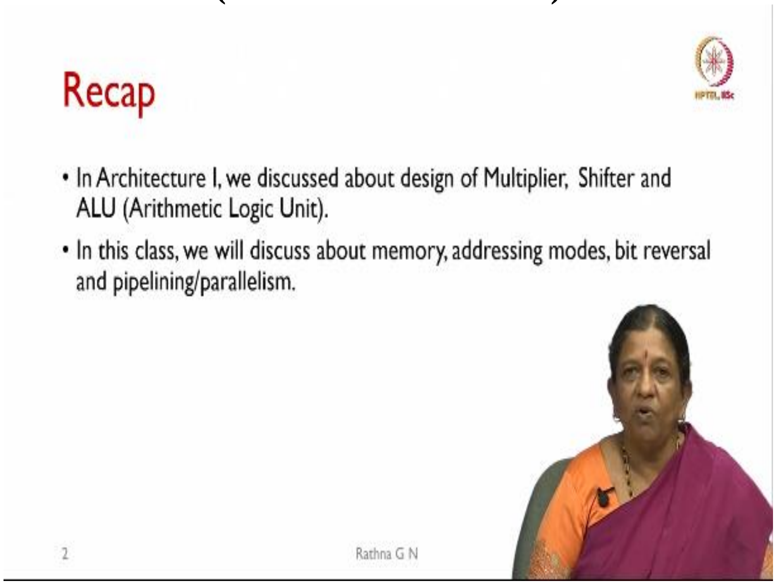
Real - Time Digital Signal Processing

Prof. Rathna G N
Department of Electrical Engineering
Indian Institute of Science - Bengaluru

Lecture - 06
W0IU5 DSP Architecture - II

So, last class we discussed about the DSP architecture 1. So, today we will see what how we are going to continue the architecture part 2.

(Refer Slide Time: 00:31)



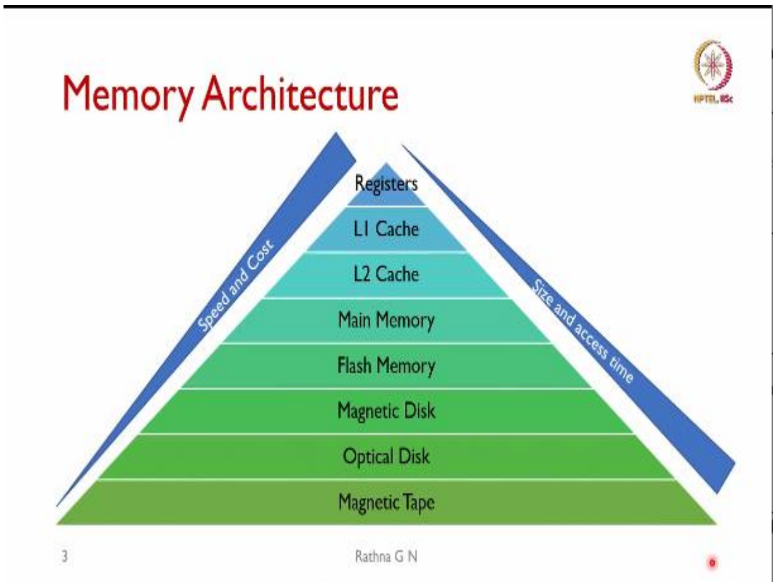
Recap

- In Architecture I, we discussed about design of Multiplier, Shifter and ALU (Arithmetic Logic Unit).
- In this class, we will discuss about memory, addressing modes, bit reversal and pipelining/parallelism.

2 Rathna G N

So, as a recap to refresh you, what we discussed in the last class. So, we discussed about the multiplier, how to design a parallel Braun multiplier, and how to design a shifter, barrel shifter what we discussed and then how to design arithmetic logic unit what we have considered. In this class, we will discuss about memory and then what are the addressing modes compared to our regular microprocessor what we need it and then why we need the bit reversal and pipelining and then parallelism of aspects of DSP processor.

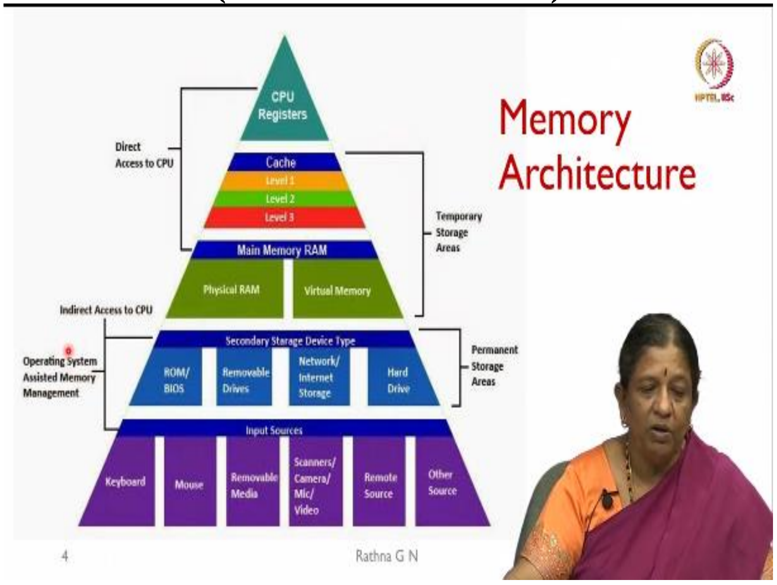
(Refer Slide Time: 01:13)



So, coming to the memory architecture so, you will see a pyramid here. So, registers are the closest to the CPU. So, and then what we have is L1 cache, L2 cache and then you will be seeing the main memory and then flash memories and then for more storage, magnetic disk, optical disk and then tape comes in the last part of it, all of us know that these are almost extinct from the present situation.

So, .seeing the left hand side of it. So, you will be seeing that speed is the highest in this case, and it is going to slow down when as we go down in this pyramid. Whereas, in the case of size and then access time as we will be seeing it sizes very small in this and access time for these registers are the smallest one whereas, for the magnetic tape is the last one sizes as we know a lot of it what we can store it and then access time is going to increase.

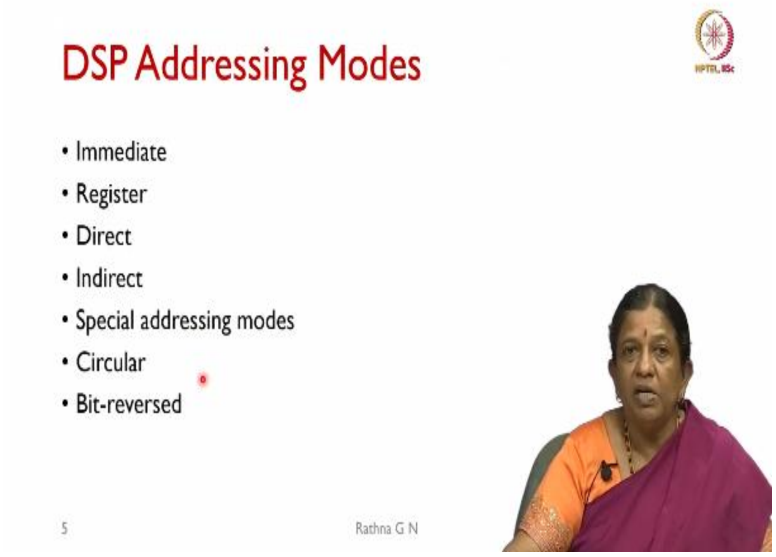
(Refer Slide Time: 02:22)



So, continuing with the memory architecture. So, how it is going to be stored in our DSP processor or how it is designed, what we will be seeing it. So, you will be seeing that CPU will be incorporated with the registers and then you will be seeing that cache for level 1, 2 and then 3. So, these are the direct CPU access what we are going to have it and then even some of the temporary storage is access what you will be seeing it, that is physical RAM virtual memory, and the other ones are it is going to be indirect access to CPU.

So, they do not have any direct bus connectivity in the thing, whereas we have to use external bus to access these memories basically. So, you will be seeing some of them are going to have an overlapping spectrum basically. So, these are the assisted memory management what we call them. So, these we call it as secondary storage devices, and then we have the input sources here. So, you will be seeing that these are the permanent storage areas and these are the temporary storage for the processor.

(Refer Slide Time: 03:48)



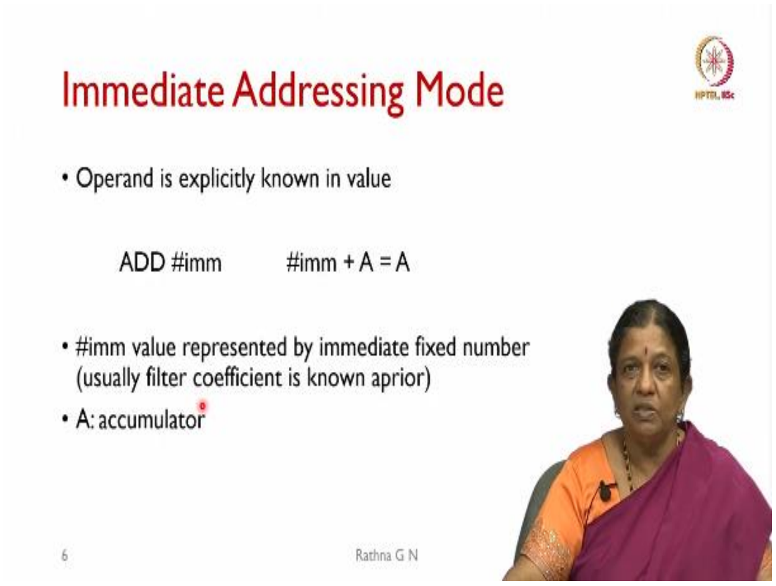
DSP Addressing Modes

- Immediate
- Register
- Direct
- Indirect
- Special addressing modes
- Circular
- Bit-reversed

5 Rathna G N

So, coming to some of the addressing modes you would have learned in your 885 or 886 course. So, we have to provide immediate addressing mode, register, direct, indirect, special addressing modes we will be seeing it as circular and bit reversed. Addressing modes are the typical to DSP processes.

(Refer Slide Time: 04:11)



Immediate Addressing Mode

- Operand is explicitly known in value

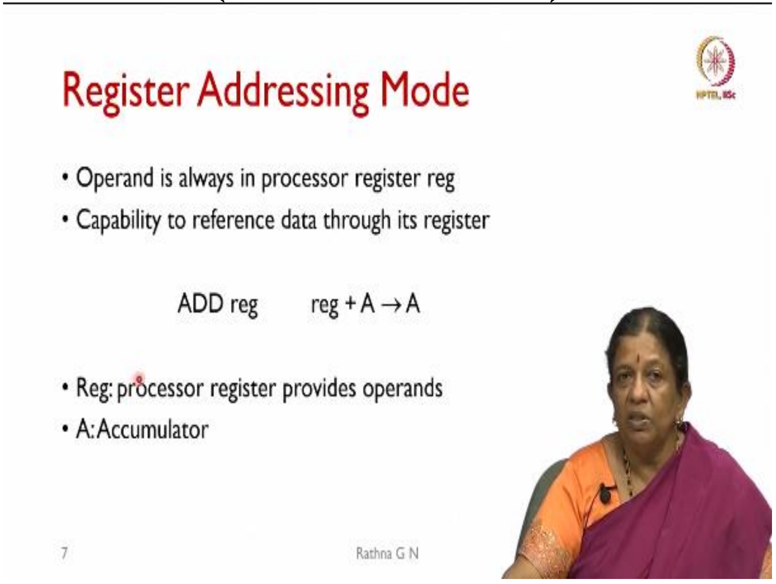
ADD #imm   #imm + A = A

- #imm value represented by immediate fixed number (usually filter coefficient is known aprior)
- A: accumulator

6   Rathna G N

So, when you come to the immediate addressing mode, all of us know that when I want to add directly some value from the memory, we will be giving it as ADD hash immediate address what we will be giving it or immediate value. So, this value is going to be added to our accumulator. And the result is going to be stored in the accumulator. That is, what it says hash immediate is the value represented by immediate fixed number. So, usually if we have to have the filter coefficient, so if we know aprior, then we can give this as an immediate value and then as normal notation A is the accumulator in this.

(Refer Slide Time: 04:53)



Register Addressing Mode

- Operand is always in processor register reg
- Capability to reference data through its register

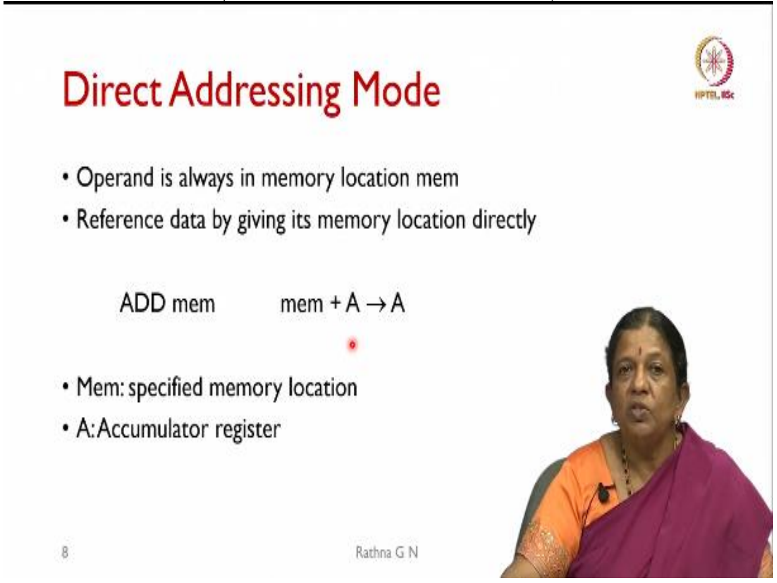
ADD reg $reg + A \rightarrow A$

- Reg: processor register provides operands
- A:Accumulator

7   Rathna G N

Coming to register addressing mode, we say operand is always in processor register We call it as reg and capability to reference data through its register, we call it as ADD register means, the value whatever stored in the register is going to be added with the accumulator and result in accumulator. So, these reg are the processor register, which provides the operands for our addition.

(Refer Slide Time: 05:23)



Direct Addressing Mode

- Operand is always in memory location mem
- Reference data by giving its memory location directly

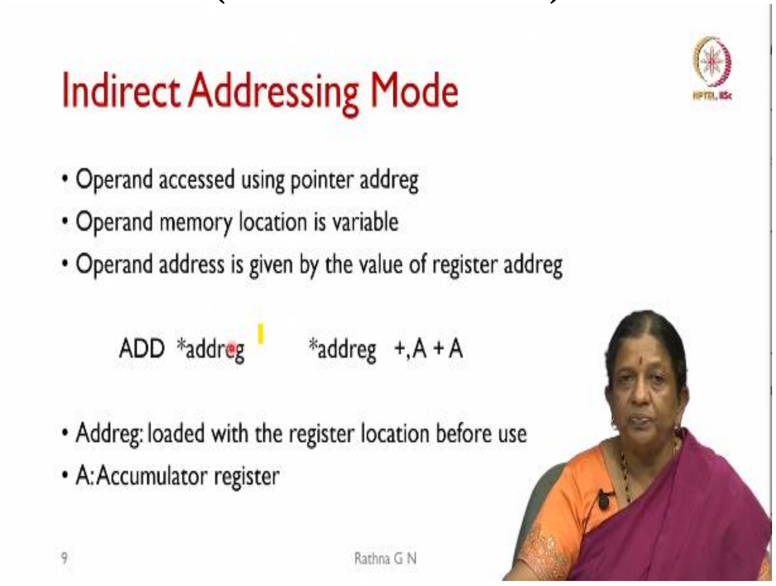
ADD mem   mem + A $\rightarrow$ A

- Mem: specified memory location
- A: Accumulator register

8   Rathna G N

So, coming to the next one, we have to have a direct addressing mode, we have to take a data directly from the memory location, we call it as mem. So, it is going to be reference data by giving its memory location directly. So, we say add memory so, we are giving the address of it whatever the value stored in it, that is mem is going to be added with our accumulator and result is in the accumulator, this is a specified memory location m e m.

(Refer Slide Time: 05:54)



Indirect Addressing Mode

- Operand accessed using pointer addreg
- Operand memory location is variable
- Operand address is given by the value of register addreg

ADD *addreg   *addreg +,A + A

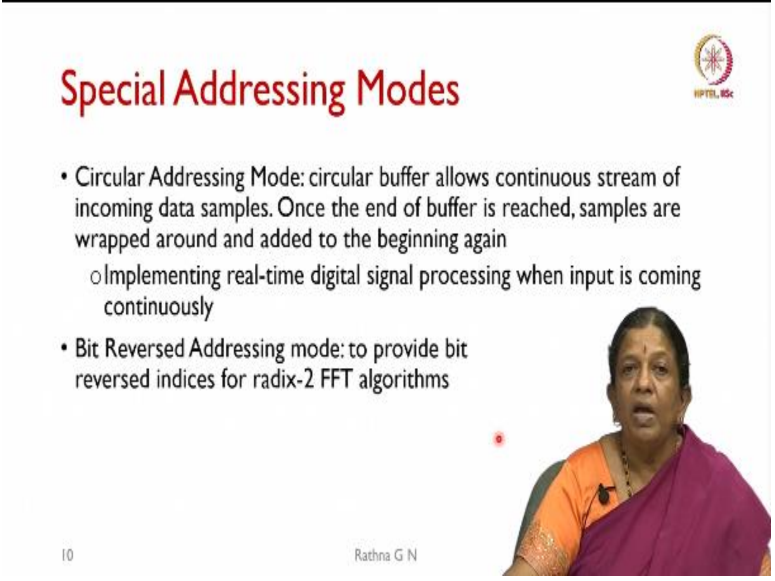
- Addreg: loaded with the register location before use
- A: Accumulator register

9   Rathna G N

And in the indirect addressing mode, we have to say that operand access is using pointers add register, because all of our operations are continuous that is sigma what we call it the from 0 to n minus 1 times what we want to multiply and add for that we should have a pointer so that I can access one after the other. So, operand memory location is the variable in this case and operand address is given by the value of register addreg in this case.

So, you will be specifying add star gives from where the memory location where it is available the data address register. So, the value provided by this whatever pointer it is register is pointing to the value is taken from that memory and then it is added with the accumulator and result is an accumulator back. So, that is what it says add register loaded with register location before use.

(Refer Slide Time: 07:00)



Special Addressing Modes

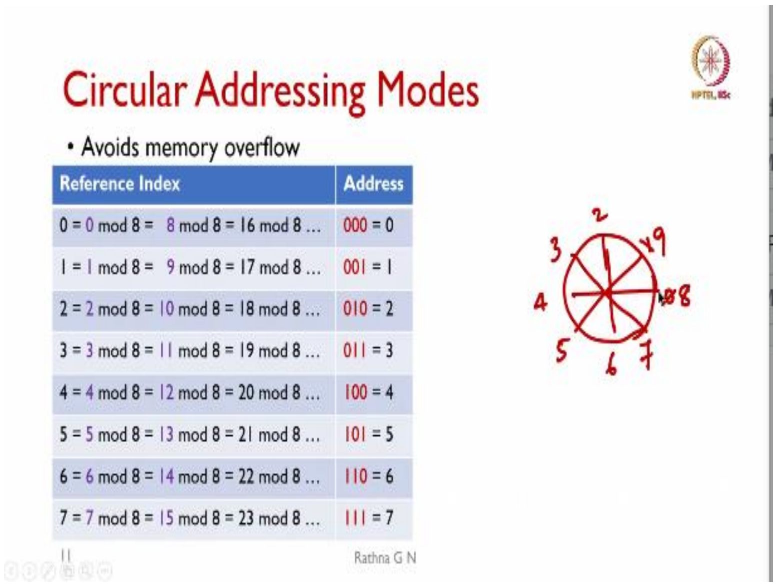
- Circular Addressing Mode: circular buffer allows continuous stream of incoming data samples. Once the end of buffer is reached, samples are wrapped around and added to the beginning again
  - Implementing real-time digital signal processing when input is coming continuously
- Bit Reversed Addressing mode: to provide bit reversed indices for radix-2 FFT algorithms

10 Rathna G N

And coming to special addressing modes. As I am telling these are the ones which classifies digital signal processor apart from normal processors, the first one is a circular addressing mode. So, here circular buffer allows continuous stream of incoming data samples. So, once the end of buffer is reached, samples are wrapped around and added to the beginning again. So, this is required for real time implementation in DSP processor. So, we know that the input is coming continuously.

The other addressing mode, what we need is the bit reversed addressing all of you who have taken the digital signal processing course will be knowing it, that for the FFT algorithm, if I am using the radix 2 FFT, I need the input in that reversed format.

(Refer Slide Time: 07:59)



Circular Addressing Modes

- Avoids memory overflow

Reference Index	Address
0 = 0 mod 8 = 8 mod 8 = 16 mod 8 ...	000 = 0
1 = 1 mod 8 = 9 mod 8 = 17 mod 8 ...	001 = 1
2 = 2 mod 8 = 10 mod 8 = 18 mod 8 ...	010 = 2
3 = 3 mod 8 = 11 mod 8 = 19 mod 8 ...	011 = 3
4 = 4 mod 8 = 12 mod 8 = 20 mod 8 ...	100 = 4
5 = 5 mod 8 = 13 mod 8 = 21 mod 8 ...	101 = 5
6 = 6 mod 8 = 14 mod 8 = 22 mod 8 ...	110 = 6
7 = 7 mod 8 = 15 mod 8 = 23 mod 8 ...	111 = 7

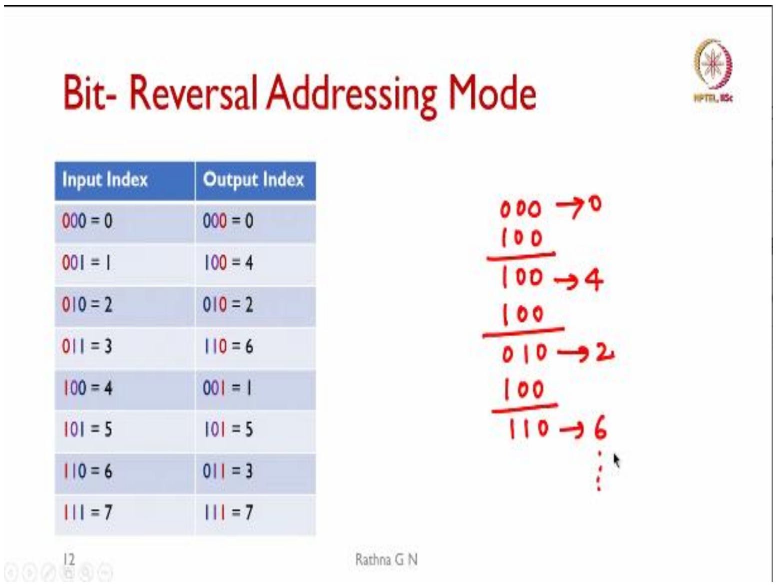
11 Rathna G N

So, first, we will see what is the circular addressing mode. So, in this case, this is the reference index, the value what I have taken here is 8 basically, that means to say that there are 8 values what we can store, if I put the circular address, so I will call it as what we have bifurcated into 8, the circle into 8 parts, so I will be seeing that this is the 0 and then this is 1, 2, 3, 4, 5, 6 and 7. So, these are the values what I needed to store them and when the 8 sample comes, what we are going to do it?

So, we have taken as it is seen here. So, we are taking 8 mod 8 is going to be 0. So, the 8 sample whichever is coming in the real time, which over writes this 0th location as the 8th sample. So, what is going to happen to the 9th sample? So, we will be overwriting on one 9, this is what it is shown with the mod values how we will be storing it only in the 7 location. So, why we need the circular addressing all of us know that we are working for the real time signal which is coming continuously.

If I start storing in the memory it is going to overflow. So, I will not have any memory to store the thing. So, how much I need it we will consider it when I take up the filters class basically. So, in this case, we have taken only 8 values are sufficient for our operations. Then once the next sample comes 8th one, so we will be discarding the first, whichever is the last sample and then we will be overwriting on it so that way we will be saving my memory in overflow.

(Refer Slide Time: 10:02)



Bit- Reversal Addressing Mode

Input Index	Output Index
000 = 0	000 = 0
001 = 1	100 = 4
010 = 2	010 = 2
011 = 3	110 = 6
100 = 4	001 = 1
101 = 5	101 = 5
110 = 6	011 = 3
111 = 7	111 = 7

$$\begin{array}{l} 000 \rightarrow 0 \\ \underline{100} \\ 100 \rightarrow 4 \\ \underline{100} \\ 010 \rightarrow 2 \\ \underline{100} \\ 110 \rightarrow 6 \\ \vdots \end{array}$$

12

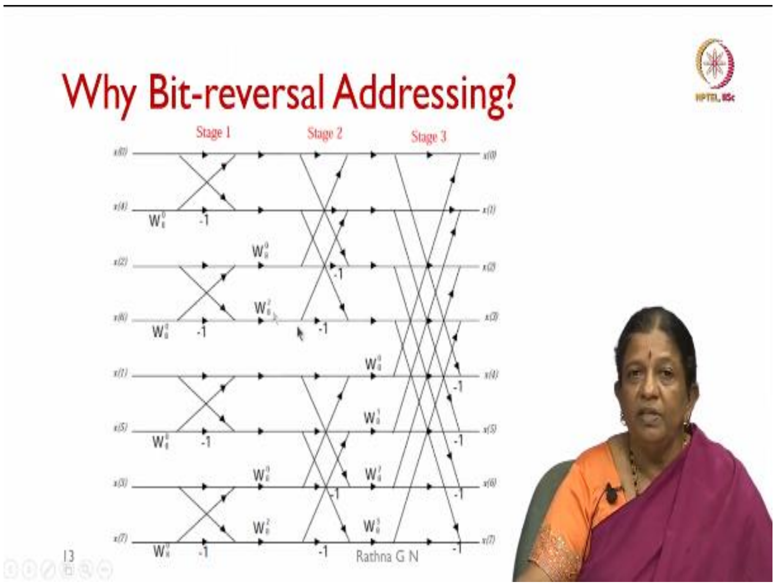
Rathna G N

So, coming to the next is why do we need bit reversal addressing? Just now, I said the thing how we are going to do that or how it is going to be represented? So, if we know that this is the input index what I want it, but output I want it in this order. So, as we know that if we do it in software, it will take multiple clock cycles, instead of that, can we do the bit reversal in the hardware.

So, as you can see in the right hand side. So, from 0 the length of the number of samples what I want capital N is equal to here also 8. So, we take it as 8 by 2 which is 4. So, number 4 in binary is 1 0 0. So, we add 1 0 0 to 000 so, you will be seeing there is no difference in this addition 2 whatever we consider example for fixed point addition. So, we get next number is 0, 4 as you can see this is the 4, now next I have to add the same this 1 0 0 to this number.

How I am going to add instead of adding from right to left for generating this bit reversal, this adder is going to do left to right. So, when I add 1 + 1 I will get 0 and I will have a carry here, which will be taken to the next stage. So, which becomes 0 1 0 so, which is nothing but 2 that is what it is listed in the table here. So, how we get the next one so, I add 1 0 0 so, this is a normal, so, it will be 1 1 0 which is 6. So, like this we continue and then these are the numbers what it has to be input to my FFT algorithm.

(Refer Slide Time: 11:58)

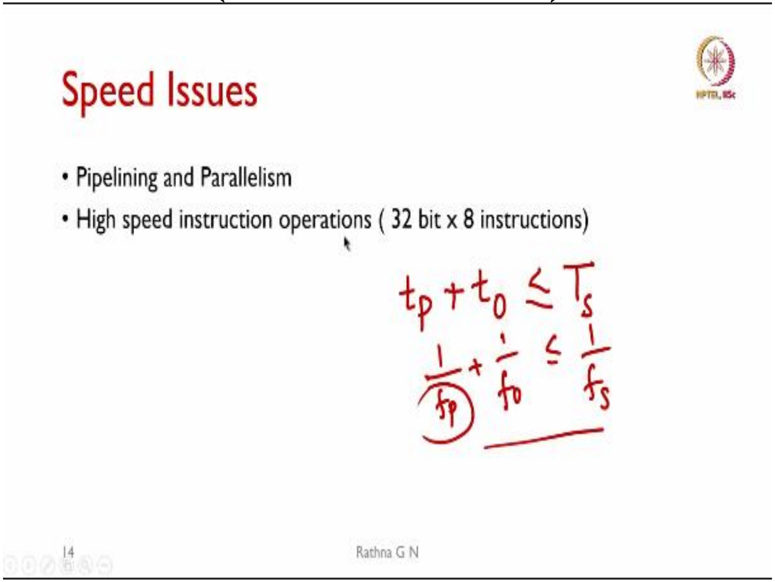


Why Bit-reversal Addressing?

13

To show that why we; need the bit reversal it is shown in this figure. So, what we have is x of 0 is the input and x(4), x(2), x(6), x(1), x(5) and then x(3) and then x(7). So, the output is going to be in order. So, always input is in the bit reversed, output will be in order. So, if I give input in order output is going to be in bit reversed order, what I will be getting it. So, this is what the thing the detail of butterfly structure what we call it for 8 point FFT. So, which has 3 stages that detail of the design, and then how we are going to implement it in real time we will discuss when I take up FFT lecture.

(Refer Slide Time: 12:51)



Speed Issues

- Pipelining and Parallelism
- High speed instruction operations (32 bit x 8 instructions)

$$t_p + t_0 \le T_s$$

$$\frac{1}{\boxed{f_p}} + \frac{1}{f_0} \le \frac{1}{f_s}$$

14

Rathna G N

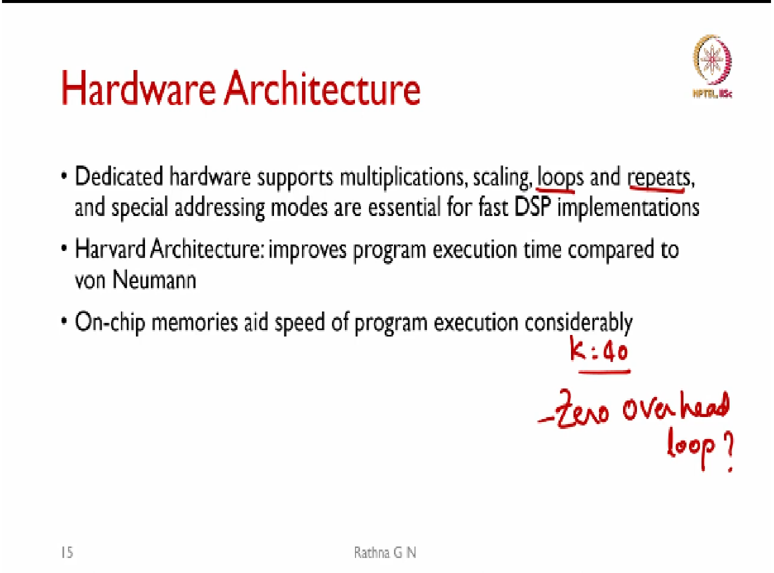
So, coming to the thing, next is what we have to worry about the speed issues. So, in this case, what we call it is why I need the speed part of it, I know that t_p is my processing type for the any processor and then I had to allow for some I O operations we know that we have to get input and then even the output has to go out. So, which we call it as t_o is the time. So, this should be less than or equal to the sampling time of what we will call it as T_s of the processor.

So, when we talk about in terms of frequency, I can call it as $\frac{1}{f_p} + \frac{1}{f_0} \leq \frac{1}{f_s}$. So, that this is going to decide my; what should be my input clock rate, what I can feed it into my processor or how I am going to select the processor if you are designing your own processor, this has to be considered depending on the application. Here we are considering only the algorithm in this case that is processing time of one of the algorithm.

If it has multiple thing you have to take the longest one which is going to take depending on it what your clock frequency has to be designed. So, if this is not going to be met, then can I use pipelining and then parallelism. So, that is we will be seeing in this case high speed instruction operations also, one is in that data what we will be considering it as pipelining and parallelism. How about in the instruction?

So, I can have a high speed instruction operation as in the 6x processors, we have 32 bit 8 instructions, I can fetch it simultaneously. So, this is one of the speed at which I can get instruction in the DSP processor.

(Refer Slide Time: 14:54)



Hardware Architecture

- Dedicated hardware supports multiplications, scaling, loops and repeats, and special addressing modes are essential for fast DSP implementations
- Harvard Architecture: improves program execution time compared to von Neumann
- On-chip memories aid speed of program execution considerably

k:40

-Zero overhead loop ?

15 Rathna G N

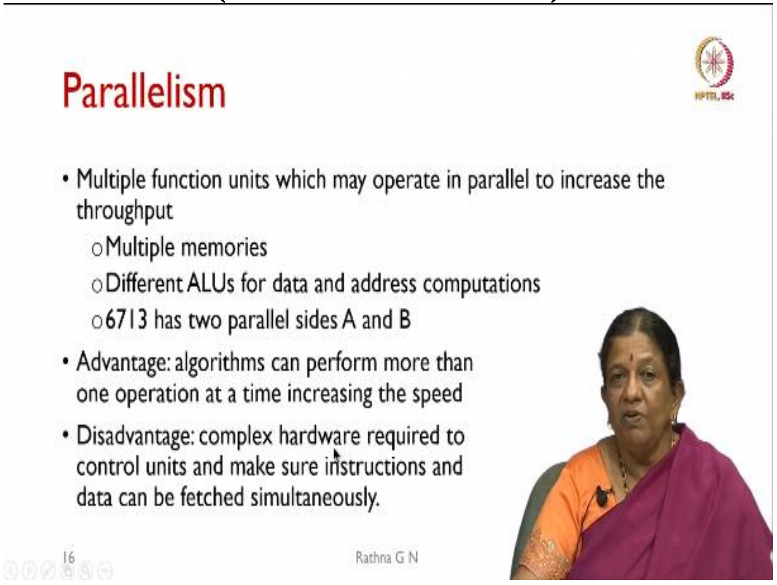
So, coming to hardware architecture so, we said that we have a design dedicated hardware supports multiplication scaling, loops and then repeats. Something here, I want to tell that what is how we are going to avoid the loop, why we need the repeats. Most of the cases, instead of loops, we want to have the repeats, what we call it as in the DSP processor as zero overhead loop. So, what is this? If I define my variable with hash assign or anything which we will discuss in the lab thing how it has to be done.

If I declare that some value, I know that what is the loop is going to run actually, so if I call it as 40, and then I will be repeating the instruction 40 times so I need not how to spend that whether decrement everything is going to happen in the background and then I need not have to spend any time on the loop to come back. So, for loop can be avoided this way. So, the other ones we said we have using the special addressing modes.

So, for these fast DSP applications, the other architecture in the last class, we said that we are going to use the Harvard architecture, which is going to improve our execution time compared to von Neumann architecture and even on chip memories aid speed of program execution considerably. So, that is the reason why I was telling in the last class that why my intermediate results has to be in the register or stored in the accumulator. So, that will not be spending, because of your memory structure.

Access in the external memory is going to cost me your number of clock cycles, which I want to avoid so that I can increase my speed of operation. This is how we will be designing our architecture.

(Refer Slide Time: 16:55)



Parallelism

- Multiple function units which may operate in parallel to increase the throughput
  - Multiple memories
  - Different ALUs for data and address computations
  - 6713 has two parallel sides A and B
- Advantage: algorithms can perform more than one operation at a time increasing the speed
- Disadvantage: complex hardware required to control units and make sure instructions and data can be fetched simultaneously.

16 Rathna G N

Now, if this is not possible, can we have the parallelism built into this? That is, is there any data dependency in my algorithm? I am going to check is if there are no dependencies, then I can have multiple function units. So, which may operate in parallel to increase my throughput. So, then I need multiple memories and we need different ALUs for these operations. And even the data and addressing computations have to be done differently for these ones.

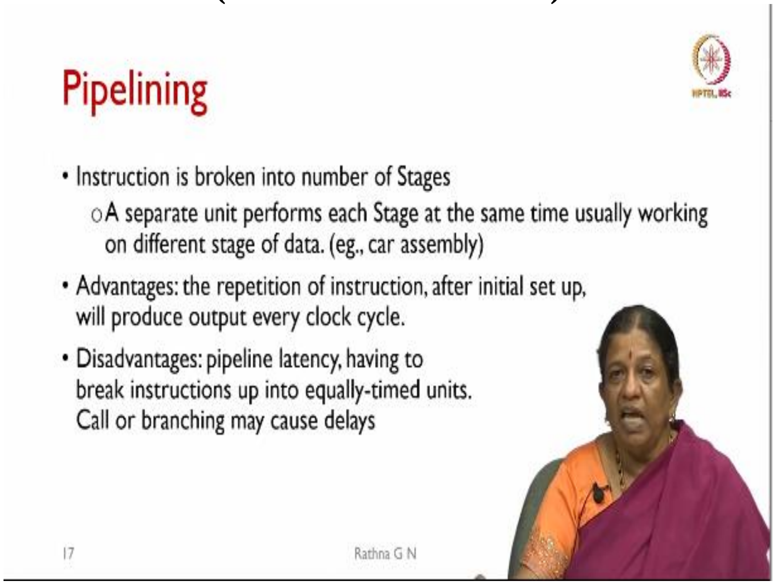
As an example, whatever DSK board we are using in this course DSK 6713 has 2 parallel sides. That is we call it as A and B, which have 4 functional units on both the sides, that means to say 2 CPUs are going to run in parallel. So, the advantage of using this parallelism is algorithms can perform more than 1 operation at a time, that how we can increase the speed.

The disadvantage of this is we need a complex hardware required to control units because which side you are working and then how we are going to transfer the data from one side to the other. And some of the issues like you will be branching, or call and then pop operations. Most of these DSP processor does not have call and then return because we do not know where which side of the processor they will be working.

We will be controlling through the branch operations. Those who are interested can go into a hex coding and other things if time permits, I will be presenting one of the examples in the lab. So, how we can decide what is the thing happening in both the sides of the CPU otherwise most of the examples will be in C code and then how we will be extracting the parallelism is going to depend on how we will be configuring our compiler.

Next is disadvantage of this is as we are mentioning it and then we have to make sure that instructions data can be fetched simultaneously, otherwise one of them is going to slow down. So, we will be landing in that whatever we are thinking that we will be getting twice that of that 1 clock cycle. So, we may not achieve so one of them delays the thing.

(Refer Slide Time: 19:26)



Pipelining

- Instruction is broken into number of Stages
  - A separate unit performs each Stage at the same time usually working on different stage of data. (eg., car assembly)
- Advantages: the repetition of instruction, after initial set up, will produce output every clock cycle.
- Disadvantages: pipeline latency, having to break instructions up into equally-timed units. Call or branching may cause delays

17 Rathna G N

Coming to pipelining, so, all of us know that a water pipe is an example I can take it we know that when the water flows when you have installed your pipe for the water line, so it is empty. Once you open the tap, it takes some time for you to get the water in your tap. That is the delay what it is going to be or one more example is going to be car assembly what you can take it different units are working at different parts of a car and once everything is done, so finally it will come for the assembly.

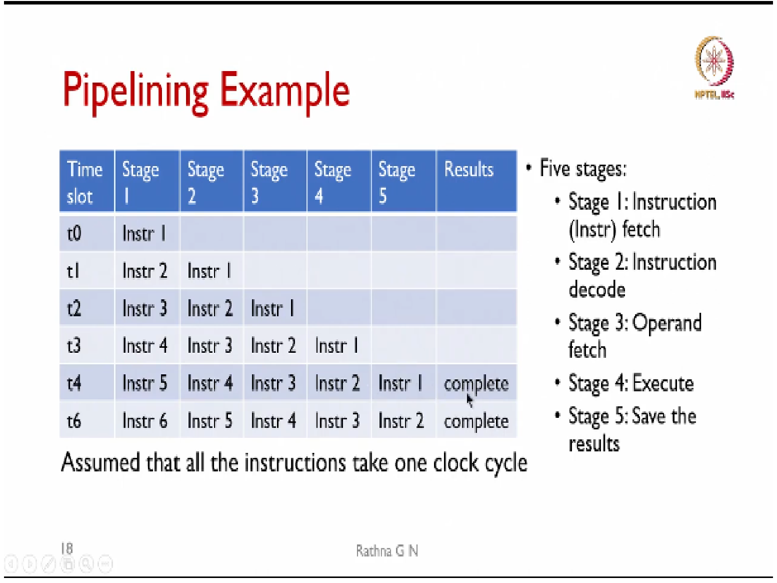
So, that is what it says separate unit performs each stage at the same time usually working on different stage of data. So, advantage of this pipelining is the repetition of instruction after initial setup, will produce output, every clock cycle, this one we call it as latency. So, the first output will depend on how many clock cycles it takes to complete it. After that we will be getting it output every clock cycle.

The disadvantage part of it is pipeline latency first one is that because we may have to wait for first car assembly or first water to get it in the pipe or tap, the pipe has to be filled in this whereas in the car assembly, the complete unit has to finish it then it has to come out. That is the longest delay what we have to wait what we will call it, just like in multiplier, we said that what was the longest path delay and then in the big break instruction up into equally timed units.

If it is so, then we will be arriving at the same time as I was just now mentioning call or branching may cause delays if because depends on what is the length of pipe what we have taken it. So, I want to clean it up then even the water pipe I have to completely empty the pipe and then start different water if it is become dirty or whatever maybe, even in the car assembly I have done 1 car design.

And then it is processes going on if there is 1 car which fails or something like that then everything has to be stopped and then restart the car building. So, these are the delays, disadvantage one has to take into account.

(Refer Slide Time: 21:59)



Pipelining Example

Time slot	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Results
t0	Instr 1					
t1	Instr 2	Instr 1				
t2	Instr 3	Instr 2	Instr 1			
t3	Instr 4	Instr 3	Instr 2	Instr 1		
t4	Instr 5	Instr 4	Instr 3	Instr 2	Instr 1	complete
t6	Instr 6	Instr 5	Instr 4	Instr 3	Instr 2	complete

Assumed that all the instructions take one clock cycle

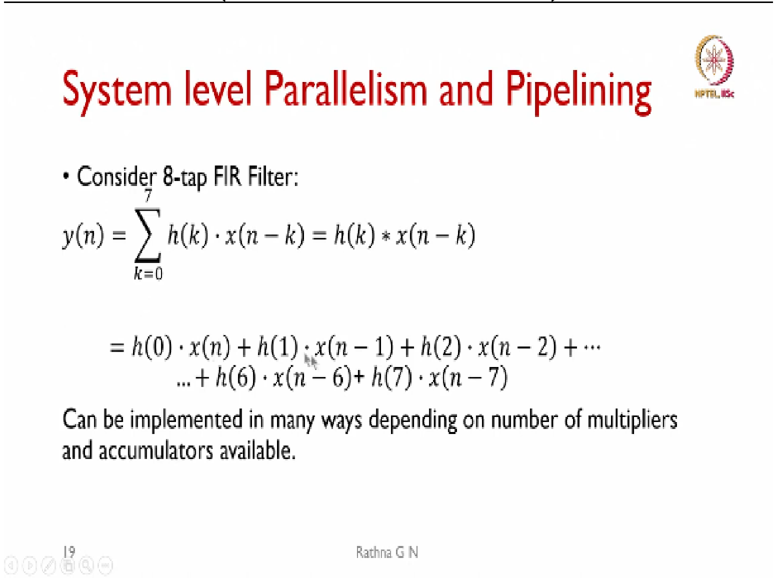
- Five stages:
  - Stage 1: Instruction (Instr) fetch
  - Stage 2: Instruction decode
  - Stage 3: Operand fetch
  - Stage 4: Execute
  - Stage 5: Save the results

18 Rathna G N

So, depending on this how many stages of pipeline what we can provide what will be looked at. So, with discussion and everything all DSP processor, most of them use 5 stage pipeline. So, the first stage is going to be instruction fetch, we represent it as Instr. So, what happened at t_0 time slot, the first instruction is fetched in the stage 1, in the second clock cycle or t_1 time.

So, instruction 2 is going to be fetched in the stage 1, whereas this instruction moves to stage 2, as you will be seeing that as clock cycle is increasing instruction is moving further in nets pipe that is what we will show it. So, finally, the result is going to be available in t_6 clock cycle, or if there are no multiple, we call it as this the last stage as the save the results. And stage 4 is execute most of the here we assume that it is going to take 1 clock cycle, that is why the once it reaches stage 5 we said that result is out actually. But if it is going to take more clock cycle, then our result maybe a little bit delayed.

(Refer Slide Time: 23:23)



System level Parallelism and Pipelining

- Consider 8-tap FIR Filter:

$$y(n) = \sum_{k=0}^{7} h(k) \cdot x(n-k) = h(k) * x(n-k)$$

$$= h(0) \cdot x(n) + h(1) \cdot x(n-1) + h(2) \cdot x(n-2) + \cdots$$
$$\ldots + h(6) \cdot x(n-6) + h(7) \cdot x(n-7)$$

Can be implemented in many ways depending on number of multipliers and accumulators available.

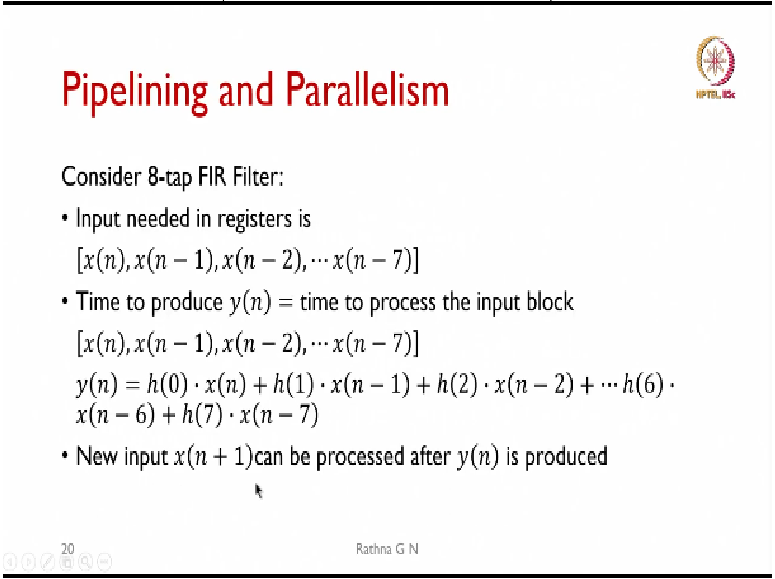
19 Rathna G N

So, just we will see how we can implement this parallelism and pipelining using an example. So, although I have not discussed my FIR filter, still I am taking this as an example 8 tap FIR filter, or you can assume it is a convolution summation what we will be doing it. So, $y(n)$ is given by the equation $\sum_{k=0}^{7} h(k) \cdot x(n-k)$, I was telling in the previous class why, I am going to consider $x(n-k)$ instead of $h(k)$.

One of the example, I gave it as my memory basically, what I will be using for $h(k)$ is the program memory which is going to be stored much earlier in the memory. So, whereas $x(n-k)$, because I want to implement the circular buffer, so, this gives me that whatever the latest after k whatever sample comes, I can be rewriting in the same memory location. So, that is how we use this equation to implement our convolution output or we call that also FIR filter.

So, you will be seeing that the normal notation what you people will be using it $h(k)$ in convolved with $x(n-k)$. So, when I expand this, this is the equation and further the summation is going to be expanded. So, then you will be seeing the sum $y(n)$ is given by your $h(0) \cdot x(n) + h(1) \cdot x(n-1) + h(2) \cdot x(n-2) + \cdots + h(6) \cdot x(n-6)+h(7) \cdot x(n-7)$. So, this can be implemented in many ways, depending on number of multipliers and accumulators available. So, we will see what is the thing is going to happen.

(Refer Slide Time: 25:13)



Pipelining and Parallelism

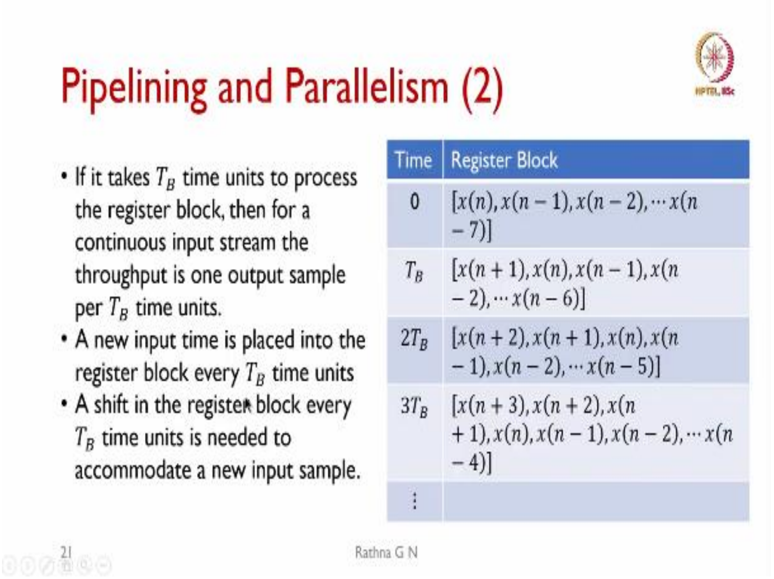
Consider 8-tap FIR Filter:

- Input needed in registers is $[x(n), x(n-1), x(n-2), \cdots x(n-7)]$
- Time to produce $y(n)$ = time to process the input block $[x(n), x(n-1), x(n-2), \cdots x(n-7)]$
  $y(n) = h(0) \cdot x(n) + h(1) \cdot x(n-1) + h(2) \cdot x(n-2) + \cdots h(6) \cdot x(n-6) + h(7) \cdot x(n-7)$
- New input $x(n+1)$can be processed after $y(n)$ is produced

20 Rathna G N

So, that is input needed in registers as we know this should be $x(n), x(n-1), x(n-2), \cdots x(n-7)$ is continuously available for us, and then when we are going to get the output that is we call it as time to produce $y(n)$, time to process the input block. So, we say that, this is my input and $y(n)$ is given by this equation. So, that new input $x(n+1)$ can be processed after $y(n)$ is produced, that is $y(n+1)$ will be with respect to $x(n+1)$ or input what it is coming.

(Refer Slide Time: 25:56)



Pipelining and Parallelism (2)

- If it takes T_B time units to process the register block, then for a continuous input stream the throughput is one output sample per T_B time units.
- A new input time is placed into the register block every T_B time units
- A shift in the register block every T_B time units is needed to accommodate a new input sample.

Time	Register Block
0	$[x(n), x(n-1), x(n-2), \cdots x(n-7)]$
T_B	$[x(n+1), x(n), x(n-1), x(n-2), \cdots x(n-6)]$
$2T_B$	$[x(n+2), x(n+1), x(n), x(n-1), x(n-2), \cdots x(n-5)]$
$3T_B$	$[x(n+3), x(n+2), x(n+1), x(n), x(n-1), x(n-2), \cdots x(n-4)]$
⋮	

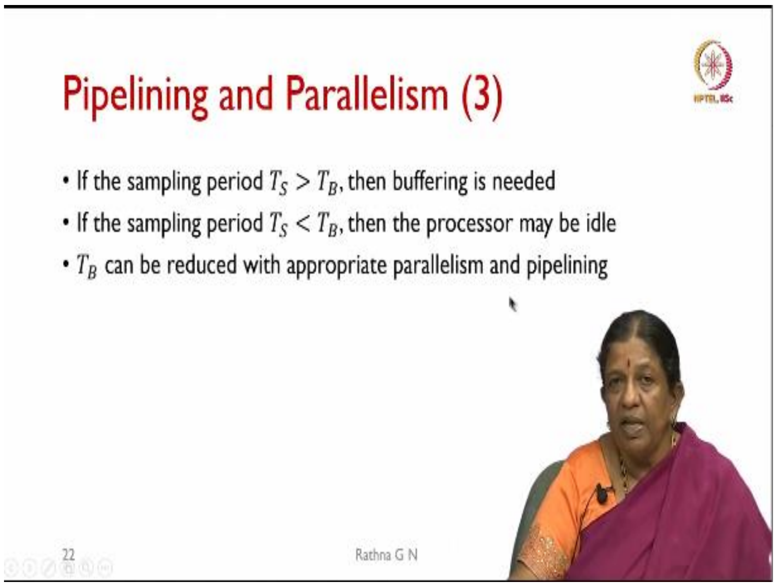
21

Rathna G N

So, how is this shown in this, as I mentioned in the one of the slides that it takes T_B, here it is shown with capital T there I showed it a small t, time units to process your register block, then for a continuous input stream, that throughput is 1 output sample per T_B time units. So, a new input time is placed into the register block every T_B time unit. A shift in the register block every T_B time units is needed to accommodate a new input sample.

This is the theory how it is going to be done as shown in this block diagram. So, time is 0 so I have all the inputs $x(n)$ to $x(n-7)$ in the register, when next clock cycle T_B, that is the delay of my computation time, basically or this block takes T_B unit, then I will be getting the new sample. So, $x(n+1)$ to $x(n-6)$ here in this case, $x(n-7)$is the last data which is going to be thrown out. So, in the second clock cycle, we will be seeing that you will be throwing out even $x(n-6)$, so on whenever new data comes into your buffer.

(Refer Slide Time: 27:15)



Pipelining and Parallelism (3)

- If the sampling period $T_S > T_B$, then buffering is needed
- If the sampling period $T_S < T_B$, then the processor may be idle
- T_B can be reduced with appropriate parallelism and pipelining

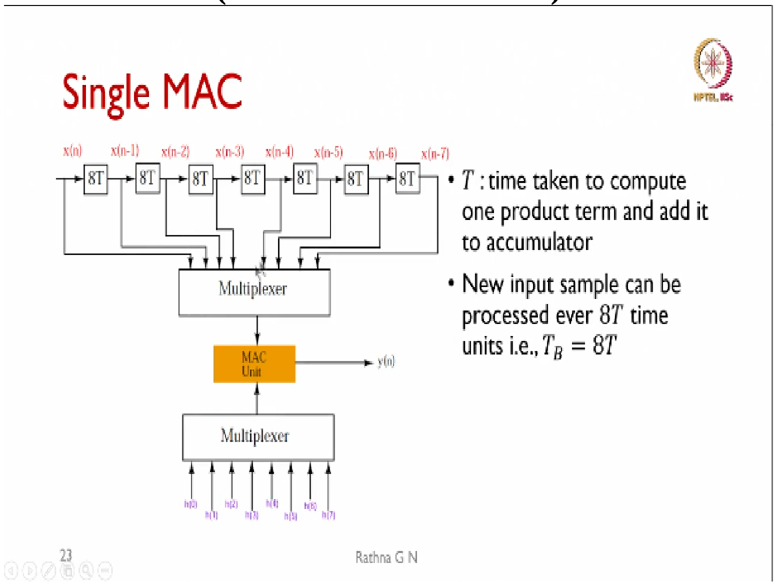
22

Rathna G N

So, the thing is, we said that sampling period we have assumed is T$_S$ which should be greater than my T$_B$, then what happens because my sampling frequency is much higher than my computation time, then whatever data sample comes, basically, I need to store them because input is coming at a much faster rate. So, which I have to store it so that I would not miss any of the sample. At the same time, if my sampling period is less than the computation time.

Then what is going to happen processor may become idle because I finished my computation, I have to wait for my data to come in. So, my processor is idling at that time. So, how we can reduce this T$_B$ with appropriate parallelism and then pipelining what we will see in the next few slides.

(Refer Slide Time: 28:10)



Single MAC

- T : time taken to compute one product term and add it to accumulator
- New input sample can be processed ever $8T$ time units i.e., $T_B = 8T$

23

Rathna G N

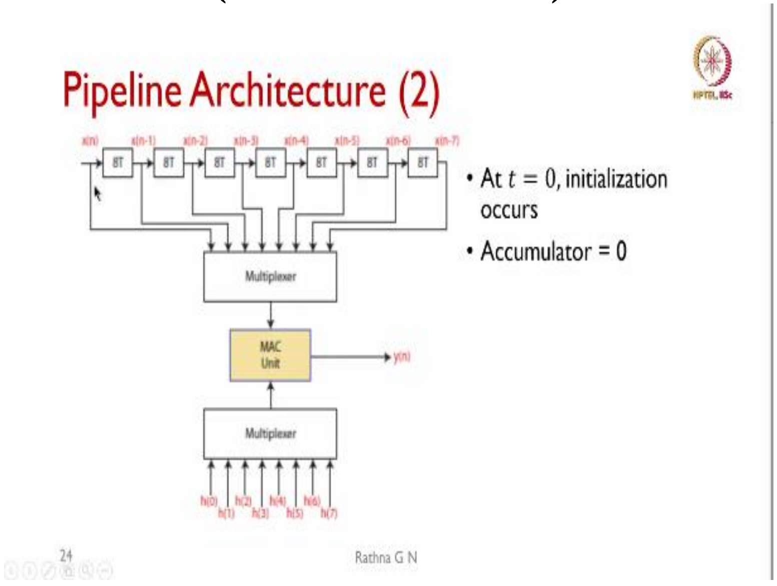
So, first one, what is it? If I am using only a single multiplier and accumulator, then what is the thing I am going to how I am going to do this filtering, basically. So, I have $x(n)$ here, $n - 1$

usually we will be assigning it as 0 all of them initially, and this is a 8T clock sample delay, because I have seen that it is going to take 8 clock cycle for that new data to come in. So, which are fed through the multiplexer.

And then this is my MAC unit, because I have a single MAC I have to do all these 8 operations. And then how I am going to feed in my coefficients basically, $h(0)$ to $h(7)$ is through other multiplexer, which is appropriately fed, whenever this data is going to be there. I will be doing the multiplication accumulation in a single block in this case, what is the example we have considered.

And then once complete thing is done after T what we say this 8 clock cycles, my $y(n)$ is a valid output although it is coming every clock cycle, but we say at the 8th clock cycle I will be getting the correct output and from there onwards, every clock cycles I will get the output. So, here you can see T is a time taken to compute 1 product term and add it to accumulator and then new input sample can be processed over 80 time units. So, our block computation time is 80 in this, so I can take the new sample after 80 time units.

(Refer Slide Time: 29:49)

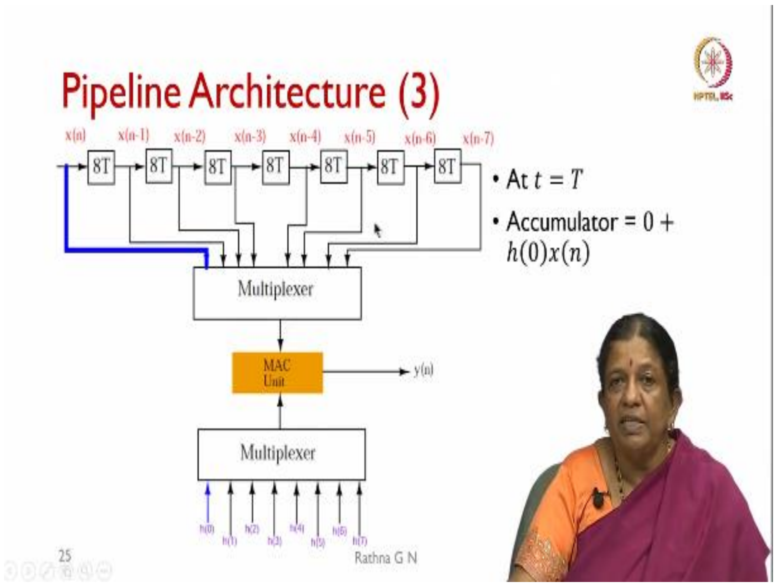


Pipeline Architecture (2)

- At $t = 0$, initialization occurs
- Accumulator = 0

So, continuing with the thing, at time $t = 0$. We will be doing the initialization and accumulator is also made 0. So, this is how the previous slide itself what it shows in this case.

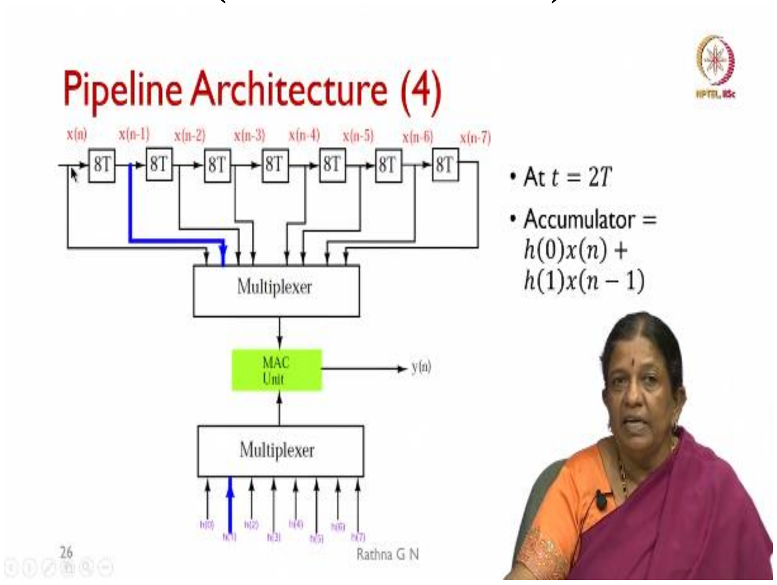
(Refer Slide Time: 30:03)



- At $t = T$
- Accumulator = $0 + h(0)x(n)$

So, in the next clock cycle what happens at $t = T$ the accumulator which was initially 0. Now, the first sample $h(0)$ is taken from here this multiplexer and then your $x(n)$ is taken the first sample from the side from through this multiplexer to your MAC. So, they get multiplied here and then added with the previous value in the accumulator.

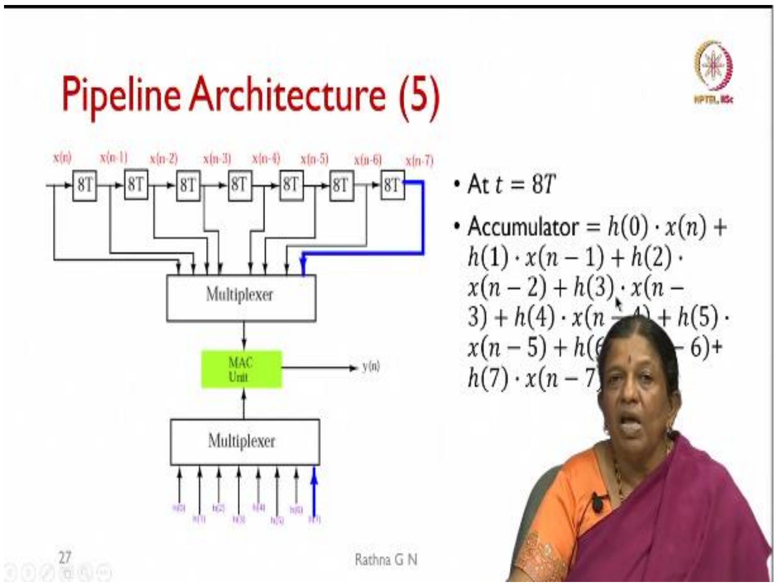
(Refer Slide Time: 30:33)



- At $t = 2T$
- Accumulator = $h(0)x(n) + h(1)x(n-1)$

So, what happens at 2T, so, you will be seeing that whatever data here $x(n)$ gets moved to here. So, this $x(n-1)$ in the next clock cycle becomes and then you have to multiply with your $h(1)$. So, that is how the samples have been chosen by these multiplexers which are fed to MAC unit. So, then you will be getting the output. So, this is the second clock cycle $h(0)$ into $x(n)$ in the previous 1, now $h(1) \cdot x(n-1)$ is going to happen.

(Refer Slide Time: 31:07)



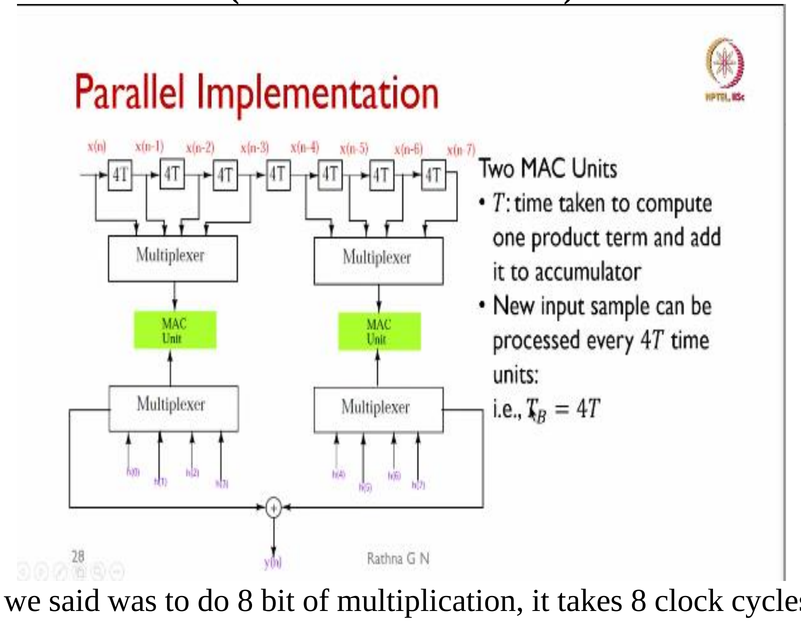
Pipeline Architecture (5)

- At $t = 8T$
- Accumulator = $h(0) \cdot x(n) + h(1) \cdot x(n-1) + h(2) \cdot x(n-2) + h(3) \cdot x(n-3) + h(4) \cdot x(n-4) + h(5) \cdot x(n-5) + h(6) \cdot x(n-6) + h(7) \cdot x(n-7)$

27 Rathna G N

So, what happens in the next clock cycle, so, we are skipping few of them, we can put it as dot, dot, dot that is a will be going with the thing at $t = 8T$ at the last at what is the thing is going to happen what is shown here. So, the last sample, your $x(n-1)$ and then $h(7)$ is going to be taken into account from the multiplexer, which is getting multiplied and all the previous samples you will be seeing it which has come will be added with multiplied value, as you are seeing in the accumulator.

(Refer Slide Time: 31:44)



Parallel Implementation

Two MAC Units

- T: time taken to compute one product term and add it to accumulator
- New input sample can be processed every $4T$ time units:
  i.e., $T_B = 4T$

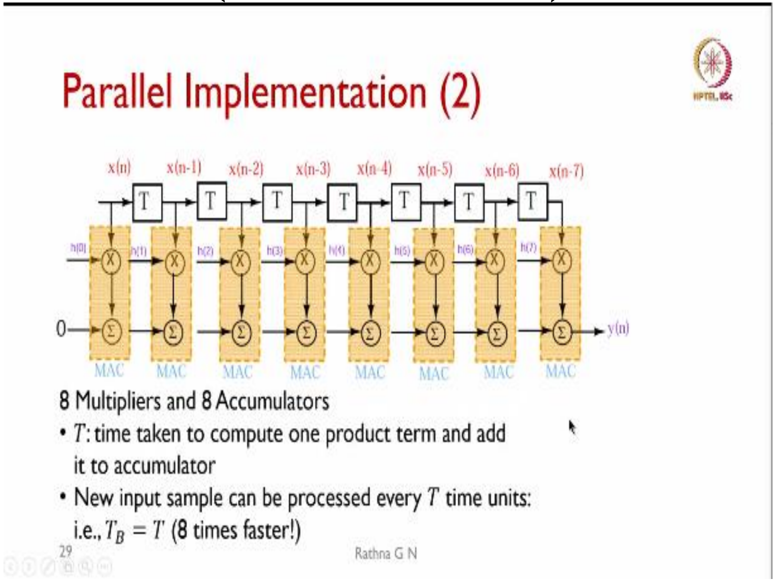
28 Rathna G N

So, that is what we said was to do 8 bit of multiplication, it takes 8 clock cycles, can I improve upon it, by doing the parallel implementation. So, how I can do it, I will show in this case, 2 MAC units I am going to provide, then I will be having as you are seeing the hardware is increasing here, there are 4 multiplexor and 2 Mac units. And you will be seeing that because they are working in parallel.

They do not have any dependency basically, whatever data they have stored it, they will be working on it, and then you will be seeing that 4 of them are connected to one of the MAC unit and 4 samples are connected to the other MAC unit. So, they can work in parallel and then we have to last sample, what we have to do is we have to add them from both the cases, then only I will be getting y of n. Approximately, we say that $T_B = 4T$.

So, although this one addition we have not accounted for there will be we assume it as delta t, 4T + delta t which is ignorable. That is what we see it but for large cases it may have to be considered. So, we say that my computation is twice that of with single MAC. So, you will be also multiple cores in your CPU you will be seeing it what you want is 8 core or 7 core what you will be telling I want to have 7 times what I have to get it. Why you would not get that result? You can see from this visually saying that I would not be getting it twice what I want it.

(Refer Slide Time: 33:26)



Parallel Implementation (2)

8 Multipliers and 8 Accumulators

- T: time taken to compute one product term and add it to accumulator
- New input sample can be processed every T time units: i.e., $T_B = T$ (8 times faster!)

29

Rathna G N

So, we will see that instead of 2 MAC units I will use for each multiplication, I know what should be my filter length; most of you know that in FPGs multiple DSP blocks are there. And even GPUs, for that matter, you have multiple multiplication and then add units are there in the GPUs, that is how you will be getting your speed up using GPUs compared to your CPUs. So, here I have used 8 MACs in this case. So, zero will be for the first adder what it is going to be pushed in and then rest of the thing can be fed from the previous stages.

So, there will be as we worked out in the previous class accumulator, it will be taking $n + 1$ clock cycle from this you will be seeing it, for this the previous one what you will be getting it in the next clock cycle, you can add it and then send it out. So, you will be having every T units, I can get the input clock in this case. So, all of them are working in parallel. And then we will be adding them up and then y of n will be my output.

So, we say 8 multipliers and 8 accumulators what we have used it T is equal to time taken to compute one product term and then add it to accumulator and new input sample can be processed every t time units, we call it as 8 times faster that is why you will be seeing exclamatory mark whether I am going to achieve this or not, as I have mentioned, there will be overheads and other things.

So, we may not get so much speed in this case. So, one has to bear with whatever we are going to get the thing with additional overheads. Happy learning and then thank you for listening to this lecture.