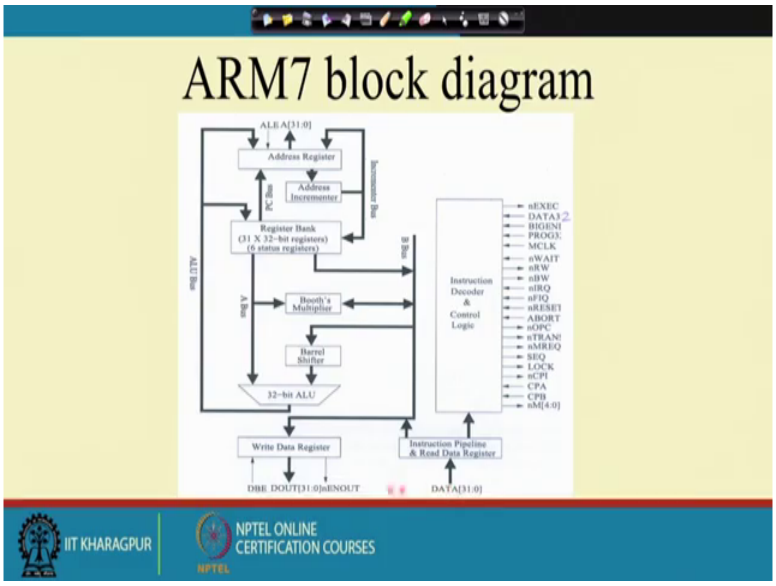
**Microprocessors and Microcontrollers**
**Prof. Santanu Chattopadhyay**
**Department of E & EC Engineering**
**Indian Institute of Technology, Kharagpur**

# Lecture – 42
# ARM (Contd.)

So, in ARM7 we have got a much more complex architecture compared to say 8051 or 8085.

(Refer Slide Time: 00:26)



## ARM7 block diagram

Ah, in the sense that it has got the data bus which is 32 bit and the address bus is also 32 bit and these there is there is no multiplexing of address and data bus, so, they are separate. So, address bus is 32 bit here and data bus is 32 bit here. Now, we have got internally there is a internal bus. So, one bus is the ALU bus which we call A bus and also it is A bus coming from the register bank and this ALU bus is coming out of the ALU and it is feeding the register bank as well as the address registers. So, sometimes we will like to take this register bank values from some of the values from some of the registers to do the operation. So, in that case this A bus will be utilized and for storing the result back; so, result may be stored back onto the register in this bank register or it may be onto some address register for some for further addressing.

And, we have got a separate incrementer. So, this is another important thing that you see here there is an address in incrementer. So, unlike 8051 where we have got the same

ALU which is doing the address incrementation after each instruction here that PC increment operation is done by this incremental module. So, it is totally separated from the other computational overhead of the processor. So, this ALU is responsible for doing the normal operations of the instructions and this incremental is for address incrementing.

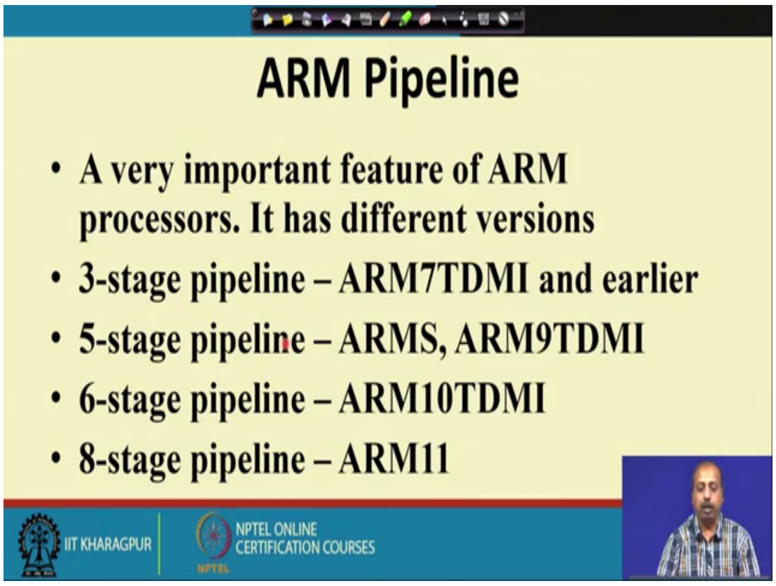
So, then its output can feed to some register bank or it might go to some address register. So, address register is actually the register which will hold the address which is visible to the outside world. So, whatever be the instruction if ultimately this memory has to be accessed the address register will contain the address of the location. So, if you are trying to write some data then this right data register will have the value in it that you want to write. So, after putting the address here and data here, so, you can say enable out. So, this enable out will be, so, this will enable this output on to the data bus.

Similarly, if you are trying to read something from the memory then this is coming to this instruction pipeline as well as read data register. So, we will explore this instruction pipeline part later. But, essentially the instruction when it is coming, so, in case of 8085 or 8051 we have got a single instruction register where the instruction opcode is coming, but here it is a more pipeline stage. So, there will be number of pipeline stages for the operation.

And, this read data register; so, this is if you are reading from data from the program memory so, it will be coming into this and internally you see that we have got a booths multiplier for doing the multiplication operation. So, for multiplication instruction, so, you have got booths multiplier. So, that, so, that way the ALU is relieved from doing that operation and we will see that multiplication operation is has also been optimized so that the it takes much less number of cycles compared to normal multiplication.

Then the barrel shifted is there. So, I have already said that you can shift the operand in a single in a single instruction before doing the arithmetic logic operation. So, you can do a shifting of the operand, so, multiplication by 2 or division by 2. So, multiplication and division by powers of 2, so, that can be done by this barrel shifter and many a times particularly in signal processing algorithms. So, we will find that there are many operations that we do our multiplication and divisions by powers of 2. So, those operations can be done very easily in this type of architecture.

(Refer Slide Time: 04:20)



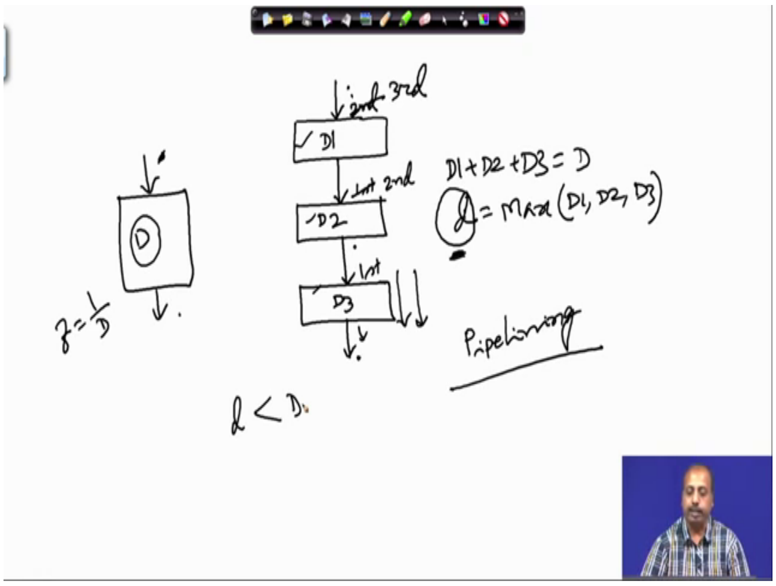
## ARM Pipeline

- A very important feature of ARM processors. It has different versions
- 3-stage pipeline – ARM7TDMI and earlier
- 5-stage pipeline – ARMS, ARM9TDMI
- 6-stage pipeline – ARM10TDMI
- 8-stage pipeline – ARM11

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, coming to the pipeline part; so, this ARM has got it has it is a pipelined architecture.

(Refer Slide Time: 04:39)



So, pipelined architecture, what do you mean by that, is that suppose we have got some functional block, ok. Suppose, we have got some functional block like this and this functional block it will have some delay. So, it has got a delay of say D unit. So, it has got a delay of D unit. Now, in a pipelined operation what is done is that this delay this module is divided into a number of sub modules, fine. So, I have got this first module.

So, so this is the say I have got this first module which is say of delay D 1 then the output of this stage goes to the next stage that has got a delay of D 2 and the third was a third output of that stage goes to the third stage that has got a delay of D 3. So, so the original block of delay D it is divided into 3 modules like this. So, previously we had a single module with a delay of D and the inputs were coming to that. So, if the delay is D. So, you can operate this module at a frequency f equal to 1 by D. So, after every D time unit. So, what is the input data when it is coming here? So, after D time unit the value will be available here.

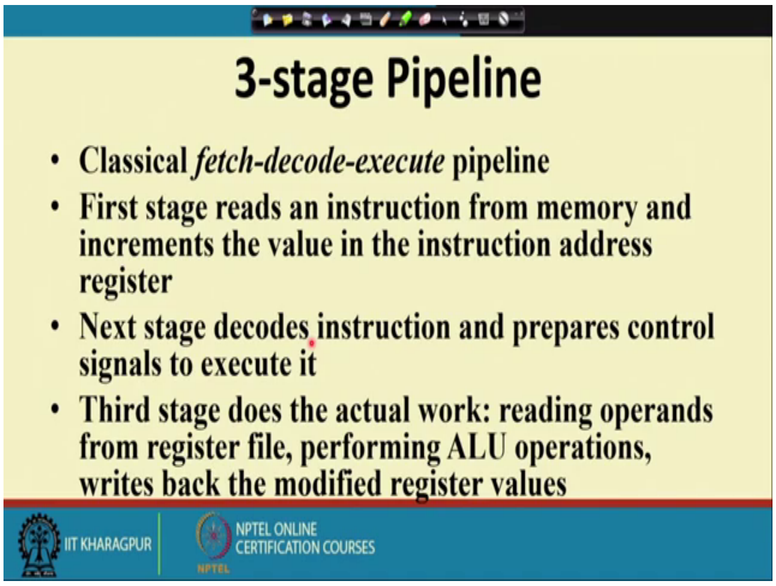
On the other hand; if I do an implementation like this then what happens is that this individual module delay, so, D 1, D 2 and D 3. So, the first data item that you have here. So, that will see a total delay of D 1 plus D 2 plus D 3 as the. So, that is equal to D. So, that it will see that amount of delay, but when this first data item has been has reached the second module. So, the second data item can be fed here and after some time the first data item will reach the third stage of processing. So, then this will be doing the second one and this will be doing the third data item.

That way if you have a continuous flow of data then you can see that after every after every small amount of time. So, this data will be outputted like say if this if is D 1. So, if suppose I have got this D which is the maximum of all these 3 values D 1, D 2 and D 3, then we can say that after every D time unit the next sample next output will be available here. Because, when the first data is coming out the second data is already here. So, second data will be coming out after the delay of D 3; similarly the third data will be coming out after another delay of D 3 like that. So, if D happens to be the maximum of these 3 stage delays then after every D time unit the data will be available at the output. So, this is the concept of pipelining.

So, this pipelining is used in different processors. We have already seen that fetch decode execute pipeline in instruction execution. In case of ARM processor, so, there that it has been made much it has been used much more and naturally since this original functionality D has been divided into 3 steps D 1, D 2 and D 3. So, D the small D value is often much less than the capital D value. So, that way I can do the processing at a much higher rate, ok.

So, so coming back to the discussion this has, so, ARM has got this a pipelining feature and there are various number of stages of pipelines that are available in different versions of the processor. Like ARM7 it has got 3-stage pipeline, ARMS and ARM9TDMI, so, they have got 5-stage pipeline, then ARM10 has got 6-stage pipeline, ARM11 has got 8-stage pipeline. So, slowly the number of pipeline stages have been increased and with the increasing number of pipeline stages the advantage is that we can we can have faster processing.

(Refer Slide Time: 08:52)



## 3-stage Pipeline

- Classical *fetch-decode-execute* pipeline
- First stage reads an instruction from memory and increments the value in the instruction address register
- Next stage decodes instruction and prepares control signals to execute it
- Third stage does the actual work: reading operands from register file, performing ALU operations, writes back the modified register values

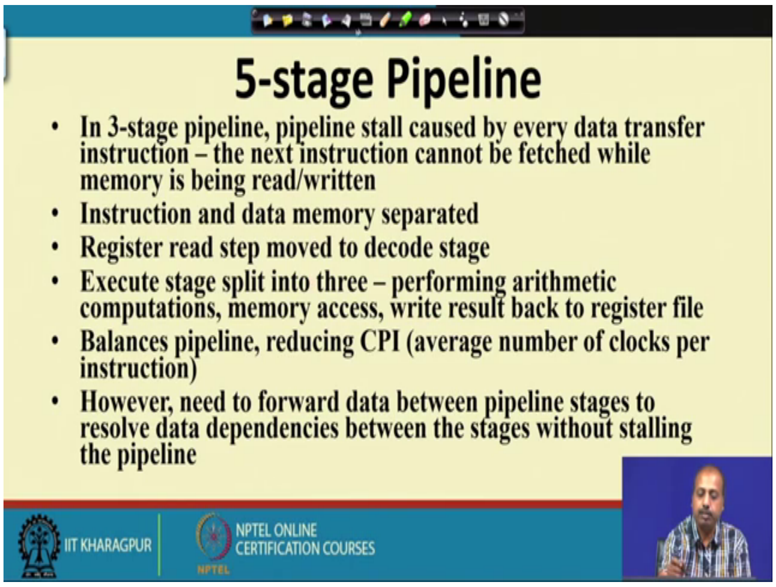
So, first one is the 3-stage pipeline. So, this is the classical fetch-decode-execute pipeline. So, it is same as other processors that we have. So, like 8085, 8051 all of them they do this fetch-decode-execute pipeline. So, first stage is the fetch state. So, this fetch stage is responsible for reading the content from the memory. So, it reads the instruction from the memory and it will increment the program counter or instruction address register. So, in case of ARM, so, is this program counter will call instruction address register. So, that value is incremented so that it can point to the next instruction.

The first stage is the fetch, second stage is that decode. So, decode will decode the instruction. So, it will understand; what is the meaning of the opcode? And it will prepare the control signals accordingly. So, if you look into this diagram, so, this after this instruction has reached here, so, it goes to this instruction decoder and control logic and it generates a number of control signals for various modules that we have here. Some of them are for are for some of them are will explain these remaining terms later, but this is they are coming to the they; the these signals will also control the operation of this control logic, but overall the instruction reaching here is decoded here and these control signals are generated for the remaining model for other modules that will do the operation.

So, the second stage you will do this decode and prepare the control signals, the third stage it does the actual work. So, what is the actual work? So, it has to read operands from the register file like. So, it may be that I want to add the content or two registers. So, those two register values will be I should be available to the ALU when the ALU will do the operation and then it will write back the value on to the value of this of the modified register.

So, this way we can do this operation ok, the operation can be carried out and in case of ARM, so, you will see that whenever we are doing an arithmetic logic operation so, the operands are always from the register bank. So, you cannot have it from external memory like that directly. So, that is why it is called it has it is this here we are mentioning explicitly that it is doing the operation on the register file.

(Refer Slide Time: 11:21)

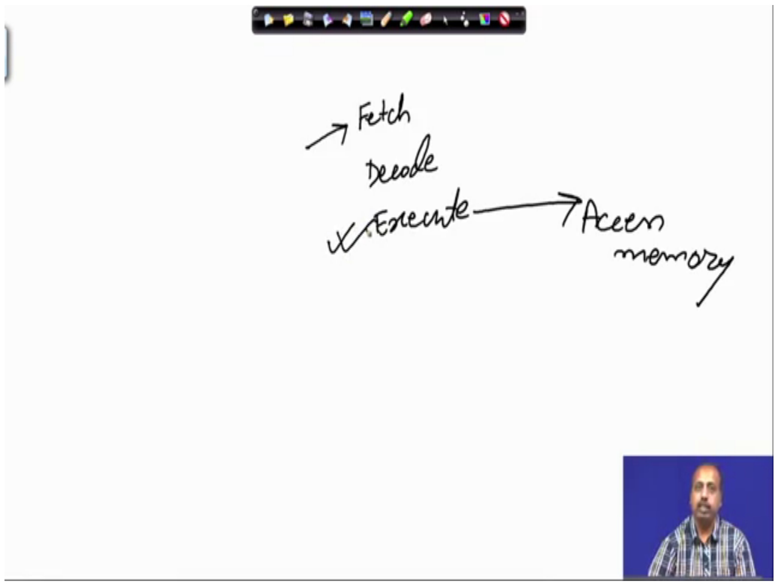


## 5-stage Pipeline

- In 3-stage pipeline, pipeline stall caused by every data transfer instruction – the next instruction cannot be fetched while memory is being read/written
- Instruction and data memory separated
- Register read step moved to decode stage
- Execute stage split into three – performing arithmetic computations, memory access, write result back to register file
- Balances pipeline, reducing CPI (average number of clocks per instruction)
- However, need to forward data between pipeline stages to resolve data dependencies between the stages without stalling the pipeline

So, 3-stage pipelining, the difficulty that we have is that if every data transfer instruction it will cause the pipeline stall.

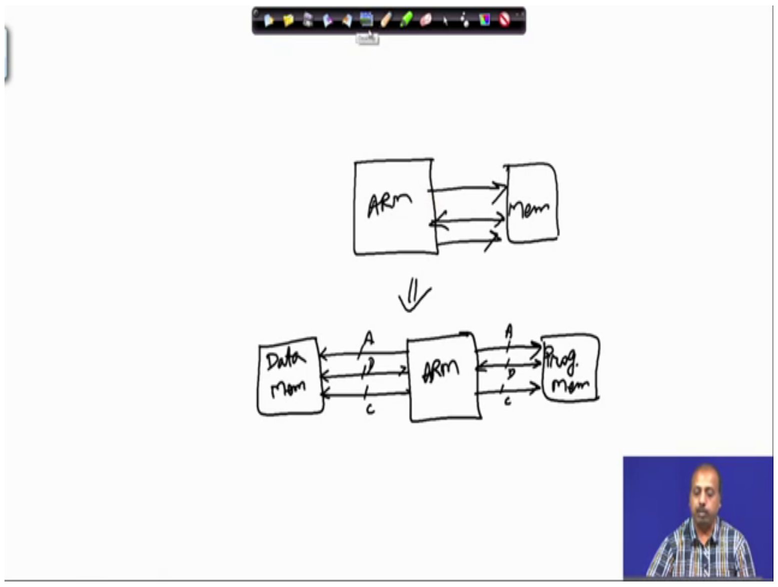
(Refer Slide Time: 11:48)



So, pipeline stall is something like this, that so, I have got this pipelining. So, if the resource is busy like say fetch, decode and execute. So, these three stages are there, now if this execute stage is also asking to access memory. So, this is also asking to access memory, like the instruction that we have may be to save the content of some register on to some memory location. So, that way this instruction will be using the memory bus. So, I cannot fetch the next instruction. So, there will be this fetching part will be stalled. So, this type of pipeline stall will occur.

And, if you look into the operations that we have outer out of these three stages to the execute stage it will take the maximum amount of time because it may be doing some complex operation. So, to, so, this is the problem. So, we have got this with every data transfer instruction. So, we will have a pipeline stall because the next instruction cannot be fetched while the memory is being read or written. So, how to solve this problem? So, this problem is solved by going to the Harvard architecture in which the program memory and the data memory they are separate. So, instruction memory and data memory are separated. So, we have got separate address bus and data bus for two types of memories.

(Refer Slide Time: 13:12)



So, if previously the structure was like this if this was the process of the ARM processor and we have got this as the memory, these are the memory. So, we have got this address bus data bus control bus like that, now in the Harvard architecture you know that it is diva it is made like this if this is the processor ARM processor now, we have got the program memory and the data memory they are separate. So, this is program memory and we have got the data memory which is separate.

Now, what will happen is that when the ARM is asking for program so it will access the program memory. So, it has got the address bus data bus and control bus for program for the program part and similarly, it will have address data and control bus for the data part for the data memory axis. So, this way we have got this the two memory they are separated and the buses are also separate so you have you have got the flexibility now that we can go for this these data axis will not hamper the fetching of the next instruction from memory. So, instruction and data memory are separated.

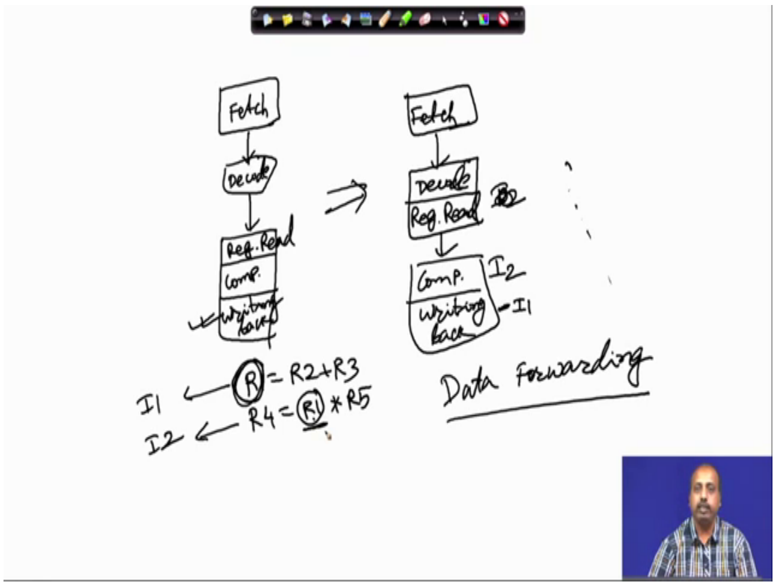
Second third thing that is done is that the next stage is to somehow make this operation of this the execute phase simpler. How do we do this? This register read step is moved to the decode stage. So, in the previous case the execute operation the first thing that it has to do is to get the operands from the register file to the ALU for the operation. So, this is simplified and now, we this register read is going to the decode stage. So, decode stage was previously less lightly loaded. Now the load has been increased to make the balancing between the stages.

So, execute stage is also split into three sub stages one for performing arithmetic computation, then memory access and then write result back to register file. So, these are the three stages into which this the execute stage is also divided. So, you have got this fetch is there decode and register it that constitutes one stage at this execute stage is divided into three stages. So, as a result we have got a 5-stage pipeline the here is three and decode one and fetch one. So, 5-stage pipeline.

So, this as it is if there is a balancing in the pipeline that we see now because the decode stage was very lightly loaded. So, that has been given enough jobs to do now. So, it reduces the CPI. So, CPI stands for Clocks per Instruction. So, how many clocks are needed per instructions. So, in a pipeline fashion, so, number of the instructions is executed in a pipelined way. So, you can you can execute more number of instruction ideally we would like to have one clock per instruction because if I have got one instruction taking say 10 stages and the architecture is a 10 stage pipelined architecture then after every clock cycle 1 instruction should be complete.

So, that way the ideally the CPI value should be 1, but it is not possible definitely it is not possible because of the different stage requirements of these instructions and the number of pipeline stages will not always match exactly with the size of the executor time of the instruction. So, that as a result it cannot be 1, but it is it will reduce the average number of average CPI value. However, it needs the; it leads to another problem which is known as data forwarding.

(Refer Slide Time: 17:17)



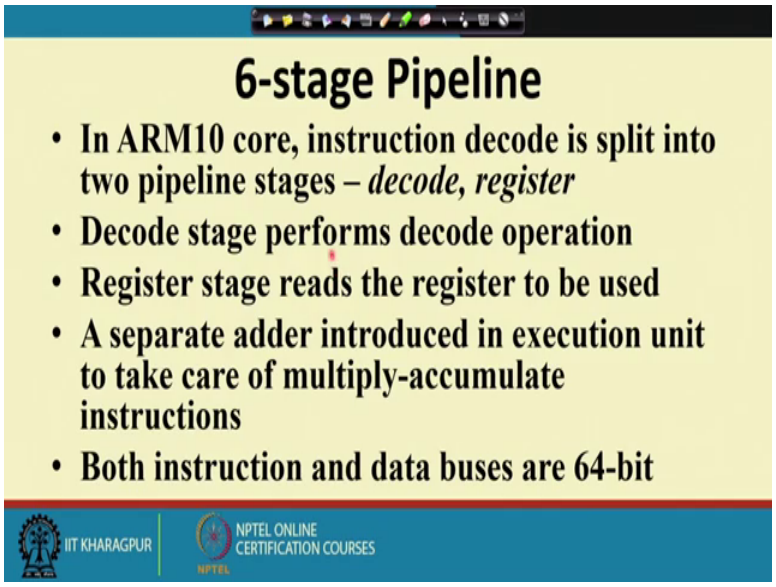
So, what is data forwarding? So, it is like this. So, what we have done is we had 3-stage pipeline this fetch, decode and execute. Fetch, decode and execute and this execute was 3 stages; the first stage was register read, first stage was register read and the next stage was the computation and the third stage was the writing back. So, these were the 3-stages. Now, what we have done in the modified version we have got this fetch this is there, but this decode is made complex. So, decode has got 2 stages now, decode and this register read, decode and register read, they are the 2 stages and then we have got this execute one that has got this computation and this writing back, these two are there.

Now, you can understand the problem because what can happen suppose I have got an instruction like R 1 say R 1 equal to R 2 plus R 3 and we have got an instruction like say R one equal to R 2 plus R 3 and the second instruction is say R 4 equal to R 1 into R 5. So, what is happening is that you see this R 1 value that is computing, so, it is available if this is instruction 1 and this is say instruction 2. So, when the instruction 1 comes to this stage or here when the instruction 1 comes to this phase then only the R 1 value will be updated, but when instruction was in 1 is in these stage instruction 2 is in the previous stage of the pipeline. So, what is happening is that the instruction 2 is actually not here a instruction 2 will be in the computation phase. So, by this time instruction 2 has already got the value of R 1. So, instruction 2 requires R 1 as one of the operand. So, by this time instruction 2 has already accessed R 1 and, but that value is not correct, because in this program what we mean is that whatever be the output of summation of R 2 and R 3. So, that should be the content of R 1. So, here I am getting some older content of R 1.

So, we have to have some sort of scheme which is known as data forwarding. So, what happens is that when the compiler will see that there is a situation like this that some operand value that is being used which is not yet stabilized. So, it is that the destination does not have it is value yet. So, between the stages there should be some way by which this data is forwarded. So, this ALU output R 1 should somehow be fade as the operand for the next instruction. So, the architecture supports this facility and it is the compilers job to identify those situations and accordingly mark that the data forwarding should take place at those places and then the architecture will be the control words will be generated. So, that it is taken care of.

So, that is the 5 stage that is about the 5-stage pipelining. So, data forwarding may be necessary between the pipeline stages to resolve data dependencies. So, whenever between stages we have got data dependency, so, either we have to stall the pipeline. So, that the previous instruction is over then only I go into the fetching of the next instruction or executing the register read of the next instruction that. So, for that much time we have to wait that is pipeline stall or we have to do some sort of data forwarding. So, both are possible.

(Refer Slide Time: 21:28)



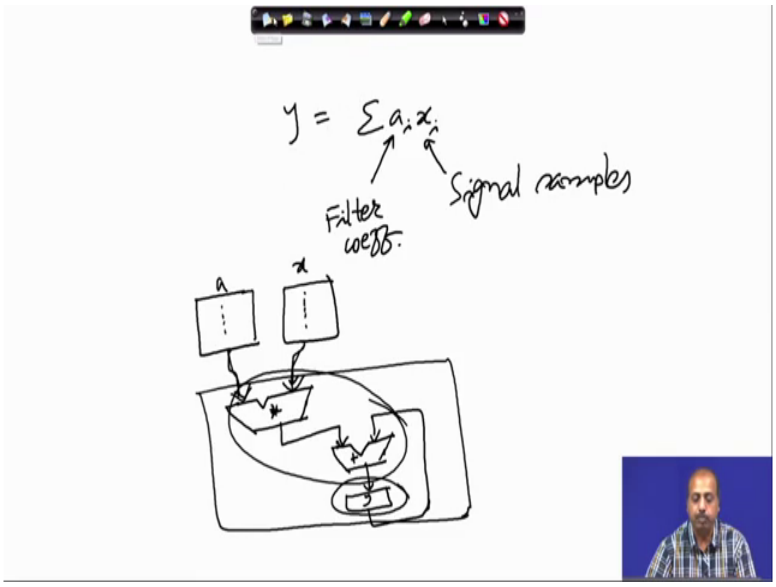
## 6-stage Pipeline

- In ARM10 core, instruction decode is split into two pipeline stages – *decode, register*
- Decode stage performs decode operation
- Register stage reads the register to be used
- A separate adder introduced in execution unit to take care of multiply-accumulate instructions
- Both instruction and data buses are 64-bit

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

So, next we will look into sixth stage pipeline. So, ARM10 has got 6-stage pipeline. Here the decode is a split into two pipeline stages as earlier to decode and register and decode stage performs. A decode operation and this register read will perform the register at the reading the register operation and, but now a separate adder is introduced in the execution unit to take care of this multiply accumulate type of instructions. So, instruction decode previously the instruction decode was done in one stage consists doing both decode and register read together. Now, it is divided into two stages. So, you have got this decode and registered it may put together put separately into two stages. So, you get some sixth stage pipeline and also so, there separate adder has been introduced for multiply accumulate type of instruction.

(Refer Slide Time: 22:41)



So, multiply accumulate is something like this. So, multiply accumulate will be doing like this. So, suppose I have got an operation like this. So, y equals to sigma a i x i. So, this is a very common operation in case of signal processing where these x i's are the signal samples this may be signal samples and this a i is are the filter coefficients. So, this is very common. So, whenever we are doing any computation in signal processing so, this is the filtering operations.

So, they often do like this now you see that if you want to write it in a if you want to do it by means of a hardware then what is required is you should have this register which will hold these a values. So, these are the a values you should have another bank of locations where it will have the x values and then the successive values should be fed to this adder they will be fed to the adder and a i and x i that will do that multiplication sorry this is a multiplier. So, this a i x i will be multiplied and then it should go to an adder where it will be doing the addition part this is the adder. So, this is the y register and this value should come here and we should have a structure like this. So, this previous value will be added and it will be doing like this.

Now, if you do not have this particular structure available in the within the processor then what you have to do is you have to write a program where it will be for doing this intermediary calculations and then it will be adding it with y. So, that way it has to be in a program we have to do that. So, if I have a single ALU for doing this operation then we have to use that ALU for doing the multiplication and then addition then again multiplication then again addition like that single ALU has to be used.

So, in case of ARM processor what has been done is one such module has been incorporated into the system. So, this is called multiply accumulate unit. So, it will multiply the successive data items coming from a i and x i and accumulate the values in the y register. So, in the internally so, we have called this hardware available. So, that by a single instruction we can do this multiply accumulate type of operation.

So, we will see that in more detail the actual instruction when we can go to the instruction set of the processor. So, this multiply accumulate is a done by a separate execution unit that takes care of this multiplier accumulator operation. Then we have the instruction and data buses are now 64 bit ok. So, now, this is the ARM10. So, it has got 64 bit address bar instruction and data buses. So, instead of being 32 bit data bus so, you have got 64 bit data bus, but internal operation remains as 32 bit only. The advantage we get is the accessing in the external memory. So, in one axis you can get two instructions now. So, two because, each instruction is 32 bit; so, you get two instructions and data bus is also 64 bit, so, in one axis so you can get two data items.

So, that a number of accesses to the outside data memory and program memory will be less and that will help in reducing the complexity in terms of this power consumption of that you of the system because bus activity will lead to bus power consumption and that will lead to at least the heating effect. So, the heat of the set temperature of the system may be high. So, as we said that ARM is there were designed to have low power. So, this is another avenue by which this power reduction is achieved in ARM10 processor.