into 2 stages so, we had the 6 stage. So, a pipeline from there we have got a new stage of shift. So, that makes a 7 then these instruction and data they are divide divided into a 2 pipeline stages. So, that is a 7 plus 2 9 pipeline stage and this execution unit is split into 3 pipelines and they can that can operate concurrently and commit instructions out of order also.

So, what do we mean by that let us try to understand. So, suppose we have suppose we have situation like this.

(Refer Slide Time: 02:22)



That we have got an instruction like say $R1 = R1 + R2$ and another instruction $R3 = R4 * R5$, now if you look into these two instructions then these instruction it does not affect the execution of this. So, they these instructions are independent of each other.

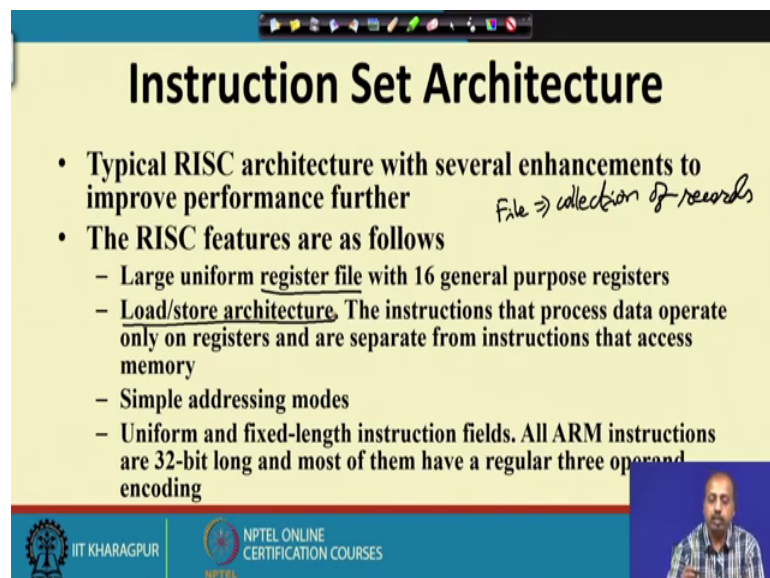
So, in that way if you write a program in some high level language, suppose this is the this is the program that we have written then you can find out into in this program several parts of the program they are that are not dependent on each other or if you are this is particularly true when you have got a loop body like say I am writing a loop for I

equal to 1 to 100 some statements are there. So, if you analyse this loop body you may find that many of this instruction they are not dependent on each other and they can be done parallelly.

So, if you have got multiple execution you needs available then you can that is you have got multiple ALU available then you can do all this operations parallelly, though for a normal human being it is difficult to find out where can we find this type of parallelism, but this is done by the compilers. So, they will find out like what are the points, what are the instruction that can be executed parallelly. So, they are actually the distributed into a number of functional modules and they are done parallelly.

So, this way we can have this we can have this execution divided into 3 pipeline stages and then we can this thing there the instructions can commit out of order also means it is not mandatory that the in the program the instructions 1 comes before instruction 2, but the in the actual execution instruction 2 may be completed before instruction 1. So, if there is no dependency between the two instructions so, it does not make any difference so, they can be completed out of order also. So, next we will see next we will see the 8 stage.

(Refer Slide Time: 04:40)



Instruction Set Architecture

- Typical RISC architecture with several enhancements to improve performance further *File ⇒ collection of records*
- The RISC features are as follows
 - Large uniform register file with 16 general purpose registers
 - Load/store architecture. The instructions that process data operate only on registers and are separate from instructions that access memory
 - Simple addressing modes
 - Uniform and fixed-length instruction fields. All ARM instructions are 32-bit long and most of them have a regular three operand encoding

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

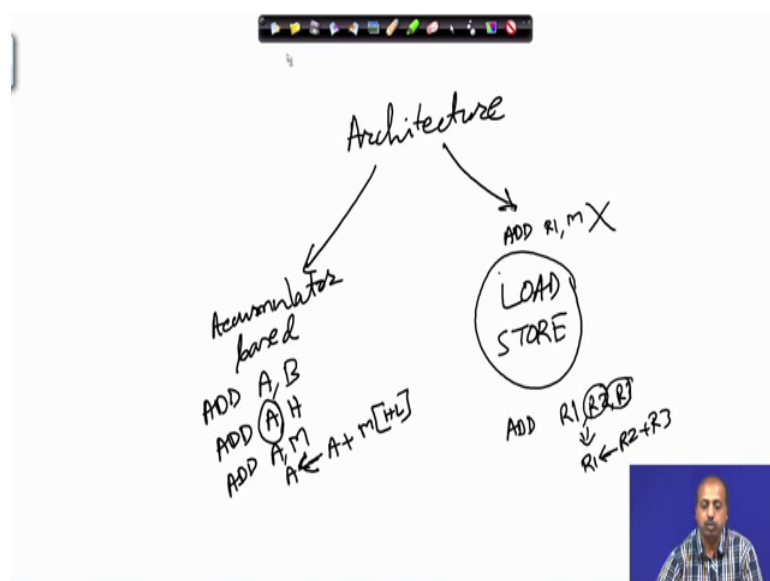
Pipeline that you it is 8 stage pipeline we have seen that completes the pipeline discussion, next we will look into the instruction set architecture for the RISC processor.

So, instruction set architecture means the instructions that are available in that further processor and also the registers that are available so, like that.

So, as we have already seen that these ARM is a RISC architecture it is a typical RISC architecture with several enhancements to improve the performance further. So, what are the RISC features it has? So, as we have noted earlier that the instructions are similar in terms of size and execution time and also this RISC processors they have got large number of registers in them. So, if we look into this ARM processor then here also we have got large uniform register file. So, this particular term file register file is this register file so, if you. So, you know that the term file is a the term file means it is a collection of records it is a collection of records so, if you look into any the data structure book so, it is collection of records.

But in case of in case of processors so, we the same thing happens like each register can be considered to be a record and we have got a collection of such registers so, they are often called register file ok. So, this is a standard terminology in computer architecture. So, this register file it has got 16 general purpose registers whereas, 16 such registers so, that that is quite big so, each register is 32 bit in all that. Then it fall it is something called a load store architecture so, load store architecture means the instructions that process data operate only on registers that separate from instructions that access memory so, will explain this.

(Refer Slide Time: 06:46)



Like say architecture what we have seen so far like say 8085, 8051 etcetera so, they are called accumulator based architecture they are called accumulator based architecture. So, why? Because if you do any operation one of the operand is the one of the operand is always that that special register accumulated and that destination is also that special register accumulated.

So, you always have instructions like ADD A comma B or you can have say ADD A comma H. So, like that so, but this one this first one of the operand is always the A register the accumulator in 8051 also we have seen that they are accumulator. So, and whenever you are accessing the memory so, you see that they are it is we can mention the address and then it can do the operation and also we also have got this is this type of instructions like ADD A comma M and we said that the for 8085 thus meaning of this instruction was A will get A plus memory location wanted to by the H1 pair so, this was the meaning ok.

So, you see that you can also add this accumulator with some memory location, in contrast so, this load store architecture that we have so, this load store architecture tells that we cannot have this arithmetic instructions to use this to use this memory like you cannot have this type of ADD say R1 comma M. So, this type of instructions are not possible for accessing memory we have got specific instructions LOAD and STORE so, LOAD and STORE.

So, these are 2 instruction that are available by LOAD instruction you can load some register with the content of some memory location and STORE you can store the content of a register on to a memory location so, but you cannot use this memory operand in the arithmetic and logic instruction so, that is possible. So, this type of architecture so, they are called load store architectures. So, the so, what happens is that if I am following a LOAD STORE architecture then whenever I am doing an operation say addition operation then the operands must be some registers. So, may be R1, R2, R3 which may mean that R1 gets the content of R2 plus R3.

So, this may be the meaning, but so, before that I had to somehow load this R2 and R3 by proper values if may be coming from memory and that way you have to use load instruction for that or there is the immediate operands are also possible. So, immediate part is not shown here, but immediate are also possible. So, this way we can have this

architecture is classified as load store architecture. So, this ARM processor they are following this LOAD STORE architecture.

So, instructions that process data operate only on registers and that separate from instruction that access memory, then addressing modes are simple ok. So, there are various addressing modes that are available in different processors like register, indirect, etc. So, we have in 8051 and 8085 we have seen that register indirect mode, but if you look into later versions of processors like ARM onwards then we will find that there are complex mode like base addressing then indexed addressing then base indexed addressing with post increment, pre decrement like that there are many such variants of the addressing modes.

And again the same thing when the architecture designers they thought about the features of the processor they thought that all these things will be helpful, but when it comes to the compilation of programs only a limited number of addressing modes are actually used by the designers by the compiler designers. So, as a result there is no point supporting those complex instruction modes and addressing modes.

So, in RISC architecture so they are made simple so; it is simple addressing mode, uniform and fixed length instruction fields. So, all the instructions they are more or less fixed their lengths are fixed and their structure is also fixed. All ARM instructions are 32 bit long and most of them have a regular 3 operand encoding. So, all instructions are 32 bit and all instructions are 3 operand instruction so, this is another point to note so, we can have instructions.

(Refer Slide Time: 11:43)

Instruction Set Architecture

- Typical RISC architecture with several enhancements to improve performance further
- The RISC features are as follows
 - Large uniform register file with 16 general purpose registers
 - Load/store architecture. The instructions that process data operate only on registers and are separate from instructions that access memory
 - Simple addressing modes
 - Uniform and fixed-length instruction fields. All ARM instructions are 32-bit long and most of them have a regular three operand encoding

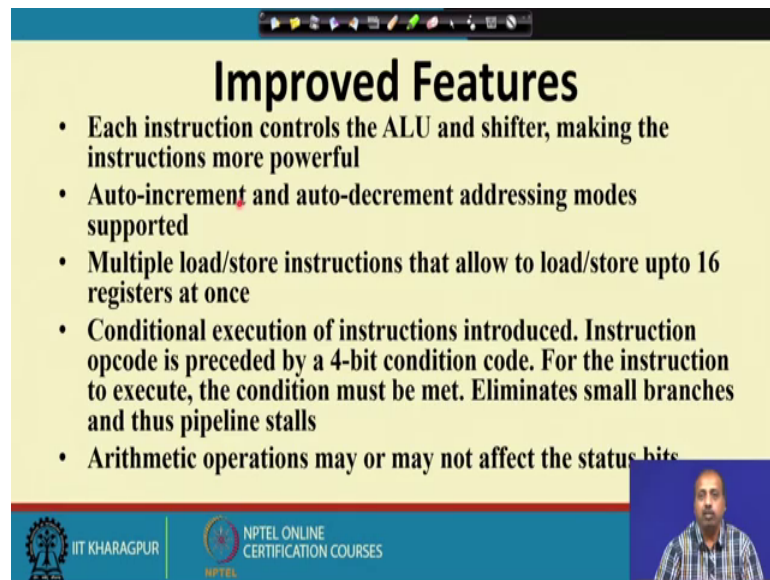
Handwritten annotations on the slide:
- "ADD A, B" with "2-operand" written next to it and "S, D" written below it.
- "ADD R1, R2, R3" with "3-operand" written next to it.
- "NOP" written below "ADD R1, R2, R3".
- "PUSH" with "3w" written below it.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Like say add R1, R2, R3. So, their R R2 have the operand 1, operand 2 and destination so, their 3 operands are there. On the other hand in 8085 or 8051 so, we have got the instructions like ADD A comma B. So, that is they are this a so, this first one is the source as well as the destination and the second one is the second operand. So, these type of instructions they are called 2 operand instruction they are called 2 operand instruction.

Now, this one this type of instruction they will be called 3 operand instructions and you can also have 0 operand instruction like say NOP, NOP instruction is 0 operand no operand is there. So, you can have one operand instruction like say PUSH so, if you look into the push instruction PUSH P s w so, this is a single operand. So, you have got single operand instruction, that way we have got different number of operands, but in ARM processor so, most of the instruction they are they are 3 operand in nature. So, it will have the 3 operand encoding.

(Refer Slide Time: 12:55)



The slide is titled "Improved Features" and lists the following bullet points:

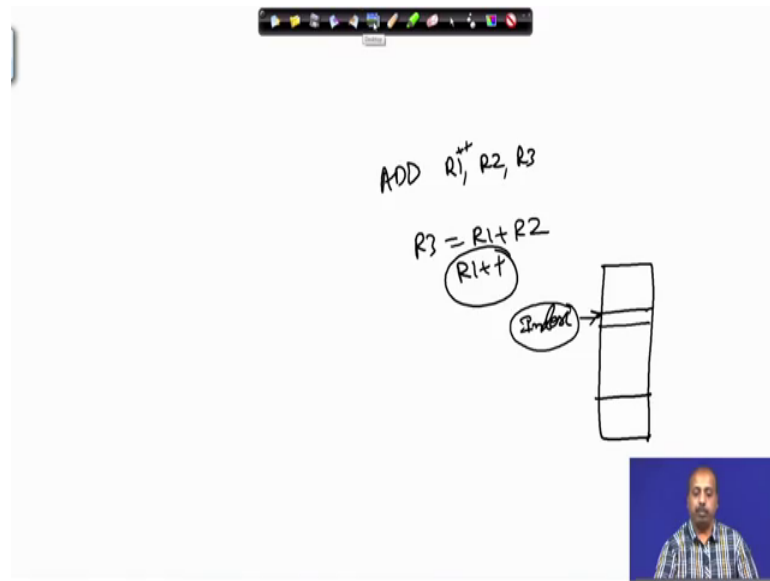
- Each instruction controls the ALU and shifter, making the instructions more powerful
- Auto-increment and auto-decrement addressing modes supported
- Multiple load/store instructions that allow to load/store upto 16 registers at once
- Conditional execution of instructions introduced. Instruction opcode is preceded by a 4-bit condition code. For the instruction to execute, the condition must be met. Eliminates small branches and thus pipeline stalls
- Arithmetic operations may or may not affect the status bits

The slide footer includes the IIT Kharagpur logo and the text "NPTEL ONLINE CERTIFICATION COURSES". A small video inset of a man is visible in the bottom right corner of the slide.

So, next we will look into some improved features. So, those are the basic features that are there, but there are many improved features also for ARM like each instruction controls the ALU and shifter and making the instructions more powerful. So, each instruction mean normally the arithmetic logic instructions so, they will be utilising this ALU and shifter modules, but in case of ARM processor we will find that this every instruction will be able to use this ALU and shifter that way instructions can become bit complex.

So, this is a diversion from RISC architecture where these instructions where the instructions are suppose to be simple, but now we are going towards a bit complex instruction so, this ARM processor is a mix of this RISC and CISC architectural features so, we will see that. Then auto-increment and auto-decrement addressing modes are supported so, auto-increment addressing mode means that you have got.

(Refer Slide Time: 13:57)

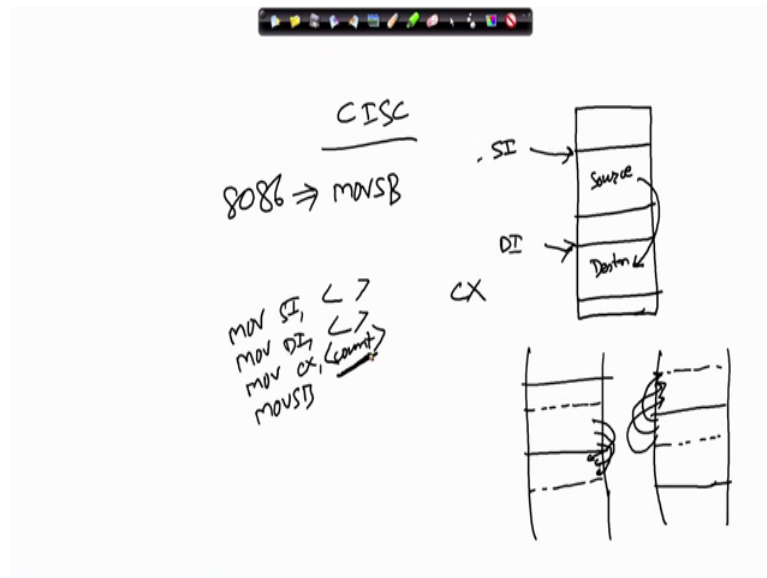


So, I can have say ADD R1 comma R2 comma R3 and then we can say that are these R1 plus sort of thing. So, that will mean that this R1 value will be added so, R3 will become say R1 plus R2 after that R1 will be incremented as well so, this is this is another job that is done.

So, this is very much useful particularly whenever you are writing code to access and array. So, array is normally stored in a memory like this. So, there is the initial address is there suppose the array is stored in this region of memory. So, initially you set your index here the first location then after this one has been access so, you want that the index should automatically increase. So, if you if that is automatically done then we do not have to spend another instruction for implementing this index part that that is why this auto increment is a very useful mode for this processor designers.

So, this though it is not a RISC feature because now; we are making the addressing mode complex ok. So, it is not a RISC feature, but this ARM processors it will have these auto increment feature, similarly auto decrement feature is also there with the ARM processor. Another diversion is multiple load store instructions that allow load store up to 16 registers at once so, multiple load store if the feature is like this so, this is basically a CISC feature.

(Refer Slide Time: 15:40)



So, this multiple load store is a CISC feature. So, in if you look into Intel 8086 family of architecture then you will find that there is a special instruction which is called the MOVSB. MOVSB, MOVSB B, MOVSB W, MOVSB word like that basically the MOVSB. So, what it does is that if you have got suppose if this is my memory and there I have got some chunk of data in some source. So, this is the source this is the source location a source chunk and from there I want to move that chunk to this I want to copy it to the destination fine.

So, what you can do you can set the source index register to point to this one and the destination index register to point to the destination, there are special registers SI and DI for source index and destination index and you can have in another register CX how many bites you want to transfer. And then after setting this SI register like MOV SI comma that source address MOV DI comma destination address and MOV CX comma count CX count. So, you can just say MOVSB for doing this movement ok.

Now this is a very useful instruction you see that so, this is in this case what has happened is that I this box is the or the blocks they are not overlapping. So, I can copy a from the first location and continue till the CX number of bites have been copied. But, consider the situation where in the memory my source block is say somewhere here. So, this is my source block and my destination is destination is like this my destination block is something like this, this dotted part is the destination.

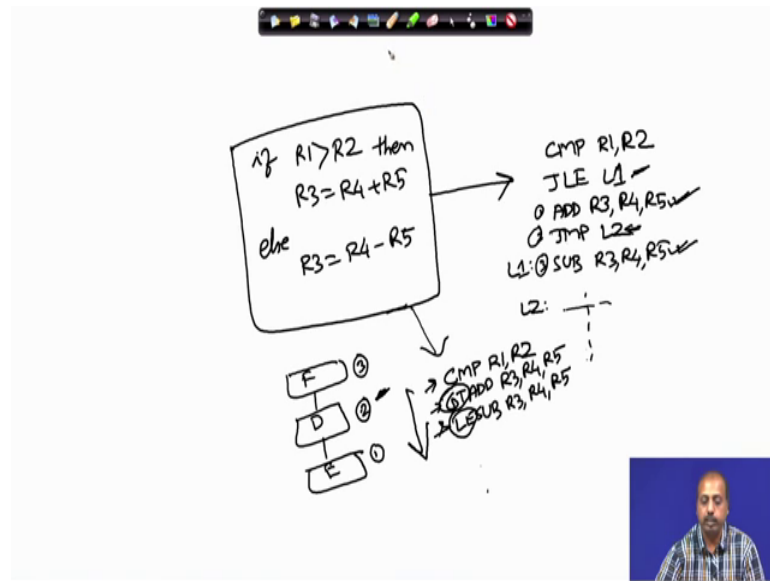
Now you see that while doing the copy I should not copy from the beginning because if I copy from the beginning then this content will be lost so, I should copy from the last location onwards. So, I should do it like this I should do it like this or if the overlapping is in the other way so, this is the source and this is your destination this is the destination. Then I should do the other way so, I should start copying from the beginning and do it this way in this fashion. So, all these are taken care of by the processor and how many cycles will it take for execution? That depends on the count value so, that is the blocks value.

How many bits are there in that block so, it will be dependent on that. So, this is a pure CISC feature because the whole operation is very complex, but at the same time this is a very useful feature because many a time in the program so, would like to move the content of one memory chunk to another that way will be very often requiring this type of operation.

So, this has been looked taken into consideration by this ARM designers and they have brought back this multiple load store instruction, but no not in the same form as we have in this as we are having in say Intel family of processors, but it is in a different way, but you can we can have this multiple load, store. Then the next important thing that we have is the conditional execution of instructions. So, instructions will be executed conditionally.

So, there is a 4 bit condition code before every instruction and if that condition code is satisfied then only the instruction will be executed otherwise the instruction will not be executed and this has the advantage of eliminating small branches and thus pipeline stalls. So, let us see how this thing happens. So, suppose we are having one if then will statement.

(Refer Slide Time: 19:57)



Like this in some high level language so, if R1 is better than R2 then R3 gets R4 plus R5 else R3 gets R4 minus R5.

Suppose in some high level language we have got this type of code, now if you consider the architectures that we are familiar with so, far 8085 or 8051. So, the conversion will be the machine code translation will be something like this, compare R1 comma R2 then jump on less or equal to L1 and then it will be here I will be doing that addition operation ADD R R3, R4, R5 meaning that R3 gets the sum of R4 and R5, then there should be a jump there should be there should be a jump to this jump for this to skip over the next instruction.

It should be a jump to L2 and then here I should have this level L1 and there I should do the operation subtract R3, R4, R5 and this is my level L2 so, this is the code. Now, you think about executing this code in a pipelined fashion so, even if I have a fetch, decode, execute pipeline fetch, decode, execute pipeline in which I am executing it so, when this instruction is executed so, when this instruction is being executed.

So, the pre the next instruction will be fetched, but you see that until and unless these instructions execution is over I do not know whether the next instruction is this one or the next instruction is this one. So, I really cannot fetch so, that the pipeline has to wait till the execution of the JLE L1 instruction is over and then only I will know what is the destination and then only the fetch operation can continued.

So, there is a pipeline stall for condition unconditional jump like here there is no problem because we know that if it is after this if a gets this jump L2 instruction so, after this one has been executed so, jump L2. So, it should fetch from this address, but anyway still you need to have that intelligence that decodes stage when this instruction is this instruction actually when say this instruction is in the execute phase then this jump L2 instruction is in the decode phase.

So, at that point I do not know what is the next instruction to fetch. So, the next instruction should be in the fetch stage. So, if number this instruction 1, 2 and 3 then when this instruction 1 is in execute stage, the instruction 2 is in the decode stage and instruction 3 should be in the fetch stage, but at this point at this point I do not know what is the next instruction. So, if this because this instructions meaning will be clear only when we come to the execute phase ok.

So, until and unless this instruction 2 comes to the execute phase I do not know what is the a next instruction. So, that way there is a problem so, whenever you have got branch instructions so, it creates difficulty, because we have got this pipeline has to be stalled in between. Now, what this a ARM people have done is that they do it like this so, you have got this compare instruction. So, compare R1 comma R2 so, that is there, then after that it will be it will be putting this instructions like this.

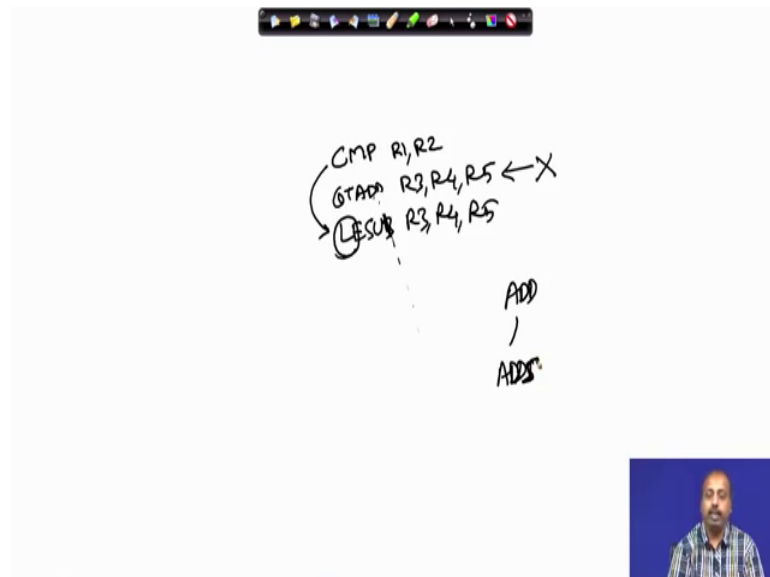
So, it is called GTADD GTADD R3, R4, R5 and we also have after that we put LESUB R3 R4, R5 fine. So, what is happening is that so, first it will be compared then the next instruction when it goes to execute phase. So, it will check whether the greater than flag is set or not so, if the greater than flag is set then only this addition will be done otherwise it is same as the NOP no operation.

Then the next instruction will come into execution and depending upon the condition if the less or equal condition is true then only this sub will the subtraction will take place. So, you see now you do not have to stall the pipeline. So, you can continue executing the instructions knowing that this prefix part that we have this conditional code that we are putting before the instruction so, that will take care of this type of problems ok. So, this pipeline stalls will be much less in case of this conditional execution with the availability of conditional execution this pipeline stalls will be less.

So, that is the thing the last point that we have here in this slide is that it is the arithmetic operations may or may not affect the status bits. So, normally we have seen that in other processors is arithmetic operation they affect the status bits um. So, that carry over flows that P s w bits are effected, but here we have got 2 variants of every instruction like ADD instruction, it has got 2 variant as we will see later.

One is add which will be affecting the status flags as well then there is another variant ADD S so, this ADD instruction will not affect the status flag, but ADD S instruction so, it will affect the status flag and this is useful because as soon as we have got conditional branching so, there may be difficult because you can say if conditional branching.

(Refer Slide Time: 26:30)



So, with that we are just the just if you look into previous instruction compare R1 comma R2 then GTADD R3, R4, R5. Now if this instruction is allowed to affect the status flag then when I am writing like LESUB R3, R4, R5 so, what I want is this flag that I want to check so, this should be the affect of this comparison and not this addition ok.

So, in other processors you know that whenever some arithmetic operation arithmetic logic operation is done the status flags will be set. So, as a result this GTADD instruction so, it will be setting this status flags as well, but we do not want that we do not want the status flag to be set by the GTADD instruction.

So, this is ensured by putting 2 variant of this ADD instruction. So, we have got we will see one is ADD another is ADD S. So, if you say ADD S then the status flags will be affected, if you do not say ADD S then the status flags will not be affected so, we have got 2 variants ADD and ADD S. So, this way we can have different features of many improved feature that have been incorporated into the ARM processor.