

COMPUTER ARCHITECTURE AND ORGANIZATION

**Prof.Indranil Sengupta,
Prof. Kamalika Datta
Computer Science and Engineering
IIT Kharagpur**



INDEX

S. No	Topic	Page No.
	<i>Week 1</i>	
1	Evolution of Computer Systems	1
2	Basic Operation of a Computer	22
3	Memory Addressing and Languages	39
4	Software and Architecture Types	56
5	Instruction Set Architecture	72
	<i>Week 2</i>	
6	Number Representation	90
7	Instruction Format and Addressing Modes	112
8	CISC and RISC Architecture	131
9	MIPS32 Instruction Set	151
10	MIPS Programming Examples	167
11	SPIM – A MIPS32 SIMULATOR	183
	<i>Week 3</i>	
12	Measuring Cpu Performance	202
13	Choice Of Benchmarks	221
14	Summarizing Performance Results	239
15	Amadahlâ€™S Law (Part 1)	252
16	Amadahlâ€™S Law (Part 2)	267
	<i>Week 4</i>	
17	Design Of Control Unit (Part 1)	280
18	Design Of Control Unit (Part 2)	297
19	Design Of Control Unit (Part 3)	313
20	Design Of Control Unit (Part 4)	325
21	Mips Implementation (Part 1)	345
22	Mips Implementation (Part 2)	361
	<i>Week 5</i>	
23	Processor Memory Interaction	372
24	Static And Dynamic Ram	388
25	Asynchronous Dram	404
26	Synchronous Dram	417
27	Memory Interfacing And Addressing	432
	<i>Week 6</i>	

28	Memory Hierarchy Design (Part 1)	448
29	Memory Hierarchy Design (Part 2)	468
30	Cache Memory (Part 1)	479
31	Cache Memory (Part 2)	497
32	Improving Cache Performance	511

Week 7

33	Design Of Adders (Part 1)	530
34	Design Of Adders (Part 2)	550
35	Design Of Multipliers (Part 1)	570
36	Design Of Multipliers (Part 2)	586
37	Design Of Dividers	604

Week 8

38	Floating-Point Numbers	625
39	Floating-Point Arithmetic	642
40	Basic Pipelining Concepts	657
41	Pipeline Scheduling	674
42	Arithmetic Pipeline	691

Week 9

43	Secondary Storage Devices	706
44	Input-Output Organization	726
45	Data Transfer Techniques	740
46	Interrupt Handling (Part 1)	754
47	Interrupt Handling (Part 2)	770

Week 10

48	Direct Memory Access	784
49	Some Example Device Interfacing	800
50	Exercises On I/O Transfer	815
51	Bus Standards	828
52	Bus Standards	845

Week 11

53	Pipelining The Mips32 Data Path	864
54	Mips Pipeline (Contd.)	879
55	Pipeline Hazards (Part 1)	890
56	Pipeline Hazards (Part 2)	907
57	Pipeline Hazards (Part 3)	925
58	Pipeline Hazards (Part 4)	939

Week 12

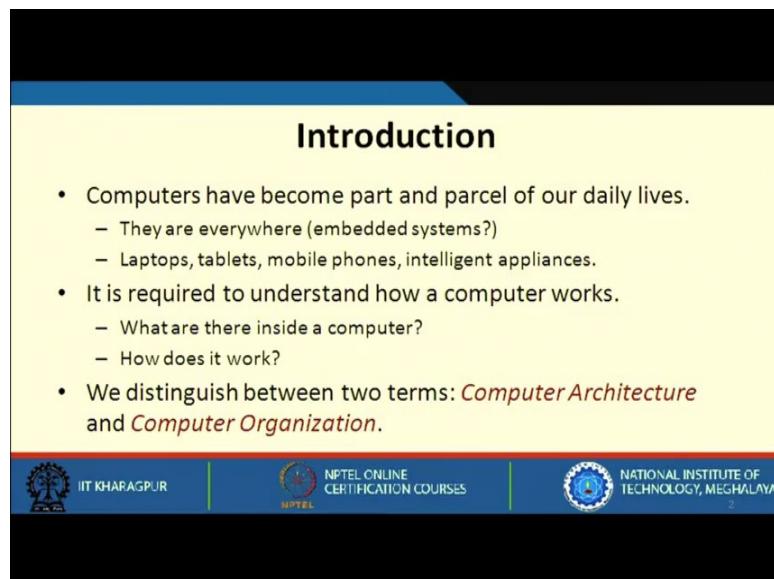
59	Multicycle Operations In Mips32	954
60	Exploiting Instruction Level Parallelism	966
61	Vector Processors	981
62	Multi-Core Processors	997
63	Some Case Studies	1011
64	Summarization Of The Course	1027

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 01
Evaluation of Computer System

I welcome you all to the MOOC course on Computer Architecture and Organization. In this particular course, we expect to cover various aspects of computer design where you will be seeing how we can make a computer faster, how a computer actually works, how the information data are stored there and various other aspects. The lectures will span over 12 weeks where we will cover the instruction set architecture, processor design, arithmetic and logic unit design, memory unit design, input-output system design and then we will also cover parallel processing and pipeline.

(Refer Slide Time: 01:28)



Introduction

- Computers have become part and parcel of our daily lives.
 - They are everywhere (embedded systems?)
 - Laptops, tablets, mobile phones, intelligent appliances.
- It is required to understand how a computer works.
 - What are there inside a computer?
 - How does it work?
- We distinguish between two terms: *Computer Architecture* and *Computer Organization*.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

To start with this course, I will come first to evolution of computer system. So, we all know that computer has become a part and parcel of our daily lives. We cannot disagree to this fact. We see everywhere computers that is some kind of processing unit. When you think about a laptop which we use in our daily use, tablets, mobile phones which are used by one and all today and intelligent appliances like off course your smart phone is one apart from that you have smart watch, and various other appliances. So, computer has become a part and parcel of our life. So, we need to understand how a computer

actually works. So, what is there inside a computer? So, we in this particular course we will be seeing all these various aspects where the two terms computer architecture and computer organization will be taken care.

(Refer Slide Time: 02:40)

- Computer Organization:
 - Design of the components and functional blocks using which computer systems are built.
 - *Analogy*: civil engineer's task during building construction (cement, bricks, iron rods, and other building materials).
- Computer Architecture:
 - How to integrate the components to build a computer system to achieve a desired level of performance.
 - *Analogy*: architect's task during the planning of a building (overall layout, floorplan, etc.).

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now coming to what is computer architecture, and what is computer organization, the title of the course. So, here computer architecture consists of those attributes of the system that are visible to the programmer. By this what we mean is how the various components of a computer system are integrated to achieve the desired level of performance. In an analogy, I can say that you think of an architect who does who plans the entire design of your house, but it is ultimately the civil engineers who actually does the exact building like what kind of construction will be taken care of, how the construction will be taken care of, how much percentage of cement, bricks will be there , will be taken care by a civil engineer. So, in that respect the design of components and functional blocks using which computer systems are built comes to the organizational part.

So, I will take a very small example like you have in your computer some functional blocks like your processor unit; inside processor unit we will be seeing that we have many other components like registers, ALU and other units. I will just take a small example let say I will have an adder, but what kind of adder I will be having that is to the discretion of the computer organization, whether we will have a carry save adder or a

carry look ahead adder or anything else. So, these are the two different aspects of computer organization and computer architecture that we will be seeing in this particular course.

(Refer Slide Time: 04:49)

Historical Perspective

- Constant quest of building automatic computing machines have driven the development of computers.
 - Initial efforts:* mechanical devices like pulleys, levers and gears.
 - During World War II:* mechanical relays to carry out computations.
 - Vacuum tubes developed:* first electronic computer called ENIAC.
 - Semiconductor transistors developed* and journey of miniaturization began.
 - SSI → MSI → LSI → VLSI → ULSI → Billions of transistors per chip

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Coming to the historical perspective, how computers have evolved over the years. So, whenever there is a need for doing certain things then only something comes up like a constant quest of building automatic computing machines have driven the development of computers. So, in the initial efforts, some mechanical devices like pulleys, levers, gears were built. During World War 2, mechanical relays were used to perform some kind of computation like using small relays people design circuits to carry out the operations. Then comes vacuum tubes using which the first electronic computer called ENIAC was developed. And from then semiconductor transistors were developed when semiconductor transistors came into picture then the journey of miniaturization started. First with small scale integration then people moved with medium scale integration then large scale integration then to very large scale integration and now the era of ultra large scale integration where we stand today.

(Refer Slide Time: 06:22)

PASCALINE (1642)

- Mechanical calculator invented by B. Pascal.
- Could add and subtract two numbers directly, and multiply and divide by repetition.

So, this is the first mechanical calculator that was invented by B Pascal. So, this particular calculator could add only two numbers, it can only add two numbers or it can only subtract two numbers. And if you wanted to do multiplication and division, it could have been done by repeated addition or repeated subtraction.

(Refer Slide Time: 06:58)

Babbage Engine

- First automatic computing engine was designed by Charles Babbage in the 19th century, but he could not build it.
- The first complete Babbage engine was built in 2002, 153 years after it was designed.
 - 8000 parts.
 - Weighed 5 tons.
 - 11 feet in length.

Then the Babbage engine came this was the first automatic computing engine designed by father of computer Charles Babbage in the 19th century, but he could not build that only he designed that. Later in 2002 that is 153 years after its design, it was built and it

consists of 8,000 parts and it weighted 5 tons and it was 11 feet in length. So, you can imagine how large it is.

(Refer Slide Time: 07:43)

The slide is titled "ENIAC" and includes the subtitle "(Electrical Numerical Integrator and Calculator)". It lists two bullet points: "Developed at the University of Pennsylvania." and "Used 18,000 vacuum tubes, weighed 30 tons, and occupied a 30ft x 50ft space." Below the text is a black and white photograph of two women standing next to the massive ENIAC computer, which is filled with vacuum tubes and cables. At the bottom of the slide are logos for IIT Kharagpur, NPTEL, and NPTEL Online Certification Courses, along with a small video frame showing a person speaking.

The first electronic computer was built which is called ENIAC - Electrical Numerical Integrated and Calculator. It was developed by University of Pennsylvania, and it uses 18,000 vacuum tubes and weighted 30 tons, and it also occupied a very large space which is 30 feet cross 50 feet. So, what is a vacuum tube? Vacuum tube is a device that controls electric current between electrodes in an evacuated container in a tube. So, using those vacuum tubes the first computer ENIAC was built. It also dissipated a huge amount of heat.

(Refer Slide Time: 08:44)

Harvard Mark 1

- Built at the University of Harvard in 1944, with support from IBM.
- Used mechanical relays (switches) to represent data.
- It weighed 35 tons, and required 500 miles of wiring.

Next was Harvard mark 1. This was built at the University of Harvard in 1944, with support from IBM and it uses mechanical relays and off course, some electric signals were also used to work with the relays to represent the data. And it also weighted 35 tons, and required 500 miles of wiring. So, these are the computers which were built in the early stages.

(Refer Slide Time: 09:20)

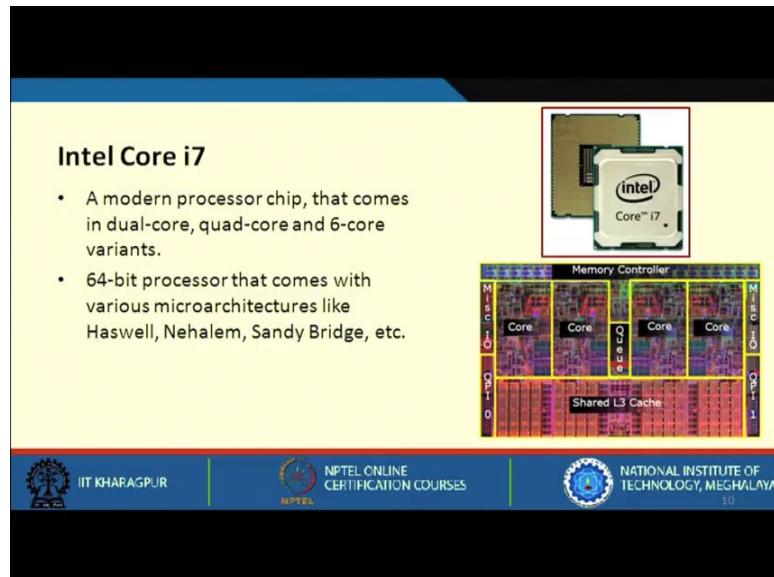
IBM System/360

- Very popular mainframe computer of the 60s and 70s.
- Introduced many advanced architectural concepts that appeared in microprocessors several decades later.

Then comes in 60s and 70s where the popular mainframe computer came into picture. So, this popular computer IBM system 360 was introduced; it introduces many advanced

architectural concepts that we use today, but you can see from the picture that how big it was. But as I said some of the architectural aspects or concepts that were used in that computer appeared in today's microprocessor several decades later.

(Refer Slide Time: 10:03)



Now, we stand, where the modern processor chips comes in dual-core, quad-core and also in 6-core variants. So, you can see that how many core today's computer has, where each core is a 64-bit processor that comes with various micro architectures. These are the various micro architectures within an I7, where they are called Haskell, Haswell, Nehalem, Sandy Bridge etc.

(Refer Slide Time: 10:52)

Generation	Main Technology	Representative Systems
First (1945-54)	Vacuum tubes, relays	Machine & assembly language ENIAC, IBM-701
Second (1955-64)	Transistors, memories, I/O processors	Batch processing systems, HLL IBM-7090
Third (1965-74)	SSI and MSI integrated circuits Micropogramming	Multiprogramming / Time sharing IBM 360, Intel 8008
Fourth (1975-84)	LSI and VLSI integrated circuits	Multiprocessors Intel 8086, 8088
Fifth (1984-90)	VLSI, multiprocessor on-chip	Parallel computing, Intel 486
Sixth (1990 onwards)	ULSI, scalable architecture, post-CMOS technologies	Massively parallel processors Pentium, SUN Ultra workstations

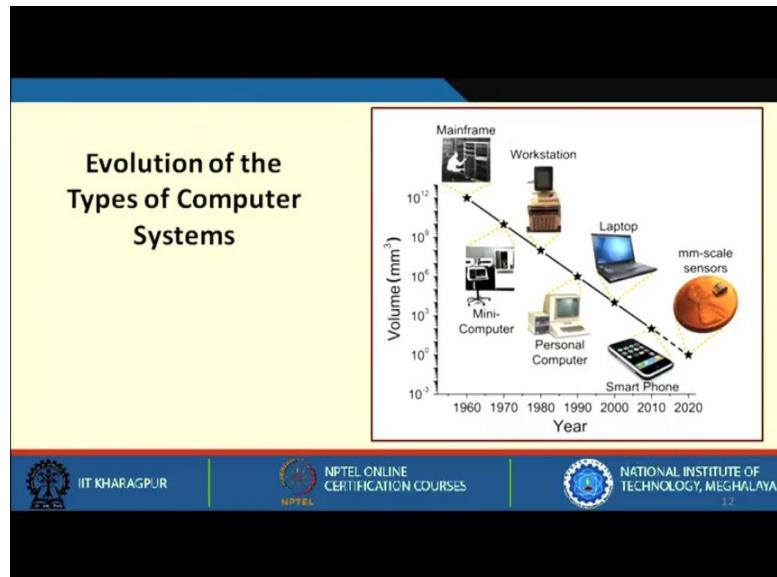
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, coming to the generation of computers. So, broadly we can classify the growth of computer, how computer has evolved into some generations. So, these actually represent some features of operating system and representative machines. So, between the years 1945 to 54, vacuum tubes and relays were used; and the representative system uses machine language. So, if you want to enter some data into the computer for processing, you have to enter in machine language or either in assembly language, no high level language were used to enter the data. Then comes the second generation where transistors, memories, IO processors are the main technologies. Here batch processing is performed, and you could enter the data in high level language.

Then comes the third generation, where we were into semiconductor industry where small-scale integration and medium-scale integration integrated circuits were used. And in that IBM 360, Intel 8008, where the first one to make the step into this computer market. Then in the fourth and fifth generation, this large scale integration and very large scale integration came into picture and we could see multi processors. Now, what has happened is like the space has become larger because the components that we use to build those basic blocks has become smaller. So, in a same space where earlier we could keep small number of components, now we could keep more number of components into the same space. And hence multiprocessors comes into picture. And now we are in the era of ultra large scale integration with many scalable architectures and post-CMOS

technologies. Now, we also have massively parallel processors like Pentium, SUN Ultra workstation etc.

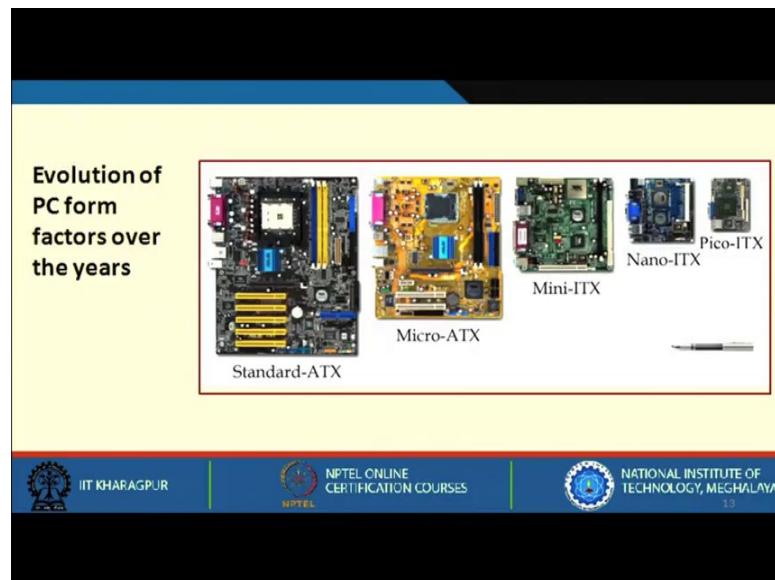
(Refer Slide Time: 13:34)



Now, you see that evolution of the types of computer. So, initially mainframes came in the year 60s and late 70 in between 60s and 70s. Then came mini computers then came workstations, and finally we see personal computers. And then further we see laptops and smart phones. So, every smart phone today is having some processing capability that is why we call it smart phone with some smart features into it. And we also have now millimeter scale sensors and these are not only sensors that it will sense something rather these sensors are having some intelligent capability like some processing capability which can process those data that are collected from the sensors and also it has the communication feature, so that it can communicate to others if required. So, this millimeter scale sensors have both the capabilities of processing as well as communication. So, we can see how the evolution of types of computer systems have come up.

And now where we stand in the future, we will see large scale IOT based systems where these sensors itself are very small, where they will have not only the sensing capability, but also the processing capability.

(Refer Slide Time: 15:25)



Now, you see this evolution of PC form factors over the years. This is the first standard ATX where you can see the motherboard slot. You can see this is a motherboard, you can see the processor, you can see the slots for memories for all other components. Now, how that miniaturization has taken place. First it was standard-ATX, then micro-ATX then goes mini-ITX, now nano-ITX now we are in pico-ITX. So, we are in the era of miniaturization. And now with miniaturization of course, we are able to do some good thing for performance, but at the same time power issue is one of the important factors that we have to also handle which we cannot deny.

(Refer Slide Time: 16:37)



Now, inside a laptop if you see what all components we have. So, we have shrunked each of the components. So, hard drive is now getting replaced by flash-based memory devices and because of miniaturization cooling is a major issue that has come up.

(Refer Slide Time: 17:23)

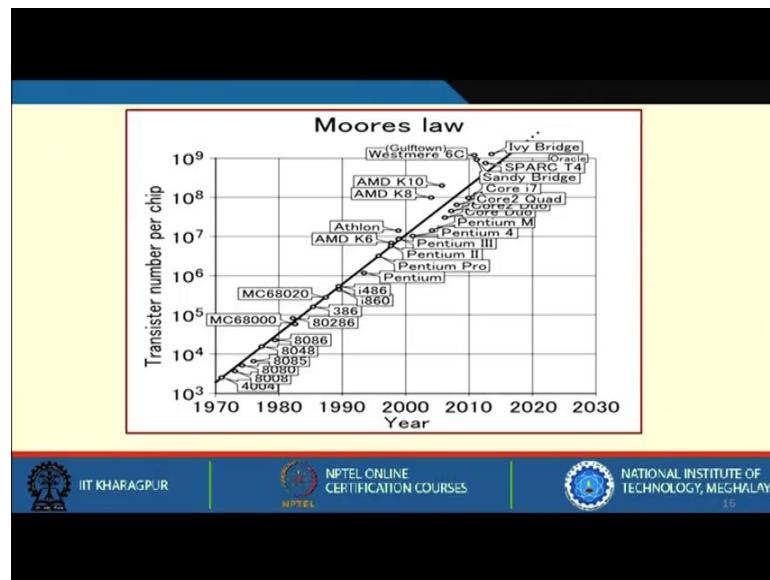
Moore's Law

- Refers to an observation made by Intel co-founder Gordon Moore in 1965. He noticed that the number of transistors per square inch on integrated circuits had doubled every year since their invention.
- Moore's law predicts that this trend will continue into the foreseeable future.
- Although the pace has slowed, the number of transistors per square inch has since doubled approximately every 18 months. This is used as the current definition of Moore's law.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

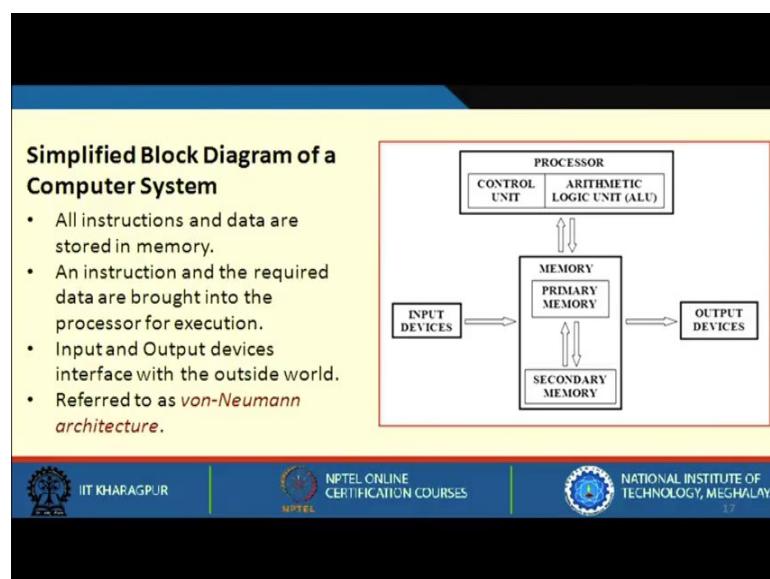
Now, this is the famous Moore's law that refers to an observation made by Intel co-founder Gordon Moore in 1965. What he noticed is that the number of transistors per square inch on integrated circuits had doubled every year since invention. And this law predicts that this trend will continue into the foreseeable picture, but now Moore's law stand here where it says that the number of transistor that can be placed in an integrated circuit gets doubled in every 18 months. And this has held over a period.

(Refer Slide Time: 18:17)



Now, you can see this diagram which how Moore's law works, this is the number of transistor per chip and this is over the years how it has grown. So, this straight line actually indicates that Moore's law holds. And starting from 4004 we are now in ivy bridge, sandy bridge, and various architectural advancement has taken place and number of transistor that can be put in a chip has doubled in every 18 months. And as long as it is a straight line, we can say that Moore's law holds.

(Refer Slide Time: 19:16)



Now, coming to the simplified block diagram of a computer system. So, in this diagram what you can see is that we have a processor. The processor is having control unit and arithmetic logic unit. We have a memory; memory is divided into primary memory and secondary memory. And you have input devices and output devices. In this architecture course we will be taking each and every aspect of this design like we will be seeing the processor unit design; in the processor unit design there are two parts we will be seeing control unit design and arithmetic logic unit design. We will also look into the memory, memory is broadly divided into primary memory and secondary memory. What we have in primary memory we will be seeing, what we have in secondary memory we will be seeing. And of course, how the inputs are given to the computer and how we get the outputs from the computer that also we will be look into.

So, all instructions and data are stored in memory. Whatever instruction or data that we want to execute that is stored in memory. And every time an instruction and data is required it is brought from the memory to the processor for execution, and input output devices are connected to it. So, if input is required input is taken from an input device processing takes place in the processor and the output is provided in the output device. This typical architecture where we store both program and data in the memory alongside we call it von-Neumann architecture we will be seeing this in more detail little later.

(Refer Slide Time: 21:31)

The slide has a yellow background with a black header and footer. The title 'Inside the Processor' is at the top. Below it is a bulleted list of facts about the processor. At the bottom, there are logos for IIT Kharagpur, NPTEL, and NIT Meghalaya.

- **Inside the Processor**
 - Also called *Central Processing Unit* (CPU).
 - Consists of a *Control Unit* and an *Arithmetic Logic Unit* (ALU).
 - All calculations happen inside the ALU.
 - The Control Unit generates sequence of control signals to carry out all operations.
 - The processor fetches an instruction from memory for execution.
 - An instruction specifies the exact operation to be carried out.
 - It also specifies the data that are to be operated on.
 - A program refers to a set of instructions that are required to carry out some specific task (e.g. sorting a set of numbers).

Now, let us see what is there inside a processor. A processor is also called a Central Processing Unit; it consists of a Control Unit and it consists of an Arithmetic Logic Unit. All the calculations happens inside the CPU. So, any arithmetic computation, like we need to add two numbers, we need to divide two numbers or any arithmetic operation that are performed inside the ALU. The control unit basically generates the sequence of control signals to carry out all operations. So, all the operations that are performed, it is the control unit that generates those sequence of control signals i.e, when I say that we will execute an instruction. So, you have to instruct the computer that, execute this particular instruction. So, giving the computer some kind of instruction, some kind of control signals that, yes now you have to execute this now you have to execute that and finally, you have to store the result or you have to display the result. So, all these steps that we are instructing to a computer is generated by the control unit.

In a processor an instruction actually specify the exact operation that is to be performed. So, the instruction will tell you ADD A, B. So, A, B are some operands that perform the required operation, like ADD. So, you have to instruct that this is an operation that has to be performed on some operands. It also specifies the data that are to be operated on.

And now let us see what is a program. I have talked about a single instruction. Now, a program is a set of instruction, like I need to add ten numbers. So, adding ten numbers I have to write a set of some instructions; those set of some instructions are written to perform that particular task. A program is a set of instructions that constitute a program.

(Refer Slide Time: 24:35)

- What is the role of ALU?
 - It contains several registers, some general-purpose and some special-purpose, for temporary storage of data.
 - It contains circuitry to carry out logic operations, like AND, OR, NOT, shift, compare, etc.
 - It contains circuitry to carry out arithmetic operations like addition, subtraction, multiplication, division, etc.
 - During instruction execution, the data (operands) are brought in and stored in some registers, the desired operation carried out, and the result stored back in some register or memory.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, what is the role of ALU? ALU or the processor unit consists of several registers; some registers are called general purpose registers, and some are called special purpose registers, and some of them are temporary storage. So, what are these registers? Registers are some storage unit, and these registers are used to store data, then again we compute some operation and again store back that results into it. We store data for computation; and after the computation is performed, we also store back the data.

It contains circuitry to carry out the logic operations. So, basically when we say we are adding two numbers, we are subtracting two numbers, we are doing some kind of operation, some kind of logic operation is performed to carry out that particular operation. So, it also contains circuitry to carry out arithmetic operations like addition, subtraction, multiplication, division. We will be seeing in more detail every aspect of ALU in a separate lecture unit. Now, during execution, the data or the operands are brought in and stored in some register, the desired operation is carried out and the result is stored back in some register or memory. So, what does it means that during an instruction execution the instructions and data are stored in some memory locations. So, we have to bring the data from those memory locations into some of the registers, perform the operation and then we store back the data into either register or into those memory locations.

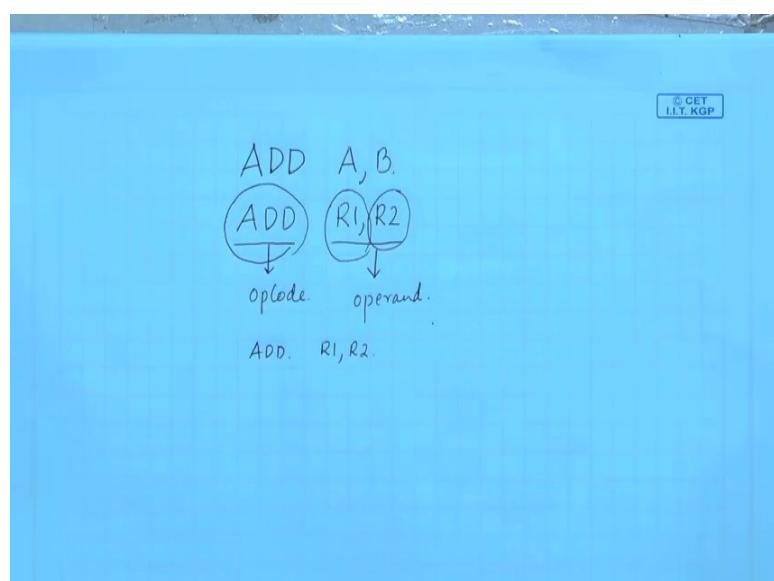
(Refer Slide Time: 26:41)

- What is the role of control unit?
 - Acts as the nerve center that senses the states of various functional units and sends control signals to control their states.
 - To carry out a specific operation (say, $R1 \leftarrow R2 + R3$), the control unit must generate control signals in a specific sequence.
 - Enable the outputs of registers R2 and R3.
 - Select the addition operation.
 - Store the output of the adder circuit into register R1.
 - When an instruction is fetched from memory, the operation (called *opcode*) is decoded by the control unit, and the control signals issued.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Finally what is the role of control unit? As I said earlier it generates the signals that is necessary to perform the task. So, it acts as a nerve center that senses the states of various functional units and sent control signals to control the states. Suppose you have to carry out an operation where you have to add R2 and R3 and store back in R1. So, what you need to do is that you need to enable the output of R2 and R3 such that the outputs of R2 and R3 are available in a place where you can do the operation. After the operation is performed it is stored in the register R3. So, you have to store the output in a circuit into the of the adder circuit into register R1.

(Refer Slide Time: 28:04)



Let us say this is an instruction. ADD A,B: this is an instruction. ADD R1,R2 is also an instruction. So, this instruction consists of two parts: the first part we call it Opcode and next part is the operand. Opcode specifies what operation we will be doing and operand is on which we will be doing the operation. So, in this particular case the operation that we will be doing is ADD and on which we will be operating are R1 and R2, these are the two registers where we will be doing the operation.

(Refer Slide Time: 29:10)

- **Inside the Memory Unit**
 - Two main types of memory subsystems.
 - *Primary or Main memory*, which stores the active instructions and data for the program being executed on the processor.
 - *Secondary memory*, which is used as a backup and stores all active and inactive programs and data, typically as files.
 - The processor only has direct access to the primary memory.
 - In reality, the memory system is implemented as a hierarchy of several levels.
 - L1 cache, L2 cache, L3 cache, primary memory, secondary memory.
 - Objective is to provide faster memory access at affordable cost.

Now, consider the memory unit. There are two types of memory subsystems, two main types. So, primary or main memory stores the active instructions and data for program being executed on the processor. And the secondary memory is used as backup and stores all active and inactive programs and data typically the files. Now the processor can only have a direct access to primary memory. As I said the programs and data is stored in your primary memory and whenever it is required the processor ask it from your primary memory and not from your secondary memory.

And in reality, the memory system is implemented as a hierarchy of several levels. So, we have L1 cache, we have L2 cache, we have L3 cache, primary memory and secondary memory. What is the objective of all these things to make the processing faster? So, we will be seeing all these aspects in course of time, but for now you must know that in the memory unit we store instructions and data. And for processing of those

instructions on data, you have to bring those instructions and data to the processors to execute it.

(Refer Slide Time: 30:59)

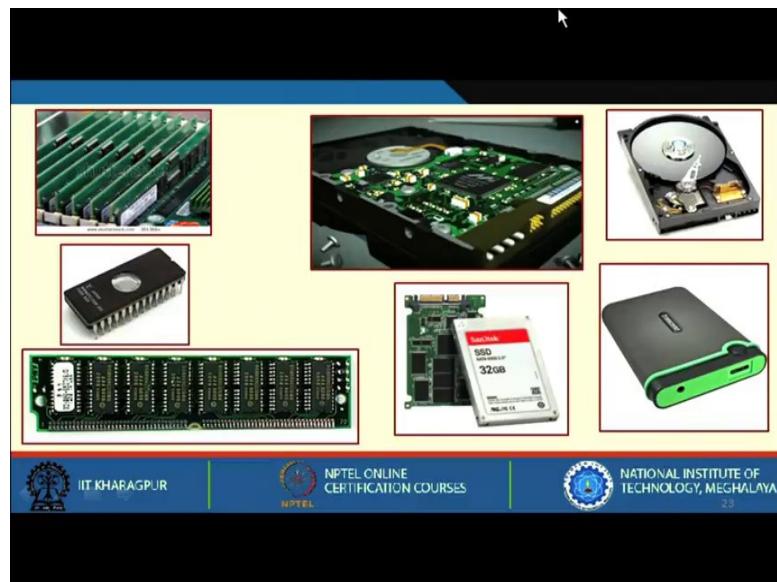
– Various different types of memory are possible.

- a) Random Access Memory (RAM), which is used for the cache and primary memory sub-systems. Read and Write access times are independent of the location being accessed.
- b) Read Only Memory (ROM), which is used as part of the primary memory to store some fixed data that cannot be changed.
- c) Magnetic Disk, which uses direction of magnetization of tiny magnetic particles on a metallic surface to store data. Access times vary depending on the location being accessed, and is used as secondary memory.
- d) Flash Memory, which is replacing magnetic disks as secondary memory devices. They are faster, but smaller in size as compared to disk.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, we have various different types of memory, Random Access Memory, Read Only Memory, we have Magnetic Disk, we have Flash Memories. Random Access Memory is used for cache and primary memory and Read and Write access times are independent of the location being accessed. This means, you either access location 1 or you access the last location or the middle location, the access time will be same. Read Only Memories are used as a part of primary memory to store some fixed data that is not required to be changed. Magnetic Disk uses direction of magnetization of tiny magnetic particles on a metallic surface to store the data and the access time vary depending on the location being accessed, and these are used in secondary memory. Now, Flash Memories are coming into market, which is replacing this magnetic disk as secondary memory. They are much faster and smaller in size, and they do not have any movable part.

(Refer Slide Time: 32:28)



Now, these are the pictures showing these. The first picture shows the RAM primary memory. Next one is the ROM. This is a hard disk. If you open a hard disk it looks like this, this one is the SSD and so on.

(Refer Slide Time: 32:53)

Input Unit

- Used to feed data to the computer system from the external environment.
 - Data are transferred to the processor/memory after appropriate encoding.
- Common input devices:
 - Keyboard
 - Mouse
 - Joystick
 - Camera

NPTEL ONLINE
CERTIFICATION COURSES

NATIONAL INSTITUTE OF
TECHNOLOGY, MEGHALAYA

Coming to the input unit, it is used to feed data to the computer system. The commonly used devices are keyboards, mouse, joystick and camera.

(Refer Slide Time: 33:03)



These are the relevant pictures that are shown.

(Refer Slide Time: 33:15)

Output Unit

- Used to send the result of some computation to the outside world.
- Common output devices:
 - LCD/LED screen
 - Printer and Plotter
 - Speaker / Buzzer
 - Projection system

A collage of four output devices: a printer, a large LCD screen, a speaker, and a projector. Each device is enclosed in a red-bordered box.

And the output unit is used to send the result of some computation to outside world like printer is used to print the data, LCD screen or LED screen is used to see the output on the screen, you have speakers, you have projection system that are also used as an output unit.

(Refer Slide Time: 33:42)



So, these are some of the relevant pictures showing the output units. So, we have come to the end of lecture 1 where we have seen how computer systems have evolved over the years and what are the main functional components of a computer system, and how these components are required and how we can execute a particular instruction.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 02
Basic Operation Of A Computer

(Refer Slide Time: 00:27)

Introduction

- The basic mechanism through which an instruction gets executed shall be illustrated.
- May be recalled:
 - ALU contains a set of registers, some general-purpose and some special-purpose.
 - First we briefly explain the functions of the special-purpose registers before we look into some examples.

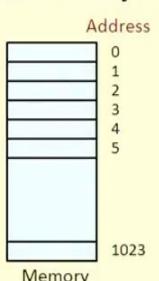
2

Welcome to the next lecture on basic operation of a computer. So, the basic mechanism through which an instruction gets executed shall be illustrated. And just recall that what we discussed in the previous lecture. Your ALU is having some registers, some registers are called special purpose registers and some are general purpose registers. And firstly, what we will do we will discuss some function of special purpose register.

(Refer Slide Time: 01:01)

For Interfacing with the Primary Memory

- Two special-purpose registers are used:
 - *Memory Address Register (MAR)*: Holds the address of the memory location to be accessed.
 - *Memory Data Register (MDR)*: Holds the data that is being written into memory, or will receive the data being read out from memory.
- Memory considered as a linear array of storage locations (bytes or words) each with unique address.



The diagram shows a vertical stack of 1024 memory cells, indexed from 0 at the top to 1023 at the bottom. The word 'Address' is written vertically above the first cell, and the word 'Memory' is written horizontally below the last cell.

 IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL
 NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

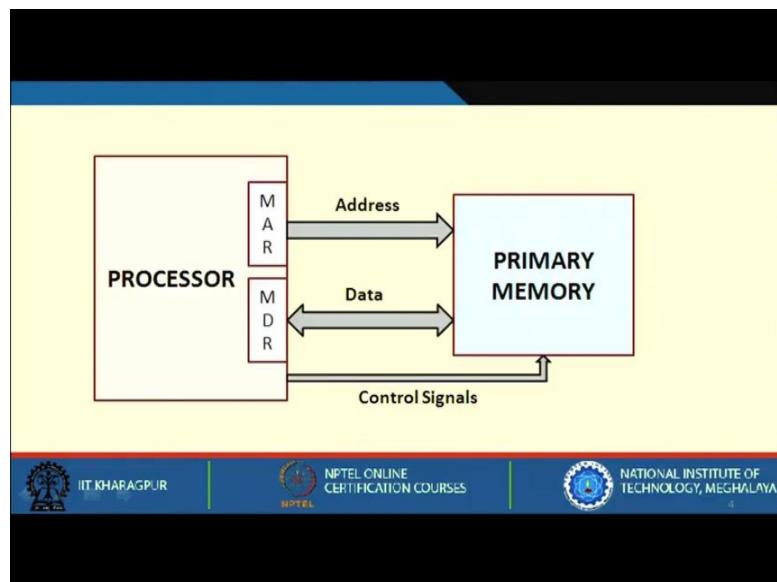
Instructions and data are stored in memory. And from memory you have to bring the data and the instruction to the processors and then you have to execute it. For this purpose, we require two special purpose registers they are called Memory Address Register(MAR) and Memory Data Register (MDR). So, let us see first what is memory address register. So, memory address register holds the address of a memory location to be accessed. So, when I say that it holds the address of memory location that is to be accessed that means, I can access a memory location for reading an instruction, I can access a memory location for reading a data and I can also access a memory location for writing back the data. So, memory address register holds the address of the instruction that is to be read or it holds the address of the data that is to be read from the memory or the address of the memory where the data is to be written.

The next register is called memory data register. Memory data register or MDR holds the data that is been written into memory or the data that, we will receive when read out from the memory location. So, when we write we always write a data into the memory, but when we read as I said, I can read an instruction or I can read a data. So, this memory data register will contain the instruction that is to be read from the memory or the data that is to be read from the memory or the data that is to be written into the memory, it can contain any of these three.

Now, you can see this that addresses it is spanning from 0 to 1023. So, the number of memory locations that we can address is starting from 0 to 1023, hence 1024 memory locations in total. So, a memory is considered as a linear array of storage locations, bytes

or words, each with a unique address. So, these are the addresses of the memory location where it is incremented one by one and it has got 1024 locations. Now, what I am trying to say is that this memory address register will hold one of such address that is either 0, 1, 4 or 5 or anything 1011 anything and the memory data register will hold the content of that location. If it is 5, whatever content will be there in that particular location that will be present in MDR.

(Refer Slide Time: 04:33)



Now, just see this diagram it shows the connection between processor and main memory. So, I talked about MAR and MDR. MAR is Memory Address Register and MDR is Memory Data Register. Memory Address Register is connected with primary memory through address bus. Memory Data Register is connected with primary memory through data bus. And some control signals are also required. So, why control signal are required. So, when I say that this particular data will be read from the memory or this particular data will be written into the memory, so we need to specify that information to the memory that from this particular location we need to read a data or from this particular location we need to write back data. Control signals are for this purpose.

First we provide the address in the address bus that hits to the primary memory then we provide the control signals either Read or Write depending on that, a particular value is read from that particular address and it comes to MDR. Or if I want to write a value, I have to put that value in MDR and the address where I want to write in MAR, and then I

activate the write control signal. Hence according to read or write a word is read or written from or into memory.

(Refer Slide Time: 06:40)

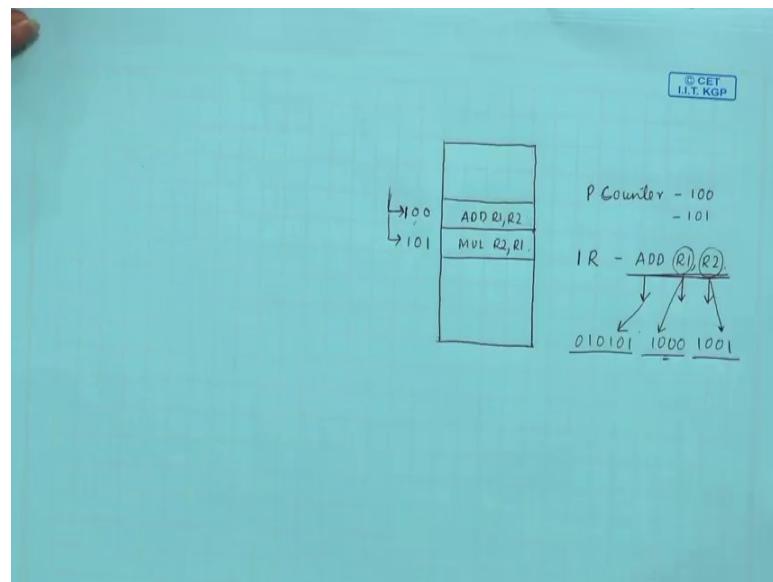
The slide contains two main bullet points:

- To read data from memory
 - a) Load the memory address into MAR.
 - b) Issue the control signal **READ**.
 - c) The data read from the memory is stored into MDR.
- To write data into memory
 - a) Load the memory address into MAR.
 - b) Load the data to be written into MDR.
 - c) Issue the control signal **WRITE**.

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the text "IIT KHARAGPUR", the NPTEL logo, and the text "NPTEL ONLINE CERTIFICATION COURSES". To the right of the footer, there is a small video frame showing a person speaking.

So, as I said this slide will summarize it. To read the data from memory, we first load the memory address into MAR then we issue the control signals i.e., read then the data from memory is read and it is stored into MDR. Now, to write into memory what we were doing as I said we have to load the memory address into MAR the data that is to be written must be loaded into MDR and then we issue write control signal. So, for reading these are the following steps that are required and for writing these are the following steps we have to issue.

(Refer Slide Time: 07:49)



Now, for keeping track of the program what I said like let us say this is my memory, these are some locations. So, this is say 100 location, this is 101 location and in these locations we have some instructions. Now, suppose you are executing at this particular location how will you move to the next location. So, for that reason there must be some counter that will point to location 100. So, you will go to location 100, you will fetch the instruction, execute the instruction. Then your counter should automatically get incremented to 101; such that after executing this instruction, you move to the next instruction and execute the other instruction.

(Refer Slide Time: 09:11)

For Keeping Track of Program / Instructions

- Two special-purpose registers are used:
 - Program Counter (PC)*: Holds the memory address of the next instruction to be executed.
 - Automatically incremented to point to the next instruction when an instruction is being executed.
 - Instruction Register (IR)*: Temporarily holds an instruction that has been fetched from memory.
 - Need to be decoded to find out the instruction type.
 - Also contains information about the location of the data.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

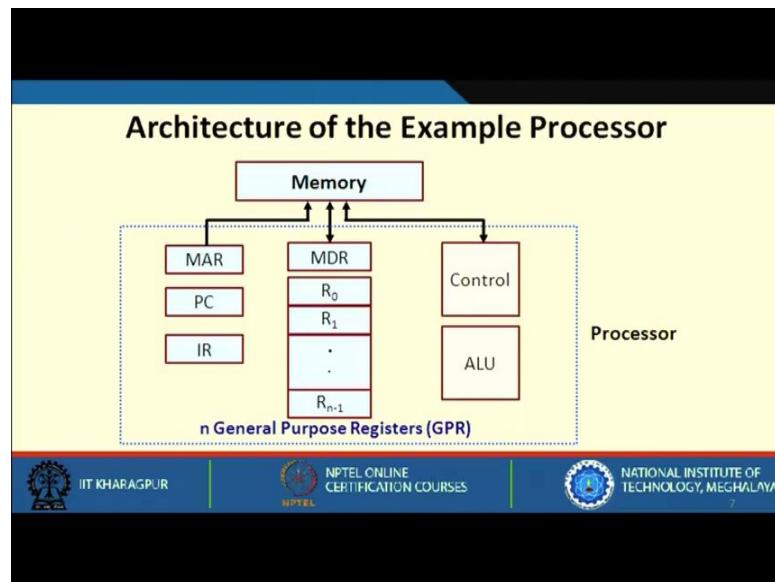
So, now we see for this purpose we have two special purpose registers. One is called Program Counter (PC) that holds the memory address of the next instruction to be executed, automatically it is incremented to point to the next instruction when an instruction is fetched and about to be executed. So, PC holds the memory address of the next instruction to be executed. So, once we read and execute an instruction, PC now must point to the next instruction that I have to execute next and so on.

Next, another register is the Instruction Register. So, now you see in this diagram, when I say that this is the location. So, PC will contain the location from where I have to read the instruction. Now, I have to read this instruction and store it somewhere, the register where it will get stored is known as Instruction Register where this entire instruction ADD R1,R2 is stored. So, this register Instruction Register temporarily holds an instruction that has been fetched from memory. Now, once an instruction is fetched from memory, you need to decode that instruction. See we understand this is add, but a computer is a layman. So, you have to instruct the computer do this, do that, then only the computer will do that.

So, when I say that ADD R1,R2. So, add is an instruction saying that add the content of register R1 with R2, and store back in R1. So, the instruction needs to be decoded and then the computer must understand, now it has to execute the instruction add the content of R1 and R2 and, store back the result in R1. So, the instruction register temporarily holds an instruction that has been fetched from memory, needs to decode to find out the instruction type, which contains information about the location of the data.

IR contains the entire instruction. So, after we get this instruction we need to decode to know this is add, and we also need to know locations of R1, R2 registers. So, ultimately after decoding we will understand this add R1, R2 is nothing but some bits of 0's and 1's. If a total of 16 registers are present then we need four bits to specify any one of the 16 registers and we can just name that R1 is a four bit register and it is represented by 1000. You can also say R2 is another register which is represented by 1001. And add is an opcode which is represented by some bits. So, ultimately this set of instruction is nothing but bits of zeros and ones. So, we will see this in near future.

(Refer Slide Time: 13:31)



Now, we can see the architecture of an example processor or in other words we can say that how memory is connected with the processor. So, this is a processor where we have some special purpose registers MAR, MDR, PC and IR you already know the functions of these registers. We have some general purpose registers and we have control unit and ALU. Now, what is the function of this ALU? When I say that ADD R1,R2, R1 and R2 are registers that are present inside your processor. So, these registers we need to bring to ALU so R1 must be brought to ALU and R2 must be brought to ALU and then the particular instruction like add or mul is executed on R1 and R2 and we get the result. So, this is an example processor how it looks like, but there will be many more things that we will see later.

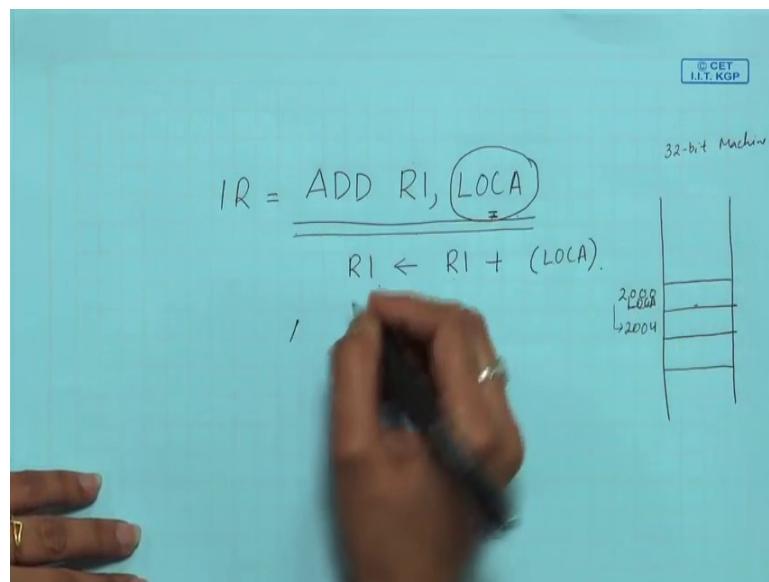
(Refer Slide Time: 15:02)

Example Instructions

- We shall illustrate the process of instruction execution with the help of the following two instructions:
 - ADD R1, LOCA**
Add the contents of memory location LOCA (i.e. address of the memory location is LOCA) to the contents of register R1.
 $R1 \leftarrow R1 + Mem[LOCA]$
 - ADD R1, R2**
Add the contents of register R2 to the contents of register R1.
 $R1 \leftarrow R1 + R2$

Now, we shall illustrate the process of instruction execution with the help of following two examples. So, the two examples that we will be taking the first one is ADD R1, LOCA.

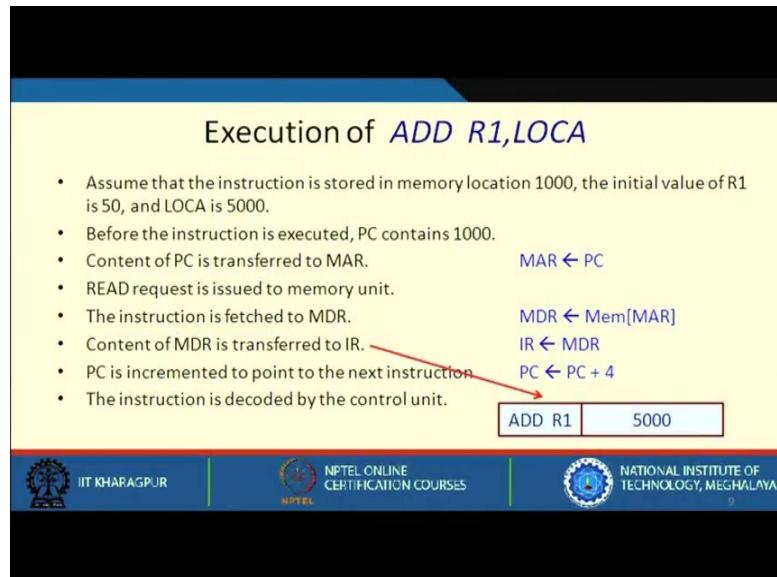
(Refer Slide Time: 15:24)



The instruction ADD R1, LOCA will add the content of R1 and location A and store back the result in R1. LOCA is a location in memory. So, this is your memory and this is some location. The content of this will be added with R 1 and will get stored back in R

- The next one is simply add the two registers that is R1 and R2 both are present in processor and store back the result in R1.

(Refer Slide Time: 16:45)



Now, let us see how we will execute this instruction. So, we have to do some assumption here. As I said that firstly, when we say this instruction, this instruction is stored in some memory location. We assume here that the instruction is stored in location 1000, and the initial value of R1 is 50, and LOCA value is 5000. Before the instruction is executed PC contains the value 1000. Now, the content of PC is transferred to MAR as we need to read the instruction. To read the instruction from memory what I said earlier that the address needs to be loaded in MAR, and we need to activate the read control. Hence 1000 now should be transferred to MAR, then a read request is issued to memory unit.

After the reading is performed, the instruction is in MDR; and then from MDR, it should be transferred to IR Instruction Register. And while doing these steps, we have to increment the PC to point to the next instruction. As I said in computer there will be sequential execution of instruction and the instructions are stored one by one. So, here first PC was pointing to 1000, we fetch the instruction from location 1000 and then the PC should point to the next location that is 1001. So, PC will now point to the next location which is 1001. And next whatever instruction that we have read from memory is loaded in MDR; and from MDR, it will be transferred to IR. And we know after it has

come to IR the instruction needs to be decoded and then it has to be executed. We will see step by step how it happens.

So, from this we can say that firstly, PC value is transferred to MAR. Read signal is activated. The content specified by the memory location(MAR) is read and it is stored in MDR. From MDR, it is transferred to IR and at the same time PC is incremented to 4. Why we have said here 4, it is depending on the word size, we will be coming to this little later, but it depends on what will be the word size. So, I will just explain it once. Let us consider a byte addressable. So, let us say if this location is 2000, and the next location is 2004 that means, we are using a 32-bit machine; and the PC is incremented by 4. If it is a 64-bit machine, the PC will get incremented by 8. So, here the PC is incremented by 4, so it is a 32-bit machine.

Now, once the instruction comes to IR it needs to be decoded and executed. Let us see how it is done. Now, location is for example, it is 5000 it is transferred from IR to MAR. So, if you consider the same instruction that is ADD R1, LOCA. Now, this is in IR. After decoding it has understood that one of the operand is in memory. If one of the operand is from memory then you have to read that particular operand the data which is there in that particular location from the memory.

(Refer Slide Time: 21:54)

- LOCA (i.e. 5000) is transferred (from IR) to MAR. $MAR \leftarrow IR[\text{Operand}]$
- READ request is issued to memory unit.
- The data is fetched to MDR. $MDR \leftarrow \text{Mem}[MAR]$
- The content of MDR is added to R1. $R1 \leftarrow R1 + MDR$

The steps being carried out are called micro-operations:

```
MAR ← PC
MDR ← Mem[MAR]
IR ← MDR
PC ← PC + 4
MAR ← IR[Operand]
MDR ← Mem[MAR]
R1 ← R1 + MDR
```

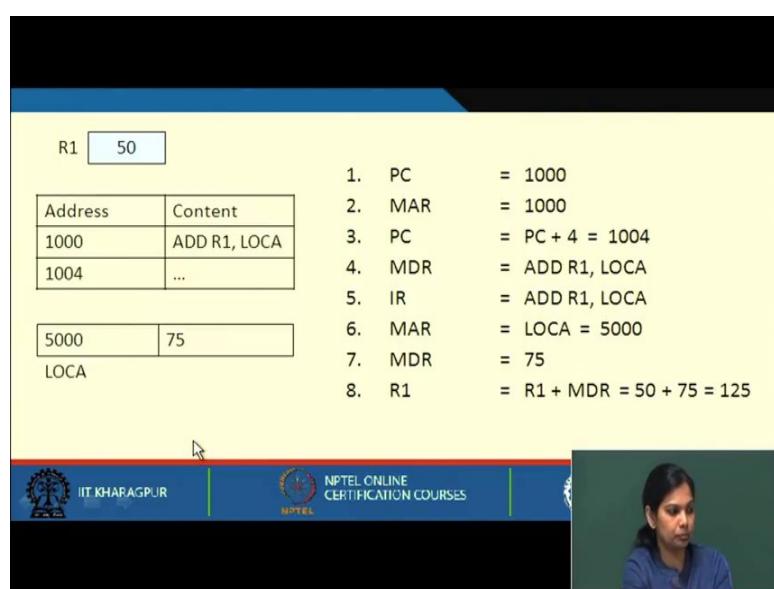
 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, you have to load that location into MAR, then activate the read control signal then the data that is present in that location LOCA that is 5000 will be read and will be

transferred to MDR. Now the MDR contains one of the operand and R1 contains one of the operand. These two will be brought into your ALU which will be added and stored back in R1. So, these are the steps that we follow to execute the instruction ADD R1, LOCA. In this instruction, how many read operation we have to do the first read operation we perform to fetch the instruction. And after decoding we again see that one of the operand is a memory location. So, we again have to read that particular location from the memory, we again read that particular data from that memory location, we execute the instructions and store back the result. So, two memory operations were required in this particular instruction.

Let us move on with the next one. So, these are the steps that are being carried out and these are called micro operation. We will see in more detail later, but just to see first PC value is transferred to MAR. Then whatever value of MAR is read from memory and it is stored in MDR. Now, from MDR the value is moved to IR. The PC gets incremented to point to the next memory location. If we see that there are some operands that is to be read from memory, then it has to be read and then transferred to MAR. And then read signal is activated and data is read and is brought into MDR. And finally, now all my data are present in processor both R1 is present in processor after reading MDR the data of LOCA is also present in MDR. So, both are processor register we need to add this and store the result in R 1.

(Refer Slide Time: 24:32)



So, whatever I have explained is that PC contain this value, MAR contain this value. PC get incremented it points to 1004 then MDR initially have the instruction ADD R1, LOCA that is moved to IR. Then after decoding LOCA value will be in MAR the data will be read and in that location the value is 75 which is stored in MDR. And finally, after adding this the value is stored back in register R 1.

(Refer Slide Time: 25:14)

Execution of *ADD R1,R2*

- Assume that the instruction is stored in memory location 1500, the initial value of R1 is 50, and R2 is 200.
- Before the instruction is executed, PC contains 1500.
- Content of PC is transferred to MAR.
- READ request is issued to memory unit.
- The instruction is fetched to MDR.
- Content of MDR is transferred to IR.
- PC is incremented to point to the next instruction.
- The instruction is decoded by the control unit.
- R2 is added to R1.

ADD R1, R2

MAR \leftarrow PC

MDR \leftarrow Mem[MAR]

IR \leftarrow MDR

PC \leftarrow PC + 4

R1 \leftarrow R1 + R2

The slide has a blue header bar with the title 'Execution of *ADD R1,R2*'. Below the title is a list of steps with corresponding assembly-like assignments to the right. A red arrow points from the 'ADD R1, R2' label to the 'IR \leftarrow MDR' assignment. At the bottom, there is a footer with the IIT Kharagpur logo, NPTEL logo, and text 'NPTEL ONLINE CERTIFICATION COURSES'. To the right of the footer is a small video window showing a person speaking.

In a similar way, let us see execution of another instruction that is ADD R1,R2. Here you can see that in this instruction both R1 and R2 are processor register. So, you need not have to bring anything from memory. So, both the operands are present in your processor register; all you need to do is that you have to read this particular instruction from the memory, and then you will execute this instruction. So, assume that the instruction is stored in memory location 1500, the initial value of R1 and R2 are 50 and 200. Before the execution, PC contains 1500 and then the content of PC is transferred to MAR. We activate the read control signal the instruction is fetched and it is stored in MDR. Then the content of MDR is transferred to IR, it is decoded. And then finally, the content of R1 and R2 that is 50 and 200 are added and stored back in R1.

(Refer Slide Time: 26:34)

The slide displays the following assembly language steps:

Address	Instruction
1500	ADD R1, R2
1504	...

Step-by-step execution:

1. PC = 1500
2. MAR = 1500
3. PC = PC + 4 = 1504
4. MDR = ADD R1, R2
5. IR = ADD R1, R2
6. R1 = R1 + R2 = 250

At the bottom, there are logos for IIT Kharagpur, NPTEL, and NIT Meghalaya.

So, these are the steps that are shown. PC will contain 1500 then MAR will also contain that. PC will get incremented by 4. And then the entire instruction is brought into MDR. IR will also contain this instruction, this instruction will now get decoded and the value of R1 and R2 will be added, and will be stored back in R1 that is 250.

(Refer Slide Time: 27:39)

Bus Architecture

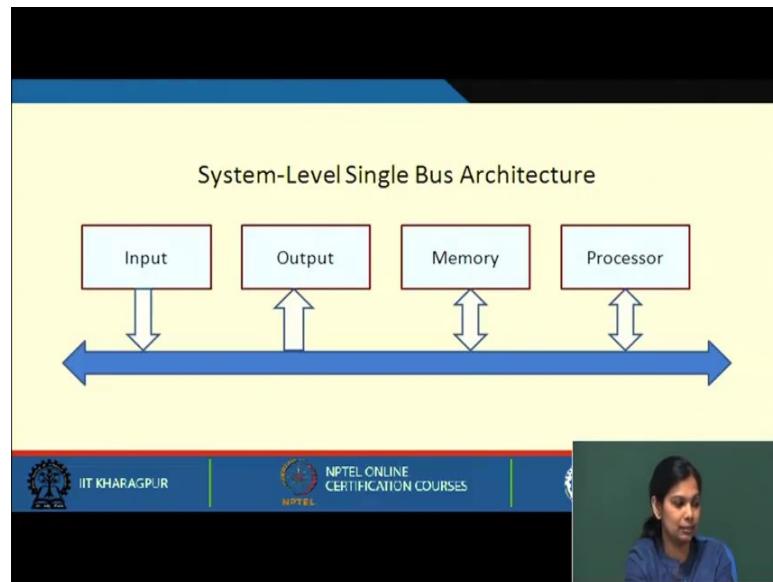
- The different functional modules must be connected in an organized manner to form an operational system.
- Bus refers to a group of lines that serves as a connecting path for several devices.
- The simplest way to connect the functional unit is to use the single bus architecture.
 - Only one data transfer allowed in one clock cycle.
 - For multi-bus architecture, parallelism in data transfer is allowed.

At the bottom, there are logos for IIT Kharagpur, NPTEL, and NIT Meghalaya.

Now, coming to the bus architecture. So, we were discussing about how an instruction will get executed. When we say that processor is a module, memory is a module, input output is a module, so all these are communicating between each other. So, they need a

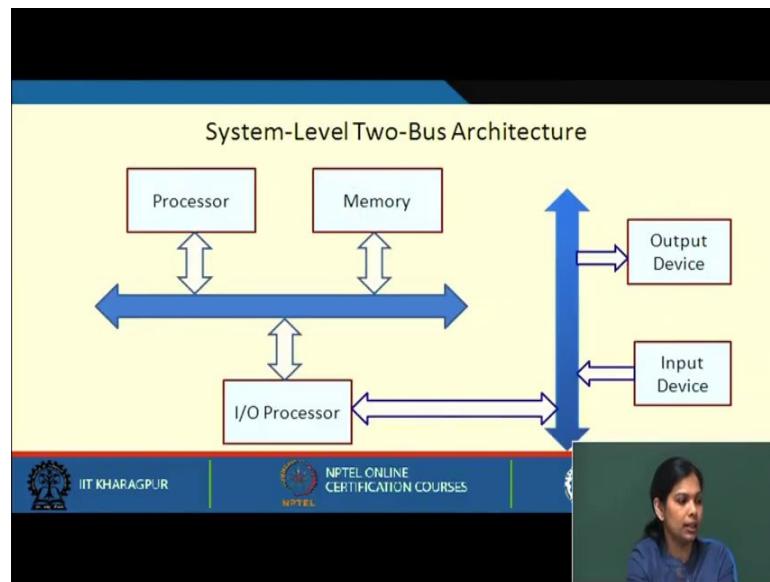
communicating pathway to communicate between each of these functional units. So, the different functional modules must be connected in an organized manner to form an operational system. By bus what we refer is to a group of lines that serve as a connecting path for several devices. So, the simplest way to connect all these is through a single bus architecture, where only one data transfer will be allowed in one cycle. For multi bus architecture parallelism in data transfer is allowed.

(Refer Slide Time: 28:15)



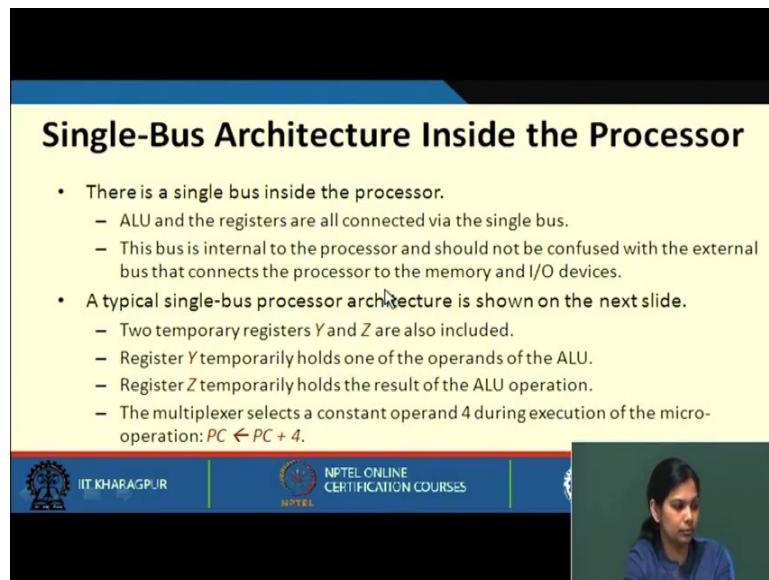
So, let us see this. So, this is a single bus that we can see where all our modules, memory, processor, input and output are connected to a single bus. So, if the processor wants to communicate with memory, it has to use this bus and no other modules will be using that at that moment. In the same way if from memory something needs to be sent to output device, no other module can communicate. This is a bottleneck of a single level system bus.

(Refer Slide Time: 28:54)



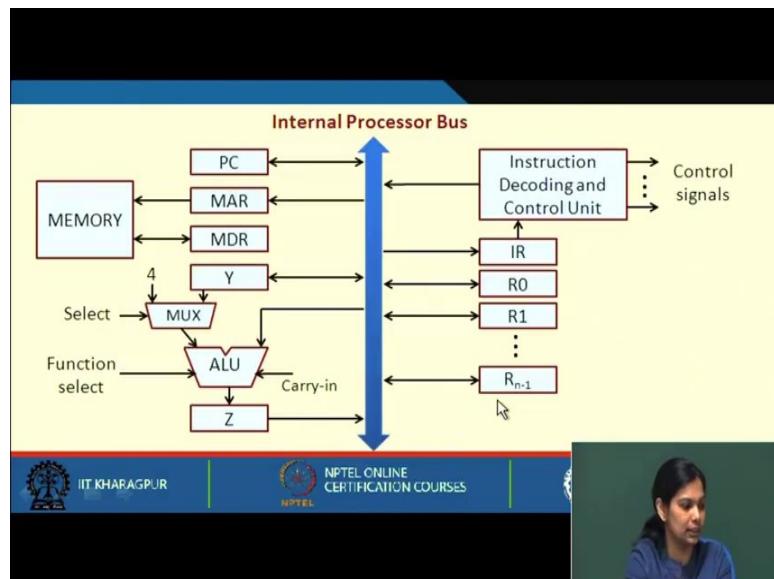
Now, coming to the system-level two-bus architecture. In this two-bus architecture, what we can see is that there is a bus dedicated to processor memory and of course, the IO processor is also connected to it. But for input and output there is a separate bus and this bus will be communicating with the IO processor in turn and the IO processor will then be communicating with the processor or the memory as and when it is required. So, when there is more communication between processor and memory then we must have a bus dedicated for this.

(Refer Slide Time: 29:43)



Now we can also see a bus that is required within the processor. Within the processor, we have seen that there are many components many transfers are taking place. So, whenever we are moving some register value within the processor, we also need a bus within the processor. So, in that bus, there are other components like, ALU and the registers that are all connected via this single bus. The bus is basically internal to the processor. So, I am talking about a bus which is inside the processor. Typical single bus processor architecture is shown in the next slide.

(Refer Slide Time: 30:49)



This is an internal processor bus. So, all these things are inside the processor PC, MAR, MDR, Y register is there and then ALU is there. There are some temporary registers Z, Y and there are some general purpose registers; IR is also there and there is a instruction decoding and control unit. So, information is getting transferred within all these modules. If I have to transfer a data from R1 to R2 or I have to transfer a data from PC to MAR or have to transfer a data from MDR to R1 or R2 or to ALU, some control signals needs to be generated. So, how they will be communicating, they will be communicating through this internal processes bus.

(Refer Slide Time: 31:44)

Multi-Bus Architectures

- Modern processors have multiple buses that connect the registers and other functional units.
 - Allows multiple data transfer micro-operations to be executed in the same clock cycle.
 - Results in overall faster instruction execution.
- Also advantageous to have multiple shorter buses rather than a single long bus.
 - Smaller parasitic capacitance, and hence smaller delay.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Now, multi-bus architectures are also there. So, modern processors use this multi-bus architecture. Here, within your processor to communicate between various registers you have multiple-bus. The advantage we get is that more operations can be performed and we get results much faster. So, there will be a overall improvement in instruction execution time. Some smaller parasitic capacitance can be there and hence smaller delay.

So, we have come to the end of lecture 2. In lecture 2, what we have studied how an instruction actually get executed, various processor registers, various registers that it present inside the processor. And finally, how we can increase the performance using either single-bus architecture or multi-bus architecture.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 03
Memory Addressing and Languages

(Refer Slide Time: 00:29)

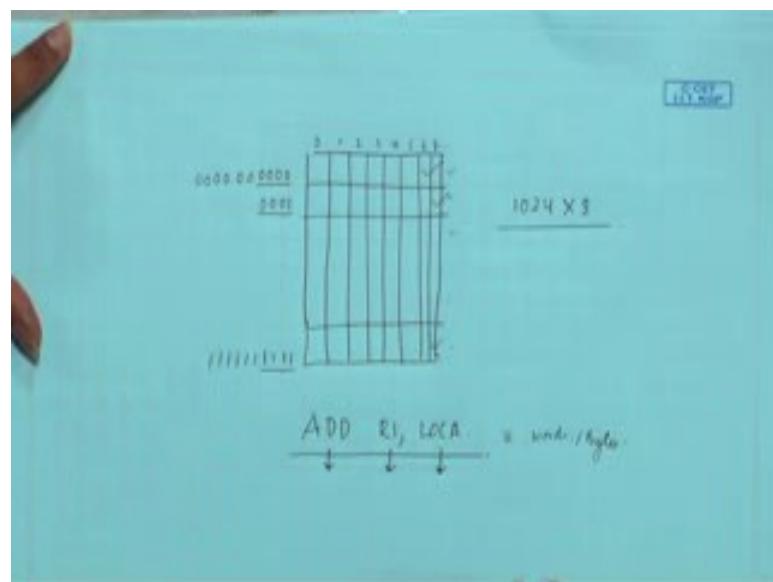
The slide has a blue header bar with the text 'Lecture - 03' and 'Memory Addressing and Languages'. Below this is a yellow section titled 'Overview of Memory Organization'. This section contains a bulleted list of points about memory organization:

- Memory is one of the most important sub-systems of a computer that determines the overall performance.
- Conceptual view of memory:
 - Array of storage locations, with each storage location having a unique address.
 - Each storage location can hold a fixed amount of information (multiple of bits, which is the basic unit of data storage).
- A memory system with M locations and N bits per location, is referred to as an $M \times N$ memory.
 - Both M and N are typically some powers of 2.
 - Example: 1024 x 8, 65536 x 32, etc.

At the bottom of the slide, there is a footer bar with three logos and text: IIT Kharagpur, NITEL Online Certification Courses, and National Institute of Technology, Meghalaya.

Welcome to the third lecture on memory addressing and languages. So, let us know about the overview of memory organization. What is memory? Memory is one of the most important subsystems of a computer that determines the overall performance. What do you mean by that? See as you have seen in the previous lecture that we are storing the instruction and data in the memory. If your memory is slower then loading the data from the memory will be slower. So, in that case we need to have a good speed memory. The conceptual view of memory is it is an array of storage locations with each storage location having a unique address. So, it is an array of memory locations.

(Refer Slide Time: 01:30)



So, as I said it is an array of memory locations. So, we have first location as 0 0 0 0, next location as 0 0 0 1 and so on, maybe the last location is 1 1 1 1. So, it is an array of storage location each with a unique address. So, these are individual locations and this is the address associated with each location. And each storage location can hold a fixed amount of information, which can be multiple of bits which is the basic unit of data storage. A memory system with M locations and N bits per location is referred to as an $M \times N$ memory, where both M and N are typically some powers of 2. An example: 1024 \times 8.

So, if I say 1024 \times 8, it means we have 10-bit in the address, and each location is having 8-bit. So, you can store these many locations in these many locations; this shows how a memory will look like.

(Refer Slide Time: 03:53)

Some Terminologies

- Bit: A single binary digit (0 or 1).
- Nibble: Collection of 4 bits.
- Byte: Collection of 8 bits.
- Word: Does not have a unique definition.
 - Varies from one computer to another; typically 32 or 64 bits.

IT Kharagpur | NETEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, some terminologies you must know when we talk about memory. What is a bit, we all know a bit is a single binary digit either 0 or 1. Nibble is a collection of 4 bits. Byte is a collection of 8 bits. And word does not have a unique definition because we can either have a 32 bit word length or 64 bit word length. So, word does not have a unique definition.

(Refer Slide Time: 04:29)

How is Memory Organized?

- Memory is often byte organized.
 - Every byte of the memory has a unique address.
- Multiple bytes of data can be accessed by an instruction.
 - Example: *Half-word* (2 bytes), *Word* (4 bytes), *Long Word* (8 bytes).
- For higher data transfer rate, memory is often organized such that multiple bytes can be read or written simultaneously.
 - Necessary to bridge the processor-memory speed gap.
 - Shall be discussed later in detail.

IT Kharagpur | NETEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

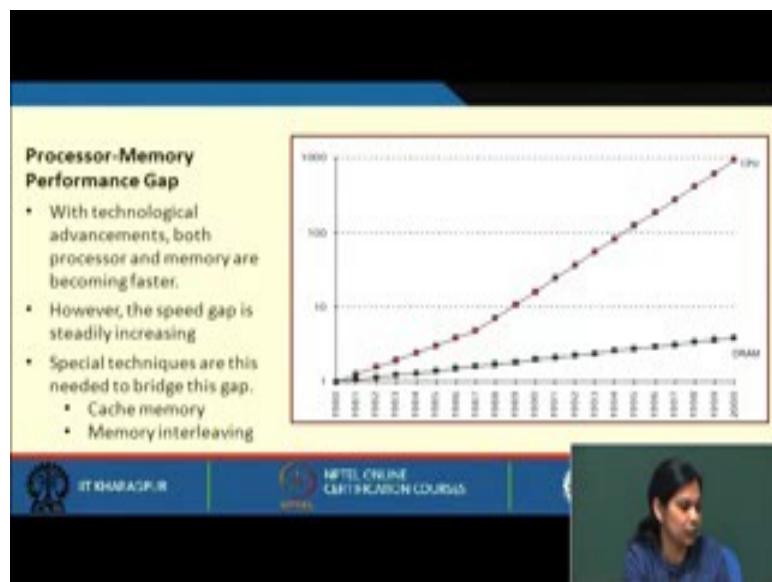
Now let us see how is memory organized. Memory is often byte organized. So, we never say that each bit is having an address, rather we say each byte is having an address, that

means every byte of the memory has a unique address. And multiple bytes of a data can be accessed by an instruction. I will just take an example: ADD R1,LOCA. So, if you consider this instruction, it is depending on how many bits this ADD will have, how many bits this register will have, and how many bits this location will have; this will define that how many words this instruction will have or how many bytes this instruction will have.

So, in that sense what I am trying to say is that how many bytes this instruction will take is dependent on various other factors like the total number of instructions available in your computer. The total number of registers present in your computer, and also the number of locations you are having based on which you can determine the number of bytes required to represent this particular instruction.

For higher data transfer rate, memory is often organized such that multiple bytes can be read or written simultaneously. This is basically needed to bridge the processor memory speed gap; we shall of course discuss this later, but I will just tell very briefly about this memory processor speed gap. So, as you know that processor speed is increasing memory speed is also increasing, but not at this pace the processor is increasing.

(Refer Slide Time: 07:06)



So, this picture will show you the processor memory performance gap. See with technological advancement both processor speed is increasing and also memory speed is increasing; however, there is a speed gap which is steadily increasing. So, earlier the

speed gap was much less, but now with technological advancement CPU speed has increased to a greater extent; memory speed has also increased, but not at the same pace as the CPU. So, we can see this.

So, some special techniques are used to bridge this gap. We will see this in the memory module design, where the concept of cache memory and memory interleaving will be talked about, but from this we can see what we can say is that there is still a huge gap between the speed of a processor and speed of your memory. So, this is where we have to agree upon that. We are still in the phase we are growing we are trying to make certain techniques to bridge this gap, but still this gap exists.

(Refer Slide Time: 08:40)

Unit	Bytes	In Decimal
8 bits (B)	1 or 2^0	10^0
Kilobyte (KB)	1024 or 2^{10}	10^3
Megabyte (MB)	1,048,576 or 2^{20}	10^6
Gigabyte (GB)	1,073,741,824 or 2^{30}	10^9
Terabyte (TB)	1,099,511,627,776 or 2^{40}	10^{12}
Petabyte (PB)	2^{50}	10^{15}
Exabyte (EB)	2^{60}	10^{18}
Zettabyte (ZB)	2^{70}	10^{21}

Now, how do you specify memory sizes. Memory sizes can be 8 bit which is a byte. It can be kilobyte 2^{10} ; it can be megabyte 2^{20} ; gigabyte 2^{30} ; terabytes 2^{40} , and many more like petabyte, exabyte and zettabyte.

(Refer Slide Time: 09:09)

The slide features a bulleted list of facts about memory storage:

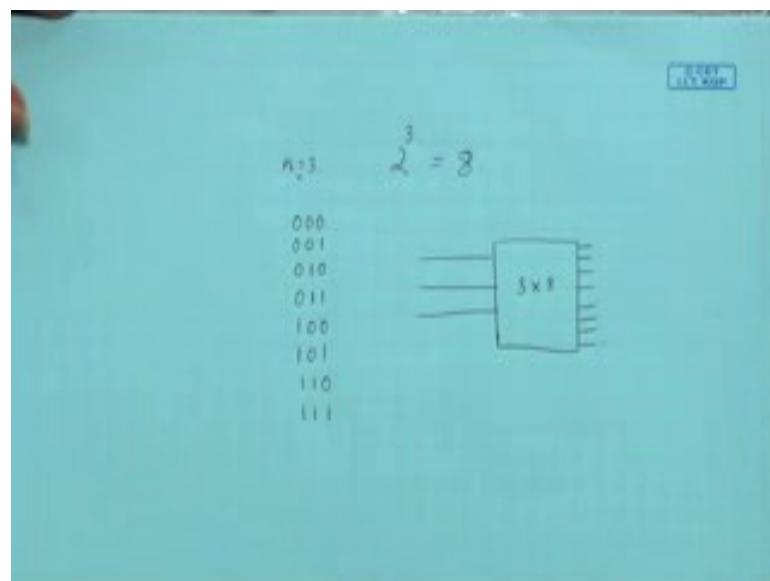
- If there are n bits in the address, the maximum number of storage locations can be 2^n .
 - For $n=8$, 256 locations.
 - For $n=16$, 64K locations.
 - For $n=20$, 1M locations.
 - For $n=32$, 4G locations.
- Modern-day memory chips can store several Gigabits of data.
 - Dynamic RAM (DRAM).

Below the list is a diagram illustrating the interface between a computer system and memory. It shows a central box labeled "MEMORY". Two double-headed arrows point to it: one from "Address (n bits)" and another from "Data (m bits)". Three single-headed arrows point to the "MEMORY" box from below: "RD", "WR", and "EN".

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo and the text "NETEL ONLINE COURSE" and "CARTIK KARAN CHOURAS". To the right of the footer, a portion of a person's face is visible.

Now, you see if there are n bits in an address, the maximum number of storage locations that can be accessed is 2^n .

(Refer Slide Time: 09:31)



This is a small example, we will take $n = 3$. So we can say how many locations we can access; $2^3 = 8$. So, the first location will be 0 0 0, the next location will be 0 0 1, next will be 0 1 0, 0 1 1, 1 0 0, 1 0 1, 1 1 0 and 1 1 1. So, with n bits we can have 2^n locations that can be accessed.

So, if we have 3 bit in the address, so maximum location that can be accessed is 8. So, for $n = 8$, 256 locations (2^8); for $n = 16$, $2^{16} = 64K$ locations; for $n = 20$, 1M locations, etc. can be accessed. So, this diagram shows the address bits; if you have n bit address we can have 2 to the power n locations that can be accessed. And modern-day memory chips can store several gigabytes of data that is our dynamic RAM. We will be looking into more details about each and every aspect of memory module.

(Refer Slide Time: 11:33)

Address	Contents
0000 0000	0000 0000 0000 0001
0000 0001	0000 0100 0101 0000
0000 0010	1010 1000 0000 0000
⋮	⋮
1111 1111	1011 0000 0000 1010

An example: $2^8 \times 16$ memory

Now, as I said for an 8 bit address, 2 to the power 8 unique locations will be there. The first locations will be all 0s, and the last location will be all 1s; and each of these locations again will have some content. So, consider an example of $2^8 \times 16$ memory. So, in each of these locations we will have some data which is 16 bits.

(Refer Slide Time: 12:02)

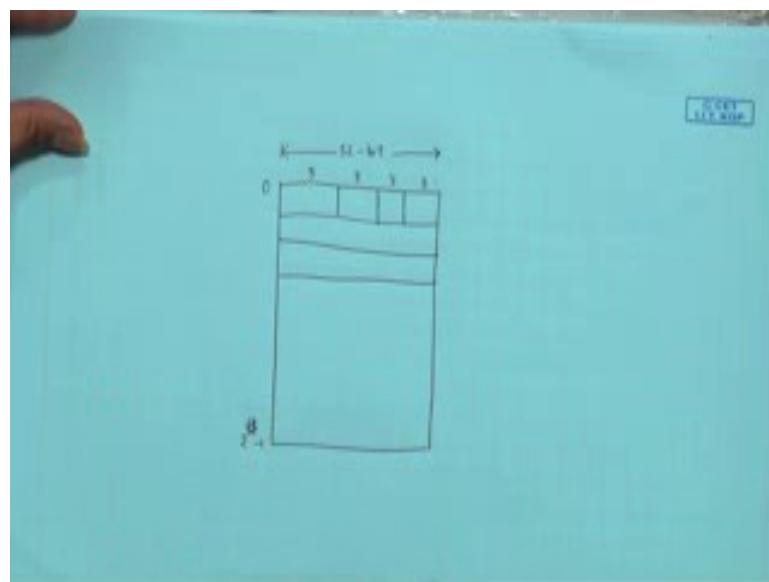
Some Examples

1. A computer has 64 MB (megabytes) of byte-addressable memory. How many bits are needed in the memory address?
 - Address Space = 64 MB = $2^6 \times 2^{20}$ B = 2^{26} B
 - If the memory is byte addressable, we need 26 bits of address.
2. A computer has 1 GB of memory. Each word in this computer is 32 bits. How many bits are needed to address any single word in memory?
 - Address Space = 1 GB = 2^{30} B
 - 1 word = 32 bits = 4 B
 - We have $2^{30} / 4 = 2^{28}$ words
 - Thus, we require 28 bits to address each word.

NETEL ONLINE
CARTI KAROON COURSES
NETEL

Let us see a computer with 64 MB of byte addressable memory. How many bits are needed in the memory address? As I already said, that $64\text{ MB} = 2^{26}$; that is, we need 26 bits to represent the address. Now, let us take another example where we say a computer has 1 GB of memory. So, we are saying total of 1 GB of memory, each word in this computer is 32 bit.

(Refer Slide Time: 13:18)



So, $1 \text{ GB} = 2^{30}$. If each word is 32 bits, that means 8, 8, 8, 8. So, total words possible will be $2^{30} / 4 = 2^{28}$. So, we require 28 bits, with address from 0 to $2^{28}-1$. If it is byte addressable, each byte can be accessed with address from 0 to $2^{30}-1$.

(Refer Slide Time: 15:02)

Byte Ordering Conventions

- Many data items require multiple bytes for storage.
- Different computers use different data ordering conventions.
 - Low-order byte first
 - High-order byte first
- Thus a 16-bit number 11001100 10101010 can be stored as either:

Data Type	Size (in Bytes)
Character	1
Integer	4
Long integer	8
Floating-point	4
Double-precision	8

Typical data sizes

11001100 10101010 or 10101010 11001100

IIT KHARAGPUR
 NETEL ONLINE COURSES
 NATIONAL INSTITUTE OF TECHNOLOGY MACHILIPATNA

Now, let us also understand what is byte ordering convention. Many data items require multiple bytes for storage. And different computers use different data ordering convention, it is known as low order byte first and high order byte first. So, these two are called basically Little Endian and Big Endian. So, you see this data type character is 1 byte, integer is 4 byte, long integer is 8, floating point 4 and double precision is 8. Thus if you have a 16 bit number which is represented like this, so in one way this is the total number high order bit is stored in high order address, and the low order is stored here and so on here it is stored differently. This is stored first and this is stored next.

(Refer Slide Time: 16:13)

The slide has a yellow header bar with the title 'Little Endian'. Below it is a white content area containing a bulleted list. The list includes a general statement about conventions, followed by two sub-points: 'a) Little Endian' and 'b) Big Endian'. Each sub-point has its own bullet list describing the convention. At the bottom of the slide, there is a blue footer bar with the text 'IT KHANAGPUR' and 'NETEL ONLINE CERTIFICATION COURSES'.

- The two conventions have been named as:
 - a) Little Endian
 - The least significant byte is stored at lower address followed by the most significant byte. Examples: Intel processors, DEC alpha, etc.
 - Same concept followed for arbitrary multi-byte data.
 - b) Big Endian
 - The most significant byte is stored at lower address followed by the least significant byte. Examples: IBM's 370 mainframes, Motorola microprocessors, TCP/IP, etc.
 - Same concept followed for arbitrary multi-byte data.

So, let us see the convention that has been named as little endian. The least significant byte is stored at lower address followed by most significant byte. So, Intel processors DEC alpha they all use little endian method, where again I repeat least significant byte is stored at lower address. Now, same concept follows for arbitrary multi-byte. So, if you also want to store multi byte then the same thing will follow there also. Now, in big endian the most significant byte is stored at lower address followed by least significant byte. So, the most significant byte is stored at lower address followed by least significant byte. IBM's 370 mainframe uses big endian concept.

(Refer Slide Time: 17:14)

The slide has a yellow header bar with the title 'An Example'. Below it is a white content area containing a bulleted list. The list asks to represent a 32-bit number in both Little-Endian and Big-Endian formats starting from address 2000 onwards. Below the list is the binary number '01010101 00110011 00001111 11000011'. A table follows, comparing Little-Endian and Big-Endian representations across four memory addresses (2000 to 2003). The table has two columns: 'Little Endian' and 'Big Endian'. In the Little-Endian column, the bytes are stored in reverse order of their addresses. In the Big-Endian column, the bytes are stored in the order of their addresses. At the bottom of the slide, there is a blue footer bar with the text 'IT KHANAGPUR' and 'NETEL ONLINE CERTIFICATION COURSES'.

Little Endian		Big Endian	
Address	Data	Address	Data
2000	11000011	2000	01010101
2001	00001111	2001	00110011
2002	00110011	2002	00001111
2003	01010101	2003	11000011

Now, let us see this representation with this example of a 32-bit number both in little endian and big endian. So, this is the number and lower byte is stored in lower address, then the next byte, then the next byte, and then the next byte. And here the higher order address higher order byte is stored lower address, then the next byte then the next byte and then this one. So, just see the difference between these two in little endian what we are storing; this is the least significant byte, we are storing in the least address that is 2000 and then the next one, next one, next one. In a similar way, the most significant byte we are storing it in the lower address and so on.

(Refer Slide Time: 18:23)

Memory Access by Instructions

- The program instructions and data are stored in memory.
 - In von-Neumann architecture, they are stored in the same memory.
 - In Harvard architecture, they are stored in different memories.
- For executing the program, two basic operations are required.
 - Load:** The contents of a specified memory location is read into a processor register.
LOAD R1, 2000
 - Store:** The contents of a processor register is written into a specified memory location.
STORE 2020, R3

Let us see memory access by instructions. Now, as I said that the program and instruction in the program instructions and the data are stored in memory. So, there are two basic ways how this can be stored. One is von-Neumann architecture, they are stored same in the same memory both the program and the data are stored in same memory. In Harvard architecture, they are stored in different memories.

So, for executing the program two basic operations are required. What are the two basic operation we need to know this. Load the content of specified memory location. So, LOAD R1,2000 means load the content from memory location 2000 into processor register R1. Another one is STORE R3,2010 that means store the content of R3 into this specific location 2020. So, either we load a data from memory into one of the processor registers, or we store the data from processor register to some location in memory.

(Refer Slide Time: 20:11)

An Example

- Compute $S = (A + B) - (C - D)$

```
LOAD R1,A  
LOAD R2,B  
ADD R3,R1,R2      // R3 = A + B  
LOAD R1,C  
LOAD R2,D  
SUB R4,R1,R2      // R4 = C - D  
SUB R3,R3,R4      // R3 = R3 - R4  
STORE S,R3
```

Now, let us take an example. Suppose we need to execute this particular instruction. We need to compute this. S, A, B, C and D are stored in memory. So, what we need to do is we need to load individual data, that is A and B and then only we can execute it. So, let us see what are the steps that are required to execute this. First of all let us do A + B. For this, I need to load first A into some processor register, B into some processor register and then add them. So, LOAD R1,A will load the content of A into R1; LOAD R2,B will load the content of B into R2. The ADD instruction will add the content of R1 and R2, and store the result in R3. So, this portion is done.

Next I load C into R1, D into R2, then I subtract C, I subtract D from C, and then I store the result in R4. Now, A + B is stored in R3, C - D is stored in R4, and now I need to subtract them. SUB R3,R3,R4 will cause the result to be stored in R3. But finally, I have to store the result in another location, that is S. So, what I should do now I have to store the content of R3 into S.

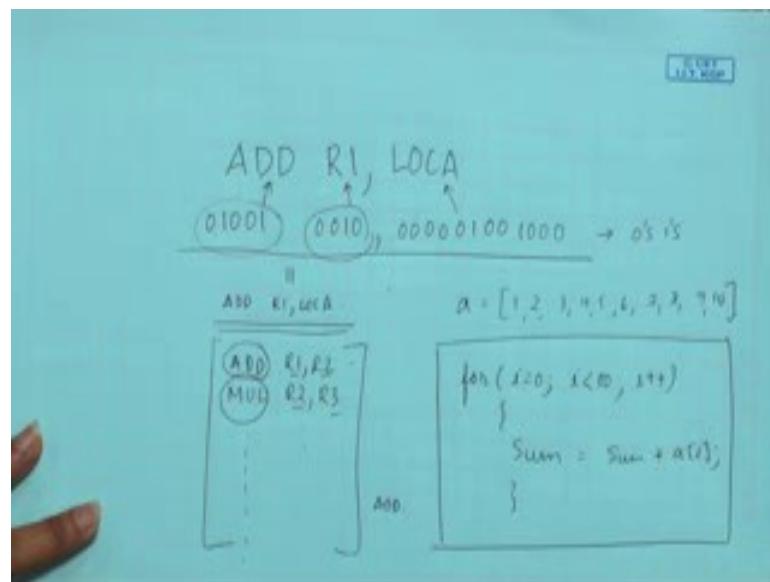
(Refer Slide Time: 22:58)

Machine, Assembly and High Level Language

- **Machine Language**
 - Native to a processor; executed directly by hardware.
 - Instructions consist of binary code: 1's and 0's.
- **Assembly Language**
 - Low-level symbolic version of machine language.
 - One to one correspondence with machine language.
 - Pseudo instructions are used that are much more readable and easy to use.
- **High-Level Language**
 - Programming languages like C, C++, Java.
 - More readable and closer to human languages.

Now, let us also understand what is machine language, assembly language, and high level language. Machine language is native to a processor and it is executed directly by in hardware. So, it only consists of binary code 1s and 0s.

(Refer Slide Time: 23:29)



So, I have talked about this if you remember that let say this is my instruction. Again I take the same example ADD R1,LOCA and as I said this can be 01001 (5-bit), this can be a 4-bit number, and this can be a 12-bit number. So, what I am specifying I am specifying the entire instruction in a sequence of 0s and 1s. So, when I specify an

instruction in the form of 0s and 1s, is called machine language, which is native to processor executed directly by hardware.

Next is assembly language. It is also a low-level symbolic version of machine language. Instead of 0s and 1s, I write it symbolically as ADD R1,LOCA. When I represent something symbolically; instead of writing 0s and 1s, I am writing it with some mnemonics.

So, this language is a low-level symbolic version of machine language; one to one correspondence with machine language is there. Pseudo instructions are used that are much more readable and easier to use. What do you mean by much more readable and easier to use? That means it will be difficult for me to remember 0110 is add, but it will be very much easy for me to remember the mnemonic ADD. So, I can write an instruction ADD R1,R2. So, this is much easier way to represent a program. So, assembly language is nothing but some kind of one-to-one correspondence with machine language.

Next comes to high-level language. Now, you see if you have to add ten numbers or you have to sort some list in an array, you need to perform certain operation. Again writing such kind of language where we say that first sum of 10 numbers you have to initialize it and then you have to repeatedly add it. So, there will be set of more instruction that are required to perform an add operation; rather for adding 10 numbers, I can very easily write let say for ($i=0; i<10; i++$), $sum = sum + a[i]$. So, this is much more easier to code.

So, generally high-level languages basically the programming languages like C or C++ or Java are used which are much more readable and closer to human languages. So, we can we can write a program in a high-level language and that can be executed by your machine.

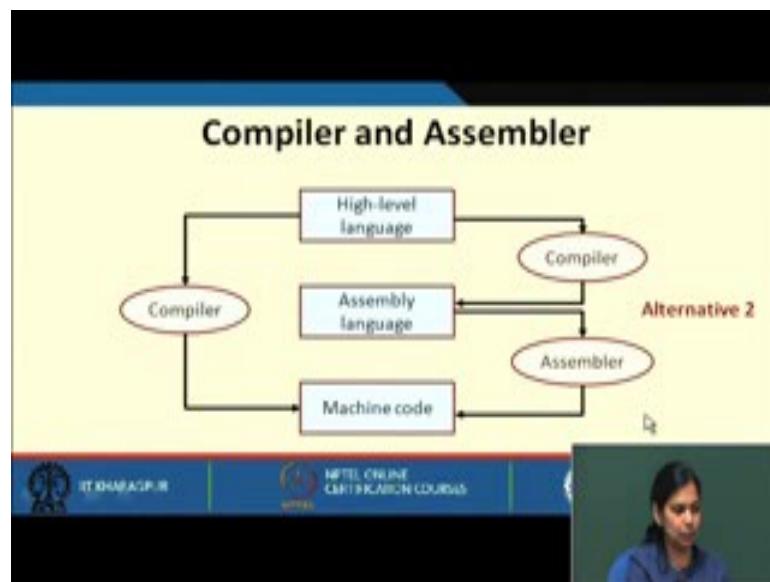
(Refer Slide Time: 29:04)

The slide has a dark blue header bar at the top. Below it is a yellow main content area with a black border. At the bottom is a dark blue footer bar containing three logos and their respective names: IIT Kharagpur, NIELIT Online Certification Courses, and National Institute of Technology, Meghalaya. The title 'Assemblers and Compilers' is centered in bold black font at the top of the yellow section. Below the title is a bulleted list:

- Assembler
 - Translates an assembly language program to machine language.
- Compiler
 - Translate a high-level language programs to assembly/machine language.
 - The translation is done by the compiler directly, or
 - The compiler first translates to assembly language and then the assembler converts it to machine code.

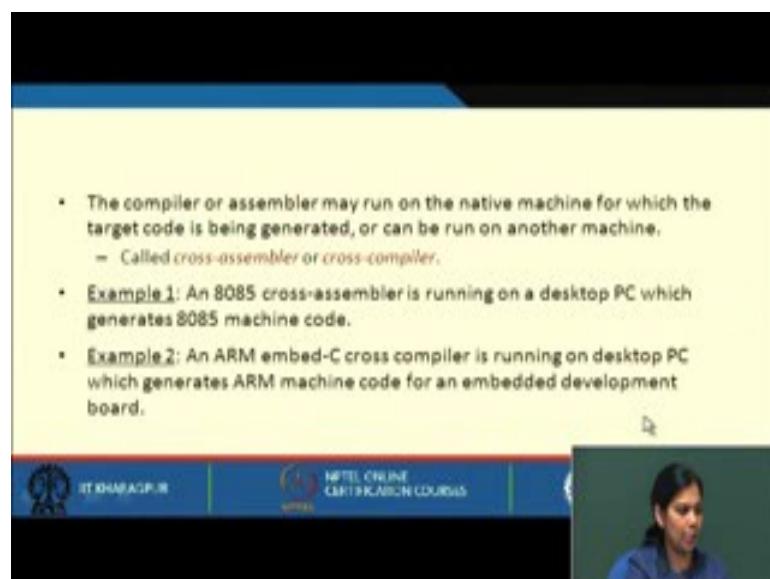
Now, if you have to execute a high-level language then you need some kind of translator that should translate your high-level language to assembly language or machine level language. So, these are the two things which are required, one is assembler that translates an assembly language program to machine language; and compiler that translates a high level language program to assembly or machine language. So, the translation is done by compiler directly or the compiler first translates to assembly language, and then an assembler can convert to a machine language, but ultimately you need to have machine language code to execute your instruction. So, any high-level language you write ultimately you have to boil down it to machine language that can only be understood by your computer.

(Refer Slide Time: 30:10)



So, as I said this compiler you have high-level language and then you can directly have a compiler that will generate an assembly language. And then that assembly language will be fed to an assembler that will generate a machine language or directly you can have a compiler that will generate your machine language, these are the best two alternatives that happens.

(Refer Slide Time: 30:39)



Now, we you can also have something called cross-assembler or cross-compiler. What it does like the compiler or assembler may run on a native machine for which the target

code is being generated or can run on any other machine that means. Take an example where you have an 8085 cross-assembler which is running in your desktop machine. And what it generates is some 8085 machine codes. Similarly, an ARM embed-C compiler which is available online may be running on a desktop PC which generates ARM machine code for the embedded development board.

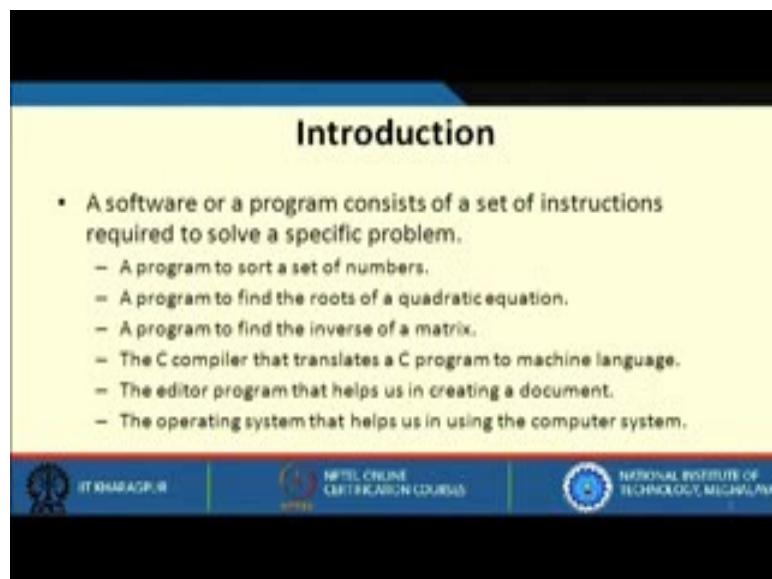
So, by this I will end lecture 3. So, in this lecture what we have discussed is how memory is organized, what are the ways to store the data in the memory. And we also looked into what are the steps that are required to transfer an instruction from one level to another like from high-level language, how it is converting into machine language through some compiler, how from assembly language it is converted into machine language and what is a machine language.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 04
Software and Architecture Type

(Refer Slide Time: 00:26)



Introduction

- A software or a program consists of a set of instructions required to solve a specific problem.
 - A program to sort a set of numbers.
 - A program to find the roots of a quadratic equation.
 - A program to find the inverse of a matrix.
 - The C compiler that translates a C program to machine language.
 - The editor program that helps us in creating a document.
 - The operating system that helps us in using the computer system.

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Welcome to the fourth lecture software and architecture types. A software or a program consist of a set of instructions required to solve a specific problem. So, by that what we mean like a program to sort ten numbers, a program to add some numbers or a program to find out inverse of a matrix or you say a compiler a C compiler that converts your high level language into some machine language. All of these are a kinds of software. So, software consists of a set of instructions. A set of instructions are provided to perform certain task. And the operating system also is an software that helps us in using the computer system.

(Refer Slide Time: 01:30)

Types of Programs

- Broadly we can classify programs/software into two types:
 - a) Application Software
 - Which helps the user to solve a particular user-level problem.
 - May need system software for execution.
 - b) System Software
 - A collection of programs that helps the users to create, analyze and run their programs.

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MANGALURU

So, what are the types of program that we have? Broadly we can classify the programs or you can say software into two types. First one is the application software which helps the user to solve a particular user level problem and it may require a system software for execution. Similarly, a system software is basically collection of many programs that helps the user to create analyze and run their programs. So, this is very important to know that we have two kinds of programs; one kind is application software, another kind is system software.

(Refer Slide Time: 02:26)

(a) Application Software

- Application software helps users solve particular problems.
- In most cases, application software resides on the computer's hard disk or removable storage media (DVD, USB drive, etc.).
- Typical examples:
 - Financial accounting package
 - Mathematical packages like MATLAB or MATHEMATICA
 - An app to book a cab
 - An app to monitor the health of a person

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MANGALURU

Now, coming in detail of application software. This helps the user to solve a particular problem like what do you mean by a particular problem. Let us say we want to have a financial accounting package, I want to do some kind of financial accounting stuffs. So, for that I need a very specific software for that purpose, specific to that particular application. What is the application the application here is the financial accounting. In a similar fashion, a mathematical package like MATLAB, which is used to perform a particular kind of mathematical operations and various programs can be written using that. So, those are specific to some mathematical operation.

Similarly, you think of an app we have in our mobile phones to call a cab that is also an application, but what that application is doing that particular application is helping us to call a cab. In a same way, there can be various apps to monitor your health, there can be various apps to do various other functions. So, all these comes under application software.

(Refer Slide Time: 04:01)

(b) System Software

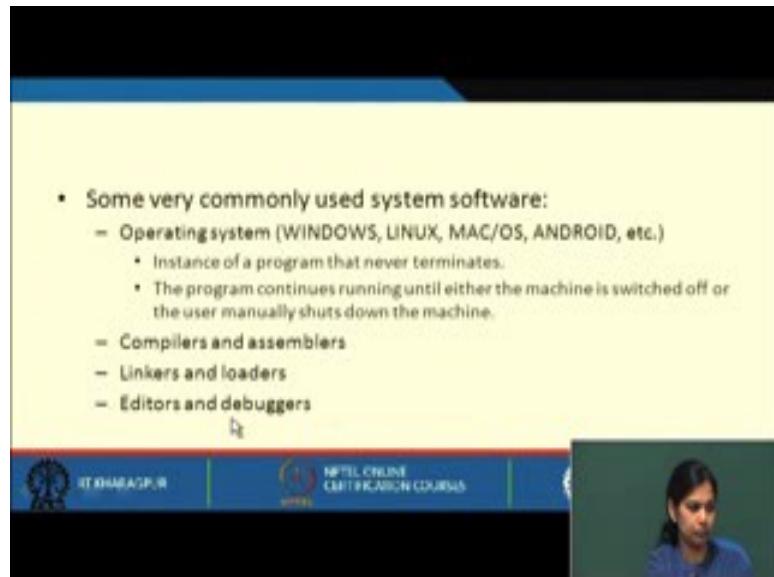
- System software is a collection of programs, which helps users run other programs.
- Typical operations carried out by system software:
 - Handling user requests
 - Managing application programs and storing them as files
 - File management in secondary storage devices
 - Running standard applications such as word processor, internet browser, etc.
 - Managing I/O units
 - Program translation from source form to object form
 - Linking and running user programs

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, coming to system software. A system software is a collection of programs which helps user run other programs. So, a system software is also a collection of program, but it also helps other user to run a particular program. So, let us see some typical operations that are carried out by a system software. What it does, it handles user request, it manages application programs and storing them as files, it also does file management in secondary storage that is very important, running standard applications such as word

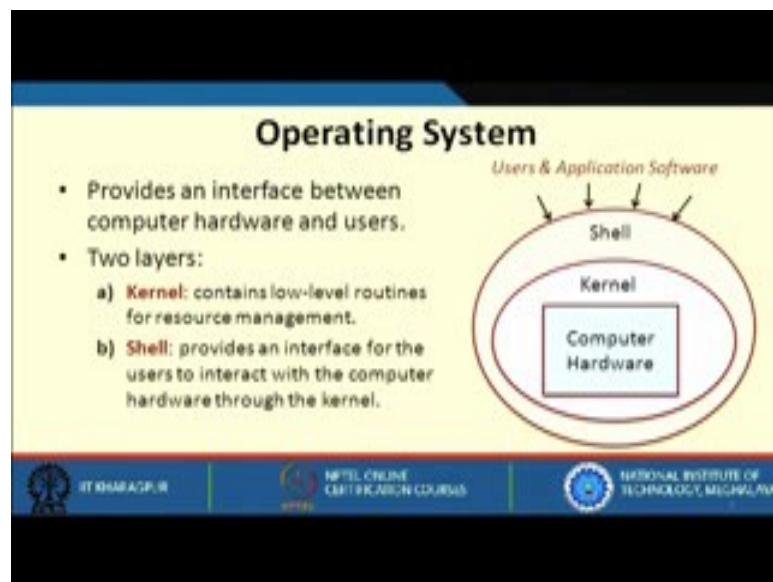
processor, internet browser, etc, managing input output unit, program translation from source form to object form, linking and running user program and many others.

(Refer Slide Time: 05:23)



So, now what are the commonly used system software that we all know is our windows machine, Linux, your MAC. So, various kind of operating system that are there consists of system software. And this program it continues running until you have switched off your machine or your machine is actually shut down. If your machine is shut down, the system software will automatically stop, but as long as your system is running those software will be running. You have compilers, you have assemblers, you have linkers and loaders, and also editors and debuggers.

(Refer Slide Time: 06:15)



So, what is an operating system? It is a system software as I said and it provides an interface between the computer hardware and the user. So, the interface between your hardware and what is your hardware here your hardware is the processor, your hardware is the memory, your hardware is the input-output. So, how does the user actually interact with the hardware is through an operating system. So, operating system is sitting in between which helps in talking between the user and the hardware. The hardware and the user talk to each other through this operating system.

And there are two layers, kernel layer contains the low level routine. So, from this diagram, we can see that hardware sit in the bottom, and then we have a kernel which contains low level routines for resource management, and then we have a shell. Actually all users and application software cannot access this computer hardware directly. So, through this shell it provides an interface for the user to interact with the computer hardware through the kernel. So, kernel is sitting between shell and computer hardware; and between kernel and user, shell is sitting. So, shell is an interface for the user or the application software to interact with the computer hardware through this kernel.

(Refer Slide Time: 08:06)

- The OS is a collection of routines that is used to control sharing of various computer resources as they execute application programs.
 - Typical resources: Processor, Memory, Files, I/O devices, etc.
- These tasks include:
 - Assigning memory and disk space to program and data files.
 - Moving data between I/O devices, memory and disk units.
 - Handling I/O operations, with parallel operations where possible.
 - Handling multiple user programs that are running at the same time.

Operating system is a collection of routines that are used to control sharing of various computer resources as they execute application programs. So, as you know that when we execute a program, some set of instructions get executed. So, for executing such instruction what we are doing we have to load those instruction into the memory, and then from memory it is brought into the processor and each time it is executed. When a processor is executing something the processor is executing only that particular instruction and then many other instructions can also come in other programs can also come in and be in the queue for execution.

So, it is the operating system task that how the various resources can be managed and allocated to processor. When the processor will be allocated to a particular process at what time, it is up to the operating system to decide upon. The task include assigning memory and disk space to programs and data files. Moving data between IO devices, memory and disk units. Handling input output operations with parallel operations were possible. And handling multiple user programs that are running at the same time. So, these are few tasks that are included. There are many more task that an OS performs.

(Refer Slide Time: 10:08)

- Depending on the intended use of the computer system, the goal of the OS may differ.
 - Classical multi-programming systems
 - Several user programs loaded in memory.
 - Switch to another program when one program gets blocked due to I/O.
 - Objective is to maximize resource utilization.
 - Modern time-sharing systems
 - Widely used because every user can now afford to have a separate terminal.
 - Processor time shared among a number of interactive users.
 - Objective is to reduce the user response time.

Now, depending on the intended use of computer system the goal of an operating system may differ like think of a classical multi programming system what happens there. There are several user programs loaded in memory and the OS can switch to another program when any other program is blocked for IO or any other purpose. So, let us say a program is running and at that point of the time that program needs some IO, an IO request has come for that particular program. So, the time it requires to handle that IO request, the processor can switch to another task and that another task can be assigned to the processor and run. So, the switching from one task to another is the main goal of classical multi programming system.

And what was the main objective, here the main objective was to maximize the resource utilization. So, the CPU must not sit idle; if it is doing some particular task and at a time that particular task requires some other resources for completion of that task then the processor has the flexibility to bring another task and execute that, and later when that particular task has completed its IO operation that task can get executed. Now, modern time sharing systems has some other properties. These systems are widely used because every user can now afford to have a separate terminal. Now, the processor time is shared among number of interactive users and here the main objective is to reduce the user response time. What is user response time, the user has requested for a task and by what time that particular request can be taken care of, so that is the user response time.

(Refer Slide Time: 12:19)

– Real-time systems

- Several applications are running with specific deadlines.
- Deadlines can be either hard or soft.
- Interrupt-driven operation – processor interrupted when a task arrives.
- Examples: missile control system, industrial manufacturing plant, patient health monitoring and control system, automotive control system, etc.

– Mobile (phone) systems

- Here user responsiveness is the most important.
- Sometimes a program that makes the system slow or hogs too much memory may be forcibly stopped.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

We have other kind of systems like real time systems. In real time systems there is a time constraint associated with it; and whenever there is a time constraint associated with it we can say that there is a specific deadline associated with the task. And these deadlines can be hard or soft. What do you mean by hard deadline? By hard deadline we mean that that particular task has to finish within that particular time. And by soft deadline that even if that deadline is not met, the system will not fail, but in a hard real time system if the deadline is not met the system may fail. Interrupt driven operations are also there where the processor is interrupted when a task arrives, this happens for some sporadic real time tasks, where there is no fixed time for task arrival. We do not know at what time a task will arrive. Some of the examples are missile control system, industrial manufacturing plant, patient health monitoring and control, and automotive control systems.

For mobile system the user responsiveness is most important. And sometime a program makes the system slow, let us say that we are running some programs in mobile. And in mobile we have limited memory, limited capability of certain things, we are putting everything in a very small space. So, if the computer is not able to handle it, what it does it stops those programs. So, it is forcibly stopped basically.

(Refer Slide Time: 14:27)

The slide has a dark blue header and footer. The main content area is yellow. It features a title 'Classification of Computer Architecture' and two bulleted lists. The first list compares computer architecture types. The second list contrasts computers with calculators based on their internal circuitry, user interaction, and program execution.

Classification of Computer Architecture

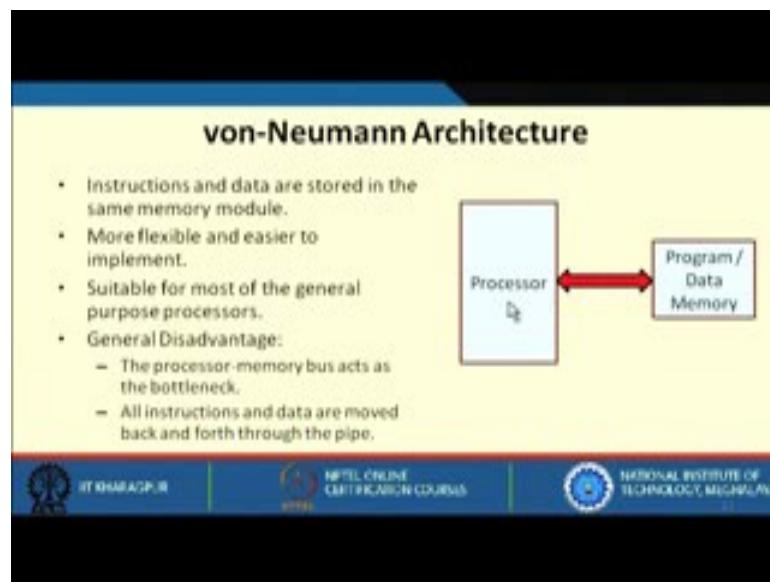
- Broadly can be classified into two types:
 - a) Von-Neumann architecture
 - b) Harvard architecture
- How is a computer different from a calculator?
 - They have similar circuitry inside (e.g. for doing arithmetic).
 - In a calculator, user has to interactively give the sequence of commands.
 - In contrast, a computer works using the *stored-program* concept.
 - Write a program, store it in memory, and run it in one go.

IT Kharagpur | NPTEL CLOUD LEARNING COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MACHILIPATNA

Now, we will see the classification of computer architecture. Broadly, it can be classified into two types von-Neumann architecture and Harvard architecture. So, we will be coming into both these kind of architecture and both these kind of architecture are used in today's computing. How is a computer different from calculator? So, what a calculator does? it has got a circuitry it is adding something, it is dividing something, it is multiplying something and we are getting the result. It is a small device where it is battery operated and you can see the result in that small space.

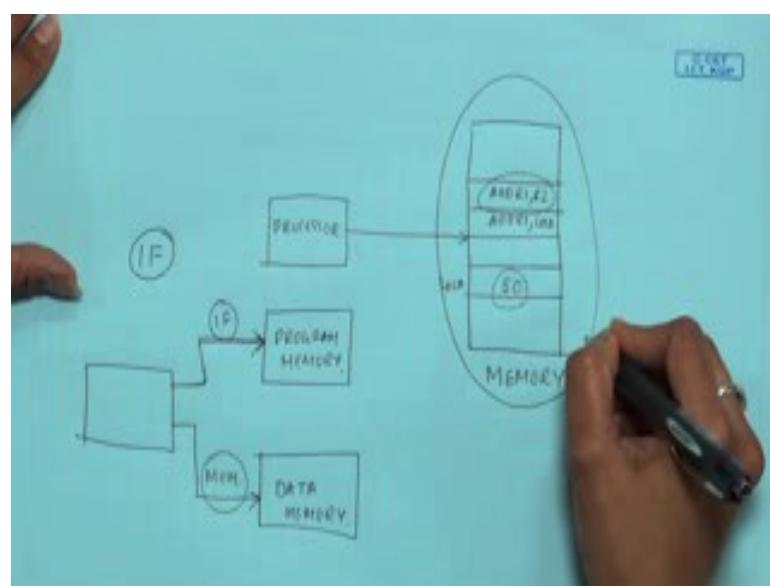
Now, is there any way that we can tell the calculator that I need to perform sequence of calculation one by one. But if you think of the old calculators which can only perform the task of adding 2 numbers or 10 numbers or 15 numbers. But in contrast to that if you see how it will be if we can load all the program into the memory and then I give the task to the computer that process all these instruction that I have stored into the memory one by one. So, this concept is known as stored program concept, where we load both the program and data into the memory and the programs are executed one by one and whatever data is required data are also read. So, we can write a program, store it in memory and we can run it in a go.

(Refer Slide Time: 16:25)



In von-Neumann architecture, both the instructions that is the program and the data are stored in the same memory module. So, this is the memory module and this is the processor both our program and our data are stored in same memory and it is very flexible and easier to implement. Suitable for most of the general purpose processor. But where is the bottleneck and what is the disadvantage? See we have loaded both the programs and data into the memory. If there is a way that I want to access both the program that is the instruction and the data at the same time, I cannot do that, why, because I have a single memory where both my program and data are stored.

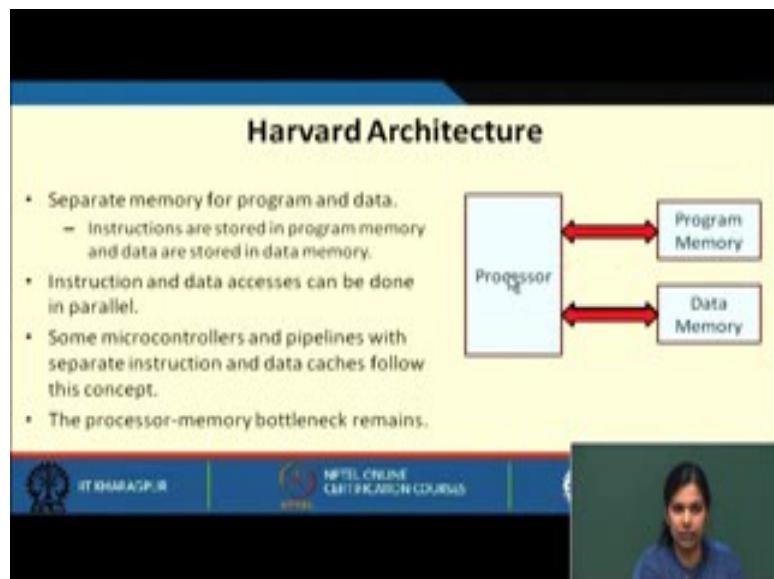
(Refer Slide Time: 17:25)



So, I am trying to say something like this. This is again my memory where in some location both my instruction as well as my data are stored; and the processor is connected to this. Now, when the processor is accessing instruction it cannot get the data at the same time, but what if we can get the instruction and the data at the same time, we will see that this feature is also required for reducing the processor and the memory speed gap.

So, we can see here this is one of the disadvantages; the processor memory bus acts as a bottleneck. So, at a time either instruction or data can be accessed. All instruction and data are moved back and forth through this pipe. So, this is the bottleneck, where processor has to wait for the programs as well as data. Now, what if we have a different program memory and a different data memory; then at the same time, we could also get the program that is the instruction, and at the same time we can get the data.

(Refer Slide Time: 19:19)



Let us see this kind of architecture is called Harvard architecture where we have separate memory for program and data. Instructions are stored in program memory and data are stored in data memory. So, we have a program memory and a separate data memory. Instruction and data access can be done in parallel; obviously, these are two different memories. So, the processor can access the program memory and the processor can also access the data memory at the same time. So, some of the microcontrollers and pipelines with separate instruction and data caches follow this concept. We will see what is

pipeline and we will also see that how separate instruction and data caches follow this approach. But here also this processor and memory bottleneck still remains. This is the processor; we will be accessing the memory, processor will be accessing the data, but multiple data cannot be brought in at the same time.

(Refer Slide Time: 22:44)

Emerging Architectures

- Several architectural concepts have been proposed that deviate significantly from the von-Neumann or Harvard model.
- There is a proposal for in-memory computing architecture, where storage and computation can be done in the same functional unit.
 - Memristors are projected to make this possible in the near future.
 - Memristors can be used in high capacity non-volatile resistive memory systems.
 - Memristors within the memory can also be controlled to carry out some computations.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, people are talking about some emerging architectures as well. So, when they are talking beyond von-Neumann architecture. So, there is a proposal for in memory computing architecture where they say that both the storage and the computation can be done in the same functional unit. So, this is an emerging area, researchers are still working on it, looking into various aspects of it. It is projected that a circuit element called memristor; we have heard of resistor, capacitor, inductor. So, memristor is another circuit element which is projected to make it possible in near future. We have to wait for that, we cannot say at this point that off course, we will be coming with a non von-Neumann architecture in near future, but people are looking into it, people are thinking about it.

Memristors can also be used in high capacity non volatile resistive memory systems and can also be controlled to carry out some computation. So, memristor can be used as memory, memristor can also be used for logic computation. Because of these features we are saying that we can have some kind of in memory computing using memristor in near future.

(Refer Slide Time: 22:22)

Pipeline in Executing Instructions

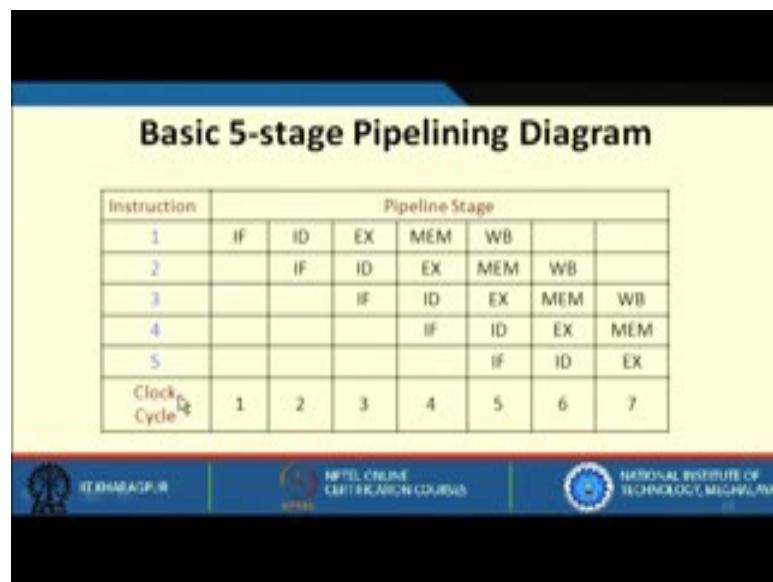
- Instruction execution is typically divided into 5 stages:
 - Instruction Fetch (IF)
 - Instruction Decode (ID)
 - ALU operation (EX)
 - Memory Access (MEM)
 - Write Back result to register file (WB)
- These five stage can be executed in an overlapped fashion in a pipeline architecture.
 - Results in significant speedup by overlapping instruction execution.

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MANGALURU

Now, let us see that as I said that we have separate program memory and data memory. So, instructions are stored in instruction memory and data are stored in data memory. Pipeline is a concept that is used to speed up the execution of instructions, we will be looking into this in the later phase of this course, but just to give you a broad idea pipelining means overlap execution of instructions. An instruction execution is typically divided into five stages, the stages are; fetch the instruction, then we decode that instruction. After decoding that instruction, we execute that instruction. And after execution of that instruction, it may be required that we need to store the result into memory or we need to store the result into any of the processor register as well. So, that write back result to register file can also be done in the fifth stage.

So, an instruction execution cycle is basically divided into these five broad steps, instruction fetch, decode, execute, memory operation and write back. And these five stages can be executed in an overlapped fashion. We are never saying that we are doing parallel processing rather we are saying, we are doing some kind of overlapped execution of instructions. And this results in a significant speed up by overlapping instruction execution.

(Refer Slide Time: 24:27)



Let us see how. So, these are the clock cycles and these are the five instructions that we are going to execute. First instruction is having fetch, decode, execute, memory operation, write back. So, once I have fetched an instruction it is now gone to the decode phase. Once I am decoding that first instruction, can I not fetch the next instruction. How, because decoding is performed in your processor and fetching is done from memory. So, we can overlap instruction decode and instruction fetch. So, when I am decoding the first instruction, the next instruction can be fetched.

Similarly, when I am executing the first instruction, execution of that instruction happens in ALU. When I am executing this particular instruction then the next instruction can get decoded in the control unit. And parallelly the third instruction can be also fetched parallelly. So, we can see that fetching one instruction, decoding one instruction and as well as executing an instruction can all happen in parallel.

Let us move on with the next stage where we see that we are fetching an instruction - the fourth instruction; we are decoding the third instruction. We are executing the second instruction and we are doing some kind of memory operation for the first instruction. But you see if you are doing some kind of memory operation for the first instruction and you are fetching another instruction you cannot do this parallelly. Why because fetching can be performed in from the memory and some memory operation will also be done in the memory. But we can allow this to happen if we have Harvard kind of architecture.

Recall what I said in Harvard kind of architecture we have a program memory and you have a data memory. And now when you are fetching instruction, you are accessing the program memory. And when you are doing some memory operation that means, you are operating on data memory because what you are doing either you are writing something into after execution or even if you are reading something. So, both of this can happen in parallel if and only if you have such kind of architecture.

If you have an architecture where both the programs and the data are stored in single memory like this it would not have happened. So, this can only happen like fetching of an instruction and memory operation if you have Harvard kind of architecture. And what speed up we are gaining just see if these five instruction would take five cycles to execute then if one by one we want to execute it, but it would have been taken twenty five cycles, but now the first result is available after five and the next four result can be available after one by one. So, more four cycles will be required that is $5 + 4 = 9$, a total of 9 cycles will be required to get the result of all five instructions.

(Refer Slide Time: 28:57)

How can Harvard Architecture Help?

- In clock cycle 4, instruction 4 is trying to fetch an instruction (IF), while instruction 1 may be trying to access data (MEM).
 - In von-Neumann architecture, one of these two operations will have to wait resulting in pipeline slowdown.
 - In Harvard architecture, the operations can go on without any speed penalty as the instruction and data memories are separate.

So, as I said how can this Harvard architecture actually help in clock cycle 4, as we have seen that instruction 4 was trying to fetch an instruction, while instruction 1 may be trying to access some data. In von-Neumann architecture one of these two operations will have to wait resulting in pipeline slowdown. But in Harvard architecture, what can

be done is both the operations can be done parallelly because we have separate data memory and separate program memory.

So, in this lecture, we have seen various software that are existing like application software, system software. The various kind of architecture that are existing, i.e. von-Neumann architecture, and the Harvard architecture, and how Harvard architecture actually helps in executing an instruction in a better and faster fashion.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 05
Instruction Set Architecture

Welcome to the fifth lecture, that is, instruction set architecture. Here we are looking into a computer system from programmer's point of view; like the assembly programmer -- what as an assembly programmer will have the view of the computer system.

(Refer Slide Time: 00:47)

Introduction

- Instruction Set Architecture (ISA)
 - Serves as an interface between software and hardware.
 - Typically consists of information regarding the programmer's view of the architecture (i.e. the registers, address and data buses, etc.).
 - Also consists of the instruction set.
- Many ISA's are not specific to a particular computer architecture.
 - They survive across generations.
 - Classic examples: IBM 360 series, Intel x86 series, etc.

IIT KHARAGPUR | INTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Instruction set architecture serves as an interface between the software and the hardware. Here by hardware we mean the processor system, like we need to know what all registers we have and what kind of features are supported by those registers. Typically consists of information regarding programmer's view of the architectures, that is, the registers, address and data buses, etc. and it also consists of the instruction set. Now, many instruction set architectures are not specific to a particular computer architecture and they survive across generations; like if you see Intel X86 series, across generations it has got no change.

(Refer Slide Time: 02:16)

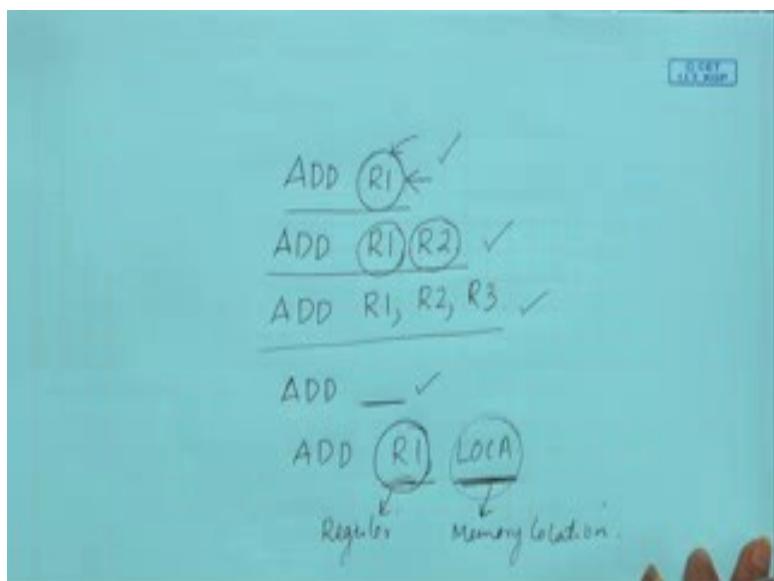
Instruction Set Design Issues

- Number of explicit operands:
 - 0, 1, 2 or 3.
- Location of the operands:
 - Registers, accumulator, memory.
- Specification of operand locations:
 - Addressing modes: register, immediate, indirect, relative, etc.
- Sizes of operands supported:
 - Byte (8-bits), Half-word (16-bits), Word (32-bits), Double (64-bits), etc.
- Supported operations:
 - ADD, SUB, MUL, AND, OR, CMP, MOVE, JMP, etc.

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MANGALURU

Now, let us see what are the important instruction set design issues that should be taken into consideration. First is number of explicit operands. By operands, what do we mean?

(Refer Slide Time: 02:34)



Let us say we have an instruction ADD. We can have several instruction variations, like ADD R1; ADD R1, R2; ADD R1, R2, R3. By number of explicit operands here we have a single operand, two operands, and three operands specified in the instructions. And if we do not specify any operand, then we implicitly take some operands for this operation. We will also see that. So, there can be zero address instruction, there can be one address

instruction, there can be two address instruction, or there can be three address instruction.

Let us now see the location of the operands. By location of operand what do we mean is:
ADD R1,LOCA -- R1 is a processor register, so this is within processor; LOCA is a location in memory. So, by location of the operands, we mean either it is in a register or it is in accumulator or it is in memory. We will see what is an accumulator in course of time. We have to specify to the computer that the first operand is a register, and so you have to look into a register to get the value. This operand will be a register address where you have to go and see the value.

Now, in LOCA you have to specify that this is a memory location and you have to go to that memory location to access this value. So, addressing mode is the way to specify the operands in your instruction. There can be various addressing modes like register, immediate, indirect, relative, etc, we will see in detail later.

Now, we talk about the size of the operands supported. It can be a byte, it can be half-word, it can be a word, it can be a double, etc. By supported operation we mean that how many operations we can specify and what are the various types. When I say ADD, MUL, SUB, these are all arithmetic operations. I can also tell you some other operations like MOVE -- these are data transfer operation, LOAD/STORE are also data transfer operations. And there can be many other kinds of operation that we will see, but by supported operation we mean that how many kinds of operation you are supporting in your architecture.

(Refer Slide Time: 07:25)

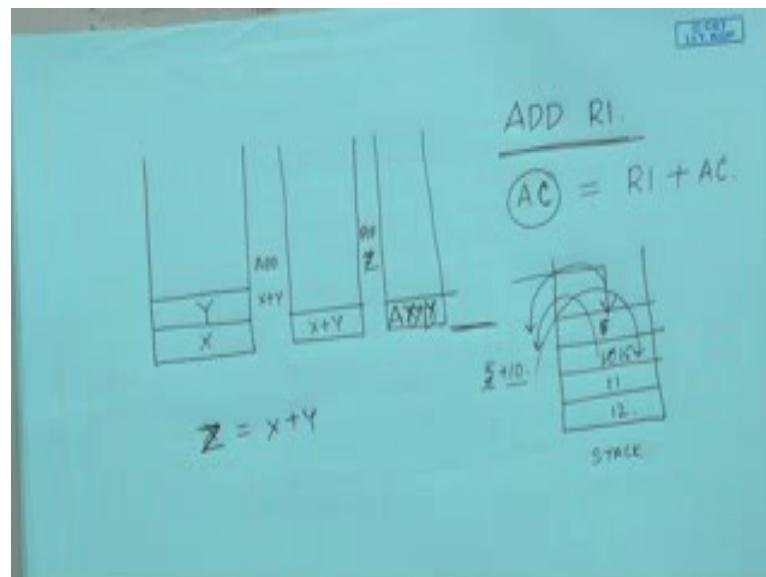
Evolution of Instruction Sets

1: 1-address Instructions:	1960's	(EDSAC, IBM 1130)
ADD X \Rightarrow ACC = ACC + Mem[X]		
2: 0-address Instructions:	1960-70	(Burroughs 5000)
ADD \Rightarrow TOS = TOS		
3: 2- or 3-address Instructions:	1970-80	(IBM 360)
ADD A,B \Rightarrow Mem[A] = Mem[A] + Mem[B]		
ADD A,B,C \Rightarrow Mem[A] = Mem[B] + Mem[C]		
4: 2-address Instructions:	1970-present	(Intel x86)
LOAD R1,X \Rightarrow R1 = Mem[X]		
5: 3-address Instructions:	1980-present	(MIPS, CDC 6600, SPARC)
ADD R1,R2,R3 \Rightarrow R1 = R2 + R3		

IT Kharagpur | NPTEL ONLINE CERTIFICATE COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MANGALURU

Now, this actually shows the evolution of instruction set. Initially in 1960s we were having accumulator based systems. By accumulator-based system let me tell what it is.

(Refer Slide Time: 07:50)

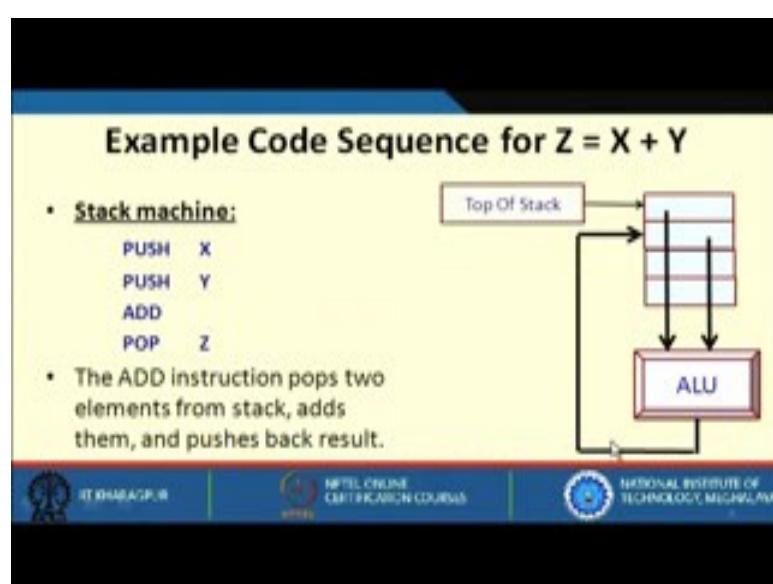


When I write ADD R1; by this R1 will be added with what? R1 will be added with a register present in your processor called **accumulator**. So, accumulator is a register which if we have an instruction like ADD R1; by default, the value of R1 is added with accumulator and the result is stored back in accumulator itself.

During 1960s to 70s another kind of instruction set emerged that is stack based. What do you mean by stack based? In stack-based architecture, a portion of memory called stack is made available. Data on the top of the stack can be accessed. If you just say ADD, and do not specify any operand here, then by default the first two elements of the stack will be taken out, that is 5 and 10, will be added and stored back here; so this becomes 15. By this what we mean is that we are doing some operation where we are loading the data, we are storing the data in the stack, and then we are performing the operation where we need not have to specify any operand --- by default the operands are taken from the stack. So, here it is a 0-address instruction.

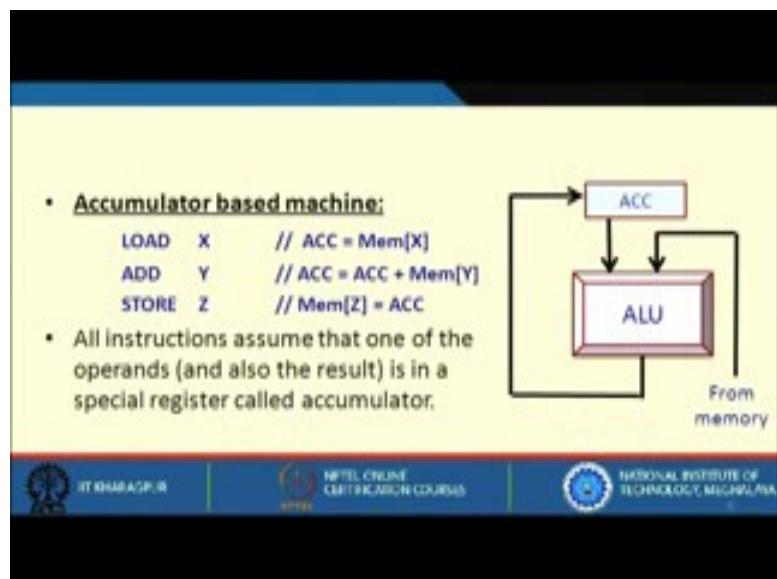
Next comes memory-memory based instructions in 70s and 80s, and the representative system is IBM 360 which has both two-address and three-address instructions. ADD A,B --- where the data from memory location A is added with data from memory location B and the result is stored back in A. Now, we also have register-memory based systems where one operand will be a register and one operand will be a memory location. So, LOAD R1,X. What do you mean by that? Load from memory location pointed by X into R1. Similarly, we can also have STORE, store the value of R3 into some other location. And finally, we have three-address instructions where we are specifying three addresses and what does it do? Here R1 stores the value of R2 and R3, the result of the addition is stored in R1. So, in a single step, we can do this.

(Refer Slide Time: 12:45)



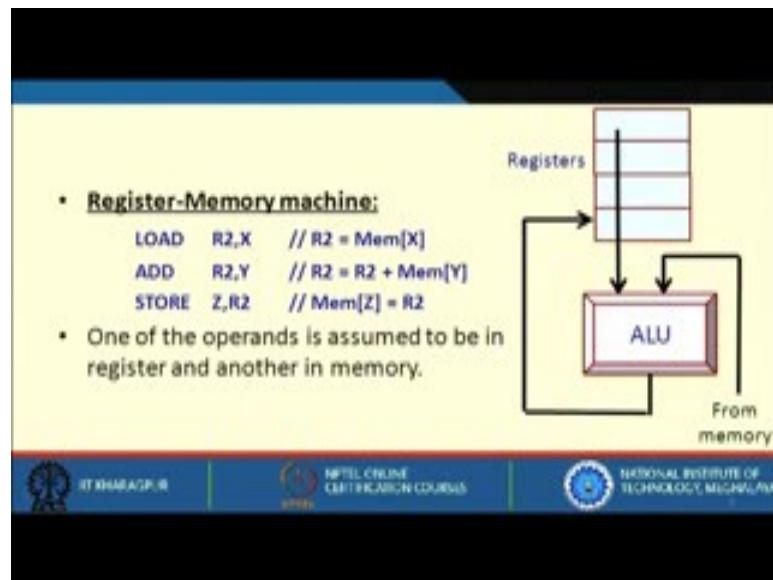
Now, let us see some example code sequence for executing some sample instruction that is $Z = X + Y$, using the various instruction set architectures that I have told in the previous slide. So, let us consider this stack-based machine. First we have to push X, next we have to push Y, and then both of these are now in top two position of the stack. When we perform this ADD, these two values are taken out, added and stored back in the top of the stack. And finally, when we do pop, then the value from top of the stack is taken out and stored back in Z. So, let me explain here in this way. So, by PUSH X, X is added here; by PUSH Y, Y is added here. And then once we perform ADD, X + Y is added and stored back here. And then when we perform POP Z, then Z is a memory location where we will get the result.

(Refer Slide Time: 14:52)



Next, see an accumulator-based system. In an accumulator-based system, if you have to perform the same operation $Z = X + Y$, then what you have to do? Both X and Y are some values stored in memory, so you have to load X in accumulator. Then ADD Y will actually add the content of location Y with accumulator and store back the result in accumulator. So, we can see in the ALU one value is coming from the accumulator and another is coming from memory; we are adding these two values and we are storing back the result in accumulator. So, all instructions assume that one of the operands and also the result is in a special register called accumulator.

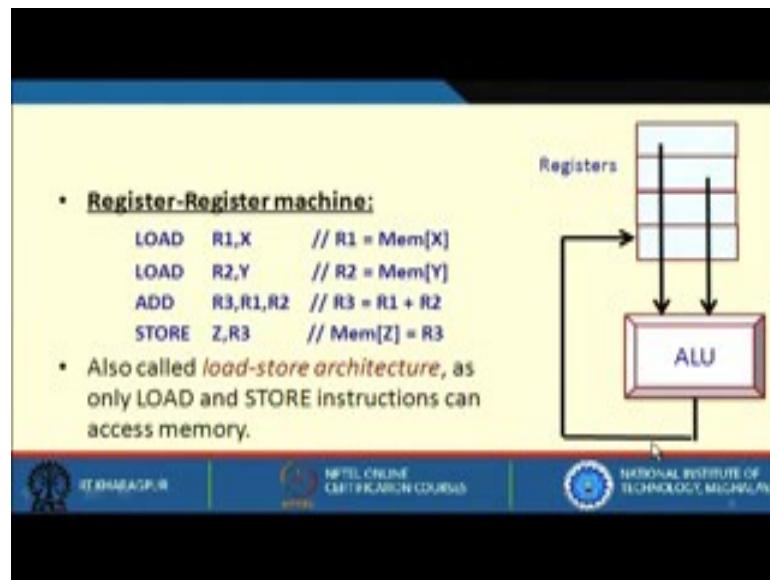
(Refer Slide Time: 16:06)



Next we see register-memory machine. In register-memory machine how this can be performed? LOAD R2,X --- from location X the data will be loaded in R2. When we do ADD R2,Y --- in R2 the content of R2 which was nothing but X will be added with Y and stored back in R2. Finally, we have to store this result R2 in Z.

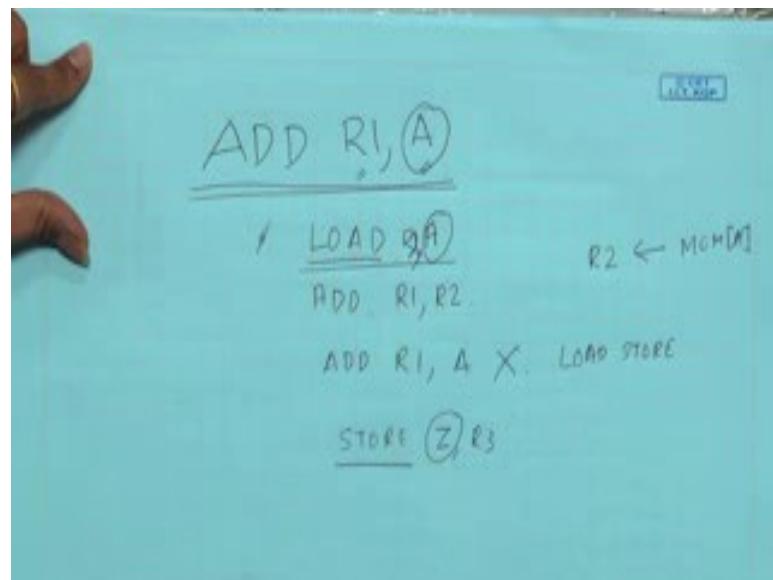
So, here one of the operands is assumed to be in register and another in memory. So, that is why in the ALU one of the operand is your register. So, one value is coming from register, another is coming from memory and then finally, the result is getting stored here in some register, and finally, the STORE will store back the result in Z memory location.

(Refer Slide Time: 17:34)



Now, we see register-register machine. So, in register-register machine what you need to do is that you have to load everything into some register first. So, here instead of doing ADD operation or any kind of operation if you want to perform, you have to perform only on registers and not on any memory location; that is why in register-register machine you have to first load all the values of the locations memory location into some register. That is why we are using two back-to-back LOAD. In the first LOAD, value of X will be loaded in R1, and in next the value of Y will be loaded in R2. Now we can add these two registers and store the value in R3. Now, finally, we have to store the result in Z. This kind of architecture is also called load-store architecture. By load-store architecture we mean that only these two instructions LOAD and STORE will be used to access the memory, and no other instructions will be used to access the memory.

(Refer Slide Time: 19:11)



Earlier we have seen that using instruction like ADD R1,A, --- where A is a memory location. In this instruction, we are allowing one register operand and one memory operand; but in LOAD/STORE architecture what will happen, you cannot access this. So, what you have to do, you have to load A using LOAD R1,A. But you cannot do ADD R1,A in LOAD/STORE architecture. You can only use load and store to access memory.

(Refer Slide Time: 20:41)

About General Purpose Registers (GPRs)

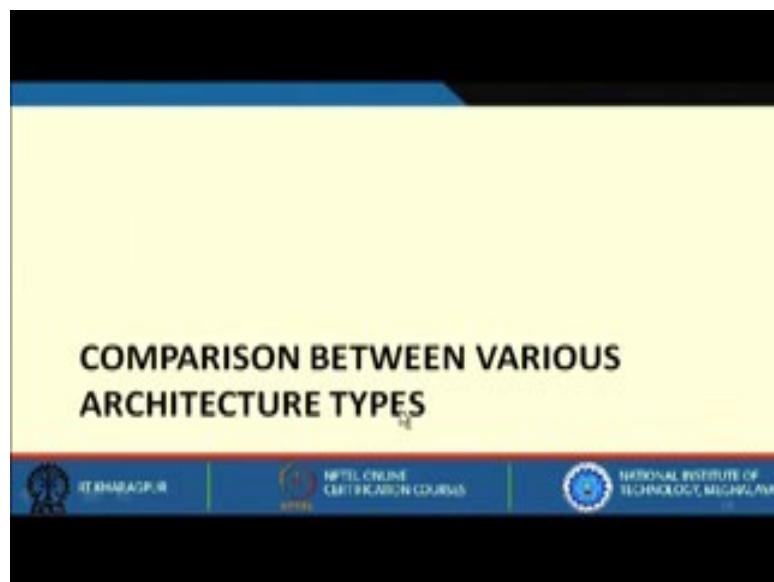
- Older architectures had a large number of special purpose registers.
 - Program counter, stack pointer, index register, flag register, accumulator, etc.
- Newer architectures, in contrast, have a large number of GPRs.
- Why?
 - Easy for the compiler to assign some variables to registers.
 - Registers are much faster than memory.
 - More compact instruction encoding as fewer bits are required to specify registers.
 - Many processors have 32 or more GPR's.

IT Kharagpur
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us see about the general-purpose registers. If you recall, older architectures have large number of special-purpose register like we talked about program counter, stack pointer, some index register, flag registers, accumulator, etc. But in newer architectures we have more number of general-purpose registers. And instead of using special purpose registers, most of the operations are performed using general purpose registers. And why that is so? The compiler can assign some variables to registers. So, there are so many variables that can be used and they can be assigned it to registers, and registers are much faster than memory. So, once you load the data into the registers and you are performing operation within the register it will be much faster. But first you have to load the data from your memory to register and then only you can perform the operation within register.

More compact instruction encoding is possible as fewer bits are required to specify register. So, the instruction encoding can be much less why they are saying like see you are bringing everything into registers and the registers cannot be unlimited as compared to the memory addresses. Memory addresses a 32-bit memory addresses will have 32-bit, but if you have 40 register or 100 registers how many bits do you require? For 100 register you will require a maximum of 7-bits to encode. So, many processors have 32 or more general purpose registers.

(Refer Slide Time: 22:42)



Now, coming to comparison between various architectural types. I have discussed about many machines: stack based, accumulator based, register-register memory. Now, let us do a comparison and figure out that which one is better than the other.

(Refer Slide Time: 23:07)

The slide is titled '(a) Stack Architecture'. It contains two main sections: 'Typical instructions:' and 'Example: Y = A / B = (A - C * B)'.

Typical instructions:

- PUSH X, POP X
- ADD, SUB, MUL, DIV

Example: $Y = A / B = (A - C * B)$

```
PUSH A  
PUSH B  
DIV  
PUSH A  
PUSH C  
PUSH B  
MUL  
SUB  
SUB  
POP Y
```

A diagram illustrates a stack with the Top Of Stack (TOS) pointing to the top element. The stack contains four elements: B, C, A, and A/B. The stack grows downwards.

NETEL ONLINE CERTIFICATION COURSES

Firstly, we will do that with the example. So, we will be executing this particular instruction. This particular instruction will be executed and we will see that with various architectures how many steps it takes to execute. This particular instruction for stack based what you have to do. So, these are all memory location data. So, what we need to do. So, we have to first PUSH A, because we need to put this value into stack, and then only we can perform the operation. Then we have to PUSH B, then we can specify the operation DIV that will perform that particular operation and store back in the stack. Next we will do PUSH A, PUSH C and PUSH B. We have pushed first A, then B, then C then B why we have done so? Just see we have performed this operation now we have to perform this operation. For doing this operation we will first perform $C * B$ and then A that is why we have first put A and then we have put C and B.

(Refer Slide Time: 24:39)

(a) Stack Architecture

- Typical instructions:
PUSH X, POP X
ADD, SUB, MUL, DIV
- Example: $Y = A / B - (A - C * B)$

TOS →

```
PUSH A
PUSH B
DIV
PUSH A
PUSH C
PUSH B
MUL
SUB
SUB
POP Y
```

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES

A woman is visible on the right side of the slide.

Now, when we do the next operation MUL then what will happen B and C will be multiplied and it will be stored back.

(Refer Slide Time: 24:54)

(a) Stack Architecture

- Typical instructions:
PUSH X, POP X
ADD, SUB, MUL, DIV
- Example: $Y = A / B - (A - C * B)$

TOS →

```
PUSH A
PUSH B
DIV
PUSH A
PUSH C
PUSH B
MUL
SUB
SUB
POP Y
```

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES

A woman is visible on the right side of the slide.

And now if we do a SUB then $A - B$ into C will happen. So, we have performed this part, we have already performed this part earlier which is stored in the stack. Now, if you just perform a SUB again then what will happen $A / B - (A - B * C)$ will take place.

(Refer Slide Time: 25:15)

(a) Stack Architecture

- Typical instructions:
PUSH X, POP X
ADD, SUB, MUL, DIV
- Example: $Y = A / B - (A - C * B)$

The stack diagram shows the Top Of Stack (TOS) pointing to the top of a stack containing four memory locations. The bottom-most location contains the value $A / B - (A - C * B)$.

```
PUSH A  
PUSH B  
DIV  
PUSH A  
PUSH C  
PUSH B  
MUL  
SUB  
SUB  
POP Y
```

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES

A woman is visible in the bottom right corner, likely the lecturer.

So, now in my stack we have $A / B - (A - B * C)$, but finally we have to store this into a memory location Y. So, we are storing this into Y by doing a POP. So, POP will take out the result from top of the stack and put the result into Y.

(Refer Slide Time: 25:33)

(a) Stack Architecture

- Typical instructions:
PUSH X, POP X
ADD, SUB, MUL, DIV
- Example: $Y = A / B - (A - C * B)$

The stack diagram shows the Top Of Stack (TOS) pointing to the top of a stack containing four memory locations. The bottom-most location contains the value $Y = RESULT$.

```
PUSH A  
PUSH B  
DIV  
PUSH A  
PUSH C  
PUSH B  
MUL  
SUB  
SUB  
POP Y
```

Y = RESULT

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES

A woman is visible in the bottom right corner, likely the lecturer.

So, how many steps basically we took to execute this? We took 10 steps.

(Refer Slide Time: 25:55)

(b) Accumulator Architecture

- Typical instructions:
 - LOAD X, STORE X
 - ADD X, SUB X, MUL X, DIV X

Example: $Y = A / B - (A - C * B)$

```
LOAD C
MUL B
STORE D //D = C*B
LOAD A
SUB D
STORE D //D = A - C*B
LOAD A
DIV B
SUB D
STORE Y
```

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES

A woman in a blue shirt is visible on the right side of the slide.

Next we see accumulator architecture. In an accumulator architecture the typical instructions that we will be using is LOAD, STORE and of course, along with that we can use other memory operation along with ADD. So, first we LOAD C then we MUL B, so B is multiplied with C and stored in the accumulator, but we have to store back the result somewhere because later we will be again using it. So, we store the result in D. Then we LOAD A again then we SUB D and we STORE D. So, the operation $A - C * B$ is performed and it is stored in D. Now, finally, what we have to do we have to perform this and we have to subtract this from this.

So, then again we LOAD A and DIV B; this will make A / B and it is stored in accumulator and then we do SUB D. So, whatever is in D will be subtracted from whatever was in accumulated. So, what was in accumulator was A / B , and then it gets subtracted and finally, we store the result in Y. So, whatever was in accumulator is the result of this we store back in Y. And now let us see how many steps we require to execute this. We require 10 steps.

(Refer Slide Time: 27:47)

(c) Memory-Memory Architecture Example: $Y = A / B - (A - C * B)$

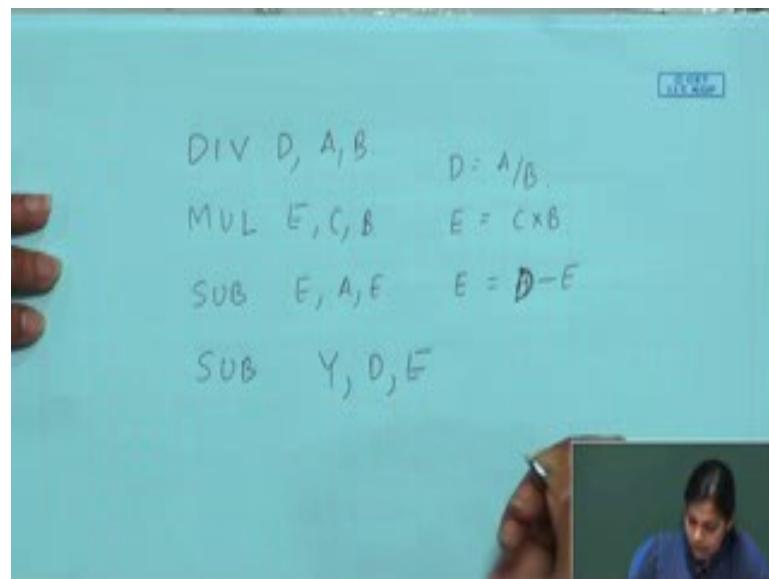
- Typical instructions (3 operands):
 - ADD X,Y,Z
 - SUB X,Y,Z
 - MUL X,Y,Z
- Typical instructions (2 operands):
 - MOV X,Y
 - ADD X,Y
 - SUB X,Y
 - MUL X,Y

DIV A,B,D	MUL E,C,B
SUB E,A,E	SUB Y,D,E
MOV D,A	MOV E,C
DIV D,B	MUL E,B
SUB A,E	SUB D,A

IT Kharagpur IIT Kharagpur NPTEL ONLINE CERTIFICATION COURSES

Let us see memory-memory architecture. The programs using both three-address and two-address instructions are shown..

(Refer Slide Time: 28:47)



(Refer Slide Time: 32:37)

(d) Load-Store Architecture

- Typical instructions:
 - LOAD R1,X
 - STORE Y,R2
 - ADD R1,R2,R3
 - SUB R1,R2,R3

Example: $Y = A / B - (A - C * B)$

LOAD R1,A	LOAD R2,B
LOAD R3,C	DIV R4,R1,R2
MUL R5,R3,R2	SUB R5,R1,R5
SUB R4,R4,R5	STORE Y,R4

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | NITI Aayog

Next is load-store. In load-store instruction as I said the instruction that will is required to access the memory is LOAD and STORE. So, first of all we have three operands A, B and C --- we will load that into three different registers R1, R2 and R3. Finally, we do DIV where we divide A by B and we store it in R4; then we do MUL B and C, that is R2 and R3 store it in R5. Then we sub R1 minus R5 and store the result back in R5. So, we got this one. We continue this way where the number of instructions required is much less.

(Refer Slide Time: 34:36)

Registers: Pros and Cons

- The load-store architecture forms the basis of RISC ISA.
 - We shall explore one such RISC ISA, viz. MIPS.
- Helps in reducing memory traffic once the memory data are loaded into the registers.
- Compiler can generate very efficient code.
- Additional overhead for save/restore during procedure or interrupt calls and returns.
 - Many registers to save and restore.

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | NITI Aayog

So, what are the pros and cons that we see. The load store architecture forms the basis of RISC (reduced instruction set computer) instruction set architecture. And in this course we shall explore one such RISC ISA, that is MIPS. What it does it helps in reducing the memory traffic once the memory data are loaded into registers. As you have seen that if only load and store instructions are used to access memory then we can load the data in a go using number of load operations and then we perform all the operations within the processor. So, the processor will be performing all the operations with some register values because we have already loaded those data from your memory into the register. And then finally, if we have to store again back from register to some memory location we can do that.

So, compiler once knows this can of course can generate very efficient code and additional overhead for save and restore during procedure or interrupt calls and return will be clear because now we have many registers to save and restore. But again save and restore will be much more it would not be so much like; what I wanted to say is that it is of course, an additional overhead for saving and restoring during procedure call because we have to save the data and store. But this can be done in an efficient way if we can store it in some other registers as well but although this is an overhead that we see. So, these are the pros and cons of registers.

So, by this we came to the end of lecture 5 and we also came to the end of Week 1 lecture. So, to summarize what we have studied in week 1 --- we started with how computers have evolved, then we have seen that how an instruction can get executed. So, to execute an instruction, we perform set of steps; instructions and data are stored in memory; to execute it we have to bring it from memory to processors, execute it and store it back. We have also seen that what kind of software are existing like application software and system software. We have also seen that both von-Neumann architecture and Harvard architecture are required. And in the last lecture, we have seen that how this instruction set architecture has evolved, what kind of machines were used in early stages, accumulator based machine then stack based machine, then finally, how we are now in a stage where we perform some kind of operation faster. So, our thrust is how we can execute programs faster.

So, in this course, in the next lecture, we will be seeing that how these concepts can be further used to enhance the speed of a computer.

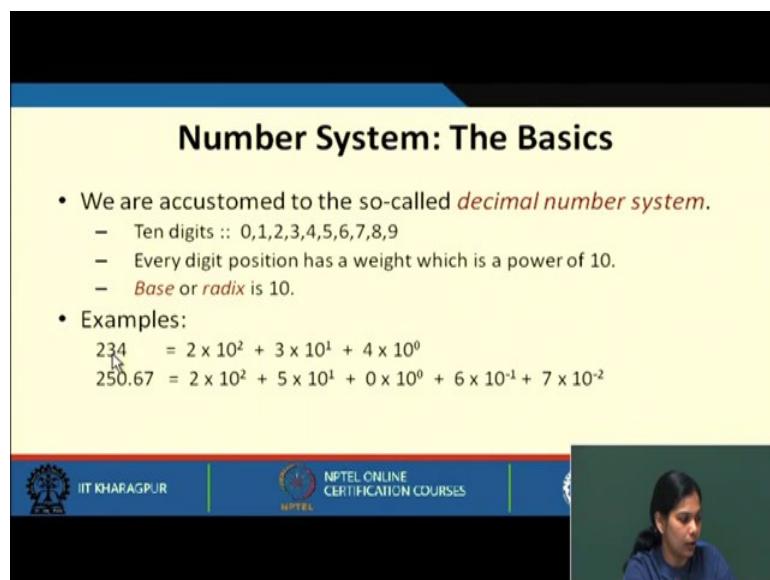
Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 06
Number Representation

Welcome back. We shall now be moving on with the detailed discussion on various aspects of computer architecture and organization. As we know that typically digital circuits work on binary number system which can ultimately be mapped to some tiny electronic switches. So, in this lecture we will be considering this number representation in computer systems.

(Refer Slide Time: 00:54)



Number System: The Basics

- We are accustomed to the so-called *decimal number system*.
 - Ten digits :: 0,1,2,3,4,5,6,7,8,9
 - Every digit position has a weight which is a power of 10.
 - *Base* or *radix* is 10.
- Examples:
$$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$
$$250.67 = 2 \times 10^2 + 5 \times 10^1 + 0 \times 10^0 + 6 \times 10^{-1} + 7 \times 10^{-2}$$

IIT KHARAGPUR | **NPTEL ONLINE CERTIFICATION COURSES** | **NPTEL**

We all are accustomed with the so called decimal number system. In decimal number system, we represent ten digits 0 to 9; and every digit position has a weight which is a power of 10, called base or radix. Let us take an example 234, so, it can be represented the weighted representation like this: 4×10^0 in the unit position, then 3×10^1 in the tenth position, and 2×10^2 in the hundredth position. Similarly, we can also represent both integer part and fractional part, where for the integer part we multiplied this number with 10^0 , 10^1 , and 10^2 and here we start with -1. So, we multiply 6×10^{-1} , 7×10^{-2} , and so on.

(Refer Slide Time: 02:26)

Binary Number System

- Two digits: 0 and 1.
 - Every digit position has a weight that is a power of 2.
 - *Base or radix* is 2.
- Examples:
110 = $1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
101.01 = $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Coming to binary number system, we all know we have two digits 0 and 1. And every digit position has a weight that is power of 2; in decimal number system we have power of 10; and here we have power of 2. So, the base or radix here is 2. Let us see how we can represent a binary number. So, 1 1 0 is this number: 0 will be multiplied with 2^0 , the next number will be multiplied with 2^1 , and the next will be multiplied with 2^2 . So, this is the weighted representation of this number. Similarly, we can also represent a fractional part. So, 101.01, it can be represented, so this will be 2^0 , this will be 2^1 , and this will be 2^2 in the same way. And again in the fractional part we will be multiplying with 2^{-1} , 2^{-2} , and so on. So, this will be $0 \times 2^{-1} + 1 \times 2^{-2}$ and so on.

(Refer Slide Time: 04:13)

Binary to Decimal Conversion

- Each digit position of a binary number has a weight.
 - Some power of 2.
- A binary number:
$$B = b_{n-1} b_{n-2} \dots b_1 b_0 . b_{-1} b_{-2} \dots b_{-m}$$
where b_i are the binary digits.

Corresponding value in decimal:

$$D = \sum_{i=-m}^{n-1} b_i 2^i$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Now, let us see some conversions, like binary to decimal, decimal to binary, binary to hexadecimal, hexadecimal to binary. So, we will be looking into all those things. So, when we are doing binary to decimal conversion, each digit position of a binary number has a weight and that weight is some power of 2. So, our binary number can be represented like this, this is the integer part and this is the fractional part. The integer part starts from b_0, b_1, b_{n-1} ; and the fractional part b_{-1}, b_{-2} to b_{-m} , where b_i are the binary digits.

So, the corresponding value in decimal will be all these binary digits multiplied by 2^i . So, if this digit is b_0 , then $b_0 \times 2^0$. If it is b_1 , then it will be $b_1 \times 2^1$ and so on, and here $b_{-1} \times 2^{-1}$, and so on, till b_{-m} . So, the corresponding value in decimal can be represented by this summation.

(Refer Slide Time: 06:14)

Some Examples

- 101011 $\rightarrow 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 43$
 $(101011)_2 = (43)_{10}$
- .0101 $\rightarrow 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = .3125$
 $(.0101)_2 = (.3125)_{10}$
- 101.11 $\rightarrow 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 5.75$
 $(101.11)_2 = (5.75)_{10}$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Let us take some examples, which will make this clear. So, we have a number 1 0 1 0 1 1, this is a binary number, how we can convert it into a decimal number? We start with multiplying 1 with 2^0 and so on, and we get 43 in decimal. Similarly if we have a fractional part to convert it we have to multiply this with 2^{-1} , and so on. Similarly, if we have both integer part and fractional part in the same way we can consider the first part as like this and the next part like this and finally, we get a value like this.

(Refer Slide Time: 07:43)

Decimal to Binary Conversion

- Consider the integer and fractional parts separately.
- For the integer part:
 - Repeatedly divide the given number by 2, and go on accumulating the remainders, until the number becomes zero.
 - Arrange the remainders *in reverse order*.
- For the fractional part:
 - Repeatedly multiply the given fraction by 2.
 - Accumulate the integer part (0 or 1).
 - If the integer part is 1, chop it off.
- Arrange the integer parts *in the order* they are obtained.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us consider decimal to binary conversion. So, we have seen binary to decimal conversion. Now, we will be seeing decimal to binary conversion. Consider the integer and fractional part separately. Here we have to consider the integer and fractional part separately let us see how. So, what we do for the integer part, for the integer part we repeatedly divide the given number by 2 and we go on accumulating the remainders until the number becomes 0.

(Refer Slide Time: 08:20)

The image shows handwritten calculations on a light blue background. On the left, the conversion of 15₁₀ to binary is shown using successive division by 2:

$$\begin{array}{r}
 2 | 15 \\
 2 | 7 \quad - 0 \\
 2 | 3 \quad - 1 \\
 \hline
 \end{array}$$

An arrow points from the quotient 0 to the next step. Below this, the result is written as $(15)_{10} = \underline{\underline{1111}}_2$.

On the right, the conversion of 0.25₁₀ to binary using successive multiplication by 2 is shown:

$$\begin{array}{r}
 0.25 \\
 \times 2 \\
 \hline
 0.50 \\
 \times 2 \\
 \hline
 1.00
 \end{array}$$

An arrow points from the quotient 1.00 back to the first step. Below this, the result is written as $\underline{(0.25)_{10}} = \underline{\underline{0.01}}_2$.

So, let us take an example. So, here we can see that the number is 15. What we say that we repeatedly divide the given number by 2, so we are dividing the given number by 2 we are getting a quotient and we are getting a remainder. And we are going on accumulating the remainder that is here it is 1; again we do it, we get here 3 and the remainder becomes 1. Again we divide it and we get the remainder as 1 and the quotient as 1 as well. And what we do finally, we arrange it in reverse order. So, in reverse order when we arrange this 15 in decimal is now converted to 1 1 1 1 in binary. So, here this is what it has been said that for the integer part we repeatedly divide a given number by 2 and we accumulate the remainder until the number becomes 0 and then we arrange it in reverse order the way.

Next, for the fractional part, we repeatedly multiply the given fraction by 2, and then we accumulate the integer part. If the integer part is 1, we chop it off. And then finally, while arranging we will be arranging the order the integer parts in the same order.

So, now let us see this with an example of 0.25. We multiply it by 2 and we get 0.50. So, here 0 is accumulated this is the integer part which is 0. Next, again we take the fractional part we do not do anything with this part. So, we take the fractional part, and again we multiply it by 2. And what we get we get 1.00, where we get this fractional, this integer part as 1. So, here we can say that 0.25 in decimal can be represented as 0.01. So, this point remains and what we got is 01 we are not writing this in reverse order rather we are writing it as 0 1 0 1. So, this is the decimal this is the binary representation of this particular decimal fractional part. This is how we do it. And we can continue doing it because it is not necessary that we will get 0 0, we can get any number. So, we can do it for some number of times and we get the result.

(Refer Slide Time: 11:49)

Example 1: Conversion of 239 to binary	
2 239	
2 119 --- 1	
2 59 --- 1	
2 29 --- 1	
2 14 --- 1	
2 7 --- 0	
2 3 --- 1	
2 1 --- 1	
2 0 --- 1	

$(239)_{10} = (11101111)_2$

Example 2: Conversion of 64 to binary	
2 64	
2 32 --- 0	
2 16 --- 0	
2 8 --- 0	
2 4 --- 0	
2 2 --- 0	
2 1 --- 0	
2 0 --- 1	

$(64)_{10} = (1000000)_2$

Example 3: Conversion of 0.634 to binary	
.634 x 2 = 1.268	
.268 x 2 = 0.536	
.536 x 2 = 1.072	
.072 x 2 = 0.144	
.144 x 2 = 0.288	
:	
$(.634)_{10} = (.10100.....)_2$	

37.0625
 $(37)_{10} = (100101)_2$
 $(.0625)_{10} = (.0001)_2$
 $\therefore (37.0625)_{10} = (100101 . 0001)_2$

Now, let us see some more examples. Here in the same way 239 in decimal can be converted to binary where we get 1 1 1 0 1 1 1 1, which we write in reverse order. Similar way, let us take another example which is 64 we go on dividing with the number by 2 and we accumulate the remainder and finally, what we get is 1 0 0 0 0 0 0.

Let us take some example with the fractional part. So, here we see that 0.634 in decimal how we can convert it into binary. In the same way we multiply it by 2 and we keep the integer part. And then we take the fractional part again and keep on multiplying with 2. And again after multiplying in we keep the integer part and again with this fractional part we multiply it again with 2, and keep on doing it.

Let us say we have done it till 1 2 3 4 and 5 times and we write it in this order. So, the value will be point 1 0 1 0 0 and if you want to do it for some more you can also do that. So, this is how we can convert a fractional decimal part to binary. Similarly, we can take an example of a number, which is having both integer part and fractional part. So, the integer part can be converted and it is represented by this and the fractional part is converted and it is represented by this, and finally you can write the answer in this fashion.

(Refer Slide Time: 14:19)

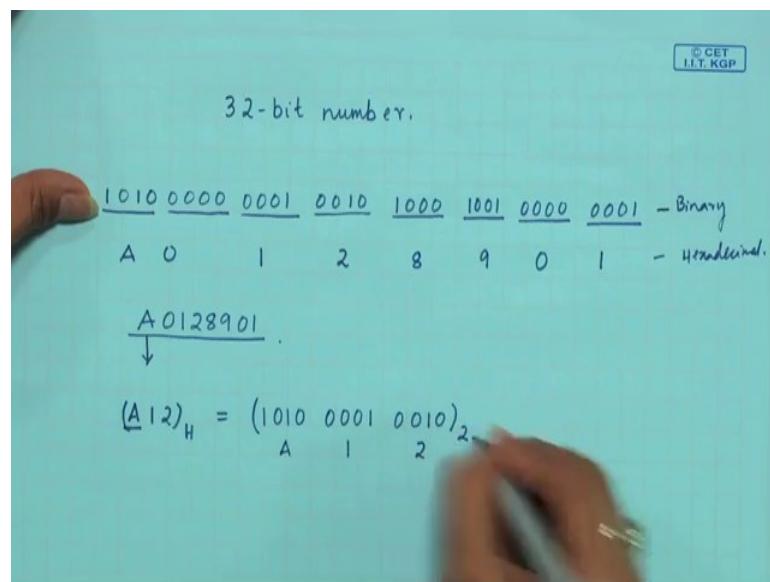
Hexadecimal Number System

- A compact way to represent binary numbers.
 - Group of four binary digits are represented by a hexadecimal digit.
 - Hexadecimal digits are 0 to 9, A to F.

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Now, coming to hexadecimal number system. Now, why do we require hexadecimal number system? Now, let us say you have to represent a 32-bit number or a 64-bit number. So, if you have to represent a 64-bit number in binary, you have to represent it as 0 1 0 1 up to 64 bits, or if it is a 32-bit number then you have to write that 0 1 0 1 whatever is the number till 32 bits. So, it will be really very long. Hexadecimal number system is a compact way to represent binary numbers. This is an efficient way where what we do basically is group four binary digits and represent by a hexadecimal digit. And the hexadecimal digits are 0 to 9, and A to F. 0 to 9 we already know and starting from 10 is represented as A, 11 is represented as B, 12 as C, 13 as D, 14 as E, and 15 as F. So, we have these representations starting from 0 to F in hexadecimal number. So, a group of four numbers will be replaced with a single digit. If we encounter 1 0 1 0, we will replace this with the digit A.

(Refer Slide Time: 16:28)



So, if we have to represent a 32-bit number, we can represent by only 8 digits. How, because 1010 is A, 0000 is 0, 0001 is 1, 0010 is 2, this is 8, this is 9, this is 0, this is 1. So, this entire number A0128901 is the hexadecimal representation of this 32-bit number.

(Refer Slide Time: 18:14)

Binary to Hexadecimal Conversion

- For the integer part:
 - Scan the binary number from *right to left*.
 - Translate each group of four bits into the corresponding hexadecimal digit.
 - Add *leading* zeros if necessary.
- For the fractional part:
 - Scan the binary number from *left to right*.
 - Translate each group of four bits into the corresponding hexadecimal digit.
 - Add *trailing* zeros if necessary.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, we will move on. So, let us see how we can convert this binary number to hexadecimal. So, for the integer part, we scan the binary number from right to left, and we translate each group of four bits into the corresponding hexadecimal digit. And if

required we add leading zeros that means, if it is required that we need to add some more bits to make it a group of four then we can add those bits.

So, for the fractional part, what we do we scan the binary number from left to right. So, there is a difference between the integer part and your fractional part. In the integer part, we are doing from right to left; and in the fractional part, we are doing from left to right. And what we do we translate each group of four bits into the corresponding hexadecimal digit that I have already shown you. And for doing so if it is required then what we do we add trailing zeroes into it.

(Refer Slide Time: 19:54)

The screenshot shows a presentation slide with a yellow background and a blue header bar. The title 'Examples' is centered in the header. Below the title, there are four numbered examples:

1. $(\underline{1011} \underline{0100} \underline{0011})_2 = (B43)_{16}$
2. $(\underline{10} \underline{1010} \underline{0001})_2 = (2A1)_{16}$ *Two leading 0s are added*
3. $(.\underline{1000} \underline{010})_2 = (.84)_{16}$ *A trailing 0 is added*
4. $(\underline{101} \underline{.0101} \underline{111})_2 = (5.5E)_{16}$ *A leading 0 and trailing 0 are added*

At the bottom of the slide, there is a navigation bar with the IIT Kharagpur logo, the text 'NPTEL ONLINE CERTIFICATION COURSES', and a small video thumbnail of a person speaking.

Let us take some more examples. Let us say we have a number which is represented in binary 1010. So, we group it from we group this from right to left. So, first we group this four bit, next four bit, then next four bit and then we find out that we have a group of three groups having four-four bits. So, what is 1011 -- it is B. What is 0100 -- it is 4 in hexadecimal; and 0011 is 3. So, we have converted a binary number into hexadecimal representation.

Similarly, here you see you do not have a group of four. So, what you have to do you have to add some zeros in the beginning. So, here we group it four here also we group it four, but here we cannot group it four. So, you add 2 more zeros. So, 2 leading zeros are added to make it 0010 which is equivalent to 1010 is A, and 0001 is 1. Similarly, for the fractional part we scan from left to right. So, scanning this from left to right, so we have

to group into four-four bits. So, this is the first four bits and this is the next four bits. To make it four, we have to add a trailing zero. So, a trailing zero is added to make it a group of four bits. So, this 1000 is 8 and 0100 is 4 which in hexadecimal.

Similarly, for this is the integer part and this is the fractional part. For the integer part, we have to add a leading 0 which makes it 0101, 0101, so which is 5 and then for this we have to add a trailing 0. So, if we add a trailing 0 this will make it as 5, and this is E. So, this is how we convert a binary number into hexadecimal number for simply representing it in a compact fashion.

(Refer Slide Time: 22:22)

The slide has a yellow background with a black header and footer. The title 'Hexadecimal to Binary Conversion' is at the top. Below it is a bulleted list of instructions and examples:

- Translate every hexadecimal digit into its 4-bit binary equivalent.
- Examples:
 - $(3A5)_{16} = (0011\ 1010\ 0101)_2$
 - $(12.3D)_{16} = (0001\ 0010.\ 0011\ 1101)_2$
 - $(1.8)_{16} = (0001.\ 1000)_2$

The footer features the IIT Kharagpur logo, the NPTEL logo, and a video frame showing a person speaking.

Now, in a similar way, we can also perform hexadecimal to binary conversion. So, we can convert a number from hexadecimal to binary, by translating every hexadecimal digit into 4 binary digit.

(Refer Slide Time: 24:21)

How are Hexadecimal Numbers Written?

- Using the suffix “H” or using the prefix “0x”.
- Examples:
 - ADDI R1,2AH // Add the hex number 2A to register R1
 - 0x2AB4 // The 16-bit number 0010 1010 1011 0100
 - 0xFFFFFFFF // The 32-bit number for the all-1 string



NPTEL ONLINE CERTIFICATION COURSES

NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, how are hexadecimal numbers written? So, like we have to represent a number as hexadecimal otherwise it will be difficult for the computer to understand. So, to do that you can either add a suffix H or you can use the prefix 0x. So, this here we are representing this 2A is a hexadecimal number, where H is added as a suffix. And here 2AB4 is a hexadecimal number --- this is a 16-bit hexadecimal number and it is added with a prefix 0x. And this is a 32-bit hexadecimal number with all 1s, so it is represented as FFFFFFFF.

(Refer Slide Time: 25:33)

Unsigned Binary Numbers

- An n -bit binary number can have 2^n distinct combinations.
 - For example, for $n=3$, the 8 distinct combinations are:
000, 001, 010, 011, 100, 101, 110, 111 (0 to $2^3-1 = 7$ in decimal).

Number of bits (n)	Range of Numbers
8	0 to 2^8-1 (255)
16	0 to $2^{16}-1$ (65535)
32	0 to $2^{32}-1$ (4294967295)
64	0 to $2^{64}-1$



NPTEL ONLINE CERTIFICATION COURSES

Now, an n-bit binary number can have 2^n distinct combination. For example, if you consider $n = 3$, so we will have 2^3 combination; starting from 0 to 2^n-1 . So, the first is 000, next is 001, 010, 011 and so on till 7. So, similarly for an 8-bit number, it will be 0 to $2^8-1 = 255$; for 16-bit it is 0 to $2^{16}-1$ and so on.

(Refer Slide Time: 26:50)

- An n -bit binary integer:
$$b_{n-1}b_{n-2}\dots b_2b_1b_0$$
- Equivalent unsigned decimal value:
$$D = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \dots + b_22^2 + b_12^1 + b_02^0$$
- Each digit position has a weight that is some power of 2.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES |

So, so n bit binary integers can be represented like this, and the equivalent unsigned decimal number is represented by this; it is multiplied by 2 to the power 0. So, each digit position has a weight that is some power of 2.

(Refer Slide Time: 27:09)

Signed Integer Representation

- Many of the numerical data items that are used in a program are signed (positive or negative).
 - Question: *How to represent sign?*
- Three possible approaches:
 - Sign-magnitude representation
 - One's complement representation
 - Two's complement representation

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES |

Now, let us come to signed integer representation. So, in signed integer representation, how do we represent negative numbers, how do we represent both positive and negative numbers? So, here the question arises how to represent the sign bit. There are three approaches that are followed; one is sign magnitude representation another is one's complement representation and another is two's complement representation.

(Refer Slide Time: 27:42)

(a) Sign-magnitude Representation

- For an n-bit number representation:
 - The most significant bit (MSB) indicates sign (0: positive, 1: negative).
 - The remaining (n-1) bits represent the magnitude of the number.
- Range of numbers: $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$

b_{n-1}	b_{n-2}			b_1	b_0
-----------	-----------	--	--	-------	-------

Sign ← → **Magnitude**

- A problem: Two different representations for zero.
+0: 0 00..000 and -0: 1 00..000

First come to sign-magnitude representation. So, in a sign-magnitude representation for an n-bit number representation, the most significant bit indicates the sign. So, if the most significant bit is 0, then we say it is a positive number; if it is 1 it is a negative number, and the remaining n-1 bits represent the magnitude of that number. So, for an n-bit number, what will be the range that can be represented, it is $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$.

(Refer Slide Time: 28:39)

SIGN MAGNITUDE	
RANGE: $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$	
Example: $n=4$	
RANGE: $-(2^3-1)$ to $+(2^3-1)$	
-7	+7
0000 : +0	1000 : -0
0001 : +1	1001 : -1
0010 : +2	1010 : -2
0011 : +3	1011 : -3
0100 : +4	1100 : -4
0101 : +5	1101 : -5
0110 : +6	1110 : -6
0111 : +7	1111 : -7

So, let us see this for a 4-bit number. So, for a 4-bit number, you can see the range is starting from -7 to +7. So, let us see how we can represent -7 to +7. So, 0 will be same as 0, and so on till 7. And starting from this is -0 because the first bit represent the sign and this is the magnitude, and then we have -1, -2, etc.

So, this is how we can represent the set of numbers for $n = 4$ using sign magnitude representation. So, these are all the magnitude and this is the sign of the number. Now, what is the problem here, the problem here is that you see you have two representations of 0. So, you have -0 and you have +0. So, this is one of the demerit of this method, we have two different representations for 0.

(Refer Slide Time: 30:31)

(b) Ones Complement Representation

- Basic idea:
 - Positive numbers are represented exactly as in sign-magnitude form.
 - Negative numbers are represented in 1's complement form.
- How to compute the 1's complement of a number?
 - Complement every bit of the number ($1 \rightarrow 0$ and $0 \rightarrow 1$).
 - MSB will indicate the sign of the number (0: positive, 1: negative).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us move on with another representation that is 1's complement representation. So, the basic idea here is positive numbers are represented exactly in sign magnitude form; and for the negative numbers these are represented in 1's complement. In 1's complement, in each number if a bit is 1 it will be converted to 0; if it is 0 it will be converted to 1. And the MSB will indicate the sign of the number; for a positive number it will be 0, for a negative number it will be 1.

(Refer Slide Time: 31:09)

Example for n=4

Decimal	1's complement	Decimal	1's complement
+0	0000	-7	1000
+1	0001	-6	1001
+2	0010	-5	1010
+3	0011	-4	1011
+4	0100	-3	1100
+5	0101	-2	1101
+6	0110	-1	1110
+7	0111	-0	1111

To find the representation of, say, -4, first note that

$+4 = 0100$

$-4 = 1\text{'s complement of } 0100 = 1011$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, this is the representation for $n = 4$. From 0 to 7, it will be the same as sign magnitude; and starting from -7 the representation is different. So, let us take the -4. How do you represent 4? We represent 4 as 0100. So, -4 is represented using 1s complement as 1011. But here also the same problem exists; we have two representations of 0.

(Refer Slide Time: 32:04)

- Range of numbers that can be represented in 1's complement:
Maximum :: $+(2^{n-1} - 1)$
Minimum :: $-(2^{n-1} - 1)$
- A problem:
Two different representations of zero.
 $+0 \rightarrow 0\ 000\dots 0$
 $-0 \rightarrow 1\ 111\dots 1$
- Advantage of 1's complement representation:
 - Subtraction can be done using addition.
 - Leads to substantial saving in circuitry.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Here also the range of numbers that can be represented is $+(2^{n-1}-1)$ to $-(2^{n-1}-1)$, but the problem here is we have two representations of +0 and -0. But there are few advantages; that is, using 1's complement subtraction can be done using addition and it leads to substantial saving in circuitry. So, you will be look into all these things in detail in the Arithmetic and Logic Unit.

(Refer Slide Time: 33:13)

(c) Two's Complement Representation

- Basic idea:
 - Positive numbers are represented exactly as in sign-magnitude form.
 - Negative numbers are represented in 2's complement form.
- How to compute the 2's complement of a number?
 - Complement every bit of the number ($1 \rightarrow 0$ and $0 \rightarrow 1$), and then *add one* to the resulting number.
 - MSB will indicate the sign of the number (0: positive, 1: negative).

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us move on. And let us see 2's complement representation. Here the positive numbers are represented exactly the way as sign magnitude form, but the negative numbers are represented in 2's complement form. By 2's complement what we mean? Firstly, we do 1's complement and then we add 1 to the resulting number. So, this is the 2's complement representation. Here also the MSB will indicate the sign of a number; 0 for positive, 1 for negative.

(Refer Slide Time: 33:53)

Example for n=4

Decimal	2's complement	Decimal	2's complement
+0	0000	-8	1000
+1	0001	-7	1001
+2	0010	-6	1010
+3	0011	-5	1011
+4	0100	-4	1100
+5	0101	-3	1101
+6	0110	-2	1110
+7	0111	-1	1111

To find the representation of, say, -4, first note that

$$\begin{aligned} +4 &= 0100 \\ -4 &= 2\text{'s complement of } 0100 = 1011 + 1 \\ &= 1100 \end{aligned}$$

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, this is the representation for $n = 4$. Now, we see that from 0 to 7 the representation is same. The negative number representations are also shown. And the main advantage is that here there is only single representation of 0. So, what is the range of number that we can have, one more than the previous one.

(Refer Slide Time: 35:01)

- Range of numbers that can be represented in 2's complement:
Maximum :: $+ (2^{n-1} - 1)$
Minimum :: -2^{n-1}
- Advantage of 2's complement representation:
 - Unique representation of zero.
 - Subtraction can be done using addition.
 - Leads to substantial saving in circuitry.
- Almost all computers today use 2's complement representation for storing negative numbers.

22

So, the range of numbers that can be represented in 2's complement is -2^{n-1} to $+(2^{n-1}-1)$. We have a unique representation of 0, subtraction can be done using addition, and it leads to substantial saving in circuitry. Almost all computers today use 2's complement representation for storing negative numbers.

(Refer Slide Time: 35:40)

- Some other features of 2's complement representation
 - Weighted number representation, with the MSB having weight -2^{n-1} .

				b_{n-1}	b_{n-2}						b_1	b_0
-2^{n-1}	2^{n-2}							2^1	2^0			

$$D = -b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \dots + b_22^2 + b_12^1 + b_02^0$$
 - Shift left by k positions with zero padding multiplies the number by 2^k .

00010011 = +19 :: Shift left by 2 :: 01001100 = +76

11100011 = -29 :: Shift left by 2 :: 10001100 = -116
--

Now, let us also see some more features of 2's complement, which will be necessary for the further lecture units. It represents a weighted number representation where MSB having the weight -2^{n-1} . So, what does it represent, it represents that all these numbers will have $2^0, 2^1, 2^{n-2}$, but the last one the magnitude will be -2^{n-1} .

(Refer Slide Time: 36:22)

-20 : 101100 in 2's complement

$$\begin{array}{r}
 -2^5 2^4 2^3 2^2 2^1 2^0 \\
 | 0 1 1 0 0 \\
 \hline
 \end{array} \equiv -2^5 + 2^3 + 2^2 = -20$$

$$\begin{array}{r}
 -2^6 2^5 2^4 2^3 2^2 2^1 2^0 \\
 | 1 0 1 1 0 0 \\
 \hline
 \end{array} \equiv -2^6 + 2^5 + 2^3 + 2^2 = -20$$

$$\begin{array}{r}
 -2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 \\
 | 1 1 0 1 1 0 0 \\
 \hline
 \end{array} \equiv -2^7 + 2^6 + 2^5 + 2^3 + 2^2 = -20$$

⇒ SIGN EXTENSION IN 2's COMPLEMENT

So, let us see this. So, how we can represent -20 in 2's complement? This is 2^0 , this is 2^1 , this is 2^2 , this is 2^3 , 2^4 , and -2^5 . So, for any negative number that we represent using 2's complement the most significant bit will have 1.

Now, let us add an extra 1 to it, and let us see that whether this changes or not. So, we added 1 more in the most significant bit and let us calculate it once more. The value remains -20. So, even if you add 1 more 1 into it, this will not change, this will be represented as $-2^7 + 2^6$, and this will remain as -20. So, this is what we call sign extension in 2's complement.

Now, what happens if we shift left by k position with 0 padding we are shifting left with 0 padding? What it does it multiplies the number by 2^k . So, we can take an example where this is the number what we say we shift left. So, we are shifting it left. So, this number is shifted to left by 2. So, we will shift it once and we shift it once more. If you shift it two times, it will become 01001100 and this is equivalent to +76 --- basically we have multiplied by 2^2 that is 4. Similarly, if this is -29, we shift left by 2 and we get -116. So, shifting left meaning multiplication by 2.

(Refer Slide Time: 39:24)

c) Shift right by k positions with sign bit padding divides the number by 2^k .

$00010110 = +22 :: \text{Shift right by 2} :: 00000101 = +5$

$11100100 = -28 :: \text{Shift right by 2} :: 11111001 = -7$

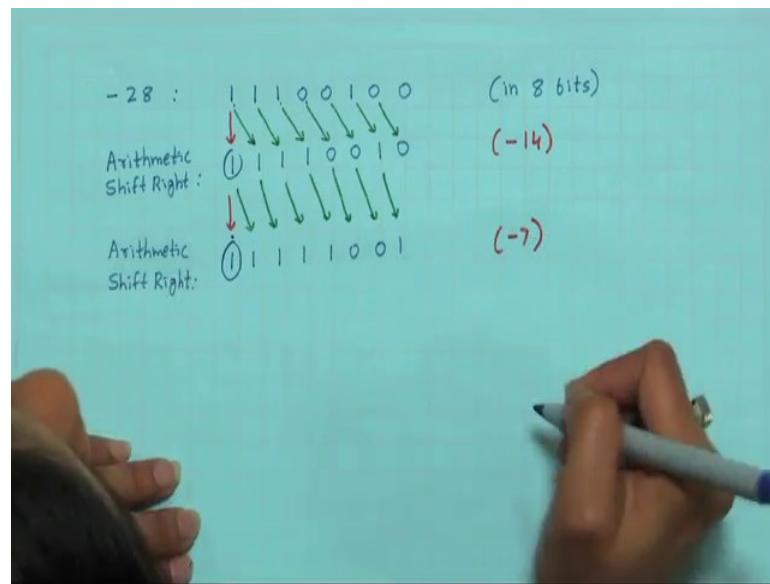
d) The sign bit can be copied as many times as required in the beginning to extend the size of the number (called *sign extension*).

$X = 00101111$ (8-bit number, value = +47)
Sign extend to 32 bits:
`00000000 00000000 00000000 00101111`

$X = 10100011$ (8-bit number, value = -93)
Sign extend to 32 bits:
`11111111 11111111 11111111 10100011`

Now, we will also see shift right by k positions with sign bit padding, what will happen divides the number by 2^k . Let us take an example. So, this example is where we shift right by 2 positions and then what is happening. So, here you have a number +22, once we divide by 2 we will get 11. Again we divide by 2 we will get 5. Similarly, -28.

(Refer Slide Time: 40:08)



So, let us take this example of -28 here. So, -28 can be represented as 11100100 in 8-bits. And then we do an arithmetic right shift. When we do an arithmetic right shift the most whatever value is a most significant bit that comes here and all other bits are shifted 1111 position see that is what we have done. So, all other bits are shifted to 1111 position and then this bit comes in the most significant bit and this representation is nothing but -14. Similarly, we are doing arithmetic shift right once more and same 1 bit comes in this position. So, 1 is coming here and all other bits are getting shift to 111 position to the right. So, we get 1111001, this is equivalent to -7. So, this is how we perform shift right by k position with sign bit and which actually divides the number by 2^k .

Now, this sign bit can be copied as many times as required I have already shown you this in the beginning to extend the size of the number called, this is called sign extension. So, this is the 8-bit number which is +47; sign extension to 32-bit you add as many number as 0 as required it will still remain +47. Similarly, a negative number -93, we can add as many 1 in the beginning to extend it to 32-bit the number will remain as -93 only. So, we can see that in 2's complement how we can represent it and how we can do sign extension.

So, we come to the end of lecture 6, where we briefly talked about number systems such that it will be easy for you to get into how instructions are represented and get executed in next lectures.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 07
Instruction Format And Addressing Modes

(Refer Slide Time: 00:27)

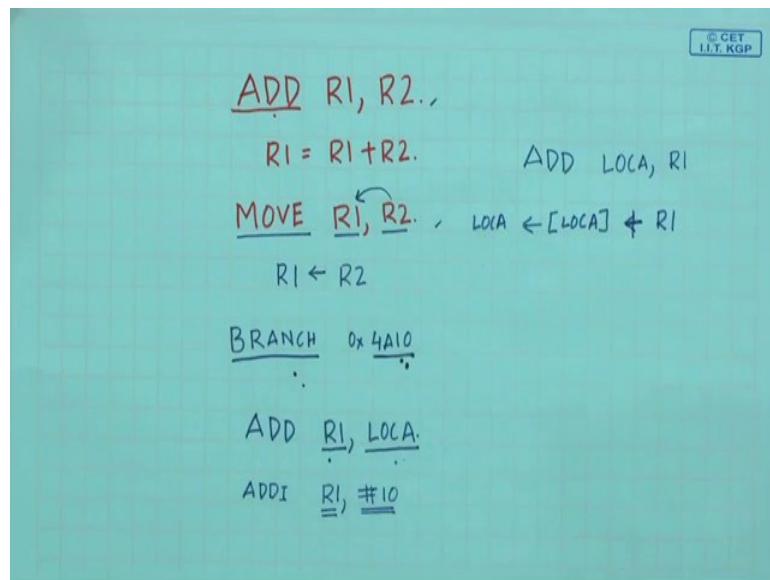
Instruction Format

- An instruction consists of two parts:-
 - a) **Operation Code or Opcode**
 - Specifies the operation to be performed by the instruction.
 - Various categories of instructions: data transfer, arithmetic and logical, control, I/O and special machine control.
 - b) **Operand(s)**
 - Specifies the source(s) and destination of the operation.
 - Source operand can be specified by an immediate data, by naming a register, or specifying the address of memory.
 - Destination can be specified by a register or memory address.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Welcome to the next lecture on instruction format and addressing modes. So, what do you mean by instruction format? We know what is an instruction. And instruction format is what it comprises of: two parts --- first part is the Opcode, and the next part is the Operand.

(Refer Slide Time: 00:57)



So, what is an opcode? Take an example ADD R1,R2. The opcode specifies the operation to be performed. The operation here is adding two register values, and result stored back in some register. So, R1 will store $R1 + R2$. ADD is the operation code that specifies the operation to be performed by the instruction and we can have various categories of instruction. This is an arithmetic instruction.

We can have an instruction called MOVE. What this instruction will do? This instruction will move the data from R2 to R1. So, here R1 will have the value of R2. Such kind of instruction is called data transfer instructions. We can have other instructions; this is arithmetic instruction, this is data transfer instruction, we can have other branching instruction. What is branching instruction? We can have an instruction called branch to some location, say, 16-bit hexadecimal number 4A10.

So, branch to this particular location will move to this particular location, and whatever data is there in this particular location it will be added with PC and it will calculate the next instruction that needs to be executed because branch to this location means in some particular location some instruction is present which we need to execute. So, this part of the instruction we call it an opcode, and this part is the operand.

Now, see what can be an operand; it specifies either a single source or there can be two source and a destination of the operation. And source operand can be specified by an immediate data or by naming a register. Just now I have shown how we can just give the

name of the register or specify a memory address like ADD R1,LOCA. Here we are specifying one operand is a register, another operand a memory location. So, an operand can be a register, a memory location or an immediate value (here 10).

So, this kind of operand can also be specified, but this operand cannot be the destination. A destination operand should always be either a register or a memory location like LOCA. So, instruction consists of two parts; one is operation code, another will be operand.

(Refer Slide Time: 06:13)

- Number of operands varies from instruction to instruction.
- Also for specifying an operand, various *addressing modes* are possible:
 - Immediate addressing
 - Direct addressing
 - Indirect addressing
 - Relative addressing
 - Indexed addressing, and many more.

The number of operands varies from instruction to instruction. We can have a zero address instruction, we can have a one address instruction, we can even have a two address or even have a three address instruction. So, number of operands that are present in an instruction may vary. Also while specifying an operand, we need to know the various addressing modes. So, coming to what is addressing modes we will be looking in more details. Addressing mode actually is a way by which the location of the operand is specified in the instruction. There can be many possible addressing modes: immediate, direct, indirect, relative and index, and many more. We will be seeing a few of them.

(Refer Slide Time: 07:15)

The slide is titled "Instruction Format Examples". It shows five different instruction formats with their descriptions:

- Format 1: opcode [] Implied addressing: NOP, HALT
- Format 2: opcode [] memory address 1-address: ADD X, LOAD M
- Format 3: opcode [] memory address [] memory address 2-address: ADD X,Y
- Format 4: opcode [] register [] memory address Register-memory: ADD R1,X
- Format 5: opcode [] register [] register [] register Register-register: ADD R1,R2,R3

The slide footer includes the IIT Kharagpur logo, the text "IIT KHARAGPUR", the NPTEL logo, and "NPTEL ONLINE CERTIFICATION COURSES". A video frame of a person speaking is visible on the right side.

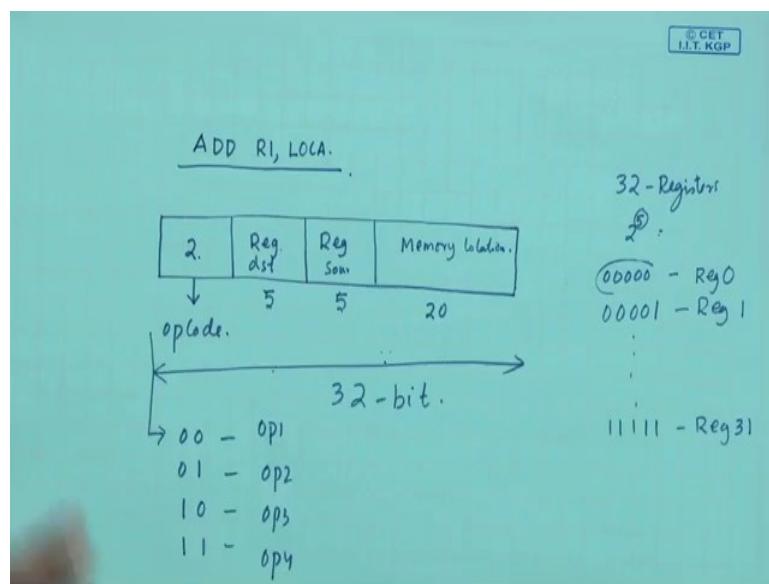
Now, let us see this instruction format. If we have just the opcode, let us take some example of NOP, NOP means no operation. No operation instruction specifies the processor that no operation will be performed at this particular cycle. HALT will specify to halt the execution. We can have one-address instruction where only one address is specified along with the opcode. We can have two-address instruction where we can specify two operands. We can have two-address instruction where both operands can be memory locations. We can have another instruction where one can be register another can be memory operation, or we can have a three-address instruction where all are registers. So, these are various instruction formats.

(Refer Slide Time: 08:39)

- Suppose we have an ISA with 32-bit instructions only.
 - Fixed size instructions make the decoding easier.
- Some instruction encoding examples are shown.
 - Assume that there are 32 registers R0 to R31, all of 32-bits.
 - 5-bits are required to specify a register.

Now, consider a 32-bit instruction example. Suppose our instruction set architecture is having only 32-bit instructions; fixed size instructions make the decoding easier. Let us understand this statement.

(Refer Slide Time: 09:17)



I am giving a example this is not corresponding to any real machine. Let us say we have an instruction ADD R1,LOCA. So, in this 32-bit instruction some bits will be reserved for opcode, some bits will be reserved for register, etc. So, this can be register destination this can be register source, and this can be your memory location. If this is so, the total is

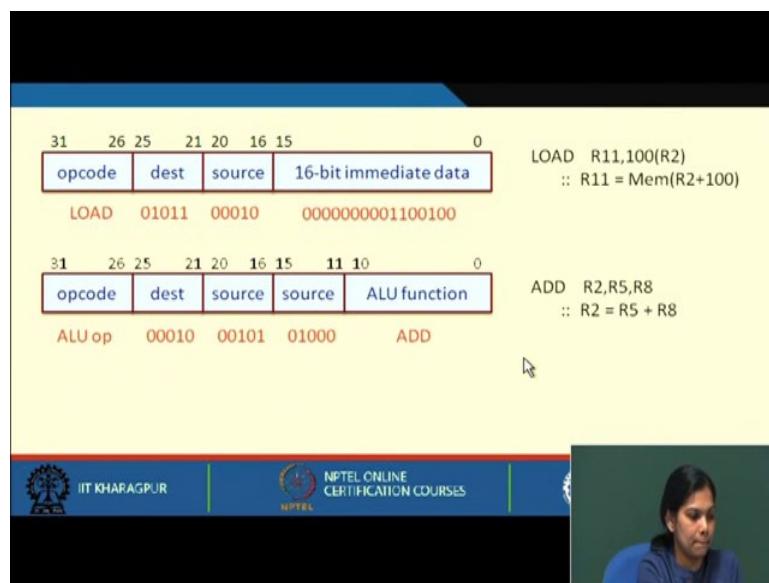
32-bit, and now we have to specify the bits. Let us say we have a total of 32 registers. Then 5-bits will be required to represent a single register.

So, this 00000 will be register 0, 00001 will be register 1 and so on, and the last register will be register 31. So, 5-bits will be required to specify one of the registers. Also suppose we can specify a memory location in 20 bits. And then how many bits are remaining for the opcode? -- we have 2 bits left for opcode.

So, if we have two bit left for opcode we can have maximum of four possibilities 00, 01, 10 and 11. There can be four operations only. I am just giving you as example and this does not correspond to a real machine, but rather just to give you an idea that how the instruction format will looks like.

Fixed size instructions make the decoding easier. We can check the opcode to know the operation. We can have various kinds of instruction like here we have only one memory location; if we afford we can have two memory locations.

(Refer Slide Time: 14:11)



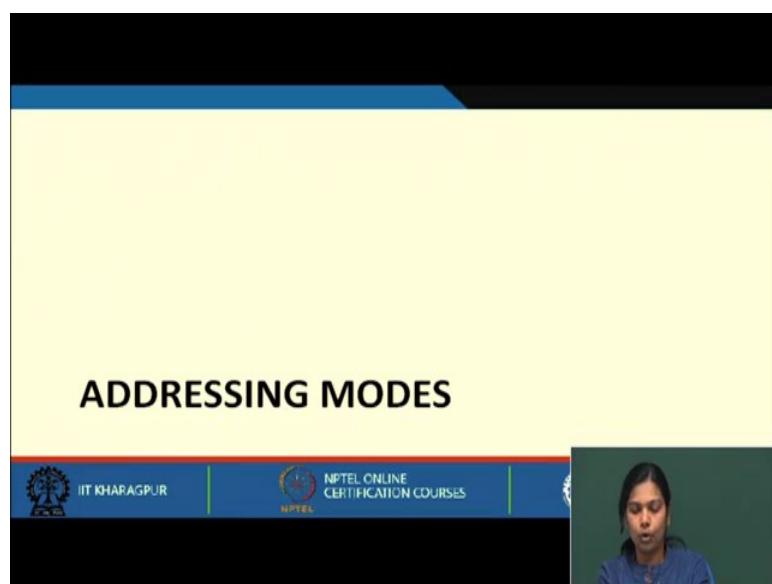
Now let us see some instruction formats. Let say this is the format where you have these bits for opcode 26, 27, 28, 29, 30 and 31 --- 6-bits. And 5-bit for register and 16-bit immediate data. Let us take an example LOAD R11,100(R2). We are loading the content from memory location pointed by $100 + R2$. So, first we have to add 100 plus R2, and then we have to put this in MAR, activate the read control signal, get the value of it. And

then we load it into R11. These are the following steps that will be required to execute this instruction.

So, let us see where all it will be placed. This is the destination register R11. So, R11 will be placed here. R11 is 01011. Opcode for load will be loaded in first 6 bits. This is the source operand; one of the source operand is R2; it is loaded here. And 100 is the immediate value which is loaded here. So, this is how we encode this instruction into binary. Again load can be having some value, say 000001.

Let us move on. Now, if we have limited opcode facility, say, we can only have 64 operations that are possible with 6-bit. So, if you want to increase that, what we can do is that here this opcode will give you that what kind of function it will be taking care of like it can be an ALU function, it can be a data transfer operation, etc. And the exact function will be specified by these 11 bits. This is ADD R2,R5,R8. So, the contents of R8 and R5 will be added and will be stored in R2. Now, we see that, this is an ALU operation, this is the destination R2. here are two sources --- first source is R5 and the next source is R8 and this ALU function ADD.

(Refer Slide Time: 17:44)



Now, moving on to addressing modes. Now, see what we have understood till now that we can see this that we have an instruction, and by seeing we are saying that this is a register, this is a memory operation, memory location. But again you have to instruct the computer that see this is a register, this is a memory location then only the processor will

do the required thing, it will go to the memory location, get the data, it will go to a particular register and get the data, etc.

(Refer Slide Time: 18:43)

What are Addressing Modes?

- They specify the mechanism by which the operand data can be located.
- Some ISA's are quite complex and supports many addressing modes.
- ISA's based on load-store architecture are usually simple and support very limited number of addressing modes.
- Various addressing modes exist:
 - Immediate, Direct, Indirect, Register, Register Indirect, Indexed, Stack, Relative, Autoincrement, Autodecrement, Based, etc.
 - Not all processors support all addressing modes.
 - We shall briefly look at the common addressing modes and how they work.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, what are addressing modes? Addressing modes are the ways by which the location of an operand is specified in an instruction. So, it specifies that the location of an operand in the instruction. How the location of the operand is specified, whether we have to get it from a register or from memory location, etc. Some instruction set architectures are quite complex and supports many addressing modes. But instruction set architectures that are based on load-store usually support very simple addressing modes. So, this is very important. If you want to have complex addressing modes, some of the instructions or architecture do have it, but the load-store architecture basically supports very limited number of addressing modes.

There are various addressing modes that exist: immediate, direct, indirect, register, register indirect, indexed, stack, relative, auto increment, auto decrement, based, etc. However, all architectures will not have all the addressing modes. So, we shall first look into some common addressing modes, and how do they work.

(Refer Slide Time: 20:34)

Immediate Addressing

- The operand is part of the instruction itself.
 - No memory reference is required to access the operand.
 - Fast but limited range (because a limited number of bits are provided to specify the immediate data).
- Examples:
 - ADD #25 // ACC = ACC + 25
 - ADDI R1,R2,42 // R1 = R2 + 42

opcode immediate data

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

NPTEL

A video frame showing a woman speaking.

Coming to immediate addressing mode, here the operand is part of the instruction itself. So, you need not have to go anywhere to get the operand, rather your operand is a part of your instruction. So, no memory access is required to get the operand and it is fast, but limited range because you can only specify a limited number using immediate mode like ADD #25. When we write #, it means is an immediate data. So, when we write ADD #25 that means, 25 will be added with accumulator and the result will be stored back in the accumulator.

Here, we have an immediate data 42, which is added to R2 and result stored in R1.

(Refer Slide Time: 22:10)

The slide has a yellow background and a black header bar. The title 'Direct Addressing' is centered in bold black font. Below the title is a bulleted list:

- The instruction contains a field that holds the memory address of the operand.

A diagram below the list shows a horizontal box divided into two sections: 'opcode' and 'operand address'.

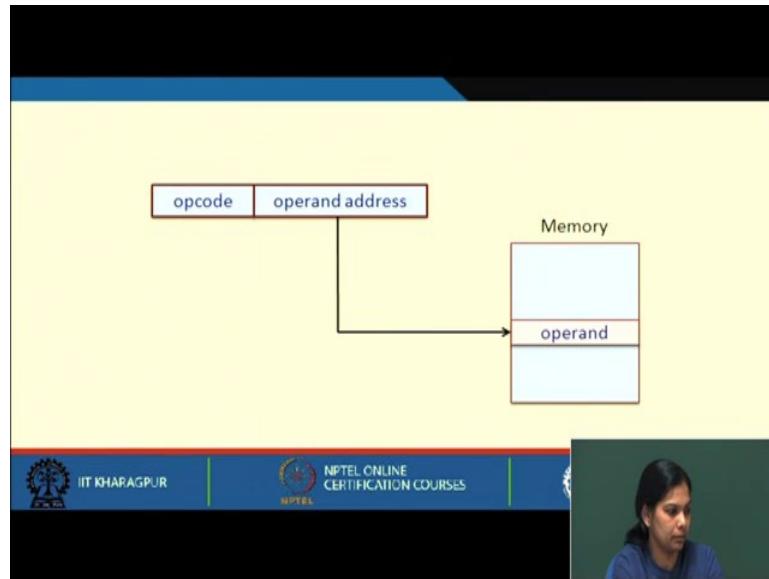
- Examples:
 - ADD R1,20A6H // $R1 = R1 + \text{Mem}[20A6]$
- Single memory access is required to access the operand.
 - No additional calculations required to determine the operand address.
 - Limited address space (as number of bits is limited, say, 16 bits).

At the bottom of the slide, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and a video camera icon representing video recording.

Moving on with direct addressing mode, here the instruction contains a field that holds the memory address of the operand. So, here the content of 20A6 will be the operand that we are looking for. Like here ADD R1,20A6 means whatever content is in 20A6 will be added with R1 and result stored back in R1.

Now, here how many memory operations are required? So, we have to go to this particular address and fetch the instruction. So, going to this particular address we need one more memory access to access the operand, no additional calculation is required to determine the operand address and limited address space. So, if this address space is 16-bit, it is limited. So, we can only have direct addressing within that 16-bit address span.

(Refer Slide Time: 23:45)



So, this is how pictorially we can show -- this is the opcode, this is the operand address. You go to that address you get that operand, this is direct addressing.

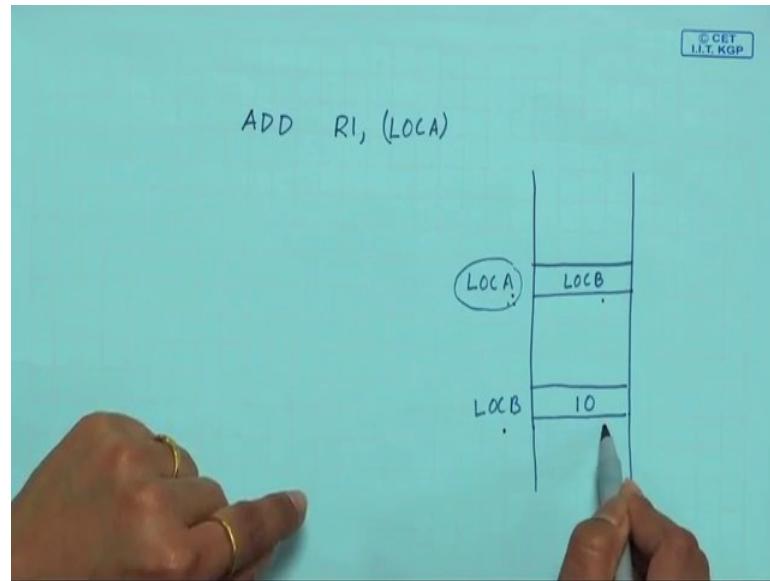
(Refer Slide Time: 24:01)

Indirect Addressing

- The instruction contains a field that holds the memory address, which in turn holds the memory address of the operand.
- Two memory accesses are required to get the operand value.
- Slower but can access large address space.
 - Not limited by the number of bits in operand address like direct addressing.
- Examples:
 - ADD R1,(20A6H) // R1 = R1 + (Mem[20A6])

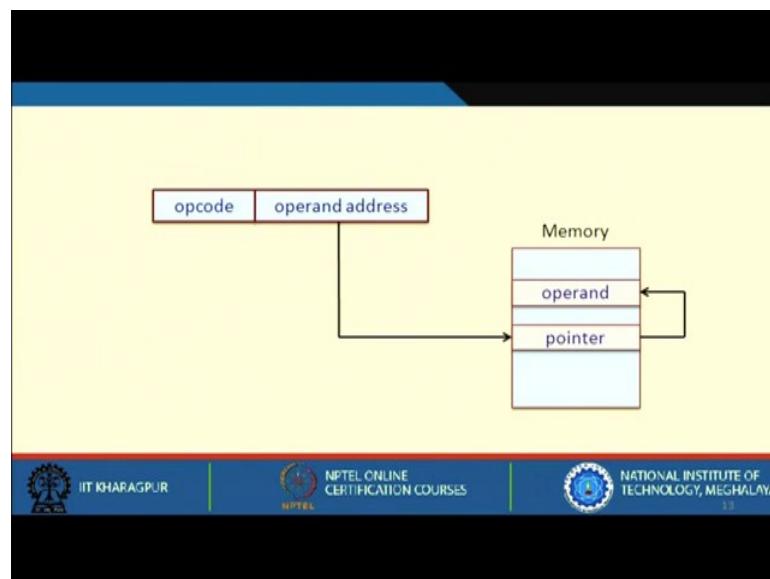
Let us move on indirect addressing. The name itself suggests when it is indirect that means, in the instruction what it contains, it contains a field that holds the memory address which in turn holds the memory address of the operand. So, let us see this with an example.

(Refer Slide Time: 21:41)



Let us say we have an instruction ADD R1,(LOCA). LOCA contain another address, say LOCB. And now you will not get your operand from LOCA, rather you will get your operand from LOCB. In this particular case, you have to go to this location, this location will give you another location and you go to that particular location that will give you the value. So, in this case you if you can see that you are requiring two memory accesses to get the operand value. This is slower, but can access larger address space. It is not limited to number of bits in the operand address like direct addressing.

(Refer Slide Time: 26:27)



So, this is the operand address. First this is a pointer as I explained and then from there you go to another address which will give you the exact operand.

(Refer Slide Time: 26:40)

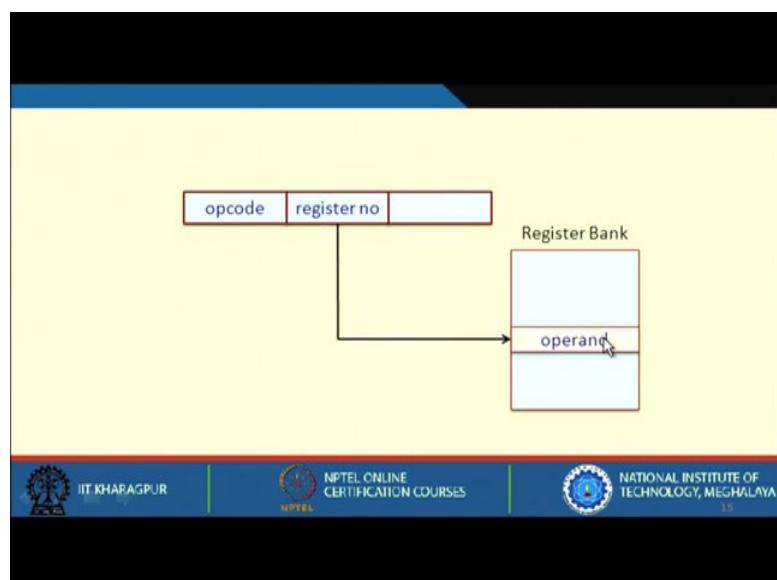
Register Addressing

- The operand is held in a register, and the instruction specifies the register number.
 - Very few number of bits needed, as the number of registers is limited.
 - Faster execution, since no memory access is required for getting the operand.
- Modern load-store architectures support large number of registers.
- Examples:
 - ADD R1,R2,R3 // R1 = R2 + R3
 - MOV R2,R5 // R2 = R5

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Moving on with register addressing. The operand is held in a register, and the instruction specifies the register number. Very few bits are needed, as the number of registers is limited. Faster execution is possible, and no memory access is required for getting the operand. The load-store architecture supports large number of registers.

(Refer Slide Time: 27:31)



So, as I said this is the register bank. The register number is specified in the instruction and you go to that particular register to get the operand.

(Refer Slide Time: 27:44)

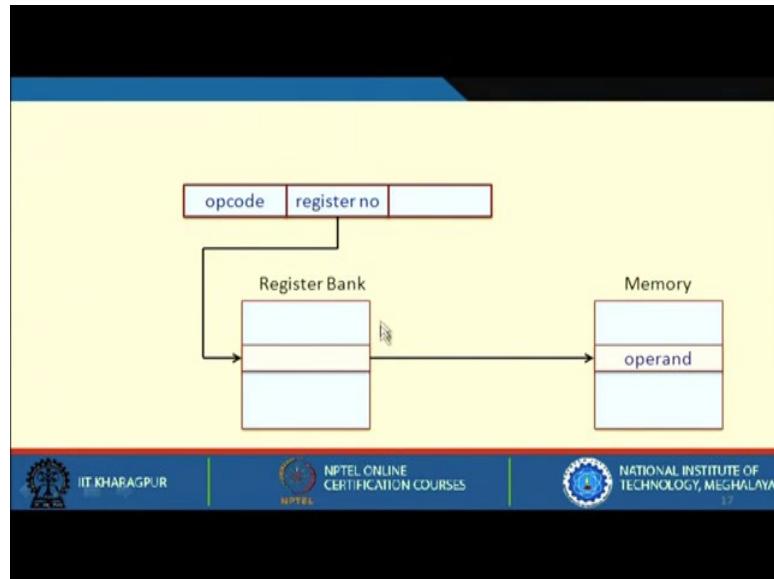
The slide has a blue header bar. The main title 'Register Indirect Addressing' is centered in a yellow box. Below the title is a bulleted list:

- The instruction specifies a register, and the register holds the memory address where the operand is stored.
 - Can access large address space.
 - One fewer memory access as compared to indirect addressing.
- Example:
 - ADD R1,(R5) // PC = R1 + Mem[R5]

At the bottom of the slide, there is a footer bar with three logos: IIT Kharagpur, NPTEL, and NIT Meghalaya, along with their respective names.

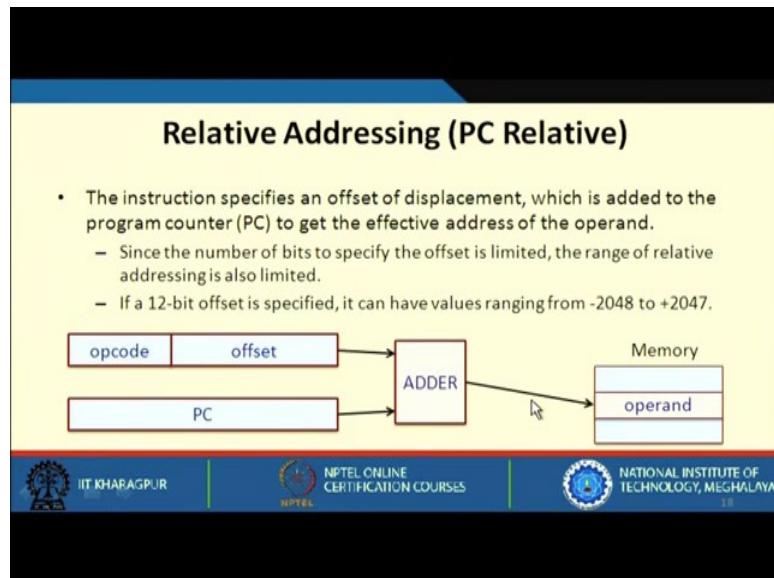
Moving on with register indirect addressing mode, here the instruction specifies a register and the register holds the memory address where the operand is stored. So, this is also a kind of indirect addressing, where instead of a memory location here we are putting the memory address in a register. And then this register holds what it holds a memory address, but not the operand, you have to go to that memory address to access the operand. One fewer memory access is required as compared to indirect addressing mode.

(Refer Slide Time: 28:59)



So, just see here this register will give you a memory location, and this memory location is fetched from the memory, the data from this memory location that is in the register is fetched from the memory, and we get the operand. So, this is register indirect addressing. In the register, we are putting an address and that address stores the operand.

(Refer Slide Time: 29:31)



Relative addressing is always with respect to PC. In this kind of addressing modes the instruction specifies an offset or displacement, which is added to the program counter to get the effective address of the operand. Since the number of bits to specify the offset is

limited, the range of relative addressing is also limited. So, if a 12-bit offset is specified, it can have values ranging from -2048 to +2047.

Let us understand this relative addressing. With respect to PC that means, relative to PC how much you can go. So, in branch instruction we specify a branch address. To go to that particular location, you have to load that particular address in PC. So, this is an offset that is given in the instruction that is added or subtracted depending on where you are branching. That particular branch address will be added with the content of the PC. So, in such kind of cases we require relative addressing mode.

So, here you have an opcode, this is the offset. The offset is added with the content of PC and then where you go and you fetch the operand.

(Refer Slide Time: 31:45)

Indexed Addressing

- Either a special-purpose register, or a general-purpose register, is used as *index register* in this addressing mode.
- The instruction specifies an offset of displacement, which is added to the index register to get the effective address of the operand.
- Example:
 - LOAD R1,1050(R3) // R1 = Mem[1050+R3]
- Can be used to sequentially access the elements of an array.
 - Offset gives the starting address of the array, and the index register value specifies the array element to be used.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Moving on with indexed addressing mode. In the previous case, we have seen in relative addressing modes, the content of PC is added with the offset value. Now, here either a special-purpose register or a general-purpose register is used as index register.

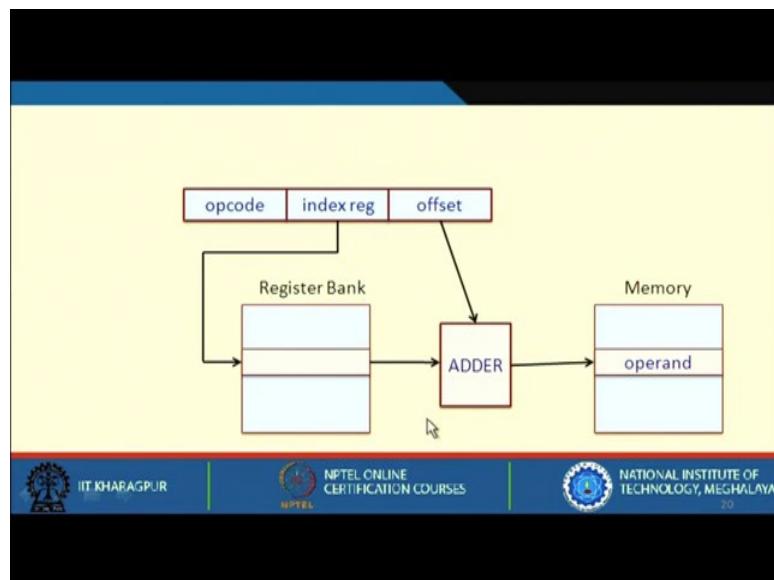
In this addressing mode, we can access an array. Array is a set of consecutive memory location. So, if you load a particular address, you know the first address of an array, how will you go to the next address, next address, and so on. In a similar fashion here in index addressing mode, you add that general-purpose register value it can be the used as index

register and this instruction specifies an offset or displacement that is added to the index register to get the effective address of the operand.

So, let us see with this example. Now, see 1050(R3); that means, content of R3 will be added with 1050, and then that location will give the operand. So, 1050 is added with R3, we get a value that value from which address in memory we get the operand, and it can be used to sequentially access the elements of an array.

So, we load the first address and then we move to the next, next, next address by adding an offset to it. So, offset gives the starting address of the array and the index register value specifies the array element to be used; the first can be 0th element, then the next, then next and so on.

(Refer Slide Time: 34:11)



So, here this index register you get this added with this offset, this particular address is searched and we get the operand from there.

(Refer Slide Time: 34:23)

Stack Addressing

- Operand is implicitly on top of the stack.
- Used in zero-address machines earlier.
- Examples:
 - ADD
 - PUSH X
 - POP X
- Many processors have a special register called the stack pointer (SP) that keeps track of the stack-top in memory.
 - PUSH, POP, CALL, RET instructions automatically modify SP.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Next, come to stack addressing. In stack addressing, we already know the operand is implicitly on the top of the stack and it is used in zero-address machines much earlier. The first two elements on top of the stack will be taken out, will be added, and stored back there. PUSH X will push the X value into top of the stack. POP X will take out the top value to this location and store in X. Many processors have a special register called a stack pointer that keeps track of the top of the stack.

(Refer Slide Time: 35:14)

Some Other Addressing Modes

- **Base addressing**
 - The processor has a special register called the *base register* or *segment register*.
 - All operand addresses generated are added to the base register to get the final memory address.
 - Allows easy movement of code and data in memory.
- **Autoincrement and Autodecrement**
 - First introduced in the PDP-11 computer system.
 - The register holding the operand address is automatically incremented or decremented after accessing the operand (like `a++` and `a--` in C).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

There are some other addressing modes as well, like base addressing mode. So, in base addressing mode, the processor has a special register called base register or segment register and then what happens here is all operand addresses generated are added to the base register to get the final memory address. Let us say the processor generates the address from 0, 1, 2, 3 and then you have stored some address in base register let say 1024, so 1024 will be added with that particular address. So, this is what base addressing means and it allows easy movement of code and data in memory.

We can also have another addressing mode, autoincrement and autodecrement addressing mode. It was first introduced in a PDP-11 computer, which was one of the most popular minicomputers in the 1980s. Autoincrement and autodecrement means that if you load a register with some address you can auto increment it you can access that value then you increment it, or auto decrement means you access the value then you decrement it. So, either way you can implement this. So, auto increment, auto decrement we have also seen in `a++` and `a--`, auto decrement and auto increment operators. So, in the similar way we can have such kind of addressing modes also.

So, we come to the end of lecture 7. What we have seen in this lecture is that addressing modes which are very important and the instruction format. We will move on in the next lecture where we will see the types of architectures.

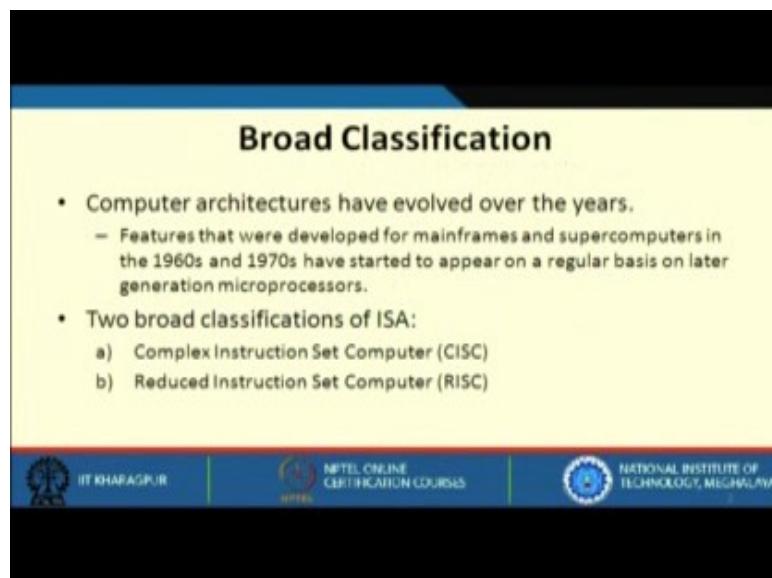
Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 08
CISC And RISC Architecture

Welcome to the next lecture on CISC and RISC architecture. Till now what we have seen how an instruction gets executed, what kind of instruction format is available and what kind of addressing modes are used. So, now we will be looking into that architecture where we can divide those sets into some groups, that is, RISC and CISC.

(Refer Slide Time: 01:06)



Broad Classification

- Computer architectures have evolved over the years.
 - Features that were developed for mainframes and supercomputers in the 1960s and 1970s have started to appear on a regular basis on later generation microprocessors.
- Two broad classifications of ISA:
 - a) Complex Instruction Set Computer (CISC)
 - b) Reduced Instruction Set Computer (RISC)

IIT Kharagpur | NITEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

This is a broad classification. Computer architectures have evolved over the years. And there are many features that were developed for mainframes and for supercomputers in the 1960s and 70s and that started to appear as regular features on the later generation of microprocessors. Based on that we have so many features; they are broadly classified into complex instruction set computer called CISC, and reduced instruction set computer called RISC.

(Refer Slide Time: 01:49)

CISC versus RISC Architectures

- Complex Instruction Set Computer (CISC)
 - More traditional approach.
 - Main features:
 - Complex instruction set
 - Large number of addressing modes (R-R, R-M, M-M, indexed, indirect, etc.)
 - Special-purpose registers and flags (sign, zero, carry, overflow, etc.)
 - Variable-length instructions / Complex instruction encoding
 - Ease of mapping high-level language statements to machine instructions
 - Instruction decoding / control unit design more complex
 - Pipeline implementation quite complex

IIT KHARAGPUR | NPTEL, ONLINE CERTIFICATION COURSES | NPTEL

Now, let us see how CISC and RISC architectures differ. Coming to CISC it is a more traditional approach. So, the main features of this complex instruction set computers are they have complex instruction set. They also have a large number of addressing modes. They also have some special-purpose registers and flags that are used to carry out various operations, and the instructions are not all of same length. So, they have variable length instructions.

So, as I have already told earlier that if you have a fixed-length instruction it becomes easy for encoding and decoding, but if you have variable-length instruction then the encoding needs to be more complex. And it also takes more time for decoding, because you have to know each of the bits and based on that a particular action will be performed. So, ease of mapping high-level language statement to machine code is possible in CISC, but instruction decoding and control unit design are much more complex. And of course, if you have variable length instruction, pipeline implementation will also be quite complex.

(Refer Slide Time: 03:40)

- CISC Examples:

- IBM 360/370 (1960-70)
- VAX-11/780 (1970-80)
- Intel x86 / Pentium (1985-present)

Only CISC instruction set that survived over generations.

- Desktop PC's / Laptops use these.
- The volume of chips manufactured is so high that there is enough motivation to pay the extra design cost.
- Sufficient hardware resources available today to translate from CISC to RISC internally.

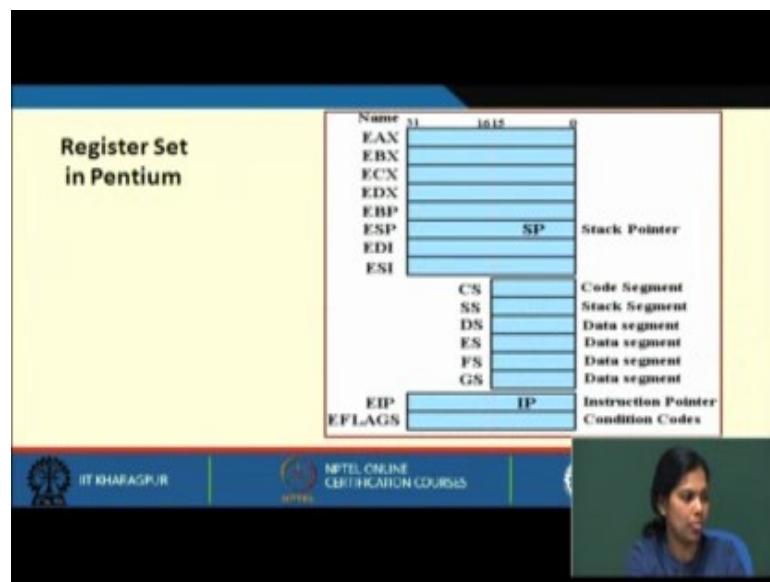
NPTEL ONLINE CERTIFICATION COURSES

IIT KHARASPUR

Now, CISC machines emerged in 60's and 70's --- we have seen IBM 360, 370, VAX-11 were popular in seventies and eighties. And from 1985 and till present we have Intel architectures that use CISC architecture; it follows a CISC architecture and has survived over generations. So, over the generation, if you see any CISC machine based on Intel; the desktop PCs, laptops, etc. But in the newer machines you have newer features, but still they are backward compatible.

And this is all possible because the volume of chips that is manufactured is much, much high. So, you can see that there is enough motivation to pay this extra cost of the design although you are paying some extra thing for this CISC architecture. And there are sufficient hardware resources that are available to translate from CISC to RISC internally.

(Refer Slide Time: 05:34)



So, this is the register set in Pentium. Here you can see that it does not have a large number of registers, although they have got some special-purpose registers like code segment, stack segment, data segment. Also there is program counter which we call instruction pointer, they have some conditional code flags, and there are several other registers to perform different kind of jobs.

(Refer Slide Time: 06:05)

Addressing Modes in VAX	Addressing Mode	Example	Micro-operation
	Register direct	ADD R1,R2	R1 = R1 + R2
	Immediate	ADD R1,#15	R1 = R1 + 15
	Displacement	ADD R1,220(R5)	R1 = R1 + Mem[220+R5]
	Register indirect	ADD R1,(R3)	R1 = R1 + Mem[R3]
	Indexed	ADD R1,(R2+R3)	R1 = R1 + Mem[R2+R3]
	Direct	ADD R1,(1000)	R1 = R1 + Mem[1000]
	Memory indirect	ADD R1,@(R4)	R1 = R1 + Mem[Mem[R4]]
	Autoincrement	ADD R1,(R2)+	R1 = R1 + Mem[R2]; R2+=1
	Autodecrement	ADD R1,(R2)-	R1 = R1 + Mem[R2]; R2-=1
	Scaled	ADD R1,50(R2)[R3]	R1 = R1 + Mem[50+R2+R3*d]

These are the addressing modes that are present in VAX machine. This was a very popular machine in the 80s. So, just see how many addressing modes are present, starting

with register direct, immediate, displacement, register indirect, indexed, direct, memory indirect, auto increment and auto decrement and scaled. So, the VAX machine included a huge set of addressing modes.

(Refer Slide Time: 06:43)

- Reduced Instruction Set Computer (RISC)
 - Very widely used among many manufacturers today.
 - Also referred to as *Load-Store Architecture*.
 - Only LOAD and STORE instructions access memory.
 - All other instructions operate on processor registers.
 - Main features:
 - Simple architecture for the sake of efficient pipelining.
 - Simple instruction set with very few addressing modes.
 - Large number of general-purpose registers; very few special-purpose.
 - Instruction length and encoding uniform for easy instruction decoding.
 - Compiler assisted scheduling of pipeline for improved performance.

Now, coming to reduced instruction set computer which is used in most of the computers today; not only computers, also the processors for microcontrollers follow RISC architecture. This is also referred as load-store architecture. So, by load-store architecture we already discussed that only load and store instruction can be used to access the memory and all other instruction will be operated on register.

So, only when memory data is required, you have to load it from the memory; once the data are available then you can access it. The main features of architecture is very simple for the sake of efficient pipeline. I already discussed very briefly about pipeline that will be discussed in more detail later, but we must know that this is one of the main motivation of RISC; that the RISC architecture is simple for the sake of pipeline.

Simple instruction set is used and very few addressing modes are used. It does not support variety of addressing modes; but it has a large number of general purpose registers. And it does not have many special function registers. Instruction length and encoding is uniform for easy decoding. See if you encode the instruction all the instruction in a specific fashion then the decoding will be much easier. And compiler

assisted scheduling of pipeline for improved performance will be there, you will be seeing this particular feature later.

(Refer Slide Time: 09:10)

– RISC Examples:

- CDC 6600 (1964)
- MIPS family (1980-90)
- SPARC
- ARM microcontroller family

- Almost all the computers today use a RISC based pipeline for efficient implementation.
 - RISC based computers use compilers to translate into RISC instructions.
 - CISC based computers (e.g. x86) use hardware to translate into RISC instructions.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, the popular RISC machine is CDC 6600 that came in 1964. So, RISC is existing since 60s and this is more now that people have moved to RISC architecture. The ARM microcontroller family is all having RISC even some other microcontroller families. So, almost all computers today use RISC based pipeline for efficient implementation. And RISC based computers use compilers to translate into RISC instructions. And CISC based computer use the hardware to translate those instruction into simpler micro-instructions that are RISC instructions.

(Refer Slide Time: 10:12)

Results of a Comparative Study

- A quantitative comparison of VAX 8700 (a CISC machine) and MIPS M2000 (a RISC machine) with comparable organizations was carried out in 1991.
- Some findings:
 - MIPS required execution of about twice the number of instructions as compared to VAX.
 - Cycles Per Instructions (CPI) for VAX was about six times larger than that of MIPS.
 - Hence, MIPS had three times the performance of VAX.
 - Also, much less hardware is required to build MIPS as compared to VAX.

A comparative study was performed in the year 1991 where a quantitative comparison between VAX 8700 which was a CISC machine and MIPS M2000 a RISC machine was made. And what findings came up are the following. MIPS required execution of about twice the number of instructions as compared to VAX that means the number of instructions that are required by RISC machine is much more compared to CISC. So, as CISC is having more complex instructions, so they require less number of instructions, but on the other hand RISC uses very simple instructions.

There is a parameter that we will be looking little later, cycles per instruction, we call it CPI. So, for each instruction what is the number of cycles that it requires; for VAX machine it was about six times larger than the MIPS machine. So, we can say that in VAX, it is taking only three instruction to execute, but each instruction is taking more number of cycles to execute, whereas MIPS had three times the performance of VAX. So, cycles per instruction were about six time larger in VAX. And, the performance of MIPS was three times the performance of VAX.

Also much less hardware is required to build MIPS as compared to VAX because VAX was having many complex instructions; for which you need to do complex decoding. So, the hardware requirement is much more. Whereas RISC uses simple instructions although they require more number of instructions. So, in that case this MIPS becomes three times faster than VAX.

(Refer Slide Time: 13:11)

Conclusion:

- Persisting with CISC architecture is too costly, both in terms of hardware cost and also performance.
- VAX was replaced by ALPHA (a RISC processor) by Digital Equipment Corporation (DEC).
- CISC architecture based on x86 is different.
 - Because of huge number of installed base, backward compatibility of machine code is very important from commercial point of view.
 - They have adopted a balanced view: (a) user's view is a CISC instruction set, (b) hardware translates every CISC instruction into an equivalent set of RISC instructions internally, (c) an instruction pipeline executes the RISC instructions efficiently.

So, the conclusion was that persisting with CISC architecture is too costly both in terms of hardware cost and also performance. And with this VAX was replaced by DEC Alpha which was a RISC machine. So, they moved from CISC to RISC after such study. So, this was a machine by DEC. So, CISC architecture based on x86 is different; because of huge number of installation base, backward compatibility of the machine code is very important from commercial point of view.

If backward compatibility is required, you need to have the instructions which you are having earlier. So, because of such thing this x86 which is a CISC machine it is still very important and they have adopted a balanced view. How is that balanced view they have adopted? A user view is a CISC instruction set and there is a hardware that translates every CISC instruction into an equivalent set of RISC instructions internally. So, ultimately internally they are having some kind of RISC instructions.

And also the instruction pipeline executes these RISC instructions efficiently. So, they still have this older CISC architecture at the user point of view level, but at the lower level they have this RISC where those instructions are converted into simpler instructions and those simpler instructions are executed, pipeline can also be performed there which makes it more efficient.

(Refer Slide Time: 15:41)

MIPS32 Architecture: A Case Study

- As a case study of RISC ISA, we shall be considering the MIPS32 architecture.
 - Look into the instruction set and instruction encoding in detail.
 - Design the data path of the MIPS32 architecture, and also look into the control unit design issues.
 - Extend the basic data path of MIPS32 to a pipeline architecture, and discuss some of the issues therein.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now, we will be going into MIPS32 architecture. This is a RISC architecture and we will be performing a case study of this MIPS. We will look into the instruction set instruction encoding in detail also the design of the data path of MIPS architecture and we will also look into the control unit design issues in the later units. And we extend the basic data path of MIPS to a pipeline architecture, and discuss some of the issues therein.

(Refer Slide Time: 16:32)

MIPS32 CPU Registers

- The MIPS32 ISA defines the following CPU registers that are visible to the machine/assembly language programmer.
 - 32, 32-bit general purpose registers (GPRs), *R0* to *R31*.
 - A special-purpose 32-bit program counter (*PC*).
 - Points to the next instruction in memory to be fetched and executed.
 - Not directly visible to the programmer.
 - Affected only indirectly by certain instructions (like branch, call, etc.)
 - A pair of 32-bit special-purpose registers *Hi* and *Lo*, which are used to hold the results of multiply, divide, and multiply-accumulate instructions.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now, MIPS32 there is a total of 32 general-purpose registers starting from *R0* to *R31*. Also, there is a special 32-bit program counter that is *PC*, which points to the next

instruction in memory to be fetched or executed. Now, this is not directly visible to the programmer and it is only affected by certain instructions. When it is affected, you think of a branch instruction, you think of a call. When you are going to a branch then PC must be loaded with the branch address; you have to calculate the branch address with an offset and then you have to go and execute that.

Now, a pair of 32-bit special purpose registers are also present we call it HI and LO which are used to hold the result of multiply, divide and multiply accumulate instruction. Now, see when you multiply two n-bit numbers the result can be $2n$ bits. So, you need to store it in a $2n$ -bit register. So, for that purpose we have two 32-bit registers, HI and LO, which are used for this multiply divide and multiply accumulate instructions.

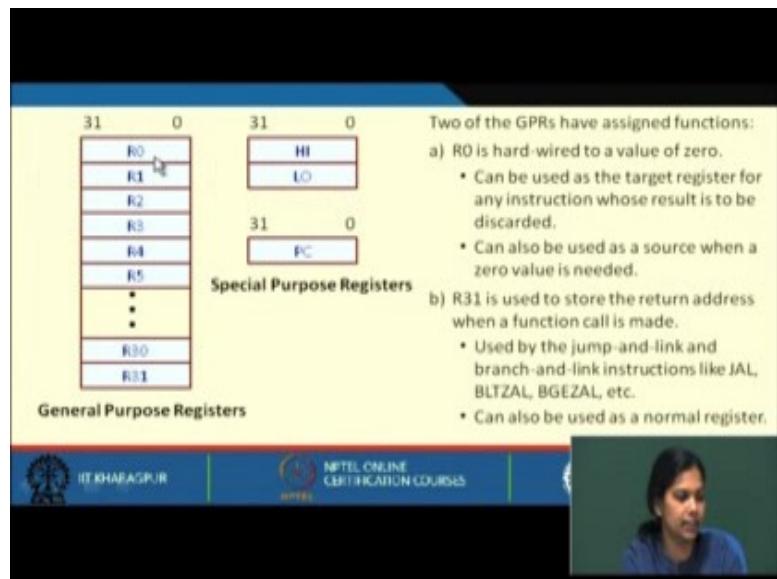
(Refer Slide Time: 18:17)

- Some common registers are missing in MIPS32.
 - *Stack Pointer (SP) register*, which helps in maintaining a stack in main memory.
 - Any of the GPRs can be used as the stack pointer.
 - No separate PUSH, POP, CALL and RET instructions.
 - *Index Register (IX)*, which helps in accessing memory words sequentially in memory.
 - Any of the GPRs can be used as an index register.
 - *Flag registers* (like ZERO, SIGN, CARRY, OVERFLOW) that keeps track of the results of arithmetic and logical operations.
 - Maintains flags in registers, to avoid problems in pipeline implementation.

Now, there are some common registers that are not present in MIPS32. What are those registers? We do not have any stack pointer. We already know a stack pointer helps in maintaining the stack in main memory, but we do not have any such kind of stack pointer in MIPS32. We do not have any index register as well that can be used for we have already seen index addressing mode, we require index address index register for some purposes, but it is not present. And what can be done is that as these are not present, you can use any register from the set of 32 registers and use it as an index register. Also there is no flag register; now you can argue that if there is no flag register then how we will be checking the operation? You have to perform the operation, store it in a register check

that value for zero or non-zero. So, we do not have any flag register like zero, sign, carry and overflow. Now, why we do not have this you have seen in pipeline there are different instructions that are coming in and they are going in a pipe in an overlapped fashion. Let us say if we have this flag registers it might happen that one instruction has updated some flag then the other instruction is not fully executed but has also updated that flag then there will be a problem. So, instead of doing so this flag can be maintained in some other fashions as done in this MIPS32 machine.

(Refer Slide Time: 20:20)



Now, these are the general-purpose registers, and these are the special-purpose registers I have already discussed. R0 register is hardwired to the value 0. So, R0 contains always zero value can be used as target register for any instruction whose result is to be discarded. Or if you add you want to add a zero value to some register you can use R0. R31 is used to store the return address when a function call is made; like as I already discussed that there is no stack pointer.

So, if there is no stack pointer when there will be a call, return, branch these kind of statement it may affect you have to store the address of the PC and then later when again we execute that branch we will come down to that particular address and we execute it. So, in such case we can use this R31 as return address and used by jump and link that is JAL, and other instruction BLTZAL and BGEZAL etc. And it can also be used as a normal register.

(Refer Slide Time: 21:58)

The slide is titled "Some Examples". It contains two boxes of assembly code and a video player on the right.

Box 1:

```
LD R4, 50(R3) // R4 = Mem[50+R3]
ADD R2, R1, R4 // R2 = R1 + R4
SD 54(R3), R2 // Mem[54+R3] = R2
```

Box 2:

```
MAIN: ADDI R1, R0, 35 // R1 = 35
      ADDI R2, R0, 56 // R2 = 56
      JAL GCD
      ...
GCD: .... // Find GCD of R1 & R2
      JR R31
```

Video Player: A video player window shows a woman speaking. The IIT Kharagpur logo and NPTEL Online Certification Courses logo are visible at the bottom of the slide.

Let us take some examples. So, LD R4,50(R3). So, this instruction will load from this particular address. It will add 50 + R3 then it will go to that particular memory location and load the value to R4. Here R1 + R4 is added and it is stored in R2. Similarly, storing means we need to store R2 into this particular location that is 54 + R3. And this is like what we are doing there is no move instruction. If you want to move a data from R5 to R3, all you can do is that you can add R5 with R0 and you can store it in R2. So, what happens R5 is moved to R2.

Now, these are some examples like we have a MAIN code where from where our execution starts and then we have another subroutine. So, what happens here, this is an ADDI, we are adding this immediate value to R0 which is 0. So, R1 will have 35 and R2 will have 56. And this jump instruction will go to a function that is labeled by GCD; and in GCD you will be seeing this some code for your GCD, and jump return (JR).

Now what you need to do now after fetching and decoding you have understood that you have to move there. So, this PC value must be stored in that return address it can be stored in R31, and then we move to this GCD we execute that. At the end of GCD you must have an instruction which is JR to return address where you have to go to R31, get the value loaded in PC, and then from this point the execution will start after returning back here.

(Refer Slide Time: 24:42)

How are the HI and LO registers used?

- During a multiply operation, the HI and LO registers store the product of an integer multiply.
 - HI denotes the high-order 32 bits, and LO denotes the low-order 32 bits.
- During a multiply-add or multiply-subtract operation, the HI and LO registers store the result of the integer multiply-add or multiply-subtract.
- During a division, the HI and LO registers store the quotient (in LO) and remainder (in HI) of integer divide.

Now, how are HI and LO registers used? As I said during multiply operation HI and LO registers store the product of an integer multiply. And during multiply-add or multiply-subtract operation the HI-LO registers store the result of an integer multiply-add or multiply-subtract, and during a division HI-LO registers store the quotient in LO and remainder in HI of an integer divide. So, HI-LO registers are used for these few purposes.

(Refer Slide Time: 25:28)

Some MIPS32 Assembly Language Conventions

- The integer registers of MIPS32 can be accessed as R0..R31 or r0..r31 in an assembly language program.
- Several assemblers and simulators are available in the public domain (like QtSPIM) that follow some specific conventions.
 - These conventions have become like a *de facto* standard when we write assembly language programs for MIPS32.
 - Basically some alternate names are used for the registers to indicate their intended usage.

Now, let us see little more MIPS32 assembly language conventions. So, the integer registers are numbered from R0 to R31. So, for doing so some simulators are available in public domain like QtSPIM. You can write assembly language code there and execute it. And these conventions have become a de facto standard when we write an assembly language program for MIPS. So, a QtSPIM is exactly a MIPS32 simulator that can be used to write assembly language programs. Basically some alternative names are used for register to indicate their intended use in this.

(Refer Slide Time: 26:31)

Register name	Register number	Usage
Sat	R1	Reserved for assembler
<p>May be used as temporary register during macro expansion by assembler.</p> <ul style="list-style-type: none"> Assembler provides an extension to the MIPS32 instruction set that are converted to standard MIPS32 instructions. 		
<p>Example: Load Address instruction used to initialize pointers</p> <pre> la R5.addr lui \$at.Upper-16-bits-of-addr ori R5.\$at.Lower-16-bits-of-addr </pre>		

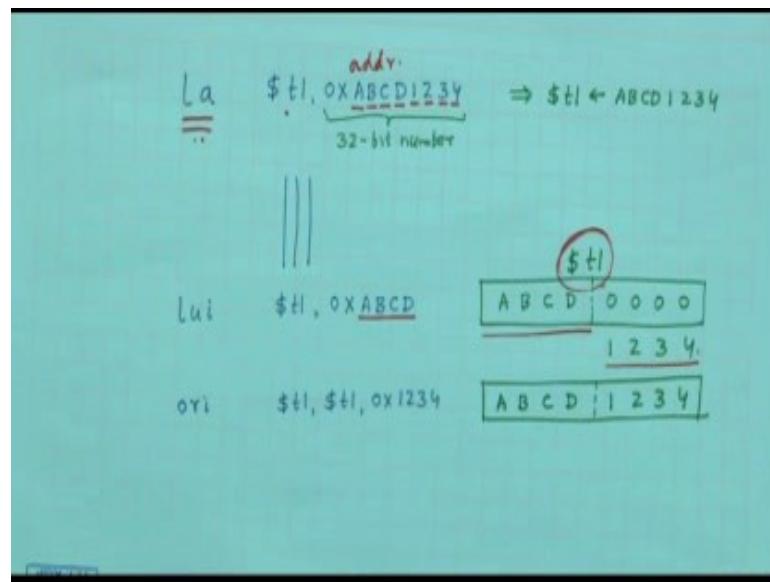




So, register zero - R0 which is a constant zero. So, register name \$zero is used to represent constant zero value whenever required in a program. As this particular register is reserved for an assembler, the name is \$at. And this may be used as temporary register during macro operation by an assembler. An assembler provides an extension to the MIPS instruction set that are converted to standard MIPS32 instructions.

Let us see an example load address instruction used to initialize the pointer that means, in R5 what we are doing we are loading an address, addr is a label from where we have stored some data. And I am loading that particular address into R5, but this instruction is not there in your MIPS. So, how MIPS will see this, MIPS will convert this into set of two instructions LUI (Load Upper Immediate). So, at this upper 16-bit of address is loaded and then we do an or immediate (ori) with the lower 16 bit address. Firstly, I have loaded the 16-bit number in the upper immediate.

(Refer Slide Time: 28:33)



Let us take this example that will make it clear. So, this la is loading this particular address into t1 and this is nothing but addr, either you write addr or actually this address. This is a 32-bit number. We do not have such kind of instruction la in MIPS. So, this instruction will break down into couple of instructions when you actually execute. So, this is the upper four nibbles A, B, C and D. So, in \$t1 the upper 4 nibbles is added.

So, now you see in t1 this 16-bit is loaded with A, B, C, D. Now, what will we do we do an ori that will add this and this with t1; in t1 you already have this particular value. You are doing an ori with 0x1234 that is the next four nibbles you are or-ing with 1234, and storing back the result in t1 itself. So, in t1 you will have ABCD you have or with 1234, 1234. So, this instruction is doing exactly what I explained.

(Refer Slide Time: 30:31)

Register name	Register number	Usage
\$v0	R2	Result of function, or for expression evaluation
\$v1	R3	Result of function, or for expression evaluation

May be used for up to two function return values, and also as temporary registers during expression evaluation.



IIT KHARAGPUR



NPTEL ONLINE CERTIFICATION COURSES



NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Similarly, we have another register that is \$v0 and \$v1; the register that are used is R2 and R3. So, result of a function or an expression evaluation is stored in \$v0 and in \$v1. Let us see, may be it can be used for up to two functions, return values and also as temporary register during expression evaluation.

(Refer Slide Time: 31:21)

Register name	Register number	Usage
\$a0	R4	Argument 1
\$a1	R5	Argument 2
\$a2	R6	Argument 3
\$a3	R7	Argument 3

May be used to pass up to four arguments to functions.



IIT KHARAGPUR



NPTEL ONLINE CERTIFICATION COURSES



NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Next set of register is \$a0, \$a1, \$a2, \$a3 we use as arguments. So, these are used to pass up to four arguments to a function.

(Refer Slide Time: 31:43)

Register name	Register number	Usage
\$t0	R8	Temporary (not preserved across call)
\$t1	R9	Temporary (not preserved across call)
\$t2	R10	Temporary (not preserved across call)
\$t3	R11	Temporary (not preserved across call)
\$t4	R12	Temporary (not preserved across call)
\$t5	R13	Temporary (not preserved across call)
\$t6	R14	Temporary (not preserved across call)
\$t7	R15	Temporary (not preserved across call)
\$t8	R24	Temporary (not preserved across call)
\$t9	R25	Temporary (not preserved across call)

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

These are temporary register \$t0 to \$t9. And these are not preserved across calls. So, it might happen that when you are using this register for some purpose, it can be also used for other. So, it is not preserved across the call.

(Refer Slide Time: 32:09)

Register name	Register number	Usage
\$t0	R8	Temporary (not preserved across call)
\$t1	R9	Temporary (not preserved across call)
\$t2	R10	Temporary (not preserved across call)
\$t3	R11	Temporary (not preserved across call)
\$t4	R12	May be used as temporary variables in programs. These registers might get modified when some functions are called (other than user-written functions).
\$t5	R13	
\$t6	R14	
\$t7	R15	
\$t8	R24	Temporary (not preserved across call)
\$t9	R25	Temporary (not preserved across call)

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

May be used as temporary variables in programs, but these registers might get modified. So, it is up to you that if you are using this then you have to be careful enough, and these register might get modified when some functions are called other than user written

functions that means, inside also we are calling some function, some instruction are breaking down into some further instructions.

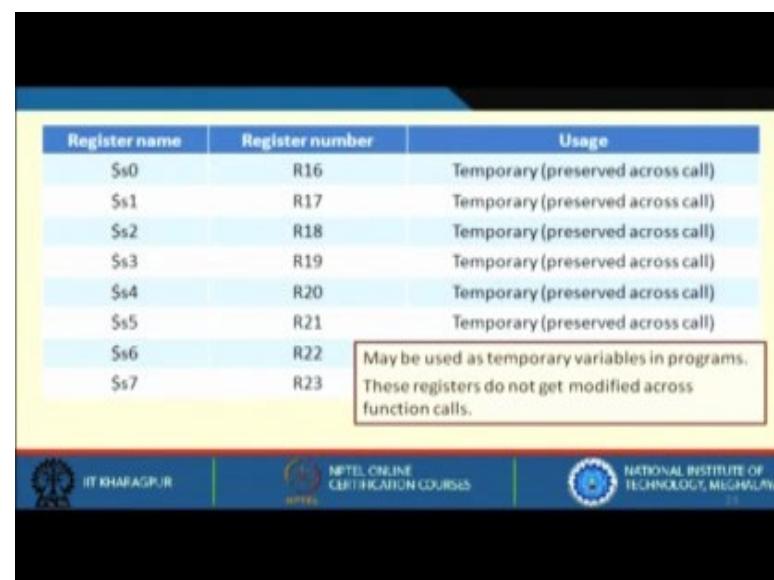
(Refer Slide Time: 32:50)



Register name	Register number	Usage
\$s0	R16	Temporary (preserved across call)
\$s1	R17	Temporary (preserved across call)
\$s2	R18	Temporary (preserved across call)
\$s3	R19	Temporary (preserved across call)
\$s4	R20	Temporary (preserved across call)
\$s5	R21	Temporary (preserved across call)
\$s6	R22	Temporary (preserved across call)
\$s7	R23	Temporary (preserved across call)

So, in this particular case you can see that it can be user other than user return function it can be overwritten, but these are preserved across call. So, even if there are some other function calls you can use this set of temporary register starting from \$s0 to \$s7.

(Refer Slide Time: 33:04)



Register name	Register number	Usage
\$s0	R16	Temporary (preserved across call)
\$s1	R17	Temporary (preserved across call)
\$s2	R18	Temporary (preserved across call)
\$s3	R19	Temporary (preserved across call)
\$s4	R20	Temporary (preserved across call)
\$s5	R21	Temporary (preserved across call)
\$s6	R22	May be used as temporary variables in programs. These registers do not get modified across function calls.
\$s7	R23	

May be used as temporary variables in the program. So, I will suggest you if you are coding this using you are writing some assembly language program then you can use this set of these set of registers for your programming purpose.

(Refer Slide Time: 33:26)

Register name	Register number	Usage
\$gp	R28	Pointer to global area
\$sp	R29	Stack pointer
\$fp	R30	Frame pointer
\$ra	R31	Return address (used by function call)

These registers are used for a variety of pointers:

- Global area: points to the memory address from where the global variables are allocated space.
- Stack pointer: points to the top of the stack in memory.
- Frame pointer: points to the activation record in stack.
- Return address: used while returning from a function.

We have another set which is \$gp, \$sp, \$fp, and \$ra. So, \$gp is pointed to global area, \$sp is your stack pointer, \$fp is your frame pointer, and \$ra is your return address. So, we are having all these, but we are not having with a special function register rather the general purpose registers are only used for this, but we know that R29 is a general purpose register which is used for stack pointer. So, these registers are used for variety of pointers. So, global pointers as I said point to the memory location from where the global variables are allocated space. Stack pointer points to the top of the stack. A frame pointer points to the activation record in stack. And return address is used while returning from a function.

(Refer Slide Time: 34:34)

The screenshot shows a presentation slide with a table and a note. The table has three columns: Register name, Register number, and Usage. It lists two entries: \$k0 (R26) and \$k1 (R27), both of which are reserved for the OS kernel. Below the table is a note in a box: "These registers are supposed to be used by the OS kernel in a real computer system. It is highly recommended not to use these registers." The slide also features the IIT Kharagpur logo, the NPTEL logo, and a video feed of a speaker.

Register name	Register number	Usage
\$k0	R26	Reserved for OS kernel
\$k1	R27	Reserved for OS kernel

These registers are supposed to be used by the OS kernel in a real computer system.
It is highly recommended not to use these registers.

We have register \$k0 and \$k1 reserved for OS kernel. So, it is highly recommended not to use these registers.

So, we come to the end of lecture 8. So, in this lecture, now we have given you an idea what we will be discussing next. We have discussed about some properties of instruction set architecture and now we are moving on with a particular architecture that is MIPS32, and I have discussed some of its features and we will be discussing many more in course of time.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 09
MIPS32 Instruction Set

Welcome to the 9th lecture on MIPS32 instruction set. In this lecture, we will be seeing the instruction set of MIPS32 --- which means the various kinds of instructions that are possible in MIPS32.

(Refer Slide Time: 00:39)

Instruction Set Classification

- MIPS32 instruction can be classified into the following functional groups:
 - Load and Store
 - Arithmetic and Logical
 - Jump and Branch
 - Miscellaneous
 - Coprocessor instruction (*to activate an auxiliary processor*).
- All instructions are encoded in 32 bits.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Broadly instruction in MIPS can be classified into load store instructions, arithmetic and logic instructions, jump and branch, we have some miscellaneous instructions, and some coprocessor instructions. And all instructions can be encoded in 32 bits.

(Refer Slide Time: 01:11)

Alignment of Words in Memory

- MIPS requires that all words must be aligned in memory to word boundaries.
 - Must start from an address that is some multiple of 4.
 - Last two bits of the address must be 00.
- Allows a word to be fetched in a single cycle.
 - Misaligned words may require two cycles.

Address	w1	w1	w1	w1
0000H				
0004H		w2	w2	w2
0008H	w2			
000CH			w3	w3
0010H	w3	w3		
0014H				w4
0018H	w4	w4	w4	

w1 is aligned, but w2, w3, w4 are not

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Now, let us see what is this word alignment. It is alignment of the words in memory. MIPS always requires that all the words that we store in memory must be aligned to word boundaries; that means, must start from an address that is some multiple of four. So, you see the first address which is 0000, it starts from here then 0004 then 0008, and so on. So, must start from an address that is some multiple of 4.

So, the last two bits of the address must be 00, 0 – 0000, 4 – 0100, 8 – 1000, as well as all others. This allows a word to be fetched in a single cycle. So, in a single cycle, we access this word and we get this entire word in one cycle. This is why we say that MIPS requires that the words to be aligned in the memory.

Now, here the first word is aligned, but see the next word it is not aligned because it is not starting with this address it is starting with the fifth one. Similar way this word - word three and word four both are not aligned only first word w1 is aligned.

(Refer Slide Time: 03:24)

(a) Load and Store Instructions

- MIPS32 is a load-store architecture.
 - All operations are performed on operands held in processor registers.
 - Main memory is accessed only through *LOAD* and *STORE* instructions.
- There are various types of LOAD and STORE instructions, each used for a particular purpose.
 - a) By specifying the size of the operand (W: word, H: half-word, B: byte)
 - Examples: LW, LH, LB, SW, SH, SB
 - b) By specifying whether the operand is signed (by default) or unsigned.
 - Examples: LHU, LBU

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

We have been discussing about load-store architecture; what it means is that all operations are performed on operands held in the processor registers. Only two instructions can load the data from memory or it can store the data into the memory; no other instruction can use a memory location. There are various types of load-store instructions that can be used for a particular purpose; like we can load a word, we can load a byte, or we can load a half word. In the same way, we can also store a word, we can store a half word, or we can store a byte. By specifying whether the operand is signed or unsigned, we can also load a half word unsigned, load a byte unsigned.

(Refer Slide Time: 04:34)

c) Accessing fields that are not word aligned.

- Examples: LWL, LWR, SWL, SWR

d) Atomic memory update for read-modify-write instructions

- Required to implement semaphores, for example.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Here we have another set of instructions, which are used for accessing fields that are not word aligned. The instruction that are used are load word left, and load word right, store word left, and store word right. And there are some other instructions as well like atomic memory updates for read-modify-write instruction. So, just think of an instruction that requires to be completed fully like when we read it, we modify it and we write back at the same go. So, we cannot have it that we read it, we update it, we do not update it. If we read it, we have to update it and then we can store that, so those are atomic operations.

(Refer Slide Time: 05:32)

Data Size	Load Signed	Load Unsigned	Store
Byte	YES	YES	YES
Half-word	YES	YES	YES
Word	YES	Only for MIPS64	YES
Unaligned word	YES		YES
Linked word (atomic modify)	YES		YES

The slide also features logos for IIT Kharagpur, NPTEL, and a female speaker in the bottom right corner.

These are the data sizes that can be accessed through load and store, but in load unsigned we can only use byte and half word. And for word this can be done only for MIPS64, and for store it can be done for all.

(Refer Slide Time: 05:54)

Type	Mnemonic	Function	Type	Mnemonic	Function
Aligned	LB	Load Byte	Unaligned	LWL	Load Word Left
	LBU	Load Byte Unsigned		LWR	Load Word Right
	LH	Load Half-word		SWL	Store Word Left
	LHU	Load Half-word Unsigned		SWR	Store Word Right
	LW	Load Word		LL	Load Linked Word
	SB	Store Byte		SB	Store Conditional Word
	SH	Store Half-word			
	SW	Store Word			

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES | 



Now, we can see some more instructions; LB is load byte, this is load byte unsigned load half word and so on. And here for an unaligned one, we have load word left, load word right and similar way store word left and store word like right. And also for atomic update, we have load linked word and store conditional word.

(Refer Slide Time: 06:28)

(b) Arithmetic and Logic Instructions	
• All arithmetic and logic instructions operate on registers.	
• Can be broadly classified into the following categories:	
– ALU immediate	
– ALU 3-operand	
– ALU 2-operand	
– Shift	
– Multiply and Divide	

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES | 



Let us move on with arithmetic and logic instructions. MIPS32 has a wide variety of arithmetic and logic instructions that can broadly classified into the following categories.

The categories include ALU immediate, ALU with 3-operand is possible, ALU with 2-operand, shift, multiply and divide.

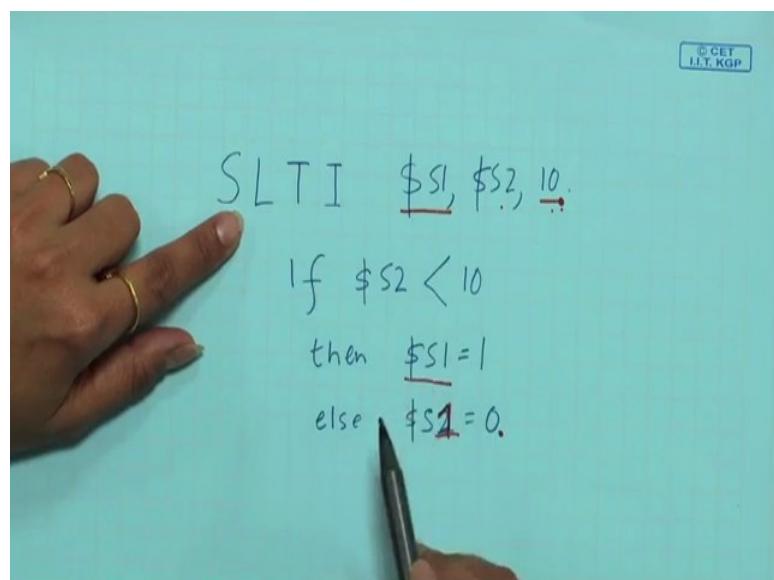
(Refer Slide Time: 06:59)

Type	Mnemonic	Function
16-bit Immediate Operand	ADDI	Add Immediate Word
	ADDIU	Add Immediate Unsigned Word
	ANDI	AND Immediate
	LUI	Load Upper Immediate
	ORI	OR Immediate
	SLTI	Set on Less Than Immediate
	SLTIU	Set on Less Than Immediate Unsigned
	XORI	Exclusive-OR Immediate

 IIT KHARAGPUR |
  NPTEL ONLINE CERTIFICATION COURSES |
  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, let us see the this set of arithmetic operation this is ADDI - add immediate word. This is ADDIU - add immediate unsigned word. This is LUI - load upper immediate. This is ORI - or immediate. This is SLTI - set on less than immediate.

(Refer Slide Time: 07:33)



Let us take an example of SLTI - set on less than immediate, the meaning of which is if \$s2 is less than this immediate value 10, then you set \$s1 to 1, else you set \$s1 to 0.

(Refer Slide Time: 08:15)

Type	Mnemonic	Function
3-Operand	ADD	Add Word
	ADDU	Add Unsigned Word
	AND	Logical AND
	NOR	Logical NOR
	SLT	Set on Less Than
	SLTU	Set on Less Than Unsigned
	SUB	Subtract Word
	SUBU	Subtract Unsigned Word
	XOR	Logical XOR

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

We have three operand instructions, where these are add, add unsigned, and, nor, set less than, set less than unsigned, sub, sub unsigned, xor.

(Refer Slide Time: 08:34)

Type	Mnemonic	Function
2-Operand	CLO	Count Leading Ones in Word
	CLZ	Count Leading Zeros in Word
Type	Mnemonic	Function
Shift	ROTR	Rotate Word Right
	ROTRV	Rotate Word Right Variable
	SLL	Shift Word Left Logical
	SLLV	Shift Word Left Logical Variable
	SRA	Shift Word Right Arithmetic
	SRAV	Shift Word Right Arithmetic Variable
	SRL	Shift Word Right Logical
	SRLV	Shift Word Right Logical Variable

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

For two-operand, we have these instructions CLO that is count leading ones in a word, or CLZ that is count leading zeros in a word. So, these instructions are also sometimes required for various programming. We have another set of instructions for rotating a word. So, we can rotate a word right, we can rotate a word with a variable, and we can specify that how many bits we need to rotate. We can shift a word left logical, that is, we

can do a logical shift; we can shift a word left logical, and we can specify the variable. And similarly we can do it for shifting a word right arithmetic - shifting a word right with arithmetic right shift; and with arithmetic right shift with some variable, and we can also do logical right shift.

(Refer Slide Time: 09:50)

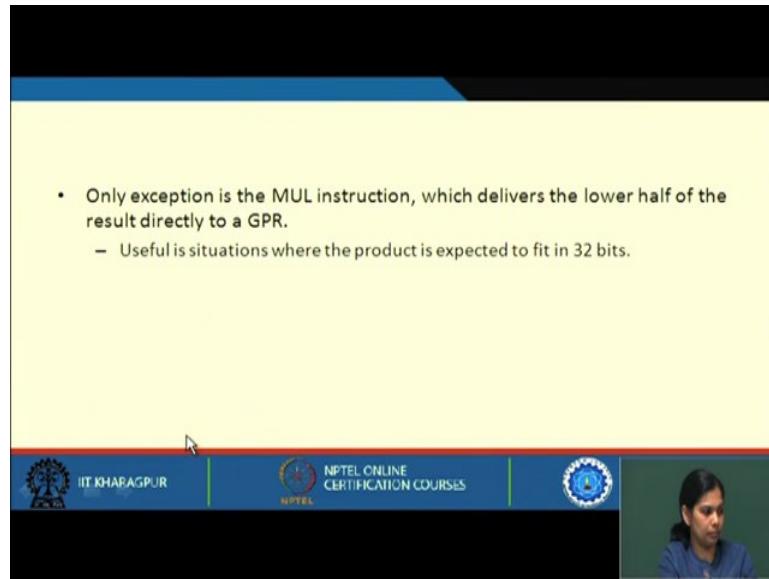
(c) Multiply and Divide Instructions

- The multiply and divide instructions produce twice as many result bits.
 - When two 32-bit numbers are multiplied, we get a 64-bit product.
 - After division, we get a 32-bit quotient and a 32-bit remainder.
- Results are produced in the HI and LO register pair.
 - a) For multiplication, the low half of the product is loaded into LO, while the higher half in HI.
 - b) Multiply-Add and Multiply-Subtract produce a 64-bit product, and adds or subtracts the product from the concatenated value of HI and LO.
 - c) Divide produces a quotient that is loaded into LO and a remainder that is loaded into HI.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us move on and see multiply and divide instructions. So, the next set of instructions that MIPS32 instruction set is having is multiply and divide. So, when two 32-bit numbers are multiplied we can get a 64-bit product; and after division also you may have to store a 32-bit quotient and a 32-bit remainder. So, where we will store this 64-bit result? We have a register called HI-LO; this is a register pair. So, for multiplication the low half of the product is loaded in LO while high half is loaded in HI. Multiply and multiply subtract produces a 64-bit product and adds or subtract the product from the concatenated value of HI and LO. And also divide produces a quotient that is loaded into LO and a remainder that is loaded into HI.

(Refer Slide Time: 11:03)



There is only one exception that for the multiplication instruction which delivers the lower half of the result directly to GPR, because it is useful for the situation when the product is expected because when we multiply two numbers always the result may not be 64-bit, the result can fit into 32-bit as well. So, in that case is an exception with MUL instruction.

(Refer Slide Time: 11:34)

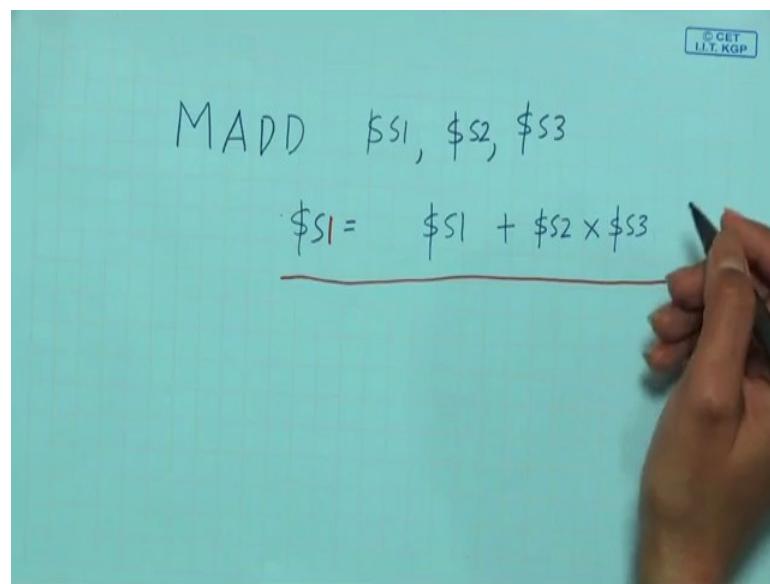
Type	Mnemonic	Function
Multiply and Divide	DIV	Divide Word
	DIVU	Divide Unsigned Word
	MADD	Multiply and Add Word
	MADDU	Multiply and Add Word Unsigned
	MFHI	Move from HI
	MFLO	Move from LO
	MSUB	Multiply and Subtract Word
	MSUBU	Multiply and Subtract Word Unsigned
	MTHI	Move to HI
	MTLO	Move to LO
	MUL	Multiply Word to Register
	MULT	Multiply Word
	MULTU	Multiply Unsigned Word

IIT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

On the right side of the slide, there is a video feed of a woman speaking.

So, these are the various instructions that I have talking about: div, divide with unsigned word multiply and add word. So, what is this multiply and add word.

(Refer Slide Time: 11:49)



Let us see this instruction. MADD means we are multiplying with three operand here. So, we are multiplying \$s2 and \$s3, and we are adding with \$s1, and also we are storing the result in \$s1. Such kind of instructions are required in digital signal processing, and is supported in MIPS32 architecture. So, we have various move from high, move from low, you have various multiply word to a register, this is generally used multiply a word, multiply unsigned word.

(Refer Slide Time: 12:43)

(d) Jump and Branch Instructions

- The following types of Jump and Branch instructions are supported by MIPS32.
 - PC relative conditional branch
 - A 16-bit offset is added to PC.
 - PC-region unconditional jump
 - A 28-bit offset is added to PC.
 - Absolute (register) unconditional jump
 - Special Jump instructions that link the return address in R31.

The slide has a dark header and footer. The footer contains logos for IIT Kharagpur, NPTEL, and a woman speaking. The text "NPTEL ONLINE CERTIFICATION COURSES" is also present.

Next set of instructions is jump and branch. The following types of jump and branch instructions are supported in MIPS32. We know that whenever we wanted to perform some kind of branching like in a program with loops, for those we require jump and branch instructions. We have PC relative conditional branch where a 16-bit offset is added to PC, or in a conditional unconditional branch a 28-bit offset is added to PC. There can be absolute unconditional branch whether absolute address can be provided in the register, and special jump functions that link the return address in R31. So, we are jumping from one location to another and after executing that particular location we have to come back to the previous one. So, the value of the PC must be loaded with that return address value.

(Refer Slide Time: 13:53)

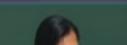
Type	Mnemonic	Function
Unconditional Jump within a 256 MB Region	J	Jump
	JAL	Jump and Link
	JALX	Jump and Link Exchange

Type	Mnemonic	Function
Unconditional Jump using Absolute Address	JALR	Jump and Link Register
	JALRHB	Jump and Link Register with Hazard Barrier
	JR	Jump Register
	JRHB	Jump Register with Hazard Barrier

In this context we have some instruction, this is unconditional branch jump and link. Jump and link exchange, and these are some unconditional jump using absolute address. So, jump and link register, jump and link register with hazard barrier. So, you will be seeing some of these instructions when we will be studying pipelining in course of time.

(Refer Slide Time: 14:24)

Type	Mnemonic	Function
PC-Relative Conditional Branch Comparing Two Registers	BEQ BNE	Branch on Equal Branch on Not Equal
Type	Mnemonic	Function
PC-Relative Conditional Branch Comparing With Zero	BGEZ BGEZAL BGTZ BLEZ	Branch on Greater Than or Equal to Zero Branch on Greater Than or Equal to Zero and Link Branch on Greater than Zero Branch on Less Than or Equal to Zero



So, related to PC there are some instructions: BEQ and BNE. We shall come across these kind of branch instruction very frequently when we do some programming. And there are some PC relative conditional branch comparing with zero. So, branch on greater than or equal to zero, or branch on greater than or equal to zero and link. We have few branch on greater than zero, branch on less than equal to zero. So, we have various instructions that we can we may use for our programming constructs.

(Refer Slide Time: 15:11)

(e) Miscellaneous Instructions
<ul style="list-style-type: none">These instructions are used for various specific machine control purposes.They include:<ul style="list-style-type: none">Exception instructionsConditional MOVE instructionsPrefetch instructionsNOP instructions



We also have some miscellaneous instructions used for various specific machine control purposes, and they include some exceptional instructions, conditional move instruction, some prefetch instructions are also there, no operation instructions. So, we will again see that some of these instructions like NOP may be used in pipeline for some purposes.

(Refer Slide Time: 15:40)

Type	Mnemonic	Function
System Call and Breakpoint	BREAK	Breakpoint
	SYSCALL	System Call

Type	Mnemonic	Function
Trap-on-Condition Comparing Two Registers	TEQ	Trap if Equal
	TGE	Trap if Greater Than or Equal
	TGEU	Trap if Greater Than or Equal Unsigned
	TLT	Trap if Less Than
	TLTU	Trap if Less Than Unsigned
	TNE	Trap if Not Equal

 IIT KHARAGPUR |
  NPTEL ONLINE CERTIFICATION COURSES |
 



Next is system call. SYSCALL is something which we will be using often in programming using QtSPIM where we use system call which is syscall. And there can be some instructions like trap; trap if equal, trap if greater than or equal, trap if greater than or equal unsigned, and so on. So, these are some OS related instructions that are used if you want to request OS for some requirement, and then the OS takes care of it.

(Refer Slide Time: 16:39)

Type	Mnemonic	Function		
System Call and Breakpoint	BREAK SYSCALL	Type	Mnemonic	Function
Type	Mnemonic	Trap-on-Condition		
Comparing Two Registers	TEQ TGE TGEU TLT TLTU TNE	Comparing an Immediate Value	TEQI TGEI TGEIU TLTI TLTIU TNEI	Trap if Equal Immediate Trap if Greater Than or Equal Immediate Trap if Greater Than or Equal Immediate Unsigned Trap if Less Than Immediate Trap if Less Than Immediate Unsigned Trap if Not Equal Immediate
		Trap if Less Than		
		Trap if Less Than Unsigned		
		Trap if Not Equal		

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL



So, these are also various kind of trap-on-condition comparing with an immediate value; these are some of those instructions.

(Refer Slide Time: 16:51)

Type	Mnemonic	Function
Prefetch	PREF	Prefetch Register+Offset
NOP	NOP	No Operation

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL



And this is prefetch and no operation.

(Refer Slide Time: 16:57)

(e) Coprocessor Instructions

- The MIPS architecture defines four coprocessors (designated CP0, CP1, CP2, and CP3).
 - Coprocessor 0 (CP0) is incorporated on the CPU chip and supports the virtual memory system and exception handling. CP0 is also referred to as the System Control Coprocessor.
 - Coprocessor 1 (CP1) is reserved for the floating point coprocessor.
 - Coprocessor 2 (CP2) is available for specific implementations.
 - Coprocessor 3 (CP3) is available for future extensions.
- These instructions are not discussed here.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Next set of instruction is coprocessor instructions. The MIPS architecture defines four coprocessors, designated as coprocessor 0, 1, 2, and 3. CP0 is incorporated within the CPU chip and this also supports the virtual memory system and exceptional handling. CP0 is also referred to as system control coprocessor.

Now, these four coprocessors may not be used for all cases, but like CP1 is reserved for floating point coprocessor. CP2 is available for specific implementations and CP3 can be used for future extension. So, we really do not know that which kind of coprocessor we will be using in near feature. So, for that reason we need some coprocessor that can be available in future, and these instruction are not discussed here. So, we are just saying that we will have some kind of instructions like coprocessor instructions.

(Refer Slide Time: 18:15)

- MIPS32 architecture also supports a set of floating-point registers and floating-point instructions.
 - Shall be discussed later.

MIPS architecture also supports a set of floating point registers and floating point instructions that shall be discussed later.

So, we came through end of lecture 9, where we discussed the various kind of instructions that are there in MIPS32 architecture.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture – 10
MIPS Programming Examples

Welcome to the 10th lecture. In this lecture we will be discussing MIPS programming examples.

(Refer Slide Time: 00:33)

The slide has a title 'Some Examples of MIPS32 Arithmetic'. It shows two examples of C code being converted to MIPS32 assembly code.

Example 1:

C Code: $A = B + C;$

MIPS32 Code: `add $s1, $s2, $s3`

Notes: B loaded in \$s2; C loaded in \$s3; A \leftarrow \$s1

Example 2:

C Code: $A = B + C - D;$
 $E = F + A;$

MIPS32 Code: `add $t0, $s1, $s2`
`sub $t0, $t0, $s3`
`add $s4, $s5, $t0`

Notes: B loaded in \$s1; C loaded in \$s2; D loaded in \$s3; F loaded in \$s5; \$t0 is a temporary; A \leftarrow \$s0; E \leftarrow \$s4

The slide also features logos for IIT Kharagpur, NPTEL, and a female speaker.

So, we will start with how C code, that is a code written in high-level language, can be converted into MIPS32 code. So, this code is $A = B + C$. This code can be converted into `add $s1,$s2,$s3`. So, B and C should be loaded in \$s2 and \$s3 and the result will be stored in \$s1, that is A.

This is a set of two set of instructions. In the first case it is performing $A = B + C - D$ and in the next we are adding F with the computed value of A. How we can do that? So firstly, add \$s1, \$s2 and store it in \$t0, then subtract \$t0 and \$s3. So, in \$s3 D should be loaded and finally, we add F with A; F is loaded in \$s5 and E is in \$s0. So, B is loaded in \$s1; C is loaded in \$s2; D is loaded in \$s3; and F is loaded in \$s5. And finally, once we compute the result it is stored in \$s4, which is E. So, from \$s4 we have to move it to E, and partially this result should be stored in A. So, we stored the result \$s0 in A.

(Refer Slide Time: 03:07)

The slide is titled "Example on LOAD and STORE". It shows the conversion of C code to MIPS32 assembly code.

C Code: A[10] = X - A[12];

MIPS32 Code:

lw \$t0, 48(\$s3)
sub \$t0, \$s2, \$t0
sw \$t0, 40(\$s3);

Annotations explain the assembly code:

- \$s3 contains the starting address of the array A
- \$s2 loaded with X
- \$t0 is a temporary
- Address of A[10] will be \$s3+40 (4 bytes per element)
- Address of A[12] will be \$s3+48

At the bottom, there are logos for IIT Kharagpur, NPTEL Online Certification Courses, and a female speaker.

Let us see some example of load and store. So, this is an array element A[10]. So, A is a location and is an array in a consecutive memory address where we have stored various data. And in the 10th location what we are trying to load X minus A[12]. Let us see how we can convert this code. Firstly, each instruction is 32-bit. So, the 12th word will be $4 * 12 = 48$. So, we need to load the initial address of A in \$s0 and then we will be adding this initial address of A. that is in \$s3 plus 48. And that particular address the word which will be having will be A[12]; that word will be loaded in \$t0, then X must be loaded in another register, let us say it is loaded in \$s2, and we do \$s2 - \$t0 and we store it in \$t0. And finally, this particular result is stored back in location A[10].

So, the address of A plus $4 * 10 = 40$. So, \$s3 contains the starting address of an array of this array A. So, this will be starting address plus $4 * 10$ and this is starting address plus $4 * 12$. So, \$s2 loaded with X and once we load this in \$t0, then this particular value because in \$s3 we have loaded the address added with 48 which will be loaded in \$t0. And finally, we subtract it and store it in \$t0, and finally this \$t0 is stored in this particular location. So, address of A[10] will be \$s3 + 44 * 10. And then address of A[12] as I said will be \$s3 + 48, that is what I have used it. So, to represent a C code we have to use the following set of assembly language code.

(Refer Slide Time: 06:02)

C Code

```
if (x==y) z = x - y;
```

MIPS32 Code

```
bne    $s0, $s1, Label
sub    $s3, $s0, $s1
Label: .....
```

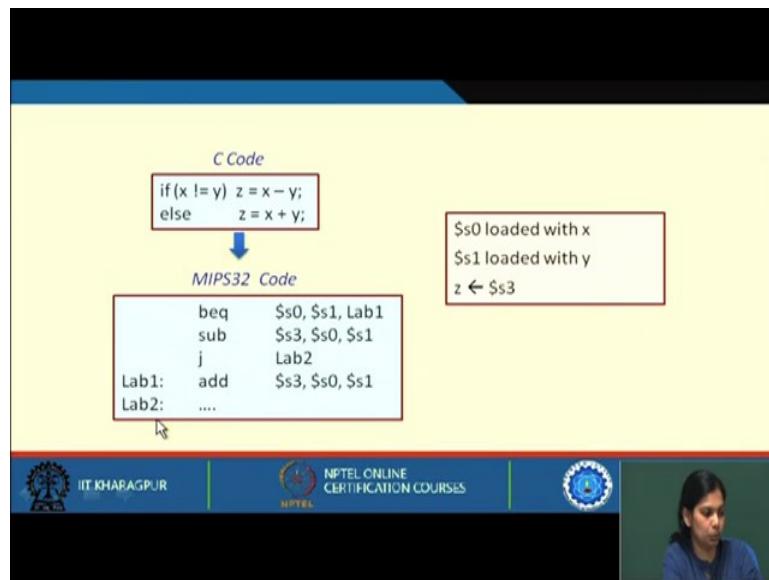
\downarrow

$\$s0$ loaded with x
 $\$s1$ loaded with y
 $z \leftarrow \$s3$

Let us see some example of control constructs. So, in C we often do this if x equals to y , then do something, or if x not equal to y do something. So, let us see how we can convert this particular C code. So, if equal to then we have to do this if it is not equal then you do something else. So, we take a branch instruction which is branch if not equal; and what this instruction is doing branch if not equal that means, $\$s0$ is not equal to $\$s1$ then go to this Label if it is not equal; if it is equal then this condition will not be met. So, the next instruction will get executed that is what we want it; if x is equal to y , z will be equal to $x - y$. So, branch if not equal. We are checking for not equal in $\$s0$ and $\$s1$. So, in $\$s0$ and $\$s1$, x and y are loaded; if it is not equal then go to this Label; and if it is equal then the next statement will execute. So, it is a sequential execution.

If this instruction it is a branch instruction and the condition here fails the condition is not matching, $\$s0$ is not equal to $\$s1$. If that is so then it will not go to this particular Label; rather it will go to the next statement that is $sub \$s3, \$s0, \$s1$. So, it will subtract $\$s0$ and $\$s1$.

(Refer Slide Time: 08:27)



Let us see some more C code. If x not equal to y then $z = x - y$; else $z = x + y$. How the MIPS code can be written branch if equal, so we are not checking for this we are checking for equal. So, if it is not equal then we have to execute this; and if it is equal then they should be executed. So, I am first checking if it is equal. So, I am doing branch if equal $\$s0$ and $\$s1$. In a similar fashion x will be loaded in $\$s0$, and y will be loaded in $\$s1$. So, we load x and y in $\$s0$ and $\$s1$ and check whether it is equal or not. If these two are equal, then we go to Lab1; that means, this else part now I am executing. If both these are equal, we go to Lab1 and add x plus y that is in $\$s0$ and $\$s1$, and store it in $\$s3$.

And let us say if it is not equal then we have to execute $x - y$. So, I am executing that if it is equal then I am going to label if it is not equal I will execute the next statement. And in the next statement I am doing sub where and subtracting $\$s0 - \$s1$ and storing it in $\$s3$ and then I am directly going to jump to Lab2; that means, I will not execute this rather I have to skip this because this will be executed based on this level. So, after this condition is satisfied that is x not equal to y then I have executed this statement; and after this I will go to later part of the code and will not execute this. So, I am jumping it to Lab2.

(Refer Slide Time: 10:45)

The slide contains the following text and code snippets:

- MIPS32 supports a limited set of conditional branch instructions:

```
beq $s2,Label // Branch to Label if $s2 = 0
bne $s2,Label // Branch to Label if $s2 != 0
```
- Suppose we need to implement a conditional branch after comparing two registers for less-than or greater than.

C Code

```
if(x < y) z = x - y;
else z = x + y;
```

MIPS32 Code

```
slt $t0,$s0,$s1
beq $t0,$zero,Lab1
sub $s3,$s0,$s1
j Lab2
Lab1: add $s3,$s0,$s1
Lab2: ...
```

Set if less than.
If \$s0 < \$s1, then
set \$t0=1; else
\$t0=0.

IIT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

MIPS support a limited set of conditional branch instructions: branch if equal, branch if not equal. Suppose we need to implement a conditional branch after comparing two registers for less than or greater than then what we will do. If let us say \$s1 is less than \$s2, I will do this; if \$s1 is greater than \$s2, I will do this; so like this if x is less than y it should do this else it should do other thing.

So, there are some instruction set if less than; that means, we will set \$t0 to 1 if it is less than that means, if \$s0 is less than \$s1 we will set \$t0 to 1. And then we will do branch if equal if \$t0 now equal to 0 then you add that if the else part we are going if it is not equal to zero; that means, x is less than y. So, we are doing sub \$s3,\$s0,\$s1. And similar way we have done in the previous thing, we are jumping to Lab2 because we do not want to execute this particular statement.

(Refer Slide Time: 12:38)

- MIPS32 assemblers supports several pseudo-instructions that are meant for user convenience.
 - Internally the assembler converts them to valid MIPS32 instructions.
- Example: The pseudo-instruction branch if less than
blt \$s1, \$s2, Label

MIPS32 Code

slt \$at, \$s1, \$s2	bne \$t0, \$zero, Label
.....
Label:

The assembler requires an extra register to do this.
The register \$at (= R1) is reserved for this purpose.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

MIPS32 assemblers supports several pseudo-instructions this I have already discussed previously that are meant for user convenience. But even if they are pseudo-instructions internally they have to be converted into some valid MIPS32 instructions. So, some pseudo-instruction we already have for branch less than (blt). This instruction will get converted into slt and bne instructions. The assembler requires an extra register to do this; this register is \$at, and the register \$at is R1 is reserved for this particular purpose.

(Refer Slide Time: 13:47)

Working with Immediate Values in Registers

- Case 1: Small constants, which can be specified in 16 bits.
 - Occurs most frequently (about 90% of the time).
 - Examples:
 - A = A + 16; → addi \$s1, \$s1, 16 (A in \$s1)
 - X = Y - 1025; → subi \$s1, \$s2, 1025 (X in \$s1, Y in \$s2)
 - A = 100; → addi \$s1, \$zero, 100 (A in \$s1)

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Let us see working with immediate values in registers. In any programming language, we need to add some constant value. So, those values are immediate values that we need to add. So, in such cases this can be specified in 16 bits and it occurs most frequently that is about 90% of the time you have to add some constant value. So, like $A = A + 16$. So, how will you do it, `addi $s1,$s1,16`. Now we want to do: `subi $s1,$s2,1025`. So, in `$s2` you have to load Y, and then finally `$s1` should be stored in X. $A = 100$; so in A you have to store this 100. So, `$s1` equals to 0, you add zero with 100 and you store it in `$s1` and `$s1`, A in `$s1` here.

(Refer Slide Time: 15:00)

The slide has a yellow background and a blue header bar. The title 'Case 2: Large constants' is at the top. Below it is a bulleted list:

- **Case 2:** Large constants, that require 32 bits to represent.
 - How to load a large constant in a register?
 - Requires two instructions.
 - A "*Load Upper Immediate*" instruction, that loads a 16-bit number into the upper half of a register (lower bits filled with zeros).
 - An "*OR Immediate*" instruction, to insert the lower 16-bits.
 - Suppose we want to load 0xAAAA3333 into a register `$s1`.

<code>lui \$s1, 0xAAAA</code>	<code>1010101010101010</code>	<code>0000000000000000</code>
<code>ori \$s1, \$s1, 0x3333</code>	<code>1010101010101010</code>	<code>0011001100110011</code>

The bottom of the slide features logos for IIT Kharagpur, NPTEL, and a female speaker, along with the text 'NPTEL ONLINE CERTIFICATION COURSES'.

In Case 2, we can see that how large constants that requires, say 32 bit to represent, can be loaded. It requires two instructions, the first one is load upper immediate. So, instruction that loads a 16-bit number into the upper half of the register, and then we OR immediate that instruction to insert the lower 16-bit. So, using these two instructions we can load a 32-bit numbers into a register. So, 32-bit number we cannot be loaded with a single instruction rather it requires two instructions to do this. So, load upper immediate. So, in the upper immediate, first we load this here and then or with this 0X3333. So, we or with this and finally, we get that entire thing.

(Refer Slide Time: 16:13)

Other MIPS Pseudo-instructions

Pseudo-Instruction	Translates to	Function
blt \$1, \$2, Label	slt \$at, \$1, \$2 bne \$at, \$zero, Label	Branch if less than
bgt \$1, \$2, Label	sgt \$at, \$1, \$2 bne \$at, \$zero, Label	Branch if greater than
ble \$1, \$2, Label	sle \$at, \$1, \$2 bne \$at, \$zero, Label	Branch if less or equal
bge \$1, \$2, Label	sge \$at, \$1, \$2 bne \$at, \$zero, Label	Branch if greater or equal
li \$1, 0x23ABCD	lui \$1, 0x0023 ori \$1, \$1, 0xABCD	Load immediate value into a register



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



So, there are other MIPS pseudo instructions like branch if less than, branch if greater than, branch if less than equal branch if greater equal. So, while doing programming you will see that you will be using many such pseudo instructions which is required for your programming, but all those pseudo instruction, when you run through any simulator it will get converted into low level MIPS32 instructions.

(Refer Slide Time: 16:48)

Pseudo-Instruction	Translates to	Function
move \$1, \$2	add \$1, \$2, \$zero	Move content of one register to another
la \$a0, 0x2B09D5	lui \$a0, 0x002B ori \$a0, \$a0, 0x09D5	Load address into a register
ble \$1, \$2, Label	sle \$at, \$1, \$2 bne \$at, \$zero, Label	Branch if less or equal
bge \$1, \$2, Label	sge \$at, \$1, \$2 bne \$at, \$zero, Label	Branch if greater or equal
li \$1, 0x23ABCD	lui \$1, 0x0023 ori \$1, \$1, 0xABCD	Load immediate value into a register



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



There are some more pseudo instructions like move, load address, branch if less than equal, branch if greater than or equal and load immediate.

(Refer Slide Time: 17:01)

The slide title is "A Simple Function Call". It compares C code and MIPS32 assembly code for a swap operation.

C Function	MIPS32 Code
<pre>swap (int A[], int k) { int temp; temp = A[k]; A[k] = A[k+1]; A[k+1] = temp; }</pre>	<pre>swap: muli \$t0,\$s0,4 add \$t0,\$s1,\$t0 lw \$t1,0(\$t0) lw \$t2,4(\$t0) sw \$t2,0(\$t0) sw \$t1,4(\$t0) jr \$ra</pre>

Annotations explain the assembly code:

- \$s0 loaded with index k
- \$s1 loaded with base address of A
- Address of A[k] = \$s1 + 4 * \$s0

Below the code, it says "Exchange A[k] and A[k+1]".

At the bottom, there are logos for IIT Kharagpur and NPTEL, and a video frame showing a person speaking.

Let us see a simple function call like for adding two numbers. We can simply add two numbers or we can also write a function that will take the two numbers and it will add it will take any two numbers and it will do the needful. Here in this case, it is a swap function. So, in temp we are taking a temporary variable of integer type, and then we are keeping A[k] in that temp, then A[k+1] is stored in A[k], and temp is stored in A[k+1].

Let us see how this can be written using a MIPS32 code. So, first we do muli \$t0,\$s0,4. So, \$s0 is loaded with index k, which index we want to swap that index. And the next one \$s1 is the loaded with the base address of A because this particular array we are taking. Then the address of A[k] will be \$s1 + 4 into that particular \$s0 what will be the index in the same way we did in the previous example. So, muli will multiply \$s0 into 4 and it will be stored in \$t0 then we are adding \$s1 and \$t0 and we are storing it in \$t0, then we load the two words. We load the two words and we store it in two registers that is \$t1 and \$t2. So, 0(\$t0), \$t0 is the base address where base address of A. So, we have added \$t0 with \$s1 where we have loaded the base address and that is what is in \$t0. So, 0(\$t0) means we will get this whatever is the value of k depending on that that particular word will be loaded in \$t1. And the next word will be loaded in \$t2.

Now, we simply have to store this particular word in this location, and this particular word in the next location. So, this is what we have done first we are loading the word from 0(\$t0) into \$t1; loading the next word that is \$t0 + 4 into \$t2. Now \$t2 we have to

store it in $0($t1)$, that is what we are doing store word $$t2$ into $0($t1)$ and then we have to store this $$t1$ into $4($t0)$ and jump to return address $$ra$. So, from the main program we came to a function that is swap, we perform this operation, and then we will go back to that function again.

(Refer Slide Time: 20:37)

MIPS Instruction Encoding

- All MIPS32 instructions can be classified into three groups in terms of instruction encoding.
 - R-type (Register), I-type (Immediate), and J-type (Jump).
 - In an instruction encoding, the 32 bits of the instruction are divided into several fields of fixed widths.
 - All instructions may not use all the fields.
- Since the relative positions of some of the fields are same across instructions, instruction decoding becomes very simple.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Let us now see MIPS instruction encoding. All MIPS instructions can be classified into three groups in terms of instruction encoding, you can classify it into three groups. So, the first one is R-type that is register typed; it has got I-type that is immediate and J-type which consists of jump. So, in an instruction encoding the 32 bits of the instructions are divided into several fields of fixed widths. We have already seen instruction encoding from a general perspective. In this case we will be seeing instruction encoding for MIPS and all instruction may not use all the fields, but those will be specified, those are fixed fields. An instruction may use it an instruction may not use it.

Since the relative positions of some of the fields are same across instruction, instruction decoding becomes very simple. So, as it is already known that this particular field from this particular position to this particular position will be this field; from this particular position to this particular position it will be the other field, and so on. So, the instruction decoding becomes very simpler.

(Refer Slide Time: 22:11)

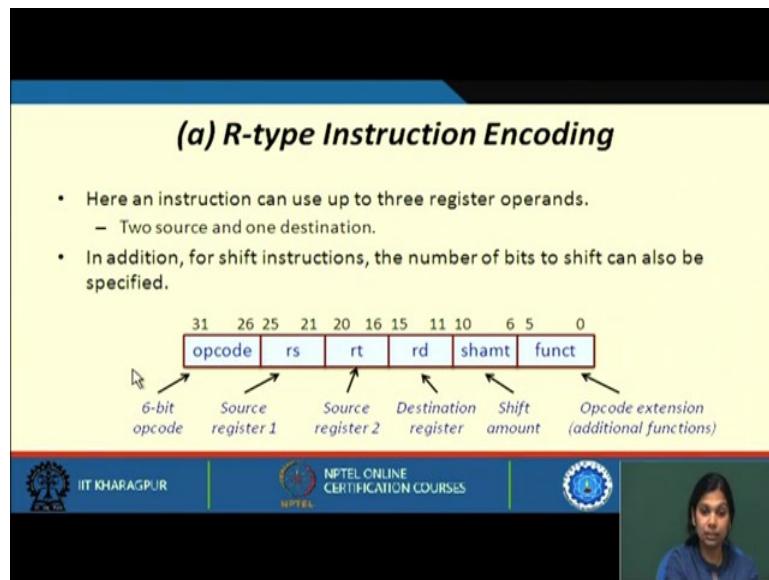
(a) R-type Instruction Encoding

- Here an instruction can use up to three register operands.
 - Two source and one destination.
- In addition, for shift instructions, the number of bits to shift can also be specified.

31	26	25	21	20	16	15	11	10	6	5	0
opcode	rs		rt		rd		shamt		funct		

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

6-bit opcode Source register 1 Source register 2 Destination register Shift amount Opcode extension (additional functions)



Let us first see R-type instruction encoding. Here an instruction can use up to three register operands. So, there will be an opcode, there will be two source registers rs and rt, and one destination rd. As you know there are 32 total registers, so 5 bits will be specified for this. In addition to this opcode and registers, you will have for shift instruction the number of bits to shift can also be specified, using this particular field shamt. Let say you want to shift a particular value right to these many position that how much bit position can be specified in this instruction itself. And for that opcode what is the function that is specified here. So, this is an R-type instruction encoding, this is as I said 6-bit opcode, source register one, this is source register two, this is destination register, this is the shift amount the amount of shifting you wanted to do that can be specified here, and the opcode extension. So, if this opcode let say 00000 then we will say that this is an ALU operation and based on that then we will see which ALU operation it is add or mul or div sub etc.

(Refer Slide Time: 24:00)

The slide contains the following text:

- Examples of R-type instructions:
add \$s1, \$s2, \$s3
sub \$t1, \$s3, \$s4
sla \$s1, \$s2, 5 // shift left \$s2 by 5 places, and store in \$s1
- An example instruction encoding: *add \$t1, \$s1, \$s2*
 - Recall: \$t1 is R9, \$s1 is R17, and \$s2 is R18.
 - For "add", opcode = 000000, and funct = 100000,

Below the text is a binary representation of the instruction:

31	26	25	21	20	16	15	11	10	6	5	0
000000	10001	10010	01001	00000	100000						

At the bottom of the slide are logos for IIT Kharagpur, NPTEL, and National Institute of Technology Meghalaya, along with the number 35.

Let us see some example of R-type instruction add \$s1,\$s2,\$s3 like this. So, these are source registers and this is the destination register recall \$t1 is R9, \$s1 is R17 and \$s2 is R18. And for add this opcode is 000000 is for ALU and for add it is 100000. So, now if you see this encoding, so this here goes your opcode that is 000000 and then the function goes here that is 100000. First register is \$t1, or R9, so this is 01001. Similarly, \$s1 and \$s2 are the two source registers R 17 or 10001, and R18 or 10010.

(Refer Slide Time: 25:17)

The slide has the title **(b) I-type Instruction Encoding**.

The text below the title lists:

- Contains a 16-bit immediate data field.
- Supports one source and one destination register.

Below the text is a diagram of the I-type instruction format:

31	26	25	21	20	16	15	0
opcode	rs	rt	Immediate Data				

Annotations below the diagram identify the fields:

- 6-bit opcode
- Source register 1
- Destination register
- 16-bit immediate data

At the bottom of the slide are logos for IIT Kharagpur, NPTEL, and National Institute of Technology Meghalaya, along with a video frame showing a person speaking.

Next move on with the next type of instruction that is I-type instruction; it contains 16-bit immediate data value. Now, suppose this total 16-bit is an immediate data and you have a source register and you have a destination register, this is all you have and you have a single opcode. So, for any immediate value, we do not require other fields, but see these values are fixed, this is source, this is destination, this is opcode, but this 16-bit immediate value has changed for I-type instruction. So, this is 6-bit opcode, source register, destination register and this is 16-bit immediate data.

(Refer Slide Time: 26:16)

- Examples of I-type instructions:

lw	\$s1, 50(\$s5)
sw	\$t1, 100(\$s1)
addi	\$t0, \$s1, 188
beq	\$s1, \$s2, Label // Label is encoded as a 16-bit offset relative to PC
bne	\$s3, \$zero, Label
- An example instruction encoding: *lw \$t1, 48(\$s1)*
 - Recall: \$t1 is R9, \$s1 is R17.
 - For "lw", opcode = 100011, and funct = 100000,

31	26	25	21	20	16	15	0
100011	10001	01001	0000000000110000				→

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

A video thumbnail of a lecture is visible on the right side of the slide.

Now, you see some of the I-type instructions in MIPS32. Load word (lw), so from this memory location $50 + \$s5$ it will load that particular word from this memory location into here. So, all these type of instructions are I-type. So, let us see this example $lw \$t1, 48(\$s1)$. $\$t1$ is the destination register and this one $\$s1$ is the source. And 48 is the immediate value and the opcode for lw is 100010. So, let us see how this will be encoded. Now, this is the opcode that I have written here $\$t1$ is R9 or 01001, I have to load the word into this destination registers and $\$s1$ is the source register, which is R17 or 10001. And this is my immediate value that is 48 is loaded here. So, this is a kind of I-type instruction.

(Refer Slide Time: 28:04)

(c) J-type Instruction Encoding

- Contains a 26-bit jump address field.
 - Extended to 28 bits by padding two 0's on the right.
- Example: *j Label*

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Let us go to J-type instruction encoding it contains a 26-bit jump address field. And this 26-bit jump address field is padded with two zeros, so that it can be extended to 28 bits. And an instruction like *j Label*. So, when we give jump to Label, so it goes to a 26-bit address by adding two more bits it becomes 28 bit and we move there. So, these this type of instruction is jump type instruction 6-bit opcode and 26-bit jump address.

(Refer Slide Time: 28:55)

A Quick View

	31	26	25	21	20	16	15	11	10	6	5	0	
R-type	opcode	rs	rt	rd	shamt	funct							
I-type	opcode	rs	rt	Immediate Data									
J-type	opcode	Immediate Data											

- Some instructions require two register operands *rs* & *rt* as input, while some require only *rs*.
- Gets known only after instruction is decoded.
- While decoding is going on, we can prefetch the registers in parallel.
 - May or may not be required later.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, now you see that there is a clear place this is your source, this is another source, this destination and this can be a destination for some. And this is the immediate data and this

is the immediate data for the next one. But now if you have to check the opcode field, you can check the opcode field for all from the first part and of course, for some you have to check this. Some instruction requires two register operands like we said rs and rt as input while some requires only rs. And get to know that only after instruction decoded, after the instruction get decoded then only we will know that ok, it has got these many source register this many destination register.

And while decoding is going on we can prefetch the register in parallel may or may not be required later. So, what I am trying to say that we already know that this is a source this is a source this is a source. So, well in advance we can prefetch it, maybe we may not be requiring it for a jet you may for this j type instruction we will not be requiring this rs and rt, but in that time we can already prefetch it because for this instruction and this instruction we may might require it later. So, if you prefetch it, we may require it and we can save some time.

So, similarly this 16-bit address and this 26-bit address immediate data, these are immediate data some address. These immediate data retrieved and then we can do a sign extension to make it 32 bit. We already know what is sign extension. So, we can extend this to 32 bit and if it is required later it is fine if it is not required later it, but we can do this.

(Refer Slide Time: 31:11)

The slide has a blue header bar with the title 'Addressing Modes in MIPS32'. Below the title is a bulleted list of addressing modes:

- Register addressing *add \$s1, \$s2, \$s3*
- Immediate addressing *addi \$s1, \$s2, 200*
- Base addressing *lw \$s1, 150(\$s2)*
 - Content of a register is added to a "base" value to get the operand address.
- PC relative addressing *beq \$s1, \$s2, Label*
 - 16-bit offset is added to PC to get the target address.
- Pseudo-direct addressing *j Label*
 - 26-bit offset if shifted left by 2 bits and then added to PC to get the target address.

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the NPTEL logo, and the text 'NPTEL ONLINE CERTIFICATION COURSES'. To the right of the footer, there is a small video window showing a person speaking.

Now, what are the addressing modes in MIPS we have, we have register addressing, we have immediate addressing, we have base addressing where the content of register is added to a base value to get the operand address, we have relative addressing here and we have pseudo direct addressing. So, the 26-bit offset is shifted by left by 2 bits and then added to PC to get the target.

So, we came to the end of lecture 10 and next we will see some of the programs that we can do using MIPS.

Thank you

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture – 11
SPIM – A MIPS32 Simulator

Welcome to lecture 11. In this lecture I will be talking about SPIM; a MIPS32 simulator; how you can write programs in SPIM.

(Refer Slide Time: 00:39)

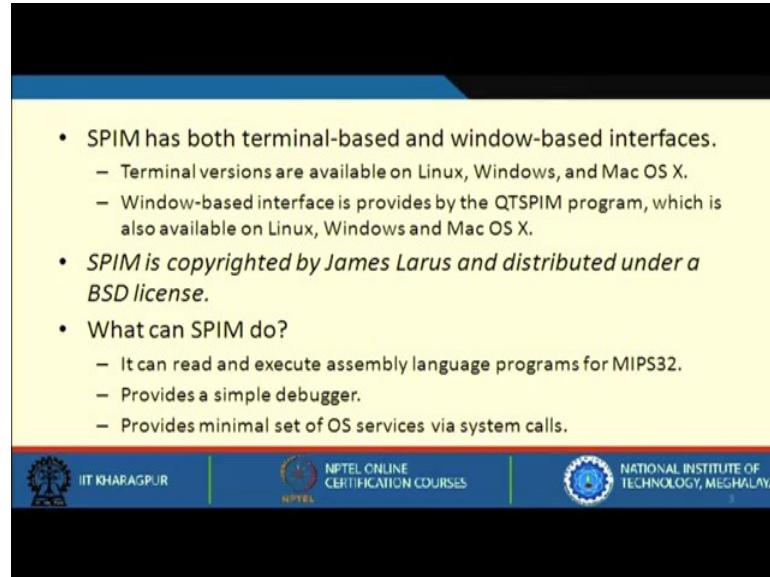
How to Run MIPS32 Programs?

- Best way to learn MIPS32 assembly language programming is through a simulator.
 - SPIM is a self-contained simulator available in the public domain that runs MIPS32 programs.
 - Available for download in <http://spimsimulator.sourceforge.net>
 - SPIM implements almost the entire MIPS32 instruction set (along with the extensions, viz. pseudo-instructions).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

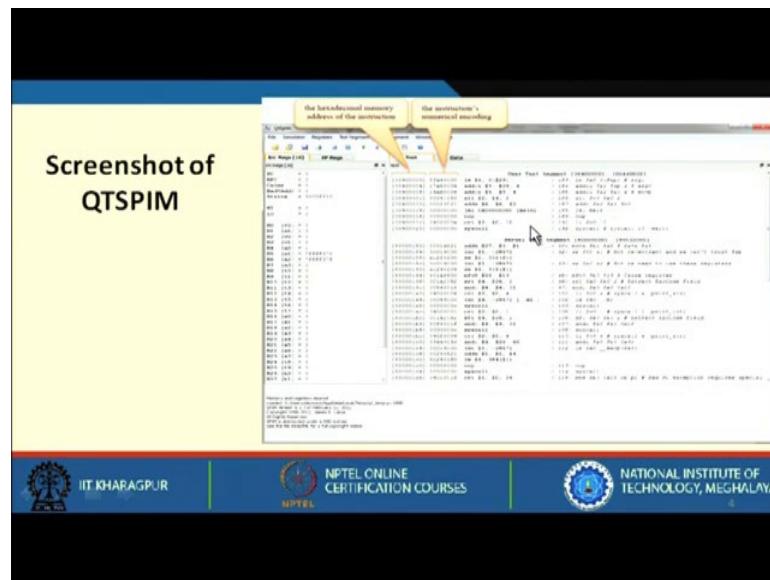
As you all know the best way to learn any assembly language is through a simulator and we should start coding. So, how to start that; we need a simulator which we will be using (SPIM) that is a self contained simulator and it is available in public domain and you can download that from this particular web page. SPIM implements almost the entire MIPS32 instruction set along with the extension with pseudo instructions. What do you mean by pseudo instructions? I already discussed about pseudo instructions like in MIPS.

(Refer Slide Time: 02:09)



SPIM has both terminal based and window based interfaces which are available; it is up to you that which you will be using. SPIM is copyrighted by James Larus and distributed under a BSD license. So, we must know what SPIM can do --- it can read and execute assembly language programs for MIPS32 and provide a simple debugger and also provide minimal set of OS services via system calls. This is a screen shot of a QTSPIM.

(Refer Slide Time: 03:09)



So, if you install QtSPIM in your machine, you will be able to see a view like this. Here these are the registers that are having different values, and these are the hexadecimal addresses, and this is the instruction after encoding.

So, these are the list of instructions and this is the encoded form of the instructions. We have already learnt how we can encode a particular instruction knowing how many bits of registers are present, etc. So, this is basically the main instructions that are getting executed, this is the encoded form of the instructions, and this is the memory addresses and you can see the memory addresses will be incremented by 4, as we have 32-bit instructions. It is byte addressable, and so it will be added plus 4.

(Refer Slide Time: 04:25)

```
.text          # code section
.globl main    # starting point, must be global
main:
        # user program code goes here
.data          # data section
        # user program data goes here
```

Now, let us see the MIPS32 assembly code layout; it has got a text section, it has got a data section. In this section we have something called **globl main**. This actually shows the starting point that must be global, and now this is .global main, and main is the label from where your program execution starts. If you do not give this in the program it will be wrong. So, you have to mention that your main is global, and this starts from a particular level (that is main) and then you can write the user program code here.

(Refer Slide Time: 05:32)

Assembler Directives

- a) **.text**
 - Specifies the user text segment, which contains the instructions.
- b) **.data**
 - Specifies the data segment, where all the data items are defined.
- c) **.globl sym**
 - Specifies that the symbol "sym" is global, and can be referred from other files.
- d) **.word w1, w2, ..., wn**
 - Stores the specified 32-bit numbers in successive memory words.
- e) **.half h1, h2, ..., hn**
 - Stores the specified 16-bit numbers in successive memory half-words.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | GATE

And the data portion is written in this part. These are some assembler directives --- .text specifies the user text segment which contains the instructions that are required to execute. We will see some examples in course of time. Then, .data specifies the data segment where we put all the data that will be used in our program. Then .globl sym specifies the starting point, but this symbol is global and can be referred from other files as well. So, from many other files you will be able to access this.

Next is .word --- it stores the specified 32-bit numbers in successive memory words.

(Refer Slide Time: 06:51)

© CET
I.I.T. KGP

```
0x 00000000+4 30
0x 00000004+4 31
0x 00000008    32
:
:
```

Let us say hexadecimal address starts from 0 0 0 0 0 0; your next address will be 0 0 0 0 0 4, and so on. So, it is increasing as plus 4 plus 4 and so on.

(Refer Slide Time: 08:00)

The screenshot shows a presentation slide with a yellow background. At the top, there is a blue header bar. Below the header, the slide content is as follows:

- f) `.byte b1, b2, ..., bn`
 - Stores the specified 8-bit numbers in successive memory bytes.
- g) `.ascii str`
 - Stores the specified string in memory (in ASCII code), but do not null-terminate it.
 - Strings are enclosed in double quotes and follow C-like convention ("\\n", etc.).
- h) `.asciiz str`
 - Stores the specified string in memory (in ASCII code), and null-terminate it.
- i) `.space n`
 - Reserve space for n successive bytes in memory.

At the bottom of the slide, there is a footer bar with three logos and text: IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and a circular logo. To the right of the footer, there is a small video window showing a person speaking.

`.byte` stores a 8-bit specified number in successive memory bytes. We can also specify characters. So, `.ascii` and `.asciiz` are the two directives that are used to store the specified string in memory, but do not null terminate it. So, it is not null terminated --- a string is null terminated by “`\0`”. `.asciiz str` specifies a string in memory and it is also null terminated.

`.space n` reserves a space for n successive bytes of memory.

(Refer Slide Time: 09:09)

Register Naming Conventions

- Already discussed earlier :: quick recall :-
 - \$zero constant zero
 - \$at reserved by assembler
 - \$v0, \$v1 for parameter passing
 - \$a0 to \$a3 for arguments
 - \$t0 to \$t9 temporary registers (not saved by callee)
 - \$s0 to \$s7 registers (saved by callee)
 - \$gp global pointer
 - \$sp stack pointer
 - \$ra return address

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

For register naming convention we already discussed about all these things, and this is how it is named: \$zero, \$at, \$v0, \$v1, \$a0 to \$a3, \$t0 to \$t9, \$s0 to \$s7, \$gp, \$sp and \$ra.

(Refer Slide Time: 09:26)

Pseudo-instructions

- The MIPS32 pseudo-instructions, as discussed earlier, are all supported by SPIM.
- SPIM converts these into MIPS32 instructions before executing them.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

As you know MIPS pseudo instructions as discussed earlier are all supported by SPIM. SPIM converts these into MIPS32 instructions before executing them. So, we can write a program using pseudo instruction, but this pseudo instruction in turn gets converted into MIPS32 instruction before it gets executed.

(Refer Slide Time: 10:00)

Operating System Interface: "syscall"			
Service	Code (put in \$v0)	Arguments	Result
print_int	1	\$a0 = integer	
print_string	4	\$a0 = address of string	
read_int	5		int in \$v0
read_string	8	\$a0 = address of buffer, \$a1 = length	
exit	10		

Other system calls for floating-point numbers also exist



IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

For operating system interface syscall is used, like when you want to get a value from keyboard, display in the monitor, etc. In such cases you need to use syscall. So, if you want to print an integer you have to put the argument in \$a0 and you have to put the code \$v0 = 1 to print an integer. Similarly for printing a string you need to put the code 4 in \$v0 and the address of the string in \$a0. And then you perform a syscall then it will print that mean on the screen this value will get displayed, either integer or a string. Similarly we reading an integer it will read from the keyboard and store it in \$v0.

Similarly, for read string the code is 8, and in \$a0 the address of the buffer where you want to read it that will be given and how much length string you want to read that is given in \$a1. Also the code for exit is 10 and after putting the required value in these registers you have to do syscall. Other system calls for floating point numbers also exists which is not included here. Let us take a sample program.

(Refer Slide Time: 12:46)

The slide has a yellow header bar with the title "Example Program 1". Below it is a white content area containing assembly code in a red-bordered box. To the right of the code is a descriptive text. At the bottom is a blue footer bar with logos for IIT Kharagpur, NPTEL, and NIT Meghalaya.

Example Program 1

```
.text
.globl main

main: la    $t0, value
      lw    $t1, 0($t0)
      lw    $t2, 4($t0)
      add   $t3, $t1, $t2
      sw    $t3, 8($t0)

.data
value: .word 50, 30, 0
```

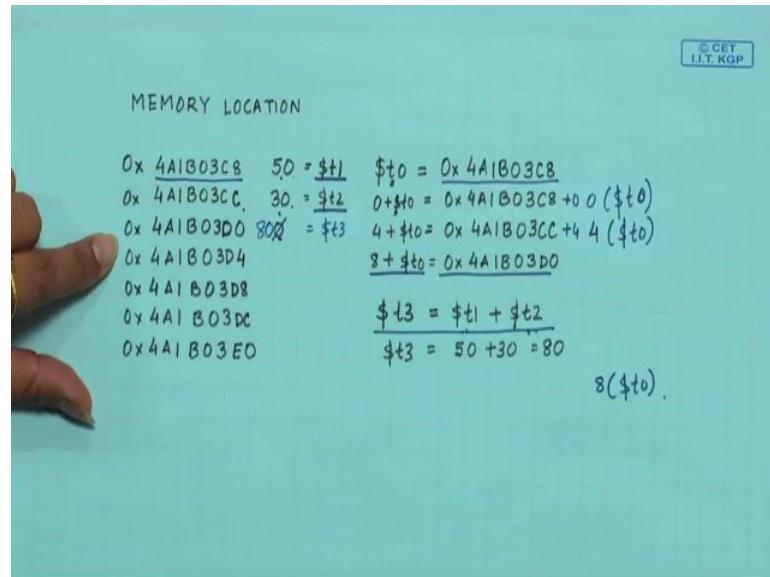
Add two numbers in memory and store the result in the next location.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us see how this program is executed. So, in the .text we write .globl main and main is the label from where the execution of program starts. So, what this program is actually doing let us also look into the value; value is also a label which consists of some word and each word is of 32 bit and what it stores in the first location; it stores 50 and in the next location it stores 30, and the result will be stored in next location which is initialized with 0.

Now, just see this --- la meaning load address from value and store it in \$t0. \$t0 is a temporary register where we are loading this value. Once we load the address of this value in \$t0 we need to load the word 50 and 30. So, let us take an example to show this.

(Refer Slide Time: 14:27)



Now, you see these are my memory location 4A1B03C8 --- in this memory location we have stored 50, in the next memory location we have stored 30 depending on the value, and in this location we have stored 0. The various steps are shown.

(Refer Slide Time: 18:42)

The screenshot displays a slide titled "Example Program 2". The program code is as follows:

```
.text
.globl main
main: add    $t1, $zero, 0x2A
      add    $t2, $zero, 0x0D
      add    $s3, $t1, $t2
```

To the right of the code, a descriptive text reads: "Add two constant numbers specified as immediate data, and store the result in a register."

At the bottom, there are logos for IIT Kharagpur, NPTEL, and National Institute of Technology Meghalaya, along with their respective names.

Now, this is a program that adds two constant numbers specified as immediate data and store the result in a register. So, here we are adding some immediate values 0x2A and 0x0D. What we are doing we are adding this immediate value with \$zero and then we are

storing it in \$t1; why because we do not have a move instruction. So, instead we are doing directly this here and then we are again loading this particular data into \$t2 and then we are adding these two and storing it in register \$s3. So, these 3 steps will add two immediate value that is 0x2A and 0x0D into \$s3.

(Refer Slide Time: 19:46)

The slide has a blue header bar at the top. Below it is a yellow main content area. In the center of the yellow area, there is a title 'Example Program 3'. To the right of the title, a text box contains the following assembly code:

```
.text
.globl main

main: add    $t1, $zero, 0x2A
      add    $t2, $zero, 0x0D
      add    $s3, $t1, $t2

      li     $v0, 10
      syscall
```

To the right of the code, a note says: 'The same program but using system call to exit.'

At the bottom of the slide, there is a red footer bar with three logos and text: 'IIT KHARAGPUR', 'NPTEL ONLINE CERTIFICATION COURSES', and 'NPTEL'.

Coming to the next program that is the same program, but using a system call. In this system call if you recall if we load 10 in \$v0 and then we do a syscall then it will exit. So, this particular program is same as the previous one that is adding two immediate values, but at the end we have to load an immediate number into \$v0. So, in \$v0 value 10 will be loaded and once we do system call then it will get exit. So, it will exit from here.

(Refer Slide Time: 20:31)

The slide has a blue header bar. Below it, the title 'Example Program 4' is displayed in bold black font. To its right is a text box containing the assembly code. At the bottom of the slide, there are three logos: IIT Kharagpur, NPTEL, and NIT Meghalaya.

Example Program 4

Read two numbers from the keyboard and print the sum.

```
.data
str1: .asciiz "Enter first number: "
str2: .asciiz "Enter second number: "
str3: .asciiz "The sum is = "

.text
.globl main

main: li $v0, 4          # print string
      la $a0, str1
      syscall

      li $v0, 5          # read integer
      syscall
      move $t0, $v0
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us see a program. Here we read 2 numbers from the keyboard and print the sum. So, first of all we have to read 2 numbers. So, initially in the data section we initialize the three strings, now see all these numbers are entered from the keyboard. So, in the data section this str1 specifies “Enter the first number:”, str2 specifies “Enter the second number:”, and str3 specifies “The sum is =”.

So, first we have to display this “Enter the first number:” and then once it is displayed we need to read a value from the keyboard, let us see how we can do this. The process is repeated for the other number also.

(Refer Slide Time: 24:29)

```
li    $v0, 4
la    $a0, str2
syscall

li    $v0, 5
syscall
move  $t1, $v0

add $t1, $t0, $t1
# $t1= $t0 +
$t1

li    $v0, 4
la    $a0, str3
syscall
```

```
li    $v0, 1
move  $a0, $t1
syscall

li    $v0, 10
syscall
```

So, this is a program that reads 2 integers from the keyboard perform addition and store back the result in another and displays the result in the keyboard itself.

(Refer Slide Time: 26:35)

Example Program 5

Calculate sum of 10 32-bit numbers stored in consecutive memory locations.

```
.data
num: .word  1, 2, 3, 4, 5, 6 ,7, 8, 9, 10
.text
.globl main

main:
    la    $t0, num
    li    $t2, 0          # holds the sum
    li    $t3, 0          # counter for loop
loop: lw    $t1, 0($t0)
      add   $t2, $t2, $t1
      addi  $t3, $t3, 1
      addi  $t0, $t0, 4  # point to next
      bne   $t3, 10, loop

    li    $v0,10
    syscall
```

Now, let us see how we can do some other programs like adding 10 numbers. So, till now we are talking about simple program loading a value adding those values storing it back, but loops and other things are very common in programming. So, we often encounter loops everywhere whenever we write a program how loops will be executed using this SPIM how we can write loops using SPIM we will be seeing in now. So, we

will see how we can calculate sum of 10 numbers and the 10 numbers are stored in consecutive memory locations, and these 10 numbers are will be added and stores back in another memory location. So, we need to have one counter that will start from here if it starts from 0 it will go till 9; 0 1 2 3 4 5 6 7 8 9. So, less than 10 if you start from 1 it will be less than equal to 10.

(Refer Slide Time: 32:25)

The slide has a blue header bar. Below it, the title 'Example Program 6' is displayed above a text box containing the assembly code. The text box has a red border. At the bottom of the slide, there is a footer bar with three sections: IIT Kharagpur logo, NPTEL Online Certification Courses logo, and a video player showing a person speaking.

Example Program 6

Check if a given number is a palindrome.

```
.data
num: .word 0
msg: .asciiz "Enter the Number: "
msg1: .asciiz "Palindrome"
msg2: .asciiz "Not Palindrome"

.text
.globl main
main:
    li      $v0, 4
    la      $a0, msg
    syscall
```

The next example is another program which checks if a given number is palindrome or not. So, what is a palindrome? A palindrome is a number if you reverse the number it will be the same.

(Refer Slide Time: 32:41)

PALINDROME

$$\begin{array}{l} 10^2 \ 10^1 \ 10^0 \\ 2 \ 1 \ 2 = 2 \times 10^0 + 1 \times 10^1 + 2 \times 10^2 \\ = 2 + 10 + 200 \\ = 212 \end{array}$$
$$\begin{array}{l} \overset{D_2 \ D_1 \ D_0}{\underline{2 \ 1 \ 2}} = \overset{D_0 \ D_1 \ D_2}{\underline{2 \ 1 \ 2}} \\ 212 = 212 \\ 123 = 321 \end{array}$$

CONSIDER 123

COUNTER :
Initially Counter = 0

$$\begin{array}{ll} 123 \% 10 = 3 & 0 \times 10 + 3 = 3 \\ 123 / 10 = 12 & \\ \\ 12 \% 10 = 2 & 3 \times 10 + 2 = 32 \\ 12 / 10 = 1 & \\ \\ 1 \% 10 = 1 & 32 \times 10 + 1 = 321 \\ 1 / 10 = 0 & \end{array}$$

So, if you have a number 2 1 2 if you reverse the number it will be first 2 will come, then 1 will come, then 2 will come. So, both the numbers are same. Now let us have a number 1 2 3 is this palindrome number? You reverse the number 3 2 and 1 this is not a palindrome number because this is not same as this.

Let us take an example 1 2 3; basically what we need to do we need to extract the last digit. So, we need to multiply it with 10 and add it with the extracted digit and we store it whatever we get again we multiply with 10 and add it with the last digit that is the remainder.

(Refer Slide Time: 34:18)

The image shows handwritten mathematical steps on a light blue background. On the left, under 'CONSIDER' in capital letters, is a box containing the number '123'. Below it, the division $123 \div 10 = 3$ is written, with '3' underlined. Next, $123 \div 10 = 12$ is shown, with '12' underlined. Then, $12 \div 10 = 2$ is written, with '2' underlined. Finally, $1 \div 10 = 1$ is shown, with '1' underlined. To the right, under 'COUNTER' in capital letters, is the note 'initially counter = 0'. Below it, $0 \times 10 + 3 = 3$ is written, with '3' underlined. Then, $3 \times 10 + 2 = 32$ is shown, with '32' underlined. At the bottom, there is a subtraction problem: $123 - 321$, with a horizontal line under the problem. To the right of the subtraction, a hand is writing the result '321' in a box, with '321' underlined.

So firstly, let me explain the concept if we consider a number 1 2 3, we take the remainder of the number. So, the remainder will be 3. So, we have extracted the last digit. So, $123 \bmod 10$ we get 3 which is the remainder. So, initially you have you have to take a counter that is 0. So, we multiply 0 with 10 and we add with the remainder. So, we get 3 here. So, we are this is the 0th one. So, initially the counter was 0 it has multiplied with 10 and added with 3 now my counter is 3 and what I do again we need to get the quotient. So, $123 \div 10$ we get 12 as the quotient.

Now, we take this 12 again we take the remainder we get 2; once you get the remainder as 2 the previous counter value was 3 we multiply 3 with 10 and we add it with 2 we get 32 now my counter has changed to 32; now how many times you will be doing it till you get a 0 quotient. So, again for the next time the quotient is 1. So, $1 \bmod 10$ it becomes one initial counter was 32 then we have to do 32 into 10 and I added one I add with one and we get 321, now 1 divided by 10 which is 0. So, we will not continue. So, we got the value which is the reverse of this 1 2 3. So, initially we have taken 1 2 3, we repeated certain steps we kept a counter in that counter we were storing the reverse value and finally, we got the value 321. So, 1 2 3 the reverse is 321, but now this is not equal to this. So, it is not a palindrome.

Let us see how we can code this using QTSPIM. So, here similarly we have to enter a number and this should be two messages; if it is a palindrome it should display palindrome, if it is not palindrome it will display not palindrome.

(Refer Slide Time: 37:18)

```

    li      $v0, 5
    syscall
    move   $t0, $v0
    move   $t3, $t0
    li      $t2, 0

loop:
    mul    $t2, $t2, 10
    rem   $t1,$t0,10
    div    $t0,$t0,10
    add    $t2, $t2, $t1
    bne   $t0,$zero,loop
    bne   $t3, $t2, np

np:
    li      $v0, 4
    la     $a0, msg1
    syscall

    li      $v0, 10
    syscall

    li      $v0, 10
    syscall

```

The screenshot shows the QTSPIM assembly editor interface. At the bottom, there are logos for IIT Kharagpur and NPTEL, along with the text "NPTEL ONLINE CERTIFICATION COURSES". On the right side of the slide, there is a small video window showing a person speaking.

Now first of all this message “Enter the number.” should come. So, we load 4 in \$v0 and the message in \$a0 and do a syscall then we enter the number. So, we load 5 in \$v0 we do a syscall the value the number entered is stored in \$v0 which is moved to \$t0 we also move the value \$t0 to \$t3. So, \$t3 and \$t0 both contains my number for which I have to check whether it is palindrome or not; now I am loading an immediate value \$t2, where in \$t2 I am initializing it with 0. So, the same process that I have explained I am doing it in a loop. So, what I am doing initially the counter value is 0. So, 0 is multiplied with 10 and it is stored in \$t2; then my number is in \$t0 the number for which have to calculate the palindrome is in \$t0, I divide it with 10 and I get the remainder in \$t1. So, my remainder is now in \$t1, I divide it \$t0 by 10, I get the quotient in \$t0. So, \$t1 contains the remainder and \$t0 contains the quotient.

Now, I am adding the counter \$t1 with the counter value that is \$t2 and I am storing back in \$t2, next what I am checking whether \$t0 is equals to 0 or not; that means, my quotient has become 0 or not; if my quotient has become 0 then I will not loop it. So, branch if not equal if \$t0 is not equal to 0; again you go to the loop. Now you are taking the updated value of \$t2 which is depending on the last digit; that last digit is multiplied

by 10 and stored in \$t2 again similarly for that value \$t1 again it for that value \$t0 we will be dividing it with 10 we get the remainder we divide with 10 we get the quotient and we again add it.

So, we keep on doing this until we get this \$t0 is 0; that means, until the quotient become 0. Once the quotient becomes 0, we check the next thing what we are checking see in the first step we have also stored the number in \$t3 why we have stored the number and \$t3 because we have to finally check because the number is palindrome or not; we first reverse that number and now I am checking whether the number is palindrome or not.

So, \$t3 is checked with \$t2 that is the counter that I have kept if it is not equal then you go to a label np that is not palindrome and where you load this message that this particular number is not palindrome, but if it is not equal then only we are going in let us see if it is equal then this statement will not get executed and we will directly come to this statement where we will display the message the number is palindrome. So, just see what we have done and then we exit it. So, we have put one exit here one exit here why because if it is not palindrome it will come here and then, the exist code it will encounter here, but if it is palindrome then it will go here and then it will also exist with this particular code.

(Refer Slide Time: 41:49)

Function Calls in MIPS32

- MIPS uses the jump and link instructions.
 - Control is transferred to the function using the `jal` instruction.
 - The `jal` instruction jumps to a label and stores the PC value in the `$ra` register.
 - To transfer the control back to the caller program we use: `jr $ra`.

IIT Kharagpur | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

So, this code actually shows whether a number is palindrome or not. So, we have written an assembly language code which shows a number is palindrome or not. So, these are

function calls in MIPS it uses jump and link instruction; we call it jal instruction and what it does it jumps to a label and it stores the PC value in the return address register \$31. And once the control after the subroutine is executed it has to return back to that particular place. So, returning back to that place it has to do jr to return automatically to that particular address; it will load the previous value of pc which was stored and it will go on executing it.

(Refer Slide Time: 42:34)

```

Example
Program

Function call and
return.

        .data
num1: .word 14
num2: .word 15
sum: .word 0

        .text
main:
        lw      $t0, num1
        lw      $t1, num2
        jal   SumFunc
        sw      $t1, sum
        li      $v0,10
        syscall

SumFunc:
        add $t1,$t1,$t0
        jr $ra

```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, this is a simple example of function call. So, 2 numbers are stored in these 2 locations. So, we load these 2 numbers here load word from num1 and num2 in these 2 registers \$t0 and \$t1 and then we are doing a jal, where we are going to SumFunc. In SumFunc we are adding \$t1 and \$t0 and storing in \$t1 and what we are writing here after this execution of this we are writing jump to return address that is \$ra. So, it will load that address and it will come here because the pc value initially when you are accessed this pc value, then the pc value was incremented to the next one which was pointing to "sw \$t1,sum" --- but now you have executed a jal instruction where it has moved here it will execute this particular instruction and then it will jump to return address and it will get the return address from register 31 which is \$ra and it will execute this statement now starting from this statement.

Now, the pc value will be loaded with this again after the execution of jr \$ra and then these 2 instructions will get executed. So, this is how function call happens in MIPS. So,

now, we have come to end of module 2. In module 2 we have seen generally what is the kind of instruction format, addressing modes that are there, and then specifically we have gone into the details of MIPS32 instruction set and we have discussed about a simulator that is SPIM where we can write programs in assembly language.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 12
Measuring CPU Performance

Welcome to week 3 lecture. In the last couple of weeks what we have studied is how we can execute an instruction, what the various architectures that are existing, what are addressing modes, instruction format and various other things that are necessary for this particular course. Apart from that, we have talked about a simulator that is SPIM, and we have seen that how we can write programs using low level assembly language. In this particular week, we will be looking into how we can measure the performance of a CPU. We know that for any program, you need some instructions to execute that particular program.

Now how you can actually measure the CPU performance. By that I mean, that you can run the same program in 2 different CPU, and then how you can tell that which one is better. So, in this particular week we will be looking into the various aspects of CPU performance. And then we can say at the end of this week lecture, that how can you say that my CPU is better than the other CPU.

(Refer Slide Time: 01:57)

Introduction

- Most processors execute instructions in a synchronous manner using a clock that runs at a constant *clock rate* or *frequency* f .
- Clock cycle time C is the reciprocal of the clock rate f :
$$C = 1/f$$
- The clock rate f depends on two factors:
 - a) The implementation technology used.
 - b) The CPU organization used.
- A machine instruction typically consists of a number of elementary micro-operations that vary in number and complexity depending on the instruction and the CPU organization used.

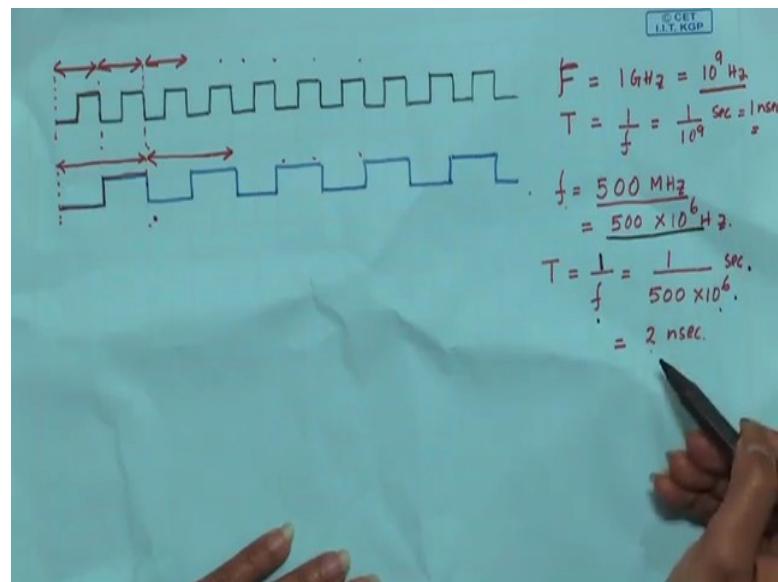
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, coming to the introduction, we know that most processors execute instructions in a synchronous manner using a clock that runs at a constant rate or frequency. So, what do we mean by that.

Now, how instructions get executed through a clock; that means, a clock is coming? And at the positive edge of the clock or within that clock period, we can say certain task is performed.

Now what is clock cycle time? Clock cycle time is the reciprocal of clock rate. That is C is $1 / f$. Clock cycle time is often termed as clock period, which is the reciprocal of frequency, that is $1 / f$. First let us see these 2 factors f and C in some detail.

(Refer Slide Time: 03:29)



So, this is a clock, and this also a clock. Let us say in this clock, this is the off period, this is the on period. And this whole is one period. And let us say we have another clock whose period is little more.

So what we are doing? Here this is the off period and this is the on period. So, this is the total clock. Similarly for this is the off period, this is the on period. So, this is your total clock and so on. Now you see that, for this particular clock the time period is small, that is, your clock cycle time is small. And in this clock, the clock cycle time is more. And what we know that, in processors we perform a task with respect to these clocks. So, whenever this clock is coming a particular task is getting performed. And so what we can

analyze from these 2 clocks. Let us see that for the first clock let us say the frequency is 1 GHz.

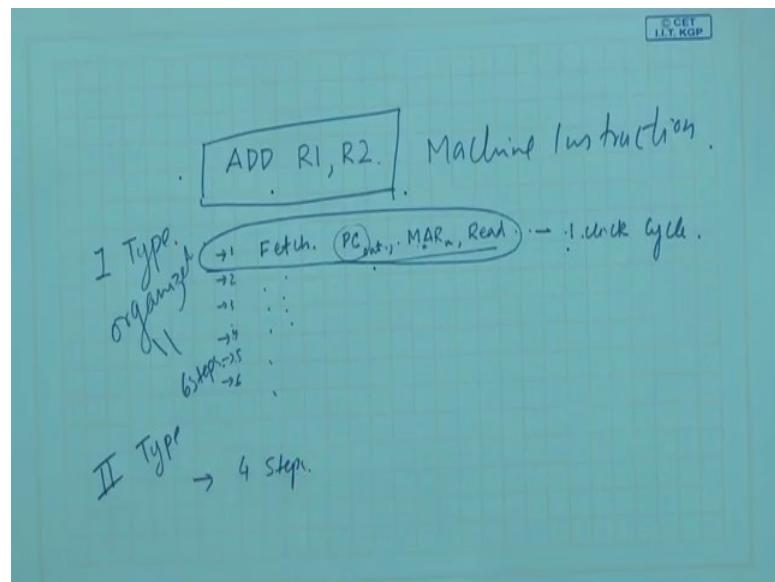
So what will be the time period? As I said time period will be $1 / f$. So, time period will be $1 / 10^9$ second, that is 1 nanosecond. Now let us see about this particular clock. In this clock the frequency is 500 MHz. So, as it is 500 megahertz, then $f = 500 \times 10^6$ Hz. So, time period $T = 1 / f = 2$ nanoseconds. That means, for the previous clock the time period is 1 nanosecond. For this one, the time period is 2 nanosecond.

So which will be faster; obviously, the first one will be faster than the next one. So, now we know how we can relate clock frequency with clock period.. Next we see that on what factor does this clock rate depend on. So, there are 2 important factors on which this clock rate depends. The first one is the implementation technology. So, by implementation technology what we mean is that with the advancement of technology, the size of transistors are becoming smaller and smaller. And with that the clock speed is becoming faster the clock is becoming faster basically.

So this is a factor on which the clock frequency depends. Another is the CPU organization. By CPU organization what we mean is that, suppose in a clock period we say, that some part of the instruction is executed. So, basically an instruction is divided into some cycles. I mean each of the work of that particular instruction is performed in those cycles. And by CPU organization we mean how we can organize the CPU such that, the number of tasks that can be performed within that clock period is maximized.

So, these are the 2 factors on which the clock rate actually depends. So, as I said just now that the machine instruction typically consists of number of elementary micro operations, that vary in number and complexity depending on the instruction, and the CPU organization used.

(Refer Slide Time: 09:32)



We will take an example here --- ADD R1, R2. To execute this instruction what we say that a machine instruction typically consists of number of elementary micro operations.

So to execute this particular instruction, we require certain steps. What are the steps? So, what we need to do first this particular instruction is stored in memory. You have to bring that from the memory. So, first is the fetch phase. Once you fetch, how will you fetch it? The content of PC should be made available. using some control signals. We will make the content of PC available, which is PCout we will see in details next. But in a simple word, I will say that once we do this PCout, the content of PC is irrelevant to some bus. And then we have to put this value in MAR memory address register.

So this we do: PCout, MARin, then we Read and then we do some other things. So, first we fetch the instruction. So, with just only these steps, we cannot fetch. We may require some more steps to fetch. Now what the point is one these particular steps can be performed, let us say in one clock cycle. And there can be many more steps to execute this machine instruction. What we require? Let us say we require 6 steps to execute this instruction. And we see that each of these steps require one clock cycle. So, this is what we mean by a machine instruction. Typically consists of number of elementary micro operations that vary in number and complexity depending on the instruction and the CPU organization used.

Now if you use a different CPU organization, these steps that I am saying might be different. Like say, for one type of organization, it requires 6 steps. For another type of organization, it may take 4 steps. So, we really cannot say that how you can differentiate. It depends on the organization used and it depends on the complexity of the instruction as well.

(Refer Slide Time: 12:54)

- A micro-operation is an elementary hardware operation that can be carried out in one clock cycle.
 - Register transfer operations, arithmetic and logic operations, etc.
- Thus a single machine instruction may take one or more CPU cycles to complete.
 - We can characterize an instruction by *Cycles Per Instruction* (CPI).
- Average CPI of a program:
 - Average CPI of all instructions executed in the program on a given processor.
 - Different instructions can have different CPIs.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, moving on, I said a micro-operation is an elementary hardware operation that can be carried out in one clock cycle. So, all the set of instruction as I said PCout, MARin, Read and so on can be executed in one clock cycle. And in one clock cycle what we can do basically is some register transfer instructions, some ALU operation instruction. Because all those are within the CPU and for that we do not have to bring it from the memory.

So whenever you have to bring it from the memory, we have to see that how much time will be required for that. CPU is much faster compared to memory. Transferring or getting a data from memory to CPU will take more time. Thus a single machine instruction may take one or more CPU cycles to complete. We can characterize an instruction by cycles per instruction. What do you mean by cycles per instruction? As I said an instruction is divided into some basic operations. Some micro-instructions are executed to execute that machine instruction. And all those those micro instruction that are getting executed requires some cycles.

So ultimately an instruction takes certain amount of cycles to execute it. So, that is called cycles per instructions. So, every instruction is taking some cycles to execute and that is termed as cycles per instruction. And what is this average CPI of a program? Average CPI of a program as we can say that see there can be many instructions and many instructions can have different CPIs. So, average CPI of all instruction executed in a program on a given processor. So, we average it; that means, some instruction, let us say takes 5 cycles, some instruction takes 7 cycles, some instruction takes 4 cycles. We average it out and then we say this is the average CPI. As different instructions can have different CPIs we will see in detail.

(Refer Slide Time: 15:38)

- For a given program compiled to run on a specific machine, we can define the following parameters:
 - a) The total number of instructions executed or *instruction count* (IC).
 - b) The average number of *cycles per instruction* (CPI).
 - c) *Clock cycle time* (C) of the machine.
- The total execution time can be computed as:

$$\text{Execution Time } XT = IC \times CPI \times C$$
- How do we evaluate and compare the performances of several machines?

IIT KHARAGPUR | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

So for a given program, compiled to run on a specific machine, we define the following. Parameters for a given program, now we are talking about the program that is compiled on a machine, these are the parameters that are important. What are the parameters? The total number of instructions executed, we call it instruction count. So, for a program what is the total number of instruction that is executed? The average number of cycles per instruction as I have already discussed that is the CPI. So, if there are 20 instructions and each requires some cycles. So, CPI is the total average number of cycles per instruction. And finally, the clock cycle time or the period of the machine.

So now what will be the total execution time? So, the total execution time can be computed as $XT = IC \times CPI \times C$. So, how do we evaluate and compare the performances

of several machines? Next we will see that for is the execution time. So, now, the execution time is the number of instructions multiplied by the CPI, multiplied by the clock cycle time. So, you multiply all these things you will get the execution time, we call it XT. Now we will see how we evaluate and come compare between the performances of several machines.

(Refer Slide Time: 17:57)

By Measuring the Execution Times

- One of the easiest methods to make the comparison.
- We measure the execution times of a program on two machines (A and B), as XT_A and XT_B .
- Performance can be defined as the reciprocal of execution time:

$$Perf_A = 1 / XT_A$$

$$Perf_B = 1 / XT_B$$
- We can estimate the speedup of machine A over machine B as:

$$Speedup = Perf_A / Perf_B = XT_B / XT_A$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So one of the easiest method that can be used to compare is like we measure the execution time of a program on two machines, that is A and B. And we see that execution time of A is XT_A and execution time of B is XT_B . So, the performance of A is $1/XT_A$ and performance of B is $1/XT_B$. So, let us say the one processor performs a task in 10 seconds and another processor performs the task in 2 seconds. Which one is better? The processor that performs a task faster means, less time is better. So, the processor that performs within 2 seconds will be better.

Now we can estimate the speedup of machine A over machine B as performance of A divided by performance of B. And performance of A is $1/XT_A$ and performance of B is $1/XT_B$. So, if you just put on these two values in this place, you will get the speed up of machine A over machine B. Now let us take some examples. Let us say a program is run on 3 different machines.

(Refer Slide Time: 19:43)

• **An example:**

A program is run on three different machines A, B and C and execution times of 10, 25 and 75 are noted.

- A is 2.5 times faster than B
- A is 7.5 times faster than C
- B is 3.0 times faster than C

• Simple for one program. But the main challenge is to extend the comparison when we have a set of programs.

- Shall be discussed later.

→

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, you have machines A, B and C. And the execution times are 10, 25 and 75. So, what you can see from this what we can say that A is 2.5 times faster than B.

So B is taking 25. So, we divide 25 by 10 and we get 2.5. So, we can say that A is 2.5 times faster than B. Similarly let us compare A and C. We can say A is 7.5 times faster than C. And similarly B is 3 times faster than C. So, B is 25. So, 75 divided by 25 we get 3. So, B is 3 times faster than C. This is simple for one program, but what if we have different set of programs, how do we compare, this shall be discussed in course of time.

(Refer Slide Time: 21:17)

Example 1

- A program is running on a machine with the following parameters:
 - Total number of instructions executed = 50,000,000
 - Average CPI for the program = 2.7
 - CPU clock rate = 2.0 GHz (i.e. $C = 0.5 \times 10^{-9}$ sec)
- Execution time of the program:
$$XT = IC \times CPI \times C$$
$$XT = 50,000,000 \times 2.7 \times 0.5 \times 10^{-9} = 0.0675 \text{ sec}$$

→

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now let us take an example. Say a program is running on a machine with the following parameters. So, what are the parameters, I give you the total number of instructions. So, what is your total number of instruction; this is your IC. So, what is the average CPI taking into account of different CPI is for these 50 million instructions, we get a CPI of 2.7. And what is the CPU clock rate? From frequency we can find out the time period by C will be your $1 / 2 \times 10^9 = 0.5 \times 10^{-9}$ second. And as we know that the execution time is IC x CPI x C. And all these values are provided here. We get XT as 0.0675 second.

(Refer Slide Time: 22:38)

	C	CPI	IC
Hardware Technology (VLSI)	X		
Hardware Technology (Organization)	X	X	
Instruction set architecture		X	X
Compiler technology		X	X
Program		X	X

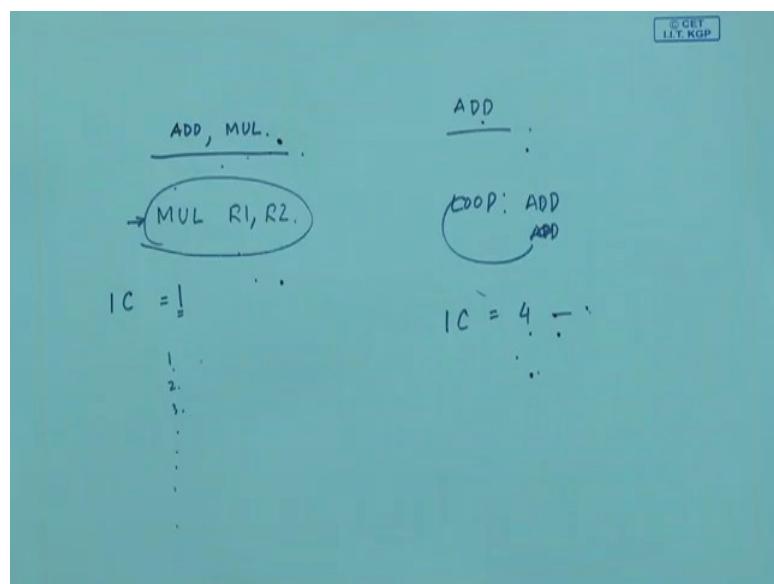
Now, see what are the factors that affect performance. First one is the hardware technology. Definitely if you make your clock cycle time smaller then it is pretty obvious, that you will be getting better result, but at the same time you have to cope up with the operations that can be performed in one clock period or one clock cycle. Next is the hardware technology, that is, the organization. So, what all factors that depends on this organization? First is the clock cycle time another is the cycles per instruction.

So let us say when we talk about organization, it depends on like a how your various hardwares are actually mapped. So, depending on that definitely your CPI that is cycles per instruction will vary you have an efficient organization, where is your cycles per instruction can be reduced. So, this particular factor that is hardware technology that is organization depends these are the factors that will govern, that is, your CPI as well as your clock cycle time. Now instruction set architecture, what all factors will depend it?

What is the instruction set architecture? We mean, that how various kinds of instruction you are putting it in your architecture. How many types how many varieties you are putting it there.

So, definitely if you restrict the number of instructions to a minimum level for a program you might require more number of instruction counts.

(Refer Slide Time: 25:13)



So, there are two things that can happen. Basically what I am trying to mean is that, let us say you have ADD and MUL instructions. And in one case you only have ADD instruction.

So in this particular architecture if you want to multiply two numbers you can simply use MUL R1,R2, but in the same case if you want to use repeated addition, then you have to use a loop. And that loop will perform that addition number of times. So, this particular ADD instruction may be used in that loop repeatedly. So, the idea is if you have more number of instructions, you might require less number of instructions in the end to execute a program. In a similar case you have less number of total instructions, but in that case you require more number of instructions to execute the same program. So, this is the difference.

So these instructions set architecture will affect two factors, one is CPI, another is instruction count. Now different compilers can generate different codes. Let us say for

the same program one compiler is taking 10 instructions, another compiler is taking 20 instructions. So this IC varies greatly on the compiler technology that is used. So, nowadays the compiler is becoming more and more intelligent and they are going hand in hand with the hardware. The compiler must know what kind of hardware architecture you are using such that it will generate the code in a fashion that will be easy for execution. And so such that it will also require less number of instructions and of course, what are the factors will depend both the CPI and the instruction count.

(Refer Slide Time: 27:34)

- IC depends on:
 - Program used, compiler, ISA
- CPI depends on:
 - Program used, compiler, ISA, CPU organization
- C depends on:
 - Technology used to implement the CPU
- Unfortunately, it is very difficult to change one parameter in complete isolation from the others.
 - Basic technologies are interdependent.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So as I discussed in the previous slide, IC depends on program used, the compiler, and the instruction set architecture. CPI also depends on program used, compiler, the instruction set architecture, as well as the CPU organization --- how you organize your CPU, how you organize your various hardware. So, final CPI will definitely depend on the CPU organization that you are using. And finally, C depends entirely on the technology used to implement the CPU. And this is very unfortunate that it is very difficult to change one parameter in complete isolation from the others. So, the basic technologies are very much interrelated to each other. So, we really cannot vary one parameter completely compared to the other.

(Refer Slide Time: 28:43)

• A tradeoff:

- **RISC**: increases number of instructions/program, but decreases CPI and clock cycle time because the instructions and hence the implementations are simple.
- **CISC**: decreases number of instructions/program, but increases CPI and clock cycle time because many instructions are more complex.

• Overall, it has been found through experimentation that RISC architecture gives better performance.

NPTEL ONLINE CERTIFICATION COURSES

IIT KHARAGPUR

NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So what is the tradeoff here? If you see a RISC machine, the number of instructions per program is more. So, increases the number of instructions per program, but at the same time it decreases the CPI and the clock cycle time. Because the instructions and hence the implementations are simple. But in CISC, decreases the number of instructions, but increases the CPI and the clock cycle time because many instructions are more complex. So, overall what has been found? It has been found that, RISC architecture gives better performance. So, let me tell you with the same example that we have taken earlier.

So we have MUL instruction and this is the instruction count, that is, IC is 1 here. And for this let us say the loop executes 4 times. So, in such cases depending on how many number 4 multiplied 5 times with 5 or 4 multiplies 10 times. So, it depends on that now IC is 1, but the number of cycles required to execute; that means, the micro operation that you are using steps in each steps we are performing something that might be more. So, CPI will be more. In this case may be IC is more, but overall it will take less number of cycles. So, CPI will be less here, but IC will be more. So, this is a tradeoff we can see.

So far in one case, we can have more IC where the CPI will be less; in some case we have less IC, but the CPI intern will be more.

(Refer Slide Time: 30:53)

Example 2

- Suppose that a machine A executes a program with an average CPI of 2.3. Consider another machine B (with the same instruction set and a better compiler) that executes the same program with 20% less instructions and with a CPI of 1.7 at 1.2 GHz. What should be the clock rate of A so that the two machines have the same performance?

We must have: $IC_A \times CPI_A \times C_A = IC_B \times CPI_B \times C_B$
Hence: $IC_A \times 2.3 \times C_A = 0.80 \times IC_A \times 1.7 \times (1 / (1.2 \times 10^9))$
We get: $C_A = 0.49 \times 10^{-9} \text{ sec}$
Thus, clock rate of A = $1 / C_A = 2.04 \text{ GHz}$



11

Let us take an example. Suppose that a machine A executes a program with an average CPI of 2.3; consider another machine B with the same instruction set and a better compiler that execute the same program with 20% less instructions. And with the CPI of 1.7 at 1.2 GHz what should be the clock rate of A so that the two machines have the same performance. So, for both the machines CPI is given. And one machine executes the same program with 10% less instruction.

So one takes 100% and another takes 20% less; that means, 80%. So, for A IC_A will remain same; this is 2.3; and this is C_A , that is what we have to find it out. Clock of A and for this IC is 20% less. So, it will become 80%. So, $0.80 \times IC \times 1.7 \times$ this is the clock rate this is the period. So, the clock rate is 1.2 GHz, that is, 1.2×10^9 ; 1 divided by that will give you the clock period of this. You will get a clock period of 0.49×10^{-9} second that is coming to 2.04 GHz.

(Refer Slide Time: 32:35)

Example 3

- Consider the earlier example with IC = 50,000,000; average CPI = 2.7, and clock rate = 2.0 GHz.

Suppose we use a new compiler on the same program, for which:

- New IC = 40,000,000
- New CPI = 3.0 (i.e. the new compiler is using more complex instructions)
- Also we have a faster CPU implementation, with clock rate = 2.4 GHz.

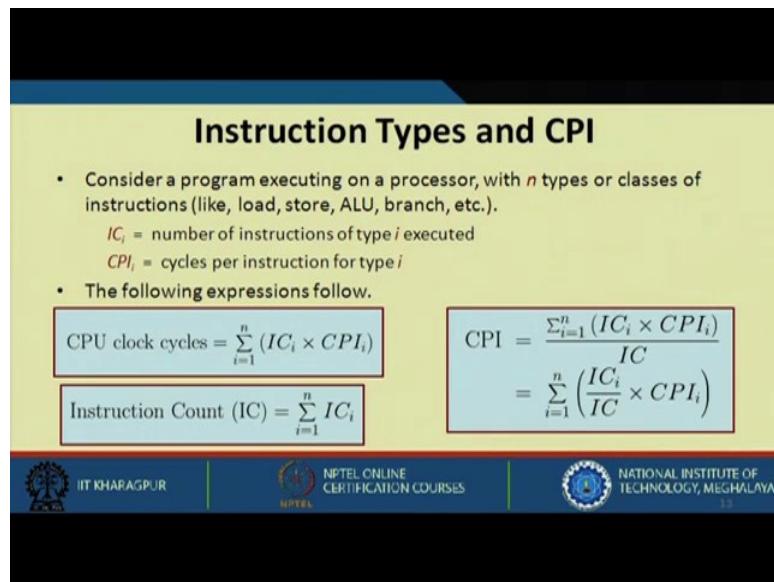
$$\begin{aligned} \text{Speedup} &= XT_{\text{old}} / XT_{\text{new}} \\ &= (50,000,000 \times 2.7 \times 0.5 \times 10^{-9}) / (40,000,000 \times 3.0 \times 0.4167 \times 10^{-9}) \\ &= 1.35 \rightarrow 35\% \text{ faster} \end{aligned}$$

IIT Kharagpur | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

So we need 2.04 GHz clock for a machine such that both the result should be same. Let us take another example where consider the earlier example with the instruction count of 50 million. Average CPI of 2.7 and clock rate of 2 GHz. Suppose we use a new compiler on the same program for which new IC is 40 million and the new CPI has also increased to 3. Also we have a faster CPU implementation with clock of clock rate of 2.4 GHz. So, these are the different. So, one is having 2 GHz, this is having 2.4, but the CPI of this is less CPI of this is more, but the instruction count of this is even more and this is less.

So if you compare this what will be the speedup? You have to find the execution time of old one compared to execution time of new. So, you just put those values you get the execution time old that is of this one. And you put all these values you get the execution time of this one, which is coming down to 1.35. So, we can say that it is 35% faster.

(Refer Slide Time: 33:48)



The slide is titled "Instruction Types and CPI". It contains the following text and equations:

- Consider a program executing on a processor, with n types or classes of instructions (like, load, store, ALU, branch, etc.).
 IC_i = number of instructions of type i executed
 CPI_i = cycles per instruction for type i
- The following expressions follow.

$$\text{CPU clock cycles} = \sum_{i=1}^n (IC_i \times CPI_i)$$
$$\text{Instruction Count (IC)} = \sum_{i=1}^n IC_i$$
$$\begin{aligned} \text{CPI} &= \frac{\sum_{i=1}^n (IC_i \times CPI_i)}{IC} \\ &= \sum_{i=1}^n \left(\frac{IC_i}{IC} \times CPI_i \right) \end{aligned}$$

Logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya are at the bottom.

Let us take another thing that is instruction types and CPI. Consider a program executing on a processor with n types or classes of instructions. So, generally we do not have one kind of instructions. We have load-store instruction; we have data transfer instruction within the CPU; we have variety of instructions basically.

So these classes are divided into let us say load-store, ALU, branch, etc. So, IC_i is number of instructions of type i executed, CPI_i is cycles per instruction of type i , the following expression follows from this. So, till now we were saying that this is the total IC, this is the total CPI now we divide it we say that there are various kinds of instructions. So, we can have various kinds of instructions and each of these instructions can have different CPIs. So, each instruction will have different CPI. So, CPU clock cycles will be $IC_i \times CPI_i$ summation of that. Similarly, instruction count will be considering all the instruction of all the types IC of type i , where i go from 1 to n .

So there are n type of instructions and what will be CPI now? CPI will be summation of instruction count and cycles per instruction divided by instruction count total instruction count. So, summation of IC_i divided by IC into CPI.

(Refer Slide Time: 35:51)

The slide is titled "Example 4". It contains a bulleted list and a table. The list states: "Consider an implementation of a ISA where the instructions can be classified into four types, with CPI values of 1, 2, 3 and 4 respectively." Below this, it says "Two code sequences have the following instruction counts:"

Code Sequence	IC _{Type1}	IC _{Type2}	IC _{Type3}	IC _{Type4}
CS-1	20	15	5	2
CS-2	10	12	10	4

Below the table, there are two boxes: one for CS-1 with the formula "CPU cycles for CS-1: 20x1 + 15x2 + 5x3 + 2x4 = 73" and CPI "CPI for CS-1: 73 / 42 = 1.74"; another for CS-2 with the formula "CPU cycles for CS-2: 10x1 + 12x2 + 10x3 + 4x4 = 80" and CPI "CPI for CS-2: 80 / 36 = 2.22".

Logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya are at the bottom.

Let us take one more example where we consider the implementation of an instruction set architecture, where the instructions can be classified into four types. So, the CPI values of these 4 types of instructions are 1, 2, 3 and 4 respectively. Two code sequences have the following instruction counts; that means, there are 2 code sequence and these are the various instruction count of type one instruction count of type 2 and so on.

So now you see the CPU cycles for CS-1: 20×1 because for type 1 the CPI is 1, type 2 multiplied by 2, 15 multiplied by 2, 5 multiplied by 3 and 2 multiplied by 4. So, this will give you the total CPU cycles that is 73. So, what will be the cycles per instruction, total instruction $20 + 15 + 5 + 2$, which is coming down to 42 total instructions. So, CPI will be CPU cycle divided by 42, which is coming down to 1.74. Similarly, for the next one, a total CPU cycle is 80 and CPI is 2.22. So, we can see that it greatly depends on both the type of instruction and what is the CPI of that type of instruction. So, both varies.

(Refer Slide Time: 37:40)

Instruction Frequency and CPI

- CPI can also be expressed in terms of the frequencies of the various instruction types that are executed in a program.
 - F_i denotes the frequency of execution of instruction type i .

$$\text{CPI} = \frac{\sum_{i=1}^n (IC_i \times CPI_i)}{IC}$$
$$= \sum_{i=1}^n \left(\frac{IC_i}{IC} \times CPI_i \right)$$
$$F_i = \frac{IC_i}{IC}$$
$$\text{CPI} = \sum_{i=1}^n (F_i \times CPI_i)$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So this is instruction frequency and CPI, where CPI can be expressed in terms of frequencies of various instruction types that are executed in a program. So, F_i denotes the frequency of execution of type i . So, F_i is IC_i divided by total instruction count and CPI we have already shown it in the previous slide can be expressed in terms of frequency. So, we substitute this here. So, we get frequency multiplied by CPI_i .

(Refer Slide Time: 38:15)

Example 5

- Suppose for an implementation of a RISC ISA there are four instruction types, with their frequency of occurrence (for a typical mix of programs) and CPI as shown in the table below.

Type	Frequency	CPI
Load	20 %	4
Store	8 %	3
ALU	60 %	1
Branch	12 %	2

$$\text{CPI} = \sum_{i=1}^n (F_i \times CPI_i)$$
$$\text{CPI} = (0.20 \times 4) + (0.08 \times 3) + (0.60 \times 1) + (0.12 \times 2)$$
$$= 1.88$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us take another example. Where suppose for an implementation of a RISC, ISA there are 4 instruction types with their frequency of occurrence for typical mix of programs let us say and CPI as shown in table below.

So this is the frequency at which load instruction is executed. This is the frequency at which store instruction is executed and this is the frequency for ALU one branch. So, and the CPI is given cycles per instruction this is the frequency at which it is happening. So, what is the frequency that is 0.2 and the CPI is 4. So, if you want to find CPI, you can find F_i multiplied by CPI_i. So, 20 percent 0.20 multiplied by 4, 8 percent 0.08 multiplied by 3 and so on. And we get 1.88.

(Refer Slide Time: 39:15)

Example 6

- Suppose that a program is running on a machine with the following instruction types, CPI values, and the frequencies of occurrence.

The CPU designer gives two options: (a) reduce CPI of instruction type A to 1.1, and (b) reduce CPI of instruction type B to 1.6. Which one is better?

Type	CPI	Frequency
A	1.3	60 %
B	2.2	10 %
C	2.0	30 %

Average CPI for (a): $0.60 \times 1.1 + 0.10 \times 2.2 + 0.30 \times 2.0 = 1.48$

Average CPI for (b): $0.60 \times 1.3 + 0.10 \times 1.6 + 0.30 \times 2.0 = 1.54$

Option (a) is better

IIT KHARAGPUR
NPTEL ONLINE
CERTIFICATION COURSES
NPTEL
NATIONAL INSTITUTE OF
TECHNOLOGY, MEGHALAYA

Let us take another example suppose that a program is running on a machine with the following instruction types. CPI values and frequency of occurrence the CPU designer gives 2 options, the first option is to reduce the CPI instruction of type A to 1.1 %. So, type A instruction we are reducing to 1.1 %. And reduce CPI instruction of type B to 1.5, CPI of type B is reduced to 1.6. So, this is type A is reduced from 1.3 to 1.1 and type B is reduced from 2.2 to 1.6. Now let us see.

So average CPI for a will be 60.60 %. So, 0.60 multiplied with 1.1 this is the new CPI and all the rest CPI remains the same. So, we get 1.4448 similarly the CPI for B is 0.60 into no change here, but here we change to 1.6 this. 2.2 becomes 1.6 and this remains same. So, we get this. So, from this what we clearly can say is that option A is better, but

you see what we have done option A is we have reduced from 1.3 to 1.1, but you see the frequency of this instruction that is getting executed that is 60 %. So, it is much more.

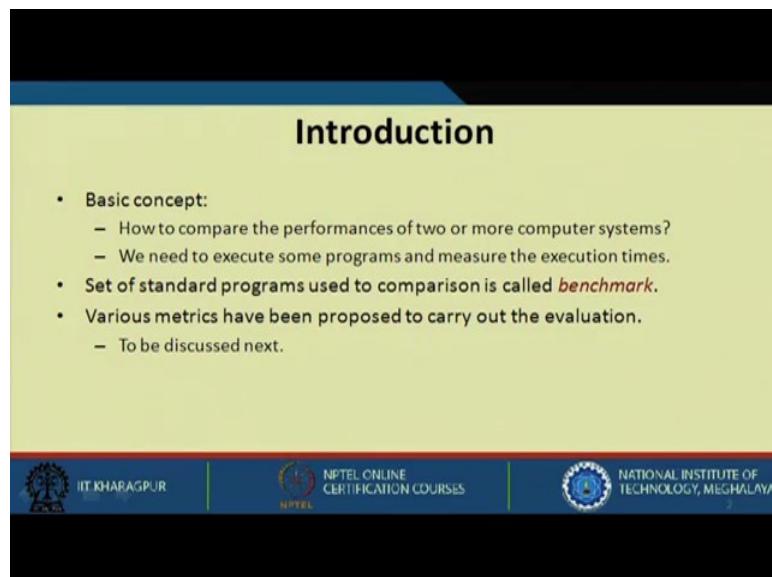
So you must take into account the frequency. Some instructions that are frequently getting executed and you reduce the CPU even less amount, but you are using that particular instruction much more. So, in that case you will get a better result even if you are reducing certain CPI to a great extent, but that is not executed more. So, you see that type B is executing only the frequency of execution is 10%. So, in that case if you reduce it to 1.6, also you are not getting a better result compared to when you are reducing A to just 1.1. So, we came to the end of lecture 12 where we have seen that various things that affect a CPU performance. And next we will see in some more detail in the next lecture.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture – 13
Choice of Benchmarks

Welcome to the next lecture. So, in this lecture we will be seeing the choice of benchmark. So, in the previous lecture what we have said is that how do we compute the execution time of a program. And then we know that these are the ways through which you execute you can compute the execution time of the program. Now still how you can say that this computer is better than this? Now comes choice of benchmark; that means, what you will be using which benchmarks to use such that by executing this set of instruction we can compare performance.

(Refer Slide Time: 01:10)



Introduction

- **Basic concept:**
 - How to compare the performances of two or more computer systems?
 - We need to execute some programs and measure the execution times.
- **Set of standard programs used to comparison is called *benchmark*.**
- **Various metrics have been proposed to carry out the evaluation.**
 - To be discussed next.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, basically here the basic concept is how to compare the performances of two or more computers. This is what we are discussing from the last lecture as well. So, we need to execute some programs and measure the execution times. Set of standard programs are used for this for this comparison and we call this as benchmarks. So, benchmarks are nothing but some set of programs, which are set as benchmarks basically for this comparison. And various metrics have been proposed to carry out the evaluation we will be discussing that next.

(Refer Slide Time: 01:54)

Some Early Metrics Used

a) MIPS (Million Instructions Per Second)

- Computed as $(IC / XT) \times 10^{-6}$
- Dependent on instruction set, making it difficult to compare MIPS of computers with different instruction sets.
- MIPS varies between programs running on the same processor.
- Higher MIPS rating may not mean better performance.
- Example: A machine with optional floating-point co-processor.
 - When co-processor is used, overall execution time is less but more complex instructions are executed (i.e. smaller MIPS).
 - Software routines take more time but gives higher MIPS value → **FALLACY**.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

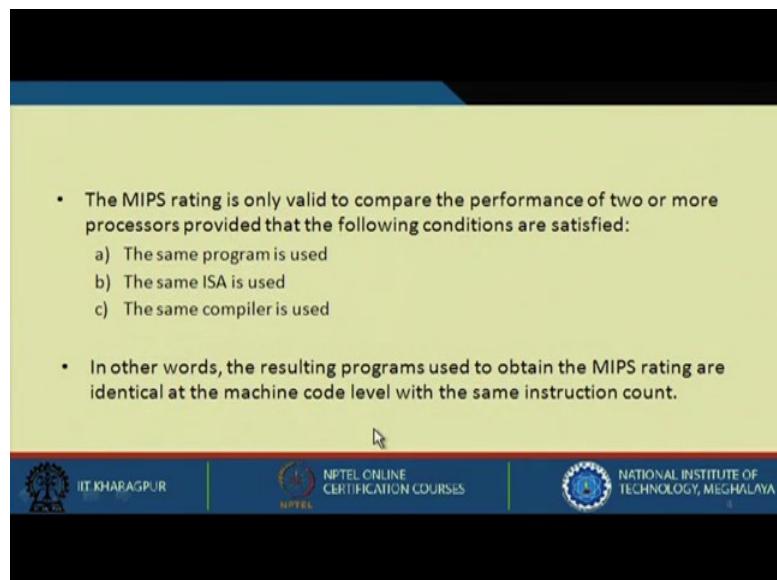
So, let us see some early metrics that are used. One is millions instructions per second (MIPS); this is computed as instruction count divided by execution time into 10^{-6} . So, this is dependent on what. So, basically we are trying to get how many millions of instructions that are executed per second. This is dependent on instruction set, which makes it difficult to compare MIPS of various computers with different instruction set, because see different computers will have different instruction set different kind of architectures, but then how we can say that this can be a metric that can be used to evaluate for all. So, this becomes difficult.

So, this is dependent on the instruction set. MIPS also varies between programs running on the same processor; why does this varies? Because different compilers will generate different codes; suppose say you have run a particular program and it has generated some set of a codes; the same program can be run on another compiler and may generate little different code. So, at that point of the time also this MIPS will be different and also it has been observed that higher MIPS rating may not mean better performance. So, we cannot say that if the MIPS rating is high, that means it performs much better. Let us take an example a machine with optional floating-point coprocessor.

So, when aco processor, meaning is we have a machine with an optional floating-point coprocessor. So, when coprocessor is used overall execution time will be less because you are using a coprocessor in which the task will can be performed in a much faster

fashion. So, in turn your execution time will become less, but for doing you may use some complex instructions. So, if you use complex instructions then it will give you a smaller MIPS value. So, when you use a coprocessor your overall execution time becomes less, but you are using more complex instruction for execution. That is the MIPS will be much less. Same way for a software routine it takes more time, but it is giving higher MIPS value why because they will be more number of instruction that are getting executed, but in turn the time using a software routine will be much more. So, this is a fallacy this is a problem here. So, we are using a coprocessor which is making the entire process faster, but still we are getting smaller MIPS, but another which is using a software routine which is getting higher MIPS, but at the same time it takes more time as well.

(Refer Slide Time: 05:38)



So, MIPS rating is only valid to compare the performance of 2 or more processors provided that the following conditions are satisfied. What are the factors? First one is the same program is used. The same instruction set architectures, used set of instruction should be same and the same compiler is used. If you have all these things together, then only we can say that MIPS rating can be taken for performance comparison.

So, in other words we can say that the resulting programs used to obtain the MIPS rating are identical at the machine code level with the same instruction count. You must have

same instruction count you must have same those machine code level instructions, then only you can say that you can use MIPS as a metric to evaluate the performance.

(Refer Slide Time: 06:53)

b) **MFLOPS (Million Floating Point Operations Per Second)**

- Simply computes number of floating-point operations executed per second.
- More suitable for applications that involve lot of floating-point computations.
- Here again, different machines implement different floating-point operations.
- Different floating-point operations take different times.
 - Division is much slower than addition.
- Compilers have no floating-point operations and has a MFLOP rating of 0.
- Hence, not very suitable to use this metric across machines and also across programs.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

This is million floating point operations per second (MFLOPS). So, it simply computes the number of floating point operations executed per second. Now this obviously, will be more suitable for certain applications where we will be using floating point computation. Let us say for certain application there are not so much floating point instructions.

So, in that MFLOPS will be much less, but that does not mean that the performance of that is poor. So, here again different machines implement different floating point operations. Different floating point operation takes different times. Addition of a floating point will take less time may be compared to the division of a floating point.

So, we cannot really say that how well MFLOPS as a metric will give you a correct performance evaluation. Compilers have no floating point operation and has MFLOPS rating as 0. Hence this is not very suitable metric across machine and across programs. MFLOPS cannot be used as a metric across any machines or across any programs because different machines have different features different characteristics. So, it might not be a good idea to rely upon a metric like MFLOP.

(Refer Slide Time: 08:53)

Example 1

- Consider a processor with three instruction classes A, B and C, with the corresponding CPI values being 1, 2 and 3 respectively. The processor runs at a clock rate of 1 GHz.

For a given program written in C, two compilers produce the following executed instruction counts.

	Instruction Count (in millions)		
	For IC _A	For IC _B	For IC _C
Compiler 1	7	2	1
Compiler 2	12	1	1

Compute the MIPS rating and the CPU time for the two program versions.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us take an example. Consider a processor with three instruction classes A, B and C with the corresponding CPI values being 1, 2 and 3 respectively. The processor runs at a clock rate of 1 GHz. So, for a given program written in C, two compilers produce the following executed instruction counts.

So, instruction count for A type instruction is 7 for compiler 1, 2 for type B, and 1 for C type C classes. Similarly, for compiler 2 the number of instruction count for A type is 12, B type is 1 and C type is 1. Let us see how do we compute the MIPS rating and the CPU time for the two program versions. So, we have been given with the CPI values for the various types of instruction that is A, B and C as 1, 2 and 3 respectively and the processor runs at a clock rate of 1 GHz. So, these are the parameters that are already given.

(Refer Slide Time: 10:21)

MIPS = Clock Rate (MHz) / CPI

CPI = CPU Execution Cycles / Instruction Count

CPU Time = Instruction Count x CPI / Clock Rate

- Solution:
 - For compiler 1:
$$\text{CPI}_1 = (7 \times 1 + 2 \times 2 + 1 \times 3) / (7 + 2 + 1) = 14 / 10 = 1.40$$
$$\text{MIPS Rating}_1 = 1000 \text{ MHz} / 1.40 = 714.3 \text{ MIPS}$$
$$\text{CPU Time}_1 = ((7 + 2 + 1) \times 10^9 \times 1.40) / (1 \times 10^9) = 0.014 \text{ sec}$$
 - For compiler 2:
$$\text{CPI}_2 = (12 \times 1 + 1 \times 2 + 1 \times 3) / (12 + 1 + 1) = 17 / 14 = 1.21$$
$$\text{MIPS Rating}_2 = 1000 \text{ MHz} / 1.21 = 826.4 \text{ MIPS}$$
$$\text{CPU Time}_2 = ((12 + 1 + 1) \times 10^9 \times 1.21) / (1 \times 10^9) = 0.017 \text{ sec}$$

MIPS rating indicates that compiler 2 is faster, while in reality the reverse is true.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Let us see how we will calculate the MIPS rating and the CPU time. So, MIPS is clock rate in MHz divided by CPI. And CPI is CPU execution cycle divided by instruction count. And CPU time will be instruction count multiplied by CPI divided by clock rate or multiplied by clock period. Let us say for compiler 1, 7 is the total number of instruction type A which is multiplied with 1 that is the CPI for that particular type A instruction. Similarly, 2 multiplied by 2. So, basically we are doing 7 multiplied by 1, 2 multiplied by 2, and 3 multiplied by 1 one for compiler 1. So, 7 multiplied by 1, 2 multiplied by 2, and 1 multiplied by 3 divided by total number of instruction. Total number of instruction was $7 + 2 + 1 = 10$. So, $14 / 10$ which is coming to 1.40.

Similarly, MIPS rating will be 1000 MHz divided by 1.40. So, we converted it to MHz because we have to find out in terms of MIPS. So, that is 1000 divided by 1.40 that is coming to 714.3. Now what is the CPU time? The CPU time can be calculated by $7 + 2 + 1$. So, we get the time as 0.014 seconds. So, this much second it is taking to execute for compiler 1 and MIPS rating for this is 714.3.

Let us take for the next compiler in the similar fashion we compute the CPI 12 into 1, 1 x 2 + x 3 divided by total number of instruction. And we get 1.21 as the CPI similarly MIPS rating can be find out by 1000 MHz divided by 1.21, that comes to 826.4 MIPS. And similarly for CPU time we will use the same instruction count multiplied by CPI divided by clock rate, which is coming down to 0.017.

So, now you see that the MIPS of this is higher. So, it has got higher millions instruction per second, but the execution time of compiler 1 is less. So, here you can clearly see that the execution time of compiler 1 is less, but the MIPS of compiler 2 is more. So, MIPS cannot be the right choice for in such cases. So, MIPS rating indicates that compile it 2 is faster while in reality the reverse is true.

(Refer Slide Time: 13:57)

Example 2

A loop in C

```
for (k=0; k<1000; k++)
{
    A[k] = A[k] + s;
}
```

MIPS32 Code

```
Loop:    LW      $t3, 0($t1)
          ADDI   $t6, $t2, 4000
          LW      $t4, 0($t3)
          ADD    $t5, $t4, $t3
          SW      $t5, 0($t2)
          ADDI   $t2, $t2, 4
          BNE    $t6, $t2, Loop
```

- \$t1 = address of s
- \$t3 = s
- \$t2 points to A[0]

- The code is executed on a processor that runs at 1 GHz (C = 1 nsec).
- There are four instruction types with CPI values as shown in the table.
- We show some calculations next.

Instruction Type	CPI
ALU	2
LOAD	5
STORE	6
BRANCH	3

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us take an example. So, this is a C loop. What we are doing inside the C loop, we are simply adding a constant value stored in variable s to A[k], and we are storing back in A[k].

Similarly, for the next one and this is going on in a loop let us write the assembly language code for this particular C code segment. So, these are the few things you have to consider. \$t1 stores the address of s, s is a variable which is a constant some value is stored here, and \$t3 stores the value of s and \$t2 points to the first location of this array of this particular array. So, here initially what we are doing we are loading the word from this location 0 of \$t1, \$t1 is having the address of s. So, value of s is stored in \$t3. We are adding an immediate value to \$t2, \$t2 points to the first location of the array. And we have to compute something for this thousand times. So, we are multiplying 4000; we are adding 4000 to \$t2 because each are 4, 4, 4 by, so, 4 multiplied by 1000. So, which is coming down to we are adding it to \$t2 and we are storing it in \$t6. So, \$t6 stores the final value. So, we have to go till that value to execute it.

Next word inside the loop these are the following statement, that are getting executed first what we are doing we are loading the word from the first location of the array that is A[0]. We are storing it in \$t4. Then what we are doing the value of s is stored in \$t3 and the array value is stored in \$t4. So, \$t3 and \$t4 we have to add and we are storing it in \$t5. So, finally, we are adding \$t4 and \$t3 and we are storing it in \$t5 and finally, we are storing back this \$t5 the added value in 0 of \$t2. So, in that location we are again storing it back. And finally, what we have to do we need to increment to the next location. So, the first part is done now we are moving to the next location. So, for the next location it is added with 4 again and then it is transferred there.

Now, branch if not equal we are doing such that whether we have reached to that point or not, \$t6 is equal to \$t2 because at every point we are adding 4 to it. So, when it reaches the last element it will come out of the loop, when till it is not equal \$t6 is not equal to \$t2 is not equal to \$t6 we will loop, when it is equal it will come out of the loop. So, these are the following assembly language code that we are executing for this set of codes the code is executed on a processor that runs at 1 GHz that is the clock period is one nanosecond, there are 4 instruction types with CPI values are shown in this table.

Now, see ALU operation which are ADDI; these are ALU operation. And the CPI of those operations is 2. Similarly, you have load. Load is LW the CPI is 5, you have SW the CPI is 6, and you have a BNE type of instruction where the CPI is 3.

(Refer Slide Time: 18:49)

- The code has 2 instructions before the loop and 5 instructions in the body of the loop that executes 1000 times.
 - Total instruction count $IC = 5 \times 1000 + 2 = 5002$.
- Number of instructions executed and fraction F_i for each instruction type:
 - $IC_{ALU} = 1 + 2 \times 1000 = 2001, F_{ALU} = 2001 / 5002 = 0.4 = 40\%$
 - $IC_{LOAD} = 1 + 1 \times 1000 = 1001, F_{LOAD} = 1001 / 5002 = 0.2 = 20\%$
 - $IC_{STORE} = 1000, F_{STORE} = 1000 / 5002 = 0.2 = 20\%$
 - $IC_{BRANCH} = 1000, F_{BRANCH} = 1000 / 5002 = 0.0 = 20\%$
- Total CPU clock cycles = $2001 \times 2 + 1001 \times 5 + 1000 \times 6 + 1000 \times 3 = 18,007$ cycles
- Average CPI = $CPU \text{ clock cycles} / IC = 18007 / 5002 = 3.6$
- Execution time = $IC \times CPI \times C = 5002 \times 3.6 \times 1 \times 10^{-9} = 18.0 \mu\text{sec}$





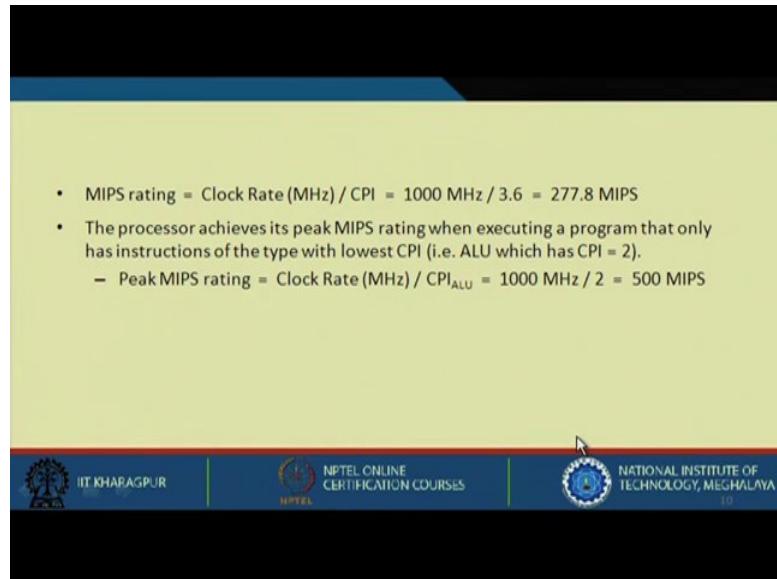
Now, let us see the code has two instructions before the loop and 5 instructions in the body of the loop that executes 1000 times. So, outside the body of the loop you see you have two instructions. And inside you have 5 instructions 1 2 3 4 5 and each of these instructions is executed 1000 times because this loop is executing 1000 times. So, what will be the total instruction count? There are 5 instructions and each instruction executes 1000 time. So, 5000 and 2 instructions outside the loop plus 2 it will become 5000. 2 number of instructions executed and fraction F_i for each instruction type. So, let us calculate this number of instruction executed and fraction of instruction F_i for each instruction type.

Let us first calculate the total number of instruction. So, outside the loop there is one ALU instruction and inside the loop there are 2 ALU instructions. So, these 2 ALU instructions each will be executed 1000 times. And this instruction will get executed one time. So, inside the loop there are 2 ALU instructions that are executed 1000 times, and this is executed one more time. So, 2001, similarly you can calculate for all load store and branch store and branch are only 1, 1 instructions are there this is store and this is from this is executed thousand time this is executed 1000 times.

So, what is the frequency --- total number of instruction of such kind divided by the total number of instructions, which is coming to 0.4, that is 40%. This is coming 20%, this is coming 20%, and this is coming 20%. Now how do we calculate this is the frequency of a loop operation, this is the frequency of load type and so on. So, total CPU clock cycles are 2001×2 , 1001×5 , 1000×6 , 1000×3 . So, we are taking all this from this CPI we are multiplying the CPI with the total number of instruction that we have found out previously and we are getting the total cycles as 18007. So, this is the total CPU clock cycle divided by instruction count you get the CPU as 3.6.

Now, you can calculate the total execution time which is IC which is total instruction which is 5002 CPI that we have calculated, and this is the clock period which is coming to 80 microsecond this is how we calculate it.

(Refer Slide Time: 22:03)



Now, how do you see the clock rating? Clock rating will be clock rate divided by CPU that is coming to 277.8 MIPS. So, the processor achieved its peak MIPS rating when executing a program that only has instructions of type with lower CPU CPI that is ALU type instruction. So, if you only execute such kind of instruction where the CPI is less than 2, then you see the CPI of ALU type is only 2, but for store load branch is moved.

Now, if you only execute ALU, which only those type of instruction where CPI is less than 2, then you can get the peak MIPS rating that is coming to 500 MIPS, but if you use with this mixture where the CPI is found out by calculating taking into consideration all the types of instruction and the frequency at which all these instructions are occurring then it will be coming to something lesser MIPS.

(Refer Slide Time: 23:16)

Choosing Programs for Benchmarking

- Suppose you are trying to buy a new computer and you have several alternatives.
 - How to decide which one will be best for you?
- The best way to evaluate is to run the actual applications that you are expected to run (*actual target workload*), and find out which computer runs them the fastest.
 - Not possible for everyone to do this.
 - We often rely on other methods that are *standardized* to give us a good measure of performance.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, next let us see choosing programs for benchmarking. Now how do we choose programs for this benchmarking? Suppose we are trying to buy a new computer and there are several alternatives possible. So, how to decide upon which one is the best. The best way that is that can be used is to run the actual application that you are expected to run that is the actual target workload; that means, that particular computer you will be using more floating point operation. So, you should have such kind of program in place that you will run on that machine and you will see that what is the performance coming.

So, actually you are running certain kind applications that will give you the best result. So, choosing the programs for benchmarking is really very important, but not possible for everyone to do this while purchasing. So, what we do we often rely on the methods that are standardized to give us a good measure of performance. So, there are some standardized methods that are used which can be considered as a good measure for this performance.

(Refer Slide Time: 24:30)

- Different levels of programs used for benchmarking:
 - a) Real applications
 - b) Kernel benchmarks
 - c) Toy benchmarks
 - d) Synthetic benchmarks

At the bottom, there are logos for IIT Kharagpur, NPTEL, and National Institute of Technology, Meghalaya.

So, different levels of programs are used for benchmarking --- one is real application, can be kernel benchmarks, some toy benchmarks, and some synthetic benchmarks. Let us see an overview of all these things.

(Refer Slide Time: 24:45)

(a) Real Applications

- Here we select a specific mix or suite of programs that are typical of target applications or workload (e.g. SPEC95, SPEC CPU2000, etc.).
- **SPEC (System Performance Evaluation Corporation)** is the most popular and industry-standard set of CPU benchmarks.
- Examples:
 - SPECint95 consists of 8 integer programs.
 - SPECfp95 consists of 10 floating-point intensive programs.
 - SPEC CPU2000 consists of 12 integer programs (CINT2000) and 14 floating-point intensive programs (CFP2000).
 - SPEC CPU2006 consists of 12 integer programs (CINT2006) and 17 floating-point intensive programs (CFP2006).

At the bottom, there are logos for IIT Kharagpur, NPTEL, and National Institute of Technology, Meghalaya.

What are real applications? We select a specific mix of suit of programs that are typical of large application or workload. Some of the examples are SPEC95, CPU2000, etc. SPEC stands for System Performance Evaluation Corporation and this is the most popular and industry standard set of CPU benchmarks. So, SPECint95 consists of 8

integer programs. SPECfp95 consists of 10 floating-point intensive programs. SPEC CPU2000 consists of 12 integer programs and 14 floating-point intensive programs, and SPEC CPU2000 consists of 12 integer programs.

So, as we are moving from 95 to 2000 to 2006 the numbers are increasing. So, we are putting more workload because their advancement in these clock speed is increasing. So, we can actually perform more operations. So, that is how it is moving.

(Refer Slide Time: 26:12)

SPEC95 Programs (Integer)	
Benchmark	Description
go	A game based on artificial intelligence
m88ksim	A simulator for Motorola 88k chip
gcc	Gnu C compiler to generate SPARC code
compress	Compression and decompression utility
lisp	LISP interpreter
jpeg	Image compression and decompression utility
perl	PERL interpreter
vortex	A database program



The slide features three logos at the bottom: IIT Kharagpur (with a tree emblem), NPTEL Online Certification Courses (with a globe icon), and National Institute of Technology, Meghalaya (with a circular emblem).

So, these are SPEC95 programs (integer) --- what all kind of programs are present; a game based on artificial intelligence, a simulator for motorola 88k chip, a gnu compiler, compression and decompression utility, lisp interpreter, image compression and decompression utility, perl interpreter, a database program. So, the SPEC95 program consists of these following benchmarks.

(Refer Slide Time: 26:47)

SPEC95 Programs (Floating-Point)		Benchmark	Description
	tomcatv		A mesh generation program
	swim		Shallow water modeling
	su2cor		Quantum physics Monte Carlo simulation
	hydro2d		Solving hydrodynamic Naiver Stokes equations
	mgrid		Multigrid solver on 3D potential field
	applu		Solving parabolic/elliptical differential equations
	trub3d		Simulates turbulence in a cube
	apsi		Solver for distribution of pollutant
	fppp		Quantum chemistry simulation
	wave5		Simulation of plasma physics

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Similarly, SPEC95 programs consist of these programs. And these are all these SPEC95 programs of floating point programs. So, they have a mesh generation program, shallow water modeling, quantum physics, Monte Carlo simulation, solving hydrodynamic naiver stokes equation, multi grid solver on 3D potential field, quantum chemistry simulation and so on. So, these SPEC95 programs were having these programs.

(Refer Slide Time: 27:32)

CINT2000 (Integer)	
1. 164.gzip : compression	9. 254.gap : Group theory interpreter
2. 175.vpr : FPGA placement / routing	10. 255.vortex : Object-oriented database
3. 176.gcc : C compiler	11. 256.bzip2 : Compression
4. 181.mcf : Combinatorial optimization	12. 300.twolf : VLSI Place / Route
5. 186.crafty : Chess playing	
6. 197.parser : Word processing	
7. 252.con : Computer visualization	
8. 253.perlbench : PERL interpreter	

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Similarly, CINT2000 (integer) consists of these following programs. So, these are the 12 programs. They added something new which is VLSI place and route. This group theory interpreter was also added which was not there previously in SPEC95.

(Refer Slide Time: 27:55)

CFP2000 (Floating-Point)

1. 168.wupwise : Quantum dynamics	9. 187.facerec : Face recognition
2. 171.swim : Shallow water modeling	10. 188.ammp : Computational chemistry
3. 172.mgrid : Multi-grid solver	11. 189.lucas : Primality testing
4. 173.applu : Differential equation solver	12. 191.fma3d : Finite-element simulation
5. 177.mesa : 3D graphics library	13. 200.sixtrack : Nuclear accelerator
6. 178.galgel : Fluid dynamics	14. 301.apsi : Pollutant distribution
7. 179.art : Neural networks	
8. 183.eqquake : Seismic wave simulation	

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, these are some of the programs of CFP2000. So, it has got quantum dynamics these are already there neural networks were added pollutant distribution is added nuclear accelerator is added and many more.

(Refer Slide Time: 28:16)

(b) Kernel Benchmarks

- Here key computationally-intensive pieces of code are extracted from real programs (e.g. Fast Fourier transform, matrix factorization, etc.).
- Unlike real programs, no user would be running the kernel benchmarks.
 - They are used solely to evaluate performance.
- Kernels are best used to isolate performance of specific features of a machine and evaluate them.
- Examples: Livermore Loops, Linpack.
 - Some compilers were reported to have been using benchmark specific optimizations so as to give the machine a good rating.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Now, let us see what kernel benchmark is. Here what happens basically is that key computationally intensive pieces of code are extracted from real programs. So, let us say there is part of the program, where the computation requirement is moved. So, they take out those part of the program from there and what they do unlike real programs no user would be running the kernel benchmarks they are solely used to evaluate performance. This is just used to evaluate the performance and as we know that kernels are also best to isolate performance of specific features of a machine and evaluate them some of the examples are Livermore loops, LINPAC, etc. And some compilers were reported to have been using benchmark specific optimizations. So, as to give the machine a good rating; that means, let us say we have so many benchmarks now.

So, now these are already available and you can use this to evaluate your performance. So, some compilers typically use some of those features to accelerate the speed of those programs only, but it may not work for any application it may work for specifically for those applications, but if you are not using such kind of constructs that are used in those programs then you will not be getting better result.

(Refer Slide Time: 30:10)

(c) Toy Benchmarks

- These are small pieces of code, typically between 10 and 100 lines.
- They are convenient and can be run easily on any computer.
- They have limited utility in benchmarking and hence sparingly used.
- Examples: Sieve of Eratosthenes, quicksort, etc.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

These are some toy benchmarks are also use the code typically between 10 to 100 lines, and they are convenient and can be run easily on any computer. They have limited utility in benchmarking and hence sparingly used.

(Refer Slide Time: 30:33)

(d) Synthetic Benchmarks

- Somewhat similar in principle to kernel benchmarks.
 - They try to match the average frequency of operations and operands of a large set of programs.
- Synthetic benchmarks are further removed from reality than kernels, as kernel code is extracted from real programs, while synthetic code is created artificially to match an average execution profile.
- Examples: Whetstone, Dhrystone, etc.
 - These are not real programs.
- Some drawbacks with synthetic benchmarks are discussed next.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA
20

Now, coming to synthetic benchmarks, what do you mean by synthetic? We know that this particular machine has this much type of ALU operation, this much type of store and load operation, etc. So, some synthetic benchmarks are generated, which will actually resemble to that particular frequency of operation which is performed for certain programs, but they are synthetic as they are not real benchmarks basically.

So, somewhat similar to the principles to kernel benchmarking, they try to match the average frequency of operations and operands of a large program. Just now what I have said let us say we have a program and we know that for this program 80% will be such kind of instruction ALU operation and 20% will be store-load operation. So, we also generate a particular program such that it uses same kind of features 80% will be ALU and 20% will be other. Synthetic benchmarks are further removed from reality than kernels, as kernel code is extracted from real programs while synthetic code is created artificially to match an average execution profile. So, these are made artificially to match an average execution profile. Some of the examples are Whetstone and Dhrystone; these are not real programs.

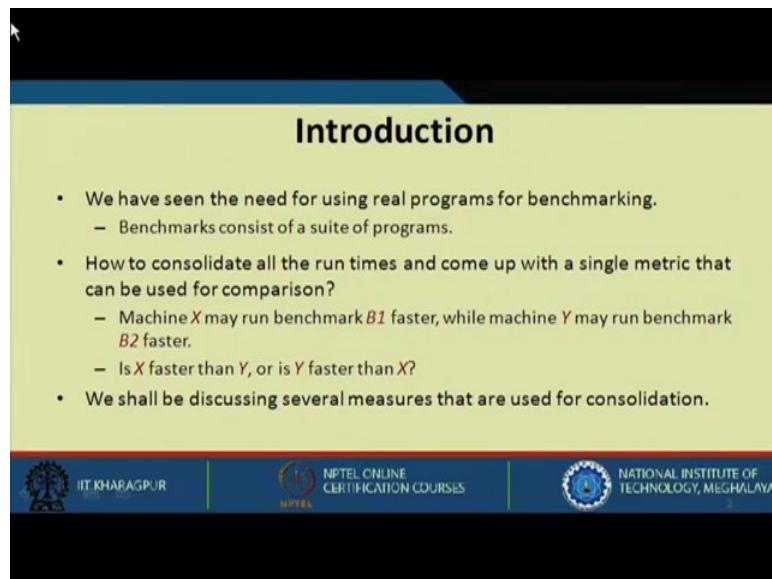
So, we came to end of lecture 13. So, where we have seen that what the choice of the benchmark, how do you choose a particular benchmark. So, that entirely depends on the application for which you are designing or you require the CPU for what kind of application.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture – 14
Summarizing Performance Results

(Refer Slide Time: 00:29)



Introduction

- We have seen the need for using real programs for benchmarking.
 - Benchmarks consist of a suite of programs.
- How to consolidate all the run times and come up with a single metric that can be used for comparison?
 - Machine *X* may run benchmark *B1* faster, while machine *Y* may run benchmark *B2* faster.
 - Is *X* faster than *Y*, or is *Y* faster than *X*?
- We shall be discussing several measures that are used for consolidation.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Hello. So, let us come to lecture 14 where we will be summarizing the performance results. What we have seen till now is using real programs for benchmarking like. So, it is very much essential that the benchmark should consist of set of programs, and those programs should be very much relevant.

Now how to consolidate all the run times and come up with a single metric that can be used for comparison? So, we have seen that MIPS is something we are calculating, we are calculating the total execution time. We have so many metrics now. And I can say that a machine A is performing some set of tasks that is taking so and so time. A machine B is also executing the same program and it is taking so on so time. Another set of programs is also been executed by both the processors, and they are giving some time.

Now, it may happen that A is giving better result compared to B for program 1. And B is giving better results for program 2 as compared to program 1. How can you tell that which processor will be better? So, we shall be discussing several measures that are used for this consolidation basically.

(Refer Slide Time: 02:01)

What about Reproducibility?

- Actual run time of a program on a machine depends on so many factors.
 - Degree of multiprogramming, disk usage, compiler optimization, etc.
- Reproducibility of the experiments is very important.
 - Anyone should be able to run the experiment and get the same results.
 - Benchmarks must therefore specify the execution environment very clearly.
- Example:- SPEC benchmarks mention details such as:
 - Extensive description of the computer and the compiler flags.
 - Hardware, software and baseline tuning parameters.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

By the term reproducibility, we mean that we can reproduce the same thing, the same result. Now, I should also say that it is not only that same program will get executed in other machine and we will check the result. I have to also give some more details. What are those some more details? Like what is the execution environment that I am using? How much is the disk utilization? What kind of cache memory I am using? What are the other features along with that program when you are executing?

So, you have to give all those parameters along with the program. Like how much swap space is possible, and so on and so forth. The actual run time of a program on a machine depends on so many factors. One is the degree of multi programming. Like, whether you are using multi programming or not. Disk usage --- how much disk usage is being done, compiler optimization. So, reproducibility of experiments is very important; that means, anyone should be able to run the experiment and get the same result. This can only happen if you actually do the same thing that other computer was doing for executing that program.

So, benchmark must specify the execution environment very clearly. For example, the SPEC benchmarks mentions details such as extensive description of the computer and the compiler flags. So, SPEC benchmark is not only giving you a program, along with that program it is giving you some more details, such that you will be able to reproduce

the same thing, if you execute that particular program. Hardware, software and baseline tuning parameters, like the swap space, like the cache memory, and so on and so forth.

(Refer Slide Time: 04:40)

How to Summarize Performance Results?

- The choice of a good benchmark suite that relates to real applications is essential to measuring performance.
- For a single program, it is very easy to say which computer runs faster.
- However, when there are multiple programs, the comparison may not be so straightforward.

An example

	CPU A (in secs)	CPU B (in secs)	CPU C (in secs)
Program P1	1	10	25
Program P2	500	250	10
Total Time	501	260	35

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, how to summarize performance results? The choice of a good benchmark suite that relates to real applications is essential to measuring performances. The meaning is, the choice of a good benchmark suite is really very important. For a single program it is very easy to say that which computer runs faster. But will it be equally easy, when you have a number of programs running on different machines? And you are getting different results. How can then you say that that this computer performs better than the other? Let us see this example, where programs are run on CPU A, CPU B, and CPU C --- 3 different processors. In seconds, program 1 is taking 1. This is taking 10 and this is taking 25, and these are taking this.

(Refer Slide Time: 05:45)

	CPU A (in secs)	CPU B (in secs)	CPU C (in secs)
Program P1	1	10	25
Program P2	500	250	10
Total Time	501	260	35

- We can make the following statements, which may depict a confusing picture when considered together:
 - A is 10 times faster than B for program P1.
 - B is 2 times faster than A for program P2.
 - A is 25 times faster than C for program P1.
 - C is 50 times faster than A for program P2.
 - B is 2.5 times faster than C for program P1.
 - C is 25 times faster than B for program P2.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, total time taken is this. Now let us see. We can make the following statements, which may depict a confusing picture, when considering together. If you consider it separately, for program 1, A is the fastest; it takes 1 second. And for program 2, C is the fastest, we can say. Now, let us say this with respect to A; A is 10 times faster than B. Very true, A takes 1 second, B takes 10 seconds. Similarly, B is 2 times faster than A for program P2. This is for program P1. Now B, which is taking 250 second and program P2 for processor A, it is taking 500.

So, this is taking twice the time. So, we can say B is 2 times faster than A for program P2. Again, A is 25 times faster than C for program P2. And C is 50 times faster than A for program P2. Because P2, this takes 500 this is taking only 10. Similarly, you can find out B is 2.5 times faster than C for program P1. C is 25 times faster than B for program P2. Now how can I say which processor is the best? It is more confusing, rather than getting some inference.

So, how to summarize this performance result? Let us see how we can do this. Choice of a good benchmark suite that relates to real applications is essential to measuring performance. So, for a single program it is very easy to say, but; however, when there are multiple programs, the comparison may not be so straightforward. So, we need to come up with some solution.

(Refer Slide Time: 08:02)

(a) Total Execution Time

- The simplest approach to summarize the relative performances is to look at the total execution times of the two programs.
 - CPU A : 501, CPU B: 260, CPU C: 35.
- Based on this measure, we can make the following comments:
 - B is $501 / 260 = 1.93$ times faster than A for the two programs.
 - C is $501 / 35 = 14.31$ times faster than A for the two programs.
 - C is $260 / 35 = 7.43$ times faster than B for the two programs.
- If the actual workload consists of running P1 and P2 unequal number of times, this measure will not give the correct result.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL



So, let us see the total execution time first. What is the total execution time? For CPU A, the total execution time is 501. For CPU B, the total execution time is 260. And for CPU C, it is 35. Based on this measure what we can see or what we can comment is, B is 1.93 faster than A for the 2 programs.

Now, we are not taking into consideration individual programs. Now we are saying something in terms of both the programs. C is 14.31 times faster than A for 2 programs. And C is 2 point 7.43 times faster than B for 2 programs. If the actual workload consists of running P1 and P2 unequal number of times, again this is something else. First we have given 2 programs and we are saying this is the execution times. And now we are saying that these programs how many times they will be running. So, the frequency of running those programs is now coming into picture.

(Refer Slide Time: 09:27)

The slide has a yellow background with a black header bar at the top. In the center, there is a white rectangular box containing text and a mathematical formula. Below the box, there is a small video frame showing a person speaking. The footer of the slide features logos for IIT Kharagpur, NPTEL, and NIT Meghalaya.

- Arithmetic Mean:
 - Defined as the average execution time for all the programs in the benchmark suite.
 - If XT_i denotes the execution time of the i -th program, and there are n programs, we can write

$$\text{Arithmetic Mean} = \frac{1}{n} \sum_{i=1}^n XT_i$$

So, if the actual workload consists of running P1 and P2 unequal number of times, this measure will not give the correct result either. So, now let us come up with something called arithmetic mean. So, this is defined as the average execution time, for all the programs in the benchmark suite. By average execution time what we mean is that, we are averaging the execution time summation of all divided by n. This is your arithmetic mean where XT_i denotes the execution time of i-th program. And there are n programs.

(Refer Slide Time: 10:02)

The slide has a yellow background with a black header bar at the top. In the center, there is a white rectangular box containing text and a bulleted list. Below the box, there is a small video frame showing a person speaking. The footer of the slide features logos for IIT Kharagpur, NPTEL, and NIT Meghalaya.

(b) Weighted Execution Time

- If the programs constituting the workload do not run equally, we can add a weightage factor to every program and carry out the calculation accordingly.
 - Thus, if 40% of the tasks in the workload is program P1, and 60% is program P2, we can define the corresponding weights as $W_1 = 0.4$, and $W_2 = 0.6$.
- Two alternate approaches are possible:
 - Weighted Arithmetic Mean
 - Normalized Execution Time

Similarly, what is weighted execution time? If the program constituting the workload do not run equally, as I said program A can run more times than program B. For some processor; program B can run more times. So, based on this how you can calculate.

So, for this if 40% of the task in workload is program P1, and 60% is program P2. We can define some weights associated with these programs. So, program P1 is having weight $W_1 = 0.4$. And $W_2 = 0.6$. Because P1 it is 40% of the tasks of the workload, and 60% for P2. Now we will see two alternative methods. One is weighted arithmetic means. Another is normalized execution time.

(Refer Slide Time: 11:21)

- Weighted Arithmetic Mean (WAM):
 - It is computed as the sum of the products of weighting factors and execution times.
 - If W_i denotes the weighting factor of program i , we can write

$$\text{Weighted Arithmetic Mean} = \sum_{i=1}^n (W_i \times XT_i)$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, arithmetic mean we have already seen. Let us see what is weighted arithmetic mean (WAM). It is computed as a sum of products of weighting factors and the execution times. So, we are not only considering only the execution time, we are taking into consideration the execution time and the weights associated with that particular execution time where W_i denotes the weighting factor of program i . This is the weighted arithmetic mean where we multiply W_i by execution time of i . And we take the sum.

(Refer Slide Time: 11:56)

Example 1

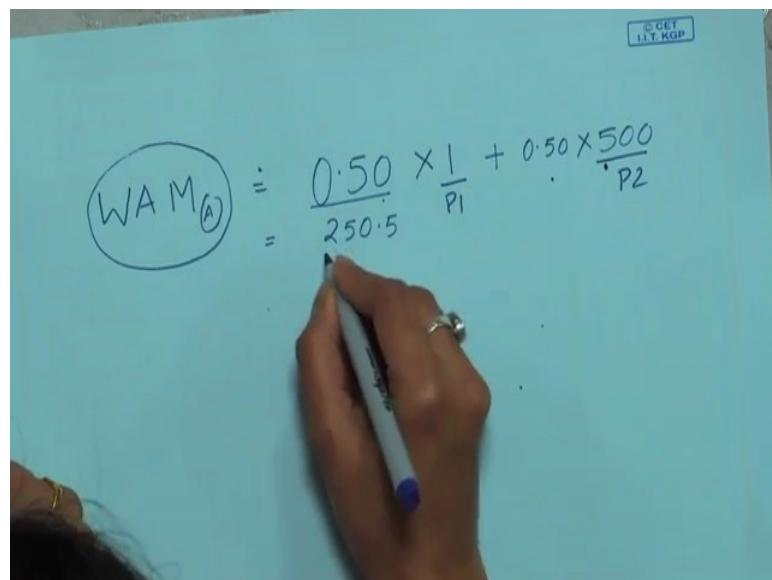
	CPU A (in secs)	CPU B (in secs)	CPU C (in secs)
Program P1	1	10	25
Program P2	500	250	10
Total Time	501	260	35

- If $W_1 = 0.50$ and $W_2 = 0.50$, we get $WAM_A = 250.5$, $WAM_B = 130$, $WAM_C = 17.5$
- If $W_1 = 0.90$ and $W_2 = 0.10$, we get $WAM_A = 50.9$, $WAM_B = 34$, $WAM_C = 23.5$
- If $W_1 = 0.10$ and $W_2 = 0.90$, we get $WAM_A = 450.1$, $WAM_B = 226$, $WAM_C = 11.5$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us see these 2 programs again. So, weight for first one is 0.50 and weight of second one is 0.50. So, we get weighted arithmetic mean of A as 250.5. How we are getting that? Let us see this. So, here you can see that 0.50 is multiplied by 1 for program P1 for CPU A.

(Refer Slide Time: 12:29)



So, this is CPU weighted arithmetic mean for A. We are multiplying this weight with the time that is 1. And similarly, we are multiplying 0.50 by 500 which is for program P2. And we are getting weighted arithmetic mean as 250.5. Weighted arithmetic mean for B

is this, and for C is this. Depending on these values, we have calculated and depending on these weights. Similarly, if you change the weight of these, the weighted arithmetic mean is different. Again if you change the weighted arithmetic mean is different. So, we here also we show you, how we can calculate the weighted arithmetic mean for several CPUs given the execution time for some programs.

(Refer Slide Time: 13:51)

(c) Normalized Execution Time

- As an alternative, we can normalize all execution times to a reference machine, and then take the average of the normalized execution times.
 - Followed in the SPEC benchmarks, where a SPARCstation is taken as the reference machine.
- Average normalized execution time can be expressed as either an arithmetic or geometric mean.

$$\text{Normalized Arithmetic Mean} = \frac{1}{n} \sum_{i=1}^n XTR_i$$

NPTEL ONLINE CERTIFICATION COURSES

NPTEL

IIT KHARAGPUR

NPTEL

Now, coming to normalized execution time. As an alternative we can normalize all execution times to a reference machine. And then take the average of the normalized execution times. So, now we are saying that there will be a reference machine, and with respect to that reference machine we will be calculating this. So, we are normalizing it with respect to a reference machine. Followed in SPEC benchmarks where a SPARC station is taken as the reference machine. So, what will be the average normalized execution time? So, this can be expressed as execution time, what is XTR? Execution time, with the reference XTR_i and summation of all of these.

So, one machine can be taken as a reference and can be calculated based on that. We will be seeing that next. That is called normalized arithmetic mean. Which is execution time of the programs with respect to reference machine summation of that divided by n.

(Refer Slide Time: 15:11)

The slide has a yellow header and footer. The header contains the title 'Normalized Geometric Mean'. The footer features logos for IIT Kharagpur, NPTEL, and NIT Meghalaya.

Normalized Geometric Mean = $\sqrt[n]{\prod_{i=1}^n XTR_i}$

- Here, XTR_i denotes the execution time for the i -th program, normalized to the reference machine.

So, here XTR_i denotes the execution time for i -th program normalized to the reference machine. This is normalized geometric mean. Earlier it was normalized arithmetic mean. Now we are going to normalize geometric mean.

(Refer Slide Time: 15:38)

The slide is titled 'Example 2'. It shows a table of execution times for Program P1 and Program P2 on three CPUs (A, B, C). It then shows the normalized values for each program relative to CPU A.

	CPU A (in secs)			CPU B (in secs)			CPU C (in secs)		
Program P1	1			10			25		
Program P2	500			250			10		
Total Time	501			260			35		

	Normalized to A			Normalized to B			Normalized to C		
	A	B	C	A	B	C	A	B	C
Program P1	1.0	10.0	25.0	0.1	1.0	2.5	0.04	0.4	1.0
Program P2	1.0	0.5	0.02	2.0	1.0	0.04	50.0	25.0	1.0
Arithmetic mean	1.0	5.25	12.51	1.05	1.0	1.27	25.02	12.7	1.0
Geometric mean	1.0	2.24	0.71	0.45	1.0	0.32	1.41	3.16	1.0

Now, let us see how do we compute this. So, program 1 has the same parameter that we were using. Now see what we are doing. These values we are calculating by with respect to normalized to A; that means, when we are saying normalized to A we are making 1 here, these 2 will become 1. And with reference to that what are these values. So, now

see this is 1. 10 divided by 1.25 divided by 1. Similarly, this one is, how do you get 1 here? You will get 1 here by dividing by 500.

So, this has become 1 you divide by 500 it will become 0.02. So, all these values are normalized with respect to A.

(Refer Slide Time: 16:49)

Let us see, how do we normalize with respect to B similarly. When we say we normalized with respect to B, then we have to make all these as 1, 1, 1. Initially it was 10 this was 250, and 0. So, what we are doing? We are making others reference normalized to B. So, what we are doing; this is 1 divided by 10, and this will become 25 divided by 10, and here it was 250. So, you have to divide it by 250.

So, we are dividing 500 divided by 250 to make it 1. So, we will get here 1. And 10 divided by 250. So, just see this what we are getting. We are normalized this with respect to 1, we are making this as 1. And now these are the 2 values we are getting. And similarly we have normalized this 2, see these 2 have become 1. And we are getting this value with respect to the difference machine. Now once we are done with this we calculate the arithmetic mean and we calculate the geometric mean.

Now, say arithmetic means still we cannot say which one is better. So, in this case this is one. In this case this is one. And in this case this one. So, it is very difficult to come up with a conclusion based on this, but you see the geometric mean seems to be consistent.

C is taking the least for here, also here. So, the geometric mean is considered as one of the metric that can be used for evaluation. Although arithmetic mean cannot be used properly.

(Refer Slide Time: 19:11)

The slide has a blue header bar. The main content area is yellow and contains the following text:

- Summary:
 - In contrast to arithmetic means, geometric means of normalized execution times are consistent no matter which machine is the reference.
 - Hence, the arithmetic mean should not be used to average normalized execution times.
 - One drawback of geometric mean is that they do not predict execution times.
 - Also can encourage hardware and software designers to focus their attention to those benchmarks where performance is easiest to improve rather than the ones that are the slowest.



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



So, in summary what we can say that, in contrast to arithmetic mean or geometric mean normalized to the execution time are consistent, no matter, which machine is the reference one. So, what we have seen, we have made A as a reference machine then also see C is better. We have made B as a reference machine in that case also C is giving better. And when you are making C as the reference machine then also it is giving better. So, the result seems to be consistent, even if different programs are executed different times.

Hence arithmetic mean should not be used to average normalized execution times, but there is one drawback of geometric means that they do not predict the execution time. You cannot give a prediction of the execution time, but it is consistent because it is giving that value where C seems to be better. And it also encouraged hardware and software designers to focus their attention to those benchmarks where performance is easiest to improve rather than the ones that are slowest. So, we generally improve where the requirement is high; that means, certain instruction are executed more we try to improve that part more. Certain instruction are executed less, but that does not mean we will leave that part also. That part also we should take into consideration. But it generally

encourages the hardware and software designer to focus attention at those benchmarks where performance is easiest to improve. That is the thing.

So, we came to the end of this lecture. So, in the last three lectures what we have tried to show is, what is performance. What performance metrics can be used?

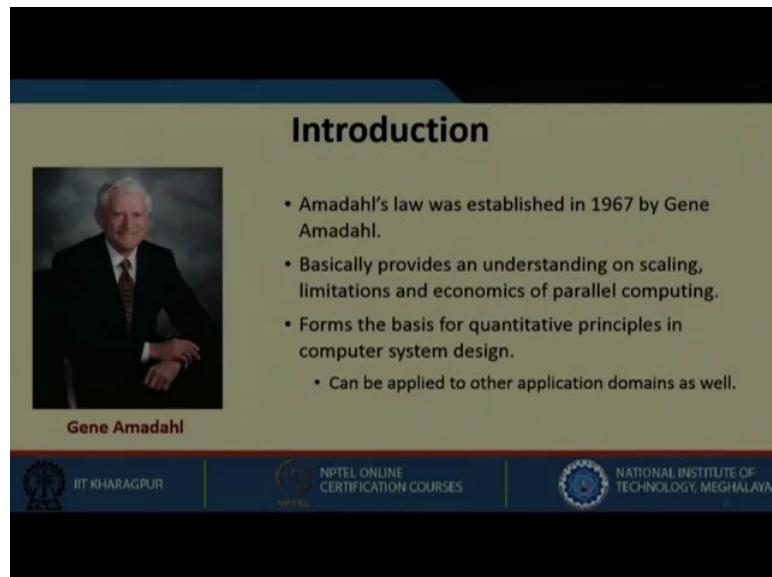
Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture – 15
Amadahl's Law (Part I)

Welcome to the 15th Lecture on Amadahl's Law. So, till previous lecture what we have seen that how we can summarize all the results, and then come up with a particular solution. Now we will see another law that will tell us little more about what we have learnt.

(Refer Slide Time: 00:46)



Amadahl's law was established in 1967 by Gene Amadahl. And what it gives is that it provides an understanding on scaling limitation and economics of parallel computing. So, by scaling limitation we can understand that the transistor size is becoming smaller and smaller. So, it gives an understanding on scaling limitation and also on the economics of parallel computing.

By economics of parallel computing what we mean is that --- suppose today we have quad core, we have multi core technologies. So, when you are investing or when you are having so many cores, are you getting the return; when you are having just one core how much was your performance, by having more number of cores are you really getting that

advantage that you are supposed to get? So, it also gives us the economics of parallel computing.

And of course, it forms the basis for quantitative principles in computer system design. We can actually in a quantitative fashion tell about the design principles of a computer system. And Amadahl's law is not only for computer system design, it can be applied to other applications as well and for other domains.

(Refer Slide Time: 02:30)

What is Amadahl's Law?

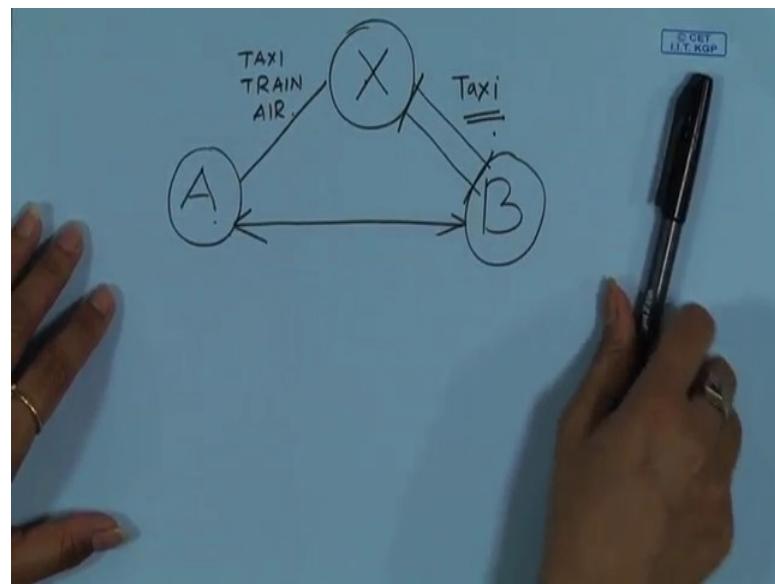
- It can be used to find the maximum expected improvement of an overall system when only *part of the system* is improved.
- It basically states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
- Very useful to check whether any proposed improvement can provide expected return.
 - Used by computer designers to enhance only those architectural features that result in reasonable performance improvement.
 - Referred to as *quantitative principles in design*.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us see what exactly is Amadahl's law. It is basically used to find the maximum expected improvement of an overall system when only a part of the system is improved. What do you mean by this? Suppose we have full overall system, but maybe part of it let us say only 40% of that part we can make an improvement. So, we want to see that by making 40% of the part improved how we can get an overall improvement.

So, I will just give you a small example. Let us see you want to travel from place A to place B.

(Refer Slide Time: 03:28)



So, I want to travel from A to B, and you have a middle point because you have to travel through this point, you have to reach this point and from this point again you have to take. Now from this part to this part let us say for travelling you can travel through taxi and there is no other way. But for travelling from A to X you can use many modes: one is taxi, one is train, another is by air. So, what you can basically do is that you cannot do anything for this part, this part is limited by the speed of this taxi only, but all we can do is that we can improve this part that is from A to X by either using taxi, using train, or by air.

So, this is where we will have to see that how can we optimize, we need to reach from A to B through X, and then only I can improve this part not this part this part. But by improving this particular part how much improvement we are getting out of it that we want to see. So, that is where Amadahls' law comes into picture.

It can be used to find the maximum expected improvement of an overall system. We will see the improvement of an overall system when only a part of which is improved, because another part we cannot improve. It basically states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of time the faster mode can be used. So, these again come into terms of execution of some instructions. So, by this what we mean is that as we are saying that

this we are actually improving a fraction of the part, and fraction of that particular part can be used more, than the overall improvement can be further increased.

So, very useful to check whether any proposed improvement can provide expected return or not; that is what I am saying. That let us say I put some improvement on that particular part where I can improve, but by putting improvement on that particular part will you actually get that return that you are expecting, or you will not be able to get that return, or how much return you will be able to get. So, all these things can be determined through Amadahl's law.

And it is used by computer designers to enhance those architectural features that result in reasonable performance improvement. And this is referred to as quantitative principles in design. So, again by this what we mean? Let us say we have an adder, a multiplier and in general purpose computing we may say we are using adder more than multiplier. So, in such cases what can happen? If you try to improve the multiplier that you are not using much, then it will not give you that particular kind of improvement.

So, that is also what you have to look into. That we are improving a part that is also used more number of times. The number of times it is used is more and we are improving that particular part, so where we can get overall improvement much more.

(Refer Slide Time: 07:32)

- Amadahl's law demonstrates the *law of diminishing returns*.
- An example:
 - Suppose we are improving a part of the computer system that affects only 25% of the overall task.
 - The improvement can be *very little* or *extremely large*.
 - With "*infinite*" speedup, the 25% of the task can be done in "*zero*" time.
 - Maximum possible speedup = $XT_{\text{orig}} / XT_{\text{new}} = 1 / (1 - 0.25) = 1.33$

We can never get a speedup of more than 1.33

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

A video player shows a woman in a pink shirt speaking.

Amadahl's law demonstrates the law of diminishing return. So, what is that? Let us take an example; suppose we are improving a part of the computer system that affects only 25% of the overall task. So, out of 100% only 25% of the work can be improved. The improvement can be very little or extremely large, let us say how. With infinite speed up, only 25% of the part can be improved; and we are saying that if we make that 25% part task can be done in 0 time maximum, that much improvement can be done that it will take no time to finish that 25% of the task, but we cannot do anything with the rest 75% of the task.

75% of the task remains, but only that 25% of the task we are making some improvement. And what kind of improvement, we are saying that we are keeping that time 0. So, in no time that particular part of task can be performed. So, maximum possible speed up can be $XT_{\text{orig}} / XT_{\text{new}}$. So, if the original execution time is 1, what will be the new execution time; out of total 100% of the task 25% can be done in new time. So, you can just take out that 0.25 which is coming to 1.33.

So, we can have a maximum of this much improvement. If you can improve that 25% and we can say that the 25% of the task can be done in new time. So, we can never get a speed up more than 1.33, even if you do whatever you want we cannot get an overall improvement because we can only make that 25% faster, because 75% still is working at the previous speed.

(Refer Slide Time: 09:46)

- Amadahl's law concerns the speedup achievable from an improvement in computation that affects a fraction F of the computation, where the improvement has a speedup of S .

Before improvement	$1 - F$	F
After improvement	$1 - F$	F/S

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

So, Amadahl's law actually concerns the speedup achievable from an improvement in computation that affects a fraction F of the computation, where the improvement has a speed up of S. So, on this fraction F we are having an improvement of S. So, this is the F portion, and this is $1-F$. So, $1-F$ there will be no change, and we can only have a speed up of S to this F portion. So, if you can reduce this so it will become F / S . So, earlier it was taking this much time and now it is taking this much time. So, after improvement we can execute the same task by this percentage.

(Refer Slide Time: 10:45)

- Execution time before improvement: $(1 - F) + F = 1$
- Execution time after improvement: $(1 - F) + F / S$
- Speedup obtained:

$$\text{Speedup} = \frac{1}{(1 - F) + F / S}$$

- As $S \rightarrow \infty$, Speedup $\rightarrow 1 / (1 - F)$
 - The fraction F limits the maximum speedup that can be obtained.

So, let us find out how we can find out the overall speed up. So, execution time before the improvement is 1. This was fractional part and this is $1-F$, basically this is 1. And what is the execution time after improvement we have seen; $1-F$ will remain as it is, and we have to make improvement on F. So, it will become F / S . So, speed up will be $XT_{\text{old}} / XT_{\text{new}}$. XT_{new} is $(1 - F) + F / S$; this is what we have got.

So, as S tends to infinity, speed up will be this. If you make this part 0, so what will happen? The speed up will be $1 / (1 - F)$. So, the fraction F actually limits the maximum speedup that can be obtained.

(Refer Slide Time: 12:12)

• Illustration of law of diminishing returns:
– Let $F = 0.25$.
– The table shows the speedup ($= 1 / (1 - F + F / S)$) for various values of S .

S	Speedup	S	Speedup
1	1.00	50	1.32
2	1.14	100	1.33
5	1.25	1000	1.33
10	1.29	100,000	1.33

1 / (1 - 0.25) =
1.33

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, let us illustrate this law of diminishing return for $F = 0.25$ by changing the value of S . So, speed up is $1 / (1 - F + F / S)$ for various values of S we have computed. So, let us say when S is 1 we are getting as total speed up of 1.00, when it is 2 we are getting 1.14, when it is 5 it is 1.25, when it is 10 it is 1.29. So, the difference if you see it is basically reducing. This difference was more, this is less, this is less, this and this is even less.

And what we can see is that after certain amount of time that is after certain speedup we are not getting any further speedup. The speed up is limited by a factor of 1.33, because we have already said we can get a maximum speed up of 1.33; that is what we are getting, even if you are increasing this fractional speed up by any amount. So, unnecessarily it is no point increasing this speedup as there is a limit to it, because the maximum speedup that can be achievable is 1.33.

(Refer Slide Time: 13:51)

• Illustration of law of diminishing returns:

- Let $F = 0.75$.
- The table shows the speedup for various values of S .

$$1 / (1 - 0.75) = 4.00$$

S	Speedup
1	1.00
2	1.60
5	2.50
10	3.08

S	Speedup
50	3.77
100	3.88
1000	3.99
100,000	4.00

So, let us take another example where $F = 0.75$. This table also shows the speed up for various values of S ; as we increase S what it depends on how it affects overall speed up. So, again we see that when S is 100000, then the speed up is 4. And the maximum achievable speed up when the fraction is 0.75 is also 4. So, even if you increase this S , this will remain the same.

(Refer Slide Time: 14:38)

Design Alternative using Amadahl's law

Diagram illustrating design alternatives using Amadahl's law:

- Loop 1: 500 lines (10% of total execution time)
- Loop 2: 20 lines (90% of total execution time)

So, these are some design alternatives using Amadahl's law; what we are saying that let us say we have a portion of a program this is loop 1 and this is loop 2. This loop is of 500

lines and this loop is of 20 lines. But this 500 lines of code takes 10% of the total execution time. What do we mean by that, is this takes certain amount of time to execute, but this particular loop is only taking 10% of that total time. And this Loop 2 which is only 20 lines of code, but it is taking 90% of the total execution time. Although this is small portion, but that small portion is taking more amount of time; so we need to also take into consideration this design alternative.

(Refer Slide Time: 15:46)

- Some examples:
- We make 10% of a program 90X faster, speedup = $1 / (0.9 + 0.1 / 90) = 1.11$
- We make 90% of a program 10X faster, speedup = $1 / (0.1 + 0.9 / 10) = 5.26$
- We make 25% of a program 25X faster, speedup = $1 / (0.75 + 0.25 / 25) = 1.32$
- We make 50% of a program 20X faster, speedup = $1 / (0.5 + 0.5 / 20) = 1.90$
- We make 90% of a program 50X faster, speedup = $1 / (0.1 + 0.9 / 50) = 8.47$

The slide footer contains the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and a video frame showing a person speaking.

So, let us see if you make some improvement on this particular case, where only 10% of the total execution time is used for Loop 1, and 90% of the total execution time is used for Loop 2. So, let us say we make 10% of the program 90 times faster, so how much speed up we will get? So, this is 10% of the program you are making 90 times speed up. So, 1 divided by this is $(1 - F)$ and this is F / S , where F is 0.1 and S is 90, because we are making 90% speed up of on this. And what we are getting? We are getting a value 1.11, the overall speed up.

Now we make 90% of the program 10 times faster only, but that portion is 90%. So, 1 divided by this is $(1 - F)$ and this is $F / 10$ and we are getting 5.26 which is much more, but we are only making a speedup of 10 times. So, making 10 times speed up of a portion that is used most amount of time, which is taking more amount of time that is 90% it is taking that will giving a better result.

Similarly make 25% of the program 25 times faster. So, 25% we are making 25 time faster and this remains $(1 - F)$ is 0.7 we are getting this much. Similarly, 50% of the program making 20% faster so this is giving 1.90. And if we make 90% of the program 50 times faster we are getting a value where we are getting maximum, because this is 90% of your program we are making a speedup of 50%. So, here we are getting a value of 8.47.

So, this is how you can see that if you are making improvement on a part of your program that is used more number of time that will give a better speedup.

(Refer Slide Time: 18:27)

Example 1

- Suppose we are running a set of programs on a RISC processor, for which the following instruction mix is observed:

Operation	Frequency	CPI _i	W _i * CPI _i	% Time	CPI = 2.08
Load	20 %	5	1.00	0.48	→ 1 / 2.08
Store	8 %	3	0.24	0.12	
ALU	60 %	1	0.60	0.29	
Branch	12 %	2	0.24	0.11	

We carry out a design enhancement by which the CPI of Load instructions reduces from 5 to 2. What will be the overall performance improvement?



IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES



Let us take another example. Suppose we are running a set of programs on a RISC processor for which the following instruction mix is observed. So, we are having load, store, ALU and branch operations. And this is the frequency at which load instructions are taking place; this is the frequency at which store instruction is taking place, and so on. And this is the CPI for load, store, ALU and branch. Now, let us find out $W_i * CPI$. So, we multiply this into 0.2 we get this, this into 0.08 we get this, and so on.

Now percentage time it is used can be found out by total CPI; total CPI will be 2.08. So, if total CPI is 2.08, we will divide 1 / 2.08 to get this. So, what we do this is the given thing and we have found out $W_i * CPI$ and also the percentage of time it is used. Now we carry out some kind of enhancement and now we will see what is the improvement that we can get.

So, we carry out a design enhancement by which the CPI of load instruction reduces from 5 to 3. So, earlier this load instruction was taking 4 CPI, now it will take 2 CPI. So, what will be the overall performance?

(Refer Slide Time: 20:55)

Fraction enhanced $F = 0.48$
Fraction unaffected $1 - F = 1 - 0.48 = 0.52$
Enhancement factor $S = 5 / 2 = 2.5$
Therefore, speedup is

$$\frac{1}{(1 - F) + F / S} = \frac{1}{0.52 + 0.48 / 2.5} = 1.40$$

Now, we want to see the overall performance that we will get. So, how do we get it? So, fraction enhanced F is 0.48. So, we can see that we are reducing from 5 to 3. So, fraction which we can actually enhance is 0.48 and unaffected portion will be 0.52, that is $(1 - F)$. So, if F is this and $(1 - F)$ is 0.52, what will be S ? Earlier it was 5 now it has become 2. So, this speedup will be old divided by new, which is coming to 2.5. So, if we just put it in this value we get 1.40.

(Refer Slide Time: 21:59)

- Alternate way of calculation:
 - Old CPI = 2.08
 - New CPI = $0.20 * 2 + 0.08 * 3 + 0.60 * 1 + 0.12 * 2 = 1.48$

$$\begin{aligned} \text{Speedup} &= \frac{XT_{orig}}{XT_{new}} = \frac{IC * CPI_{old} * C}{IC * CPI_{new} * C} \\ &= \frac{CPI_{old}}{CPI_{new}} = \frac{2.08}{1.48} = 1.40 \end{aligned}$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Therefore, the speed up will be 1.40 with that enhancement. Similarly we can solve this using the other way; what is the other alternative? Let us just go back to two slides and figure out we have found out the CPI that is 2.08. Now we will make an enhancement and make this as 2, and then we will calculate the CPI once more. So, let us see how to do this.

So, old CPI was 2.08 and the new CPI we can find as 1.48. Earlier it was 2.08 when for the load instruction the CPI was 5. Now we have made some improvement and made the CPI = 2 and we have figured out the new CPI is 1.48. So, with this, what will be the speedup?

So, the formula of execution time if we recall from our previous lecture is instruction count x cycles per instruction. So, we are getting 1.40. So, either way if we do it we are getting 1.40. So, these are the two alternative ways through which you can perform this.

(Refer Slide Time: 23:59)

Example 2

- The execution time of a program on a machine is found to be 50 seconds, out of which 42 seconds is consumed by multiply operations. It is required to make the program run 5 times faster. By how much must the speed of the multiplier be improved?
 - Here, $F = 42 / 50 = 0.84$
 - According to Amadahl's law,
$$S = 1 / (0.16 + 0.84 / S)$$

or, $0.80 + 4.2 / S = 1$
or, $S = 21$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Let us take another example. In this example what we are saying the execution time of a program on a machine is found to be 50 seconds, out of which 42 seconds is consumed by multiply operation. So, most of the time multiply operation is taking more time. It is required to make the program run 5 times faster.

So, the overall speedup of the entire system you want to make it 5 times by how much speedup of the multiplier, by how much must the speed of the multiplier be improved. So, let us say it has been given that you want this 5 times speedup. So, how much this multiplier part can be improved? So, we will just put that in the formula F will be $42 / 50$ that is 0.84, and according to Amadahl's law speedup is 5 will be equal to $1 / (1 - F)$.

So, I am doing dividing that by S because we need to find out this how much speedup on the multiplier you have to do such that the overall speed up is 5. So, if you solve this equation you will get S as 21. So, you have to make a speedup of 21 times to the multiplier; so as to get an overall speed up of 5. So, this is how this problem can be solved.

(Refer Slide Time: 26:05)

Example 2a

- The execution time of a program on a machine is found to be 50 seconds, out of which 42 seconds is consumed by multiply operations. It is required to make the program run 8 times faster. By how much must the speed of the multiplier be improved?
 - Here, $F = 42 / 50 = 0.84$
 - According to Amadahl's law,
$$8 = 1 / (0.16 + 0.84 / S)$$

or, $1.28 + 6.72 / S = 1$
or, $S = -24$

No amount to speed improvement in the multiplier can achieve this.
Maximum speedup achievable:
$$1 / (1 - F) = 6.25$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Let us take another example. It is the same example, but I want to make it more fast. The execution time of the program and the machine is found to be 50 seconds out of which 42 seconds is consumed by multiply operation it is required to make the program 8 times faster. So, similar way earlier it was 5 now it is 8, I put it 8 and I try to solve it. And what I get? I am getting a negative value. Why I am getting a negative value? We need to see that can we at all make 8 times faster, because there is a limitation of something that how much part the enhancement can be performed.

So, in this case we can see that $1 / (1 - F)$ is 6.25. So, the maximum achievable speed up is 6.25 and you are saying you want to get 8; that is not possible. So, in this case you cannot have a speed up of more than this achievable speedup; as we have already seen that earlier. So, no amount of speed improvement in the multiplier can achieve this.

(Refer Slide Time: 27:40)

Example 3

- Suppose we plan to upgrade the processor of a web server. The CPU is 30 times faster on search queries than the old processor. The old processor is busy with search queries 80% of the time. Estimate the speedup obtained by the upgrade.
 - Here, $F = 0.80$ and $S = 30$
 - Thus, speedup = $1 / (0.20 + 0.80 / 30) = 4.41$

The video player at the bottom shows a woman in a pink shirt speaking. The NPTEL logo and course information are visible in the background.

Let us take another example. Suppose we plan to upgrade the processor of a web server. The CPU is 30 times faster on search queries than the old processor. So, the CPU is 30 times faster on the search queries than the old processor and the old processor is busy with search queries 80% of the time. So, the old processor was always busy 80% of the time with the search queries only. Estimate the speed up obtained by this upgrade.

So, we are making an upgrade of 30% on the search queries and the old processor was using 80% of the time that search queries. So, F will be 0.80 and $(1 - F)$ will be 0.20. So, you substitute in this particular equation in the speedup and you get 4.14. So, with this upgrade we can get the speed up of 4.14.

So, in this lecture we have seen what is Amadahl's law and we have also seen that how the maximum speedup achievable can be found through this Amadahl law. And we have also seen that while designing certain system, you have to take in to consideration that if the speedup is not achieved on the entire system, it can be only made on the part of the system; we need to analyze that based on the part of the system how much overall speedup you can get. So, Amadahl's law actually states that on the part where you are making improvement how much maximum speedup you can make, that is the overall speedup you can achieve.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture – 16
Amdahl's Law (Part II)

(Refer Slide Time: 00:25)

Example 1

- The total execution time of a typical program is made up of 60% of CPU time and 40% of I/O time. Which of the following alternatives is better?
 - Increase the CPU speed by 50%
 - Reduce the I/O time by half

Assume that there is no overlap between CPU and I/O operations.

CPU	I/O	CPU	I/O	CPU	I/O
-----	-----	-----	-----	-----	-----

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Welcome to lecture 16 where we will continue with Amdahl's law little more. Let us start with an example where we see that, the total execution time of a typical program is made up of 60% of the CPU time, and 40% of the IO time. So, out of total number of tasks we are dividing into 2 types. One is, CPU bound job another is IO bound job. So, CPU bound job is taking 60% of CPU time and another is taking IO bound is taking 40%.

So, let us see this overall thing as CPU I/O CPU I/O. And we assume that there is no overlap between CPU and IO operation; that means, when CPU is used, only CPU bound job will be taken care of. When IO bound job is getting performed, only IO bound jobs will be taking care of and so on. So, we want to see that, in this particular scenario, there are 2 alternatives that we are trying to improve the performance. So, we want to see which one is better. So, first one is increase the CPU speed by 50%. And in another case we are reducing the IO time by half; that means; when you are increasing the CPU speed

by 50% we are not doing anything on this 40%, and when we are reducing the IO time by half, we are not doing anything on this 60%. So, let us see this.

(Refer Slide Time: 02:22)

- Increase CPU speed by 50%
 - Here, F = 0.60 and S = 1.5
 - Speedup = $1 / (0.40 + 0.60 / 1.5) = 1.25$
- Reduce the I/O time by half
 - Here, F = 0.40 and S = 2
 - Speedup = $1 / (0.60 + 0.40 / 2) = 1.25$

Thus, both the alternatives result in the same speedup.

Firstly, increase CPU speed by 50%. So, here F clearly is 0.60. And S will be 1.5. And so overall speedup will be 1.25.

Similarly, we say that we reduce the IO time by half. So, as you are reducing IO time by half; that means the speedup you are increasing by a factor of 2. So, in this case F is 0.40 and S is 2. You put it in the formula of speedup and we are getting the same result. So, both alternatives result in the same speed up, as what I said just.

(Refer Slide Time: 04:04)

Example 2

- Suppose that a compute-intensive bioinformatics program is running on a given machine X, which takes 10 days to run. The program spends 25% of its time doing integer instructions, and 40% of time doing I/O. Which of the following two alternatives provides a better tradeoff?
 - a) Use an optimizing compiler that reduces the number of integer instructions by 30% (assume all integer instructions take the same time).
 - b) Optimizing the I/O subsystem that reduces the latency of I/O operations from 10 μ sec to 5 μ sec (that is, speedup of 2).

The slide footer includes logos for IIT Kharagpur and NPTEL, and text for NPTEL Online Certification Courses.

Now, let us take another example. Suppose that compute intensive bioinformatics program is running on a given machine X that takes 10 days to run. So, it is compute intensive. Lot of computation is going on for this program. The program spends 25% of its time doing integer instructions, and 40% of the time doing IO.

So, 25% of the time is spent on doing integer, and 40% of the time doing IO. Which of the following 2 alternatives provides a better tradeoff? The first one is, use an optimization compiler that reduces the number of integer instructions by 30% assume all instruction take the same time. And the next one is we are optimizing the IO subsystem, that reduces the IO operations from 10 micro second to 5 micro second. That is again we are saying a speedup of 2. Let us see the 2 alternatives.

(Refer Slide Time: 05:44)

- Alternative (a):
 - Here, $F = 0.25$ and $S = 100 / 70$
 - Speedup = $1 / (0.75 + 0.25 * 70 / 100) = 1.08$
- Alternative (b):
 - Here, $F = 0.40$ and $S = 2$
 - Speedup = $1 / (0.60 + 0.40 / 2) = 1.25$

So, in the first case F is 25%, and S is we are reducing that by 70 because 30% is reduced. And the program spends 25% of the time in integer operation. So, S becomes $100 / 70$. So, 0.75 if there is no change on that. And 0.25 we are making a change, and we are getting a speedup of 1.08.

Let us take the next alternative where IO bound job is 40%. And the rest 60% we are not doing anything. But on that 40% we are having a speedup of 2. So, in this case if we solve for the speedup equation, we are getting 1.25. So, in this case we can say that, alternative B is better than alternative A, where here the speedup on 25% is made and here the speedup on 40% is made.

(Refer Slide Time: 07:10)

Amadahl's Law Corollary 1

- Make the common case fast.
 - Here "*common*" means most time consuming, and not "*most frequent*".
 - According to Amadahl's law, improving the "*uncommon*" case will not result in much improvement.
 - The "*common*" case has to be determined through experimentation and profiling.
 - When optimizations are carried out, a case that was common earlier may become uncommon later, or vice versa.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us move on with the Amdahl's law Corollary 1. What it says is, make the common case fast. What do you mean by common case? By common case we mean that most time consuming and not most frequent. So, let us understand this. There are 2 things. One is when we say that we are more frequently doing something. But, more frequently with which we are doing it is taking less time. But the most common means it is consuming the most amount of time. So, it is most time consuming. According to Amdahl's law, improving the uncommon case will not result in much improvement. Rather if we make improvement on the common case, the improvement will be much more; that means, the portion of the task that is taking more time we will try to improve on that part. Rather than which is more frequently used because most frequently that use that is not taking much time.

So, the common case has been determined through experimentation and profiling. So, how we can say that this part is common, through experimentation and through profiling. When optimizations are carried out, a case that was common earlier may become uncommon later or vice versa. So, you need to analyze the program. You need to analyze it time and again to understand this. So, when we are saying common part, common part means most time consuming part. So, you have made some improvement and you have reduced that time. And then again if you can finally say that now once this common part I have reduced, next time when you do that some it might change. So, this particular phenomenon says that when optimizations are carried out, a case that was common

earlier may became uncommon later because you have already taken care of that particular case.

(Refer Slide Time: 09:47)

Amadahl's Law Corollary 2

- Amadahl's law for latency (L)
 - By definition, $L_{new} = L_{old} / Speedup$
 - By Amadahl's law, $L_{new} = L_{old} * ((1 - F) + F / S)$
 - We can write:
$$L_{new} = L_{old} * F / S + L_{old} * (1 - F)$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now Amdahl's law Corollary 2, for latency what it says let us see. $L_{new} = L_{old} / \text{speedup}$.

(Refer Slide Time: 10:33)

Amadahl's Non-Corollary

$$L_{new} = L_{old} * F / S + L_{old} * (1 - F)$$

- Amadahl's law does not bound slowdown.
 - Things can get arbitrarily slow if we hurt the non-common case too much.
- Example: Suppose $F = 0.01$ and $L_{old} = 1$
 - **Case 1:** $S = 0.001$ (1000 times worse)
$$L_{new} = L_{old} * 0.01 / 0.001 + L_{old} * 0.99 = 10 * L_{old}$$
 - **Case 2:** $S = 0.00001$ (10^5 times worse)
$$L_{new} = L_{old} * 0.01 / 0.00001 + L_{old} * 0.99 = 1000 * L_{old}$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us see this Amdahl's non-corollary. So, here what it says that, Amdahl's law does not bound the slowdown; that means, we are always saying that a part of the program where improvement can be made, and you can finally, get an overall speedup of this

much. But what if those portions where we are not making, if you slowdown that particular portion then how it will affect?

So, the parts that are not used so much and you are trying to further make some techniques such that that particular part will reduce. I mean, the performance of that part will further reduce. So, you are not making any speedup on that and you are for making something such that, that performance of that part will further reduce. Then what will happen? So, things can get arbitrarily so slow if we hurt the non-common case too much. So, we should not do that as well. So, those portions which are not so common we cannot hurt that too much. So, we will see this example suppose F is 0.01 and L_{old} is 1.

Now, we are making a part 100 times worse. So, this becomes the speedup becomes 0.001 on the uncommon part that that is now your F . So, L_{old} will become multiplied by 0.01. That is, 0.99. So, how much it is coming? It is coming 10 times the old value. So, L_{new} is 10 times the old value. So, it has worsened. And in case 2, if it is 10^5 times worse then what is happening? Finally, we are getting $1000 * L_{\text{old}}$, which is even worse. So, we cannot just make the non-common part too much bad.

(Refer Slide Time: 13:18)

Extension to Multiple Enhancements

- Suppose we carry out multiple optimizations to a program:
 - Optimization 1 speeds up a fraction F_1 of the program by a factor S_1
 - Optimization 2 speeds up a fraction F_2 of the program by a factor S_2

$1 - F_1 - F_2$	F_1	F_2
-----------------	-------	-------

$1 - F_1 - F_2$	F_1 / S_1	F_2 / S_2
-----------------	-------------	-------------

$$\text{Speedup} = \frac{1}{(1 - F_1 - F_2) + F_1 / S_1 + F_2 / S_2}$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Now, let us see this extension of Amdahl's law to multiple enhancements. By multiple enhancement what we are meaning is that, earlier we were we were saying that we have a total portion, out of which one part we are making some improvement. And the other part remains the same. Now let us say, in a computation, we have IO bound job, we have

CPU bound job we have some other kinds of job as well. And now we are making improvement on these parts. So, earlier we were just making improvement on one, now we are saying that we will make improvement on these as well, others as well. So, in that case what will happen? How this multiple enhancement can be taken care in Amdahl's law? Let us see we have already seen for one part one fractional part if you improve what final speed up we will gain. Now we will see that we have multiple parts and we are improving multiple parts, and now how much you will gain.

Suppose we carry out multiple optimizations to a program. So, optimization 1 speeds up fraction F_1 of a program by factor S_1 , optimization 2 speeds of fraction F_2 of a program by a factor S_2 . So, F_1 and F_2 are two fractions, on which we are making the improvement now. On which we are making improvement S_1 for F_1 , S_2 for F_2 . So, this is the part where no improvement can be made. And this is a part on which we are making improvement S_1 . And this is a part where we are making improvement S_2 . So, earlier it was taking F_1 and F_2 and it has got reduced, and now we are getting F_1 / S_1 and F_2 / S_2 .

So, similarly what will be the speedup? The speedup formula will be same. So, this will be the total improvement when we are saying that we have multiple enhancements. We are not making enhancement on a single portion, rather we are making enhancements on both the portions.

(Refer Slide Time: 16:11)

- In the calculation as shown, it is assumed that F_1 and F_2 are disjoint.
 - S_1 and S_2 do not apply to the same portion of execution.
- If it is not so, we have to treat the overlap as a separate portion of execution and measure its speedup independently.
 - F_{1only} , F_{2only} , and $F_{1&2}$ with speedups S_{1only} , S_{2only} , and $S_{1&2}$

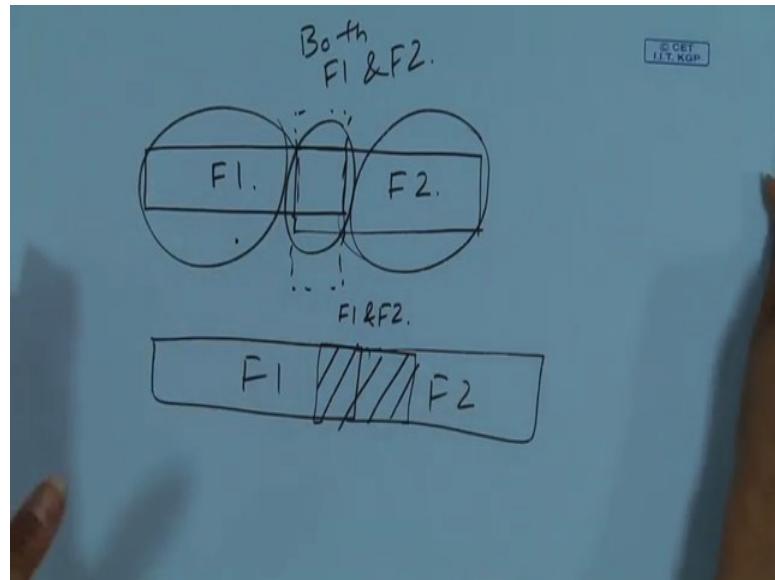
$1 - F_{1only} - F_{2only} - F_{1&2}$	F_{1only}	$F_{1&2}$	F_{2only}
$1 - F_{1only} - F_{2only} - F_{1&2}$	F_{1only} / S_{1only}	$F_{1&2} / S_{1&2}$	F_{2only} / S_{2only}

$$\text{Speedup} = \frac{1}{(1 - F_{1only} - F_{2only} - F_{1&2}) + F_{1only} / S_{1only} + F_{2only} / S_{2only} + F_{1&2} / S_{1&2}}$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us say in the calculation as shown it is assumed that F1 and F2 are disjoint. So, in this particular case F1 and F2 are disjoint. And what we are doing that we are making improvement on this part only and this part only, but now let us say there can be a situation like this.

(Refer Slide Time: 16:38)



So, in this situation what happens? This is your part, with says this is your F1 part and this is your F2 part. And this part is having both F1 and F2. So, this enhancement can be taken care of separately, this part enhancement can be taken care separately, this part this part, and this part separately. Earlier it was having disjoint. So, earlier the case was this was one part this was another part. So, this is considered F1 and is considered F2. No common part was there, but now we are extending F1 till here and we are extending F2 till here. So, this has become now the common part, which contains both F1 and F2. Now under such scenario, let us see how we will calculate the speedup. So, S1 and S2 do not apply to the same portions of execution.

If it is not so, we have to treat the overlap as a separate portion of execution, and measure its speedup independently. Now this is very true that, this is one part, this is one part. And this should be also considered another part. And we have to calculate speedup separately. Although, we have a common part of both F1 also is here and F2 is also here, but we have to take care of this. If it is not so we have to treat the overlap as a separate portion as it is not disjoint. So, we have to take care this part, this part, and this part

separately. So, in this case what will happen? This is the portion where no change will happen, and these are the 3 portions where you are making some changes. Where this part is F1 only, this part is F2 only, and this part is both F1 and F2.

So, what we are doing? F1 only with S1 only, F1 and F2 with S1 and S2, and F2 only with S2 only. It was initially this much, after this reduction it has become this much. Initially this much after reduction it has become, finally the speedup equation can be said as this. So, we are taking into consideration all the parts, plus F1 only divided by S1 only, plus F2 only divided by S2 only, and then take into consideration both F1 and F2 divided by S1 and S2.

(Refer Slide Time: 19:55)

- General expression:
 - Assume m enhancements of fractions F_1, F_2, \dots, F_m by factors of S_1, S_2, \dots, S_m respectively.

$$\text{Speedup} = \frac{1}{(1 - \sum_{i=1}^m F_i) + \sum_{i=1}^m \frac{F_i}{S_i}}$$

Now, this is the general expression just now what we have said. So, we assume m enhancements of fractions. So, in that case what will happen? The overall speedup will be 1 divided by $(1 - \text{summation of all these } F_i\text{'s})$. How many enhancements will be there? Those many F_i 's will come here. Plus, summation of F_i divided by S_i . So, this is a general expression of speedup when multiple enhancements are taken into consideration.

(Refer Slide Time: 20:45)

The slide is titled "Example 3". It contains a bulleted list of points about a memory system, a diagram of the memory hierarchy, and a "Solution" section with a speedup formula.

- Consider an example of memory system.
 - Main memory and a fast memory called cache memory.
 - Frequently used parts of program/data are kept in cache memory.
 - Use of the cache memory speeds up memory accesses by a factor of 8.
 - Without the cache, memory operations consume a fraction 0.40 of the total execution time.
 - Estimate the speedup.

Solution

$$\text{Speedup} = \frac{1}{(1 - F) + F/S} = \frac{1}{(1 - 0.4) + 0.4/8} = 0.91$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Now, let us take an example. Although we have not discussed about cache memory yet, I can just tell you that cache memory is a fast memory that sits between CPU and main memory. And it is used to enhance the overall performance of the memory, i.e. overall speedup of the memory. So, consider an example of memory system. Where main memory and a fast memory called cache memory is used. So, this is your cache memory, this is your main memory. Frequently used parts of the program or data are kept in cache memory.

Suppose, use of cache memory speeds up memory access by a factor of 8. So, whenever cache memory is used then the speedup of memory is by a factor of 8. Without cache the memory operation consumes a fraction 0.40 of the total execution time. So, what will be the speedup? So, in this case as without cache memory it was taking this much. Now using cache memory, you can actually make us speedup on this 0.4 by a factor 8; and the remaining will not change. So, if you put this in the formula of speedup you will be getting 0.91.

(Refer Slide Time: 22:41)

Example 4

- Now we consider two levels of cache memory, L1-cache and L2-cache.

Assumptions:

- Without the cache, memory operations take 30% of execution time.
- The L1-cache speeds up 80% of memory operations by a factor of 4.
- The L2-cache speeds up 50% of the remaining 20% memory operations by a factor of 2.

We want to find out the overall speedup.

JIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NPTEL

Let us take another example here where we consider 2 levels of cache memory where we see that the first level is called L1 cache. And the next level is called L2 cache. The assumption here is, without the cache memory, the memory operations take 30% of execution time. When L1 cache is used it speeds up 80% of the memory operation by a factor of 4.

Now, out of total execution time only memory operation is 30%. Now out of that 30%, if L1 cache is used 80% of the memory operation improves by a factor of 4. And the L2 cache speeds of 50% of the remaining 20% of the memory operation by a factor of 2. So, there are 2 things out of total execution time, the memory operation is taking 30%. Out of this 30%, now 80% of the memory operations are found in L1 cache. And 20% of the memory operation is found in L2 cache.

So, when L1 cache is used 80% the memory operation is improved by a factor of 4. And when L2 cache is used, 50% of the memory operation speeds up by 20% memory operation by a factor of 2. Because this is 80% to remaining 20% of the total 30% is 50%. So, L2 cache speedup is 50%. We want to find out the overall speedup. Let us see how we can find out the overall speedup.

(Refer Slide Time: 25:02)

• Solution:

- Memory operations = 0.3
- $F_{L1} = 0.3 * 0.8 = 0.24$
- $S_{L1} = 4$
- $F_{L2} = 0.3 * (1 - 0.8) * 0.5 = 0.03$
- $S_{L2} = 2$

Speedup

$$\frac{1}{(1 - F_{L1} - F_{L2}) + \frac{F_{L1}}{S_{L1}} + \frac{F_{L2}}{S_{L2}}} = \frac{1}{(1 - 0.24 - 0.03) + \frac{0.24}{4} + \frac{0.03}{2}} = 1.24$$

Memory operation 0.3, that is 30%. So, F_{L1} will be when L1 is used; as L1 is used 80% of the time it will be $0.3 * 0.8 = 0.24$. And S_{L1} is 4 that is already given. Similarly, for F_{L2} that is 0.3, multiplied by it was 80%. So, this will be 20%, out of 20% it is 50% used. So, 0.5 that is coming on to 0.03 and S_{L2} will be 2. Now we will put this value here and we are getting a speedup of 1.24.

So, we came to the end of lecture 16 and which will end the week 3 lectures. So, in this week we have actually seen how we can calculate performance. How Amdahl's law is actually happening. And how it is helping us to determine that, how much speedup can be actually achievable when only a part you are actually making the improvement.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 17
Designs Of Control Unit (Part 1)

Welcome to week 4. In this week we shall be looking into the design of control unit. So, till now what we have seen? We have seen how an instruction gets executed; what are the hardware blocks that are required for the execution of the instruction. And for executing that instruction, basically some steps are required. That we are saying that, the content of the PC will go to MAR. And then that particular content in MAR will hit the memory through the address bus, and then we will get the data. And then something else will happen. So, for any instruction certain steps are required to be executed. And for executing those steps we require some hardware, like the registers, the ALU, and other interconnecting blocks, other registers like IR, PC and all other.

So, in the design of control unit, we will be seeing what are the signals that needs to be generated for executing an instruction. So, we will be seeing this in course of the lectures that are there in week 4.

(Refer Slide Time: 01:54)

Instructions

- Instructions are stored in main memory.
- Program Counter (PC) points to the next instruction.
 - MIPS instructions are 4 bytes (32 bits) long.
 - All instructions starts from an address that is multiple of 4 (last 2 bits 00).
 - Normally, PC is incremented by 4 to point to the next instruction.

12					
8					
4					instruction word
0					instruction word

IIT Kharagpur | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

We already know that instructions are stored in main memory. The program counter points to the next instruction. So, here we have an instruction, next the PC points to this

location, then the next one, and so on. So, each time an instruction is fetched, the PC gets incremented to the next one, such that once the execution of this particular instruction is completed, the next instruction can be fetched, and again the PC get incremented and so on. Generally, MIPS instructions are all 4 bytes or 32 bits long. All instruction starts from an address that is some multiple of 4. Last two bit is will be 0. And normally the PC gets incremented by 4 to point to the next instruction. We know about this particular thing from the very beginning.

(Refer Slide Time: 03:03)

Binary Number System

- Two digits: 0 and 1.
 - Every digit position has a weight that is a power of 2.
 - *Base or radix* is 2.
- Examples:
$$110 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$
$$101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Let us recall that in binary number system we have 2 digits 0 and 1. This is how it is represented; both in binary and in decimal; addressing a byte in memory. We know that memory is byte addressable.

(Refer Slide Time: 03:19)

Addressing a Byte in Memory

- Each byte in memory has a unique address.
 - Memory is said to be *byte addressable*.
- Typically the instructions are of 4 bytes, hence the instruction memory is addressed in terms of 4 bytes (word length = 32 bits).
- When an instruction is executed, PC is incremented by 4 to point to the next instruction.

<img alt="Speaker icon" data-bbox="974

execute that particular instruction. And then we move on to the next one, from the value pointed by PC. And again we do the same thing. Again we fetch, we decode and we execute.

(Refer Slide Time: 04:51)

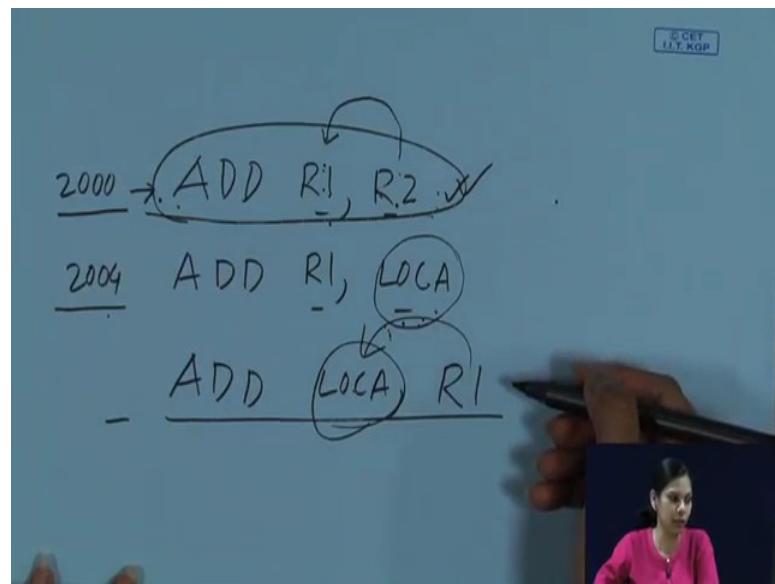
The Fetch-Execute Cycle

- Fetch the next instruction from memory.
- Decode the instruction.
- Execution Cycle:
 - Gets data from memory if needed (data not available in the processor)
 - Perform the required operation on the data.
 - May also store the result back in memory or register.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, this is the fetch-execute cycle. We fetch the next instruction from memory, decode the instruction and execute the instruction. In the execution process what we do. It might happen we have various kinds of instruction. Let us say, we can have an instruction which is ADD R1,R2. We can also have an instruction called ADD R1,LOCA. In this particular case we can say the operands are present in the processor register as well.

(Refer Slide Time: 05:14)



In this case, we need to bring a data from memory and then we can process this data. So, in this case no further memory access will be required, but for this particular case memory access will be required. So, in the execution cycle once we decode the instruction, we know that this is the addressing mode, and so on and so forth. Now we need to execute it. So, we get the data from memory, if needed data is not available in the processor. We perform the required operation on data, and may also store the result back in memory or in register as and when it is required.

So, in these two particular cases, we need not have to store back in memory. But in this particular case, it might happen we need to store the result back in memory. So, we first fetch this instruction, then we fetch this particular data, then we perform the operation, and finally, we store back in this particular memory location. So, this is the fetch-execute cycle.

(Refer Slide Time: 06:54)

Registers: IR and PC

- Program Counter (PC) holds the address of the memory location containing the next instruction to be executed.
- Instruction Register (IR) contains the current instruction being executed.
- Basic processing cycle to be implemented:
 - Instruction Fetch (IF)
 $IR \leftarrow Mem[PC]$
 - Considering the word length of the machine is 32 bit, the PC is incremented by 4 to point to the next instruction.
 $PC \leftarrow PC + 4$
 - Carry out the operations specified in IR.

The register we already know about this is the instruction register (IR) and this is program counter (PC). Program counter holds the address of the memory location containing the next instruction to be executed. Instruction register contains the current instruction being executed. So, if this is my current instruction, then PC value is 2000. Next PC will get incremented and it will have 2004, and so on. So, the PC will point to the next instruction. Once we fetch this instruction, IR will contain this particular instruction.

So, instruction register contains the current instruction being executed. Basic processing cycle to be implemented is after PC points to the memory location where it is. So, memory location of the PC will be transferred to IR. Now IR contains the current instruction. Considering the word length of the machine is 32 bit, the PC is incremented by 4 to point to the next location. Now PC will have $PC + 4$. Then we carry out the operation specified in IR. So, whatever is specified in IR, we decode that and we then need to perform the specific operation.

(Refer Slide Time: 08:24)

Example: Add R1, R2

Address	Instruction
1000	ADD R1, R2
1004	MUL R3, R4

a) PC = 1000
b) MAR = 1000
c) PC = PC + 4 = 1004
d) MDR = "ADD R1, R2"
e) IR = "ADD R1, R2"
(Decode and finally execute)
f) R1 = R1 + R2

IIT KHARAGPUR | **NPTEL ONLINE CERTIFICATION COURSES** | **NPTEL**

Let us take an example. ADD R1,R2 and MUL R3,R4. So, PC initially contains 1000. MAR contains 1000. PC now contains 1004; MDR will contain this entire instruction. IR will also contain this instruction. Finally, it will get decoded and executed. And then after adding these two, the result is stored back in R1.

Now, you see these are some steps that are happening. We can see this in terms of some values because we know that this particular address is having this value, now it will go to MAR. But, if we require our computer to do this, some signals need to be generated in proper sequence, to perform this particular operation depending on certain hardware that is present.

(Refer Slide Time: 09:26)

Requirement for Instruction Execution

- The necessary registers must be present.
- The internal organization of the registers must be known.
- The data path must be known.
- For instruction execution, a number of micro-operations are carried out on the data path.
 - May involve movement of data.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, what is the requirement for instruction execution? The necessary registers must be present. We require to have the registers for that operation. Internal organization of the registers must be known. This is very important and this is what we will be looking into in this particular lecture, that internal organization of the registers.

What do you mean by internal organization? I say that, you will be having registers, you will be having an ALU. You will be having other registers like PC, MAR, MDR. How these are connected? We need to know internally how these are connected. MDR and MAR are connected to the memory bus. But we need to know the internal structure of the organization of the hardwares, such that we must know that how the registers are connected, how the ALU is connected, to perform an operation in ALU what needs to be brought in, how it should be brought in and everything

So, we need to know a complete picture of what is there inside. So, the internal organization of the registers must be known. The data path must be known, that is, we will be seeing what is data path. So, for instruction execution a number of micro operations are carried out on the data path, may involve movement of data; that means, when we are performing ADD R1,R2; how this data is actually moving? So, all these operation how it is happening we need to know. The steps that are required for execution are known as micro operations. So, micro operations should be carried out on the data path provided.

(Refer Slide Time: 11:59)

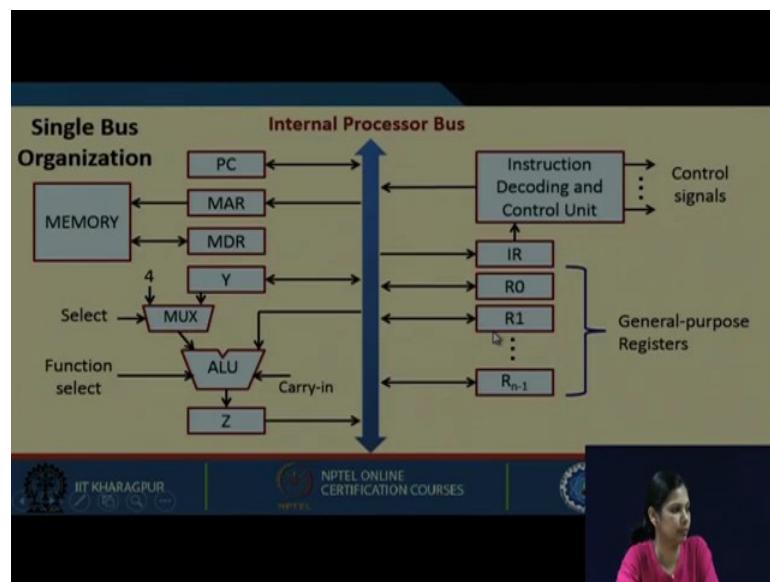
Kinds of Data Movement

- Broadly three types:
 - Register to Register
 - Register to ALU
 - ALU to Register
- Data movement is supported in the data path by:
 - The Registers
 - The Bus (single or multiple)
 - The ALU temporary Register (Z)



So, let us see the kinds of data movement. Broadly it can be register to register, it can be register to ALU, or ALU to register. We will be seeing all these kind of transfers in course of time. So, the data movement are supported by the data path. And the data path contains what the registers, the bus through which the data will move, the ALU, and of course, some of the temporary registers. Some temporary registers are needed for this. So, all these together are supported in the data path. Coming to the single bus organization.

(Refer Slide Time: 12:38)



This is a very simple single bus organization that we are showing, which is internal to the processor bus. And now see, these are the buses. This is the data bus and this is the address bus, which are connected to the memory, through these two registers MAR and MDR. In this internal processor bus we have PC. So, data from this bus can come in here. And from PC also the data can be available in this bus. We have MAR and MDR.

So, this line is missing. So, there will be a connection, between internal processor bus as well, and then this connection will be there. So, a two-way connection will be there. So, data from this bus can also come into MDR, and from MDR the data can come into this processor bus. Now you see this ALU. This ALU performs the required operation. And there are two inputs of the ALU and one output. We see that one input of the ALU is directly coming from the bus. So, whatever data is there in the bus, can be directly connected to this input of the bus. If we say this is A input and this is B input, in this B input we can see that it is available. And another data is coming through this Y register. And from Y there is a MUX. Now see that MUX is selecting either the output of Y or it is selecting this 4. Why this 4 is required? We will be seeing little later.

But let us understand for the moment, that when this select line is 1, either we select 4 if the select line is 0, or we select Y depending on how you have implemented it. So, at a time either 4 comes into ALU, or Y comes into ALU. After any particular function that is performed by ALU, the data is transferred to Z register, and from this Z register the data can be available in the internal processor bus; and from this internal processor bus now the data can move to any of these registers. This is the IR, and this is the instruction decoding and control unit. Instruction decoding and control unit is required to generate the control signals. So, ultimately this unit will be generating the control signals necessary to execute an instruction.

(Refer Slide Time: 16:35)

Single Internal Bus Organization

- All the registers and various units are connected using a single internal bus.
- Registers R_0-R_{n-1} are general-purpose registers used for various purposes.
- Y and Z are used for storing intermediate results and never used by instructions explicitly.
- The multiplexer selects either a constant 4 or output of register Y.
 - When PC is incremented, a constant 4 has to be added.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, this is our single internal bus organization. So, let us see some of the features that I already discussed. This is the single internal bus organization. All the registers and various units are connected using a single internal bus. We have only one bus through which it is connected. Registers are R_0 to R_{n-1} . So, we have n general-purpose registers used for various purposes. Y and Z are used for storing intermediate results. The intermediate result of any operation is stored in these registers, and they are never used by an instruction.

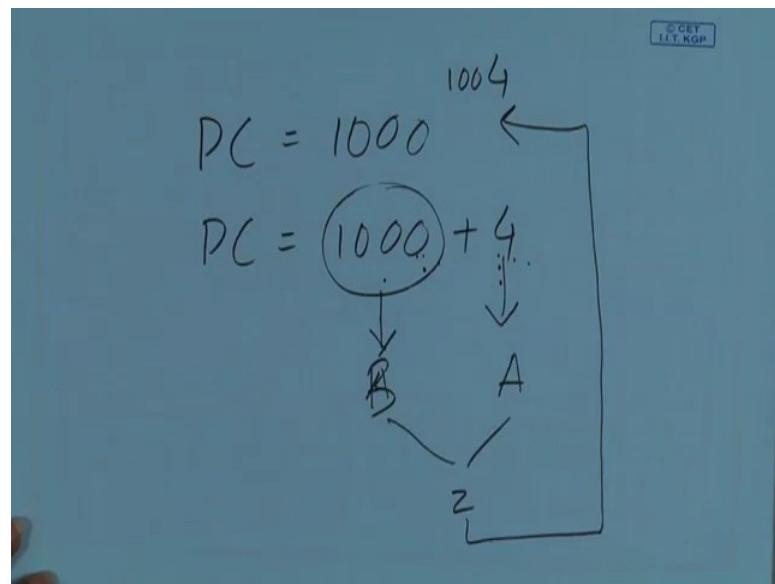
(Refer Slide Time: 17:31)

© CET
I.I.T KGP

ADD R1, (Y)
ADD R1, R2

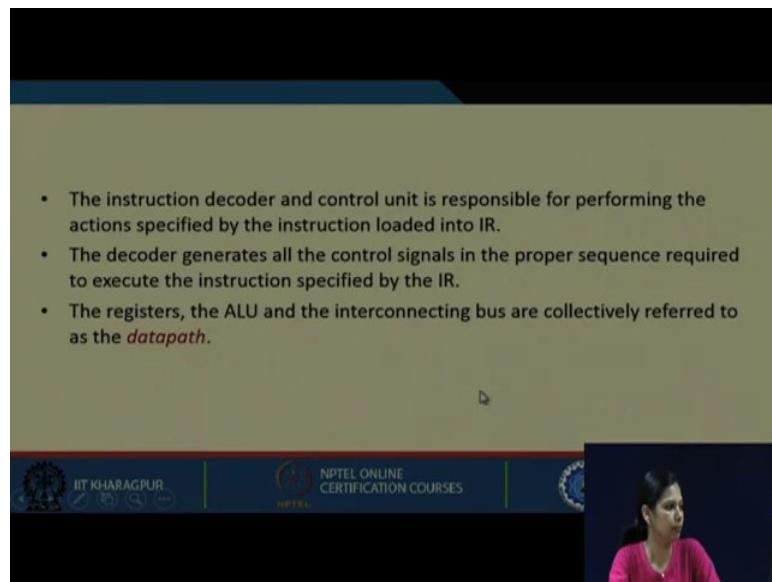
This means, we will never see that we are doing something like ADD R1,Y. We will never do that. We will only have ADD R1,R2 etc, which is the general-purpose register or memory location. The MUX selects either a constant 4 or the output of register Y. When PC is incremented a constant 4 has to be added. Now understand this. What happens, when we see that this is my PC? PC is now 1000.

(Refer Slide Time: 18:25)



Now, the PC needs to get incremented. So, you have to do $1000 + 4$. How will you do this? We cannot simply do this; we need a circuit to do this. And for doing this, this input must come into one of the inputs of ALU. So, if it comes to any one input of the ALU, then we can add 1000 plus 4, and then the result can be stored in Z. And from there it can be again put it in PC. So, it becomes 1004 again. So, this is how it can be done. So, when PC is incremented a constant 4 has to be added.

(Refer Slide Time: 19:20)



The instruction decoder and control unit is responsible for performing the action specified by the instruction loaded into IR. Now once the instruction is fetched from the memory, it comes through MDR, and then it goes to IR using that single bus. Once it is loaded in IR, it is the responsibility of the decoder unit to decode that particular instruction. And then it generates whatever needs to be done; if it has to bring the data from memory again it will do the required operation, if the data is already present in the processor register, then it has to add it or multiply it with whatever action is specified it needs to be done.

So, the instruction decoder in the control unit is responsible for performing the action specified by the instruction loaded into IR. The decoder generates all the control signals in proper sequence required to execute the instruction specified by IR. Now the decoder decodes the instruction. After decoding the instruction it generates the control signals that are required for that particular instruction in a proper sequence. Now what is data path then? The registers, the ALU, and the interconnecting bus are collectively referred to as the data path; that means, though this particular path, the data are moving for performing the operation.

So, for performing the operation it has to come to ALU. Then from ALU it has to again go to some register. So, how it is going? The registers are involved, the ALU is involved,

and the connecting bus through which the data is moving. So, data are moving through all these places. Collectively this is referred to as data path.

(Refer Slide Time: 21:40)

Kinds of Operations

- Transfer of data from one register to another.
MOVE R1, R2
- Perform arithmetic or logic operation on data loaded into registers.
ADD R1, R2
- Fetch the content of a memory location and load it into a register.
MOVE R1, LOCA
- Store a word of data from a register into a given memory location.
MOVE LOCA, R1

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, now let us see what are the kinds of operations that are performed. Transfer of data from one register to another. Let us say moving a data from R2 to R1 is required. Perform arithmetic or logic operation on data loaded into register. Let us say the data is loaded in R1 and R2; all we need to do is that we perform such an operation, and store it back here. So, here also this is a kind of operation that is required. Fetch the content of memory location and load it into register, basically load.

So, we are loading a data from this memory location and we are storing it in R1. Or store a word of data from a register into the memory location. So, we are storing a word that is R1 with whatever value is stored in R1 into LOCA or memory location. So, this is for load, this is for store. So, these are the various kinds of operation that we can have.

(Refer Slide Time: 22:48)

The slide has a dark blue header and a light beige main content area. At the top center, it says 'Three Bus Organization'. Below that is a bulleted list of features. At the bottom, there are three logos: IIT Kharagpur, NPTEL, and NIT Meghalaya, along with their respective names.

- A typical 3-bus architecture for the processor datapath is shown in the next slide.
 - The 3-bus organization is internal to the CPU.
 - Three buses allow three parallel data transfer operations to be carried out.
- Less number of cycles required to execute an instruction compared to single bus organization.

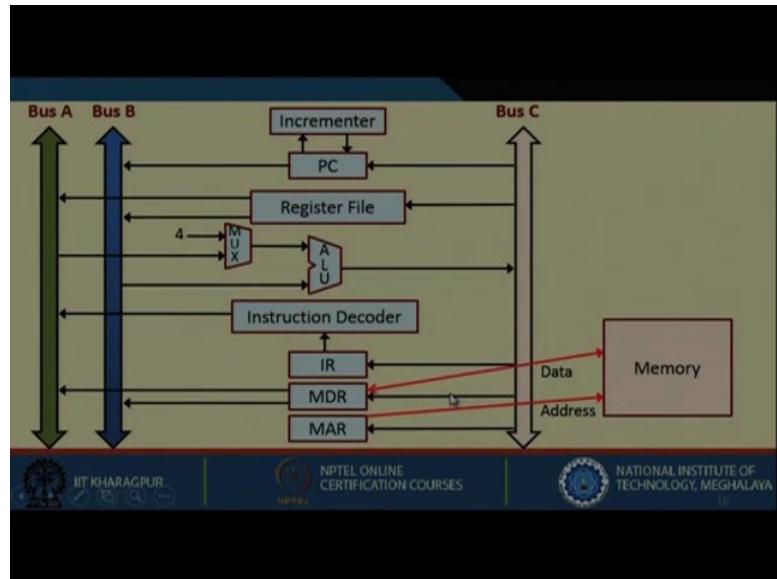
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us come to 3-bus organizations. Now in the previous case you have seen a single internal bus organization. And in that single internal bus organization we will be seeing in later lectures that only at a time, of particular value can be available in the bus. And that particular value can go to any number of registers. But at a time only one data can be available in that bus. If we want to make more data available then, what we need to do? One possible way of doing so is having multiple-bus structure.

So, what happens in multiple-bus architecture? In multiple-bus architecture we have multiple internal buses inside a processor. The MDR and MAR will be connected to the same system bus. But, internally we will not have a single bus. We will see that a single bus will restrict some operation to be done parallelly. If you want to perform some operation to be done parallelly, we require multiple-bus architecture.

So, these are just some of the features. A 3-bus organization is internal to the CPU, as I said, we will be looking into a bus organization which is internal to the CPU. We have already seen a bus organization, which is a single-bus organization. Now we will be seeing a 3-bus organization. The 3 buses allow 3 parallel data transfer operations to be carried out. Less number of cycles in turn will be required to execute an instruction, compared to single bus organization. We will be looking into this with examples later.

(Refer Slide Time: 25:05)



Now, let us see this particular multibus organization that is a 3-bus organization. In this 3-bus organization let us see what we have. This is a register file. In this register file using VLSI technology what we can do is that, we can read multiple data. But we can write in one data into the register file. So, two registers can be read at a time because we have two buses. And the data from these register file is going to two different buses. But write can happen only once, and it is coming also from a different bus. PC is incremented by a different circuitry. That is an incrementor circuit, where PC will get incremented by 4.

Now, this is the ALU; the input of ALU is coming from two different buses. One is from bus A another is from bus B. The advantage we can get here is that, we can make available the data of R1 and R2 here, and if we perform that operation and both R1 and R2 can be present at the same time. And we can also perform this ADD operation. And we can also store back here at the same time. But in a single-bus organization, only one particular data will be available in the bus at a time. As for multiple buses multiple data can be available.

This is an instruction decoder. So, after from MDR the data will be available. And then this particular data will be moved to IR, and the instruction decoder will decode the instruction and specified operation will be carried out. MAR and MDR are connected to the address and data bus of the memory as well. So, this is a 3-bus organization. We have

seen single bus organization; we have seen multiple bus organization. We will be seeing in detail what is the advantage you get in course of time when we execute a particular instruction using these bus organizations.

So, now we have come to the end of this lecture, where we have discussed about the overall internal bus organization, how internally within the CPU the buses are organized.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture – 18
Design Of Control Unit (Part 2)

Welcome to the next lecture on control unit design, design of control unit part 2. We will continue from where we left in the previous lecture.

(Refer Slide Time: 00:28)

The slide has a title 'Organization of a Register' and contains the following text:

- A register is used for temporary storage of data (parallel-in, parallel-out, etc.).
- A register R_i typically has two control signals.
 - $R_{i,in}$: used to load the register with data from the bus.
 - $R_{i,out}$: used to place the data stored in the register on the bus.
- Input and output lines of the register R_i are connected to the bus via controlled switches.

On the right side of the slide, there is a diagram of a register R_i . It is represented by a blue rectangular box labeled 'Register R_i '. Two arrows point to this box: one from the top labeled $R_{i,in}$ and one from the bottom labeled $R_{i,out}$. Both arrows pass through small red circles containing a cross symbol, which represent controlled switches. A large green double-headed vertical arrow is positioned to the right of the register box, indicating bidirectional data flow between the register and the bus.

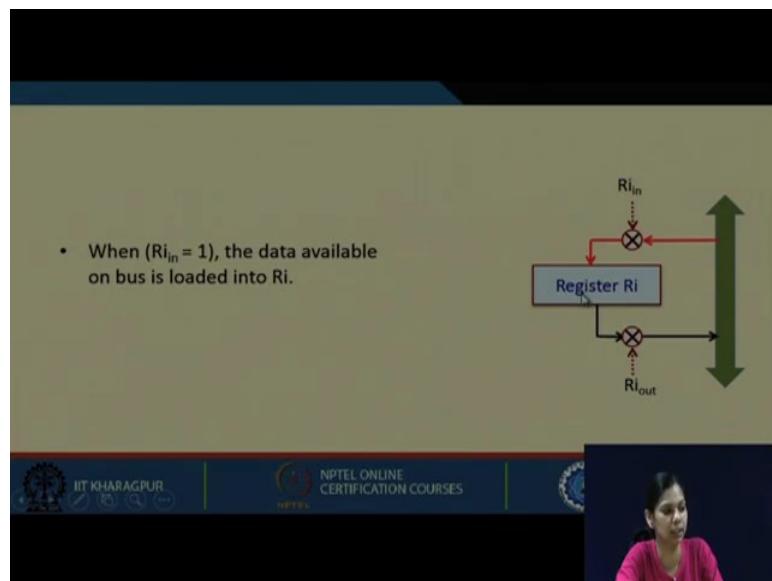
So, in the previous lecture we have seen a single-bus architecture, we have seen also a multi-bus architecture. Now, we will be specifically looking into how we can perform an operation. So, for performing an operation what is actually required? Let us see the organization of a register. A register is used for temporary storage of data. So, in this particular register, we actually store data. And now we are saying that we require the data when we are doing ADD R_1, R_2 . We require the data from R_1 to be available in ALU; we required the data R_2 to be available in another input of the ALU. In such cases, the data from this register needs to be moved out through this bus and it reaches the ALU, and in the same way after the operation is performed the data must be brought into a particular register.

So, a register R_i typically has two control signals what is that $R_{i,in}$ and $R_{i,out}$. $R_{i,in}$ means something is coming into R_i . So, data available from the data bus will be available to

register R_i through the control signal $R_{i,in}$. Next we also sometime require the data from register to come out of this bus and then go to somewhere else. So, what is the control signal required for that? The control signal required for that is $R_{i,out}$. So, $R_{i,out}$ will place the value from this particular register R_i into the bus. So, two signals are very much important $R_{i,in}$ that will take the data from bus into this particular register, $R_{i,out}$ that is used to place the data stored in the register on the bus. So, from the register, the data will be put into the bus. So, from the arrow you can easily make out.

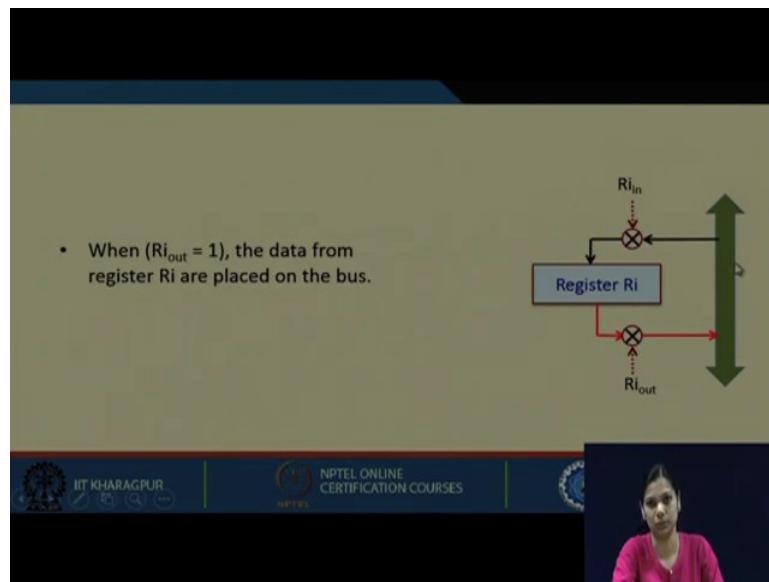
Input and output lines of the register R_i are connected to the bus via control switches. So, basically these are some switches. So, these switches are made on and off to do the particular function. So, what we have to do we have seen that $R_{i,in}$ is used to load the value here, and $R_{i,out}$ is used to take the value from the register and put it in this bus.

(Refer Slide Time: 03:43)



Now, let us see when $R_{i,in}$ is 1; that means, $R_{i,in}$ is active, the data from bus will be put into the register.

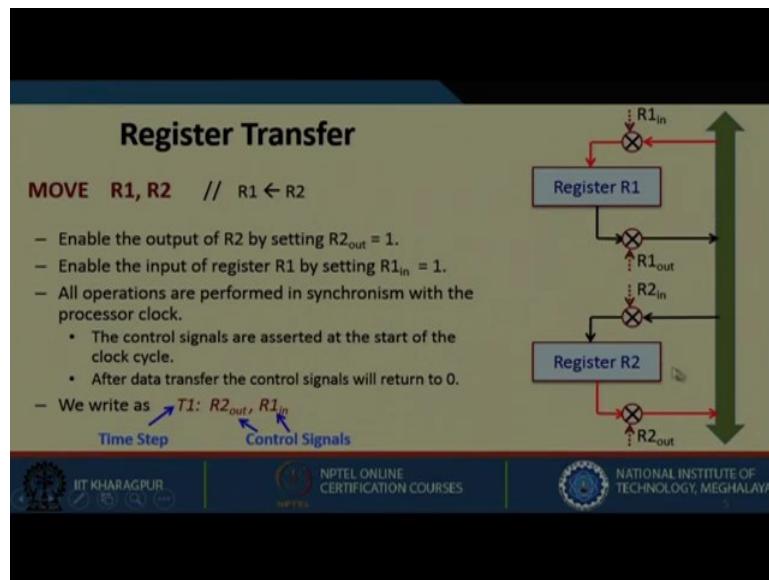
(Refer Slide Time: 04:04)



- When ($Ri_{out} = 1$), the data from register Ri are placed on the bus.

Similarly, what happens when Ri_{out} is 1? The data from register Ri will be made available in the bus. So, the data from registered Ri are placed into the bus. Now, you see what will happen when this is 0, there will be no change. So, this signal will not get activated, so nothing will happen neither in nor out.

(Refer Slide Time: 04:42)



Let us see this register transfer. So, similarly for individual registers we will be having some kind of signal like this, let say this is register R1 and this is register R2. So, for this register, one signal will be there $R1_{in}$, that will make the data available from bus to this

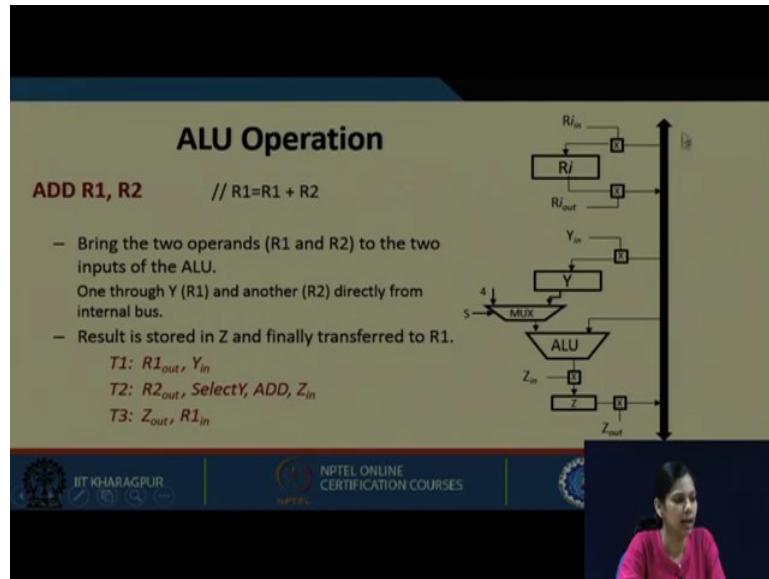
register; and they will be another signal $R1_{out}$ that will make available the data from register to the bus. Similarly, for R2 it is also there.

For register transfer - MOVE R1,R2 that means, content of R2 should be available into R1. Firstly, what we need to do is that content of R2 must be available in the bus. So, if I want the data of R2 to be available in the bus, which signal is required; $R2_{out}$. So, $R2_{out}$ is required to be 1. So, how do we do it, enable the output of R2 by setting $R2_{out} = 1$. Now, enable the input of register R1 by setting $R1_{in} = 1$. Now, we also want that the data from the bus should be made available in to register R1. In that case what we need to do the signal $R1_{in}$ should be 1. So, for outputting $R2_{out}$, we require $R2_{out}$ for making available; inside R1, we require $R1_{in}$, $R2_{out}$, $R1_{in}$.

All operations are performed in synchronism with the processor clock. So, this particular register transfer is within the processor which is much more faster. The control signals are asserted at the start of the clock signal; after the data transfer the control signal will return to 0. So, when the control signal will return to 0, what will happen? When the control signal is 0, no data will be available either on the bus, or no data will be available in the register also. Only when that signals are activated the data from either bus comes in into a particular register, or from a particular register the data goes into bus. So, after the data transfer ,as I said the control signal will return to 0.

So, to perform this what control signals are required in a particular step? So, at T1 we can perform $R2_{out}$, $R1_{in}$; $R2_{out}$ will make available the data of R2 into the bus; and $R1_{in}$ at the same time will bring the data from the bus into R1 that is happening in a time step T1. These are called control signals and this is the time step where this particular control signals are activated.

(Refer Slide Time: 08:21)



Let us see an ALU operation. For performing an ALU operation ADD R1,R2 what needs to be done. We need to make available both R1 and R2 in these two places; one is directly connected to the bus and another value is coming from this Y register through this MUX. We need to add the two register values, we have those registers in place; one of the register output needs to be made available into one of the inputs of the ALU; another register output needs to be available in another ALU input. And we must understand one thing that we have a single bus. So, only one transfer can take place at a time.

So, the first thing, we bring the two operands R1 and R2 to the two inputs of the ALU as I just now said. So, one will be coming through this Y, and another is directly coming from the internal bus, this is how the organization is made. So, this is how the data transfer will also be done. And finally, after this ALU operation, the result will be stored in Z, and then it will be made available through this control signal into the bus, and then it will move to R1.

Now, let us see what are the various time steps and what control signals are required to be activated. So, in the first go let us make R1 available through Y. If we make R1out and Yin, the data will be available in Yin. So, data from R1 through this bus comes in and it goes to Y using the control signal R1out and Yin. Now, we have to make another signal available, that is R2out, if we just do R2out we see that it is directly connected, it

is not connected through any bus. So, at any point of the time whatever is out here is directly available into one input of the ALU. So, when we do R2out, this particular value is directly available here. And if we do select Y, then the Y value through this MUX will come in and will be available to this input of the ALU.

So, what we are doing here? We are doing R2out which is directly connected here, we are selecting Y using this MUX. So, the value which is there in Y, it was R1, will come in through another input of ALU and then we can perform the operation because both the inputs are now available. We do add and after adding what we do, we do Zin, because we can directly make this input into this through this particular signal that is directly connected to the ALU.

So, at T2 we are performing R2out, SelectY, ADD and Zin. So, now using this particular operation the value is available in Z. Now, Z contains the result, but we need to store the result in R1. So, for storing the result in R1, what we need to do? Now my value is available in Z, I do Zout. So, the data will be available in the bus and I do an R1in. So, it will go into R1. So, these are the three times steps that are required to perform this particular ALU operation.

(Refer Slide Time: 12:58)

The slide has a dark blue header and a light blue footer. The title 'Fetching a Word from Memory' is in bold black font at the top. Below the title is a bulleted list of steps:

- The steps involved to fetch a word from memory:
 - The processor specifies the address of the memory location where the data or instruction is stored.
 - The processor requests a read operation.
 - The information to be fetched can either be an instruction or an operand of the instruction.
 - The data read is brought from the memory to MDR.
 - Then it can be transferred to the required register or ALU for further operation.

The footer features the IIT Kharagpur logo, a search icon, and the text 'NPTEL ONLINE CERTIFICATION COURSES'. On the right side of the footer, there is a small video frame showing a woman speaking.

Let us move on. Now, we also need to fetch a word from the memory. So, for fetching a word what is required let us see that. These are the steps that are involved to fetch a word from memory. The processor specifies the address of the memory location where the

data or instruction is stored. The processor request for a read operation for fetching their reading actually; the information to be fetched can be either an instruction or it can be a data, accordingly it is fetched.

The data read is brought from memory to MDR. We already know all these things, we have already discussed in previous classes. So, when we fetch an instruction or data, we put the address in MAR, we activate the read control signal. Once it is read from the memory, it is brought it into MDR. And now from MDR that is an internal processor register, we can move the data or instruction wherever we want it. Then it can be transfer it to the required register or ALU. It can either go to IR if it is an instruction; or it can go to ALU for further processing if it is a data.

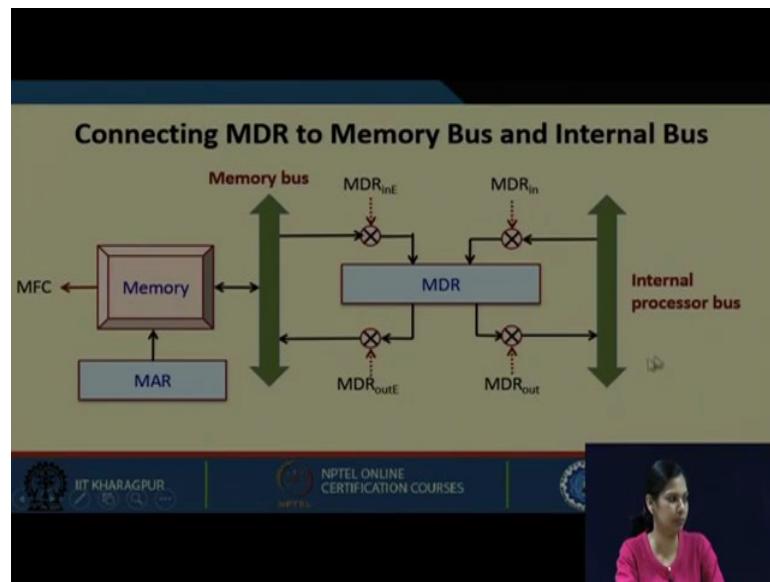
(Refer Slide Time: 14:18)

Storing a Word into Memory

- The steps involved to store a word into the memory:
 - The processor specifies the address of the memory location where the data is to be written.
 - The data to be written is loaded into MDR.
 - The processor requests a write operation.
 - The content of MDR will be written to the specified memory location.

Now, what happens for storing a word into memory? For storing the word the steps involved is first the processor specifies the address of the memory location where the data is to be written. Now, here the thing is different; you have to load the particular address of the memory where the data is to be written into MAR, and then you have to load the data that you want to write in that particular location into MDR. So, you have the data in MDR you have the address where to be written in MAR and the processor request for a write operation. So, the content of MDR will be written to the specified memory location.

(Refer Slide Time: 15:06)



Now, let us see this is the connection of MDR. Now, you already know that MDR is a register that is connected to the internal processor bus and it is also connected to the memory bus of the system. So, in this particular case, we require four switches, one as it is connected to the internal processor bus when we do MDR_{out} the value available from MDR will be made available through MDR_{out} into this internal processor bus. And when we are doing MDR_{in} then the data from internal processor bus will be made available to MDR. As MDR is also connected to an external memory bus we require two more signals MDR_{inE} , MDR_{outE} , E stands for external. So, MDR_{inE} means data available in the memory bus will be made available to MDR, and the data available in MDR can be transferred to the memory bus using MDR_{outE} . So, whenever we are doing MDR_{outE} then the data from MDR may be available in the memory bus. So, these are the control signals that are required.

(Refer Slide Time: 16:53)

- Memory read/write operation:
 - The address of memory location is transferred to MAR.
 - At the same time a read/write control signal is provided to indicate the operation.
 - For read the data from memory data bus comes to MDR by activating MDR_{inE} .
 - For write the data from MDR goes to memory data bus by activating the signal MDR_{outE} .

Memory read write operation. What happens here? The address of the memory location is transferred to MAR, as I have already said at the same time the read or write control signal is provided to indicate the operation. Once we put the address in MAR we need to also specify whether it is a read or write. For read MDR is not coming into picture initially, but for write you also need to write that particular data and for the read the data from the memory will come into MDR. So, for read the data, from memory data bus comes to MDR by activating MDR_{inE} as I just now said. For write the data from MDR goes to data bus by activating the signal MDR_{outE} . So, these are the signals that are required for communicating with the external memory bus.

(Refer Slide Time: 17:56)

- When the processor sends a read request, it has to wait until the data is read from the memory and written into MDR.
- To accommodate the variability in response time, the process has to wait until it receives an indication from the memory that the read operation has been completed.
- A control signal called *Memory Function Complete* (MFC) is used for this purpose.
 - When this signal is 1, indicates that the content of the specified location is read and are available on the data line of the memory bus.
 - Then the data can be made available to MDR.

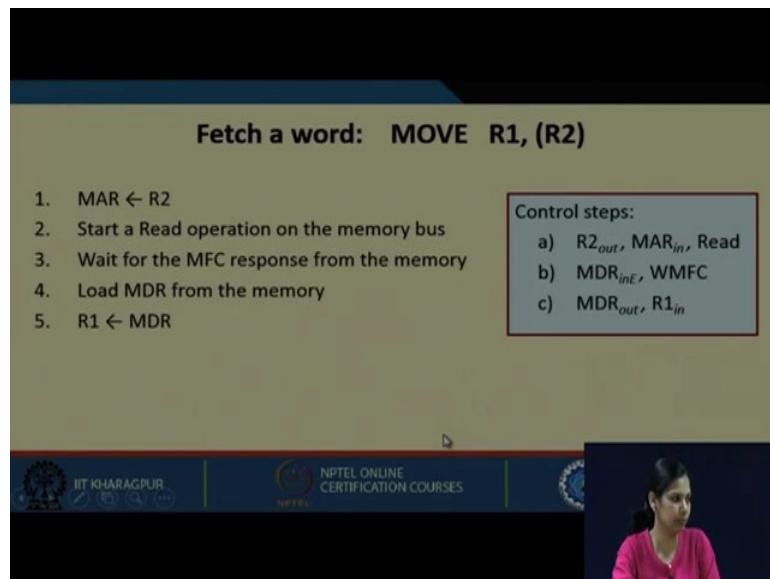
So, when the processor sends a read request it has to wait until the data is read from the memory and return into MDR. It is pretty obvious that the memory is slower than processor. Once the processor sends a read request it has to wait for some time till the data is available in MDR. To accommodate this variability in response time as there is no fixed response time that how much time it will be required, the processor has to wait until it receives an indication from memory that the read operation has been completed.

Please try to understand this. For a read operation, what we do is we put the address from where we need to read into MAR, and we activate the read control signal. Now, we really do not know at what time the data will be available, but the data will be available after sometime. So, till that time the process has to wait, but when the processor will know when a signal will be available from the memory to the processor that, the memory function has been completed. So, a signal is required to be sent from the memory to the processor to know that the data is now available in the particular register in that particular MDR. So, to accommodate this variability in response time the processor waits until it receives an indication from the memory. This indication is provided by a control signals known as memory function complete (MFC).

Whenever MFC is provided then we understand that now the data is available in MDR either way; for a write you have put the data into MDR, the address into MAR, and you have activated the write control signal; once the write is performed, it will inform. So,

when this signal is 1, it indicates that the content of specified location is read and are available on the data lines of the memory bus then the data can be made available to MDR because it is directly connected to MDR from the data lines of memory.

(Refer Slide Time: 20:42)



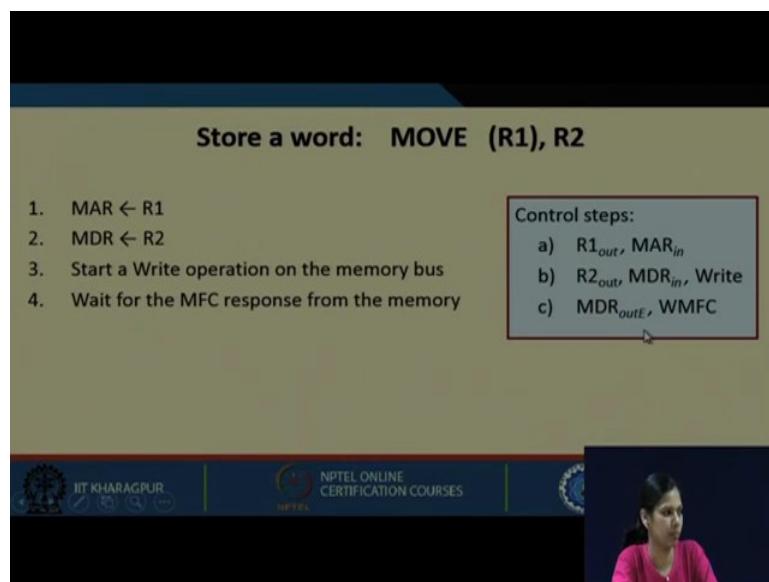
So, let us see what are the control steps or what are the time steps that are required for this particular operation. Fetch a word MOVE R1,(R2). (R2) is a memory location. So, what we need to do we have to put this particular memory address R2 value into MAR, and then we have to activate the read control signal. Once that is done then we need to wait for a signal that is MFC. Once that signal is available then from MDR the value can be move to R1.

So, let us see what are the steps that are followed. R2 goes to MAR we start a read operation on the memory bus, wait for the MFC response from the memory, load MDR from the memory and then from MDR it is transferred to R1. So, let us see the steps. First we do R2out and MARin, because in MAR the address should be present, and we activate the Read control signal. We perform all these things in one go, in one time step. After it is read the data will be available in MDR, how it will be available, it will be available through this signal MDRinE and we have to wait for MFC. Or the other way around if you think that when wait for MFC, this particular signal will get activated from the memory. Then automatically this MDRinE should be on made available such that the

data comes into MDR. And once the data is available in MDR it can be transferred to R2 through MDRout and R1in.

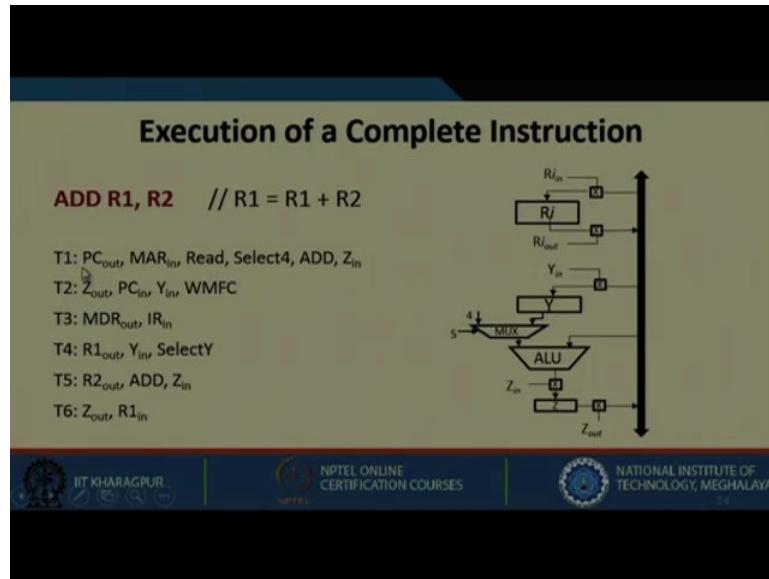
So, these are the following steps which will be required first we are making R2out putting it in MAR, using signal MARin, we activate the Read control signal. Then we make MDRinE, and we wait for MFC, when this signal comes and this is already on; that means, the data will be available in MDR. Once the data is available in MDR in this particular step, then we perform MDRout and R1in.

(Refer Slide Time: 23:23)



For storing a word, two things are required; one is that particular address where we need to store should be in MAR, and the data what we need to store should be in MDR. So, we are doing that. Here in this particular location R2 must be stored. So, R1 should be in a MAR and R2 should be in MDR because R2 is the data that needs to be stored by the address pointed by R1. So, R1 goes to MAR, R2 goes to MDR, start the Write operation on the memory bus, and wait for the MFC. So, in this case, we do R1out, MARin, R2out, MDRin. So, in MAR the address is present; in MDR the data is present, we perform write operation and then we do MDRoutE and wait for MFC.

(Refer Slide Time: 24:34)



Now, let us see the execution of a complete instruction. So, for executing a complete instruction what is required? We need to first fetch that instruction. In the same way we need to put the content of PC into MAR, we need to activate the Read control signal at the same time we have to increment the PC value, and then we have to wait for MFC because we are performing a Read. Once it is done and then the data that is present will be brought in through MDR, and then from MDR it will go to IR. So, we have to fetch the instruction in this particular case.

So, for fetching the instruction let us see what steps are required. So, we do PCout content of the PC stores the address of the next location that is to be executed. We do PCout, MARin, we activate the Read control signal. At the same time we Select4 because once you do PCout it is available in the B input. So, this particular diagram is not entire single bus architecture, but let say you have PC here in that. So, when you do PCout; that means, it is available in this bus and from this bus it is directly connected to one input of the ALU. So, it is available here. Once you do PCout, it is automatically available here and then you are putting that address in MAR activating the Read control signal.

And at the same time, we are doing Select4. If we select 4, then this particular input will have 4. So, this is having PCout and this is having 4, and we do an ADD and then after adding we put the value in Zin, so that means in Z now you have the incremented PC

value. What needs to be done to this PC value, this PC value needs to be transferred to PC again that we will be doing the next step. So, in this particular step we are putting the value of the particular PC MAR, we are activating the Read control signal, through the select we are selecting 4. Now, we put it in Z, and next we do Zout and PCin. Now, PC contains the increment value, that is $PC + 4$, we have done through ALU.

We are also doing Yin. Why we are doing Yin we will be seeing it for the case of branch instruction that it is required. And then as we have done Read operation here, in the next we need to wait for MFC. Once this is asserted and this is performed in T2, then the data is available in MDRout. Now, you see we are not giving MDRinE here. We wait for MFC. Once this is performed, MDRout we perform because the instruction pointed by PC is now available in MDR, we are transferring that to IR. We are doing MDRout and IRin, that means in these three steps what we have done we have just fetched the instructions from memory.

Now, what we need to do, we need to execute that instruction. For executing that instruction, in the same way, you have to bring R1 into one of the inputs of the ALU, and R2 into another input of the ALU. So, we have already performed these steps for an example previously. We do R1out, Yin. So, R1 is made available through the bus to Y and we do selectY. Similarly, we do R2out that is directly connected to ALU, and then we do ADD. As this is already selected, so we perform add here and then we perform Zin. So, the output of $R1 + R2$ is present in Z finally, it has to be brought into R1. So, we do Zout and R1in. So, for executing a complete instruction part of which we already knew it, so this is only fetch phase. So, every instruction will have these three steps in common for single bus architecture.

(Refer Slide Time: 30:08)

Example for a Three Bus Organization

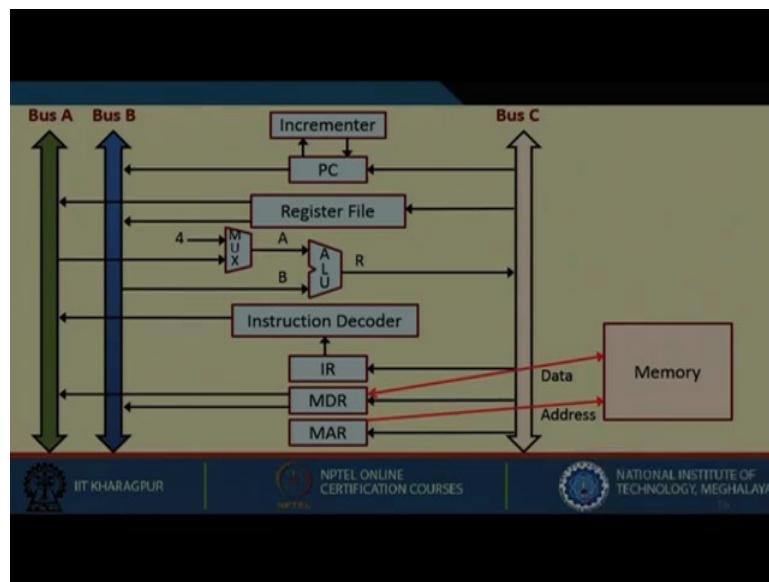
```
SUB R1, R2, R3 // R1 = R2 + R3

T1: PCout, R = B, MARin, READ, IncPC
T2: WMFC
T3: MDRoutB, R = B, IRin
T4: R2outA, R3outB, SelectA, SUB, R1in, End
```



Let us move on with the three-bus architecture where we will be performing with operation SUB R1,R2,R3. So, R1 will be R2 minus R3. So, in this particular case same way we need to fetch from memory, and this is the bus organization.

(Refer Slide Time: 30:35)



Now, PC has got an incrementer, and from this register one input goes to A another input goes to B. The input which goes to A has to come through this MUX and made available to the A input of the ALU. And another input is directly connected, so one input is directly connected to the B input of ALU. So, with this architecture, now we will execute

this particular instruction. So, first we do PCout, R=B. So, R is a signal, we are making R equal to B. We do MARin. So, basically PCout when we do it is available through B bus. So, as it is available through this B input. So, R=B, MARin, Read and IncPC. So, we are performing it in one go.

In any case, if we have performed MARin, we have to wait for MFC. Once the signal is available to the processor, then in MDR the data will be available. And we do MDRout R=B and IRin. Let us see this. We do MDRoutB. So, MDRout can be made available to A also to B, but we are making MDRoutB then R=B and through this R, it is going to IRin. So, this is the path which it is following; we are doing MDRoutB and then we are doing R=B. Now, R=B means the B value through this R will be made available to C bus; and from this C bus, the data will go into IR.

Now, these three steps are required to fetch the instruction. Now to execute the instruction, as we already know that two registers can be read simultaneously and one register can be written simultaneously, so two register read and one write can be performed simultaneously. Similarly, for this instruction we require something like this; we need to read the value from R2 and R3 and we have to write the value back in R1. So, what we are doing R2outA. So, R2out will be available to A bus, R3outB it is available to B bus. We need to selectA, because it is connected the A bus, then we provide the operation that needs to be performed that is SUB. And then at the same time we do R1in; two different reads and one write is performed in one particular step and finally End.

Let us see this. What we are doing we are doing R2out we are doing R2outA. So, from this register file we are doing R2outA. Then R3outB, B is directly connected no problem. As A is connected to this MUX we are doing selectA, so it is connected, we are performing SUB and we are doing R1 in that is what we are doing. R2outA, R3outB, selectA, SUB, and R1in and finally, we End. So, what we can see that for performing the same operation earlier we were requiring three steps, that is now reduced to one single step; in one single step, we are able to perform this SUB operation or any ALU operation because at the same time we are reading from two registers and writing into one register.

So, we have come to the end of this lecture. And in the next lecture, we will be looking forward to how we can execute various other instructions.

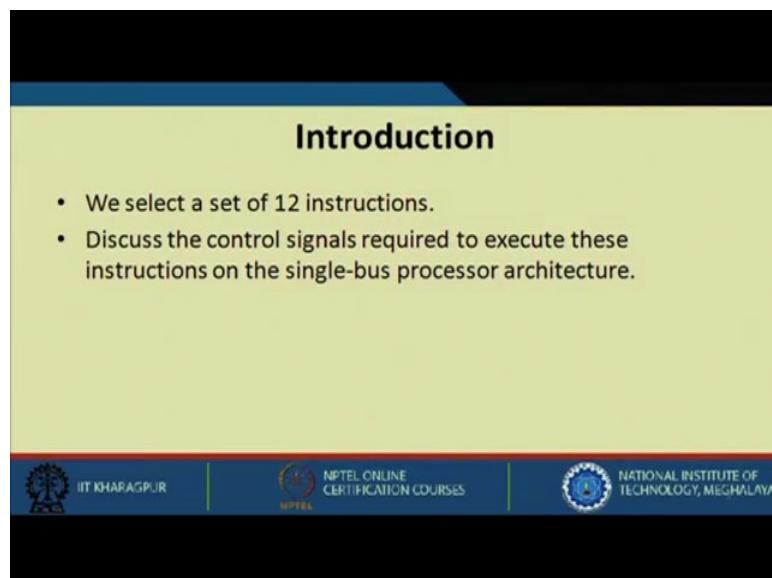
Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture – 19
Design of Control Unit (Part 3)

Welcome to the next lecture. In this lecture, we will continue with the design of control unit. Till now we have seen single bus organization internal to CPU, and multi bus which is a three-bus organization again that is internal to CPU. Now, we will be looking into how we can execute various kinds of instructions using a typical single bus architecture.

(Refer Slide Time: 01:06)



So, for this what we have done; we have selected a set of 12 instructions. And we will discuss the control signals required to execute these instructions on a single-bus processor architecture.

(Refer Slide Time: 01:23)

Various instructions: Control sequence

1. ADD R1, R2 // $R1 = R1 + R2$
2. ADD R1, LOCA // $R1 = R1 + \text{Mem}[LOCA]$
3. LOAD R1, LOCA // $R1 = \text{Mem}[LOCA]$
4. STORE LOCA, R1 // $\text{Mem}[LOCA] = R1$
5. MOVE R1, R2 // $R1 = R2$
6. MOVE R1, #10 // $R1 = 10$
7. BR LOCA // $PC = LOCA$
8. BZ LOCA // $PC = LOCA$ if Zero flag is set
9. INC R1 // $R1 = R1 + 4$
10. DEC R1 // $R1 = R1 - 4$
11. CMP R1, R2 // $R1 - R2$
12. HALT // Machine Halt

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, for showing the control signals for various instructions, we have chosen these set of instructions. This set includes ADD with register, ADD with memory operand, loading a word, storing a word, moving data between register, moving an immediate data to a register, unconditional branch, conditional branch, increment, decrement, compare and of course, halt. We shall be showing the various control signals that are required to execute the following set of instructions.

(Refer Slide Time: 02:20)

1. ADD R1, R2 (R1 = R1+R2)

Steps	Action
1	PC _{out} , MAR _{in} Read, Select4, Add, Z _{in}
2	Z _{out} , PC _{in} , Y _{in} , WMFC
3	MDR _{out} , IR _{in}
4	R1 _{out} , Y _{in}
5	R2 _{out} , SelectY, Add, Z _{in}
6	Z _{out} , R1 _{in} , End

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Let us see this. As we know that for all the instructions that I will be showing you, the first three steps will be the same. And I have taken single bus architecture to show all the instructions. So, let us see this; PCout, MARin, Read, Select4, Add, Zin. This set of control signals needs to be activated at time step 1 for execution. So, what it will do, it will output the value of PC into the bus. And then it will be put it into MAR, then we activate the Read control signal, Select4, PCout. So, it will be available to the A input, and when we do Select4 then A input will be having 4. We perform add, and after addition we put the result in Zin. Now the incremented value of PC is in Zin. We do Zout, and we again put it in PC using PCin signal, and then we also put it in Y. We will be seeing course of time why we are putting it in Y, and then we wait for MFC.

In the next step we do MDRout and IRin; that means, after the MFC is set by the memory unit. What we can do is that we know that the instruction is now available in MDR. Once the instruction is available in MDR, it can be brought into IR. Now, at this stage it is ready for decoding and execution. So, from the next step the instruction gets decoded, and after decoding it is ready for execution. So, this instruction basically adds the content of R1 and R2 and stores back in R1. So, for this one input should directly come through bus and the other input should come through Y. So, we are doing R1out, Yin. So, one of the register value is coming through Y. Next we are doing R2out, SelectY, Add, Zin. So, now R2out is available directly from the bus, we perform SelectY, Y will get selected we perform Add and put the result in Zin. And finally, we do Zout and the data is now available in R1 and finally, End.

(Refer Slide Time: 05:37)

2. ADD R1, LOCA ($R1 = R1 + \text{Mem}[LOCA]$)

Steps	Action
1	PC_{outr} , MAR_{in} , Read, Select4, Add, Z_{in}
2	Z_{outr} , PC_{in} , Y_{in} , WMFC
3	MDR_{outr} , IR_{in}
4	Address field of IR_{out} , MAR_{in} , Read
5	$R1_{outr}$, Y_{in} , WMFC
6	MDR_{outr} , SelectY, Add, Z_{in}
7	Z_{outr} , $R1_{in}$, End

Let us move on. Now, this is for an extra memory access. So, what this instruction will do, it will load the value from LOCA in memory, it will add it with content of R1, and it will store back to R1. Let us see how we can do it. Now, I will not be repeating this again because first three steps are again the fetch, which is same for all instructions. Let us see from next step what we need to do. In the first step, this is a memory location. So, this particular value should be put in MAR and we need to activate the Read control signal. Once WMFC signal will arrive, then the data available in MDR will be brought in, added with R1 and will store back in R1.

Again let us see what are the steps that are required for this operation. First you need to make available this; this can be available from the address field of IRout. So, address field of IRout will make available this content which goes to MARin at the same cycle, and we perform Read, we activate the Read control signal. After this is done then it is hit to the memory and it starts reading the data. In the next step, we need to wait for the MFC. At the same time, we can also make available the value of R1 into Y. So, we can do $R1_{out}$, Y_{in} . So, R1 value is now available in Y register.

Next what we do after this MFC signal is provided by the memory, then the data will be available in MDR, we do MDR_{out} , we do SelectY because already in Y the R1 value is present, we perform Add and we do Z_{in} . So, after performing this Z_{in} the result is now

in Z register, but the result has to be in R1. So, we perform Zout, R1in and finally, we End.

(Refer Slide Time: 08:22)

3. LOAD R1, LOCA (R1 = Mem[LOCA])

Steps	Action
1	PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}
2	Z _{out} , PC _{in} , Y _{in} , WMFC
3	MDR _{out} , IR _{in}
4	Address field of IRout, MAR _{in} , Read
5	WMFC
6	MDR _{out} , R1 _{in} , END

Next let us see the instruction to load the value from a memory location. We already did it in the previous instructions, but just for loading let us see what to do. In the same way first three steps will be same, and from the fourth step this will be present in the address field of IRout that is put in MAR using MARin, reactivate the Read control signal. And then we wait for MFC. Once MFC is performed then the value is available in MDR; and from MDR we have to put it in R1. So, we do MDRout, R1in and End for loading a word from memory.

(Refer Slide Time: 09:16)

Steps	Action
1	PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in}
2	Z_{out} , PC_{in} , Y_{in} , WMFC
3	MDR_{out} , IR_{in}
4	Address field of IR_{out} , MAR_{in}
5	$R1_{out}$, MDR_{in} , Write
6	MDR_{out} , WMFC, End

Let us move on how to store a word into memory. Once we activate the Read control signal, the data is read, and it is stored in MDR. For write operation, similarly we know that we have to put the address in MAR and the data that needs to be written will be put in MDR, and then only we can activate Write control signal. So, we need to do these two things; earlier we were just putting the value that the address that needs to be read into MAR, and we were activating the Read control signal, but for Write, you need to put the data in MAR as well as in MDR and then you activate the Write control signal. So, what we are doing here from step 4, address field of IR_{out} , MAR_{in} , $R1_{out}$, MDR_{in} because the content of $R1$ needs to be written in to LOCA. So, $R1$ content should be in MDR and activate Write control signal. And in next step we wait for MFC, after the MFC has arrived then we can End it.

(Refer Slide Time: 10:40)

5. MOVE R1, R2 (R1 = R2)

Steps	Action
1	PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}
2	Z _{out} , PC _{in} , Y _{in} , WMFC
3	MDR _{out} , IR _{in}
4	R2 _{out} , R1 _{in} , END

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

This is very simple moving R2 to R1. So, R2out, R1in --- the content of R2 will be moved to R1.

(Refer Slide Time: 10:57)

6. MOVE R1, #10 (R1 = 10)

Step	Action
1	PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}
2	Z _{out} , PC _{in} , Y _{in} , WMFC
3	MDR _{out} , IR _{in}
4	Immediate field of IR _{out} , R1 _{in} , END

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Many cases we require to store an immediate value into a register. So, immediate value is stored in the immediate field of IRout. So, the immediate field of IRout value we need to put it in R1. So, we simply do immediate field of IRout. R1in. So, value of this that is 10 will be moved to R1.

(Refer Slide Time: 11:32)

The slide is titled "7. BRANCH Label (PC = PC + offset)". It contains a table with 5 rows, each representing a step with its corresponding action:

Step	Action
1	PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}
2	Z _{out} , PC _{in} , Y _{in} , WMFC
3	MDR _{out} , IR _{in}
4	Offset-field-of-IR _{out} , SelectY, Add, Z _{in}
5	Z _{out} , PC _{in} , End

At the bottom of the slide, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Now, coming to branch instruction. So, in unconditional branch there is no condition; if branch is there we need to load the PC with the branch address. So, we know that every time we fetch an instruction the PC value gets incremented to point to the next instruction. In any instruction in the fetch phase we do not know whether it is a branch instruction or not. So, in any case we are incrementing the PC value, but after decoding the instruction, we will know that this is a branch instruction. Once we know after decoding that this is a branch instruction then what we need to do, we need to load the value of the PC with the branch address that needs to be calculated, and then it should be loaded into this again.

So, let us see what the steps that are required for this purpose. Now, recall our discussion where we said that in step 2, what we are doing, we are doing Zout, PCin and we are also doing Yin. And I told you at that point that this Yin is required for branch. Now, let us see why it is required for branch. Branch to Label that means content of the PC value should be added or subtracted depending on the offset, and we get a new PC value where we have to go and fetch the instruction. So, in this case, what we are doing offset field of IRout will be available; and if we SelectY because you see in Y, PC value is incremented. PC value is already there. So, in Y as incremented PC value is already there, we need to add that PC value with offset, so that is what we are doing, offset field of IRout, SelectY, because Yin already contains the updated PC value we Add and we do

Zin. And finally, we do Zout, PCin, and End. So, these are the following steps required when it is an unconditional branch.

(Refer Slide Time: 14:33)

Step	Action
1	$PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}$
2	$Z_{out}, PC_{in}, Y_{in}, WMFC$
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR _{out} , SelectY, Add, Z _{in} , If Z=0 then End
5	Z_{out}, PC_{in}, End

Let us move on and see for a conditional branch. So, for a conditional branch what happens if the condition is met then only it will branch; otherwise normally it will proceed to the next instruction. So, similarly here first three steps will be same and finally, here offset field of IRout, SelectY, Add, Zin. Here we need to check if Z is equal to zero or not, that means if the zero flag is set or not. If Z is equal to 0 that means, zero flag is not set. If the zero flag is not set then there is no need to do anything you can stop here, and then the PC value that already got incremented to the next one. So, it will fetch the next instruction as usual and it will do it because the branch condition is not satisfied.

Let say if Z is equal to 1, then what happen we have already calculated Z. This register Z value here which is offset value is added with content of PC; we will do set out PCin and then End. Now, PC will be loaded with the conditional branch address. With this that we have calculated using this particular offset. So, branch if zero which is a conditional branch, a condition needs to be checked that is here the flag, and then accordingly it will be Z if the condition is not satisfied it will End here and the PC will next fetch the next instruction. And if it is satisfied then it will load the PC with the branch address.

(Refer Slide Time: 16:43)

9. INC R1 ($R1 = R1 + 4$)

Steps	Action
1	$PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}$
2	$Z_{out}, PC_{in}, Y_{in}, WMFC$
3	MDR_{out}, IR_{in}
4	$R1_{out}, Select4, Add, Z_{in}$
5	$Z_{out}, R1_{in}, End$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Let us see the next instruction INC R1, where R1 is incremented by 4 and stored back in R1. First three steps will be again same, what we have to do R1out, Select4. If you Select4 then 4 will be selected from the MUX, which will go to one of the inputs of ALU. We add it and then we put it in Z. And finally, the value of Z is put in R1 by Zout and R1in and finally, we End. So, this is for increment.

(Refer Slide Time: 17:25)

10. DEC R1 ($R1 = R1 - 4$)

Steps	Action
1	$PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}$
2	$Z_{out}, PC_{in}, Y_{in}, WMFC$
3	MDR_{out}, IR_{in}
4	$R1_{out}, Select4, SUB, Z_{in}$
5	$Z_{out}, R1_{in}, End$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Similarly, we can have for decrement where everything will be same the only difference being this operation that is SUB. Here we will do R1out, Select4, SUB and then Zin and

then Zout and R1in; the value of Z will be put it into R1. The decremented value will now will be present in R1.

(Refer Slide Time: 17:52)

The slide is titled "11. CMP R1, R2". It contains a table with two columns: "Steps" and "Action".

Steps	Action
1	PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}
2	Z _{out} , PC _{in} , Y _{in} , WMFC
3	MDR _{out} , IR _{in}
4	R1 _{out} , Y _{in}
5	R2 _{out} , SelectY, Sub, Z _{in} , End

At the bottom of the slide, there are three logos: IIT Kharagpur, NPTEL, and NIT Meghalaya.

Compare R1 and R2. So, generally when we perform an operation depending on that operation the flags gets affected, the flags are set or reset. So, how do we compare? If you subtract these two numbers, if we subtract R1 - R2 and the result is zero, then the zero flag will be set. Let say we subtract R1, R2 and the result is positive; that means, R1 is greater than R2; let us say we subtract R1, R2 and the result is negative, that means R2 is greater than R1.

So, depending on on the subtraction various flags will get affected, and accordingly we can compare it. So, what we are doing here, we are doing R1out, Yin, R2out, SelectY, and then we do SUB and then Zin. So, after SUB and Zin, after the operation of ALU, flags will get affected; and accordingly we can see based on the flag value. So, after performing this, we have to check a flag value to see whether R1 is greater than R2, or they are equal, or R2 is greater than R1 depending on this.

(Refer Slide Time: 19:36)

The slide is titled "12. HALT". It contains a table with two columns: "Steps" and "Action".

Steps	Action
1	$PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}$
2	$Z_{out}, PC_{in}, Y_{in}, WMFC$
3	MDR_{out}, IR_{in}
4	End

The slide footer includes logos for IIT Kharagpur, NPTEL, and NPTEL Online Certification Courses, along with a video feed of a speaker.

The last instruction is HALT. HALT will stop the execution of the instruction. But HALT itself is an instruction. So, for every instruction, what we do we fetch the instruction and then we decode the instruction. So, the first three steps will be same for HALT and after at the end of step 3 once it is decoded, it is known that this is a HALT instruction. So, for a HALT instruction finally, what it will do, it will End. In step 4 it will End.

So, in this lecture we have shown basically the various instructions, we have taken a set of 12 instructions and we have shown you that how using single bus architecture we can execute these instructions.

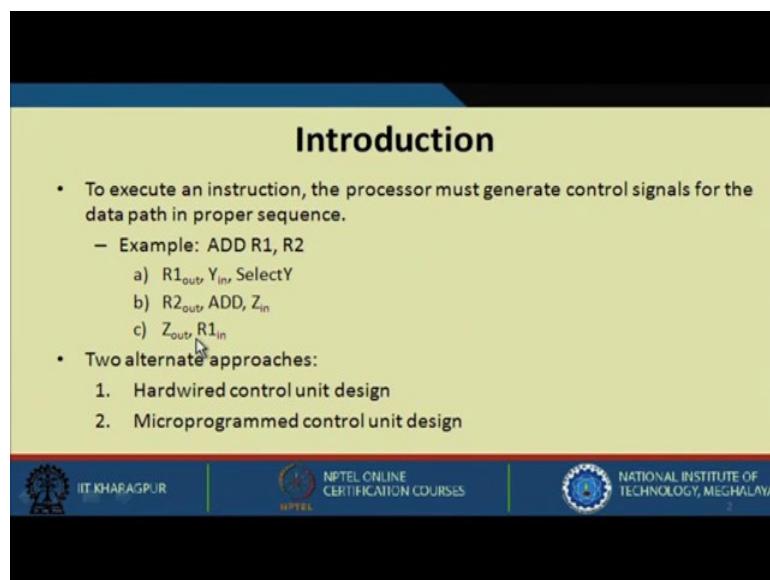
Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture – 20
Design of Control Unit (Part 4)

Welcome to lecture 20; the design of control unit part 4. Till now we have seen the various internal bus architecture and we have also seen that how various instructions are executed. Now we will look into the approaches that are required for generation of these control signals; what kind of approaches are there. Broadly there are 2 types of approaches.

(Refer Slide Time: 01:00)



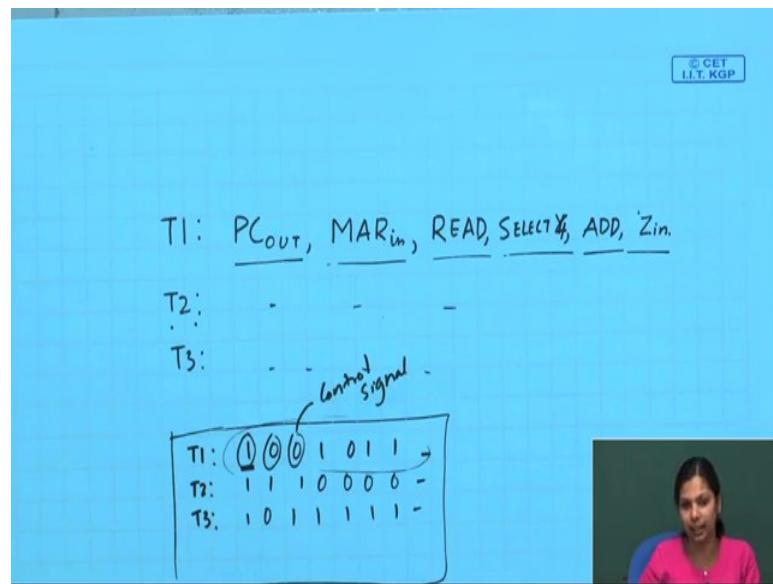
Introduction

- To execute an instruction, the processor must generate control signals for the data path in proper sequence.
 - Example: ADD R1, R2
 - a) $R1_{out}, Y_{in}, \text{SelectY}$
 - b) $R2_{out}, \text{ADD}, Z_{in}$
 - c) $Z_{out}, R1_{in}$
 - Two alternate approaches:
 1. Hardwired control unit design
 2. Microprogrammed control unit design

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

One is hardware control unit design, and microprogrammed control unit design. So, what we are trying to say, we know that for this instruction these are the control signals that are required to be generated.

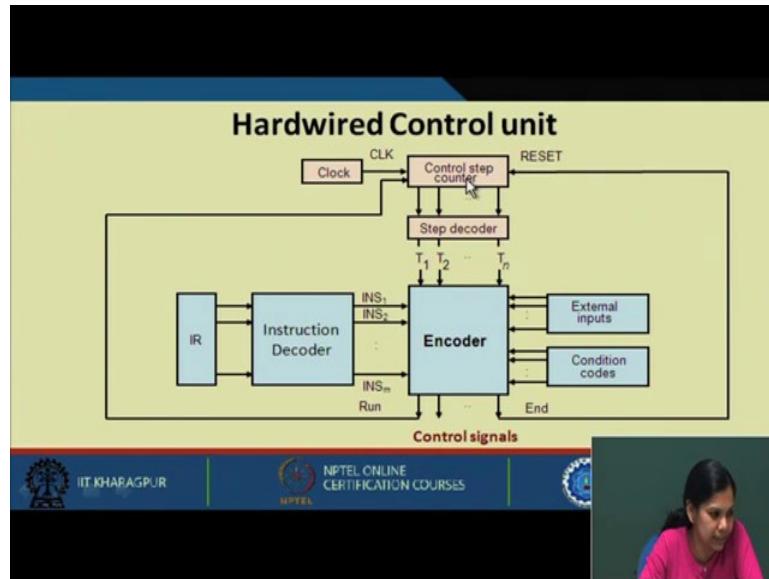
(Refer Slide Time: 01:31)



But now these control signals must be generated in a proper sequence what do you mean by that. So, let us take an example.

So, we say that in T1; what is performed PCout, MARin, READ, Select4, ADD, Zin. So, at T1 for all the instructions these signals must be generated. Similarly in step 2 some more signals, step 3 some more signals and so on. So, the processor must generate the control signals for the data path in a proper sequence. We will be looking into two approaches, one is hardware control unit design, and another is microprogrammed control unit design.

(Refer Slide Time: 03:16)



Coming to hardware control unit design, let us see what we have here. We have a clock that is hitting the control step counter. The control step counter is connected to a step decoder. Basically the step counter generates the steps that are required by an instruction to generate the control signals. So, at these steps T_1, T_2, T_3 various control signals will get generated depending on which instruction we are using, what is the step of that instruction, whether some external inputs are required for it or not, if some conditional codes needs to be checked or not. So, depending on all these, we see that in this encoder the content of this step decoder the content of instruction decoder and the external inputs and the condition codes all are getting input, and the encoder is encoding based on that the control signals are generated.

So, we will be looking into each and every aspect of this instruction decoder. We already know conditional codes for every ALU operation; certain flags gets sets and depending on the condition codes can be considered external inputs what can be the external input an MFC memory function complete that is an external input that comes from a different module that is memory and there are 2 more signals one is Run another is End we will be seeing why we are requiring this, but you see this entire structure. So, this entire structure is basically doing what it is generating control signals. How it is generating control signals depending on which instruction and which step it is, and if there is some external inputs for that depending on all these this encoder will generate the control signals.

(Refer Slide Time: 05:51)

Sequence of control signals for ADD R1, LOCA

Steps	Action
1	PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}
2	Z _{out} , PC _{in} , Y _{in} , WMFC
3	MDR _{out} , IR _{in}
4	Address field of IR _{out} , MAR _{in} , Read
5	R1 _{out} , Y _{in} , WMFC
6	MDR _{out} , SelectY, Add, Z _{in}
7	Z _{out} , R1 _{in} , End

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, and this we already know that this is a sequence of control signals for ADD R1, LOCA. So, these are the set of microinstruction that are executed.

(Refer Slide Time: 06:08)

Hardwired Control Unit Design

- Assumption:
 - Each step in this sequence is completed in one clock cycle.
- A counter is used to keep track of the time step.
- The control signals are determined by the following information:
 - Content of control step counter
 - Content of instruction register
 - Content of conditional code flags
 - External input signals such as MFC (Memory Function Complete)

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

In hardwired control unit design there is an assumption that each step in the sequence is completed in one clock cycle, but remember one thing that when we are reading something from the memory it may not be completed in one clock cycle. We cannot ensure that if we are reading from memory it will be completed in 1 clock cycle, but this is an assumption that we have made. A counter is used to keep track of the time step.

A step counter is there which keeps track of the time step. Let us see the previous instruction, here 7 steps are required; some instruction might take 8 steps, some instruction might takes 4 steps or 5 steps. So, this has to be taken care of. So, counter is used to keep track of the time step. The control signals are determined by the following information: what are the following information the content of the control step, counter content of the instruction register, content of conditional code flags and external inputs such as MFC. We have already seen that the encoder is taking input of the step decoder, taking input of the instruction register, and it is also taking the conditional code and external inputs depending on all it is generating the control signals. So, that is what it is saying that the control signals are determined by the following information content of control step counter, content of instruction register, and content of conditional code flags and external input signals such as MFC.

(Refer Slide Time: 08:24)

- The encoder/decoder circuit is a combinational circuit which generates control signals depending on the inputs provided.
- The step decoder generates separate signal line for each step in the control sequence (T_1, T_2, T_3 , etc.).
 - Depending on maximum steps required for an instruction, the step decoder is designed.
 - If a maximum of 10 steps are required, then a 4×16 step decoder is used.
- Among the total set of instructions, the instruction decoder is used to select one of them. (That particular line will be 1 and rest will be 0).
 - If a maximum of 100 instructions are present in the ISA then a 7×128 instruction decoder is used.

Now, let us see this; the encoder-decoder circuit is a combinational circuit that generates the control signals depending on the inputs provided. So, whatever be the input from the step decoder, instruction register, external inputs, and conditional code, the encoder-decoder circuitry will generate the control signals. Now see the step decoder generates separate signal line for each step in the control sequence. So, at step 1, PCout, MARin, READ, Select4, ADD, Zin happens. At T2, Zout, PCin, Yin is happening, and in T3 MDRout, IRin is happening. So, at various time steps different control signals are getting executed.

So, how many steps we require is dependent on the maximum steps required by an instruction. The step decoder is designed depending on the maximum steps required for an instruction. So, let us say we have 20 instruction in our instruction set architecture, and every instruction takes 4 steps, 6 steps, 7 steps, and maximum steps that is required for an instruction is let us say 8. None of the instructions take more than 8 steps, then what will be the size of your step decoder we need to generate a maximum of 8 steps.

So, a 3×8 decoder will do, but if the maximum step is 10 in that case step decoder size will be 4×16 . In case we require only maximum of 8 steps for any instruction in that case a 3×8 decoder will do, but for other if it requires more then we will be requiring 4×16 decoder. So, if a maximum of 10 steps are required then a 4×16 step decoder is used. Among the total set of instructions, the instruction decoder is used to select one of them. That particular line will be 1 and the rest will be 0. So, depending on total number of instruction let us say we have a total of 30 instructions.

So, if you have a total of 30 instructions how many bits will be required to encode? If we will require 5 bits, because 5×32 decoder will be used. So, output at any point of time depending on the input any one line will be 1 and the rest will be 0. So, it can encode 30 instructions; similarly if you have 100 instructions how many bits will be required to encode that? We would require 7 bits and we require a 7×128 decoder.

(Refer Slide Time: 12:14)

- At every clock cycle the RUN signal is used to increment the counter by one.
 - When RUN is 0 the counter stops counting.
 - This signal is needed when WMFC is issued.
- END signal starts a new instruction.
 - It resets the control step counter to its starting value.
- The sequence of operations carried out by the control unit is determined by the wiring of the logic elements and hence it is named **hardwired**.
- This approach of control unit design is fast but limited to the complexity of instruction set that is implemented.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Then a 7×128 instruction decoder is used at every clock cycle. The RUN signal is used to increment the counter by one; when RUN is 0 the counter stops running this signal is needed when WMFC is issued. Now see what we are trying to say. So, when the step decoder value is at step T1 for instruction ADD, certain set of control signals will get generated. After that step is performed then we need to move to the next step that is T2. So, while moving to T2 what we need to do we need to move from T1 to T2.

So, the step counter will increment to T2 and now whatever is the assigned job for T2 it will get executed, and similarly the step counter will go on incrementing. So, this RUN signal is basically used to help the step counter to implement the steps of any instruction. Similarly when RUN is 0, the counter stops counting and when this RUN is required to be 0 this is required for WMFC; see when we are doing WMFC, we will not be executing the next one until we get this confirmation that the data is available, then only we can take that data because we have to operate on that data if that data is not present we cannot open it. So, when that is. So, the RUN becomes 0 at that particular time and when it starts to run again it will keep on running. So, this is for within an instruction incrementing to the next; next step, next step and next step, the End signal starts a new instruction --- it means we have completely executed one instruction and now we will be moving to the next instruction. Once we have completely executed one instruction it will be fully done and then we have to again reset the step decoder.

So, the step counter basically needs to be reset. So, this is done using the End signal. So, the End signal starts a new instruction, it resets the control step counter to its starting value. So, that the next instruction can be started now. In the hardware control unit design, the sequence of operation carried out by the control unit is determined by the wiring of the logic elements, and hence it is named hardwired. Now we see that it is basically a combinational circuit; we are giving inputs and we are getting some output.

So, this basically depends on the wiring how you have put on everything in place, how you have placed the encoder, how you have used the step decoder everything, and ultimately this is the wiring. Hence it is called hardwired control unit design, and this approach of control unit design is fast, but limited to the complexity of instruction set that is implemented. So, we cannot have very complex instructions implemented using hardware, but simple instructions can be executed and it will be much fast and efficient,

but we cannot have very complex instruction implemented here. With more complex structures the complexity increases and that flexibility will go off.

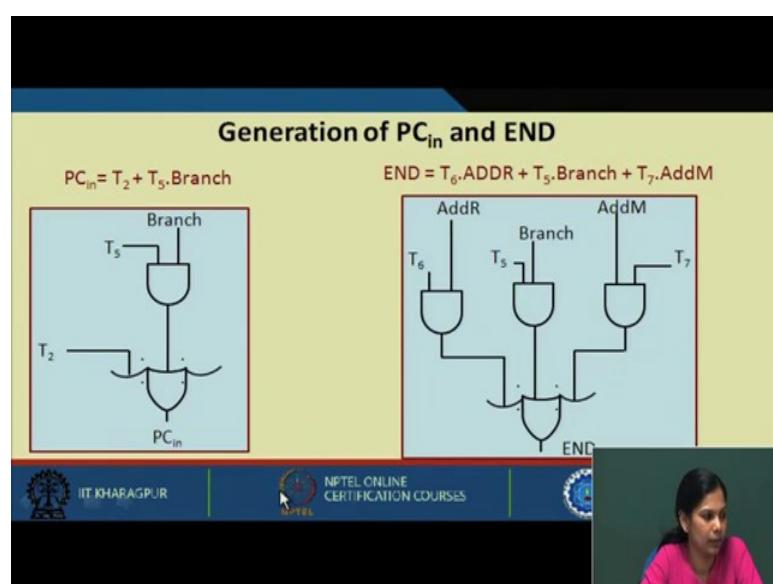
(Refer Slide Time: 16:38)

Generation of Control Signals		
ADD R1, R2	ADD R1, LOCA	BRANCH Label
1 PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}	1 PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}	1 PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}
2 Z _{out} , PC _{in} , Y _{in} , WMFC	2 Z _{out} , PC _{in} , Y _{in} , WMFC	2 Z _{out} , PC _{in} , Y _{in} , WMFC
3 MDR _{out} , IR _{in}	3 MDR _{out} , IR _{in}	3 MDR _{out} , IR _{in}
4 R1 _{out} , Y _{in}	4 Address field of IROUT, MAR _{in} , Read	4 Offset-field-of-IR _{out} , SelectY, Add, Z _{in}
5 R2 _{out} , SelectY, Add, Z _{in}	5 R1 _{out} , Y _{in} , WMFC	5 Z _{out} , PC _{in} , End
6 Z _{out} , R1 _{in} , End	6 MDR _{out} , SelectY, Add, Z _{in}	
	7 Z _{out} , R1 _{in} , End	

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, keeping the flexibility of this hardware simple instructions are basically implemented using hardwired control unit design. Let us now see these 3 instructions together. These are the control signals required for ADD R1, R2; ADD R1,LOCA; and BRANCH Label. So, at various time steps following control signals are getting generated.

(Refer Slide Time: 17:04)

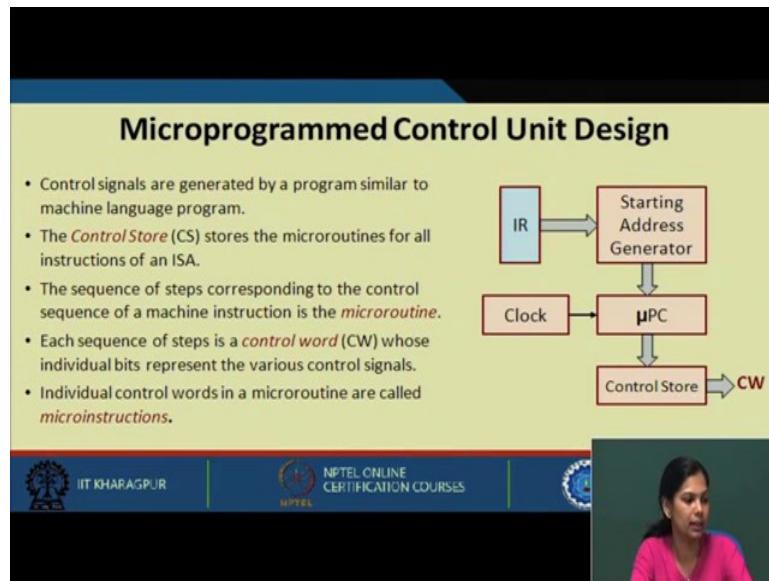


For these particular 3 instructions I will now generate the signal using hardwired control unit hardwired design for PCin and END. Now see what is PCin. If you go back we have to see where we have PCin; T1 we do not have PCin even for anyone, and in T2 we have PCin for all the instructions, and then we see that we do not have any PCin for ADD, we do not have any PCin for ADD R1,LOCA, but we have PCin in step 5 for BRANCH.

So, we can take this into consideration that PCin signal is activated at T2 for all the instructions, and PCin is activated for branch instruction in T5. So, we can write the control signal PCin as T2 for all instructions plus at T5 when it is branch instruction only. So, this logic expression can be written in the form of this logic gate where T5 and BRANCH is connected with an AND gate which is connected to the input of an OR gate.

So, this expression can be implemented using this logic function. Similarly let us see END. END happens at T6 for AddR, at T5 for Branch, and at T7 for AddM. So, we can have a logic expression at T6 for AddR, at T5 for Branch, and at T7 for AddM. So, you can see that we can implement this using the following logic expression and following logic gate where these are connected with AND gates, and this is connected finally with an OR gate. So, END signal can be generated using this particular circuit.

(Refer Slide Time: 20:30)



Now, coming to microprogrammed control unit design. In microprogrammed control unit design we have a structure like this. So, what do we have here the instruction register will give a starting address generator, and this starting address generator will hit

a muPC. So, we will have within computer a small place where we will be doing a similar kind of function like a computer. So, here the IR will provide the starting address, which will hit to the muPC. So, the muPC will hit to a memory, known as control store. So, at every clock this muPC content will hit to some very high-speed memory called control store, and based on a particular instruction provided by the IR and the starting address generator, this control store will provide a control word. And this control word is nothing but it will give the information about the control signals which will be on and off, which will be required to be activated at what time period.

So, let us see in at this point of time we need to understand about some of the definitions. Control signals are generated by a program similar to machine language program like we were doing what we were fetching an instruction from memory, then the PC was getting incremented to the next address and again we were fetching, we were performing certain operation similar to that we will be doing something here. So, the control signals are generated by a program similar to machine language program the control store; control store is a place which stores micro-routines for all the instructions in the instruction set architecture.

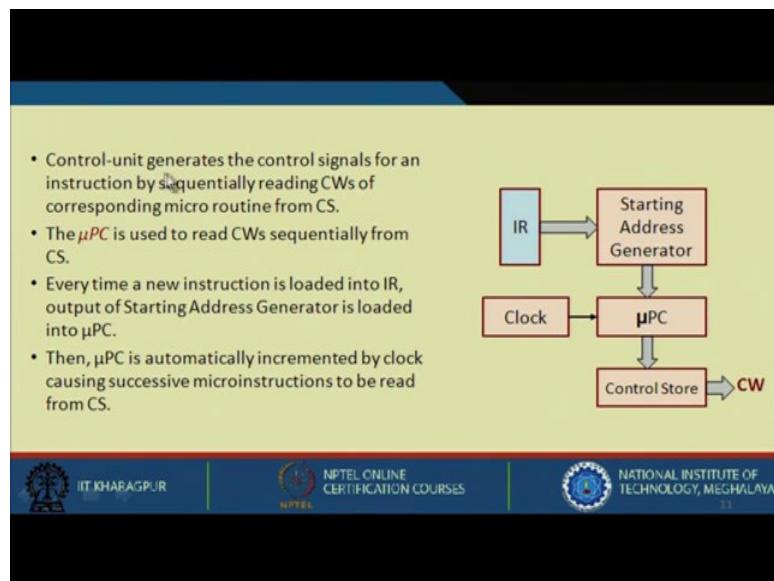
So, the micro-routines are nothing but all the control signals that are required for a particular instruction. So, the control store stores the micro-routine for all the instructions of an instruction set architecture. The sequence of steps corresponding to a control sequence of a machine instruction is specified by the micro-routine. So, if an instruction requires 4 steps for execution, all the sequence of steps can be regarded as a micro routine for that particular instruction, each sequence of steps in a control word whose individual bits represent the various control signals.

So, each sequence of step is a control word; so, control word is a memory where we are storing so many things. So, let us say this is a control word 1 0 0 1 0 1 0 1 1 1 0 0 0 0 1 0 something like this we have. So, what we can say here is that, let us say this is step 1,, this is step 2 and this is step 3. So, this is basically my control store and what we are storing here? We are storing control words --- these are control words, and each sequence of steps in a control word whose individual bits represent the various control signals, meaning this is a control word and its individual bits represent some control signals. So, these are individual control signals that are on or off; that means, it is either

active or not active, and individual control words in a micro-routine are called micro instructions.

Now, let us see this. So, as I said IR depending on an instruction, will generate the starting address that will hit to the muPC, and then muPC will hit the control store and from where the control words will get generated at every clock period. So, at every clock period the muPC will get incremented by the required amount, and then from the control store each of the control words will be generated, and the control word will give which particular control signal is required to be 1 or which particular control signal is required to be 0. So, as I have already discussed this control unit generates the control signals for an instruction by sequentially reading the control words of the corresponding micro routine from control store.

(Refer Slide Time: 26:18)



So, control store stores the control words, or we can say it stores a micro-routine. The muPC is used to read the control word sequentially from the control store. So, muPC hits the control store, and it is sequentially reads the control word one by one by one. So, every time a new instruction is loaded into IR because we when we are executing one instruction that particular starting address will be generated in muPC, and accordingly this will happen same way if the next instruction needs to get executed. Then the next instruction will get loaded in the muPC and then the starting address of that particular instruction will get loaded into the muPC and accordingly from the control store the

control words will get generated. The muPC is automatically incremented by clock causing successive micro instructions to be read from control store.

(Refer Slide Time: 27:39)

Control Store for "ADD R1, R2"																		
Micro-instr.	:	PC _{in}	PC _{out}	MAR _{in}	Read	MDR _{out}	IR _{in}	Y _{in}	Select	Add	Z _{in}	Z _{out}	R1 _{out}	R1 _{in}	R2 _{out}	WMFC	End	:
1	0	0 1	1 1	1 0 0 0	1 1	0 0 0 0	0 1 0 0	1 0 0 0	1 1 1 0	0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	
2	0	1 0	0 0	0 0 0 0	0 0	0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1	0 0 0 0	0 0 0 0	
3	0	0 0	0 0	0 0 0 0	0 1	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	
4	0	0 0	0 0	0 0 0 0	0 0	0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	
5	0	0 0	0 0	0 0 0 0	0 0	0 0 0 0	0 0 1 0	1 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1	0 0 0 0	0 0 0 0	0 0 0 0	
6	0	0 0	0 0	0 0 0 0	0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 1 0 0	1 0 0 0	0 0 0 1	0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	





So, this is how it is stored. Let us say there will be many more control signals for ADD R1,R2. In step 1 what all control signals are 1, rest will be all 0. So, PCout, MARin, READ, Select4, ADD, Zin these signals will be 1, and rest will be all 0. Similarly in step 2; we need to do Zout, PCin, Yin. So, see Zout will be 1, Yin is 1, and PCin is 1 at the same time we have to wait for MFC because we have performed a read operation here in the first step. Similarly in this step these 2 signals will be activated and rest will be 0. So, this is MDRout, IRin. Similarly in step 4 for this instruction we need to perform R1out, Yin and then what we are performing we are performing R2out, SelectY, ADD and Zin. So, in Z now the result is which should be stored in R1.

So, I have to do as a Zout and R1in. So, this is a control store for ADD R1,R2. This is a micro routine for this particular instruction, and these are the micro instruction and these are stored in control store, and these are the control words that are read each time. So, the muPC will be loaded with the starting address and then at every clock period this will get incremented by 1; and it will be fetched from the control store similarly for branch we have shown. So, similarly for branch first 3 will be same.

(Refer Slide Time: 30:04)

Micro-instr.	:	PC _{in}	PC _{out}	MAR _{in}	Read	MDR _{out}	IR _{in}	Y _{in}	Select	Add	Z _{in}	Z _{out}	IR _{out}	WMFC	End	:
1	0	0	1	1	1	0	0	0	1	1	1	0	0	0	0	0
2	0	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0
3	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0
5	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0

And for this particular thing we have to do SelectY. So, if Select4 is 1, SelectY will be 0 and we perform ADD, Zin now and then we also perform IRout because the offset field of IRout should be available, that will be added with. Y contains the previous PC value; we do perform an addADD, we do Zin and finally, we perform Zout and PCin. PC will be loaded with the new value that needs to be performed for the branch; that means, we need to go to that particular location for that branch.

So, this is how the control store looks like for branch.

(Refer Slide Time: 31:05)

Horizontal versus Vertical Microinstruction Encoding

- Broadly there are two alternate schemes to code the control signals in the control memory.
 - Horizontal Micro-instruction Encoding**
 - Each control signal is represented by a bit in the micro-instruction.
 - Fewer control store words, with more bits per word.
 - Vertical Micro-instruction Encoding**
 - Each control word represents a single micro-instruction in encoded form.
 - k-bit control word can support up to 2^k micro-instructions.
 - More control store words, with fewer bits per word

Now, there are 2 alternative ways to code the control signals in the control memory. The first way is the horizontal micro instruction encoding the one I have just shown. So, each control signals is represented by a bit in the micro instruction and fewer controls store words with more bits per word is required. So, fewer control store words are required, but we require more bits per word meaning here you see that only a few signals will be 1, rest all signals many are 0; only few are 1, but we still need to keep all why because we have made it in such a fashion we are keeping for all the set of control signals. So, the size of this control word will be dependent on the total number of control signals that are present.

So, fewer control store words will be required with more bits per word, and each word will contains more number of bits compared to vertical micro instruction encoding. Here each control word represent a single microinstruction in encoded form. So, if k bit control word can support up to 2^k microinstructions and more control store words with fewer bits per word.

(Refer Slide Time: 33:51)

- There can be a tradeoff between horizontal and vertical micro-instruction encoding.
 - Sometimes referred to as **Diagonal Micro-instruction Encoding**.
 - The control signals are grouped into sets S_1, S_2, \dots , such that the control signals within a set are mutually exclusive.
- Summary:
 - Horizontal encoding supports unlimited parallelism among micro-instructions.
 - Vertical encoding supports strictly sequential execution of micro-instructions.
 - Diagonal encoding does not sacrifice the required level of parallelism, but uses less number of bits per control word as compared to horizontal encoding.

Let us say we have a total of 100 control signals. In case of horizontal control unit design we require 100 bits for each control word, but in this case we require only 7 bits. So, at a time only one control signals can get activated because we are encoding using just 7 bits, earlier for horizontal it was 100 bits, but for this we only require 7 bits.

So k bit control word can support up to 2^k micro operation. So, if you have 100, then only 7 bits will suffice.

So, there can be tradeoff between horizontal and vertical microinstruction encoding, sometimes refer to as diagonal microinstruction encoding. So, what we do here is that the control signals are grouped into some sets and such control signals within a set of mutually exclusive. So, let us consider R1out; let us consider the single bus architecture. So, can we perform R1out and R2out together or MDRout together on any one of the operations at a time? So, we can group it into a mutually exclusive set. We require less number of bits to encode; say if there are 32 instruction, we will just require 5 bits to encode that, earlier we were using 32 bits for that.

So, the control signal are grouped into sets such that the control signals within a set are mutually exclusive. So, what is the summary out of this? So, horizontal encoding supports unlimited parallelism among microinstruction. So, many micro operations or those control signals can be activated at a time. Vertical encoding supports strictly sequential execution of micro instructions; only fewer bits are required. We require less storage, but it is sequentially performed, and it will not help for fast execution. Where the demand is for fast execution, diagonal encoding does not sacrifice the required level of parallelism, but uses less number of bits per control word as compared to horizontal encoding.

So, we can see that diagonal encoding is the best where we analyze the control signals depending on the architecture that we are using and based on that we divide into 2 groups, and we know that within that group only one operation can be performed. So, in such cases we can perform this.

(Refer Slide Time: 36:24)

(a) Horizontal Micro-instruction Encoding

- Suppose that there are k control signals: c_1, c_2, \dots, c_k .
- In horizontal encoding, every control word stored in control memory (CM) consists of k bits, one bit for every control signal.
- Several bits in a control word can be 1:
 - Parallel activation of several micro-operations in a single time step.

0 | 1 | 0 | 1 | 1 | 0 → c_2, c_4 and c_5 are activated together

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, diagonal encoding is considered as the best among horizontal and vertical. So, I will explain it in some detail. So, for horizontal micro instruction encoding, these are the control signals. Suppose there are k control signals, then in horizontal encoding every control word in the control memory consists of k bits, one bit for every control signals. So, in this case several bits in the control word can be 1; parallel activation of the several micro operation in a single time step is possible, but it is seen that in such kind of scenario where we have a very large control word many bits are 0s and few bits are 1.

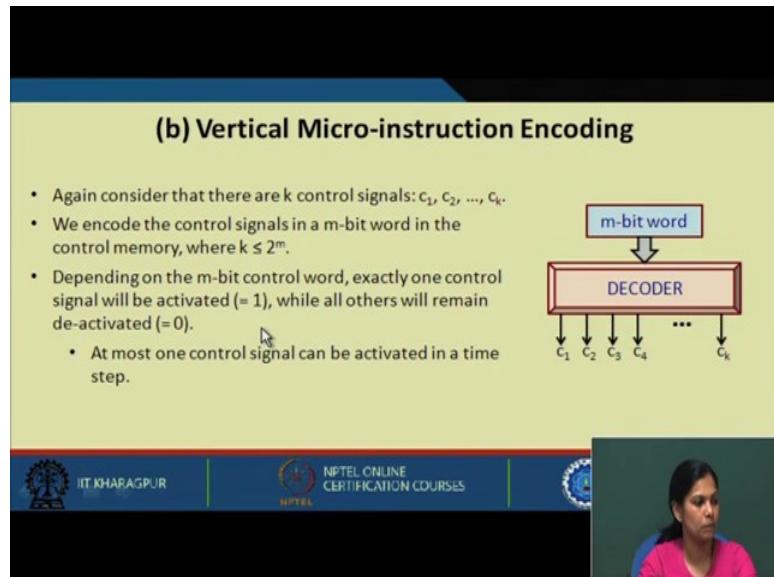
(Refer Slide Time: 37:22)

- Advantage:**
 - Unlimited parallelism is possible in the activation of the micro-operations.
- Disadvantage:**
 - Size of the control memory is large (word size is much longer).
 - Cost of implementation is higher.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

The advantages are that unlimited parallelism is possible in the activation of the micro operations, but the disadvantage being the large size of the control memory.

(Refer Slide Time: 37:46)



So, we also need to look into this because if you want to have a control memory that is very large this cost will be much higher. So, in vertical micro instruction encoding where we see that for m bits, a decoder is used for it. So, for k control signals, we require k that is less than equals to 2^m . So, m bits will be required for it depending on m bit control word exactly one control signal will be activated which is 1 while all others will be 0, but the number of bits required is much less. At most one control signal can be activated at a time. So, we cannot do PCout, MARin and all those signals at one go; we can only do first PCout then in the next step MARin, then READ, and so on.

(Refer Slide Time: 38:42)

- **Advantage:**
 - Requires much smaller word size in control memory.
 - Low cost of implementation.
- **Disadvantage:**
 - More than one control signals cannot be activated at a time.
 - Requires sequential activation of the control signals, and hence more number of time steps.

The slide also features the IIT Kharagpur logo, the NPTEL logo, and a video feed of a woman speaking.

So, the advantage here is it requires much smaller word size in control memory, low cost of implementation. If you do not want high speed implementation you can go for such kind of thing, but when speed is a vital concern this method cannot be taken into consideration. So, the disadvantage here is more than one control signals cannot be activated at a time; requires sequential activation of the control signal and hence more number of time steps will be required here. The option is diagonal micro instruction encoding which is quite flexible.

(Refer Slide Time: 39:16)

(c) Diagonal Micro-instruction Encoding

The diagram illustrates the process of diagonal micro-instruction encoding. It shows multiple m_i -bit fields (represented as m_1 -bit, m_2 -bit, m_3 -bit, ..., m_i -bit) being decoded by separate decoders. Each decoder takes its respective m_i -bit field as input and outputs a group of control signals, labeled $c_{1,1}, c_{1,k_1}, c_{2,1}, c_{2,k_2}, \dots, c_{m,1}, c_{m,k_s}$.

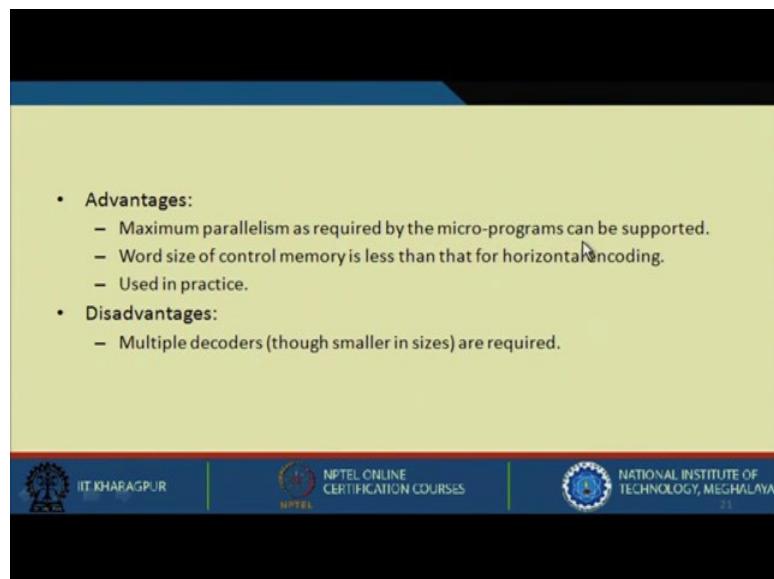
- Suppose we group the set of k control signals into s groups, containing k_1, k_2, \dots, k_s signals.
- We encode the control signals in groups as shown, where $k_i \leq 2^{m_i}$.
 - Within a group, at most one control signal can be activated in a time step.
 - Parallelism across groups is allowed.

The slide also features the IIT Kharagpur logo, the NPTEL logo, and a video feed of a woman speaking.

So, here also we require decoder. So, we group the set of k control signals into as groups containing k₁, k₂, k₃, etc. So, this group is having mutually exclusive instruction, this group is having mutually exclusive instruction ,and we know that at a time only one signal from this group will get activated.

So, we encode the control signals in group as shown where k_i is less than equals to 2^{m_i} this is depending on the size. So, within a group at most one control signal can be activated in a time step; parallelism across groups is allowed; also the number of bits required is less. So, by studying the architecture well in advance we can design such kind of microinstruction encoding to design a control unit.

(Refer Slide Time: 40:29)



So, the advantage here is maximum parallelism as required by the micro program can be supported and the word size of the control memory is less than the for horizontal encoding and this is used in practice. This is basically what is used in practice because parallelism can be exploited as well as the space is minimized, multiple encoders those smaller in sizes are required. So, that will take up some space, some area let us take an example suppose there are 100 control signals in a processor data path.

(Refer Slide Time: 41:00)

The slide is titled "Example 1". It lists three cases for control signals:

- Suppose there are 100 control signals in a processor data path.
- For horizontal encoding, control word size = 100 bits.
- For vertical encoding, control word size = $\lceil \log_2 100 \rceil = 7$ bits.
- For diagonal encoding, suppose after analysis of the micro-programs, we divide the control signals into 5 groups, containing 25, 15, 40, 5 and 15 control signals respectively.
 - We have: $m_1 = 5, m_2 = 4, m_3 = 6, m_4 = 3, m_5 = 4$
 - Control word size = $5 + 4 + 6 + 3 + 4 = 22$ bits.

Below the list, there is a table with three rows:

$25 \leq 2^5$	$15 \leq 2^4$
$40 \leq 2^6$	$5 \leq 2^3$
$15 \leq 2^4$	

The slide footer includes logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

So, for horizontal encoding control word size will be 100 bits. So, each will be 100 bits for vertical encoding $\log_2 100 = 7$ bits will be required and for diagonal encoding suppose after analysis. So, it has been generated that the groups the following groups can be made that is the following groups of mutually exclusive signals can be made that is 25, 15, 40, 5 and 15.

So, in each of the cases let us understand how many bits will be required and what will be the size of the decoder that is required. So, we have m_1 first group 25. So, 2^5 that is 25 is less than 32. So, 5 bits similarly this is 15 which is less than 2^4 . So, it will be 4 this is less than 2^6 . So, it will be 6 this will be 2^3 less than that. So, it will be 3 and this will be 4. So, the control word size will be $5 + 4 + 6 + 3 + 4$ which is 22. So, we if we just need 7 bits it will be much slower we cannot afford to have this, and if we take 100 bits the space is too large. So, this is quite a feasible thing that we can take up, that is horizontal that is diagonal encoding where we exploit both the features.

So, we came to the end of lecture 20 where we have discussed about the various ways we can generate control signals and by this we have also come to the end of control unit design, but in next 2 lectures, we will be looking particularly how the control unit of MIPS is designed.

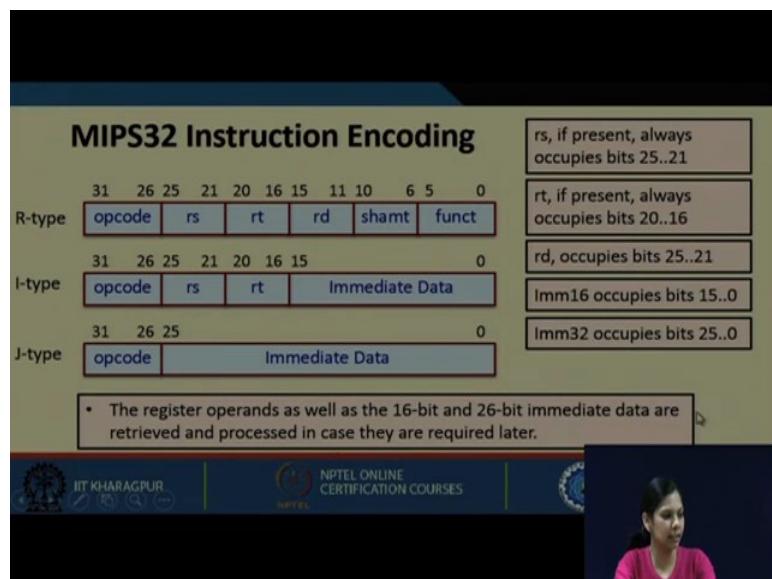
Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture – 21
MIPS Implementation (Part 1)

Welcome to the next lecture. Till now what we have seen that how we can design a control unit. We have also seen both hardware and microprogrammed control unit design methods. So, in this lecture we will be now looking into MIPS implementation. We have seen the general way of designing a control unit. We will be in this lecture specifically seeing how in MIPS the data path is there, how the instructions are actually getting executed with respect to MIPS instruction set.

(Refer Slide Time: 01:14)



We have already discussed this in week 2 lectures, MIPS32 instruction encoding. So, we have R type instruction, we have I type instruction and we have J type instruction. In R type instruction, we have an opcode field, we have 3 register fields (2 source register, one destination register), we have shift amount value, and this is opcode extension function. In the I type that is immediate type instructions, we have opcode, 2 registers, and we have an 16 bit immediate value.

Similarly, in J type instructions, we have an opcode, and we have a immediate value of 26 bit. So, rs if present always occupies bits 21-25. If rt is present it always occupies bits 16-20. Similarly, rd occupies from bits 11-15.

This immediate field contains bits 0-15; it is 16 bit. This immediate field occupies 0-25; it can be extended by 2 more bits, later we can see. So, the register operands as well as 16 bit and 26 bit immediate operand are retrieved and processed in case they are required later. So, as you see that this is the opcode. These are source register and destination register, these are the immediate data. So, what we can do we can retrieve these data in advance.

(Refer Slide Time: 03:48)

A Simple Implementation of MIPS32

- We consider the integer instructions and data path of MIPS32.
- Basic idea:
 - Different instructions require different number of register operands and immediate data (16 bits or 26 bits).
 - Relative positions of register encodings and immediate data are the same across instructions.

So, in a simple implementation of MIPS, we consider the integer instructions and data path of MIPS. So, what the basic idea goes here is that different instructions require different number of register operands. And relative positions of the register encoding and immediate data are the same across instructions. So, by this what we mean that we use any instruction, does not matter it can be a J type it can be I type. But this is fixed that from this particular bit to this particular bit, it will have this particular data; from this particular bit to this particular bit, it will be this particular register; from this to this it will be another register. So, that is fixed. So, this information is known to us.

(Refer Slide Time: 04:43)

The slide has a dark blue header and footer. The main content area is light beige. It contains a bulleted list under the heading '• A Naïve Approach:'.

- A Naïve Approach:
 - After fetching and decoding an instruction, identify the exact register(s) and/or immediate operands to use, and handle them accordingly.
 - The number of register fetches and immediate operand processing will vary from instruction to instruction.
 - We do not utilize the possible overlapping of operations to make instruction execution faster.
 - Before instruction decoding is complete, fetch the register operands and immediate data in case they are required later.



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



NATIONAL INSTITUTE OF
TECHNOLOGY, MEGHALAYA

Let us take a naive approach. In a naive approach what happens after fetching and decoding an instruction, we identify the exact register or immediate operand to use and handle them accordingly. What we are saying that we will fetch an instruction, we will decode that instruction, and then we will come to know that this particular instruction performs this particular task. So, we can extract the register, if it is for an immediate operand we can extract the immediate operand. So, we are not doing something well in advance rather after decoding it we are starting to do all these things.

The number of register fetches and immediate operand processing will vary from instruction to instruction; obviously. We do not utilize the possible overlapping of operations to make the instruction execution faster. If we are just fetching one by one, then we really cannot take the advantage of this overlapped execution of instruction, that is pipeline. So, we will not be able to take advantage of that. Why because, we are fetching these instruction one by one, and then we are decoding then we are getting all the other immediate value or register value, etc.

So, before instruction decoding is complete, we fetch the register operands and immediate data in case they are required later. So, this is a better way. Before the decoding is performed, we want to fetch --- we already know that this particular bit will be a register, this particular bit will be an immediate data, this will be destination register, this will be source register. So, why not let us take those, fetch and keep in

some proper place. If it is not required later we will not use it, but at least if it is required, then it will be very easy for us to get the data; we do not have to fetch it again.

(Refer Slide Time: 07:08)

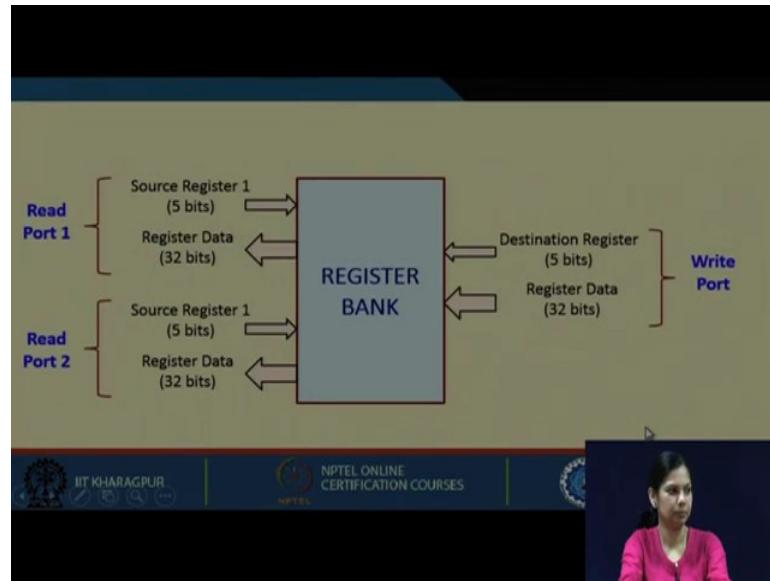
An Assumption

- An instruction can have up to two source operands:
 $ADD \quad R1, R5, R10$
 $LW \quad R5, 100(R6)$
- There are 32 32-bit integer registers, $R0$ to $R31$.
 - We design the register bank in such a way that two registers can be read simultaneously (i.e. there are 2 read ports).
 - We shall later see that performance can be improved by adding a write port (i.e. 2 reads and 1 write operations are possible per cycle).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

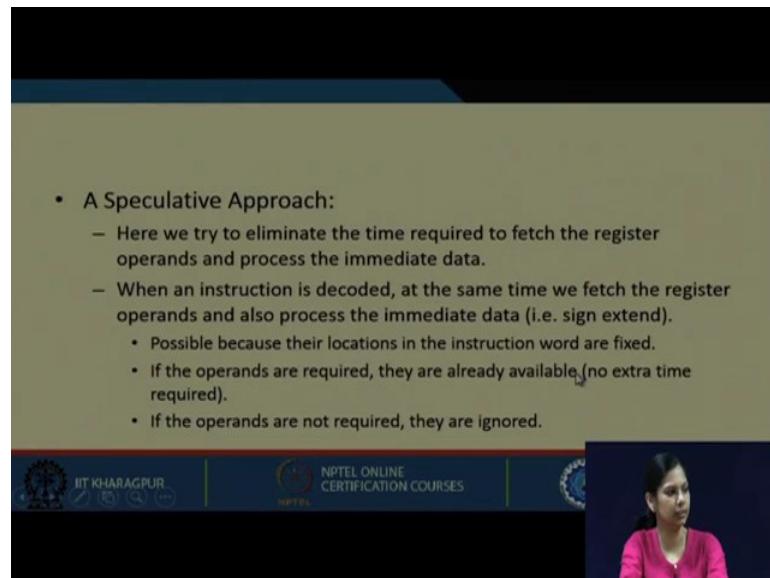
So, this is an assumption that an instruction can have up to 2 source operands. Basically one is ADD R1,R5,R10 and for LW R5,(100)R6. So, there are 32, 32-bit registers, R0 to R31. We design the register bank in such a way that 2 registers can be read simultaneously. That is, there are 2 read ports we already have seen in multibus architecture, that this might be possible that a particular register has two read ports and one write port. We shall see later that the performance can be improved by adding a write port; that is, 2 read and 1 write are possible per cycle.

(Refer Slide Time: 08:10)



So, this is the story. All together we have a register bank where we can read from 2 registers and we can write into 1 register. So, read port 1, read port 2, and we have one write port. So, source register 1 will be 5 bits, and the data will be 32-bit, source register 2 will be 5-bits and the register data can be 32-bit, and this is the destination register.

(Refer Slide Time: 08:42)

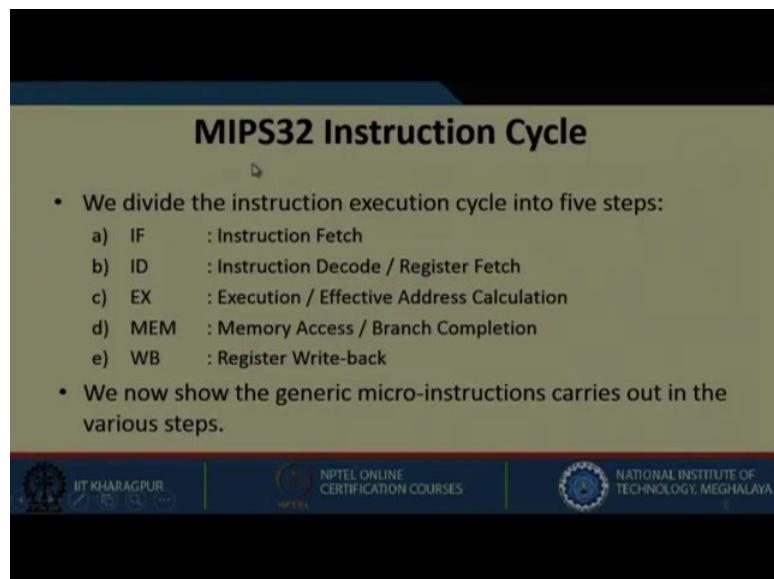


Let us now come to a speculative approach. Let us try to speculate something here; we try to eliminate the time required to fetch the register operands and process the immediate data. So, as we said that when an instruction is decoded at the same time we

fetch the register operands and also process the immediate data; that means, we have already seen that in MIPS architecture the immediate data is sign extended to make it 32 bit.

So, all these things can be done once it is decoded. We really do not know at this point of time whether it is required or not. But still let us do it, because their locations in instruction word are fixed, and because of this fixed location we are able to do this. If the operands are required, they are already available; no extra time will be required because we have already done this fetching; and if the operands are not required, they are simply ignored.

(Refer Slide Time: 10:30)



The slide has a dark blue header bar. Below it, the title "MIPS32 Instruction Cycle" is centered in a light blue box. The main content area is white and contains two bullet points:

- We divide the instruction execution cycle into five steps:
 - a) IF : Instruction Fetch
 - b) ID : Instruction Decode / Register Fetch
 - c) EX : Execution / Effective Address Calculation
 - d) MEM : Memory Access / Branch Completion
 - e) WB : Register Write-back
- We now show the generic micro-instructions carried out in the various steps.

At the bottom, there is a footer bar with three logos and text: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Now, MIPS32 instruction cycle is divided into certain steps. So, what are the steps? Instruction fetch, instruction decode or we can say register fetch, execute where effective address calculation is also done, this is memory access and branch completion, and write back to a register. We now show the generic micro instructions that are carried out in the various steps.

(Refer Slide Time: 11:14)

(a) IF : Instruction Fetch

- Here the instruction pointed to by *PC* is fetched from memory, and also the next value of *PC* is computed.
 - Every MIPS32 instruction is of 32 bits (i.e. 4 bytes).
 - For a branch instruction, new value of the *PC* may be the target address. So *PC* is not updated in this stage; new value is stored in a register *NPC*.

IF:

```
IR ← Mem[PC];
NPC ← PC + 4;
```

The video player shows a woman in a pink shirt speaking. The footer of the slide includes logos for IIT Kharagpur and NPTEL, and the text "NPTEL ONLINE CERTIFICATION COURSES".

In the instruction fetch what happens, we know we fetch an instruction. As we have already seen for single bus architecture. Here also it is pretty same, but it is specific to MIPS. Here the instruction pointed to by PC is fetched from memory, and also the next value of PC is computed. So, every MIPS instruction is 32 bits, that is 4 bytes. For a branch instruction, new value of the PC may be the target address. So, PC is not updated in this stage; the new value is stored in a register called NPC.

So, this is little bit different than we have done earlier. What we are doing we are updating the PC value, and if there is a branch later at that point, we are doing Yin at certain stage, and that Yin can be added with that particular offset to go to the branch location. There we were doing like this. But in MIPS, we are updating the PC value, but not updating it in the PC register. They are adding the PC value, and the new value is stored in another register. So, for this purpose they have kept another register called NPC, where the updated PC value is stored and not in the PC. So, we do $\text{Mem}[\text{PC}]$. So, the content of memory location pointed by PC is brought into IR, and PC is incremented by 4 and it is stored in NPC.

(Refer Slide Time: 13:16)

(b) ID : Instruction Decode

- The instruction already fetched in *IR* is decoded.
 - *Opcode* is 6-bits: bits 31..26, with optional *function* specifier: bits 5..0
 - First source operand *rs*: bits 25..21, second source operand *rt*: bits 20..16
 - 16-bit immediate data: bits 15..0
 - 26-bit immediate data: bits 25..0
- Decoding is done in parallel with reading the register operands *rs* and *rt*.
 - Possible because these fields are in a fixed location in the instruction format.
- In a similar way, the immediate data can be sign-extended.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Now, let us see what happens in instruction decode. The instruction already fetched in IR is now decoded. As we said that the opcode is a 6-bit from 0 to 5. These are also stored; first the source operand rs, second the source operand rt, 16-bit immediate data and 26-bit immediate data are also fetched. So, all these are fetched we do not know whether we will be requiring it or not, but we have fetched it decoding is done in parallel with reading the register operand rs and rt. And this is done within the processor, because our instruction is in IR, and from IR we are taking it one by one. Possible, because these fields are in fixed location in the instruction format. In a similar way the immediate data can be sign extended. So, the immediate data can be sign extended to make it 32-bit.

(Refer Slide Time: 14:29)

The slide shows a block of pseudocode:

```
ID: A ← Reg[rs];
B ← Reg[rt];
Imm ← (IR15)16 ## IR15..0 // sign extend 16-bit immediate field
Imm1 ← IR25..0 ## 00 // pad 2 0's to 26-bit immediate field
```

A note below the code states: *A, B, Imm, Imm1 are temporary registers.*

The footer of the slide includes logos for IIT Kharagpur and NPTEL, along with the text "NPTEL ONLINE CERTIFICATION COURSES". A video camera icon is also present.

So, this is what we are doing. In A we are bringing Reg[rs], in B we are bringing Reg[rt]. Immediate data which is 0 to 15 we are sign extending with the first bit, that is IR₁₅ 16 times. And the next immediate field is padded with 2 zeros. So, this 26 bits are kept. So, Imm and Imm1 are temporary registers that are loaded in the instruction decoding phase with these particular values.

(Refer Slide Time: 15:29)

The slide title is **(c) EX: Execution / Effective Address Computation**.

- In this step, the ALU is used to perform some calculation.
 - The exact operation depends on the instruction that is already decoded.
 - The ALU operates on operands that have been already made ready in the previous cycle.
- We show the micro-instructions corresponding to the type of instruction.

The footer of the slide includes logos for IIT Kharagpur and NPTEL, along with the text "NPTEL ONLINE CERTIFICATION COURSES". A video camera icon is also present.

Let us see what happens in execution phase. In the execution phase, the effective address computation is performed. So, in this step the ALU is used to perform some

calculation. The exact operation depends on the instruction that is already decoded. The ALU operates on operands that have been already made ready in the previous cycles. We have already fetched and kept it in A and B registers in the decoding phase, and it is an ALU operation; then finally, in the execute phase the operation specified can be performed. So, we show the micro operations corresponding to the type of instruction.

(Refer Slide Time: 16:25)

Memory Reference:	Example: LW R3, 100(R8)
$ALUOut \leftarrow A + Imm;$	

Register-Register ALU Instruction:	Example: SUB R2, R5, R12 [operation specified by func field (bits 5..0)]
$ALUOut \leftarrow A \text{ func } B;$	

Register-Immediate ALU Instruction:	Example: SUBI R2, R5, 524 [operation specified by func field (bits 5..0)]
$ALUOut \leftarrow A \text{ func } Imm;$	

Branch:	Example: BEQZ R2, Label [op is ==]
$ALUOut \leftarrow NPC + (Imm << 2);$ $cond \leftarrow (A \text{ op } 0);$	

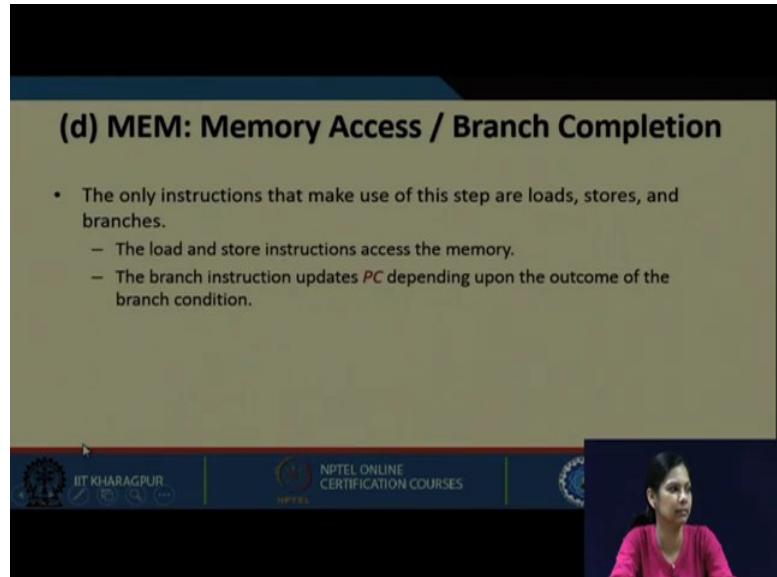
Now, in execute phase what can happen? If it is a LW then this is an immediate value added with R8. So, R8 goes in A, added with the immediate value that is 100 ,which goes in the output of ALU. For register to register, both A and B are present for this instruction; R5 and R12 both are present, it will be added or subtracted depending on the function and it will be stored in ALUOut.

Similarly, register immediate. In this case, R5 is subtracted with an immediate value. So, R5 is available in A, and this immediate value is available in Imm, and this function that is subtraction can be performed and the output is available in ALUOut.

Now, for branch, what happens? For the branch the immediate value that we have got it needs to be left shifted twice, and then added with NPC because NPC contains the incremented value that will come to ALUOut. And if we have a branch instruction like BEQZ; that means, if R2 equal to 0 then only branch. We have to check for this condition, and this condition will be an operation with 0. So, if this particular operation if

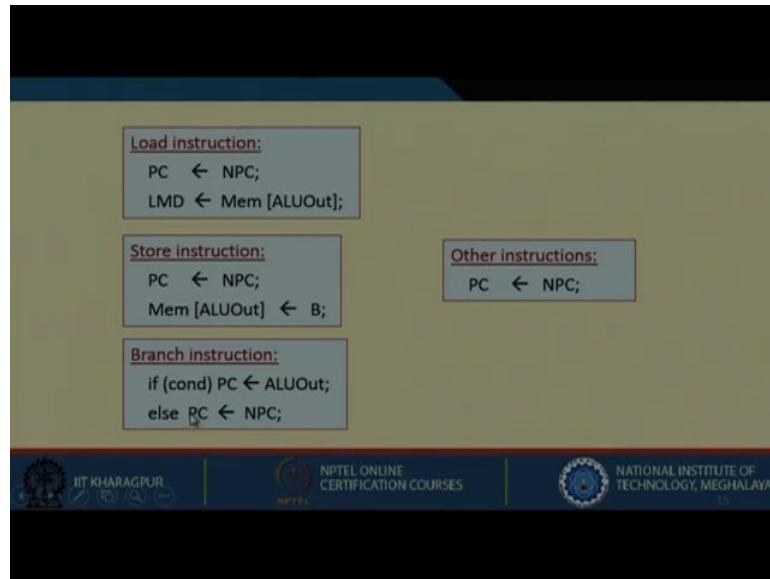
this condition is satisfied then you will branch. So, it will do some operation and it will set the condition and accordingly branch will take place.

(Refer Slide Time: 18:33)



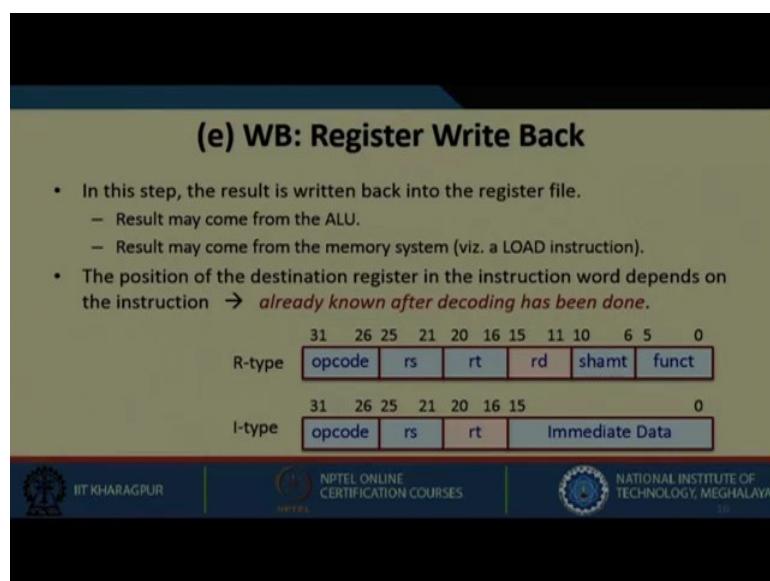
Now, what happens in MEM --- memory access or branch completion? The only instructions that make use of this step are load and store, and of course branches. The load-store instruction accesses the memory. So, the memory operation will actually happen here. The branch instruction updates PC depending upon the outcome of the branch condition. So, this also happens here. So, these are the two things that happen in MEM phase.

(Refer Slide Time: 19:06)



So, now NPC will be loaded in PC, and for the load instruction output of ALU location pointed by that will come to LMD. Similarly, NPC will come to PC, and then B will be put into that particular location whose address is in ALUOut; and for the branch what happens if the condition is satisfied, then that ALUOut will go to PC, and else NPC, which we have already calculated will be loaded to PC. And for all other instruction, we have already calculated the PC value in NPC, which will be put in PC.

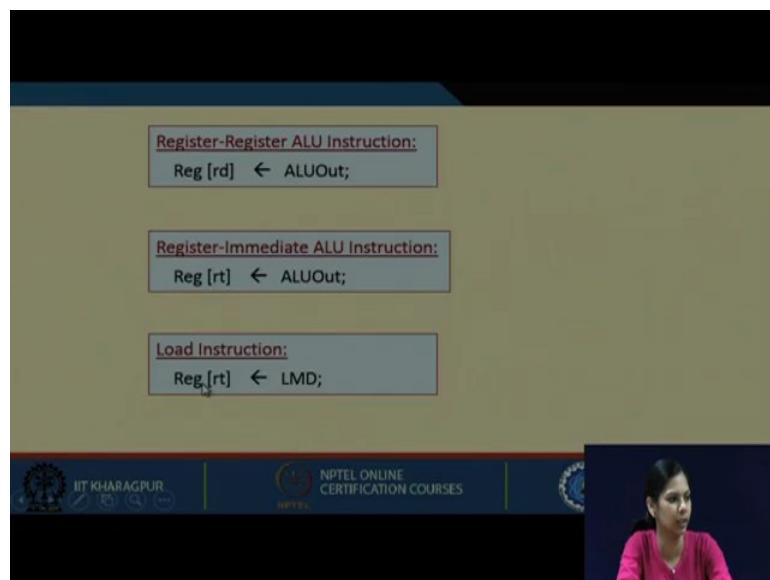
(Refer Slide Time: 20:25)



Let us see in Write Back what happens; register write back occurs in this step. The result is written back into the register file. Result may come from memory system via load as well. The position of the destination register in the instruction word depends on the instruction already known in the decoding phase. So, this is basically the destination register in this particular case, and this is the destination register.

So, the position of the destination register we already know from the decoding phase, and then the result may be put in there in this particular step.

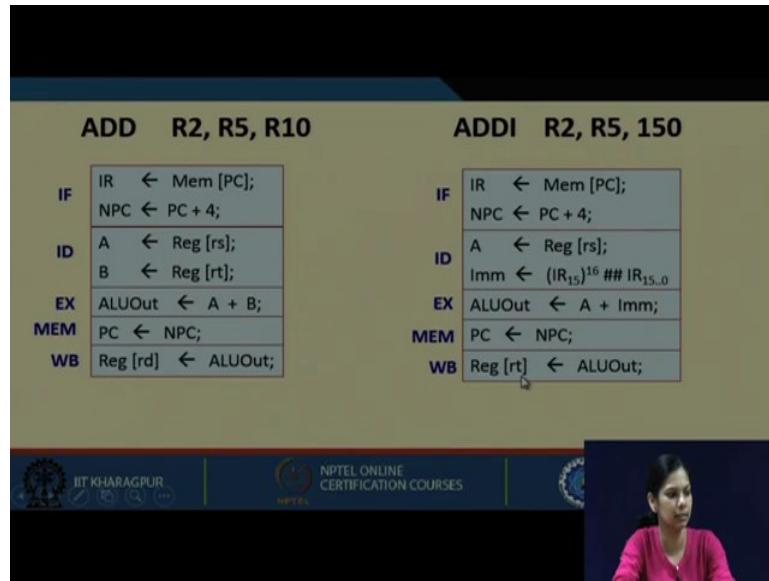
(Refer Slide Time: 21:16)



So, whatever value was there in ALU can be put in for register transfer. Here also ALUOut will be put in Reg[rt]; and in load instruction in LMD we have stored that value, now it will go to the required register.

Let us see some example instructions. Now we have seen step by step how in MIPS the instructions are executed.

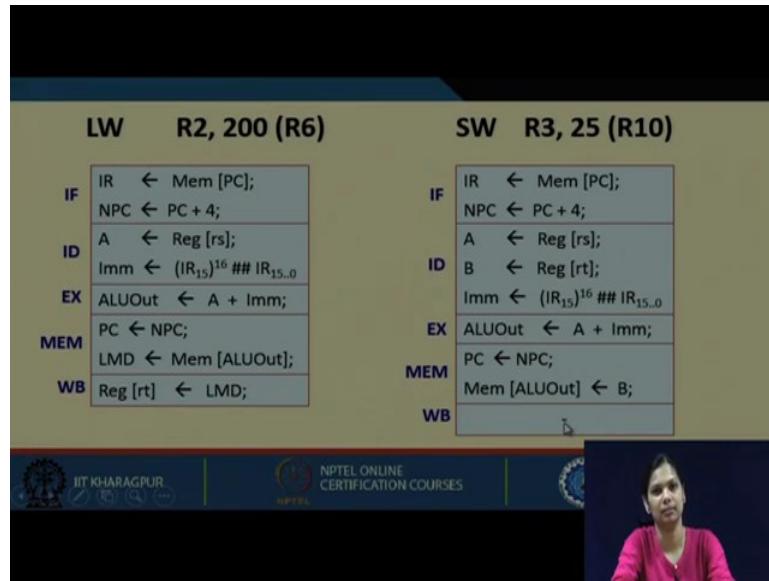
(Refer Slide Time: 22:09)



Now, let us see a complete execution of an instruction ADD R2,R5,R10. So, in the instruction fetch phase IR will have Mem[PC]. So, this entire instruction is in IR, and NPC will have PC plus 4. Similarly A will have Reg[rs], B will have Reg[rt]. And in the execute phase both the source operands are added, and it is available in ALUOut. And then in the MEM phase the PC is loaded with NPC value, and finally, the value of ALUOut will be put it into the destination register, that is R2 here.

So, in five steps we are executing it, but for all instruction all these steps will be required, let us see how we can add an immediate value. In this case, this is an immediate value. So, this immediate value which is 16-bits concatenated with a MSB value, and we get the total 32-bit immediate value, and A is R5 here. Finally, ALUOut we will be adding this value with immediate value, and NPC is loaded in PC in this particular step, and finally, we write back the output into R2 that is the destination register here.

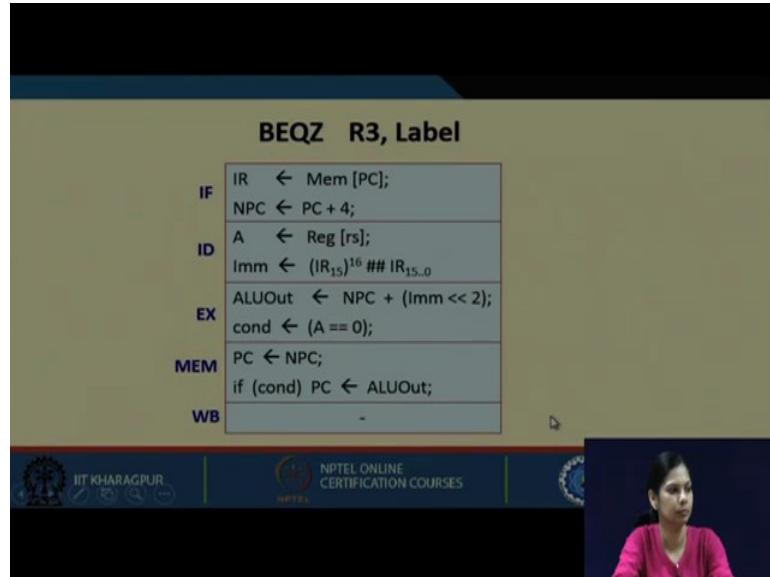
(Refer Slide Time: 24:01)



Now for load instruction For load instruction, we can see that similarly in instruction fetch these 2 steps will be performed, A will have the source operand, that is R6 here, immediate value will be stored in Imm, we will add R6 with 200 and it will be stored in ALUOut. And then NPC will be put in PC, and memory operation is performed here. So, the output of ALUOut from this particular memory location, we need to read the value, that is what we are doing which is loaded in LMD. Now LMD contains the value that should be put in R2. So, LMD will be stored back here.

Similarly, for storing what we have to do, we fetch and decode. After decoding same way we are adding this and this immediate value, and this register value NPC is loaded to PC. And now instead of getting it from the memory, what we are doing? The value of B that is in R3 is stored in Mem[ALUOut], because ALUOut is the location. We are storing B into Mem[ALUOut], and in Write Back there will have nothing to store.

(Refer Slide Time: 26:10)



Let us see the next instruction branch if equal to 0. So, what we are doing here. So, in the instruction fetch and in the decode in the same way we are fetching it. In ALUOut what we are doing? We are adding NPC with the immediate value, which is left shifted twice and finally, we are putting that condition based on R3. R3 is loaded in A. If we are checking this condition if R3 equals to 0 or not, according to this cond will be set. If it is 0, then only branch will takes place. So, otherwise NPC will be PC. So, if the condition is met then ALUOut will be put in PC, else this NPC will be in PC, and there will be nothing in the Write Back phase. So, this is how the branch instruction is executed in MIPS.

So, we have come to end of lecture 21. In this particular lecture, we have seen that how the instructions are executed in MIPS architecture. How the data part is there in MIPS and how the instructions are executed.

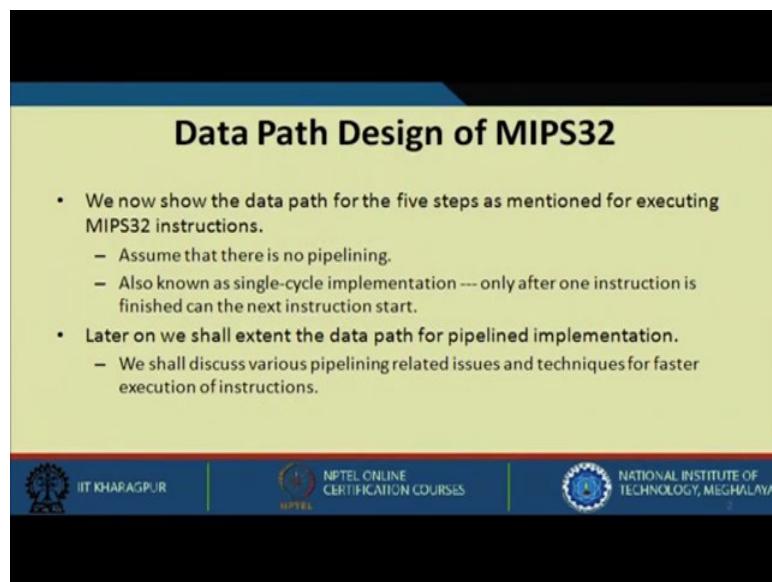
Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture – 22
MIPS Implementation (Part 2)

Welcome to the last lecture of week 4. So, we have already discussed about MIPS implementation. We will continue with that here.

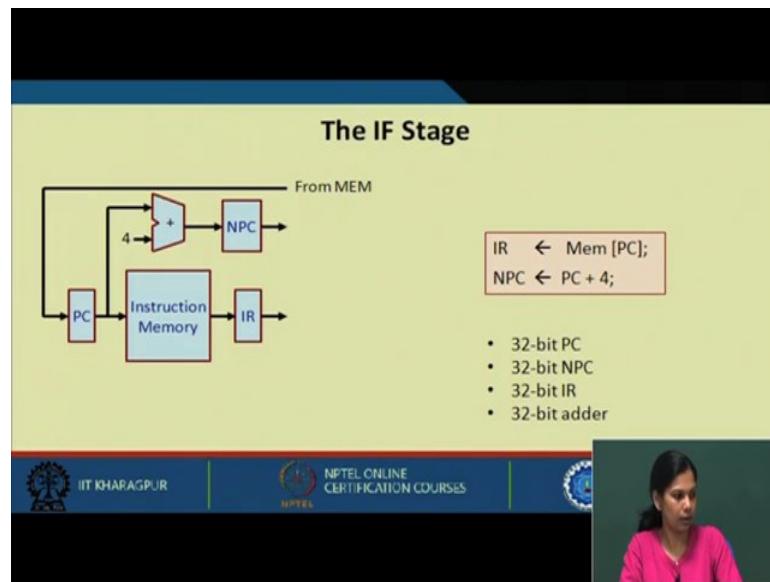
(Refer Slide Time: 00:39)



First, the data path design of MIPS. Individually, we have seen that how we are actually executing a particular instruction, what are the micro-operations that are required to be executed. How the data path is actually designed we will see here.

We now show the data path for the five steps as mentioned for executing MIPS32 instructions, and we are assuming here that there will be no pipelining. This is also known as single-cycle implementation; only after one instruction is finished, can the next instruction start. In a single cycle we execute one instruction fully. And then we move to the next instruction, and again we move to the next instruction after executing the previous one. Later we shall extend the data path for pipeline implementation; we shall discuss various pipelining issues related to the techniques for faster execution of instruction in later phase of this particular course.

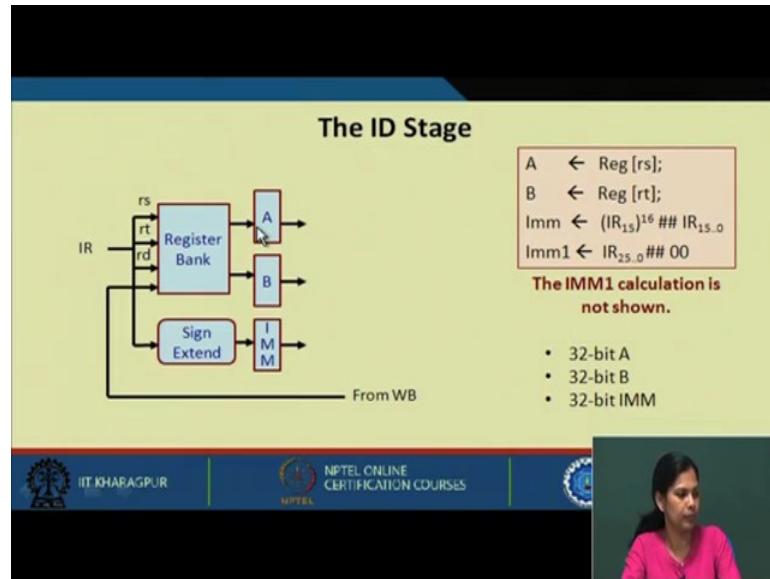
(Refer Slide Time: 02:04)



Now, let us see how the data path will be for the various stages. First take into consideration the IF stage. So, in the IF stage, from Mem[PC] we load the content of the memory location pointed by the PC. We will fetch the data and put it in IR; at the same time PC will be incremented by 4, and it will be stored in NPC. So, we require a 32-bit PC, a 32-bit NPC, a 32-bit instruction register, and of course, a 32-bit adder.

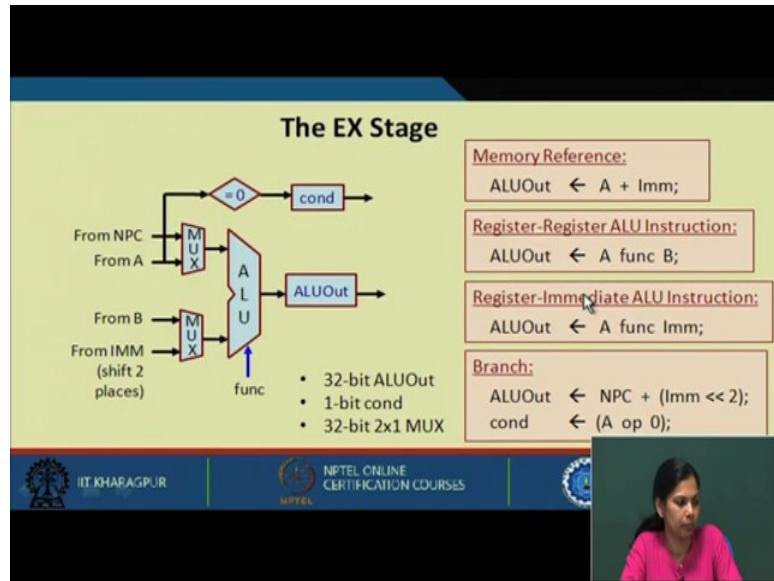
So, now we see that this PC will hit the instruction memory. PC contains the address of the next instruction to be executed. So, it will hit the instruction memory and the data will be available in IR, and this PC is added with a constant 4 and this is stored in NPC. So, these two operations can be performed using the following data path that is there.

(Refer Slide Time: 03:49)



Next move on with the ID stage. In the ID stage, what we said we decode that instruction, and after decoding that instruction we also fetch the particular fields that are the registers --- source register, the destination register, the immediate field, etc. and we make it available for that particular instruction execution. So, from the register bank this rs, rt and rd from IR it is coming, and from this register bank it is going to A and B; and after sign extension it is going to IMM. So, one of the source register values is coming depending on rs, another source register value is coming from rt, the immediate value which is sign extended to make it 32-bit if stored in IMM; Imm1 calculation is not shown; it is pretty similar.

(Refer Slide Time: 05:21)



Now, let us see what happens in the EX stage. So, we are looking here step-by-step fetch, decode, then execute, and for execute how the data path will be. For the execution we know that there can be various kinds of operations that can be present; one can be memory reference, another can be register-register ALU instruction, register-immediate ALU instruction, and branch instruction.

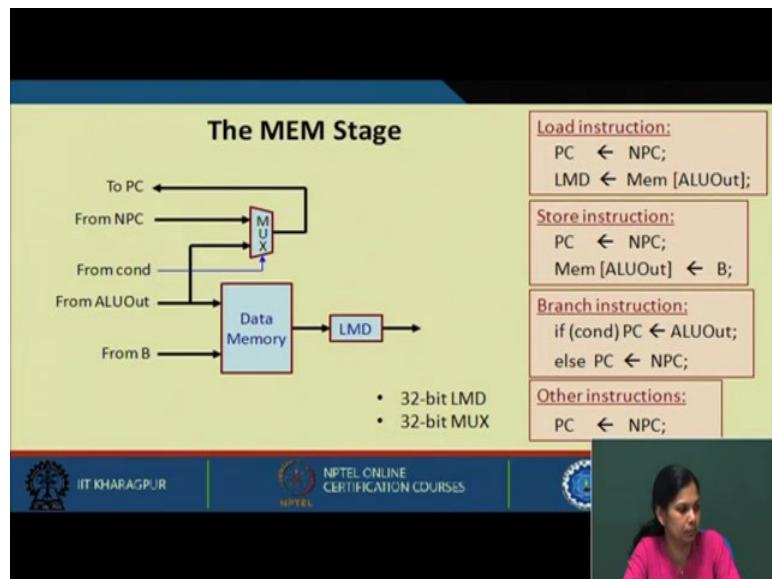
So, for all these kinds of instruction we see that this particular data path will be sufficient. We need to add with the immediate value. So, there is a MUX which is attached to this ALU is having two inputs, and in this ALU one is coming from NPC another is coming from A. So, depending on the select line this can be taken care which input will be going. So, for memory reference A will come in, accordingly the select line will get activated, and immediate value will be coming. So, here we can see that A and B are coming from these two MUX, and along with B that immediate value can also come in here, which is available after ALU operation. The ALU output is available at ALUOut. So, we are doing this operation and putting it in ALUOut. So, 32-bit ALUOut, 1-bit cond is required for condition checking, and 32-bit 2x1 MUX is also required.

Now, let us see for register-register ALU operation. So, if it is register-register then input A should go and B input should go; in that case the select line will be selected and A input will go to one input of ALU, and B input will go to another input of ALU, and the operation will get performed. Similarly for register-immediate ALU operation, A will

come in from the first MUX to one of the inputs of ALU, and Imm will come from another MUX and will go in into the next input of the ALU. And for branch the NPC value must be added with the immediate value by shifting it to left 2 positions.

So, in that case from this MUX NPC value will come in, and from this MUX this Imm value will go in, and the operation will be performed similarly for the cond. The cond will be available based on this $A \text{ op } 0$. So, A will be operated with 0. So, input is coming, will be checked with 0, and the condition will be outputted here. So, all these operations are actually performed in the execute stage using the following data path.

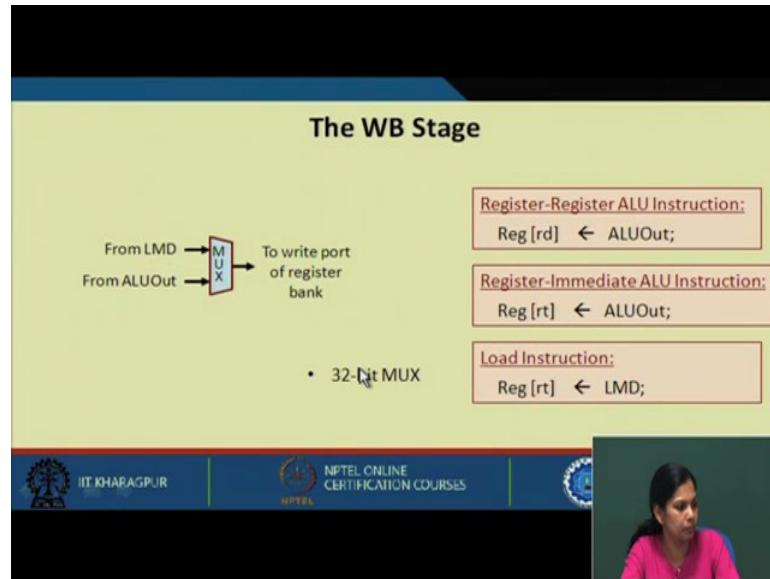
(Refer Slide Time: 09:30)



Let us move on Mem stage. In the Mem stage what basically happens which all instruction will require --- memory load operation, store operation, branch operation, and for all other operation NPC is stored in PC in this particular Mem stage. So, let us see the data path for Mem stage. So, what happens this is the data memory from where we will fetch the data, now we know that if we need to read it from a particular location we need to put that address in MDR basically and then we hit the data memory.

So, in this case ALUOut will be put in to the data memory, and then the value will be read and can be stored in LMD. So, for the load operation this happens where memory location pointed by ALUOut is read, and it is stored in LMD. And in all cases we can see that updated value of PC in NPC is stored in PC, so, similarly all other operation can be taken care using this particular data path.

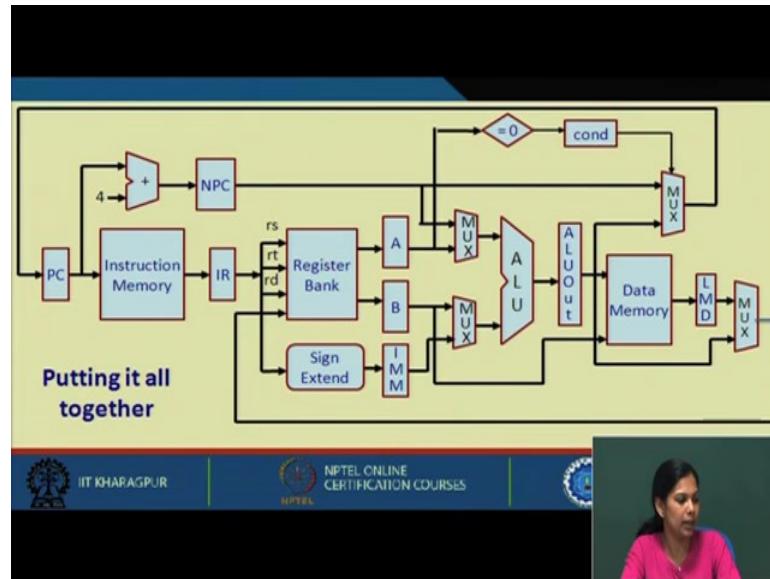
(Refer Slide Time: 12:04)



Let us move on with the WB stage. In the WB stage what happens, we write the value into the register. So, finally, whatever value; either that value is coming from the ALUOutput or the value is coming from memory, it should be written back into the destination register. So, in some cases rd is the destination register, and in some cases rt is the destination register. So, accordingly for register-register ALU operation, this ALUOutput is stored in rd; for register-immediate ALU operation ALUOut stored in rt. And for load instruction the LMD value will be stored in rt. So, this is how it can be done it can be selected from the MUX either from LMD or from ALUOut.

Till now we have seen that the various data path that is required in the various stages in IF, ID, EX, MEM and WB. So, in 5 stages we can see that; what is the data path that is required in MIPS.

(Refer Slide Time: 13:42)



Now, let us put it together all. So, whatever we have discussed starting from IF to ID then to EX and then to MEM and finally, to WB. This overall picture gives you an idea how it is actually happening. This is the data path that is there for MIPS. So, here initially the PC hits the instruction memory; the instruction is read and stored in IR. After it is stored in IR the instruction needs to be decoded; at the same time when it gets decoded some of the other register gets populated, that is, A, B, Imm, etc. And next we execute it. For execution it is required that we get the data from A and B, or from an immediate value; accordingly MUX are in place to select any one of the values out of the two which is fed to the ALU, and from ALU we are getting ALUOut.

Now, once the data is available in ALUOut it is some time required to get the data from memory. So, in that case it is again hit to the data memory, and the data is read and it is stored in a register known as LMD. Finally, from LMD it is written to the register bank in the WB step. So, all the steps that we have seen in bits and pieces in previous slides, here we have put them all together. So, this is the overall picture.

(Refer Slide Time: 16:20)

Simplicity of the Control Unit Design

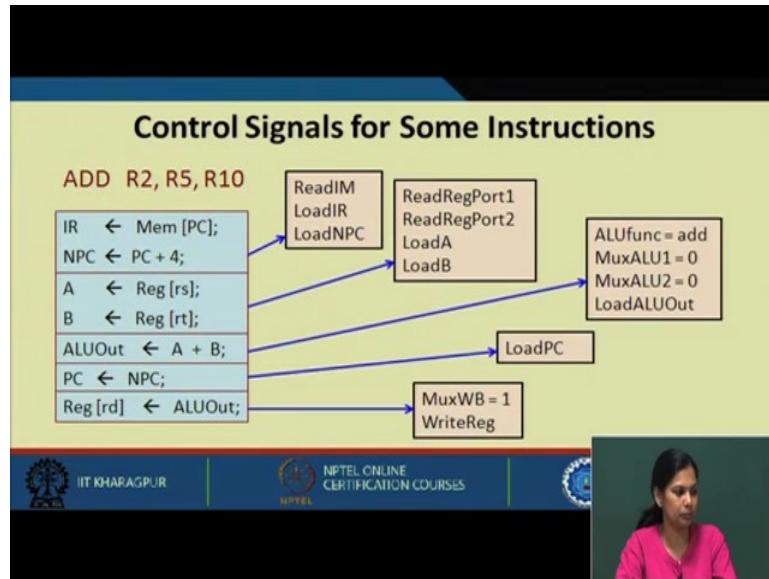
- Due to the regularity in instruction encoding and simplicity of the instruction set, the design of the control unit becomes very easy.
- Control signals in the data path:
 - a) LoadPC i) LoadIMM q) LoadLMD
 - b) LoadNPC j) MuxALU1 r) MuxWB
 - c) ReadlM k) MuxALU2 s) WriteReg
 - d) LoadIR l) ALUfunc
 - e) ReadRegPort1 m) LoadALUOut
 - f) ReadRegPort2 n) MuxPC
 - g) LoadA o) ReadDM
 - h) LoadB p) WriteDM

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, what is the simplicity of the control unit design here. You see that there is a regular structure here; that means, all the instruction we know are of fixed length; we know that this particular field is for this particular register, that is source or destination, this is an immediate field, and so on and so forth. So, we actually understand that for MIPS the instructions are very regular. So, all the instruction will be pretty much same depending on what kind of instruction it is. So, because of this due to the regularity in instruction encoding and simplicity of instruction set, the design of control unit becomes very easy.

So, control signals in the data path are as shown. Using these control signals in the data path, we can generate all the control signals for any instruction.

(Refer Slide Time: 18:40)

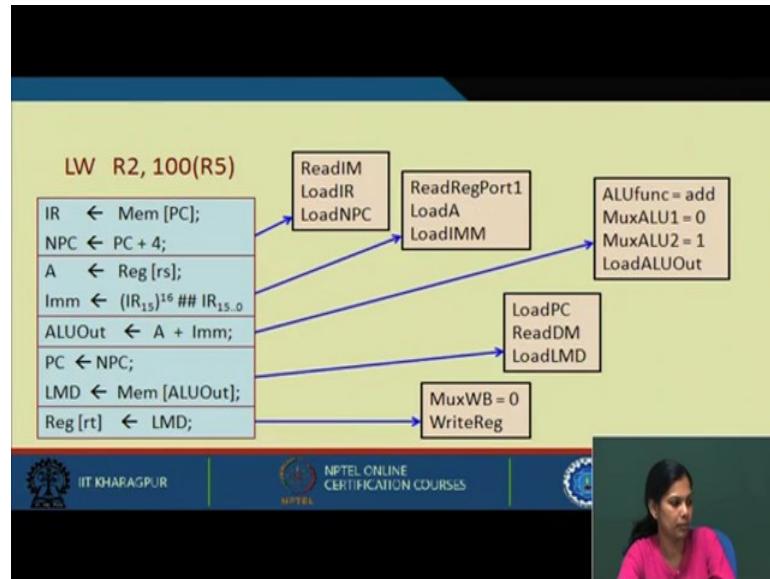


Now, let us see this control signals for some of the instructions. You recall our discussion for single bus architecture; we have seen how corresponding to the data path the control signals are generated, and how we can execute it. Similarly here also we have shown that these are the control signals available for MIPS. Now we will show how a particular instructions gets executed using those control signals. Take ADD R2,R5,R10. So, here we fetch the instruction: ReadIM, LoadIR, and then LoadNPC. So, in the first fetch step, these are the control signals that are activated.

Now, after doing, so, in the next step we are loading ReadRegPort1, ReadRegPort2 after reading from here load it to A and B, that is the decode phase. At this phase we are also decoding, and we are also fetching the value from the instruction and storing it in A and B. Now we are doing the exact ALU function. So, here ALUfunc will be add; this is add operation, MuxALU1 will be 0, we need to select A so MuxALU2 will also be 0, we need to select B and then after this operation LoadALUout. So, ALUOut will be loaded with the operation performed.

Now, here are the operations performed in the Mem phase. In the Mem phase we load PC. So, PC is loaded with the new value of the NPC and finally, in the WB phase the output of ALU that was present will be written to R2; in this MuxWB will be set to 1, and WriteReg will write it into the destination register. So, to execute this particular instruction these are the micro operation or control signals generated.

(Refer Slide Time: 22:09)



Let us see another instruction that is LW R2,100(R5); that means 100 and R5 will be added then the value will be read from memory and it will be loaded in R2.

So, at different phases different things will happen. So, let us say this is the fetch phase. The fetch phase is standard. In the decode phase register value of rs will be loaded in A, and of course, other values will also get loaded, but here we are showing which are required for us which are needed for us. So, Imm will actually store the IR value 0 to 15 bit, and it is sign extended to make it 32-bit. We have to add this Imm with A, and we have to store it in ALUOut. So, ALUOut will have A + Imm; we add these two values and finally, this NPC will be loaded in PC. Now the content of ALUOut will be brought in from data memory, and it will be put in LMD; after this is done finally, from LMD it will be put into the destination register rd or rt.

So, let us see the corresponding control signals. So, first we ReadIM, LoadIR and LoadNPC. Similarly here we will ReadRegPort1, we will LoadA and you will LoadImm. Now you see MuxALU2 will be 1; earlier when it was A and B the MuxALU1 was 0 and MUXALU2 was also 0, but now it will be the immediate value. So, in this case MUXALU2 will be 1; the next value will get selected depending on the MUX select line and then we load. So, this will be added and will be loaded in ALUOut.

Next in the Mem phase of course, we will LoadPC that will be done for all the instructions, and now this ALUOut contains the address from where we have to read the

data. So, as this contains the address where to read the data we will read it from the data memory and we will load it to LMD. So, data is now read from the data memory and loaded in LMD in this particular phase.

And finally, we write back we write back the data into Reg[rt]. So, MuxWB will be 0 and we perform WriteReg. When we perform WriteReg, the LMD value is loaded in Reg[rt]. So, this is how we can perform the operation.

So, we have come to the end of lecture 22 and of course, week 4. So, in this week we have seen how we can design a control unit, the various methods that are present in the design of control unit. And by this time we have also shown that for MIPS what is the control unit, how we can design the control unit and so on.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 23
Processor Memory Interaction

Welcome to week 5. In this week we shall be discussing on memory system design. We will look into the various technologies that are used to build the memory that we use in computers. And we will also look into how these memories are used to design and organize them in the system.

(Refer Slide Time: 00:49)

Introduction

- Memory is one of the most important functional units of a computer.
 - Used to store both instructions and data.
 - Stores as bits (0's and 1's), usually organized in terms of bytes.
- How are the data stored in memory accessed?
 - Every memory location has a unique address.
 - A memory is said to be byte addressable if every byte of data has a unique address.
 - Some memory systems are word addressable also (every addressed locations consists of multiple bytes, say, 32 bits or 4 bytes).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

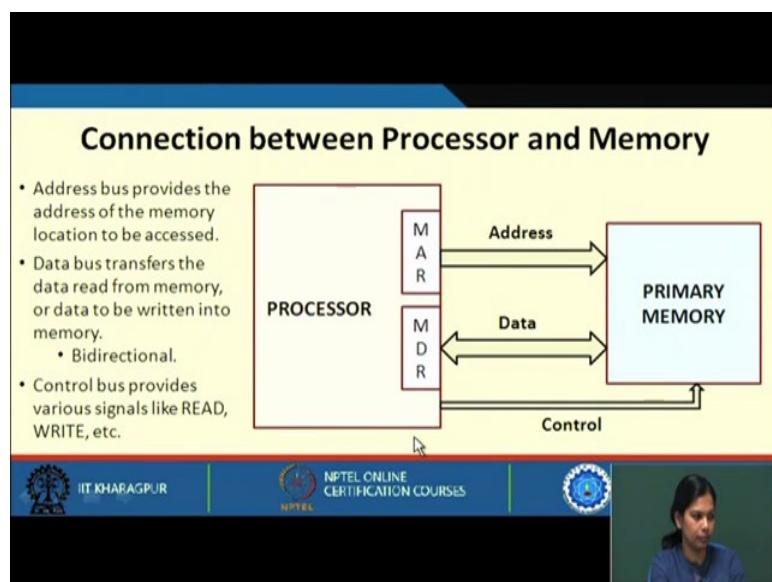
Memory is one of the most important functional units of a computer. We all know that. It is used to store both instructions and data. And it stores bits like 0's and 1's. So, as we have already seen, we encode an instruction with bits of 0's and 1's. So, in the memory location we say we store both instructions and data, those instructions and data are organized in bits of 0's and 1's. And they are usually organized in terms of bytes.

We will see here how are the data stored in the memory are accessed. We need to know the mechanism how we can access the data from the memory. We should also know how we store data into the memory. These are the two things we need to look into. Every memory location has a unique address.

And memory is byte addressable, that is, every byte i.e. a group of 8 bits, has a unique address. Some memory systems are word addressable. And by word addressable we mean that, each location consists of multiple bytes depending on the word size. If one word is 4 bytes or 32 bits, then the memory location will be changed as 0, 4, 8, and so on.

So, if it consists of 8 bytes or 64 bits then the word length is 64, then the memory address will be incremented by 8.

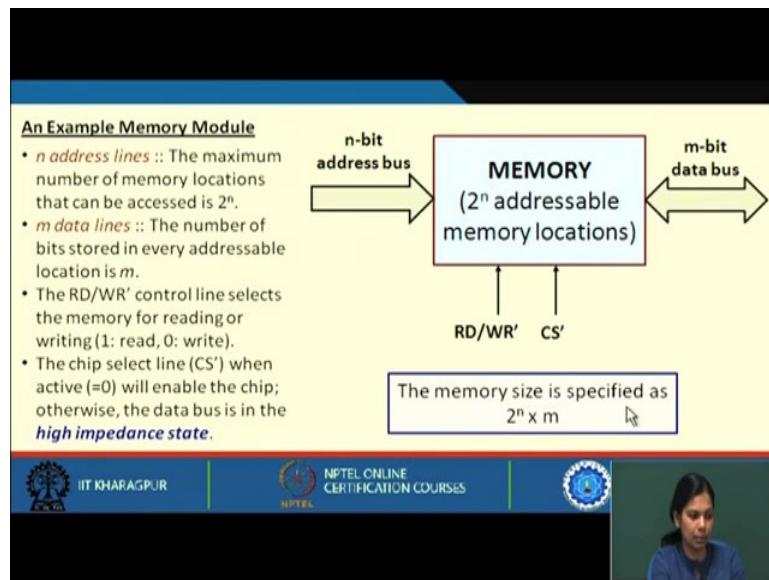
(Refer Slide Time: 03:10)



Now, see the connection between processor and memory. As you know that in processor we have two important registers. One is memory address register (MAR), another is memory data register (MDR). MAR contains the address of an instruction or data that is to be read from the memory, or the address of data that is to be written into the memory. And that particular data which is to be read comes through this data bus. So, this is the data bus, and whatever address is here that address is hit and then from that address whatever data is present that data comes through this data bus, and it comes to MDR. Now you see that the data bus is bidirectional because, we can read the data from the memory. So, the data is coming from the memory to MDR and for write we have to write the data.

So, from the MDR it will go through this data bus into memory. And along with this we also require some control signals, like read, write etc.

(Refer Slide Time: 04:34)



So, if we have a n-bit address bus then, the memory addressable memory location will be 2ⁿ. Like we already discussed if you have a 3-bit address bus then the total number of location will be 2³. So, there will be 8 locations starting from 000, 001, we go on to 111. So n-bit address bus can have a maximum of 2ⁿ addressable memory locations. And we can have a m-bit data bus. So, in that particular address the data that is present is m-bit, and m-bit data at a time can be transferred to memory. And we have other signals like read, write, and chip select.

We will be seeing that why chip select is required in course of time. So, the maximum number of memory location that can be accessed is 2ⁿ. For m-bit data line the number of bits stored in every addressable location is *m*, and the read/write control signal selects the memory for reading or writing. So, for reading it is 1, for writing it is 0. As I said, chip select line; this is active low. So, it is active when it is 0. This will enable the chip when it is 0. Otherwise the data bus is in high impedance state. So, this memory module will not be selected in that case.

So, here we have n-bit address bus. We have 2ⁿ addressable locations, and m-bit data bus. So, the total size of the memory is 2ⁿ x m.

(Refer Slide Time: 06:38)

Classification of Memory Systems

a) Volatile versus Non-volatile:

- A *volatile* memory system is one where the stored data is lost when the power is switched off.
 - Examples: CMOS static memory, CMOS dynamic memory.
 - Dynamic memory in addition requires periodic refreshing.
- A *non-volatile* memory system is one where the stored data is retained even when the power is switched off.
 - Examples: Read-only memory, Magnetic disk, CDROM/DVD, Flash memory, Resistive memory.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Now, classification of memory system; how we can classify a memory system? One way to classify memory system is volatile versus non-volatile, i.e. with respect to volatility. A volatile memory system is one where the stored data is lost when the power is switched off; that means, as long as the power is applied to it the data will remain, but as long as the power is taken off the data goes off; that means, it is volatile it goes off after the power is cut off. CMOS static memory and CMOS dynamic memory both these are volatile memories; that means, as long as power is supplied the data remains.

But in case of dynamic memory, even if we are supplying the power then also it requires periodic refresh. So, data cannot be retained for longer period of time. So, periodic refresh is necessary.

Now, what is non-volatile memory? A non-volatile memory system is one where the stored data is retained even when the power is switched off. So, where you will see such kind of non-volatile memory? We see that in read only memories where, once the data is there it retains. For example, magnetic disk, CDROM, DVD, flash memory, and some resistive memories. These are all non-volatile memories. So, even if the power is not supplied the data will remain.

(Refer Slide Time: 08:36)

b) Random-access versus Direct/Sequential access:

- A memory is said to be *random-access* when the read/write time is independent of the memory location being accessed.
 - Examples: CMOS memory (RAM and ROM).
- A memory is said to be *sequential access* when the stored data can only be accessed sequentially in a particular order.
 - Examples: Magnetic tape, Punched paper tape.
- A memory is said to be *direct or semi-random access* when part of the access is sequential and part is random.
 - Example: Magnetic disk.
 - We can directly go to a track after which access will be sequential.

Again we can differentiate a memory with respect to random access versus direct or sequential access. What do you mean by that random access? By random access we mean that, the read and write time is independent of the memory location being accessed. That means, you either hit location 0 or you hit the last location or the middle location, the access time is same.

So, whichever location you access the access time will be same irrespective of the location. The example is CMOS memory that is RAM and ROM, both are random access. And then what is sequential access? A memory is said to be sequential access, when the stored data can only be accessed sequentially in a particular order. Like, an example is magnetic tape. Here the data are accessed sequentially, one by one.

A memory is also said to be direct or semi-random access when, a part of the access is sequential, and a part is random; like your magnetic disk. Here we can directly go to a particular track, but after reaching that particular track we have to sequentially get the data one by one. This kind of memory is semi-random access, which is somewhat sequential, somewhat random.

(Refer Slide Time: 10:26)

c) **Read-only versus Random-access:**

- *Read-only Memory* (ROM) is one where data once stored is permanent or semi-permanent.
 - Data written (programmed) during manufacture or in the laboratory.
 - Examples: ROM, PROM, EPROM, EEPROM.
- *Random Access Memory* (RAM) is one where data access time is the same independent of the location (address).
 - Used in main / cache memory systems.
 - Example: Static RAM (SRAM) → data once written are retained as long as power is on.
 - Example: Dynamic RAM (DRAM) → requires periodic refreshing even when power is on (data stored by charge on tiny capacitors).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Next let us see read only versus random access. What is read only memory? Read only memory is one where the data once stored, is permanent or semi permanent. What do you mean by permanent, what you mean by semi permanent? By permanent we mean that once we write into it, no changes can be made to it. And semi permanent means when we write, it remains, but if later we want to change it we can still do it. It remains permanent for a period of time. And again, if you want to change it we can change it and then it will again remain. So, the examples are PROM programmable read only memory, Erasable programmable read only memory, and electrically erasable programmable read only memory. So, these are all classes of read only memory, where the data are written during manufacturing, but can be changed later also.

So, ROM comes first, then PROM, then EPROM, and so on. Now random access memory is one where the data access time is same independent of the location. So, we access the first location or the last location the access time will be same. And where it is used? We will be talking extensively about your main memory and your cache memory. So, in both the memories such kind of memory, that is random access memory, are used. Some of the examples are static RAM.

Here once the data is written it retains as long as the power is supplied to it. And dynamic RAM is having the same feature of a RAM, but even if the power is supplied to it, it requires periodic refresh. So, periodic refresh is required, even if the power is

supplied to it. And here the data is stored as charge on tiny capacitors. We will be looking into more details of static RAM and dynamic RAM in course of time.

(Refer Slide Time: 13:15)

Access Time, Latency and Bandwidth

- Terminologies used to measure speed of the memory system.
 - Memory Access Time:** Time between initiation of an operation (Read or Write) and completion of that operation.
 - Latency:** Initial delay from the initiation of an operation to the time the first data is available.
 - Bandwidth:** Maximum speed of data transfer in bytes per second.
- In modern memory organizations, every read request reads a block of words into some high-speed registers (LATENCY), from where data are supplied to the processor one by one (ACCESS TIME).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

At this point of time, we need to know some of the terminologies that we will be using it very often. They are called access time, latency and bandwidth.

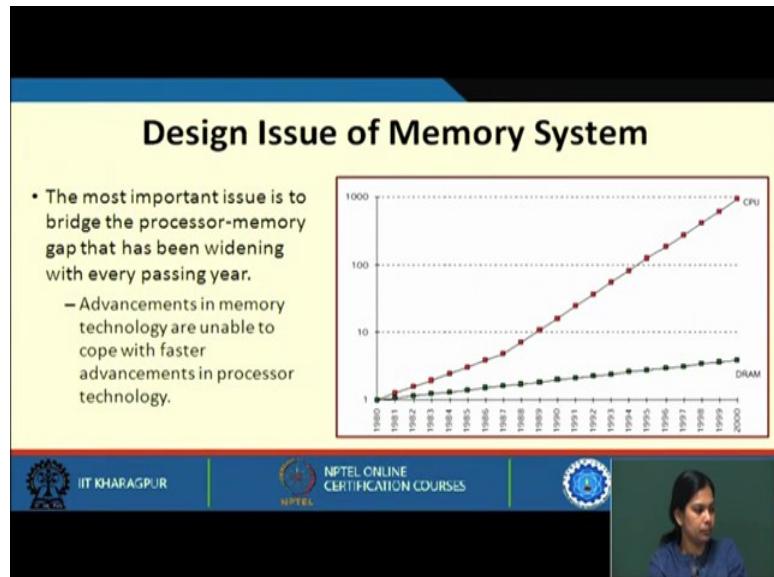
So, what is memory access time? By memory access time we mean that the time between initiation of an operation, either it can be read or write, and the completion of that operation. How much time it is required to access the particular data?

Next is latency. Latency is the initial delay from the initiation of an operation to the time the first data is available. Let me tell one thing at this point of time, that when we access a particular location in the memory, we do not just access or retrieve that particular data. We always transfer a block of data. That is why this latency is an important term because it will give the time required to access the first data. And then the subsequent data that are present can be accessed in a much faster rate.

So, latency is the initial delay from the initiation of an operation to the time the first data is available. And what is bandwidth? Bandwidth is the maximum speed of data transfer in bytes per second. In modern memory organization every read request reads a block of words into some high-speed register. That is when the first word is available. And from

then the data are supplied to the processor one by one. So, the total access time will be depending on not only a single word, but a block transfer.

(Refer Slide Time: 15:45)



Now, this graph I have already shown you earlier, while talking about evolution of computer systems, but now let us see the design issue of memory system. This red line shows the growth of processor in course of time and this green line shows the growth of memory technologies. Although you can see both are growing, but processor design is growing at a much higher pace, and memory design advancement is coming at a lower rate. But both are advancing. Technology is advancing in both, but this speed gap is steadily increasing. So, the most important issue is to bridge this processor-memory gap that has been widening with every passing year. Advancements in memory technologies are unable to cope with faster advancement in processor technology, but there are many techniques that are used to bridge this speed gap. At this point some important questions arise. How to make a memory system work faster? It has a limitation.

(Refer Slide Time: 17:32)

The slide has a yellow background with a black header and footer. The footer features logos for IIT Kharagpur, NPTEL Online Certification Courses, and a circular emblem. A video feed of a woman speaking is visible in the bottom right corner.

- Some important questions?
 - How to make the memory system work faster?
 - How to increase the data transfer rate between CPU and memory?
 - How to address the ever increasing storage needs of applications?
- Some possible solutions:
 - *Cache Memory*: to increase the effective speed of the memory system.
 - *Virtual Memory*: to increase the effective size of the memory system.

But how we can make it faster such that the processor and memory speed gap can be reduced, how to increase the data transfer rate between CPU and memory, the transfer of data; how it can be made faster, and how to address the ever increasing storage need of application? We need large memory as well. Not only we need faster, we also need larger memories. Because, there are various applications that require larger memory space.

So, we need to look into all the issues. First issue is how we can make this memory work faster. How we can have a larger memory and by all these things how we can reduce the speed gap between processor and memory. Some possible solutions are cache memory and virtual memory. What a cache memory does, we will be looking into detail in later weeks. But what it does is, it increases the effective speed of memory system. And what virtual memory does? It increases the effective size of memory system. So, we will be looking into these in some detail in the later part of this week.

(Refer Slide Time: 19:19)

What is Cache Memory?

- A fast memory (possibly organized in several levels) that sits between processor and main memory.
- Faster than main memory and relatively small.
- Frequently accessed data and instructions are stored here.
- Cache memory makes use of the fast SRAM technology.

The diagram illustrates the memory hierarchy. It shows four rectangular boxes arranged horizontally from left to right. The first box is labeled 'CPU'. The second box is labeled 'Level-1 Cache'. The third box is labeled 'Level-2 Cache'. The fourth box is labeled 'Main Memory'. Each box has a thin red border around it.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

A small video player window is visible in the bottom right corner, showing a woman with dark hair wearing a blue top, likely the instructor or lecturer.

So, very briefly what is cache memory? It is a fast memory that sits between CPU and main memory. And we can have many levels of cache memory. Why cache memory is in place? We will see this because of properties of computer programs called locality of reference. One is temporal locality of reference; other is spatial locality of reference.

So, we will see this in detail later. But for now let us understand that cache memory is a memory, which sits between CPU and main memory, and there can be many level of caches. But the cache memory cannot be very large. It is much smaller compared to main memory. So, frequently accessed data or instruction can only be brought here and executed. And what technology is used to build this cache memory? We use static RAM technology to build this cache memory.

(Refer Slide Time: 20:32)

What is Virtual Memory?

- Technique used by the operating system to provide an illusion of very large memory to the processor.
- Program and data are actually stored on secondary memory that is much larger.
- Transfer parts of program and data from secondary memory to main memory only when needed.

The diagram illustrates the concept of Virtual Memory. It shows three components: CPU (Central Processing Unit), Main Memory (represented by a rectangle), and Secondary Memory (represented by a cylinder). Arrows indicate the flow of data between the CPU and Main Memory, and between Main Memory and Secondary Memory.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Virtual memory is basically a concept that is used to give an illusion, that we have a very large memory space at our disposal. But actually we have space equal to main memory. But it gives an illusion to the programmer that you have a larger space to execute. So, the technique used by operating system to provide an illusion of a very large memory to the processor. Program and data are actually stored on secondary memory that is much larger. And data and instruction are brought into main memory as and when it is needed.

So, secondary memory is a concept where we say that we have a very large memory, but whenever we want to execute it we need to bring those data or instruction into main memory, and then it can be executed.

(Refer Slide Time: 21:44)

How a Memory Chip Looks Like?

- Memory cells are organized in the form of an array.
- Every memory cell holds one bit of data.
- Present-day VLSI technology allows one to pack billions of bits per chip.
- A memory module used in computers typically contains several such chips.

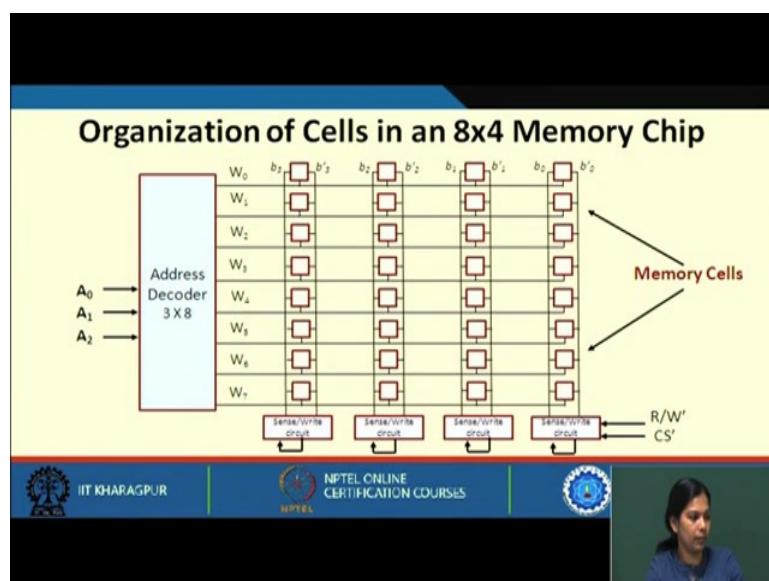


IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL



Now, let us see how a memory chip looks like. So, this is on a PCB (printed circuit board). So, these are memory. This is a separate memory modules that are placed. These memory cells are organized in the form of array. This may be a 4 GB memory and each having, say 1 GB, 1 GB, 1 GB like that. So, present-day VLSI technology allows one to pack billions of bits per chip. Memory modules used in computers typically contain several such chips. These chips are put into the memory slots that are present in the PC.

(Refer Slide Time: 22:47)



Now, let us see organization of cells in a 8×4 memory chip. So, this is a 8×4 memory chip. Let us see how it is organized. So, you have 8 rows; this is the first row, second row, third row, fourth row, and is the 8th row. And in each row, there are 4 bits that can be taken out. Let us consider this as a row. This row is connected with the word line. This is called word line: W_0 , W_1 , W_2 ; these are word line. And individual cell is connected to 2 bit lines. One is b another is $b\text{-bar}$ that is, complement of the other. And which is connected to the sensor write circuitry, and this sensor write circuitry is further connected to the data lines.

Here there are 8 rows. We have to select any one of the 8 rows. For that reason we require a 3×8 address decoder. So, these A_0 , A_1 , and A_2 are applied to this address and then depending on this, say it is 0 0 0, the first word line will get selected. And then all the bits of the word line is transferred through this sensor write circuit to the data lines, if you want to read the data.

And suppose I want to write the data into the cells, then what will happen? The data present in these data lines that is coming from your MDR will get stored in these bits through this sensor write circuit. So, in this organization we can see that a 8×4 memory chip is there. So, the address is decoded with using a 3×8 decoder. And then each of the bit, each of the memory cells are connected to 2 bit lines. One is the complement of the other, which is connected to the sensor write circuitry and through the sensor write circuitry, is connected to the data lines. And we have to also supply these signals that are: either you want to read a data or want to write a data.

So, this is how the memory chip is organized. So, as I said a 32-bit memory chip is organized as 8×4 as shown in the previous figure.

(Refer Slide Time: 26:01)

- A 32-bit memory chip organized as 8×4 is shown.
- Every row of the cell array constitutes a memory word.
- A 3×8 decoder is required to access any one of the 8 rows.
- The rows of the cells are connected to the word lines.
- Individual cells are connected to two bit lines.
 - Bit b and its complement b' .
 - Required for reading and writing.
- Cells in each column are connected to a sense/write circuit by the two bit lines.
- Other than address and data lines, there are two control lines: R/W' and CS' (Chip Select).
 - CS is required to select one single chip in a multi-chip memory system.

Each row of the cell array constitutes a memory word. So, the entire row entire one word is the row. So, every row of the cell will constitute a memory word we need a 3×8 decoder to access any one of the 8 rows. And the rows of the cells are connected to the word lines. Individual cells are connected to 2 bit lines. One is b another is its complement, and it is required for reading and writing.

And cells in each column are connected to sensor write circuitry. So, this is one column, this is the next column, third column and fourth column. The cells in each column are connected to this sensor right circuitry. Other than the address and data lines there are 2 control lines, read/write and chip select. And why chip select is required? It is required to select one chip in a multi-chip memory system. We will be seeing this with examples later.

So, basically this read/write and chip select is connected, such that either it will specify that you have to read the content of any one of the words, or you have to write data into one of the words. Now in this diagram, how many external connections are required? What do you mean by external connection? Externally that is provided not within this memory chip. You can clearly make out that these address lines are A_0 , A_1 and A_2 are externally provided to this decoder, and it is decoded and a particular word is selected.

Now once you select a particular row, then all these bits will be transferred to the data line, through the sensor write circuit. So, it is connected to 4 bits of this. So, there will be

4 data lines through which this data will go. So, those are 4 more external signals that are required here. So, 3 for this address 4 for the data lines. And then you have 2 more signals, that is read/write and chip select, that should be also provided externally. Because the processor will tell either you have to read the data or you have to write a data. And there will be two more, that is, power supply and ground.

(Refer Slide Time: 29:06)

External Connection Requirements

- The 8 x 4 memory requires the following external connections:
 - Address decoder of size: 3 x 8
 - 3 external connections for address.
 - Data output : 4-bit
 - 4 external connections for data.
 - 2 external connections for R/W' and CS'.
 - 2 external connections for power supply and ground.
 - Total of $3 + 4 + 2 + 2 = 11$.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, external connection requirements that are there for this 8 x 4 memory, is 3 external connections for address, 4 external connections for data, 2 external connections for read/write and chip select, 2 external connections for power supply and ground. So, a total of $3 + 4 + 2 + 2 = 11$ is required for this 8 x 4 memory chip.

(Refer Slide Time: 29:44)

What About a 256 X 16 Memory?

- Here the total number of external connections are estimated as follows.
 - Address decoder size: 8×256
 - 8 external connections for address.
 - Data output : 16-bit
 - 16 external connections for data.
 - 2 external connections for R/W' and CS'.
 - 2 external connections for power supply and ground.
 - Total of $8 + 16 + 2 + 2 = 28$.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL



Now, let us see what about this 256 x 16 memory. There will be 256 rows. To select any one of the 256 rows, you require a 8×256 decoder. So, the address decoder size will be 8×256 . So, 8 external connections for address will be required. Then the data output is 16. So, 16 external connections will be required to transfer the data, either to read the data or to write the data. Similarly, 2 external connections for read/write and chip select, and 2 for power supply and ground. So, a total of 28 external connections will be required.

So, we come to the end of this lecture where we briefly discussed about what is memory? How memory chips can be organized? And in the next few lectures, we will be seeing what kind of memory technologies are actually used to build this.

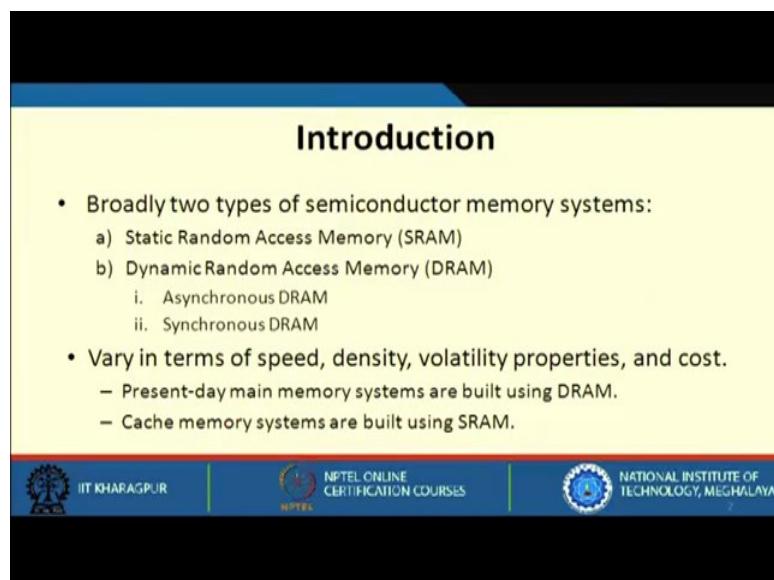
Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 24
Static And Dynamic Ram

Welcome to lecture 24. In this lecture we will be looking into static and dynamic RAM. So, broadly two types of semiconductor memory systems will be seen. And we will be seeing how a single-bit SRAM or DRAM cell is built. How a single bit is built and then how you can extend it to any size?

(Refer Slide Time: 00:55)



Introduction

- Broadly two types of semiconductor memory systems:
 - a) Static Random Access Memory (SRAM)
 - b) Dynamic Random Access Memory (DRAM)
 - i. Asynchronous DRAM
 - ii. Synchronous DRAM
- Vary in terms of speed, density, volatility properties, and cost.
 - Present-day main memory systems are built using DRAM.
 - Cache memory systems are built using SRAM.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, as I said broadly two types of semiconductor memory systems exist, static random access memory and dynamic random access memory. Under dynamic random access memory, we have two more types; one is called asynchronous DRAM, another is called synchronous DRAM.

Now, how these types of semiconductor memory vary? How you can differentiate among them? They vary in terms of speed; that means, how fast it is? How fast is your SRAM? Or how fast is your DRAM? In terms of density; that means, within the same area, how many bits you can pack using SRAM? Or how much bits you can pack using DRAM? That is meant by density. Then comes volatility.

So, if you have power supply given, then static RAM will have its data. But in case of dynamic RAM, even if you have supplied the power the data may not retain. You have to do periodic refresh to keep the data. So, these memories vary in terms of speed, access time, density, volatility, and cost. We will see that all these things are very much related to each other.

We will see the pros and cons of these properties. In present day our main memory system are built using DRAM, and cache memory systems are built using SRAM. And we will see that our cache memory system is relatively faster, but it is small. And our DRAM is relatively slower,

(Refer Slide Time: 03:53)

Static Random Access Memory (SRAM)

- SRAM consists of circuits which can store the data as long as power is applied.
- It is a type of semiconductor memory that uses bistable latching circuitry (flip-flop) to store each bit.
- SRAM memory arrays can be arranged in rows and columns of memory cells.
 - Called *word line* and *bit line*.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

but it can store a large amount of data. Coming to SRAM, it consists of circuits that can store the data as long as power is applied. In this type a semiconductor memory, a bistable latch circuit or flip flop, is used to store each bit of data.

To store individual bits of data bistable latching circuitry is used. And SRAM memory arrays can be arranged in rows and columns. We have already seen that how a memory system is organized. A memory chip is organized in terms of rows and columns.

So, SRAM memory arrays can be arranged in rows and columns. And these are called word line and bit line.

(Refer Slide Time: 05:27)

- SRAM technology:
 - Can be built using 4 or 6 MOS transistors.
 - Modern SRAM chips in the market uses 6-transistor implementations for CMOS compatibility.
 - Widely used in small-scale systems like microcontrollers and embedded systems.
 - Also used to implement cache memories in computer systems.
 - To be discussed later.

Now, SRAM memory cell can be built using 4 or 6 MOS transistors. But modern SRAM chips in the market it uses 6-transistor implementation or CMOS compatibility. And this kind of SRAM chip which uses 6-transistor are widely used in small scale systems like microcontrollers and embedded system. We know that today's microcontrollers and embedded systems are everywhere.

It does not require very large memory, but it requires faster memory. So, SRAM chips are used in these microcontrollers and embedded systems. And are also used to implement cache memory in the computer system which will be discussed later.

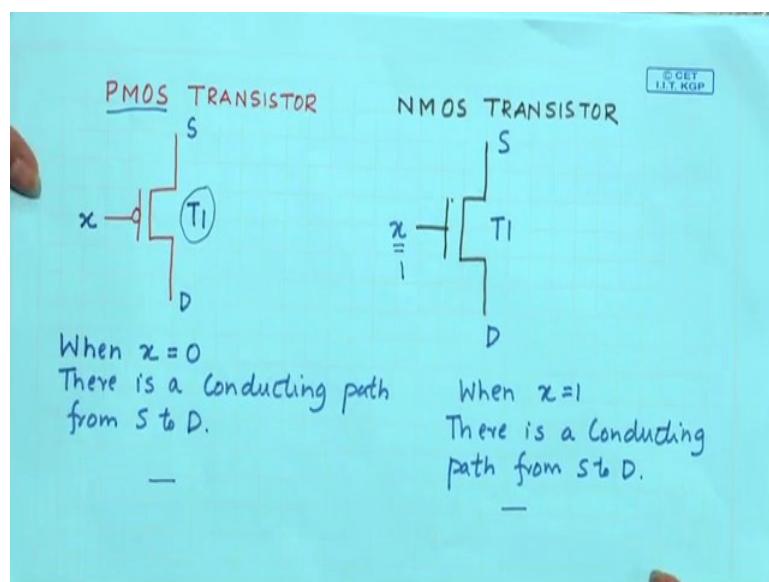
(Refer Slide Time: 06:33)

A 1-bit SRAM Cell

- Two inverters are cross connected to form a latch.
- The latch is connected to two bit lines with transistors T_1 and T_2 .
- Transistors behave like switches that can be opened (OFF) or closed (ON) under the control of the word line.
- To retain the state of the latch, the word line can be grounded which makes the transistors off.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

(Refer Slide Time: 06:39)



Now, before going to this slide I will just discuss two things, one is PMOS transistor, another is NMOS transistor. In this PMOS transistor you can see that, there is an input that is x , and this is source or drain. Basically, this transistor acts as a switch. And this is controlled by this input. If this x input is 0 for PMOS transistor there is a conducting path from S to D.

So, T_1 is on. If x is 0 then there is a conducting path from S to D and this will make T_1 on. This is the feature of PMOS transistor. Now see the NMOS transistor. In NMOS

transistor in the same way, if this value is 1 then only there is a conducting path from S to D. So, what is the difference between these two? In PMOS if the input is 0, then only it will be conducting. And in NMOS if this input is 1 then only this transistor will be on. That is, there will be a conducting path from S to D.

Now, come to a 1-bit SRAM cell, how it looks like. This is a single bit SRAM cell. Here we have two inverters; this is a symbol of inverter. So, two inverters are cross connected to form a latch. So, if you give a 0 input here this will be 1. If you give a 1 input here this will be 0. And now this latch is now connected to two transistors T1 and T2. Now what is T1 and T2? You see this is a NMOS transistor.

So, if you want to activate this transistor, what input you have to give? You have to give a 1 input here. Then only this transistor will be on. So, this particular latch is connected to the 2 bit lines. If you recall our discussion, each of the cell is connected to 2 bit lines, b and b-bar. In the same way, this is a single cell which is connected to 2 bit lines b and b-bar, through transistors T1 and T2.

And these transistors behave like switches.

To retain the state of the latch, the word line can be grounded that makes the transistor off. Now if we want to retain the value which is there in the latch, in that case what we have to do? We have to give this word line to ground. If it is ground this will be 0. Then there is no conducting path. Whatever value is in the latch, it will remain there. So, we have bit lines, two inverters are cross connected to form a latch, and these are connected to 2 transistors through the bit lines through this transistor, and this transistor can be made on and off. Depending on that, we can read the value or we can write the value into the cell. Let us see that.

(Refer Slide Time: 11:59)

READ Operation in SRAM

- To read the content of the cell, the word line is activated (= 1) to make the transistors T_1 and T_2 on.
- The value stored in latch is available on bit line b and its complement on b' .
- Sense/write circuits connected to the bit lines monitor the states of b and b' .

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Read operation. How we will read the value? That means, if the value here is let us say 1, then 1 will be at this A, and 0 will be at this B,, because the bit line b will have 1 and b-bar that is the complement will have 0.

To read the content of the cell, what we need to do? The word line is activated. So, now, I want to read this content whatever is in A and B. So, we need to activate this word line. By activating means we are supplying 1 here. So, if we make it on if we activate this word line, then what will happen? This transistor T_1 and this transistor T_2 will be on. If this transistor T_1 and T_2 is on, then the value which is stored in A and B that is in the latch will be available on bit line b, and will be available on bit line b-bar.

So, if the value is 1, then 1 will be available in the bit line b, and 0 will be available in bit line bbar. In the same way if the value is 0, then 0 will be available in bit line b and 1 will be available in bit line bbar. And then a sense or write circuit connected to the bit lines will monitor the state of b and bbar. And accordingly it will figure out whether it is 1 or 0. So, this is how we perform read operation here.

(Refer Slide Time: 14:18)

WRITE Operation in SRAM

- **To write 1:** The bit line b is set with 1 and bit line b' is set with 0. Then the word line is activated and the data is written to the latch.
- **To write 0:** The bit line b is set with 0 and bit line b' is set with 1. Then the word line is activated and the data is written to the latch.
- The required signals (either 1 or 0) are generated by the sense/write circuit.

The diagram illustrates the SRAM write operation. It shows a latch consisting of two inverters connected in series. The inputs are labeled A and B. Transistor T_1 is connected between the ground rail and the input A, with its gate controlled by the bit line b . Transistor T_2 is connected between the output of the first inverter and the input B, with its gate controlled by the bit line b' . The word line is shown activating the transistors T_1 and T_2 . The outputs of the inverters are labeled Bit lines, and the word line is labeled Word line.

Now, moving on let us see the write operation in SRAM. Now for writing we can either write 1 or we can either write 0. First see if I want to write 1, then what I need to do? I need to set 1 in bit line b . In this b I have to set 1, and I have to set 0 in bit line b' . So, bit line b will have 1, bit line b' will have 0. Then I will activate the word line. Which will make the transistor T_1 and T_2 on whatever value is in the bit line will be available in A. And whatever value is in b' will be available in B. So, the data is written to the latch.

Similarly, if I have to write 0, then I apply 0 in the bit line b , and 1 in bit line b' . And in the same way I activate the word line. By activating the word line the transistor will be on and whatever data will be in b will be stored in this latch, and whatever will be in the b' will come here. So, this latch will now have the value which is there in these bit lines will be available in this latch. And now as I said, I can either write 0 or I can write 1; the required signals that is either 0 or 1 will be generated by the sense or write circuit. So, this is how write operation happens in SRAM.

(Refer Slide Time: 16:08)

6-Transistor Static Memory cell

- 1-bit SRAM cell with 6-transistors are used in modern-day SRAM implementations.
- Transistors ($T_3 \& T_5$) and ($T_4 \& T_6$) form the CMOS inverters in the latch.
- The data can be read or written in the same way as explained.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | © IIT KGP

Now, see before moving here, if you consider this diagram you have a NOT gate. So, let us see the CMOS realization of NOT gate. Here is the CMOS realization of NOT gate.

(Refer Slide Time: 16:32)

CMOS REALIZATION OF NOT GATE

WHEN $X=0$ T_1 is ON (closed)
 T_2 is OFF (open)
HENCE $Y \approx 5V = 1$

WHEN $X=1$ T_1 is OFF
 T_2 is ON
HENCE $Y \approx 0V = 0$

© IIT KGP

You can see here that this is the circuit, or PMOS transistor. A PMOS transistor is connected here. This is T1 and an NMOS transistor is connected here which is T2. Now, this is a NOT gate. Let us see how this will act as a NOT gate; it means if I give x input as 1, the output should be 0. If I give x as 1, then the transistor T1 will not be conducting. Because, this is a PMOS transistor and it will only conduct when the input here is 0.

So, there will be no connection from this is not connected. This is open. T1 is off. Now if this is 1, the bottom transistor T2 which is a NMOS transistor will be on. And there is a path from this Y to ground. So, if there is a path from this Y to ground then this Y will have the value approximately equals to 0. So, when this x equals to 1, T1 is off, this transistor is off, and T2 is on. Hence Y will be have approximately a value of 0 volt, which is equivalent to 0.

Now, let us take x as 0. If x is 0, I must get the output as 1. If this is 0, then the above transistor that is T1 will be conducting, but the below transistor that is T2 will not be conducting. If this is conducting there is a path from this 5 volt to Y. So, this Y will have roughly equivalent to 5 volt, which is equivalent to 1. So, if we give input 0, the transistor T1 will be conducting and we will have an output 1. And if x is 1 then, the below transistor will be conducting and we will have the output Y.

Now, what we will do? Once I have shown you this CMOS realization of NOT gate. Now moving on, we will see 6-transistors static RAM cell. This is the CMOS realization of NOT gate. So, I have just replaced this NOT gate with the CMOS realization. And now what we are getting? We are naming these various transistors. So, this is transistor T1 and T2 initially it was there. Now this transistor is T3 and T5. And this transistor is T4 and T6.

Now, one bit SRAM cell with 6-transistors are used in modern day SRAM implementation. So, you can see T3 and T5 and T4 and T6 forms the CMOS inverters that I have just now explained. And reading a data, the data that is to be read or written can be done in the same way as explained in the previous example.

(Refer Slide Time: 21:00)

In State 0

- In state 0 the voltage at **X** is low and the voltage at **Y** is high.
- When the voltage at **X** is low, transistors (**T4 & T5**) are on while (**T3 & T6**) are off.
- When word line is activated, **T1** and **T2** are turned on and the bit lines **b** will have **0** and **b'** will have **1**.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

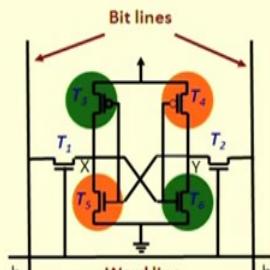
Now, let us see in state 0 what happens. In state 0; that means, this X is having 0 and this Y is having 1. In state 0 the voltage at X is low. So, here it will be low and the voltage at Y is high, X is low. So, X input is going to this T6 and it is going to this T4.

So, if it is going to this T6 and this is 0, then this will be off. But this T4 will be on. And now Y input is having 1. So, if this is 1 then this T5 will be on, but this T3 will be off. So, that is what T4 and T5 will be on, while T3 and T6 will be off. Now when the word line is activated T1 and T2 are turned on. And the bit line b will have a 0 value.

(Refer Slide Time: 22:53)

In State 1

- In state 1 the voltage at X is high and the voltage at Y is low.
- When the voltage at X is high, transistors (T_3 & T_6) are on while (T_4 & T_5) are off.
- When word line is activated, T_1 and T_2 are turned on and the bit lines b will have 1 and b' will have 0.



The diagram shows a SRAM cell structure. It consists of six transistors labeled T1 through T6. Transistor T1 is between the Word line and node X. Transistor T2 is between the Word line and node Y. Transistor T3 is between node X and Bit line b. Transistor T4 is between node Y and Bit line b'. Transistor T5 is between node X and ground. Transistor T6 is between node Y and VDD. The Bit lines b and b' are shown with arrows indicating they are high (1) and low (0) respectively. The Word line is also active, turning on T1 and T2.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL



So, this is how in state 1 it happens. Similarly state 0 it happens like this. Now let us move on what happens in state 1. State 1 means X will now be high and Y will be zero; that means, the bit line b should have 1 and b' will be 0. So, as X is at 1. So, X is going here and here. So, this will be off, but this will be conducting. And similarly Y is 0. So, Y is low or 0. So, this will make this as non-conducting, but this will be conducting. So, T_3 and T_6 will be conducting, and T_4 and T_5 will be off. Now when the word line is activated in the similar fashion T_1 and T_2 will be on will be turned on. And the bit lines b will have 1 and bit line b' will have 0.

So, this is how what happens in state 1. Let us see some features of SRAM. So, here the current flows in the cells only when the cell is accessed. This is a CMOS cell property.

(Refer Slide Time: 24:00)

Features of SRAM

- Moderate / High power consumption.
 - Current flows in the cells only when the cell is accessed.
 - Because of latch operation, power consumption is higher than DRAM.
- Simplicity – refresh circuitry is not needed.
 - Volatile :: continuous power supply is required.
- Fast operation.
 - Access time is very fast; fast memories (cache) are built using SRAM.
- High cost.
 - 6 transistors per cell.
- Limited capacity.
 - Not economical to manufacture high-capacity SRAM chips.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, current flows in the cells only when the cell is accessed. Because of this latch operation power consumption is little higher. What is the simplicity? No refresh circuitry is required. It is of course volatile but as long as the power is supplied to it, you need not have to do any kind of refresh. It is much faster. Access time is very fast. So, the fast memories like cache are built using these kinds of cells. But the cost is high. Why? We see that here we require 6 transistors to build 1-bit memory; also the space it takes is more.

So, the cost is high. And of course, it has got limited capacity. Because we cannot build a very large a SRAM cell, as it requires 6-transistors per cell.

(Refer Slide Time: 25:27)

Dynamic Random Access Memory (DRAM)

- Dynamic RAM do not retain its state even if power supply is on.
 - Data stored in the form of charge stored on a capacitor.
- Requires periodic refresh.
 - The charge stored cannot be retained over long time (due to leakage).
- Less expensive than SRAM.
 - Requires less hardware (one transistor and one capacitor per cell).
- Address lines are multiplexed.

Bit line

Word line

Sense/Write Circuit

1-transistor DRAM Cell

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Next coming to dynamic random access memory: in dynamic random access memories as we know that it does not retain the state even if power is supplied to it. So, here the data are stored in the form of charge on the capacitor. And this charge cannot be stored for longer period of time. And this happens due to some leakage property of the capacitor as well as this transistor.

But you see; how simple is the cell. You have one transistor that is connected to the bit line and this transistor is also connected to the word line. Because through word line it will get activated and the transistor is also connected to this capacitor which is grounded. And then it is connected to the sense or write circuit. This is a 1-transistor DRAM cell. But it requires periodic refresh because we are storing the data as charge in the capacitor. And this charge can be retained not for longer period of time.

These are less expensive, we can see that because only one transistor is required to build it; one transistor and one capacitor is required to build this. And here the address lines are multiplexed we will be seeing this little later.

(Refer Slide Time: 27:04)

READ Operation in DRAM

- The transistor of the particular cell is turned on by activating the word line.
- A sense amplifier connected to bit line senses the charge stored in the capacitor.
- If the charge is above threshold, the bit line is maintained at high voltage, which represents logic 1.
- If the charge is below threshold, the bit line is grounded, which represent logic 0.

The diagram illustrates the DRAM cell structure during a read operation. A capacitor labeled 'C' is connected to one terminal of a transistor labeled 'T'. The other terminal of the transistor is connected to a 'Bit line'. A 'Word line' is shown above the cell, with a connection point to the gate of the transistor 'T'. Below the cell, a 'Sense/Write Circuit' is shown with a feedback loop connecting back to the Bit line. The Bit line is also shown with a downward arrow indicating it can be grounded or maintained at a high voltage level.

Now, let us see how we can read the value here. So, as we said that the data is stored as a charge in this capacitor, and that represent whether a 1 is stored or 0 is stored. So, let us see for reading the data from this cell the transistor of this particular cell is turned on by activating the word line. So, this is the word line we activate the word line, such that this particular transistor T is on. Now we have a sense amplifier connected to the bit line. And this line it senses the charge stored in the capacitor. Once this is on there is a connection between this bit lines to this through this transistor.

If the charge is above certain threshold, then we say that the bit is maintained at high voltage, that is 1. And it will represent logic 1. If the charge is below certain threshold, then we say that the then the bit line is grounded, which represent logic 0. Now we see that if we read a cell, automatically it is getting refreshed. Because, we are keeping the required data that is to be stored in this capacitor. So, if it is 1 then this is made on and we sense the charge in the capacitor that automatically refreshes. In the same way if it is 0, then also automatically refreshes.

(Refer Slide Time: 29:06)

WRITE Operation in DRAM

- The transistor of the particular cell is turned on by activating the word line.
- Depending on the value to be written (0 or 1), an appropriate voltage is applied to the bit line.
- The capacitor gets charged to the required voltage state.
- Refreshing of the capacitor requires periodic READ-WRITE cycles (every few msec).

The diagram illustrates the write operation in DRAM. A vertical Bit line is connected to the gate of a MOSFET transistor labeled 'T'. The other end of the Bit line is connected to a capacitor labeled 'C' which is also connected to ground. A horizontal Word line is connected to the drain of the transistor 'T'. Below the circuit, a box labeled 'Sense/Write Circuit' contains a small arrow pointing upwards, indicating the direction of signal flow from the sense/write circuit to the Bit line.

In the write operation what happens? Through this sense or write circuit this bit line will have the available data, either 0 or 1. The transistor of this particular cell is turned on by activating the word line and depending on the value that is to be written, either you have to write 0 or 1, an appropriate voltage is applied to this bit line.

And, as an appropriate voltage is applied to this bit line, this capacitor gets charged to the required voltage. If it is 0 a required voltage it is charged to that required voltage if it is 1, it is charged to that particular required voltage. And refreshing of the capacitor requires periodic read/write cycles every few milliseconds; so in every few milliseconds if even if you are not reading you have to do refresh to store the data to keep the data.

(Refer Slide Time: 30:07)

The slide has a blue header bar with the title 'Types of DRAM'. Below it is a yellow section containing two boxes: 'a) Asynchronous DRAM (ADRAM)' and 'b) Synchronous DRAM (SDRAM)'. The ADRAM box lists three points: timing handled asynchronously, a special memory controller circuit generates signals asynchronously, and DRAM chips produced between the early 1970s to mid-1990s used asynchronous DRAM. The SDRAM box lists five points: memory operations synchronized by a clock, concept came in the 1970s, commercially made available in 1993 by Samsung, replaced almost all types of DRAMs by 2000, and performance is much higher compared to other existing DRAMs. At the bottom, there are logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Types of DRAM	
a) Asynchronous DRAM (ADRAM)	b) Synchronous DRAM (SDRAM)
<ul style="list-style-type: none">- Timing of the memory device is handled asynchronously.- A special memory controller circuit generates the signals asynchronously.- DRAM chips produced between the early 1970s to mid-1990s used asynchronous DRAM.	<ul style="list-style-type: none">- Memory operations are synchronized by a clock.- Concept of SDRAM came in the 1970s.- Commercially made available only in 1993 by Samsung.- By 2000 SDRAM replaced almost all types of DRAMs in the market.- Performance of SDRAM is much higher compared to all other existing DRAM.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, there are various kinds of dynamic RAM. One is asynchronous DRAM another is synchronous DRAM. As the name suggest, in asynchronous DRAM the timing of the memory devices handled asynchronously. What do you mean by that? Here there is no fixed timing; I mean when you give a request for read and when the data will be available. The processor has to take care of when the data is available.

But in case of synchronous DRAM it is not like that; there is timing involved to it. And after a particular time data will be available. In asynchronous DRAM a special memory controller circuit generates the signal asynchronously. The DRAM chips that are produced between early 1970s to mid 1990s all used asynchronous DRAM, but today's computers all use synchronous DRAM. So, here the memory operations are synchronized by a clock.

A clock is there which synchronizes it. And this concept was already available in the 70s, but commercially it was available in 1993, and by 2000 SDRAM replaced almost all types of DRAMs in the market. So, there is no asynchronous DRAM these days. We have all synchronous DRAM. And the performance of SDRAM is much higher compared to all other existing DRAMs. So, we have seen in this lecture what all semiconductor technologies are used to build SRAM and DRAM. What the various kinds of DRAMs? And now we will be seeing specifically the various kinds of DRAMs.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 25
Asynchronous DRAM

Welcome to lecture 25, where we will be discussing in detail about Asynchronous DRAM.

(Refer Slide Time: 00:33)

Asynchronous DRAM

- The timing of the memory device is controlled asynchronously.
- The device connected to this memory is responsible for the delay.
- Address lines are divided into two parts and multiplexed.
 - Upper half of address:
 - Loaded into *Row Address Latch* using *Row Address Strobe* (RAS).
 - Lower half of address:
 - Loaded into *Column Address Latch* using *Column Address Strobe* (CAS).

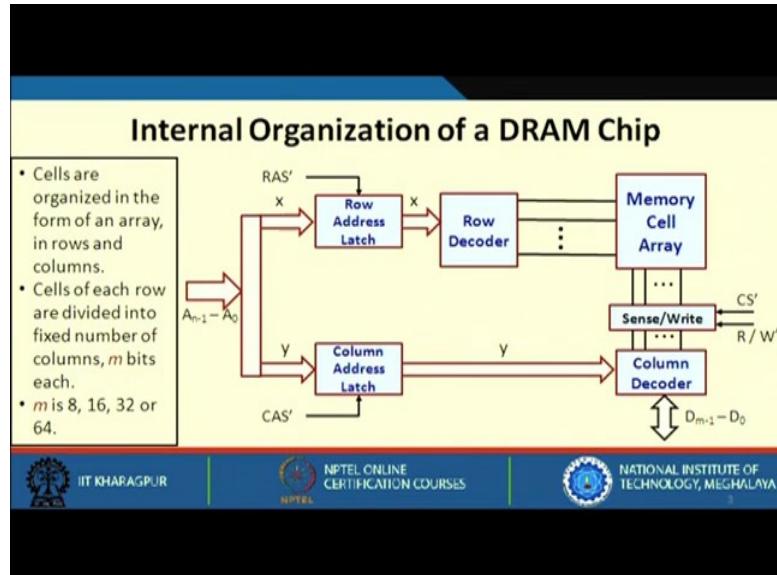
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Here, the timing of the memory device is controlled asynchronously. So, the device connected to this memory is responsible for the delay. If you say that the processor is connected to the memory, then it is the responsibility of the processor to take care of that timing, when the data is available, etc. Here the address lines are divided into two parts and multiplexed. The upper half of the address is loaded into a latch called row address latch using a signal called row address strobe RAS.

And the lower half of the address is loaded into column address latch using column address strobe CAS. So, what do you mean by that it is multiplexed? That means, from outside we are providing a set of address, and that particular address is first hitting a particular row using this row address latch, and it is stored there by the signal RAS, and then the column address is provided. So, in the same address line the column address is provided

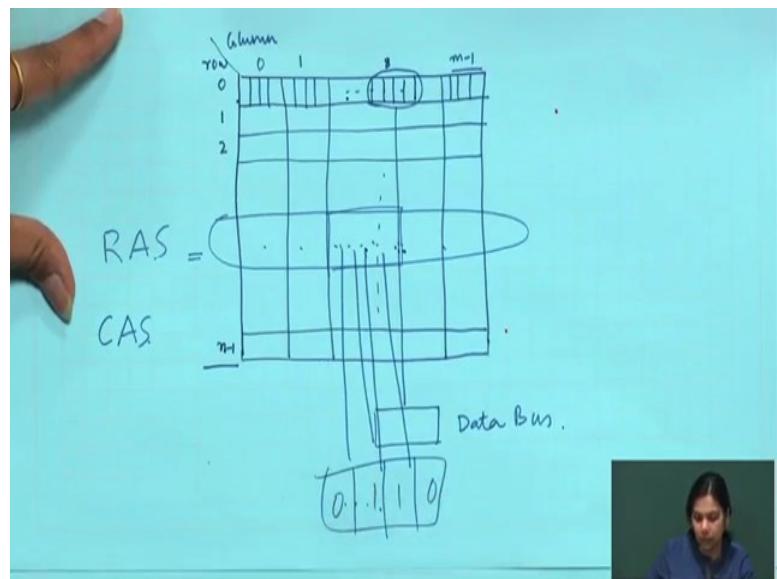
and that is latched using CAS in column address latch. So, these are the two things how it is divided. We will be seeing this with examples later.

(Refer Slide Time: 02:24)



Let us now come to the internal organization of a DRAM chip. As I said the cells are organized in the form of an array in rows and columns. Cells of each row are divided into fixed number of columns of m bits each. Let me explain this.

(Refer Slide Time: 03:04)



So, this is my memory. These are my rows. This is the first row, this is second row, and so on. So, the whole memory is divided into rows and columns. So, I have n rows and m

columns. And in each of these m columns, I can have some bits. So, what I will do basically I will hit a particular row. And then there are m columns. I will hit a particular column.

Let us say this column. This is say 8th column I hit this and then, this particular column's data will be transferred to this sense or write circuit to the data bus. So, basically it is organized like this. It is divided into rows and columns, and each column is having some bits, let us say m . m is 8, 16, 32 or 64. Now let us see what we are doing here; we are applying the entire address 0 to $n-1$ --- let us say total address is n bits. So, you have your addresses 0 to $n-1$ that is divided into two parts, X and Y .

X address will go and hit to row address latch in response of the signal RAS. And this address is now fed to the row decoder, because there are number of rows. We need to select one particular row from set of these rows. The row decoder will be used to select one of the rows. Now this Y address will be provided to this column address latch in response of the signal CAS. And this Y will hit to the column decoder, because we also have many number of columns. And we do not know that in that particular row from which column I have to get the data; let say in this particular row I have to get the data from 10th column.

So, accordingly I have to pass a value here. So, a column decoder is in place that will decode a particular column that is available in this particular row. And so the sense or write circuit it will be read the data from that particular column, and it is transferred to the data bus lines. Or it can be written; in that case whatever data is in the data bus will be written into that particular column of that particular row. So, this is what is the internal organization of a DRAM chip.

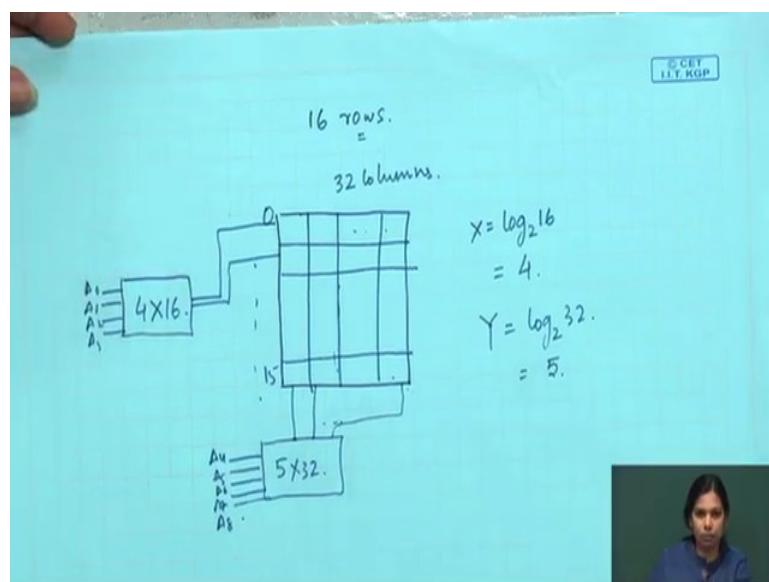
(Refer Slide Time: 08:00)

- Suppose that the memory cell array is organized as $r \times c$.
 - r rows and c columns.
- An x -bit address is required to select a row r , where $x = \log_2 r$.
- An y -bit address is required to select a column c , where $y = \log_2 c$.
- Total address bits: $n = x$ (high order) + y (low order)

So, suppose that a memory cell array is organized as row and column. So, r rows and c columns are there. An X bit address is required to select a row. The number of bits required for X will be $\log_2 r$.

And now Y bit address is required to select a column, where Y will be $\log_2 c$. So, the total address bit will be n that is X (high order bit) and Y (low order bit). I will just take a small example and explain this.

(Refer Slide Time: 08:53)



Here you see you have 16 rows So, how much size decoder you will require? You require a 4×16 decoder that will be connected. And accordingly each of these rows will be selected at a time. So, depending on this address a particular row will get selected. So, we have a total of 0 to 15, that is 16 rows. So, now, if you have 16 rows how many bits will be required; $\log_2 16$ because, r is the number of rows. So, $X = 4$.

Similarly, let us say you have 32 columns. If you have 32 columns that is Y, then you require 5 bits for your column address that will be provided. And you require a 5×32 decoder that will be connected to these. The 5 inputs will be let us say A₄ A₅ A₆ A₇ and A₈. So, if you have 16 rows and 32 columns you require a 4×16 decoder for the rows, and you need a 5×32 decoder to select a particular column.

(Refer Slide Time: 11:34)

READ or WRITE Operation

- For a read operation, the x-bit row address is applied first.
 - It is loaded into *Row Address Latch* in response to the signal *RAS*'.
 - The read operation is performed in which all the cells of the selected row are read and refreshed.
- After loading of row address, the column address is selected.
- In response to *CAS*' the column address is loaded into *Column Address Latch*.
- Then the column decoder selects a particular column from *c* columns and an appropriate group of *m* sense/write circuits are selected.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Let us see how READ or WRITE operation will be performed. For a READ operation the X bit row address is applied first, because I have to go to a particular row first. And it is loaded into row address latch in response to the signal RAS. The READ operation is performed in which all the cells of the selected row are read and refreshed. After loading the row address the column address is selected. Now we have already selected a particular row. Now I have to know that within that row which column data to get. So, for doing, you have to apply the column address.

So, we apply the column address in the column address latch in response to CAS signal. Then the column decoder selects a particular column from *c* columns and an appropriate

group of m sense or write circuits are selected. I will just take the same example here. Let us say I have selected this particular row. So, once we select this particular row I put this address on RAS. And then I have to select any one of these columns. Then I apply CAS signal and let us say this particular column gets selected. And the m bits whatever bits there let us say 4 bit data will be output and will be available.

So, for reading what we need to do? We apply the column address first, we apply the row address first using RAS. Then we apply the column address using CAS. Then we select a particular column of that particular row and then finally, what we do we sense that set of bits from that particular column, and output it for a READ operation. And similarly for a WRITE operation whatever data is present in these 4 bits, let us say 0, 1, 1, 0; that set of data will be stored in this particular column of that particular row that has been provided.

(Refer Slide Time: 14:36)

- For a READ operation, the output values of the selected circuits are transferred to data lines D_{m-1} to D_0 .
- For a WRITE operation, the data available on the data lines D_{m-1} to D_0 is transferred to the selected circuits.
 - This information is stored in the selected cell.
- Both RAS' and CAS' are active low signals. That is they cause latching the addresses when they move from high to low.
- Each row of the cell array must be periodically refreshed to prevent data loss.
- Cost is low but access time is high compared to SRAM.
- Very high packing density (few billion cells per chip).
- Widely used in the main memory of modern computer systems.

IIT KHARAGPUR | **NPTEL ONLINE CERTIFICATION COURSES** | **NPTEL**

This is how READ and WRITE operations take place. So, for a READ operation the output values of the selected circuit are transferred to the data line.

I said for a WRITE operation the data available on the data lines are transferred to the selected circuits. So, for read it is transferred to the data lines, for the write from data lines it will be stored in that particular step. This information is stored in that selected cell. Now both RAS' and CAS' are active low signals; this dash symbolizes this is active low. So, it will be high when it will be activated when the input is 0. So, they cause latching the addresses when they move from high to low. So, this will not happen when we move

from low to high. Rather it will happen when we move from high to low. Each row of the cell array must be periodically refreshed to prevent data loss.

So, in this case each row of the cell array must be periodically refreshed. Cost is low because we have already seen that a single transistor and a single capacitor are required to build this. And of course, the cost is low, but the access time is high compared to SRAM, but billions of cells can be packed per chip.

And this particular kind of memory is used in main memory. We say that we have a main memory 4 GB. Can we extend it to 8 GB? Of course, you can extend it to 8 GB, provided you have that much available address bus.

Now I will take an example of 1 Gbit asynchronous DRAM chip.

(Refer Slide Time: 17:16)

An Example: 1 Gbit ADRAM Chip

- We assume that the 1 Gbit memory cells are organized as 32768 (2^{15}) rows and 32768 (2^{15}) columns.
- Let us assume that data bus is 32-bit long.
- So, the memory can be organized as $(2^{15}) \times (2^{10} \times 2^5)$.
 - Total number of address lines is 25 bits.
- High order 15 bits of the address is used to select a row.
 - Requires a 15×32768 row-address decoder.
- Low order 10 bits of the address is used to select a column.
 - Requires a 10×1024 column decoder.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

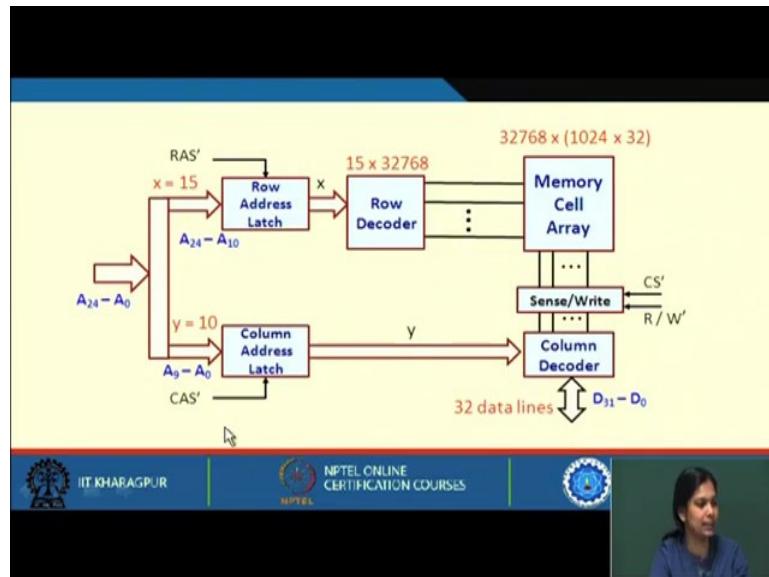
We assume that 1 GB memory cells are organized as 32768 that is, 2^{15} rows and 32768 ($2^{10} \times 2^5$) columns. So, let us assume that the data bus is 32 bit long.

So, the memory cell can be organized as $2^{15} \times 2^{10} \times 2^5$. Total number of address lines is 25, because 15 bits are required to select a particular row, and 10 bits to select a particular column.

So, I will have 2^{10} columns and each of these columns will have 2^5 . So, total coming down $2^{15} \times 2^{10}$ that is 1 gigabit. But total number of address line will be 15 for this row and, 10 bits for columns. So, 15 bits for selecting a row, and 10 bits for selecting a column, that is why 25 bits.

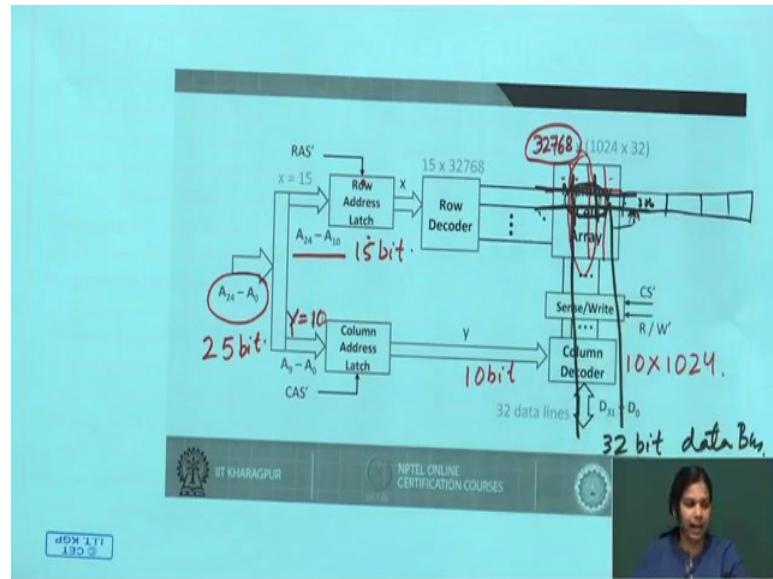
So, the high order 15 bits of the address will be used to select a row. So, we require a 15×32768 row address decoder. And the lower order 10 bits of the address will be used to select a column in that particular row. So, how many total columns are there? 2^{10} , so we require a 10×1024 column address. So, total is 25, 15 for row and 10 for column.

(Refer Slide Time: 20:26)



Now, let us see how it is organized. The total address bit we are having is how much?

(Refer Slide Time: 20:33)



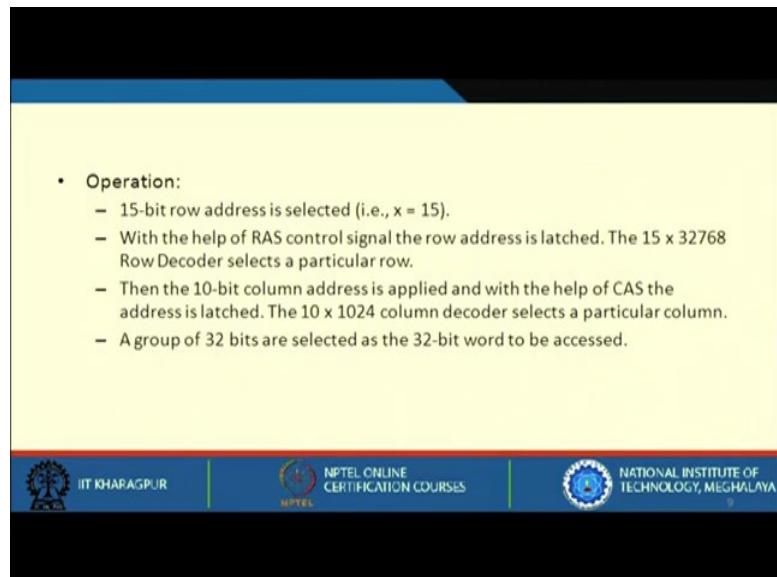
Total address bits are 25, 0 to 24. The high order 15 bits A10 to A24 will be fed to this row address latch in response to RAS signal. And then it is fed to the decoder that will decode that particular 15 bit address, and then it will map into one of the memory cells, one of the row of the total number of rows here is 32768 it will select any one row depending on this particular address. And then we apply the 10-bit column address; there is a column decoder. Depending on that, there are 1024 columns and any one of the columns will get selected and a 32-bit data will be transferred to the data bus.

So, let us see this from here, our total address is 25 bit. High order 15 bit is applied to row address latch in response to RAS signal. And then it is fed to the row decoder. So, these 15 bits fed to the row decoder will select any one of these 32768 rows. And any one of the row will get selected. Now I have 10-bit column address; so these 10 bit columns address A0 to A9. Low order 10-bit column address will be applied to this column address latch in response to the signal called CAS. This will be fed to the column decoder.

Now, this column decoder is a 10 x 1024 column decoder. It accepts 10 bits and it will select any one of the 1024 columns. Let us say it has selected this particular col column. I have selected this particular row. And now this column decoder has selected this particular column. Now, on this row this particular column got selected, and this will be transferred to your data bus.

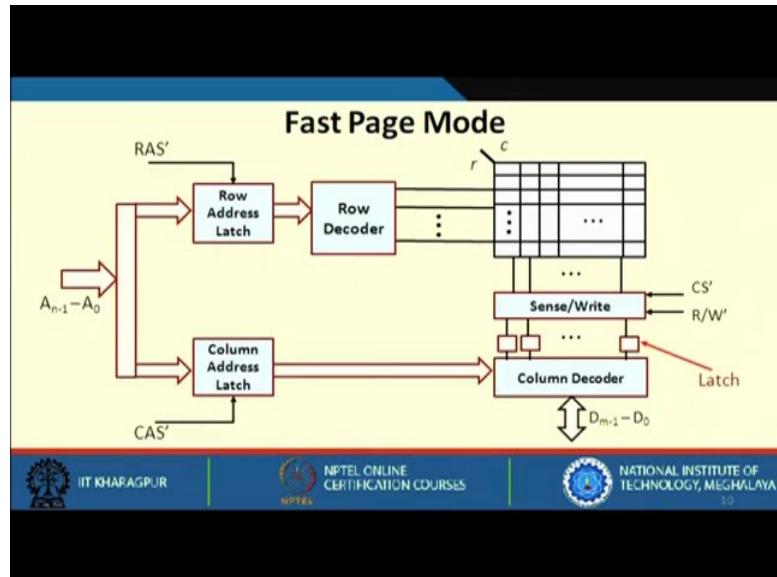
So, this is your 32 bit data bus, where this particular data will be transferred. So, how we are doing this decoding? We apply the total 25-bit address. High order bit goes to the row address. The row decoder will select a particular row. Then the low order bits are applied to the column address. This is 10 bit through this column decoder any one of the 1024 columns will get selected. So, we have a row, we have a column and then 32 bits of data transferred. So, this is how it happens in asynchronous DRAM.

(Refer Slide Time: 25:24)



Now, this is what just now I explained. 15-bit row address is selected, with the help of RAS control signal. The 15×32768 row decoder selects a particular row. And then the 10-bit column decoder is applied. And with the help of CAS the address is latched. The 10×1024 column decoder will now select a particular column. And then a group of 32 bits may be accessed. Next we will discuss another thing that is fast page mode access.

(Refer Slide Time: 26:16)



If you recall this what we were doing, when we are applying this row address and column address, we are in this particular row. And in this particular row there are any columns, but what we are doing? We are actually bringing a data of only one column, but accessing we are doing of this particular row, but we are only accessing one column data. What if you want to access other set other columns data as well in the same group? Because I have already said that we never bring a single bit of a single word rather we bring multiple words that is a block, we transfer a block. So, in this case you can see that this particular column is selected now.

What if there is a column counter and then I can also select other values of the same column? So, there are other columns, and all other columns are having this 32-bit data. Instead of again selecting a row, we have already selected a row. We have already selected a column. And let us now select next set of columns. So, then the access will be much faster. Because I have another counter that will count that will bring me to the next column address, and I will fetch the next data, and so on. This particular feature is known as fast page mode, provided the row address through this row decoder, provided the column address to this column decoder.

But only we need to have a latch in this place, because output of this will be stored here and then it will be transferred to the data lines. Then the next column address will be stored and then it will be transferred, then the next column, then the next column and so

on. So, in a single row access what we are achieving, we are not only accessing one bit of data, but one set of data that is 32 bit in this case.

(Refer Slide Time: 29:20)

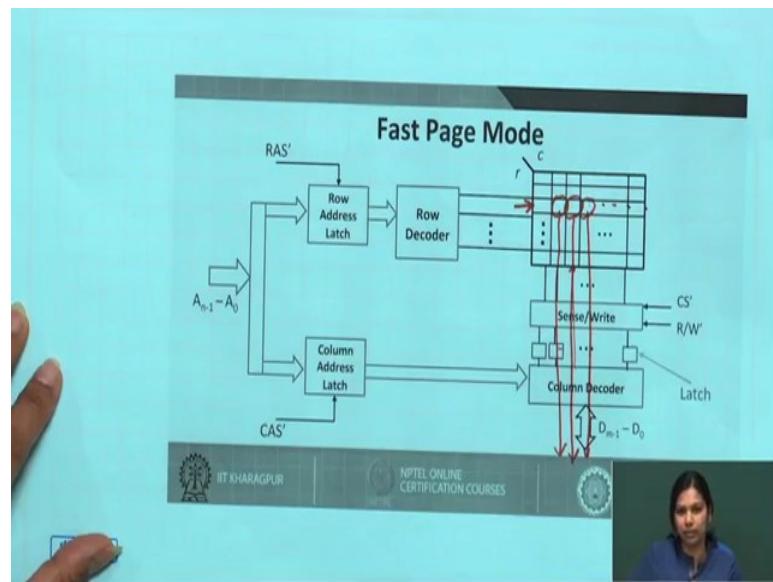
- Operation:
 - When the DRAM cell is accessed only an m-bit word (data bus width) is transferred.
 - But when we select a row, we can select not only the data of a single column but multiple columns as well.
 - A latch can be connected at the output of the sense amplifier in each column.
 - Once we apply a row address, the row gets selected.
 - Different column addresses are required to place different bytes on the data lines.
 - Hence consecutive bytes can be transferred by applying consecutive column addresses under the control of successive CAS signals.
 - This also helps in faster transfer of blocks of data.
 - This block transfer capability is termed as *Fast Page Mode* access.

So, just now what I have said, when the DRAM cell is accessed only m bit of word that is width of the data bus is transferred. But when we select a particular row, we can select not only the data of a single column, but multiple columns as well. Just now I have mentioned this. We can simply do this by connecting a latch to the output of the sense amplifier in each column.

Once we apply a row address the row gets selected. And then different column addresses are required to place different bytes on the data line; that means, we already accessed a particular row, now I have already access to one particular column; now to access the next column, I just have to give the next column address, next column address, and so on. So, I need not have to select the row again. Hence consecutive bytes can be transferred by applying consecutive column addresses under the control of successive CAS signals. But we need to give successive CAS signals for that. This also helps in faster transfer of blocks of data because this is useful when we transfer blocks of data.

So, this block transfer capability is termed as fast page mode access that I just now explained.

(Refer Slide Time: 31:20)



Here you can see that I hit this particular row, I hit this particular column, and then the data is available. Now I apply the next column address through this column decoder and I get the next data, then the next, then the next, and so on. We need not have to apply this row address again. So, this makes the transfer much faster. This is termed as block transfer capability, which is fast page mode access.

So, we came to the end of lecture 25. In the next lecture we will see synchronous DRAM. Here we have discussed about asynchronous DRAM, where the timing is dependent on whichever model is connecting to the memory. But in synchronous DRAM we will see that there is a clock involved.

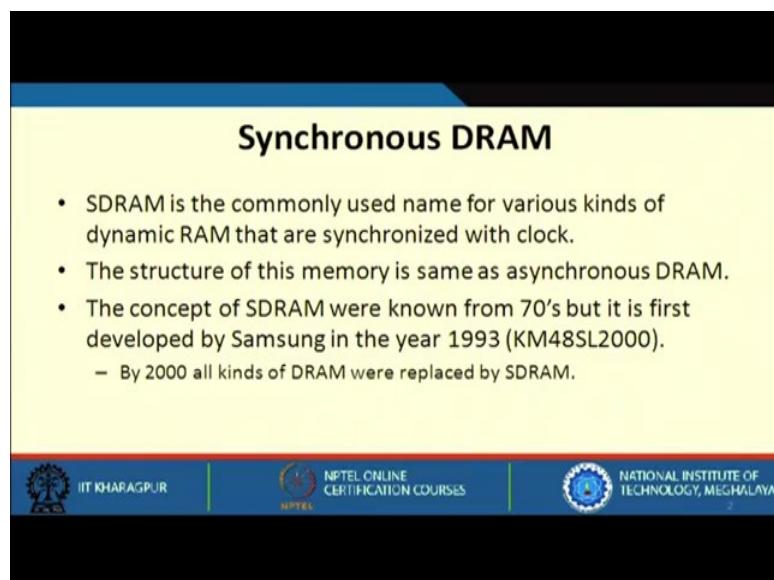
Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 26
Synchronous DRAM

Welcome to lecture 26. In this lecture we will be discussing about Synchronous DRAM.

(Refer Slide Time: 00:27)



Synchronous DRAM

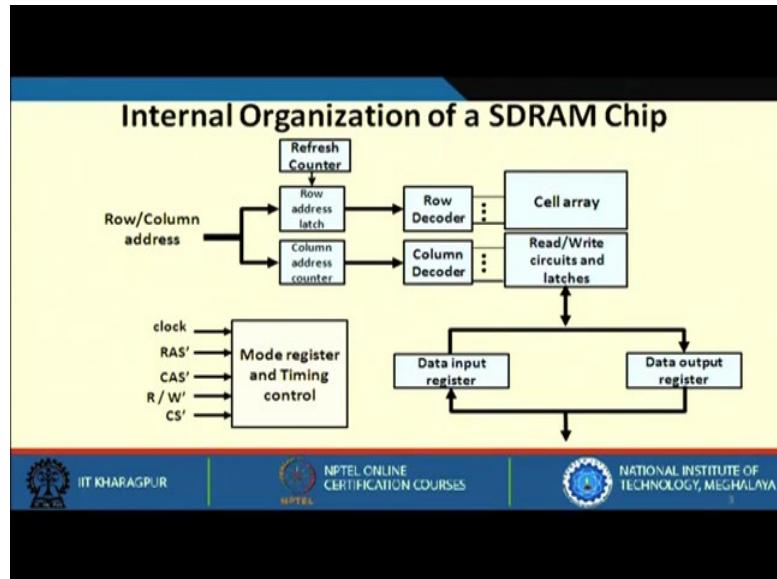
- SDRAM is the commonly used name for various kinds of dynamic RAM that are synchronized with clock.
- The structure of this memory is same as asynchronous DRAM.
- The concept of SDRAM were known from 70's but it is first developed by Samsung in the year 1993 (KM48SL2000).
 - By 2000 all kinds of DRAM were replaced by SDRAM.

IIT KHARAGPUR **NPTEL ONLINE CERTIFICATION COURSES** **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

As the name suggests synchronous, so it is synchronized with clock. SDRAM is the commonly used name for various kinds of dynamic RAM that are synchronized with clock. All such kind of DRAM that are synchronized with clock. In the previous lecture we have seen a synchronous DRAM where there is no concept of clock; the module that is connected to the memory has to take care of that, but in this case here it is synchronized with clock.

The structure of this memory is same as a synchronous DRAM. And the concepts were also known from the 70s, but it was developed in the year 1993. And by the year 2000 almost all kinds of dynamic RAM were replaced by SDRAM. So, there was no asynchronous DRAM anymore.

(Refer Slide Time: 01:52)



Now let us see the internal organization of SDRAM chip. As I said the internal organization of SDRAM is very similar to a synchronous DRAM, but it is having some more features. So, you can see that the row and column address can be provided. So, once we apply a row address, that particular address go to row decoder and a particular cell get selected, but here you see we have column address counter. So, instead of column address latch there is a counter that will be used to count to the next, next, next column for faster access.

And the read write circuit and these latches are connected to the data input register and data output register. Data input register will be required when we have to transfer a data from a data bus to this particular cell. That will come through data input register. And if you want to output something from this chip to data output register then it will be output. Along with this we have a mode register and timing control where the clock is provided, other signals like RAS and CAS are also provided, and two more signals that is read/write and chip select these are also provided here.

Apart from this we have an inbuilt refresh counter that refreshes the rows of these cells periodically. This overall diagram shows the internal organization of SDRAM that we can say that it is very much similar to a synchronous DRAM with some more features, like we have a data input register data, output register we have a column address counter

instead of column address latch we have a clock involved here and we have a mode register that we will be seeing what is the purpose of that particular register.

(Refer Slide Time: 04:34)

The slide has a yellow background with a black header and footer. The footer contains the IIT Kharagpur logo, the NPTEL logo, and a video player showing a person speaking.

- In SDRAM address and data connections are buffered by registers.
- The output of individual sense amplifier is connected to a latch.
- Mode register is present which can be set to operate the memory chip in different modes.
- To select successive columns it is not required to provide externally generated pulses on CAS line.
- A column counter is used internally to generate the required signals.

In SDRAM the address and data connections are buffered by registers, we have seen that. The output of individual sense amplifiers is connected to a latch. Mode register is present and what it does is it can be set to operate the memory chip in different modes. We will be seeing that this mode register is very much important and it can be used for speeding up, we are always trying to make memory faster such that the memory processor speed gap can be minimized.

So, these are few things that are added in synchronous DRAM to make this happen. So, mode register is present which can be set to operate the memory chip in various modes. To select successive columns it is not required to provide externally pulses generated on CAS line. What do you mean by that? You recall our previous discussion where we said that one row can be selected and for selecting the data from different columns we give a column address then we give the next column address, and so on.

For giving a column address we have a column counter that will help internally to generate the required signals. This column counter will be used internally which will be helpful in generating the signals to select various columns.

(Refer Slide Time: 06:56)

READ and WRITE Operations

- For READ operation, the row address is applied first, and in response to the column address, the data present in the latches for the selected columns are transferred to the data output register.
 - Then the data is available on the data bus.
- For WRITE operation, the row address is applied first, and in response to the column address, the data present in the data bus is made available to the latches through data input register.
 - The data is then written to the particular cell.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us see what happens for READ and WRITE operation. For READ operation the row address is applied first, and in response to the column address the data present in the latches of the selected columns are transferred to the data output registers, and then the data is available in the data bus. So, what happens in the same way for reading a cell value we apply a column address, we apply a row address first, and then we apply a column address. In response to that particular column address the column data are selected and it is stored in the data output register. In this data output register it is transferred to the data lines finally, then the data will be available on the data lines; that is the data bus and then it goes to the processor.

Similarly, for write operation the row address is applied first and in response to the column address, the data present in the data bus is made available to the latches through the data input register. And then finally, the data is written to the particular cell. So, for reading and for writing the required cell needs to be selected. After the cell gets selected the particular data will be read through this data output register or will be written to this data input register.

(Refer Slide Time: 08:48)

Burst mode transfer

- As in fast page mode, after the column address is applied the data from successive column addresses are read out or written into.
- In same way burst operation of different lengths can be specified.
- This uses the same block transfer capability of fast page mode.

Now let us see; what is this burst mode transfer. We already discussed about fast page mode where we select a particular row and a particular column and then we need not have to select that particular row again, rather next data can be available just by incrementing the column addresses. So, here also we will see that how this burst mode transfer happens. As in fast page mode, after the column addresses are applied the data from successive column addresses are read out or written into.

In the same way burst operation of different lengths can be specified. Burst operation means burst of data; a set of words are transferred. So, here we can specify how many words we want to transfer. For the burst mode transfer different lengths can be specified. This uses the same block transfer capability of the fast page mode. So, whatever we discussed in the fast page mode this block transfer mode also uses the same feature.

(Refer Slide Time: 10:18)

- Here the control signals required are provided internally by column counter and clock signals.
- New set of data are available in the data lines after every clock cycle.
- All the operations are triggered at the rising edge of the clock.
- It has built-in refresh circuitry, and refresh counter is part of it.

So, what happens here the control signals required are provided internally by the column counter and clock signals, as a clock is also associated with it. So the control signals that are required for this purpose are provided internally by the column counter and the clock signal. New set of data are available in the data lines after every clock cycle.

All operations are triggered at the rising edge of the clock, but we will see that with advancement this has changed, and we could do data transfer both in the rising edge as well as in the falling edge. It also has a built in refresh circuitry and refresh counter is part of it. So, there is a built in refresh circuitry that refreshes the different rows of the cells of the memory chip.

(Refer Slide Time: 11:45)

The slide has a black header and footer. The main content area is yellow. The title 'Standardization' is in bold black font. Below it is a bulleted list:

- SDRAM families are standardized by *Joint Electron Device Engineering Council* (JEDEC).
 - Various DDRx standards have been formulated and published by JEDEC.
 - Board manufacturers comply by the standards.

At the bottom, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Now, we must know that the standardization of this SDRAM. So, see when some products are generally built, so there must be some standardization so that different companies when they build that particular system they can be used by others. So, there should be standardization; so we standardize.

SDRAM families are standardized by Joint Electron Device Engineering Council, we call it JEDEC. So, various DDRX - DDR stands for double data rate, x stands for it can be 2, it can be 3, can be 4, and so on. So, various DDRX standards have been formulated and published by JEDEC, and board manufacturers who are building these memory chips must comply with these standards. So, whichever chip is produced must be complied with JEDEC standard.

(Refer Slide Time: 13:12)

Types of SDRAM

- Single data rate SDRAM (called SDR) can accept one command and transfer one word of data per clock cycle.
 - Data transferred typically on the rising edge of the clock.
- Double data rate SDRAM (called DDR) transfers data on both the rising and falling edges of the clock.
- DDR SDRAM was launched in 2000.
- DDR2 (2003), DDR3 (2007), DDR4 (2014).

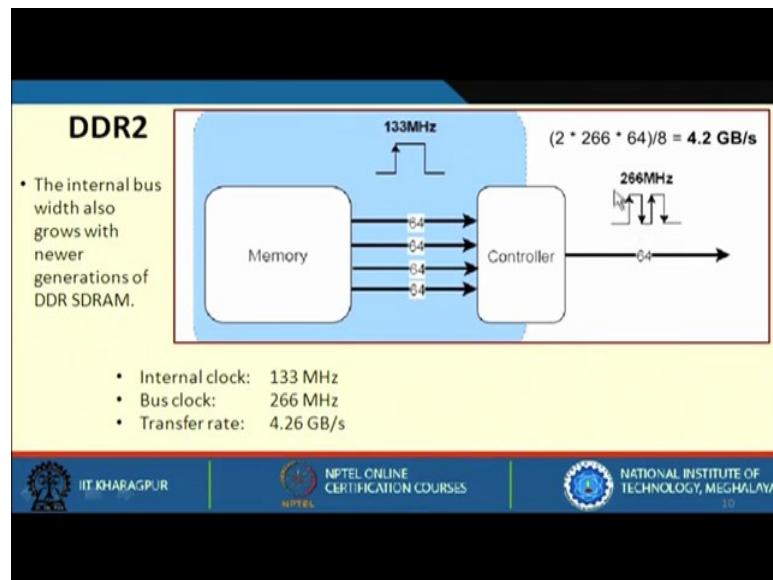
The diagram illustrates the timing of clock signals and data transfer for SDR and DDR SDRAM. The top section, labeled 'sdrclk' and 'sdrata', shows a single data transfer per clock cycle, occurring on the rising edge. The bottom section, labeled 'ddrclk' and 'ddata', shows two data transfers per clock cycle, one on the rising edge and one on the falling edge. Arrows point from the labels to the corresponding signals in each row.

Now, let us see various types of SDRAM. When SDRAM came into market it was single data rate SDRAM it is called SDR. What it can do? It can accept one command and transfer one word of data per clock cycle. That is why it is called single data rate. Data transferred typically on the rising edge of the clock.

Now, you see this diagram where it shows SDR clock, so clock is coming and you see this is the SDR data; so here this data is transferred in the rising edge of the clock, only one data is transferred. So, this is called single data rate. Let us move on with double data rate SDRAM, which is called DDR and it transfers data on both rising edge and falling edge of the clock. So, we see that within a clock two data are transferred one on the rising edge another on the falling edge.

So, these types of SDRAM are called double data rate RAM. DDR SDRAM was launched in 2000 and then with time various versions of DDR SDRAM came into market. So, DDR2 came in 2003, DDR3 came in 2007, DDR4 took 7 more years to come. And people are also saying that DDR5 is going to come in the future.

(Refer Slide Time: 15:55)



Now let us see; what is DDR2. Now we see that this is our memory and internally there is 64-bit data bus; so the internal bus width also grows with the newer generation of DDR RAM. So, with newer generation we see that the internal data bus also grows and the clock speed is 133 MHz. This is the internal clock of the memory. It has got a controller. And in double data rate RAM this bus clock must be twice this. So, if the bus clock has to be twice this it should be 266.

Now, and the data transfer will take place both twice in the falling edge and in the rising edge as well as in the falling edge. Now let us see how this calculation is actually happening. So, internal clock is 133 MHz, the bus clock is double of that and we have to calculate the transfer rate.

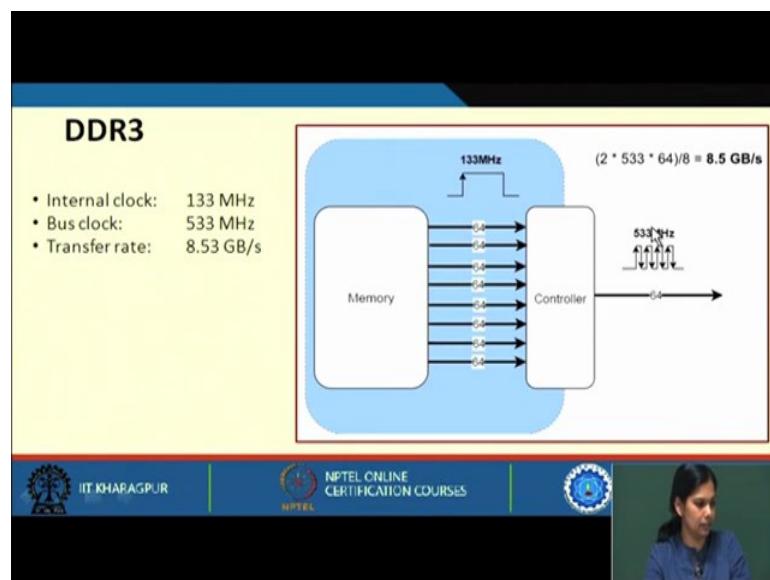
(Refer Slide Time: 17:20)

$$\begin{aligned} & \text{In 1 sec } 266 \text{ M clocks.} \\ & \Rightarrow 2 \times 266 \text{ M clock edges} \\ & \quad \text{F}\downarrow \\ & \Rightarrow 2 \times 266 \text{ M} \times 64 \text{ bits.} \\ & \Rightarrow \frac{2 \times 266 \text{ M} \times 64}{8} \text{ bytes} \\ & = 4.26 \text{ GB/s.} \end{aligned}$$

Now, in 1 second it is 266 MHz, 266 M clocks will be coming. So, if 266 M clocks will be coming and in each clock there is a rising edge and there is a falling edge. So, 2×266 M clock edges will be there. And in each rising edge and in each falling edge 64 bits of data will be transferred. So, a total of $2 \times 266M \times 64$ bits will be transferred.

So, the transfer rate comes down to 4.26 GBps in case of DDR2.

(Refer Slide Time: 18:43)

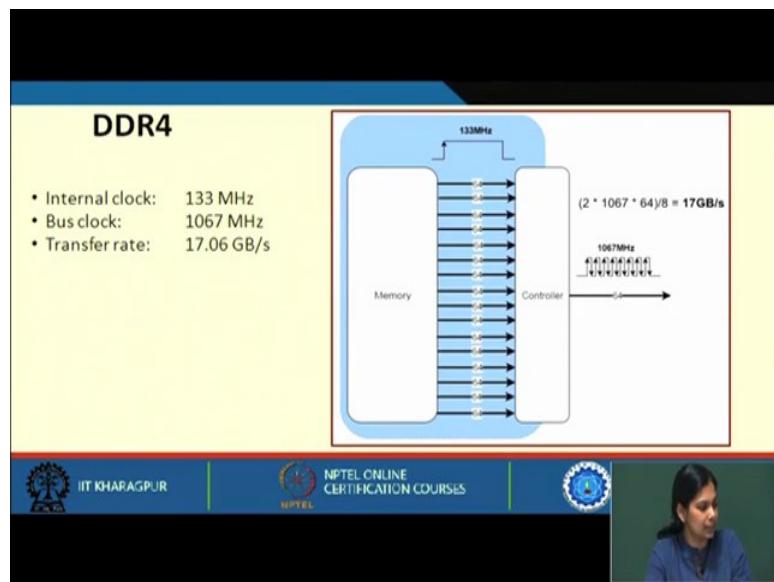


Now, let us see what happens in DDR3. Now DDR3 the internal clock is still this, but now this bus clock has become 533 MHz; means you have how many clocks in between

you have 4 clocks here. So, the width is 64, but you need to have more number of data internal bus between memory and controller. So, we have 8 here.

So, in the same way we can calculate it. So, it will be 2×533 MHz now, and each time there will be 64 bit of data will be transferred. If you divide by 8 you will get 8.53 GB per second. So, from DDR2 to DDR3 it was 4.26, now it is 8.53.

(Refer Slide Time: 20:15)



Similarly in DDR4; this is 4 times and then you have 16 such data buses in between. And here you will multiply $2 \times 1067 \times 64$ because each time 64 bits of data will be transferred divided by 8 that is roughly coming around to 17.06 GB per second.

So, as we move on the transfer rate is increasing; transfer rate is becoming faster.

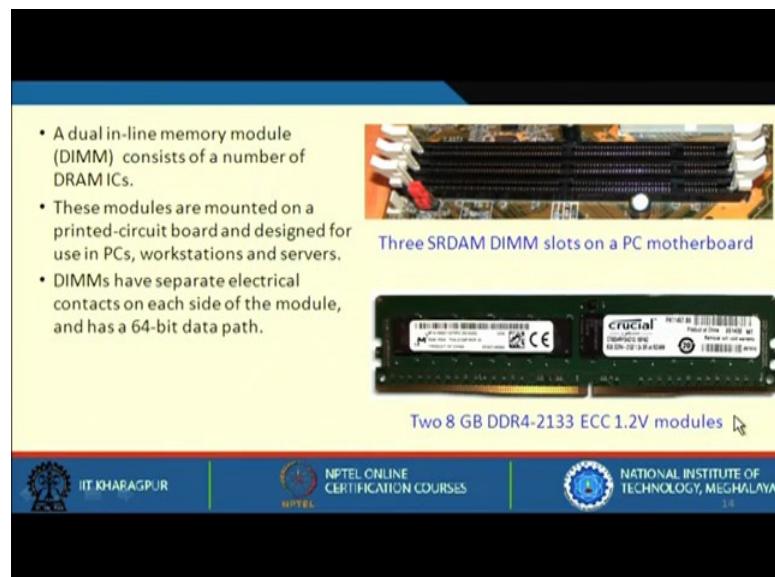
(Refer Slide Time: 21:08)

Generations of DDRx SDRAM	Name	Internal Clock	Bus Clock	Transfer Rate
DDR2-400	100 MHz	200 MHz	3.20 GB/s	
DDR2-400	133 MHz	266 MHz	4.26 GB/s	
DDR2-667	166 MHz	333 MHz	5.33 GB/s	
DDR2-800	200 MHz	400 MHz	6.40 GB/s	
DDR3-800	100 MHz	400 MHz	6.40 GB/s	
DDR3-1066	133 MHz	533 MHz	8.53 GB/s	
DDR3-1333	166 MHz	667 MHz	10.67 GB/s	
DDR3-1600	200 MHz	800 MHz	12.80 GB/s	
DDR4-1600	100 MHz	800 MHz	12.80 GB/s	
DDR4-2133	133 MHz	1066 MHz	17.06 GB/s	
DDR4-3200	200 MHz	1600 MHz	25.60 GB/s	

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, across generation we can see that the internal clock frequency has increased. And as the internal clock frequency has increased in the same way bus clock has also increased. And in turn finally, this transfer rate has become very fast, so we have very high transfer rate.

(Refer Slide Time: 21:49)

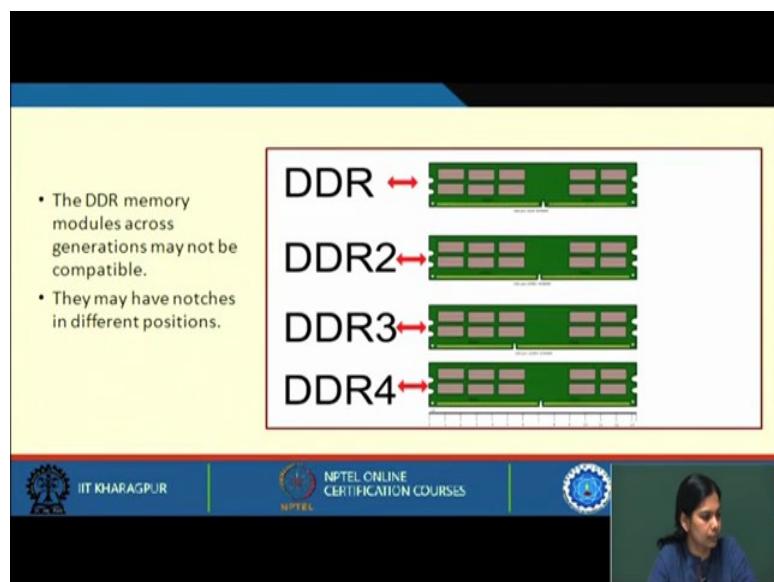


This is a dual in line memory module. These are the slots where these memory modules can fit in. A dual in line memory module consists of number of DRAM ICs. So, you can have number of DRAM ICs put in these places and these modules are mounted on a PCB

and designed for use in PCs, workstations or servers. Inside the PC will be finding that there are some slots that can be put in there.

So, this dual in line memory module have separate electrical contacts on each side of the module and has a 64 bit data paths. So, on each side of the module on this side as well as on the other side it has 64 bit data path.

(Refer Slide Time: 22:54)



Now, this is one thing we must know that the DDR memory modules across generations may not be compatible. So, you see that this, and this, so DDR2 and DDR4 are compatible almost, but DDR3 is not compatible; they may have notches in different positions. So, the notches may not be in the same positions. So, if a particular slot where we are using DDR3 we may not be able to use DDR4.

(Refer Slide Time: 23:32)

DDR Generations: To Summarize

- SDR SDRAMs can transfer one word of data per clock cycle.
- DDR (or DDR1) SDRAMs can transfer two words per clock cycle.
- DDR2 SDRAM doubles the minimum read or write unit again, to 4 consecutive words per clock cycle.
- DDR3 continues the trend, doubling the minimum read or write unit to 8 consecutive words per clock cycle.
- DDR4 extends the trend again to 16 consecutive words per clock cycle.
- In March 2017, a DDR5 standard under development has been announced.



Now to summarize what we discussed about various DDR generations. SDR SDRAM can transfer one word of data per clock cycle. DDR SDRAM can transfer two words per clock cycle; that is double data rate. DDR2 SDRAM doubles the minimum read or write unit again to four consecutive words per clock cycle. In DDR3 it is 8 consecutive words per clock cycle. And DDR4 extends the trend again to 16 consecutive words per clock cycle.

(Refer Slide Time: 25:03)

Speed of DDR Memories Across Generations

Year	Chip size	Type	Slowest DRAM	Fastest DRAM	CAS transfer time	Cycle time
2000	256 Mb	DDR1	65 ns	45 ns	7 ns	90 ns
2002	512 Mb	DDR1	60 ns	40 ns	5 ns	80 ns
2004	1 Gb	DDR2	55 ns	35 ns	5 ns	70 ns
2006	2 Gb	DDR2	50 ns	30 ns	2.5 ns	60 ns
2010	4 Gb	DDR3	36 ns	28 ns	1 ns	37 ns
2012	8 Gb	DDR3	30 ns	24 ns	0.5 ns	31 ns



So, this is the speed of DDR memories across generations. Across years we have data till 2012; this is how the chip size has grown, these are the types of DDR memories that we were using, and this is the slowest DRAM that were available, and this is the fastest DRAM that are available. And you see finally, the CAS transfer time because CAS is very important through which will actually determine finally the transfer rate, because we are once we select a row then we just give different columns address to get the data. And finally, the total cycle time comes to this. So, DDR3 with 8 GB chip size has got a cycle time of 31 nanosecond.

So, by this we came to end of lecture 36 where we discussed about synchronous DRAM and the various kind of synchronous DRAM that are there in market that are used today to bridge the speed gap between processors and memory. We will see further that how we can now build larger memories from smaller memory chips, in the next lecture.

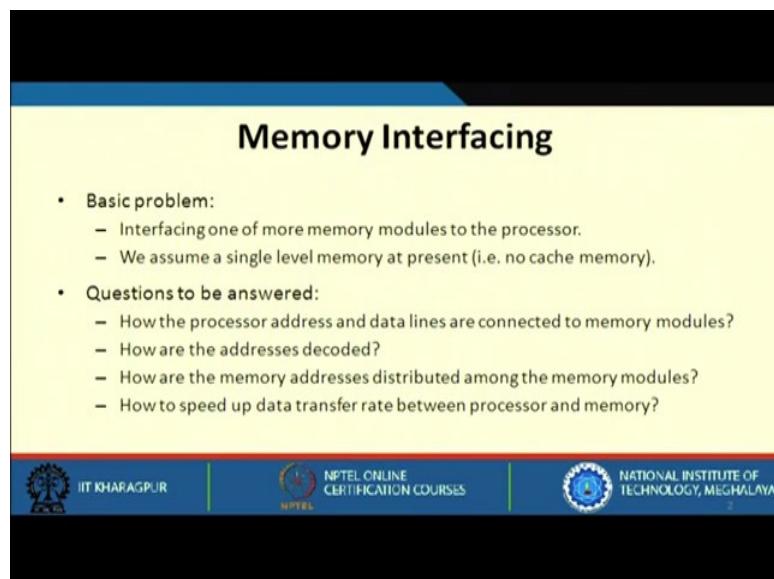
Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 27
Memory Interfacing And Addressing

Welcome to lecture 27 on Memory Interfacing and Addressing. Till now we came to know how semiconductor memory is built what are the technologies used. In this lecture we will see that we have a memory chip available to us, but we want to build a larger memory system, how we can do that. This is one thing we will be seeing in this lecture. Another thing that we will be looking into is that how we can further increase the speed of this memory.

(Refer Slide Time: 01:06)



Memory Interfacing

- **Basic problem:**
 - Interfacing one or more memory modules to the processor.
 - We assume a single level memory at present (i.e. no cache memory).
- **Questions to be answered:**
 - How the processor address and data lines are connected to memory modules?
 - How are the addresses decoded?
 - How are the memory addresses distributed among the memory modules?
 - How to speed up data transfer rate between processor and memory?

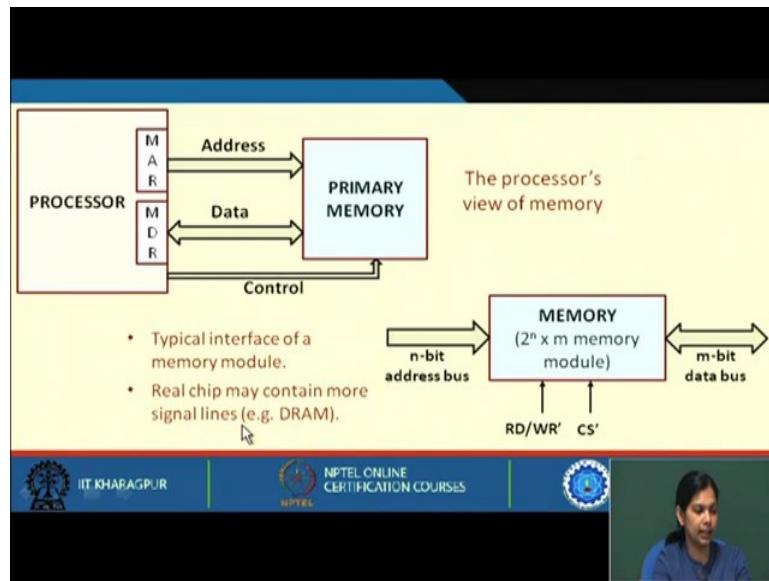
IIT Kharagpur | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

Memory interfacing is the basic problem of how we can interface one or more memory modules to the processor. Here we assume that we have a single level of memory at present, so no cache memory for the time being. The question that is to be answered here is how the processor address and data lines are connected to the memory modules. Because now we are saying that we will not be having one single memory chip, rather multiple memory chips to make that memory, how the address and data lines from the processor will be connected to this. How are the addresses decoded, how the decoding of the address will take place? Because here you have to select a module and then select

an address within that module. How are the memory addresses distributed among the memory modules that also we will be looking into, and how to speed up data transfer read between processor and memory?

So, these are the four things that we will be seeing in this lecture.

(Refer Slide Time: 02:27)



As we know that for an n-bit address bus we can have 2^n memory locations, and m data bus we have read write and control signal. And now this memory address register and memory buffer register connected through address and data lines to the primary memory. So, this is the processor's view of the memory, and of course, we have some control lines.

Now we will be seeing that now this primary memory or whatever memory we are talking about will be connected with this address line and this data line. So, typical interface of the memory module real chip may contain more signal lines, so that is what we need to see.

(Refer Slide Time: 03:25)

A Note About the Memory Interface Signals

- The data signals of a memory module (RAM) are typically bidirectional.
 - Some memory chips may have separate data in and data out lines.
- For memory READ operation:
 - Address of memory location is applied to address lines.
 - RD/WR' control signal is set to 1, and CS' is set to 0.
 - Data is read out through the data lines after memory access time delay.
- For memory WRITE operation:
 - Address of memory location is applied to address lines, and the data to be written to data lines.
 - RD/WR' control signal is set to 0, and CS' is set to 0.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

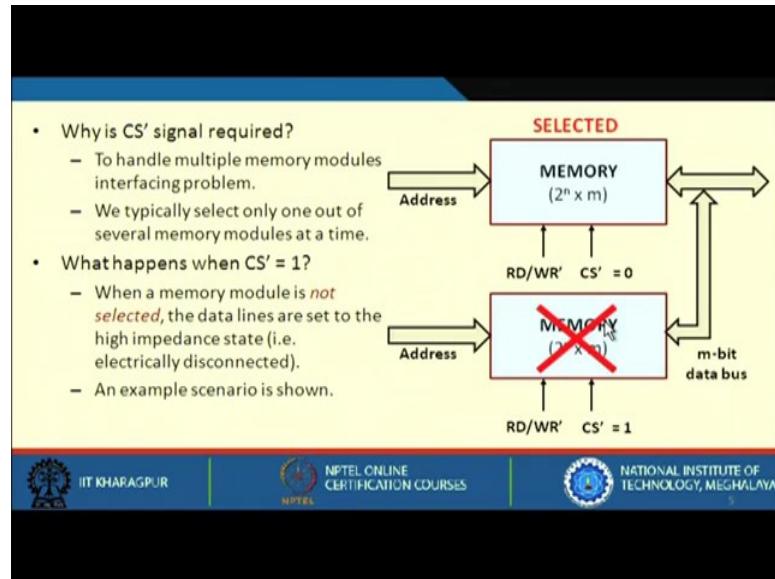


A note about this memory interface signal is that the data signals of the memory module are typically bidirectional, we have seen that the data bus is bidirectional because from memory also, some data can be passed to the data bus and written into memory, and from memory we read a data that comes to the data bus. So, it is bidirectional.

Some of the memory chips may have separate data in and data out lines. For a memory read operation what happens? The address of the memory location is applied to the address lines. Then for reading the signal this read/write signal will be set to 1 and the chip select is set to 0. Data is read out through the data lines after memory access time delay.

Similarly for a write operation, the address of the memory location is applied to the address line. The data to be written has to be written to put on the data lines. And then this read/write control signal is set to 0, and chip select is set to 0 again.

(Refer Slide Time: 05:17)



So, why is this chip select signal required? It is not required if you have a single module. It will be required only when you have multiple modules and you have to select one particular module from multiple modules. What happens when chip select is 1?

So, when the chip select line is 1, then the memory module is not selected and the data lines are set to high impedance state; that is electrically disconnected. Let us see an example scenario. Here chip select is 0 so memory module is selected, and here this chip select is 1; so this memory module is not selected. Let us take an interfacing problem where we consider that.

(Refer Slide Time: 06:18)

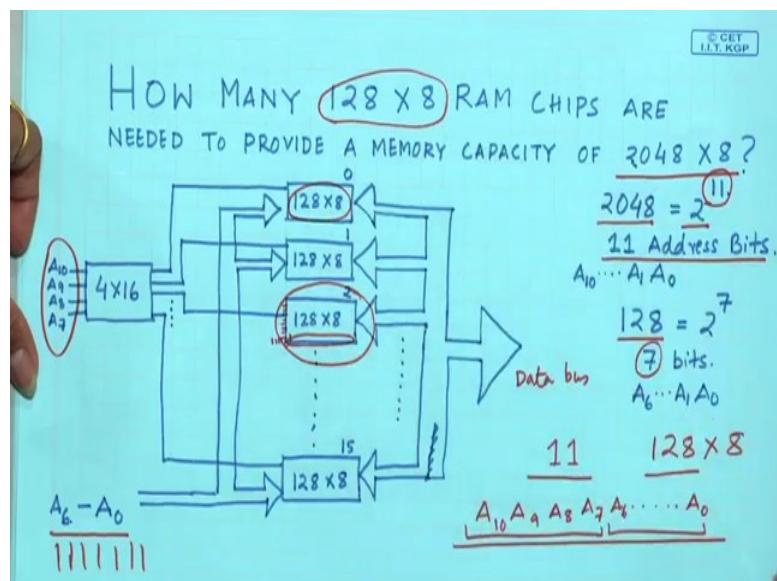
An Example Memory Interfacing Problem

- Consider a MIPS32 like processor with a 32-bit address.
 - Maximum memory that can be connected is $2^{32} = 4$ Gbytes.
 - Assume that the processor data lines are 8 bits.
- Assume that memory chips (RAM) are available with size 1 Gbyte.
 - 30 address lines and 8 data lines.
 - Low-order 30 address lines ($A_{29}-A_0$) are connected to the memory modules.
- We want to interface 4 such chips to the processor.
 - Total memory of 4 Gbytes.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, before taking an interfacing problem I will take a small example to show how a larger memory is built from smaller memory.

(Refer Slide Time: 06:38)



Let us take this example. In this example we have to find out how many 128×8 memory RAM chips are needed to provide a memory capacity of 2048×8 . Basically all you need to do is that you will divide this by this and you will get the number of memory chips required to build that; that is coming to $(2048 \times 8) / (128 \times 8) = 16$. So, we require 16 chips to built up 2048×8 memory.

Now, 2048 is your total memory that is 2^{11} , so we need to have 11 bits total in the memory address. So, total address bit is A0 to A10. Now each module you have a smaller chip which is 128 x 8. So, this module is 128, that is 2^7 . So, here you require 7 bits to select a particular row. So, once a particular row is selected then this set of 8 bits can be transferred, because one chip is 128 x 8. So, the total address bit is now 11.

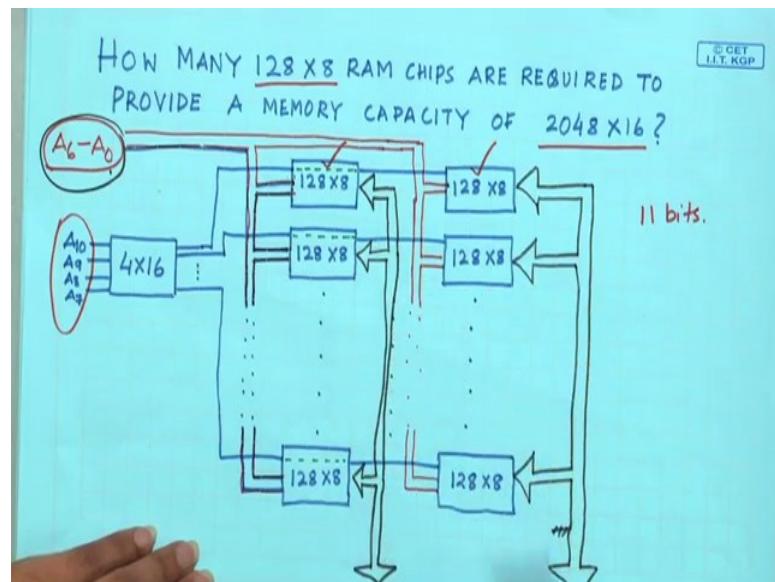
How this total address bit will be divided? We have already seen that, total address bit now here it is 11 we need to select a particular row first. In this case how many rows are there, how many chips are there in total? We have 16 total chips. So, we require a 4 x 16 decoder to select a particular chip. So, the high order 4 bits that is A10 to A7. So, the total address bit will be A10, A9 to A0 this is the total address bit 11. The high order 4 bits A10 to A7 will be connected to this 4 x 16 decoder that will select one of these 16 chips that is available. So, we have total 16 128 x 8, starting from 0 to 15. Now the output of this decoder is connected to all these chips.

Now see in the individual chip how it is organized it is organized as 128 x 8. So, the lower order 7 bits that is A0 to A6 will be connected to all the chips. So, we will first apply high order 4 bits of the total addresses to the decoder, it will decode and it will select any one of the chips, let us say this particular chip is getting selected.

Once this chip gets selected then we have to select one location from this chip, one particular row from this particular chip that will be A0 to A6. If it is the last location then the value will be 1111111. So, it will be this particular chip and the last location that is the location is 1111111.

So, this is the data bus connected here. So, this is how we can actually make a larger memory chip from a smaller memory module.

(Refer Slide Time: 12:14)



We will take another example where we will see that we need a capacity of 2048×16 and we have a chip which is 128×8 . So, we have a same 128×8 and we need a memory capacity of this much. So, if you divide this by this how many chips do you require? You will be requiring 32 chips. So, we require 32 128×8 chip to built a memory capacity of 2048×16 . So, let us now see again there are 11 bits in the address, how those 11 bits will be organized. In the same way high order 4 bits will be connected to the decoder to select any one of the 16 chips, but we have a total of 32 chips.

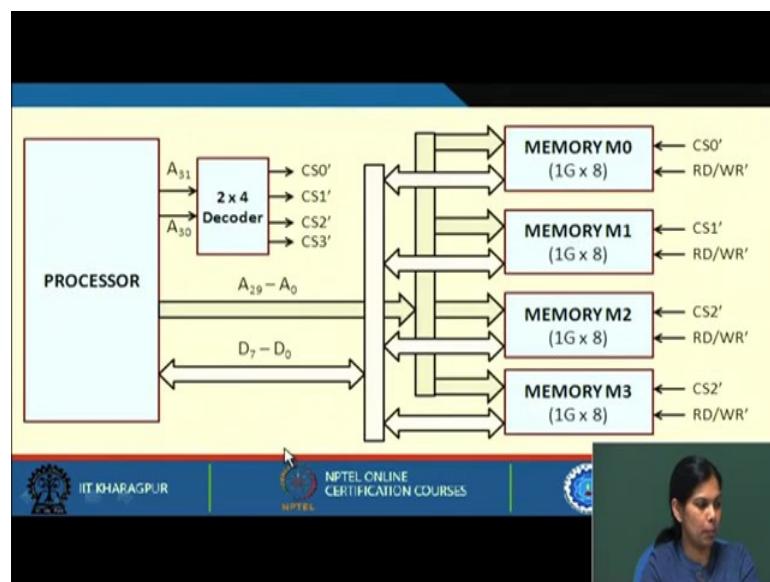
So, this decoder is now see it is connected to both the chips in the same row. So, this is a row in this particular row when I give 0 0 0 0 then this chip gets selected and also this chip gets selected simultaneously because both are connected. Now once we select this chip now I will apply the lower order bits, that is A_0 to A_6 . So, whatever address we put in the lower order address bit it goes to this particular chip and it goes to this particular chip as well. So, from here a group of 8 bits will come out, and from here a group of 8 bits will come out.

So, this is how we can organize even larger memory from a smaller memory module. Now let us take this a little bigger example. So, here we will consider a MIPS like processor with 32-bit address; maximum memory that can be connected is 2^{32} that is 4 GB. And the assumption is that the processor data lines are 8 bit.

So remember this, the processor data line is 8 bit. Assume that the memory chips are available with size 1 GB and we need a maximum memory, we need to build a memory that is 4 GB. So, roughly how many chips will be required? We will be requiring four chips. For 1 GB, 30 address lines and 8 data lines are required. So, lower order 30 address line A₀ to A₂₉ are connected to the memory modules.

And now we want to interface for such chips to the processor. Let us see how we can do this.

(Refer Slide Time: 16:44)



So, total memory of 4 GB we need to make. Let us see how we are doing here. So, we require 4 such chips. These are the 4 chips and then the 30-bit address A₀ to A₂₉ is connected to all the memory modules.

We need two more extra bits that is A₃₀ and A₃₁. So, a 2 x 4 decoder is required depending on these two values any one of the four chips will get selected. So, if you want to select chip 0, then it has to be 0 0, if you want to select chip 1, it has to be 0 1, and so on. So, this is how it is organized. And at a time when we select a particular memory module, then a group of 8 bits will be transferred through this data bus to the processor.

(Refer Slide Time: 18:11)

The slide content is as follows:

- High order address lines (A_{31} and A_{30}) select one of the memory modules.
- When is M0 selected?
 - Address is: 0 0 xxxxxxxxxxxxxxxxxxxxxxxxx
 - Range of addresses is: 0x00000000 to 0x3FFFFFFF
- When is M1 selected?
 - Address is: 0 1 xxxxxxxxxxxxxxxxxxxxxxxxx
 - Range of addresses is: 0x40000000 to 0x7FFFFFFF
- When is M2 selected?
 - Address is: 1 0 xxxxxxxxxxxxxxxxxxxxxxxxx
 - Range of addresses is: 0x80000000 to 0xBFFFFFFF
- When is M3 selected?
 - Address is: 1 1 xxxxxxxxxxxxxxxxxxxxxxxxx
 - Range of addresses is: 0xC0000000 to 0xFFFFFFFF

At the bottom of the slide, there are logos for IIT Kharagpur, NPTEL, and a woman speaking.

So, as I said higher order address line A30 and A31 selects one of the memory modules. So, what happens when M0 get selected? So, when M0 get selected the higher order address line will be 0 0, when this higher order address line is 0 0 then only memory module 0 is selected. So, what will be the range of address? So, the first address this will be 0 0 it is a 32 bit address these all are 30 bits and these two are 0 0. So, total 32 we can group it into hexadecimal digits.

So, what will be the range of address? So, as we know that this will be 0 0, first two bits will be 0 for all and the address range will actually start from here. So, till this, this will be 0 0 0 0 0 and the last address will be this last bit will be this will be 0 0 because this is the 0 th module and this will be 1 1 1 1.

So, what will be the range? The range will be 00000000 to 3FFFFFFF. This will be the range for module 0. Similarly for module 1 this will be 0 1, if this is 0 1 all will be 0 1s and all will be 1. So, if all is 0 then this will be 0 1 0 0 that is 4 first this set of 4 bits will be 4 so it will be 40000000 to 7FFFFFFF. Similarly when M2 is selected it will be 80000000 to BFFFFFFF. And similarly when M3 is selected it will be like this. So, the range of address will be like this.

(Refer Slide Time: 20:45)

- An observation:
 - Consecutive block of bytes are mapped to the same memory module.
 - For MIPS32, we have to access 32 bits (4 bytes) of data in parallel, which requires four sequential memory accesses here.
 - We shall look at an alternate memory organization later that would make this possible.
 - Called *memory interleaving*.

We can observe one thing that the consecutive block of bytes is mapped to the same memory modules. For MIPS we have to access 32 bits, that is 4 bytes, of data in parallel. So to do that we require 4 sequential memory accesses, because it is 8 bit, so one-by-one-by-one we have to access it.

Now we shall look into an alternative memory organization if there is a way that we can do something such that all the modules can be selected at once and then the data can be transferred. This is called memory interleaving.

(Refer Slide Time: 21:45)

Improved Memory Interface for MIPS32

- We make small changes in the organization so that 32-bits of data can be fetched in a single memory access cycle.
 - Exploit the concept of memory interleaving.
- The main changes:
 - High order 30 address lines (A_{31} - A_2) are connected to memory modules.
 - Low order two address lines (A_1 and A_0) are used to select one of the modules.

We make a very small change here. So, we organize these 32 bits of data that can be fetched in a single memory access, we exploit this concept of memory interleaving. The main change that we do here is that high order 30 address lines that is A₂ to A₃₁ are connected to the memory modules. And the lower order to address lines that is A₀ and A₁ are used to select one of the modules.

Now earlier the higher order bits were used to select the module. Now we are using the lower order bits to use the module. So, what will happen how the addresses will be mapped?

(Refer Slide Time: 22:45)

- How are the addresses mapped to memory modules?
 - **Module M0:** 0, 4, 8, 12, 16, 20, 24, ...
 - **Module M1:** 1, 5, 9, 13, 17, 21, 25, ...
 - **Module M2:** 2, 6, 10, 14, 18, 22, 26, ...
 - **Module M3:** 3, 7, 11, 15, 19, 23, 27, ...
- Memory addresses are *interleaved* across memory modules.
- What we can gain from this mapping?
 - Consecutive addresses are mapped to consecutive modules.
 - Possible to access four consecutive words in the same cycle, if all four modules are enabled simultaneously.

IIT KHARAGPUR | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

So, in the module 0 it will be 0, 4, 8, 12, 16, 20, and 24. Module 1 it will be 1, 5, 9; so in a block of four words consecutive addresses are now mapped into separate modules. Now in module 0 earlier we were having 0, 1, 2, 3, 4 like this.

Here, we are having the consecutive words in consecutive modules. So, what advantage can we get if we have consecutive words in consecutive modules? If there is a way to select all the modules, and can transfer the data from all these modules simultaneously, then we can get 32-bit data at a time. First address of the memory is in module 0, the next address in next module, next address in next module, next address in next module. So, this is the way to represent; we call it memory addresses are interleaved across the memory modules.

So, what we can gain from this mapping? As I said if the consecutive addresses are mapped in consecutive modules then possibly we can access the four consecutive words in the same cycle. If all four modules are enabled simultaneously, let us say all four modules are enabled simultaneously. So, from this address you can access the word from next address, from next address, from next address. So, all four words which are there in these four different modules can be accessed together.

(Refer Slide Time: 25:02)

The slide has a blue header bar at the top. Below it is a yellow main content area. At the bottom is a dark blue footer bar containing logos and text for IIT Kharagpur, NPTEL, and NIT Meghalaya.

- Motivation for word alignment in MIPS32 data words.
 - 32-bit words start from a memory address that is divisible by 4.
 - Corresponding byte addresses are (0, 1, 2, 3), (4, 5, 6, 7), (8, 9, 10, 11), (12, 13, 14, 15), etc.
 - Possible to transfer all the four bytes in a single memory cycle.
 - What happens if a word is not aligned?
 - Say: (1, 2, 3, 4) or (2, 3, 4, 5) or (3, 4, 5, 6).
 - Two of the bytes will be mapped to the same memory module.
 - Hence the word cannot be transferred in a single memory cycle.

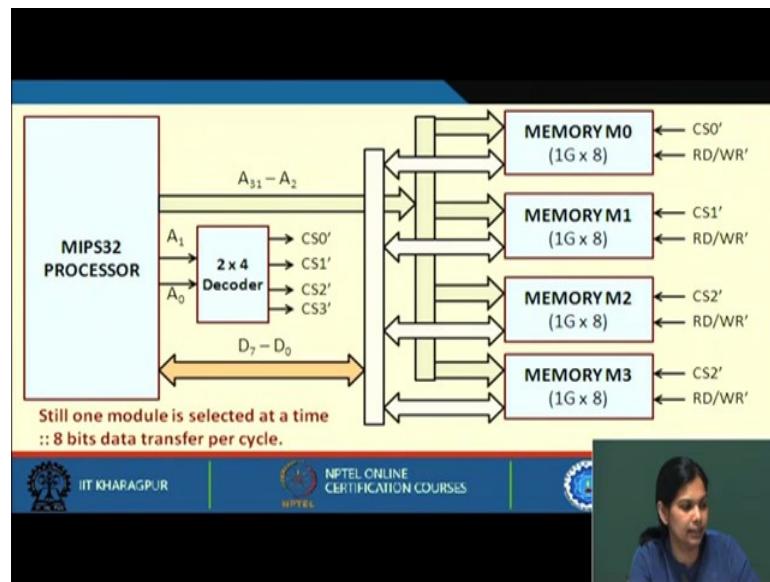
2 memory cycles required

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, now we can see the motivation for this word alignment in MIPS data word. If the words are not aligned what can happen. See the 32 bit words starts from a memory address that is divisible by 4. So, corresponding byte addresses are (0, 1, 2, 3); (4, 5, 6, 7); all these. So, these are in different modules, and it is possible to transfer all the four bytes in a single memory access. What happens if the words are not aligned? Let us say some words are starting from 1 some words are starting from 2 then what will happen? My words are in consecutive modules. So, (1, 2, 3, 4) I require; this will not help us, this is not aligned because it is not starting from 0.

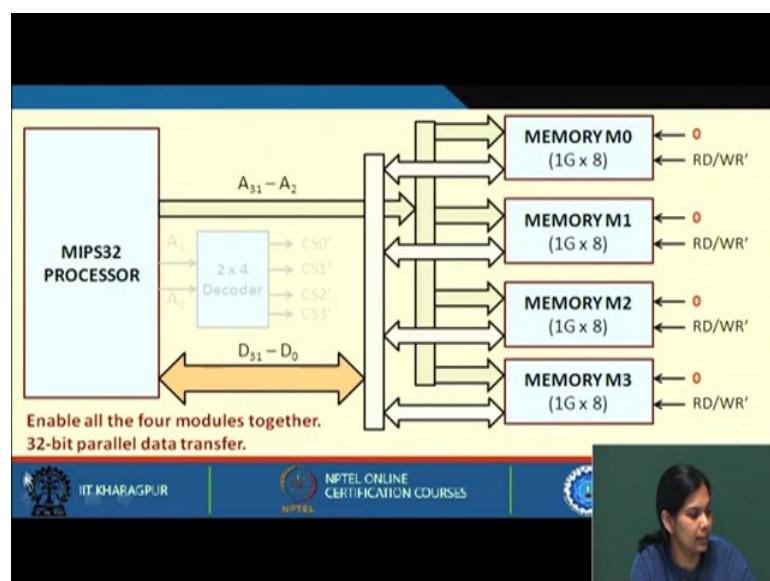
So, in that case I need 0 as well, so to do that in one memory access I can get this, and in another memory access I also need to get this 0-th location. Hence, the word cannot be transferred in single memory access. It will be accessed in a single memory cycle only when it is aligned; if it is not aligned it will require more number of memory access, in this case two memory cycles will be required.

(Refer Slide Time: 26:35)



Now, let us see what we are saying. So, the higher order bits are connected to the address modules and the lower order bit is connected to the decoder; it is still connected to the decoder and through this decoder at a time one of the memory modules gets selected. Once one of the memory modules get selected then 8 bit of data is transferred. So, a bottleneck still exists that we have a data bus of 8 bit. Still one module is selected at a time and 8 bits data transfer per cycle is taking place.

(Refer Slide Time: 27:18)



Now, let us say all these all modules are selected; all modules are selected at all times. And one more change we have done we have made this data bus as 32 bit. And the high order address is connected to all the memory modules. In all the memory modules the high order address is connected.

So, in this case what will happen? As all the modules are selected whenever we give that address first address then the data from all the four modules get selected and it will be transferred to this 32-bit data bus. So, it from this module 8 bit will come from this, this, this, and then it will be transferred to this 32-bit data bus to the processor. So, the advantage we get here is that all the modules are selected. And we apply a same address to get different data from different modules. It would not have been possible if the addresses are mapped in a single module. It is only possible because the addresses are interleaved across the modules.

So, it enables all four modules together, so 32 bit parallel data transfer is possible.

(Refer Slide Time: 28:55)

Memory Latency and Bandwidth

- **Memory Latency:**
 - The delay from the issue of a memory read request to the first byte of data becoming available.
- **Memory Bandwidth:**
 - The maximum number of bytes that can be transferred between the processor and the memory system per unit time.

We already discussed about this regarding memory latency and bandwidth, so we will take a small example to explain this. What is latency? Latency is the delay from the issue of the memory read request to the first byte of data becoming available. So, it is the time required to access the first data and then consecutive data can be accessed in a much quicker time.

What is memory bandwidth? The memory bandwidth is the maximum number of bytes that can be transferred between the processor and the memory system per unit time.

(Refer Slide Time: 29:58)

- Example 1:
Consider a memory system that takes 20 ns to service the access of a single 32-bit word.
 - Latency L = 20 ns per 32-bit word.
 - Bandwidth BW = $32 / (20 \times 10^{-9}) = 200$ Mbytes per second.
- Example 2:
 - The memory system is modified to accept a new (still 20ns) request for a 32-bit word every 5 ns by overlapping requests.
 - Latency L = 20 ns per 32-bit word (*no change*).
 - Bandwidth BW = $32 / (5 \times 10^{-9}) = 800$ Mbytes per second.

Let us consider this example. So, here a memory system takes 20 nanosecond to service the access of a single 32-bit word. That means, 20 nanosecond is the latency; in 20 nanosecond it is transferring a single 32-bit word. What will be the bandwidth? Bandwidth will be $32 / (20 \times 10^{-9})$.

(Refer Slide Time: 30:36)

$$\frac{32}{20 \times 10^{-9}} \text{ bits/sec}$$
$$= 1600 \times 10^6 \text{ bits/sec.}$$
$$= 1600 \text{ M bits/sec.}$$
$$= 200 \text{ Mbytes/sec.}$$

So, you can just see this that latency is 20 nanosecond. This is the bandwidth so you can just do a simple calculation which is coming down to 1600 megabits per second. Finally, if you divide it by 8 you will get 200 megabytes per second. So, the bandwidth is 200 megabytes per second.

Let us take another example.

Here the memory system is little bit modified to accept a new, still this 20 nanosecond request, for a 32 bit word every 5 nanosecond by overlapping the request. Now the next word, next word, next word is overlapped; that means every 5 nanosecond the next word is available. So, how we can change this? So, the latency will be still 20 nanosecond per word so there is no change there, but now the bandwidth can be increased, because after every 5 nanosecond the next, next, next word is available. So, it is 32 divided by 5 into 10 to the power minus 9 which is coming down to 800 megabytes per second.

So, we have seen some of the examples, and what is latency and bandwidth, and how are they important in the context of memory system design. So now, we came to the end of lecture 27. In the next lecture we will be looking into how we can further make the memory faster. So, we will be moving on with memory hierarchy, cache memory, virtual memory, etc.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 28
Memory Hierarchy Design (Part I)

Welcome to week 6. In this week we will be looking into Memory Hierarchy Design, and of course cache memory. In previous week we discussed about memories, we discussed about static memories, dynamic memories, RAM; we also discussed about how we can actually design larger memory modules from smaller memory modules and memory interleaving.

In this week we will be looking mostly into how we can make memory faster by incorporating some strategies. One of the methods that we will be seeing in more detail is the cache memory. And we will also be focusing on memory hierarchy design.

(Refer Slide Time: 01:37)

The slide has a dark blue header bar. Below it, the title 'Introduction' is centered in a large, bold, black font. The main content area is a light beige color. It contains a bulleted list of points. At the bottom, there is a dark footer bar with three logos and their respective names: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

Introduction

- Programmers want unlimited amount of memory with very low latency.
- Fast memory technology is more expensive per bit than slower memory.
 - SRAM is more expensive than DRAM, DRAM is more expensive than disk.
- Possible solution?
 - Organize the memory system in several levels, called *memory hierarchy*.
 - Exploit temporal and spatial locality on computer programs.
 - Try to keep the commonly accessed segments of program / data in the faster memories.
 - Results in faster access times on the average.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

The programmers always want unlimited amount of memory with a very low latency. We need high speed and we also need more memory. We have also seen that fast memory technology is more expensive per bit than slower memory. SRAM is much more expensive than DRAM, but DRAM is again more expensive than disk. So, SRAM cannot be made much larger, where DRAM can be made much larger compared to

SRAM. Again DRAM cannot be made as large as disk, and disk speed cannot match the speed of DRAM.

So, what is the possible solution? Organize the memory system in several levels, which is called memory hierarchy, and exploit both temporal and spatial locality of computer programs. We will look into the details of what is temporal and spatial locality. And we also try to keep the commonly accessed segment of programs or data in a faster memory called cache memory. So by this, what we mean is that the frequently used data or instructions can be kept in a high speed memory, because a particular data which I am requiring now it might happen I will be requiring it after some time again.

So, instead of keeping it in a slower memory let us keep it in a fast memory and as and when required by the processor it can get it from the faster memory and not from the slower memory.

(Refer Slide Time: 03:51)

Quick Review of Memory Technology

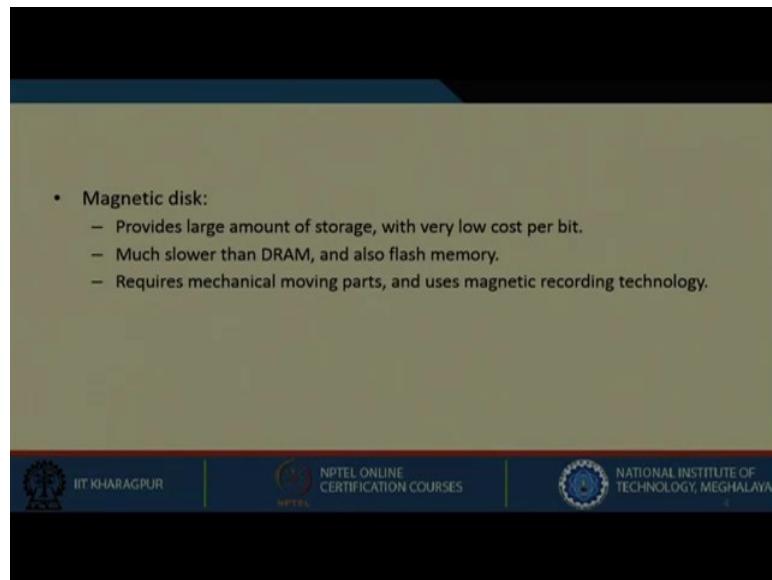
- Static RAM:
 - Very fast but expensive memory technology (requires 6 transistors / bit).
 - Packing density is limited.
- Dynamic RAM:
 - Significantly slower than DRAM, but much less expensive (1 transistor / bit).
 - Requires periodic refreshing.
- Flash memory:
 - Non-volatile memory technology that uses floating-gate MOS transistors.
 - Slower than DRAM, but higher packing density, and lower cost per bit.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

This results in faster access time on an average. Let us have a quick review of the memory technology that we discussed last week. Static RAM, which is very fast but expensive memory as it requires 6 transistors per bit, and the packaging density is limited. So, within a small area we cannot have very large memory in place. Whereas, dynamic RAM is significantly slower than SRAM, but much less expensive that is only one transistor per bit is required. And it also requires periodic refreshing which is not required in static RAM.

So, DRAM is much slower than SRAM, but it is much less expensive, and also it requires periodic refreshing which is not required in SRAM. And flash memory is a non-volatile memory technology that uses floating gate MOS transistors. It is of course slower than DRAM, but has higher packaging density and lower cost per bit.

(Refer Slide Time: 05:09)



And magnetic disk provides large amount of storage and the cost per bit is also pretty less, but it is much slower than DRAM and also flash memory. And compared to other memories this requires a mechanical moving part and uses magnetic recording technology. The disk moves around and we actually get the data from different tracks and sectors. So, there is a moving part, whereas in DRAM or SRAM no such thing is there.

(Refer Slide Time: 05:53)

Memory Hierarchy

- The memory system is organized in several levels, using progressively faster technologies as we move towards the processor.
 - The entire addressable memory space is available in the largest (but slowest) memory (typically, magnetic disk or flash storage).
 - We incrementally add smaller (but faster) memories, each containing a subset of the data stored in the memory below it.
 - We proceed in steps towards the processor.

IIT KHARAGPUR | NPTEL: ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Coming to memory hierarchy, the memory system is organized in several levels. By hierarchy we mean it is divided into many levels using progressively faster technologies as we move towards the processor. Thus there are different levels of memory, and the level that is closest to the processor is faster, and which are little further from the processor are slower.

The entire addressable memory space is available in the largest, but slowest memory; typically magnetic disk or flash storage. The addressable space can be as large as data on the disk, but we are actually implementing the levels one by one. So, at the lowest level where we have cache that is much smaller, then we go to next level which can be L2 cache or it can be main memory, it can be little larger. But how we can speed up? We can transfer data by replacing the data that is currently in that particular fast memory we will move it to the slower memory, again from the slower memory will bring to the faster memory. This is how we perform the things.

So, we incrementally add smaller, but faster memories each containing a subset of data stored in memory below it. We proceed in steps towards the processor.

(Refer Slide Time: 08:09)

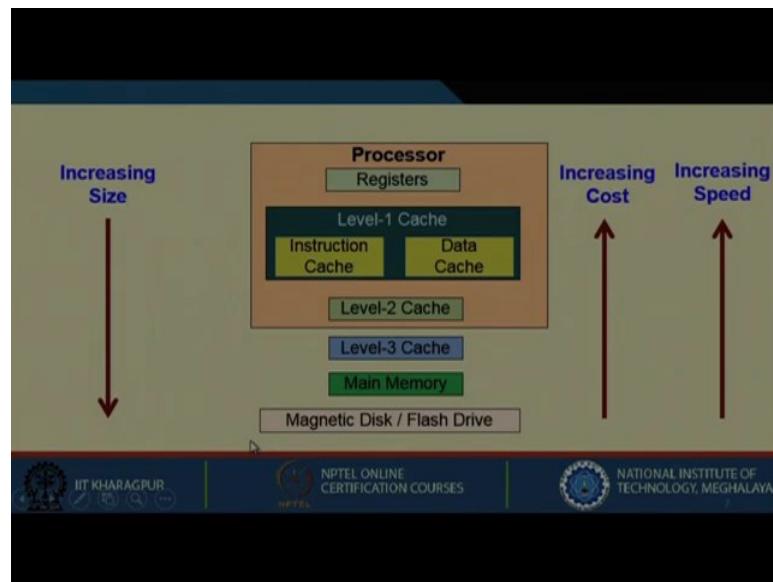
- Typical hierarchy (starting with closest to the processor):
 1. Processor registers
 2. Level-1 cache (typically divided into separate instruction and data cache)
 3. Level-2 cache
 4. Level-3 cache
 5. Main memory
 6. Secondary memory (magnetic disk / flash drive)
- As we move away from the processor:
 - Size increases
 - Cost decreases
 - Speed decreases

Let us see this. Typical hierarchy starts with the one closest to processor, which are the processor registers. Then we have Level-1 cache, typically divided into separate instruction and data cache. We have already talked about Harvard and von Neumann architecture in the first week. If you recall we said that if we have separate data and instruction memory, then instruction fetch and data access can be done at the same time.

So, we typically divide Level-1 into instruction cache and data cache; we can have level 2 cache, then level 3 cache, we then have main memory, and finally we have secondary memory. So, processor cache will be the smallest one, then the level 1 cache and so on, the secondary memory will be the largest memory.

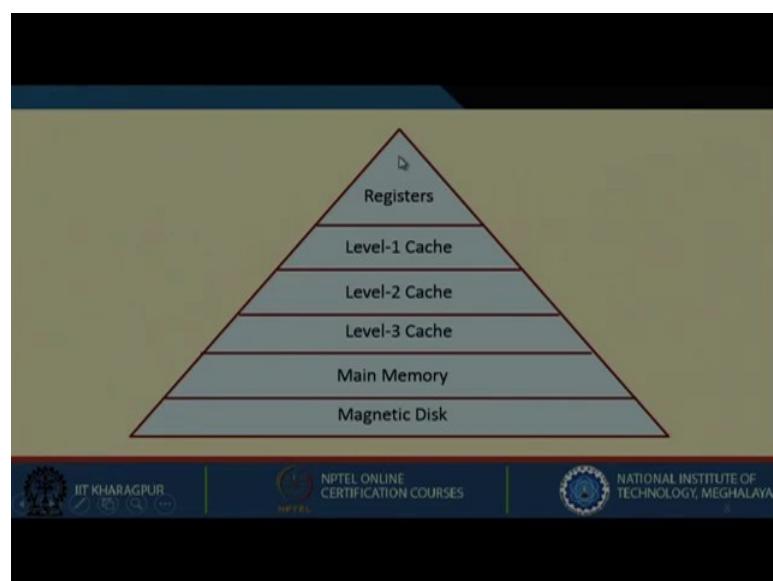
As we move away from processor the size increases. So, this is the smallest one, then the size increases little more, little more. The cost also decreases, because as you are closest to the processor it is much faster, but as we are moving away from the processor the cost slowly decreases, but at the same time the speed also decreases. So, this is the trade off you can see.

(Refer Slide Time: 10:05)



Let us see this. This is processor register, level 1 cache; we have instruction and data cache, then level 2 cache, level 3 cache, main memory and magnetic disk. As we move from processor to magnetic disk the size increases, so the size of magnetic disk is the maximum. But as you move up the speed increases as well as the cost increases; so the cost becomes much more as you are moving to the memory that is closest to the processor.

(Refer Slide Time: 10:58)



So, this is basically a pyramid structure that shows registers, then level 1 cache, then level 2 cache, and so on. So, the size is increasing, but there are few things that are also decreasing with the size.

(Refer Slide Time: 11:18)

The slide title is "A Comparison". It contains a table comparing memory levels based on typical access time and capacity. The table has four columns: Level, Typical Access Time, Typical Capacity, and Other Features. The rows include Register, Level-1 cache, Level-2 cache, Level-3 cache, Main memory, and Magnetic disk. The "Other Features" column indicates whether the memory is On-chip or Off-chip.

Level	Typical Access Time	Typical Capacity	Other Features
Register	300-500 ps	500-1000 B	On-chip
Level-1 cache	1-2 ns	16-64 KB	On-chip
Level-2 cache	5-20 ns	256 KB – 2 MB	On-chip
Level-3 cache	20-50 ns	1-32 MB	On or off chip
Main memory	50-100 ns	1-16 GB	
Magnetic disk	5-50 ms	100 GB – 16 TB	

Logos for IIT Kharagpur, NPTEL, and NIT Meghalaya are at the bottom.

Now, this is a comparison that has been made. For registers the typical access time is of the order of picosecond, level 1 cache this is 1 to 2 nanosecond, level 2 cache is 5 to 20 nanosecond. So, the access time is increasing slowly, and at the same time if you see the capacity, it is also increasing. The L1 cache and L2 cache can be on chip and L3 can be off chip or it can also be on chip.

(Refer Slide Time: 12:02)

Major Obstacle in Memory System Design

- Processor is much faster than main memory.
 - Has to spend much of the time waiting while instructions and data are being fetched from main memory.
 - Memory speed cannot be increased beyond a certain point.

The graph illustrates the exponential growth of processor performance compared to the slower, linear growth of memory performance over three decades. This increasing gap is identified as a major obstacle in memory system design.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, what is the major obstacle in memory system design? We have already seen this slide before. Processor is much faster than main memory; this is the growth of the processor speed and this is the growth of the memory. So, basically you see this gap is always increasing. So, memory speed cannot be increased beyond a certain point that is why we are coming up with many techniques through which we can actually increase the speed.

(Refer Slide Time: 12:54)

Impact of Processor / Memory Performance Gap

Year	CPU Clock	Clock Cycle	Memory Access	Minimum CPU Stall Cycles
1986	8 MHz	125 ns	190 ns	$190 / 125 - 1 = 0.5$
1989	33 MHz	30 ns	165 ns	$165 / 30 - 1 = 4.5$
1992	60 MHz	16.6 ns	120 ns	$120 / 16.6 - 1 = 6.2$
1996	200 MHz	5 ns	110 ns	$110 / 5 - 1 = 21.0$
1998	300 MHz	3.33 ns	100 ns	$100 / 3.33 - 1 = 29.0$
2000	1 GHz	1 ns	90 ns	$90 / 1 - 1 = 89.0$
2002	2 GHz	0.5 ns	80 ns	$80 / 0.5 - 1 = 159.0$
2004	3 GHz	0.33 ns	60 ns	$60 / 0.33 - 1 = 179.0$

Ideal memory access time = 1 CPU cycle
Real memory access time >> 1 CPU cycle

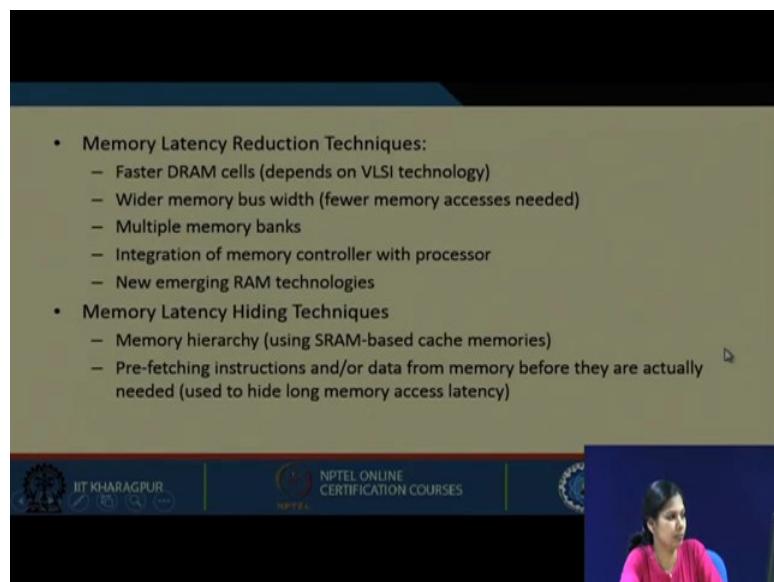
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us see the impact of processor and memory performance gap over the years. You can see this is the CPU clock, this will be the clock cycle time, and this is the memory access

time. So, what is happening is that the clock cycle time and the memory access time gap is increasing; the processor clock speed is becoming higher, with that the clock cycle time is getting reduced, but the gap between clock cycle time and the memory access time is more.

The minimum CPU stall cycles can be given by this. The data is provided till 2004 that shows that minimum CPU stall cycle will be 179.

(Refer Slide Time: 14:21)



Memory latency reduction techniques say how we can reduce the access time. If it is reduced what are the techniques that can be used. One is faster DRAM cell that will depend on VLSI technology, and wider memory bus width with fewer memory access needed. So, we access once and we get the data all together.

So, actually you are using multiple memory banks with memory interleaving, integration of memory controller with processor, we can also use some emerging RAM technologies. And under memory latency hiding techniques we have memory hierarchy using SRAM-based cache memory. So, we are having a fast memory and we will see that most of the access will be made to this particular memory. Prefetching instruction or data from memory before they are actually needed will also help. Prefetching is a technique that can be used to hide this memory latency.

(Refer Slide Time: 15:50)

Locality of Reference

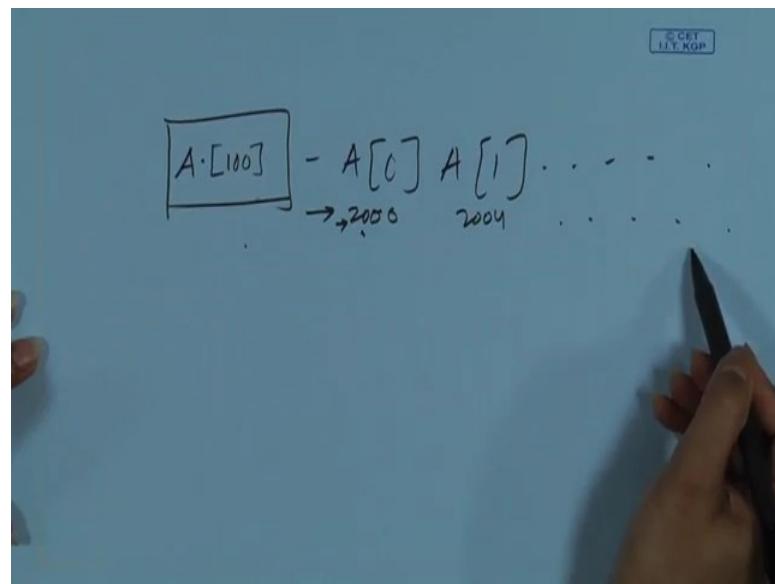
- Programs tend to reuse data and instructions they have used recently.
 - Rule of thumb: 90% of the total execution time of a program is spent in only 10% of the code (also called 90/10 rule).
 - Reason: nested loops in a program, few procedures calling each other repeatedly, arrays of data items being accessed sequentially, etc.
- Basic idea to exploit this rule:
 - Based on a program's recent past, we can predict with a reasonable accuracy what instructions and data will be accessed in the near future.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, we come to locality of reference. There is a property that programs tend to reuse data and instruction they have used recently; that means, an instruction that is used at time t it is much likely that it will be used again at some point of time very soon. This is called locality of reference; the rule of thumb says that 90% of the total execution time of the program is spend in only 10% of the code. This is called 90/10 rule.

So, only 10% of the code is been used because of loops. If you consider a loop where certain statement and certain instructions are getting executed repeatedly. If we bring those instructions into some faster memory, you can actually have a better access time because you have brought the data from a slower memory into a faster memory, and now you are accessing repeatedly from the faster memory. That is why cache can be helpful in such scenario although we are bringing the data from one memory to another memory, but in turn we are getting advantage out of it.

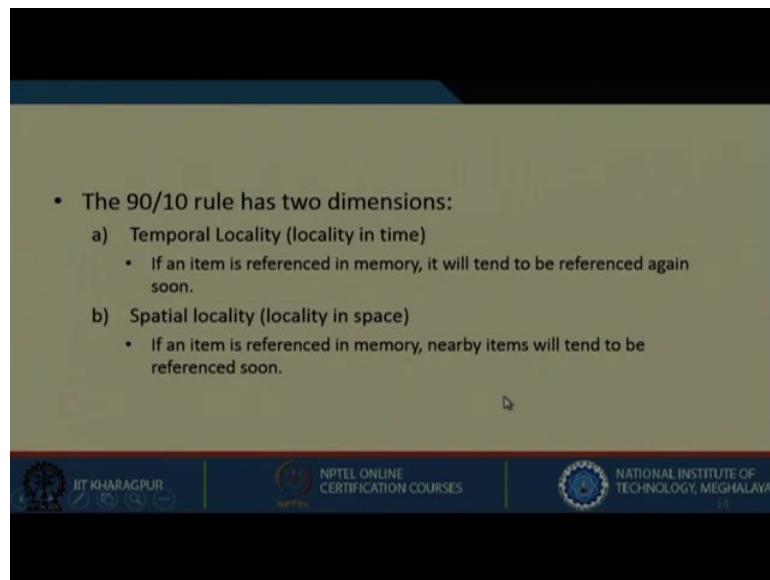
(Refer Slide Time: 18:12)



Let us say this is an array of 100 elements. So, you will access first element, you will access next element, and so on. So, actually you are making an access to some memory location let us say 2000, then 2004, and so on. So, if an instruction or data is required which is in some location, it is expected that data in the nearby locations is also required in the near future. So, instead of bringing only this we can bring the whole set of array together into the cache. So, this is where it helps and this locality of reference is coming into picture. So, there are two things; we call it spatial locality of reference and temporal locality of reference.

The basic idea is that based on program's recent past we can predict with a reasonable accuracy what instructions and data will be accessed in near future.

(Refer Slide Time: 19:35)



The 90/10 rule has two dimensions, one is called temporal locality --- locality in time; that means, if I am accessing an element at time t it is likely that I will be accessing that same element at time $t + \text{something}$. So, if an item is referenced in memory it will tend to be referenced again very soon because of loops.

So, if an item is referenced in memory nearby items will tend to be referenced soon; that means, let us say we have written a code and that code takes some, say 20 words, to store that particular program. Now, if you take one word at a time it will not help because when you are bringing one particular word it is likely that we require 19 more words associated with that program. So, why not to bring the entire thing into cache memory such that next time when you are accessing you will get it from the cache memory and not from the main memory. So, this is spatial locality.

(Refer Slide Time: 21:11)

(a) Temporal Locality

- Recently executed instructions are likely to be executed again very soon.
- Example: computing factorial of a number.

fact = 1;
for k = 1 to N
fact = fact * k;

Loop:
ADDI \$t1,\$zero,1
ADDI \$t2,\$zero,N
ADDI \$t3,\$zero,1
MUL \$t1,\$t1,\$t3
ADDI \$t3,\$t3,1
SGT \$t4,\$t3,\$t2
BNEZ \$t4,Loop

The four instructions in the loop are executed more frequently than the others.

JIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us take this example of temporal locality; recently executed instructions are likely to be executed again very soon. The example is computing factorial of a number.

It says that because of loop structure this instruction will not be required for us to bring it from a slower memory, because we will bring it once from slowest memory to fastest memory and then we will keep it there. So, this is temporal locality this is an example of temporal locality.

(Refer Slide Time: 24:04)

(b) Spatial Locality

- Instructions residing close to a recently executing instruction are likely to be executed soon.
- Example: accessing elements of an array.

sum = 0;
for k = 1 to N
sum = sum + A[k];

Loop:
SUB \$t1,\$t1,\$t1
ADDI \$t2,\$zero,N
ADDI \$t3,\$zero,1
ADDI \$t5,\$zero,A
LW \$t8,0(\$t5)
ADD \$t1,\$t1,\$t8
ADDI \$t3,\$t3,1
SGT \$t4,\$t3,\$t2
BNEZ \$t4,Loop

Performance can be improved by copying the array into cache memory.

JIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us see spatial locality. Instructions residing close to recently executing instructions are likely to be executed soon; that means, this instruction is in close proximity of other instruction. So, if I am bringing this particular instruction it is also better that we bring some more instruction that is in the close proximity of this instruction.

(Refer Slide Time: 26:58)

Performance of Memory Hierarchy

- We first consider a 2-level hierarchy consisting of two levels of memory, say, M_1 and M_2 .

```
graph LR; CPU[CPU] --- M1[M1]; M1 --- M2[M2]
```

The diagram illustrates a 2-level memory hierarchy. It consists of three rectangular boxes arranged horizontally. The first box on the left is labeled "CPU". An arrow points from the "CPU" box to the second box, which is labeled "M₁". Another arrow points from "M₁" to the third box, which is labeled "M₂".

IIT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

We first consider a 2 level hierarchy consisting of two levels of memory, say M₁ and M₂. CPU is first hitting M₁, and if it is found here it will take it send the data to CPU, and if it is not found it is brought from M₂ to M₁, and then may be transferred to CPU.

(Refer Slide Time: 27:26)

• Cost:

- Let c_i denote the cost per bit of memory M_i , and S_i denote the storage capacity in bits of M_i .
- The average cost per bit of the memory hierarchy is given by:
↳ Cost $c = \frac{c_1S_1 + c_2S_2}{S_1 + S_2}$

- In order to have $c \rightarrow c_2$, we must ensure that $S_1 \ll S_2$.

So, how we can calculate the cost? Let c_i denote the cost per bit of memory M_i , and S_i denote the storage capacity in bits of M_i . The average cost per bit of the memory hierarchy is given by this expression.

What we are trying to say is that c will be roughly equivalent to c_2 ; that is, cost should be less, but for that we must ensure that S_1 is much less than S_2 , the size of M_1 memory should be less than size of M_2 .

(Refer Slide Time: 28:49)

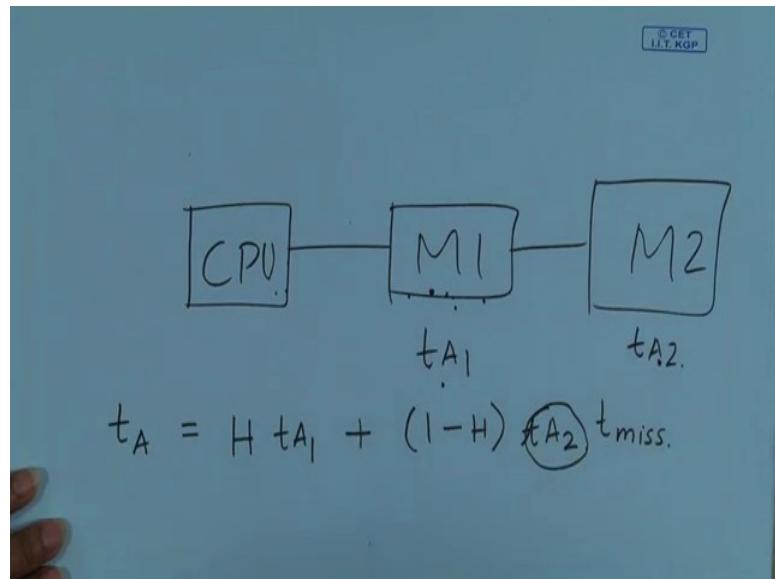
• Hit Ratio / Hit Rate:

- The hit ratio H is defined as the probability that a logical address generated by the CPU refers to information stored in M_1 .
- We can determine H experimentally as follows:
 - ↳ • A set of representative programs is executed or simulated.
 - The number of references to M_1 and M_2 , denoted by N_1 and N_2 respectively, are recorded.

$$H = \frac{N_1}{N_1 + N_2}$$

Coming to hit ratio or hit rate what do you mean by that? The hit ratio H is defined as the probability that a logical address generated by the CPU refers to information stored in M.

(Refer Slide Time: 29:15)



So, let us see this; this is your CPU and you have M1 level and you have M2 level, and we are saying the CPU will be hitting this particular memory first. This means CPU will hit M1 and it will get the data from M1; that is hit ratio. So, the percentage time the data is found in M1 is the hit ratio.

So, hit ratio H is defined as the probability that logical address generated by the CPU refers to the information stored in M1; that means, the data which I am looking for is present in M1. We can determine H experimentally as follows. A set of representative programs is executed or simulated; then the number of references to M1 and M2 denoted by n1 and n2 respectively is measured, and then hit ratio can be n1 divided by n1 + n2.

The quantity 1-H is called the miss ratio; that means, the number of times it is not found in M1 cache.

(Refer Slide Time: 31:10)

The slide contains a list of bullet points and associated text:

- Access Time:
 - Let t_{A1} and t_{A2} denote the access times of M_1 and M_2 respectively, relative to the CPU.
 - The average time required by the CPU to access a word in memory can be expressed as:
$$t_A = H \cdot t_{A1} + (1 - H) \cdot t_{MISS}$$
where t_{MISS} denotes the time required to handle the miss, called miss penalty.

At the bottom of the slide, there is a navigation bar with icons for back, forward, search, and other course-related links. Logos for IIT Kharagpur, NPTEL, and NIT Meghalaya are also present.

So, now let us see the access time. Let t_{A1} and t_{A2} denote the access times of M_1 and M_2 respectively relative to CPU. How we can actually tell the average time required by CPU to access the word. It is given by this expression.

(Refer Slide Time: 33:56)

The slide contains a list of bullet points and associated text:

- The miss penalty t_{MISS} can be estimated in various ways:
 - a) The simplest approach is to set $T_{MISS} = t_{A2}$, that is, when there is a miss the data is accessed directly from M_2 .
 - b) A request for a word not in M_1 typically causes a block containing the requested word to be transferred from M_2 to M_1 . After completion of the block transfer, the word can be accessed in M_1 .
If t_B denotes the block transfer time, we can write
$$t_{MISS} = t_B + t_{A1} \quad [\text{since } t_B \gg t_{A1}, t_{A2} \approx t_B]$$
Thus, $t_A = H \cdot t_{A1} + (1 - H) \cdot (t_B + t_{A1})$
 - c) If t_{HIT} denotes the time required to check whether there is a hit, we can write
$$t_{MISS} = t_{HIT} + t_B + t_{A1}$$

At the bottom of the slide, there is a navigation bar with icons for back, forward, search, and other course-related links. Logos for IIT Kharagpur, NPTEL, and NIT Meghalaya are also present.

The miss penalty t_{MISS} can be estimated in various ways. The simplest approach is to set t_{MISS} as t_{A2} , that is, when there is a miss the data is accessed directly from M_2 . So, a request for a word not in M_1 typically causes a block containing the requested word to

be transferred from M2 to M1. After completion of the block transfer the word can be accessed from M1.

When accessing a particular word, generally we do not transfer a single word rather we transfer a block of word. So, the block containing that particular word should be transferred to the cache, and then from the cache it will be transferred to the processor. So, this is what is said a request for the word not in M1 typically causes a block containing the requested word to be transferred from M2 to M1.

So, first the block is transferred from M2 to M1, and after completion of the block transfer the word can be accessed in M1. t_B denotes the block transfer time. The expression is shown.

(Refer Slide Time: 36:53)

- Efficiency:
 - Let $r = t_{A2} / t_{A1}$ denote the access time ratio of the two levels of memory.
 - We define the access efficiency as $e = t_{A1} / t_A$, which is the factor by which t_A differs from its minimum possible value.

$$\text{Efficiency } e = \frac{t_{A1}}{H \cdot t_{A1} + (1 - H) \cdot t_{A2}} = \frac{1}{H + (1 - H) \cdot r}$$

NPTEL ONLINE CERTIFICATION COURSES

NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, what is efficiency? Let us consider r as the access time ratio of the 2 levels. Efficiency is defined as shown in the expression.

(Refer Slide Time: 38:05)

- Speedup:
 - The speedup gained by using the memory hierarchy is defined as $S = \frac{t_{A2}}{t_A}$.
 - We can write:

$$S = \frac{t_{A2}}{H \cdot t_{A1} + (1 - H) \cdot t_{A2}} = \frac{1}{H / r + (1 - H)}$$

- The same result follows from Amadahl's law.

NPTEL ONLINE CERTIFICATION COURSES

NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, coming to speedup, the speedup gained by using memory hierarchy is time-old divided by time-new.

(Refer Slide Time: 39:28)

Some Common Terminologies Used

- Block: The smallest unit of information transferred between two levels.
- Hit Rate: The fraction of memory accesses found in the upper level.
- Hit Time: Time to access the upper level
 - Upper level access time + Time to determine hit/miss
- Miss: Data item needs to be retrieved from a block in the lower level.
- Miss Rate: The fraction of memory accesses not found in the upper level.
- Miss Penalty: Overhead whenever a miss occurs.
 - Time to replace a block in the upper level + Time to transfer the missed block

NPTEL ONLINE CERTIFICATION COURSES

NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, there are some common terminologies that we must know for rest of the lectures for this week. What is block? --- the smallest unit of information transferred between 2 levels. Hit Rate --- the fraction of memory accesses found in upper level. Hit Time --- the time to access the upper level. And so on.

So, these are some of the terminologies that we will be using throughout the week 6 lectures.

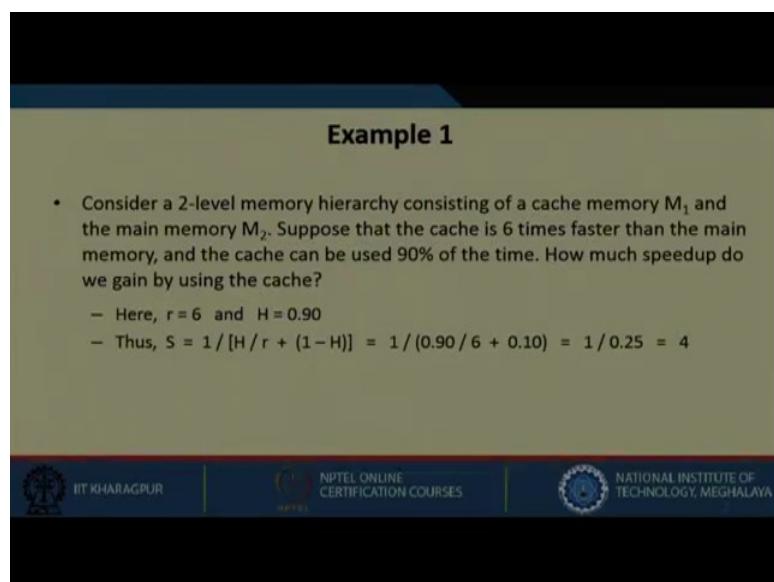
Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 29
Memory Hierarchy Design (PART II)

Welcome to next lecture 29. Here we will be discussing about memory hierarchy design in detail.

(Refer Slide Time: 00:28)



Example 1

- Consider a 2-level memory hierarchy consisting of a cache memory M_1 and the main memory M_2 . Suppose that the cache is 6 times faster than the main memory, and the cache can be used 90% of the time. How much speedup do we gain by using the cache?
 - Here, $r = 6$ and $H = 0.90$
 - Thus, $S = 1 / [H / r + (1 - H)] = 1 / (0.90 / 6 + 0.10) = 1 / 0.25 = 4$

The slide footer includes logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

Let us take this example here. Consider a two level memory hierarchy consisting of a cache memory M_1 and main memory M_2 . Generally, the level which is closest to the processor we call it cache memory and then the next level of memory can be different levels of cache memory; or we can also have a main memory.

Suppose that cache is 6 times faster than main memory and the cache can be used 90% of the time. How much speed up do we gain by using cache?

So, here r will be 6 and H will be 0.90. So, we simply put in into this formula and we get 4. So, we are getting a speed up of 4.

(Refer Slide Time: 02:25)

Example 2

- Consider a 2-level memory hierarchy with separate instruction and data caches in level 1, and main memory in level 2.

The diagram illustrates a 2-level memory hierarchy. At the top level, there is a CPU represented by a red rectangle. It has two output lines: one to an I-Cache (Instruction Cache) and one to a D-Cache (Data Cache), both represented by blue rectangles. Both the I-Cache and D-Cache have output lines leading to a large blue rectangle labeled "Main Memory".

This is pretty similar to what we discussed way back in Amadahl's law. Now I take another example where we consider two-level memory hierarchy with separate instruction and data caches in level 1, and main memory in level 2. So, here the scenario is we have I-cache, and we have D-cache. The instructions are brought and kept in instruction cache the data are brought and kept in data cache.

The advantage would be if you want to access data and instructions simultaneously, you can do that.

(Refer Slide Time: 03:13)

- The following parameters are given:
 - The clock cycle time is 2 ns.
 - The miss penalty is 15 clock cycles (for both read and write).
 - 1 % of instructions are not found in I-cache.
 - 8 % of data references are not found in D-cache.
 - 20 % of the total memory accesses are for data.
 - Cache access time (including hit detection) is 1 clock cycle.

The diagram shows a list of parameters for a memory access problem. The parameters are listed under a bullet point:

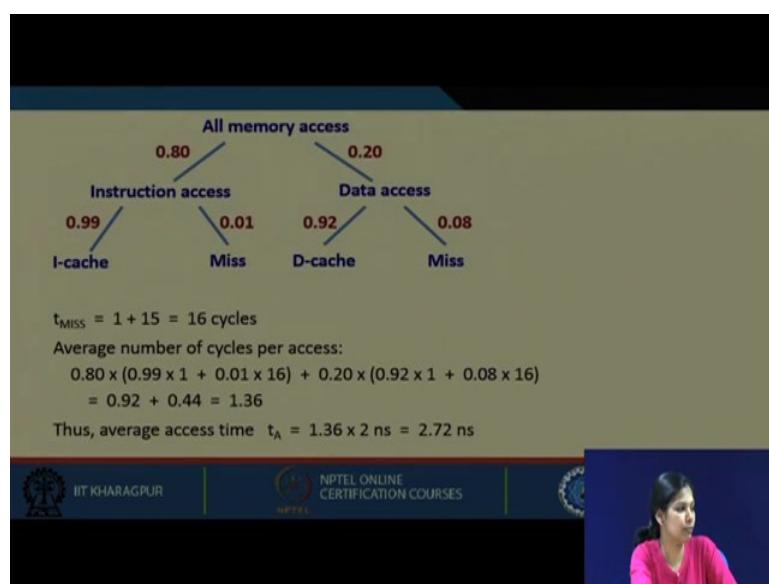
- The following parameters are given:
 - The clock cycle time is 2 ns.
 - The miss penalty is 15 clock cycles (for both read and write).
 - 1 % of instructions are not found in I-cache.
 - 8 % of data references are not found in D-cache.
 - 20 % of the total memory accesses are for data.
 - Cache access time (including hit detection) is 1 clock cycle.

The following parameters are given.

For miss penalty total 15 clock cycles are required. Out of which it says that 1% of instructions are not found in I-cache; and 8% of data references are not found in D-cache. 20% of the total memory access are for data, and cache access time including hit detection is 1 clock cycle.

So, these are the information that are provided and we will be seeing how we can find the average access time.

(Refer Slide Time: 04:32)



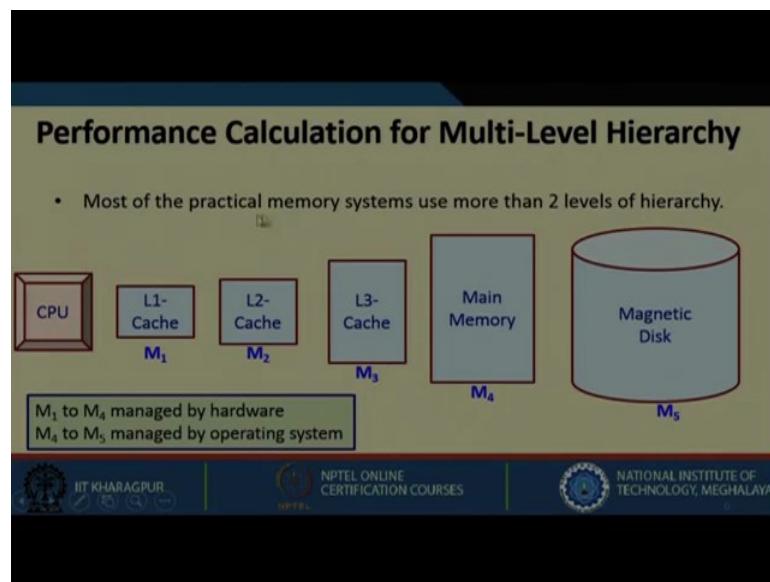
So, let us see, out of total memory access 80% are for instruction and 20% are for data. Out of this 80% of instruction, I-cache can be used 99%, and miss rate is 0.01 where the data is not found in I-cache.

Similarly, for data access 8% there is miss. So, 0.92 will be hit.

The average access time is calculated as shown.

So, in this example we have seen what happens when there is a miss, what happens when data is found in cache, and now the cache is divided into two different levels, one is I-cache and another is D-cache.

(Refer Slide Time: 07:55)



So, when you have two different caches, the calculation will be made in this fashion. Let us see performance calculation for multi level hierarchy.

In general, we have multi-level hierarchy rather than two-level hierarchy. In two-level hierarchy, we have only two levels M₁ and M₂, but in reality we have multiple levels. For multiple levels whatever we have done for two levels can be extended to multi levels.

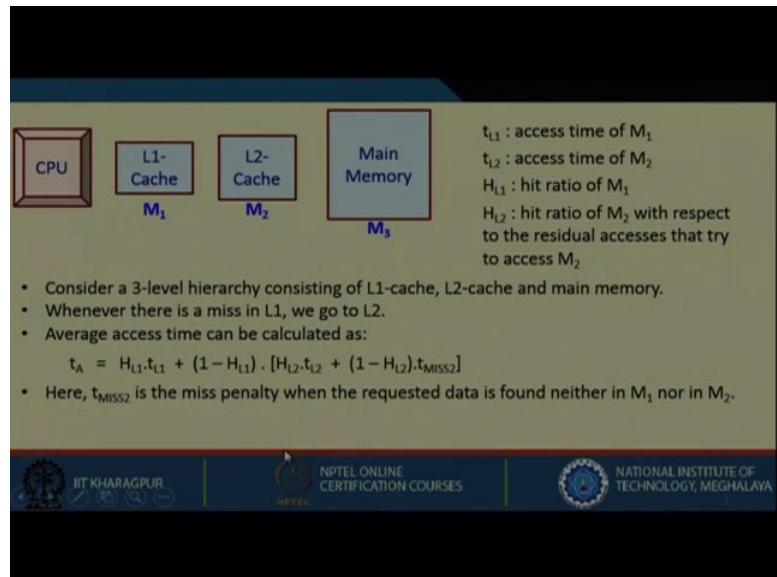
So, if you see this particular diagram which is considered as the most practical memory system that; obviously, we use more than two levels of hierarchy. The first level is M₁ which is L1 cache, the next level is M₂ which is using L2 cache, the next level is M₃ which is L3 cache, then M₄ is the main memory, and finally M₅ is the magnetic disk.

So, how do we calculate? As we can see the access time of this will be much faster than this, then this, then this, and data from main memory is brought through L3 to L2 and then to L1, and when it is in L1 because of locality of reference, we will get the data whenever CPU is asking for it. But initially remember there will be a miss because there will be no data in the cache.

Whenever the data is loaded in main memory, whenever it is required it has to be first brought from main memory through different levels to L1 cache, and finally it goes to CPU. So, for the first time there will be miss always, but for consecutive times there will

be hit because of locality of reference. So, M1 to M4 is managed by hardware; this is very important. But M4 to M5 is handled by operating system.

(Refer Slide Time: 10:31)

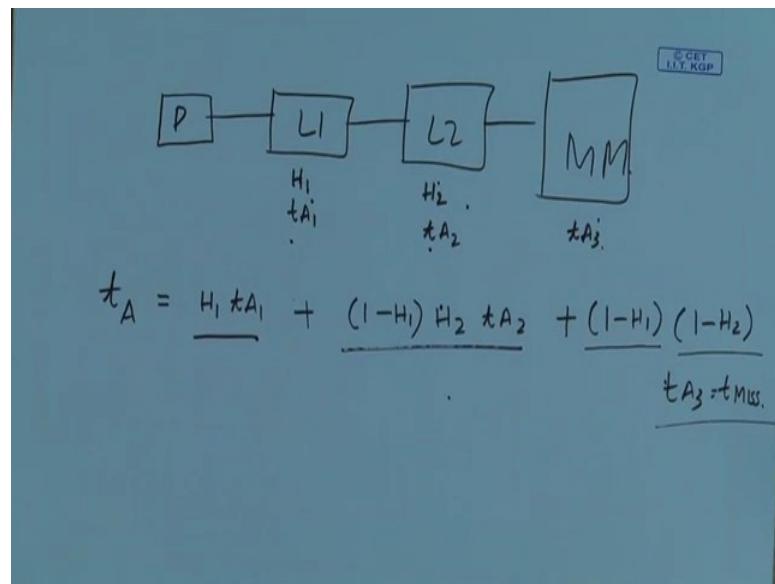


Now, let us see this we have one more level; three levels of memory hierarchy, where t_{l1} is the access time of M₁, t_{l2} is access time of M₂, H_{l1} is the hit ratio of M₁; that means, the percentage time the data or instruction is found in L1 cache. H_{l2} is the percentage time the data or instruction is found in L2 cache, which is not found in L1 cache.

So, hit ratio of M₂ is with respect to the residual access that try to access M₂. Consider a three level hierarchy that consists of L1 cache, L2 cache and main memory. Whenever there is a miss in L1 we go to L2. Obviously, if we do not find the data or instruction in L1 cache we move to L2 cache. What will be the average access time now? Earlier we have done average access time calculation taking into consideration that CPU first access L1 cache.

The access time t_A is given by this expression.

(Refer Slide Time: 13:03)



So, what will be the average access time? The expression is shown.

This is the equation for three levels of memory hierarchy.

(Refer Slide Time: 16:14)

Implications of a Memory Hierarchy to the CPU

- Processors designed without memory hierarchy are simpler because all memory accesses take the same amount of time.
 - Misses in a memory hierarchy implies variable memory access times as seen by the CPU.
- Some mechanism is required to determine whether or not the requested information is present in the top level of the memory hierarchy.
 - Check happens on every memory access and affects hit time.
 - Implemented in hardware to provide acceptable performance.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, what is the implication of the memory hierarchy to the CPU? We can say that the processors designed without memory hierarchy are simple because all memory access takes same amount of time. Misses in a memory hierarchy implies variable memory access time as seen by the CPU, because access time of L1 cache is different from access time of L2 cache, and it is different from main memory.

Some mechanism is required to determine whether or not the requested information is present in top level of memory hierarchy. So, how can we check this? Check happens on every memory access and affects the hit time and implemented in hardware to provide acceptable performance. So, what we are doing.

We are first checking into the top level L1. Is there any mechanism that we in advance know that this particular word is there or not, or the checking can it be made little fast so that hit time can made little faster. So, there are some kinds of things that can be done, check happens on every memory access and that is why it affects the hit times. If we can have some kind of hardware implementation for this matching, it can be performed in reasonable time.

(Refer Slide Time: 18:33)

The slide has a dark blue header and footer. The main content area is light grey. It contains a bulleted list of points related to memory hierarchy and transfer mechanisms.

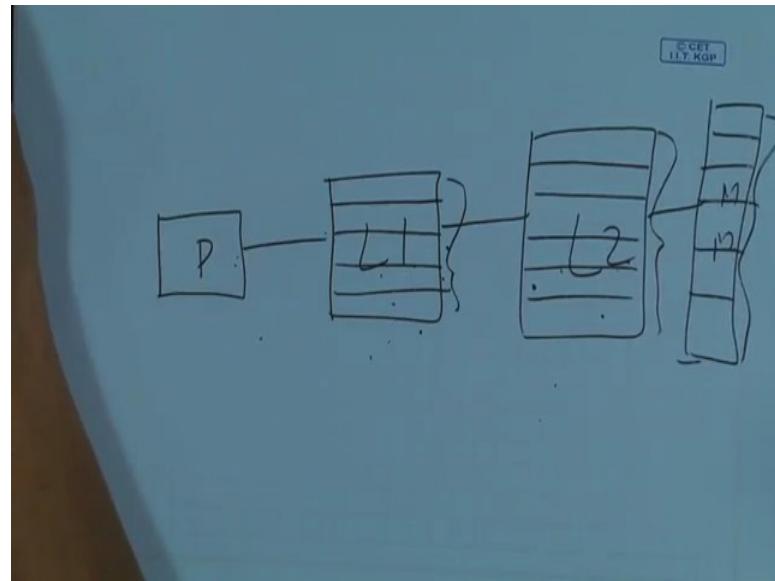
- Some mechanism is required to transfer blocks between consecutive levels.
 - If the block transfer requires 10's of clock cycles (like in cache / main memory hierarchy), it is controlled by hardware.
 - If the block transfer requires 1000's of clock cycles (like in main memory / secondary memory hierarchy), it can be controlled by software.
- Four main questions:
 1. *Block Placement*: Where to place a block in the upper level?
 2. *Block Identification*: How is a block found if present in the upper level?
 3. *Block Replacement*: Which block is to be replaced on a miss?
 4. *Write Strategy*: What happens on a write?

At the bottom, there are three logos with their respective names: IIT Kharagpur, NPTEL, and National Institute of Technology, Meghalaya.

Some mechanism is required to transfer blocks between consecutive levels; as we know that if there is a miss in both L1 and L2, you have to bring a block from main memory to L2 level and then from L2 to L1 level. So, whenever there is a miss in some level you have to transfer a block. If the block transfer requires 10s of clock cycles like cache or main memory, it is controlled by hardware; but if the block transfer requires 1000s of clock cycles like in main memory, secondary memory etc., it can be controlled by software.

At this point of time few questions arise. One is block placement. Let us say we have different levels of memory.

(Refer Slide Time: 19:45)



So, again we have processor, we have L1 cache, we have L2 cache, and we have main memory. Now it has got some blocks.

Now, the point is where we will make the placement of block. From here we will bring the block and place it here. It needs to be understood next is block identification. What is block identification? How is a block found if present in a upper level? So, we must be matching something when the CPU generates a logical address and it says that this is the logical address I am looking for this particular word, then something much must be matched to know if that block is present in the upper level or not. So, block identification is also important.

Similarly, block replacement ... which block is to be replaced on a miss. What do you mean by that? Now you see in this particular diagram we have more number of blocks, here we have less number of blocks, and we are saying that every time the processor accesses the L1 cache it will be the fastest. So, all the programs or data are brought into L1 cache and then it is accessed.

Now, the size of this is small. So, at a time limited number of blocks can be present here. If you want to run your program that requires more number of blocks, in that case what you have to do? You have to replace some of the existing blocks from here and bring a new block. So, for that some block replacement strategies must be there; that means,

some blocks can be removed from this particular level and then some blocks from this level can be brought in. So, this is called block replacement.

Write strategy means what happens on a write whenever we are performing a write operation. What strategies must be used for this write operation?

(Refer Slide Time: 22:56)

Common Memory Hierarchies

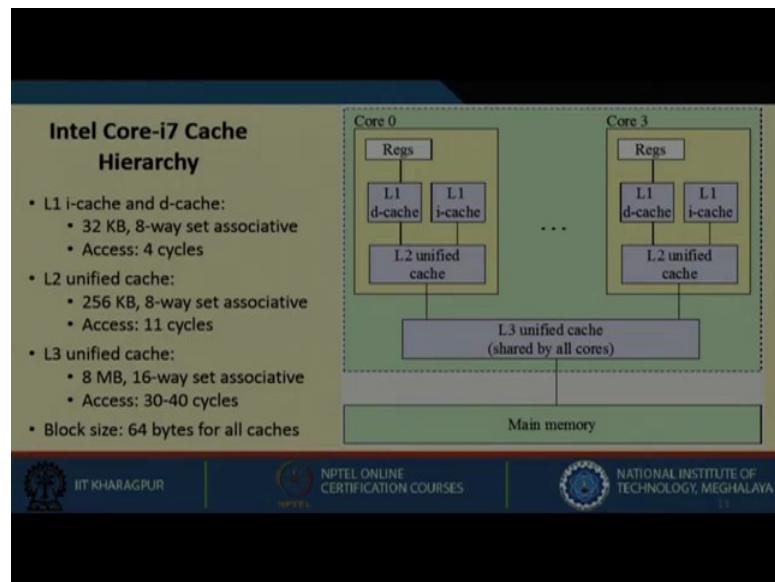
- In a typical computer system, the memory system is managed as two different hierarchies.
 - The Cache / Main Memory hierarchy, which consists of 2 to 4 levels and is managed by hardware.
 - Main objective: provide fast average memory access.
 - The Main Memory / Secondary Memory hierarchy, which consists of 2 levels and is managed by software (operating system).
 - Main objective: provide large memory space for users (virtual memory).

In a typical computer system the memory system is managed as two different hierarchies, cache memory / main memory hierarchy which consists of 2 to 4 levels and is managed by hardware. The main objective of this particular hierarchy is to provide fast average access time for instructions and data.

Then we have main memory / secondary memory hierarchy that consists of two levels and is managed by software, that is by the operating system which is the interface between the user and the hardware part of our computer. The main objective is to provide large memory space for the user, called virtual memory.

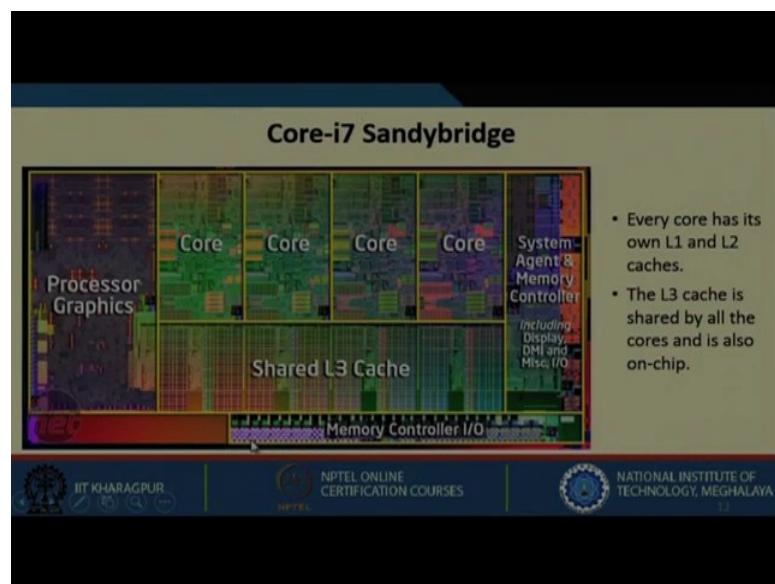
So, basically this virtual memory is a concept where it says that you have a large memory space at your disposal, but actually you are having the space as main memory only. Programs and data are brought from secondary storage to main memory as and when it is required. This is called virtual memory, which will not be taken up in this particular course in detail.

(Refer Slide Time: 24:51)



This is the typical memory hierarchy of Intel core i. There are four cores; core 0, core 1, core 2, and core 3. Inside each of the cores you have registers, you have separate L1 cache for data and instruction, we have an unified L2 cache.

(Refer Slide Time: 27:39)



Next is core-i7 SandyBridge. Here you have 4 cores, every core has its own L1 and L2 cache, and L3 is shared by all the cores and is also on chip. This is the processor, graphics processor, the memory controller, etc.

So, we come to the end of lecture 29 where we discussed about memory hierarchy in details. We have seen that how memory hierarchy actually affects the overall performance.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 30
Cache Memory (Part I)

Welcome to the next lecture on Cache Memory. I have already discussed and made the ground for cache memory. Now, I will be discussing in detail about the read and write strategies, the block replacement strategies, and of course; the mapping techniques that are used.

We know that cache memory is a fast memory that is in between processor and main memory. Frequently used data or instructions are brought into this memory and they are brought to the processor in turn for execution. For faster execution frequently used instruction and data brought here. So, the next time when the processor needs that data or instruction, it is getting from the cache.

So the benefit we are getting is most of the time we are accessing the data from the cache rather than from main memory, but for the first time there will be only miss and the data must be brought from lower level of the memories that is main memory or L2 cache or L3 cache into your L1 cache memory.

(Refer Slide Time: 01:45)

Introduction

- Let us consider a single-level cache, and that part of the memory hierarchy consisting of cache memory and main memory.

JIT Kharagpur NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us consider a single level cache, that is, the memory hierarchy consists of the cache memory and main memory. So, this is the CPU, this is the main memory and this is the cache memory.

(Refer Slide Time: 01:58)

- Cache memory is logically divided into *blocks* or *lines*, where every block (line) typically contains 8 to 256 bytes.
- When the CPU wants to access a word in memory, a special hardware first checks whether it is present in cache memory.
 - If so (called *cache hit*), the word is directly accessed from the cache memory.
 - If not, the block containing the requested word is brought from main memory to cache.
 - For writes, sometimes the CPU can also directly write to main memory.
- Objective is to keep the commonly used blocks in the cache memory.
 - Will result in significantly improved performance due to the property of locality of reference.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Cache memory is logically divided into blocks or lines, where every block or line typically contains 8 to 256 bytes. When the CPU wants to access a word in memory, a special hardware first checks whether it is present in cache memory. If so we call it a cache hit and the word is directly accessed from the cache memory. If not, the block containing the requested word is brought from main memory to cache memory.

For writes, sometimes the CPU can also directly write to the main memory, or it can be written back into the cache, and later it is updated into the main memory. So, cache memory is divided into blocks and whenever the CPU generates some address, we first check whether the requested word is present in the cache or not. If it is present in the cache that particular word is sent to the processor. If that particular word is not present in the cache then we bring that particular word from main memory into cache memory, and then it is transferred to the processor.

So whenever a cache hit occurs; that means, the word is in cache memory, we can directly access the word from there. If it is a miss, then the requested word is brought from main memory to cache memory; similarly for write also. We will see in detail that in write what exactly happens. Objective is to keep the commonly used blocks in the

cache memory. It will result in significantly improved performance due to property of locality of reference.

(Refer Slide Time: 04:43)

Q1. Where can a block be placed in the cache?

- This is determined by some mapping algorithms.
 - Specifies which main memory blocks can reside in which cache memory blocks.
 - At any given time, only a small subset of the main memory blocks can be held in main memory.
- Three common block mapping techniques are used:
 - a) Direct Mapping
 - b) Associative Mapping
 - c) (N-way) Set Associative Mapping
- The algorithms shall be explained with the help of an example.

IT-KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

You remember we said that we will be answering 4 questions. The first one is, where can a block be placed in the cache? Firstly, where my block will be placed in the cache is determined by mapping algorithms. This specifies which main memory block can reside in which cache memory block. Which block of the main memory will be in which block of the cache must be determined using some mapping technique.

In this context we have 3 mapping techniques: direct mapping, associative mapping, and set associative mapping. One thing is clear, that cache is small so we cannot bring everything into cache. Some blocks of main memory at a time can be brought into cache. So, some blocks that are required can be brought into the cache, and can be used by the processor; and when some other blocks are required then some of the blocks that are present in the cache must be replaced such that we can bring new blocks from the cache. Again for that we have separate strategies we will look into that later.

(Refer Slide Time: 06:38)

Example: A 2-level memory hierarchy

- Consider a 2-level cache memory / main memory hierarchy.
 - The cache memory consists of 256 blocks (lines) of 32 words each.
 - Total cache size is 8192 (8K) words.
 - Main memory is addressable by a 24-bit address.
 - Total size of the main memory is $2^{24} = 16\text{ M}$ words.
 - Number of 32-word blocks in main memory = $16\text{ M} / 32 = 512\text{K}$

The slide footer includes logos for IIT Kharagpur, NPTEL, and NIT Meghalaya.

Now let us see the algorithms one by one. Consider a 2-level memory hierarchy having cache memory and main memory; with this example we will be taking into consideration all the mapping algorithms. The cache memory consists of 256 blocks or lines of 32 words. So, each block is having 32 words. Total cache sizes 8192, that is 8 Kwords. And how is the main memory organized? Main memory is addressable by a 24 bit address.

So, the main memory is addressable by 24 bit address; it is having 16 M words. So, 16 M words; that means, the total number of blocks in main memory will be total 16 M divided by 32. So, we have 512 K blocks in main memory. So, what is important here is to know how many blocks in main memory is there and how many blocks in cache memory is there. So, we have a total of 512 K blocks in main memory with this organization, and we have a total of 256 blocks in cache memory.

(Refer Slide Time: 08:20)

(a) Direct Mapping

- Each main memory block can be placed in only one block in the cache.
- The mapping function is:
$$\text{Cache Block} = (\text{Main Memory Block}) \% (\text{Number of cache blocks})$$
- For the example,
$$\text{Cache Block} = (\text{Main Memory Block}) \% 256$$
- Some example mappings:
$$0 \rightarrow 0, 1 \rightarrow 1, 255 \rightarrow 255, 256 \rightarrow 0, 257 \rightarrow 1, 512 \rightarrow 0, 513 \rightarrow 1, \text{etc.}$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

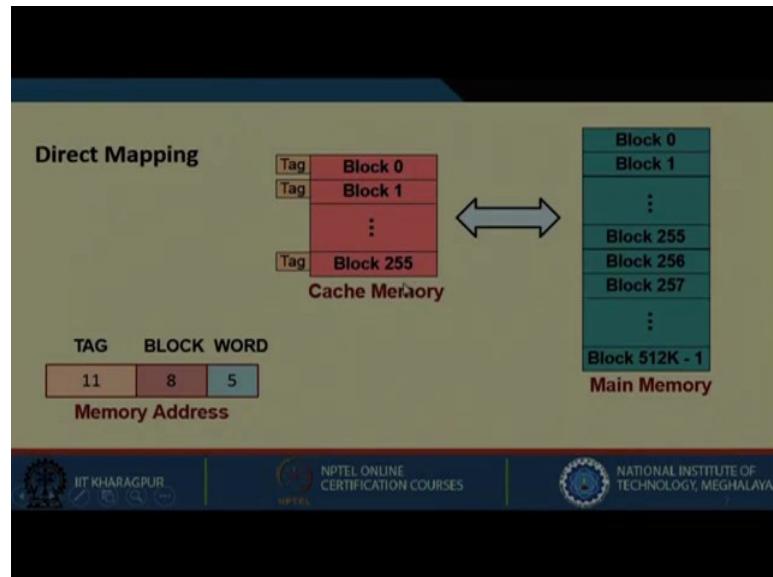
Now let us see direct mapping. What happens in direct mapping? The name suggests it is very direct, so let us see how it is. Each main memory block can be placed in only one block in the cache depending on this particular function. What is that function? Which main memory block will be placed in which cache block is determined by this particular formula. We get a main memory block; we make a modulus with number of cache blocks. So, we already know total number of cache blocks is 256, and any main memory block modulo this will give me the particular cache block.

So any main memory block can be placed in some particular cache memory block using this particular formula. What is that formula? Main memory block % total number of cache blocks. So, with the example if we take this particular formula into consideration, 0 will be mapped to 0 block of cache; how? 0 modulo 256 will be 0. So, 1 modulo 256 will be 1. Similarly 255 modulo 256 will be 255.

Similarly 256 modulo 256 is 0; so it will be again placed in block 0, 257 modulo 256 will be 1 so this will be placed in 1. So, the idea is now you see that block 0 of main memory block 256 of main memory both will be placed in block 0 of cache memory. So, there are many blocks of main memory that can be mapped into the same block of cache memory. This is direct mapping, where we have given a formula; through that formula you are mapping any block of the main memory into some block of the cache memory.

So at a time we cannot have block 0 and block 256 at the same time in the cache. This is a problem. If in a program you require both block 0 as well as block 256; then this can be a problem.

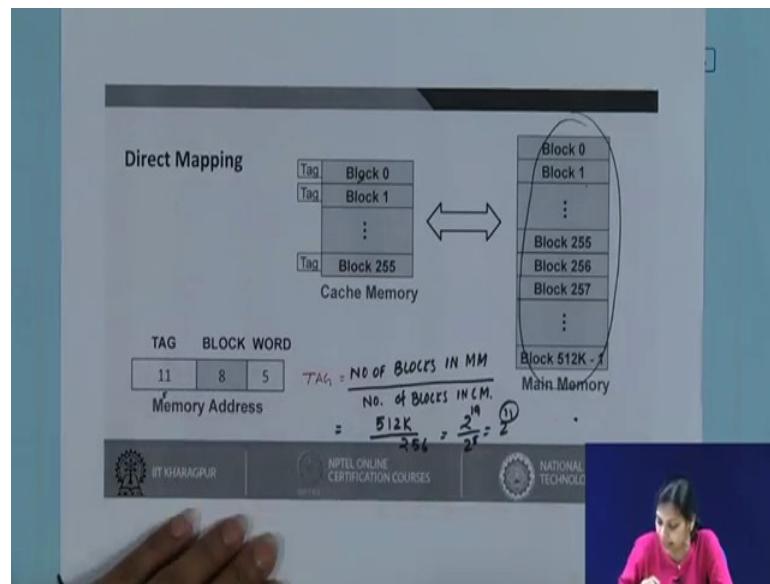
(Refer Slide Time: 12:44)



Now in direct mapping the memory address is divided into 3 parts basically we call it TAG, BLOCK and WORD. Now each block is having 32 words. So, how many bits will be required to access a word within that block? We will be requiring 5 bits, because we have 32 words. So, 5 bits will give you any one of the word within a block.

So 5 least significant bits will be required to access a WORD within this block. Now we need to know how many blocks are there. The total number of blocks here is 256. Now if total number of blocks is 256 then we require 8 bits to represent a block. So, this BLOCK will have 8 bits.

(Refer Slide Time: 14:31)



Now, finally, is the TAG, which will tell basically that which block of main memory is mapped to a particular block of cache memory.

So let us see here what happens is that as we can see that there are many blocks in main memory and there are few blocks in cache memory how we can find the number of bits in the TAG.

(Refer Slide Time: 15:16)

$$\begin{aligned} \text{TAG} &= \frac{\text{No. of Blocks in MM}}{\text{No. of Blocks in CM.}} \\ &= \frac{512K}{256} = \frac{2^{19}}{2^8} = 2^{11} \end{aligned}$$

The TAG field for direct mapping will be number of blocks in main memory divided by number of blocks in cache memory. Here it will require 11 bits.

So, first we match the particular TAG; if that is present then we get that word and then it is transferred to processor. If it is a miss then, first it is brought from main memory to this cache memory, and then it is transferred to the processor.

(Refer Slide Time: 17:10)

- Block replacement algorithm is trivial, as there is no choice.
- More than one MM block is mapped onto the same cache block.
 - May lead to contention even if the cache is not full.
 - New block will replace the old block.
 - May lead to poor performance if both the blocks are frequently used.
- The MM address is divided into three fields: TAG, BLOCK and WORD.
 - When a new block is loaded into the cache, the 8-bit BLOCK field determines the cache block where it is to be stored.
 - The high-order 11 bits are stored in a TAG register associated with the cache block.
 - When accessing a memory word, the corresponding TAG fields are compared.
 - Match implies HIT.

So the block replacement algorithm is trivial as there is no choice. But more than one main memory block are mapped onto the same cache block. This may lead to contention even if the cache is not full; that means, even if there is space in the cache, because of that mapping formula restriction we cannot place any block of the main memory anywhere.

So, we cannot have any other option new block will replace the old block. So, old block has to be replaced when a new block is brought in; may lead to poor performance if both blocks are frequently used. We are trying to say that in a program we require both block 0 and block 256 simultaneously in a loop. Block 0 and block 256 both cannot stay at the same time.. So, when one is staying then the other has to be removed and then again the other has to be brought in the other has to be removed.

So if the blocks both the blocks are required frequently then this kind of mapping will give poor performance. The main memory address is divided into 3 fields as we already have seen, TAG, BLOCK and WORD. So, when a new block is loaded into cache, the 8-bit BLOCK field determines the cache block, where it is to be stored with that formula, the high order 11 bits are stored in the TAG register associated with the cache block. So,

when accessing a memory word, the corresponding TAG fields are compared; match implies a hit.

So basically when we say that whether the word is found in cache or not we actually match that TAG. So, if the TAG matches then we say that that particular word is present.

(Refer Slide Time: 19:38)

(b) Associative Mapping

- Here, a MM block can potentially reside in any cache block position.
- The memory address is divided into two fields: TAG and WORD.
 - When a block is loaded into the cache from MM, the higher order 19 bits of the address are stored into the TAG register corresponding to the cache block.
 - When accessing memory, the 19-bit TAG field of the address is compared with *all the TAG registers* corresponding to all the cache blocks.
- Requires associative memory for storing the TAG values.
 - High cost / lack of scalability.
- Because of complete freedom in block positioning, a wide range of replacement algorithms is possible.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Next, let us move on with associative mapping. Here a main memory block can potentially reside in any cache block position. In case of direct mapping, we have seen the problem, even if there is space we cannot keep a block. We cannot bring any block from main memory because of that condition. So, here in this associative mapping this condition is relaxed.

Here any block of main memory can be brought into any block of the cache. This of course, will make the utilization much more. The memory address is divided into 2 fields only, we have TAG and WORD, because there is no concept of block. Any block can be brought in here; when a block is loaded into the cache from main memory the higher order 19 bits of the address are stored into the TAG register corresponding to the cache block.

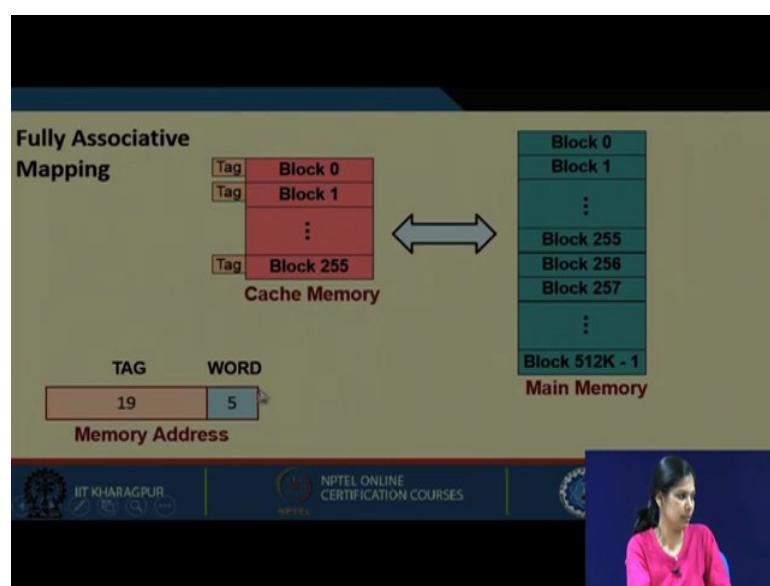
When accessing memory, the 19 bit TAG field of the address is compared with all the TAG registers corresponding to all the cache blocks. So, this is an disadvantage again.

When the processor checks the TAG field, it has to check all the TAG registers whether that particular tag is present corresponding to any of the blocks.

So there we have 256 blocks. So, TAG associated with 256 blocks must be checked to know whether it is a hit or not. So, when accessing memory the 19 bit TAG field of the address is compared with all the TAG registers corresponding to all the cache blocks. This requires associative memory for storing the TAG values. Associative memory is much more costlier; results in higher cost and lack of scalability. We cannot have very large associative memory in place. Because of complete freedom in block positioning a wide range of replacement algorithm is possible.

So this particular associative mapping we can clearly see is much more efficient because the space will be utilized to the maximum; any main memory block can be kept in any cache block. So, the entire space of the cache is utilized very nicely; we have to replace a particular block when the cache is full, if the cache is having any empty space then any block can be brought in, but with that what we are adding up is the checking. We need to have an associative memory for storing the TAG values.

(Refer Slide Time: 23:47)



So in fully associative mapping we have total of 256 blocks in cache memory. Any 256 blocks from main memory can be brought in here, and which block of main memory is here is determined by this 19 bit TAG field.

(Refer Slide Time: 24:25)

(c) N-way Set Associative Mapping

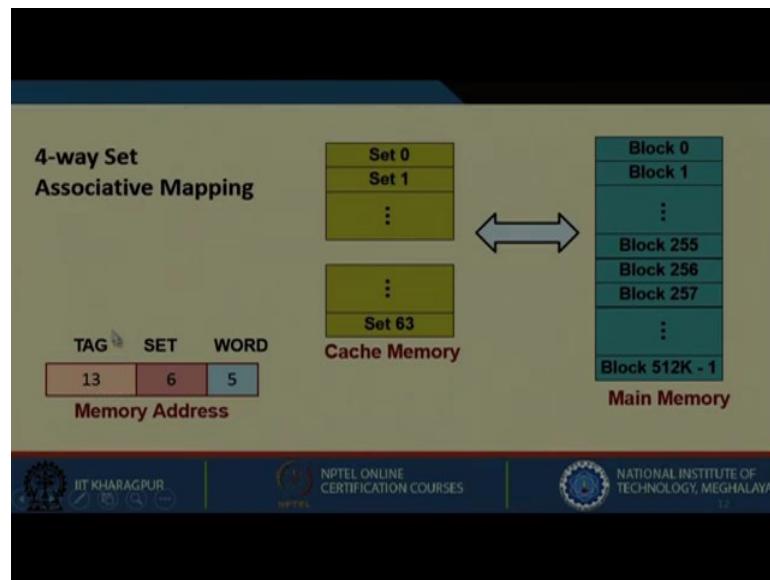
- A group of N consecutive blocks in the cache is called a set.
- This algorithm is a balance of direct mapping and associative mapping.
 - Like direct mapping, a MM block is mapped to a set.
- Set Number = (MM Block Number) % (Number of Sets in Cache)
 - The block can be placed anywhere within the set (there are N choices)
- The value of N is a design parameter:
 - N = 1 :: same as direct mapping.
 - N = number of cache blocks :: same as associative mapping.
 - Typical values of N used in practice are: 2, 4 or 8.

Let us now come to another mapping algorithm that is called N-way set associative mapping. We have seen that in direct mapping there is some advantage and disadvantage, and in associative mapping of course, we have advantage and as well as some disadvantage. So, what we are trying to do let us take some of the properties of direct mapping and some of the properties of associative mapping. We combine these two and we have set associative mapping. What this algorithm says is a group of N consecutive blocks in cache is called a set.

So words are combined to form a block, and blocks are combined to form a set. This algorithm is a balance of both direct and associative mapping. Like in direct mapping, a main memory block is mapped to a particular set. So, the set number of the cache can be determined by main memory block number modulo number of sets in the cache; that means, in direct mapping we have main memory block number modulo number of blocks in cache, but here we have number of sets in cache. So, the block can be placed anywhere within the set that is there are N choices for it.

The value of N is a design parameter. So, when N=1, it is same as direct mapping, and when N is the total number of cache block then it is same as associative mapping. But the typical value of N that is used in practice can be 2, 4, 8, or 16.

(Refer Slide Time: 27:28)



Let us consider a 4 way set associative mapping where we have 64 sets and we have 512K memory blocks.

(Refer Slide Time: 28:40)

A handwritten derivation of the TAG formula for 4-way set associative mapping. The formula is:

$$TAG = \frac{\text{TOTAL NO. OF BLOCKS IN MM}}{\text{TOTAL NO. OF SETS IN CM.}}$$

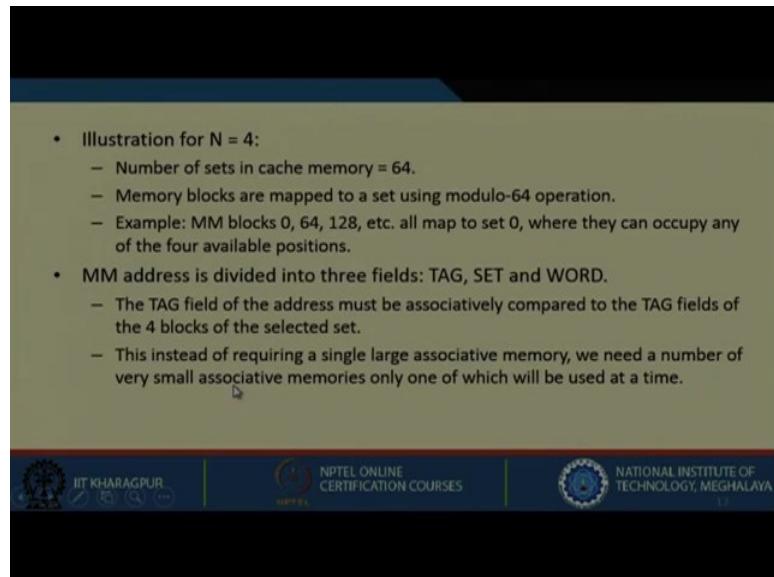
Given:

$$= \frac{512K}{64} = \frac{2^{19}}{2^6} = 2^{13}$$

We will determine the TAG here in the same way for direct mapping; instead of total number of blocks here we will have total number of sets in cache memory. So, in the same way like direct mapping here it was having total number of blocks we are having total number of sets here. So, let us say total number of blocks is 512 K and total number

of set is 64. So, it is 2 to the power 19 divided by 2 to the power 6 coming to 2 to the power 13.

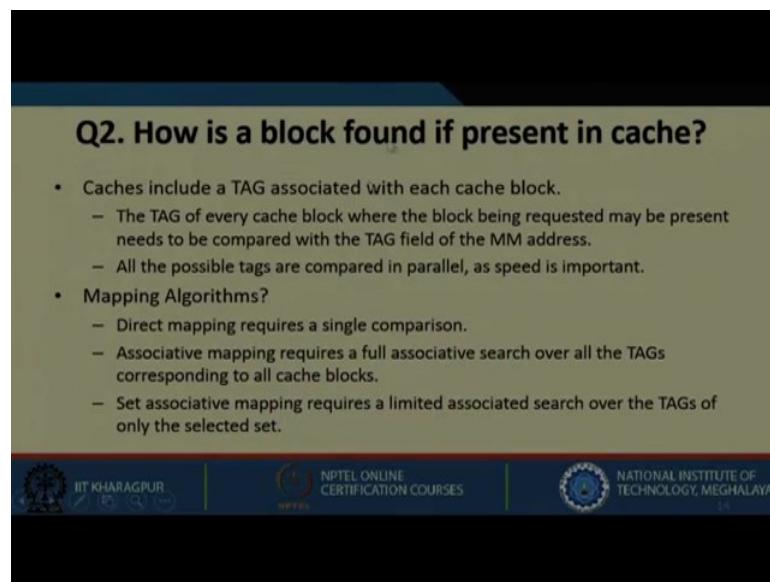
(Refer Slide Time: 29:48)



So if you see we have 13 bit for TAG, SET is 6 and WORD is 5. So, let us illustrate for N equals to 4. For 4-way associative set associative mapping, number of sets in cache memory is 64. Memory blocks are mapped to a set using modulo 64 operation. So, main memory blocks 0, 64, 128, etc. all map to set 0 where they can occupy any of the 4 available positions.

This instead of requiring a single large associative memory we need a number of very small associative memories only one of which will be used at a time.

(Refer Slide Time: 32:00)



Q2. How is a block found if present in cache?

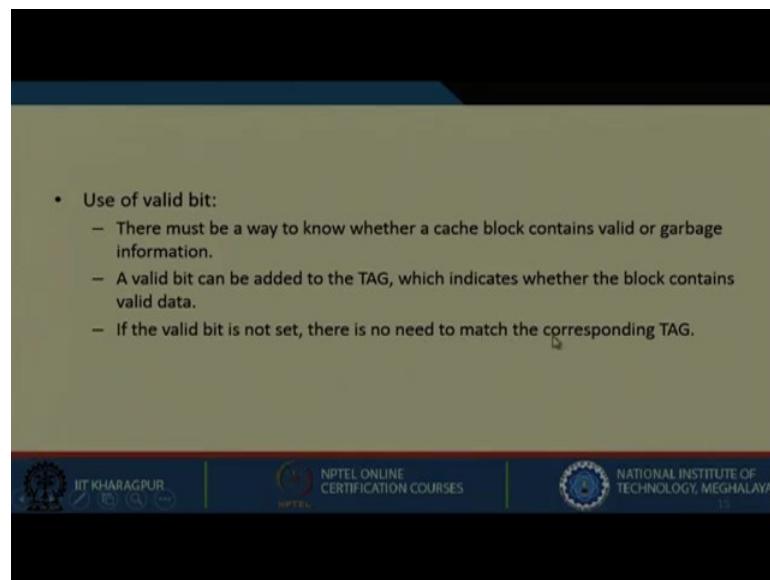
- Caches include a TAG associated with each cache block.
 - The TAG of every cache block where the block being requested may be present needs to be compared with the TAG field of the MM address.
 - All the possible tags are compared in parallel, as speed is important.
- Mapping Algorithms?
 - Direct mapping requires a single comparison.
 - Associative mapping requires a full associative search over all the TAGs corresponding to all cache blocks.
 - Set associative mapping requires a limited associative search over the TAGs of only the selected set.

NPTEL ONLINE CERTIFICATION COURSES

NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So cache include a TAG associated with each cache block. The TAG of every cache block needs to be compared with the TAG field of the main memory address. So, the TAG field of the main memory address is compared with the TAG of every cache block; all the possible tags are compared in parallel as speed is very important.

(Refer Slide Time: 33:18)



- Use of valid bit:
 - There must be a way to know whether a cache block contains valid or garbage information.
 - A valid bit can be added to the TAG, which indicates whether the block contains valid data.
 - If the valid bit is not set, there is no need to match the corresponding TAG.

NPTEL ONLINE CERTIFICATION COURSES

NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

(Refer Slide Time: 34:41)

Q3. Which block should be replaced on a cache miss?

- With fully associative or set associative mapping, there can be several blocks to choose from for replacement when a miss occurs.
- Two primary strategies are used:
 - Random:** The candidate block is selected randomly for replacement. This simple strategy tends to spread allocation uniformly.
 - Least Recently Used (LRU):** The block replaced is the one that has not been used for the longest period of time.
 - Makes use of a corollary of temporal locality:
"If recently used blocks are likely to be used again, then the best candidate for replacement is the least recently used block"

Logos for IIT Kharagpur, NPTEL, and National Institute of Technology Meghalaya are at the bottom.

The next question arises which block should be replaced on a cache miss. With fully associative or set associative mapping, there can be several blocks to choose for the replacement when a miss occurs; that means, within an associative or if you think of set associative where you have some blocks.

Now for that two primary strategies are used, one is random. The candidate block is selected randomly for replacement. This simple strategy tends to spread allocation uniformly. Another is least recently used. Here what it says that the block replaced is the one that has not been used for longest period of time.

Say we have 4 blocks associated with a particular set, and we want to replace a particular block. This least recently used says that a block which is in the cache for longest period of the time, but it has not been used; we will replace that particular block.

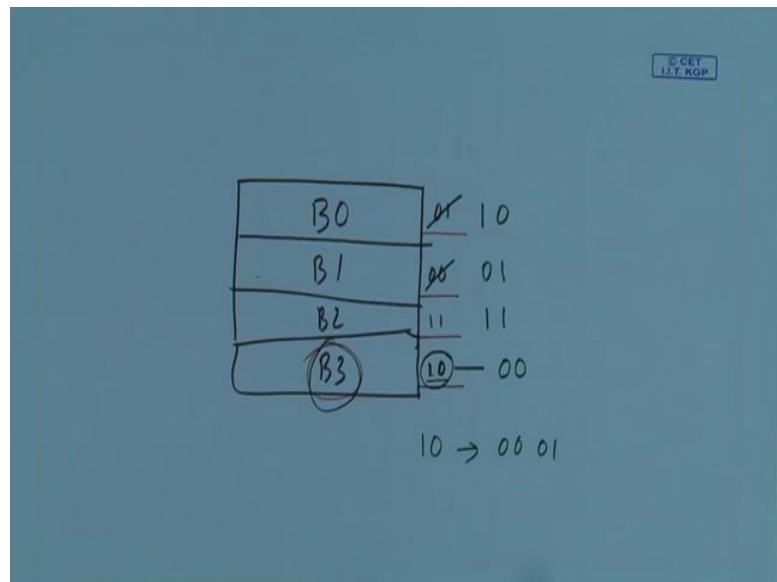
(Refer Slide Time: 37:08)

- To implement the LRU algorithm, the cache controller must track the LRU block as the computation proceeds.
- Example: Consider a 4-way set associative cache.
 - For tracking the LRU block within a set, we use a 2-bit counter with every block.
 - When hit occurs:
 - Counter of the referenced block is reset to 0.
 - Counters with values originally lower than the referenced one are incremented by 1, and all others remain unchanged.
 - When miss occurs:
 - If the set is not full, the counter associated with the new block loaded is set to 0, and all other counters are incremented by 1.
 - If the set is full, the block with counter value 3 is removed, the new block put in its place, and the counter set to 0. The other three counters are incremented by 1.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

To implement LRU algorithm the cache controller must track the LRU block as the computation proceeds. We need to keep track of the LRU block. A 2-bit counter can be used with every block for this purpose, as explained.

(Refer Slide Time: 38:49)



So, let us say we have we have 4 blocks this is block 0, block 1, block 2, and block 3; and the values are like this; this is 0 1, this is 0 0, this is 1 1, and this is 1 0.

What the algorithm say the counter of the reference block is reset to 0. The counters with values originally lower than the referenced one are incremented by 1, and others remain

unchanged. So, this is the current value of the counter. So, the current value of the counter is 0 1, this is 0 0, this is 1 1, and this is 1 0; now let us see I am referencing B3 block again. So, the counter associated with this will become 0 0 and all the counter value less than this will be incremented by 1 and all other counter values will remain unchanged. This process continues.

(Refer Slide Time: 42:04)

B0	10	11	00
B1	00	01	10
B2	01	10	11
B3	00	00	01

(Refer Slide Time: 44:34)

- It may be verified that the counter values of occupied blocks are all distinct.
- An example:

x	Block 0
x	Block 1
x	Block 2
x	Block 3
Initial	
Miss: Block 2	
1	Block 0
2	Block 1
0	Block 2
3	Block 3
Miss: Block 0	
Miss: Block 3	
1	Block 0
2	Block 1
0	Block 2
3	Block 3
Miss: Block 1	
Hit: Block 0	
Hit: Block 3	
Hit: Block 2	
Hit: Block 0	
Miss: Block 1	

x	Block 0
x	Block 1
0	Block 2
x	Block 3
Initial	
Miss: Block 2	
1	Block 0
2	Block 1
0	Block 2
3	Block 3
Hit: Block 3	
Hit: Block 2	
Hit: Block 0	
Miss: Block 1	

JIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, we will move on with an example with 4 blocks. Initially, nothing is there. Now say block 2 is referenced. So, the counter value associated it is a miss. Initially it will be

a miss because this is all empty. Now in the block 2 a value is brought in and the reference block value become 0, next block 0 is brought in the value associated with the new block become 0 and the previous block is incremented by 1; it has become 1, next block 3 this is also miss.

So the counter value associated with this will become 0, and all others will get incremented by 1; similarly block 1 is accessed and these are the values associated with this will become 0 and all others will get incremented by 1. 1 has become 2, 2 has become 3, and this continues.

(Refer Slide Time: 48:35)

Q4. What happens on a write?

- To be discussed next.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, the next question is what happens on a write will be discussed next. We have come to the end of lecture 30 where we have discussed about the various mapping techniques and the block replacement strategies.

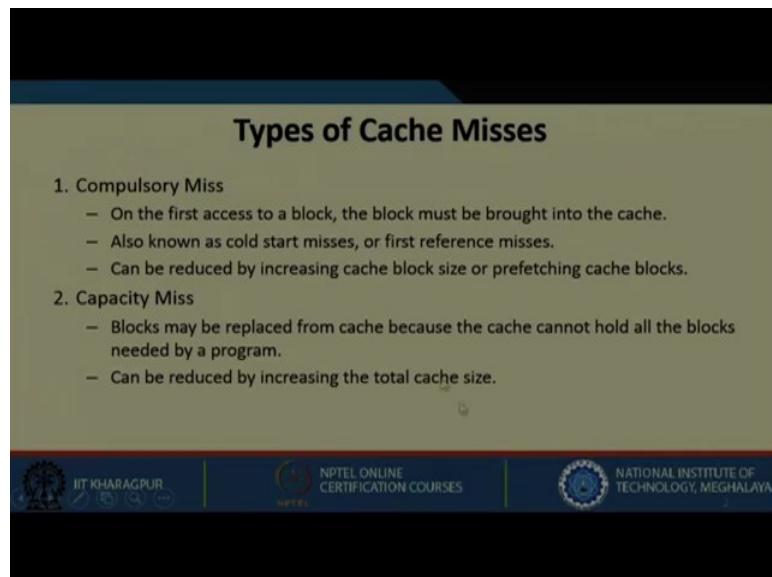
Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 31
Cache Memory (Part II)

Welcome to the next lecture on Cache Memory.

(Refer Slide Time: 00:24)



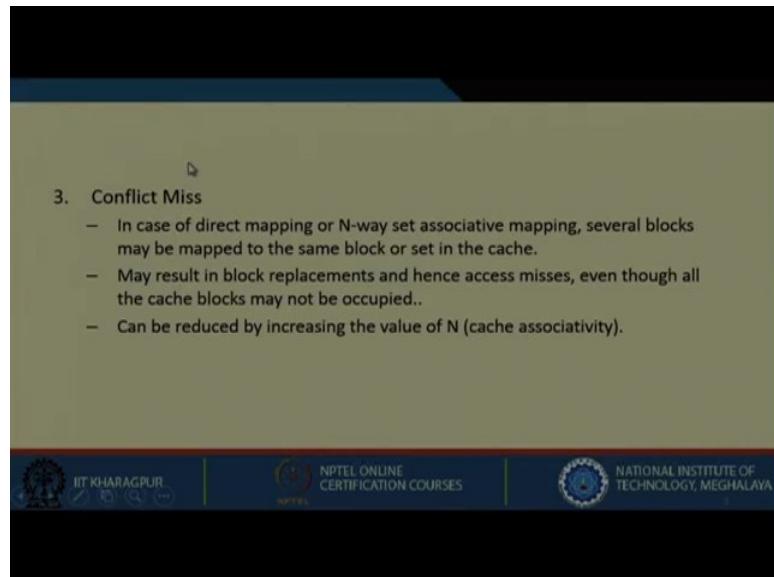
Let us now understand the type of cache misses; by cache misses we mean that we want to perform certain operation like read or write, and that particular data that we are trying to access is not present in the cache. So, we have to bring it from some lower level memory into cache. Under that we have many kinds of misses, the first one is called compulsory miss. As you know that every time when we access a data or an instruction, first we try to access it from the cache. Initially cache is empty. The programs and data are loaded in main memory, and when it is required it is brought from main memory to cache memory, and then to the processor.

So, every time there will be a miss initially, and then due to the locality of reference that particular instruction which has been brought will be accessed again and again. So, the first one which is compulsory miss on the first access to a block, the block must be brought into cache. Also this is known as cold start misses or first reference misses. So, every time for all the instruction or data, there will be a first reference miss. How this can

be reduced? This can be reduced by increasing the cache block size, or prefetching cache blocks. If you can prefetch a cache block early, then this can be reduced.

Next is capacity miss. Blocks may be replaced from cache because the cache cannot hold all the blocks needed by a program. We have seen that in direct mapping; even if there is space we cannot bring all blocks at the same time. Under such situation what can be done; that is called capacity miss. The blocks may be replaced from cache because the cache cannot hold all the blocks needed by the program. How this can be reduced? This can be reduced by increasing the total cache size.

(Refer Slide Time: 03:30)



Another kind of miss is called conflict miss. In case of a direct mapping or N-way set associative mapping we have seen that several blocks may be mapped to the same block or set of the cache.

This may result in block replacement and hence access misses even though all the cache blocks may not be occupied. We say there is a conflict, meaning both for direct mapping and N-way set associative mapping we have seen that the blocks where the data is present may result in block replacement and hence access misses even though all the cache blocks may not be occupied. So, there are some free blocks available, but still we cannot bring the data in those free blocks.

So, in this case there is a restriction that we cannot bring those blocks in any other blocks of the cache memory. This can be reduced by increasing the value of N where we can have the possibility of many main memory blocks to be mapped to one set, and in that set we have to do some searching for getting the particular data. So, this comes under conflict miss.

(Refer Slide Time: 05:12)

Q4. What happens on a write?

- Statistical data suggests that read operations (including instruction fetches) dominate processor cache accesses.
 - All instruction fetch operations are read.
 - Most instructions do not write to memory.
- Making the common case fast:
 - Optimize cache accesses for reads.
 - But Amdahl's law reminds that for high performance designs we cannot ignore the speed of write operations.

The slide footer features logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Let us now try to answer our fourth and last question that is what happens on a write. Now let us see some statistical analysis. Statistical data suggests that the read operation including instruction fetches dominate processor cache accesses. All instruction fetch operations are read, most instruction do not write to memory; that means, out of the total memory accesses most are reads and less are writes.

We can optimize the read operation of the memory; again think of making the common case fast. So, we optimize cache accesses for reads. For all the read operations the cache can be optimized, but we need to remember from Amdahl's law that we cannot ignore the speed of write operations as well.

So, what we try to do is that we make the common case fast. By doing so, we are making read operations faster, but it does not mean we will leave out the write operations. The write operations are performed with less frequency than read, but still we must take care of that as well. The common case, that is the read operation, is relatively easy to make faster.

(Refer Slide Time: 07:22)

- The common case (read operations) is relatively easy to make faster.
 - A block(s) can be read at the same time while the TAG is being compared with the block address.
 - If the read is a HIT the data can be passed to the CPU; if it is a MISS ignore it.
- Problems with write operations:
 - The CPU specifies the size of the write (between 1 and 8 bytes), and only that portion of a block has to be changed.
 - Implies a read-modify-write sequence of operations on the block.
 - Also, the process of modifying the block cannot begin until the TAG is checked to see if it is a hit.
 - Thus, cache write operations take more time than cache read operations.

 IIT KHARAGPUR |  NPTEL: ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

We can make this read operation faster; a block or blocks can be read at the same time while the TAG is being compared with the block address. If the read is a hit, the data can be passed to the CPU; if it is a miss we can ignore it.

What problems do we have with write operations? The CPU specifies the size of the write, and only that portion of the block has to be changed. What does this imply? This implies a read-modify-write; that means, we read it, then we modify it, and then again we write it back. Also the process of modifying the block cannot begin until TAG is checked to see if it is a hit.

So, in read operation while we are checking we are simultaneously reading it, but here it is not possible. Unless and otherwise there is a match, we are not doing any operation. Thus we can say that cache write operations take more time than cache read operation.

(Refer Slide Time: 09:45)

Cache Write Strategies

- Cache designs can be classified based on the write and memory update strategy being used.
 1. Write Through / Store Through
 2. Write Back / Copy Back

```
graph LR; CPU[CPU] --> CM[Cache Memory]; CM --> MM[Main Memory]
```

The diagram illustrates the basic architecture of a cache system. It shows three main components: a CPU, a Cache Memory, and a Main Memory. The CPU is connected to the Cache Memory, which in turn is connected to the Main Memory. This represents a simple write-through strategy where any write operation to the cache is also immediately reflected in the main memory.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Cache design can be classified based on the write and memory update strategy being used. One is write through or store through, another is write back or copy back. We will see both the methods in detail. First let us understand what write through means; we are saying we will be updating directly in main memory. We will write in cache memory and main memory simultaneously, and write back means we will update something in cache memory for the time being and later when that particular block need to be replaced we will update in main memory.

(Refer Slide Time: 10:42)

(a) Write Through Strategy

- Information is written to both the cache block and the main memory block.
- Features:
 - Easier to implement
 - Read misses do not result in writes to the lower level (i.e. MM).
 - The lower level (i.e. MM) has the most updated version of the data – important for I/O operations and multiprocessor systems.
 - A write buffer is often used to reduce CPU write stall time while data is written to main memory.

```
graph LR; CPU[CPU] --> CM[Cache Memory]; CPU --> WB[Write Buffer]; CM --> MM[Main Memory]; WB --> MM
```

The diagram illustrates the Write Through strategy. It shows the CPU connected to both the Cache Memory and a Write Buffer. The Cache Memory is also connected to the Main Memory. Arrows indicate that data from the CPU can be written directly to both the Cache and the Main Memory. The Write Buffer is shown with an arrow pointing to the Main Memory, indicating its role in managing写操作 to the lower-level memory.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us see write through strategy. Here the information is written to both the cache block and the main memory block. We are writing the information in both cache memory as well as in main memory. This is easy to implement as we need not have to think of any other details; we are simply updating using write through strategy by updating the cache memory as well as the main memory. The important feature is it is very much easier to implement and read misses do not result in writes to the lower level, whenever there is a read miss it does not result in writes to the lower level of the memory, that is the main memory.

The lower level has the most updated version of the data, and this is also very important both for IO operation and in multiprocessor. Because in case of IO operation as well as for multiprocessor system as we are updating both cache memory and main memory at the same time, so main memory is always updated. A write buffer is often used to reduce CPU write stall time when data is written to main memory.

So, when we write data into main memory we can have a write buffer and what this write buffer will do? Instead of directly writing into main memory, write buffer will be faster. We will be writing into this write buffer, and in turn the write buffer will be putting in to the main memory. So, we can often reduce the CPU write stall time if we have a write buffer in place.

(Refer Slide Time: 12:56)

The slide has a dark blue header bar with the title 'Write Buffer' in white. Below the header is a light gray content area containing two bulleted lists and some mathematical formulas. At the bottom of the slide is a footer bar with logos for IIT Kharagpur, NPTEL, and a woman speaking.

- Perfect Write Buffer:
 - All writes are handled by write buffer; no stalling for write operations.
 - For unified L1 cache.
$$\text{Stall Cycles / Memory Access} = \% \text{ Reads} \times (1 - H_{L1}) \cdot t_{MM}$$
- Realistic Write Buffer:
 - A percentage of write stalls are not eliminated when the write buffer is full.
 - For unified L1 cache,
$$\text{Stall Cycles / Memory Access} = (\% \text{ Reads} \times (1 - H_{L1}) + \% \text{ write stalls not eliminated}) \times t_{MM}$$

If we have perfect write buffer, all writes are handled by the write buffer; no stalling for write operations is required. But there can be situation where we do not have this perfect write buffer rather we have a realistic write buffer.

In case of realistic write buffer what we generally do, we actually write it at the speed of this buffer; and this buffer size cannot be very much larger. So, a percentage of write stall are not eliminated when the write buffer is full. So, it might happen we are writing more number of words and the write buffer is full in that case we cannot write. Some of the writes can only be performed when this buffer is empty again. So, in that case it cannot be taken care of.

(Refer Slide Time: 14:09)

(b) Write Back Strategy

- Information is written only to the cache block.
- A modified cache block is written to MM only when it is replaced.
- Features:
 - Writes occur at the speed of cache memory.
 - Multiple writes to a cache block requires only one write to MM.
 - Uses less memory bandwidth, makes it attractive to multiprocessors.
- Write-back cache blocks can be *clean* or *dirty*.
 - A status bit called *dirty bit* or *modified bit* is associated with each cache block, which indicates whether the block was modified in the cache (0: clean, 1: dirty).
 - If the status is *clean*, the block is not written back to MM while being replaced.

IIT KHARAGPUR | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

So, there are two things perfect write buffer and realistic write buffer. Now coming to write back strategy, in this case information is written only to the cache block. A modified cache block is written to main memory only when it is replaced. So, the writing is happening at the speed of cache and we are not updating the main memory. So, when will we be updating the main memory? The main memory will get updated when we want to replace that particular block from the cache memory. In that case we need to have some mechanism through which we know that whether this particular block has been updated or it has been not updated.

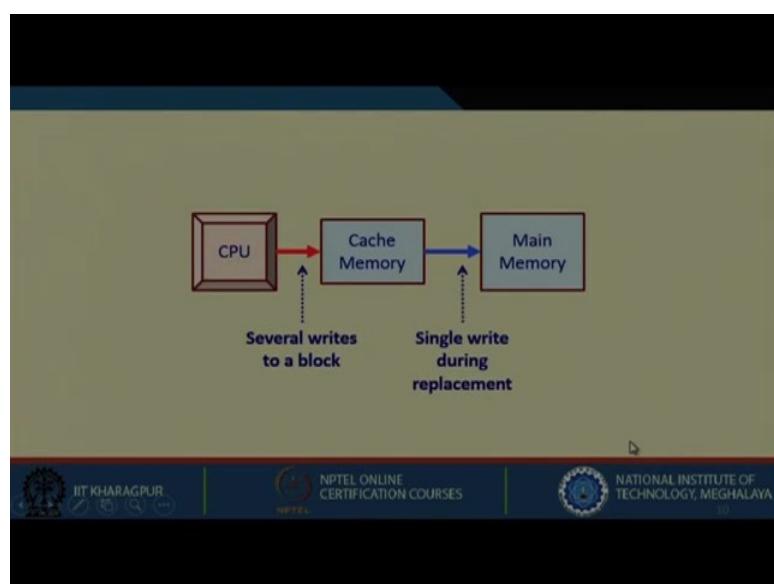
We can see write occurs at the speed of cache memory; multiple writes to the cache block requires only one write to the main memory. That means, even if we are writing

multiple times to the cache block, but at the end only one time it has to be written into the main memory. This is because we are updating not in the main memory, we are updating in the cache block repeatedly; if the block is written repeatedly at end it is required to be written back into the main memory.

So, it uses less memory bandwidth and makes it attractive for multiprocessor. Write back cache blocks can be clean or dirty. What do we mean by clean and dirty? There is a status bit called dirty bit or modified bit with each cache block. It indicates whether the block was modified in the cache or not. If it has been modified then it is marked as 1, that is, it is dirty. If it is not modified it is clean, we mark it as 0. If the status is clean the block is not written back to the main memory while it is being replaced.

So, in a cache block in write back strategy we update a cache block and we keep a status bit associated with it. If that status bit is 1 meaning that particular block when it is residing in the cache memory has been updated then that change should be reflected in main memory. While it is replaced it has to be changed in main memory. Likewise if it is clean; that means, it need not have to be written back into the main memory, because that block has been brought from main memory to cache memory, but it has not been updated. So, no write back to main memory is required.

(Refer Slide Time: 18:18)



What happens between CPU and cache memory? Multiple writes take place between this, but between cache memory and main memory only one time it is written, when the cache block is replaced and new block is brought in.

(Refer Slide Time: 18:45)

Cache Write Miss Policy

- Since information is usually not needed immediately on a write miss, two options are possible on a cache write miss:
 - a) Write Allocate
 - The missed block is loaded into cache on a write miss, followed by write hit actions.
 - Requires a cache block to be *allocated* for the block to be written into.
 - b) No-Write Allocate
 - The block is modified only in the lower level (i.e. MM), and not loaded into cache.
 - Cache block is *not allocated* for the block to be written into.

JIT KHARAGPUR | NPTEL | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, let us see some cache write miss policy. When we write into cache what is the cache write miss policy? Since information is usually not needed immediately on a write miss, two option are possible on a cache write miss. When we want to write the cache and a miss occurs what policy can be adopted? First one is write allocate; that means, the missed block is loaded into the cache on a write miss followed by write hit action.

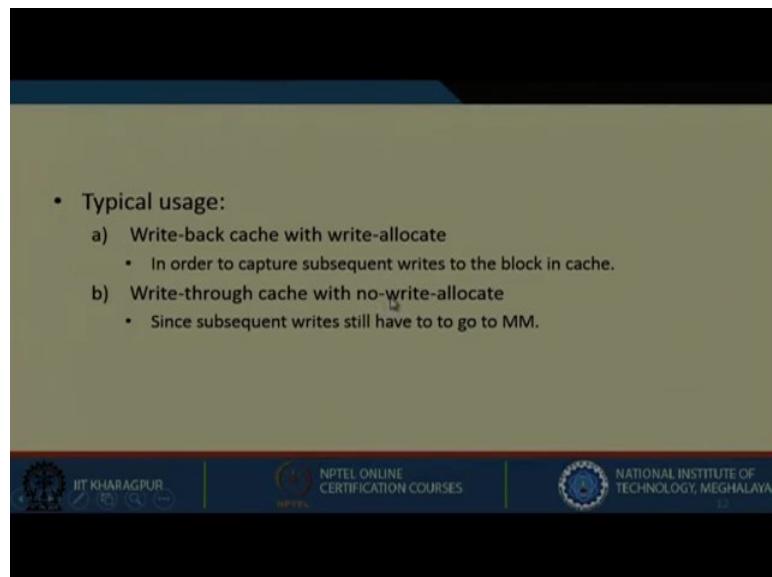
Suppose we write a block into cache, but that block is not present. Write allocate means the missed block is first read and it is brought into the cache, and then it is returned into the cache on a write means followed by a write hit action then write is performed on the that. It requires a cache block to be allocated for the block to be written into. If the cache is not full then it is fine the cache block can be allocated for this purpose. If let us say the cache is full we have to replace one of the blocks from here and put it into the main memory, and then only you can bring the next block and you can go ahead in write allocate.

So, this is what is done. It requires a cache block to be allocated for the block to be written. In such cases if the cache is full then a particular block from the cache needs to be first replaced, and then this block needs to be brought in for write operation. The next

one is no write allocate. In no write allocate the block is modified only in the lower level that is main memory, and not loaded into the cache. No write allocate means we are not allocating any space here. No allocation is done whenever there is a write miss we directly writing into the main memory, and we are not bringing it to the cache memory and then writing it back.

So, cache block is not allocated for the block to be written into. In this case a cache block was allocated, but here there is no need for allocation because the block is modified only in the main memory that is the lower level, and not loaded into the cache. In no write allocate whenever a cache miss occurs these are the two ways to which we can perform it.

(Refer Slide Time: 22:04)



Let us see the typical usage of write back cache with write allocate. Write back means, we will write back later we will update most of the time in cache memory and finally, when the block needs to be replaced we will be writing it back to the main memory. So, in write back cache with write allocate.

So, whenever we are doing write through meaning we are updating directly in main memory, and we are not requiring any space. So, no space in the cache is actually required since subsequent write still have to go to main memory. So, all the write has to be in the main memory. Of course, the access time will be more, but the writing is directly on the main memory.

(Refer Slide Time: 23:54)

The slide has a dark blue header and footer. The main content area has a light beige background. The title 'Estimation of Miss Penalties' is at the top. Below it is a bulleted list:

- Write-Through Cache
 - Write Hit Operation:
 - Without write buffer, miss penalty = t_{MM}
 - With perfect write buffer, miss penalty = 0
- Write-Back Cache
 - Write Hit Operation
 - Miss penalty = 0

At the bottom, there are three logos with text: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Let us now see the estimation of miss penalties for write operation. When it is write through cache, write hit operation without write buffer what will be the miss penalty. Without any write buffer the miss penalty will be access time of main memory only. So, whenever a write hit occurs we are updating directly into the main memory. And with perfect write buffer, there will be no miss penalty because we are writing into the buffer, and the buffer is taking care of that. So, the miss penalty becomes zero here.

In write back cache when write hit operation happens, the miss penalty is zero because we are doing it in the cache itself and the data is present in the cache.

(Refer Slide Time: 25:17)

- Write-Back Cache (with Write Allocate)
 - Write Hit Operation
 - Miss penalty = 0
 - Read or Write Miss Operation
 - If the replaced block is clean, miss penalty = t_{MM}
 - No need to write the block back to MM.
 - New block to be brought into MM (t_{MM}).
 - If the replaced block is dirty, miss penalty = $2 t_{MM}$
 - Write the block to be replaced to MM (t_{MM}).
 - New block to be brought into MM (t_{MM}).

Now, let us see write back cache with write allocate. Here when write hit operation happens, the miss penalty will be zero. For read or write miss operation what happens? If the replaced block is clean; miss penalty is t_{MM} ; no need to write the block to main memory, and new block to be brought into main memory. So, basically we need not have to write the block into the main memory only a new block will be brought from main memory. So, we are accessing t_{MM} once only.

Now if the replaced block is dirty; miss penalty will be $2 t_{MM}$. So, you are writing it to main memory the time will be t_{MM} , a new block to be brought into main memory that will be again t_{MM} . So, twice time of main memory will be required.

So, we have seen so many cases what happens when a write hit occurs, what happens in a write miss occurs, both in case of write through and write back.

(Refer Slide Time: 27:32)

Choice of Block Size in Cache

- Larger block sizes reduce compulsory misses.
- Larger block sizes also reduce the number of blocks in cache, increasing conflict misses.
- Typical block size: 16 to 32 bytes.

Regarding choice of block size in cache, larger block size reduced compulsory misses. If you have larger block size then the compulsory misses can be reduced, and larger block size also reduces the number of blocks in cache increasing the conflict misses.

Typical block size is 16 to 32 bytes.

Next is instruction only and data only caches.

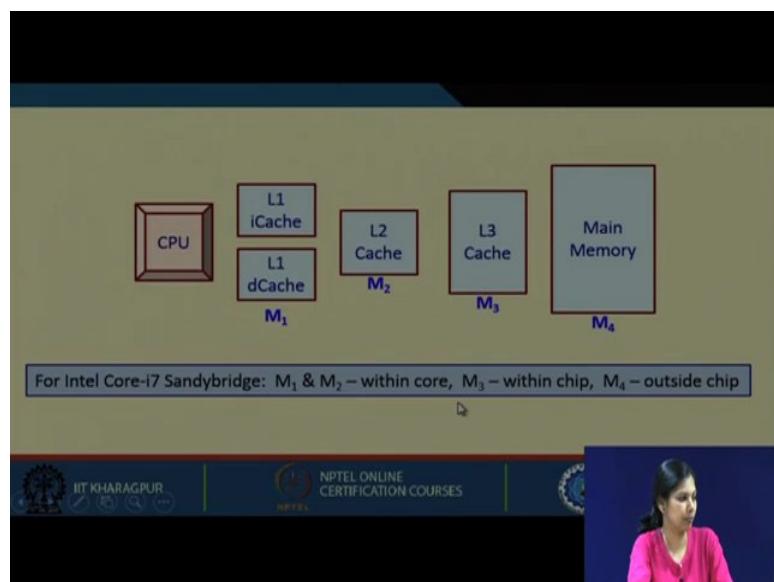
(Refer Slide Time: 28:20)

Instruction-only and Data-only Caches

- Caches are sometimes divided into instruction-only and data-only caches.
 - The CPU knows whether it is issuing an instruction address or a data address.
 - There are two separate ports, thereby doubling the bandwidth between the CPU and the cache.
 - Typical L1 caches are separated into *L1 i-cache* and *L1 d-cache*.
- Separate caches also offers the opportunity of optimizing each cache separately.
 - Instruction and data reference patterns are different.
 - Different capacities, block sizes, and associativity (i.e. *N*).

We have already seen that for instruction only and data only caches, the CPU knows whether it is issuing an instruction address or a data address; there are two separate ports thereby doubling the bandwidth between the CPU and the cache. Typical L1 caches are separated into L1 I-cache and L1 D-cache. Separate caches also offer the opportunity of optimizing each cache separately, now we know that read operations are more. So, the instruction cache can be improved in a different manner. And data references patterns are also different the way the instruction are different and way the pattern are different. So, different capabilities, block sizes and associativity can be implemented for two different kinds of caches.

(Refer Slide Time: 29:33)



This is for an Intel core-i7 SandyBridge, where M_1 and M_2 are within the core, and M_3 is within the chip, and M_4 is outside the chip. Main memory is always outside the chip.

So, we come to the end of lecture 31. In the next lecture we will be looking into some of the strategies to improve the cache performance further.

Thank you.

Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 32
Improving Cache Performance

Welcome to the last lecture on Cache Memory. In this lecture we will be seeing some methods for improving the cache performance, prior to that we will look into two examples.

(Refer Slide Time: 00:31)

The screenshot shows a presentation slide with a dark blue header and a light brown main content area. The title 'Example 1' is centered at the top of the content area. Below the title is a bulleted list of instructions. At the bottom of the slide, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya. Below these logos, there are two equations: 'Number of memory accesses per instruction = 1 + 0.15 + 0.15 = 1.3' and '% Reads = (1 + 0.15) / 1.3 = 88.5%' followed by '% Writes = 0.15 / 1.3 = 11.5%'.

Example 1

- Consider a CPU with average CPI of 1.1.
 - Assume an instruction mix: ALU – 50%, LOAD – 15%, STORE – 15%, BRANCH – 20%
 - Assume a cache miss rate of 1.5%, and miss penalty of 50 cycles ($= t_{MM}$).
 - Calculate the effective CPI for a unified L1 cache, using *write through and no write allocate*, with:
 - a) No write buffer
 - b) Perfect write buffer
 - c) Realistic write buffer that eliminates 85% of write stalls.

Number of memory accesses per instruction = $1 + 0.15 + 0.15 = 1.3$

% Reads = $(1 + 0.15) / 1.3 = 88.5\%$ % Writes = $0.15 / 1.3 = 11.5\%$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us consider a CPU with average CPI as 1.1, let us assume that an instruction mix like ALU having 50%, load having 15%, store 15%, branch 20%, and assume that a cache miss rate is 1.5%, and the miss penalty that means, if there is a miss what is the time required to bring it from the next level that is; let us a main memory is 50 cycles. We need to calculate the effective CPI for a unified L1 cache that uses write through and no write allocate.

Along with this we will be calculating with no write buffer, next we will be doing with perfect write buffer, and another one we will be doing with realistic write buffer that eliminates 85% of write stalls.

Let us first calculate the number of memory accesses per instruction. When there will be a memory access there will be a memory access for load and store. So, let us say for fetching the instruction we need to have one cycle; we require one cycle for that and then 0.15 for load 15 percent of load instruction, and 0.15 for store where we will require memory operation. So, total is coming to 1.3.

So, for all memory accesses all instruction accesses is 1 and 15 percent for load and 15 percent for store. Now percentage read will be how much? Percentage read will be once we are reading a particular instruction and then again when we are reading when we are loading a word. So, 1 for reading every instruction, and 0.15 for load instruction, divided by total number of memory accesses per instruction which is coming down to 88.5 percent. So, percentage read is 88.5 percent.

Now, coming to percentage writes for the write we will require this 15 percent only, 15 percent divided by 0.13 that is the total number of memory access per instruction. So, now, we have found out percentage read, percentage write and we also know the total number of memory accesses per instruction per instruction what will be the total number of memory accesses.

(Refer Slide Time: 03:54)

The slide contains the following text and calculations:

- Solution:**
- a) With no write buffer (i.e. *stall on all writes*)
 - Memory stalls / instr. = $1.3 \times 50 \times (88.5\% \times 1.5\% + 11.5\%) = 8.33$ cycles
 - CPI = $CPI_{avg} + \text{Memory stalls / instr.} = 1.1 + 8.33 = 9.43$
- b) With perfect write buffer (i.e. *all write stalls are eliminated*)
 - Memory stalls / instr. = $1.3 \times 50 \times (88.5\% \times 1.5\%) = 0.86$ cycles
 - CPI = $1.1 + 0.86 = 1.96$
- c) With realistic write buffer (*85% of write stalls are eliminated*)
 - Memory stalls / instr. = $1.3 \times 50 \times (88.5\% \times 1.5\% + 15\% \times 11.5\%) = 1.98$ cycles
 - CPI = $1.1 + 1.98 = 3.08$

At the bottom of the slide, there is a footer with the IIT Kharagpur logo, NPTEL logo, and the text "NPTEL ONLINE CERTIFICATION COURSES". To the right of the footer, a video frame shows a woman speaking.

Now, coming to the solution, in the first case we need to consider where we say that there is no write buffer. So, in this particular case there will be stall on all writes. So,

whenever there is no write buffer. So, we are not writing it to a high-speed hardware rather we are writing it back to the main memory or you can say the lower level memory.

So, in that case how do we consider the number of stalls ... there will be stalls on all writes. So, the memory stalls per instruction will be 1.3 multiplied by 50, into we have percentage read of 88.5 percent, this is percentage write of 11.5 percent and assume that there is a cache miss of 1.5 percent. So, we need to consider that as well. So, if we consider that and what is 50? 50 is our miss penalty. So, for miss penalty there will be 50 cycles. So, when we multiplied 1.3 into 50 into this percentage, we get roughly 8.33 cycles. This is the total number of cycles that we are getting; that means, these many memory stalls per instruction will be there.

So, what will be the CPI? CPI will be average CPI plus we have to consider this memory stalls CPI. So, this memory stalls per instruction is 8.33 we have calculated. So, total CPI will come down when we have no write buffer as 9.43 when we are using write through policy without any no write allocate.

Let us see the next case where we have a perfect write buffer; in this particular case all write stalls are eliminated. So, there will be no write stalls now. So, it will be 1.3 multiplied by miss penalty, multiplied by the reads percentage read and the miss rate we have to multiply these 2 only, and we are not considering anything for the write in the previous case we have considered this 11.5, but here we are not considering that.

So, for this it is coming to 0.86 cycles. So, the CPI when we have a perfect write buffer we have 1.1 plus 0.86 which is coming to 1.96. So, we can see that there is a huge difference between when we have no write buffer and when we have some write buffer. Now let us consider a realistic write buffer. So obviously, perfect write buffer is difficult to have because there might be some writing going on and at the same time some read operation is also required.

So, in such cases if there are 85 percent of the write stalls are eliminated; that means, 15 percent will still be there, but 85 percent there will be no write stall. So, in that case how do we find out 1.3 into miss penalty, multiplied by this was the miss plus a read percentage for the read percentage it will be 1.5 and there will be 15 percent for the write.

So, let us just see this is 11.5 percent and for this 11.5 percent this will be 1.5 this is not 15 this should be 1.5. So, which will come down to if you solve this particular equation it will come down to 1.98 cycles. So, it is now coming down to CPI 1.1 plus 1.980, which is roughly equal to 3.08 which is to some extent a realistic value because we will definitely have some even if we have write buffer there will be some stalls.

(Refer Slide Time: 08:46)

The slide has a dark blue header and a light beige body. At the top center, it says "Example 2". Below that is a bulleted list of tasks:

- Consider a CPU with average CPI of 1.1.
 - Assume the instruction mix: ALU – 50%, LOAD – 15%, STORE – 15%, BRANCH – 20%
 - Assume a cache miss rate of 1.5%, and miss penalty of 50 cycles ($= t_{MM}$).
 - Calculate the effective CPI for a unified L1 cache, using *write back and write allocate*, with the probability of a cache block being dirty is 10%.

Below the list is the formula: "Number of memory accesses per instruction = $1 + 0.15 + 0.15 = 1.3$ ".

The footer of the slide shows the IIT Kharagpur logo, the NPTEL logo, and the text "NPTEL ONLINE CERTIFICATION COURSES". To the right of the footer, a video camera icon is visible, indicating a live video feed of a speaker.

Let us consider another example where we have similar kind of values, but now we have calculate the effective CPI for a unified L1 cache using write back and write allocate; that means, we will not be writing it through to the main memory always, rather we will be writing it into the cache memory with write allocate; that means, we must have a space in our cache memory to write that particular thing.

With the probability of cache block being dirty is 10%, and we assume that 10% of the time the cache block will be dirty and if the cache block is dirty we know we knew what we need to do it in that particular case, we have to write back the data into the main memory and then we have to again read or write whatever we need to do the next.

(Refer Slide Time: 09:58)

The image shows a presentation slide with a dark blue header and a light brown main content area. The slide contains a bulleted list under the heading 'Solution:' and some mathematical calculations. Below the slide, there is a video feed of a woman in a pink shirt speaking. The video feed is framed by a dark border. At the bottom of the screen, there is a navigation bar with icons for back, forward, and search, along with the text 'IIT KHARAGPUR' and 'NPTEL ONLINE CERTIFICATION COURSES'.

- Solution:
 - Memory accesses per instruction = 1.3
 - Stalls / access = $(1 - H_{L1}) \cdot (t_{MM} \times \% \text{ clean} + 2t_{MM} \times \% \text{ dirty})$
= $1.5\% \times (50 \times 90\% + 100 \times 10\%) = 0.825 \text{ cycles}$
 - Memory stalls / instr. = $1.3 \times 0.825 = 1.07 \text{ cycles}$
 - Thus, effective CPI = $1.1 + 1.07 = 2.17$

So, the number of memory accesses per instruction will be 1 plus 0.15 plus 0.15 that is 1.3. Now let us see the memory accesses per instruction which is 1.3, stalls per accesses will be one minus hit ratio of L1 x t_{MM} x percentage time it is clean, because when it is clean then we have to only write it once. If it is dirty then we have to do it twice we have already seen that in our previous lecture. The calculation is shown, which comes to 0.825 cycles.

Now we see the memory stalls per instruction. We will just multiply the total memory accesses per instruction and the memory stalls per instruction, and we get 1.07 cycles. Thus what will be the effective CPI $1.1 + 1.07$ which is coming to 2.17 which is even less than the realistic write buffer if we use in our previous case when we are using write through policy.

So, these are the 2 examples we have discussed for writing into the cache using write through and write back policies.

(Refer Slide Time: 11:46)

Introduction

- We shall discuss various techniques using which the performance of cache memory can be improved.
- We consider the following expression for average memory access time (AMAT):
$$AMAT = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$
- When we talk about improving the performance of cache memory systems, we can try to reduce one or more of the three parameters: *Hit time, Miss rate, Miss penalty*.

AMAT = Hit time + Miss rate × Miss penalty

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Now, coming to the methods that we will be looking into for cache improvement, how we can further improve the cache. Let us consider this expression of average memory access time AMAT, which is the hit time (basically hit ratio multiplied by the access time of the cache) plus miss rate into miss penalty. So, when we talk about improving the performance of this cache memory system, ultimately we need to reduce AMAT. So, either we can reduce hit time, or reduce miss rate or miss penalty.

(Refer Slide Time: 12:48)

Basic Cache Optimization Techniques

- We can categorize the techniques into three categories based on the parameter that is being optimized:
 - **Reducing the miss rate:** we can use larger block size, larger cache size, and higher associativity.
 - **Reducing the miss penalty:** we can use multi-level caches and giving priority to reads over writes.
 - **Reducing the cache hit time:** we can avoid the address translation when indexing the cache.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, we need to see how we can take care to reduce all these parameters. In basic cache optimization techniques we categorize the techniques into three groups based on the parameters. How we can reduce the miss rate? Miss rate can be reduced by using larger block size. If you use a large block size then if my program is pretty large and you have a large block, the whole program is brought to some block of the cache and you can access it nicely. So, larger block size in turn reduces the miss rate.

Now how do we reduce miss penalty? By miss penalty we mean that if you have multi level cache we bring the data from main memory into the upper level of memory, and then from upper level of memory it goes to cache. The miss penalty can be reduced by having multi level caches.

Now, how do we reduce the cache hit time? When we are hitting cache and we are searching for the TAG, we are matching something. From the address part we have already seen we have certain parts for different kinds of mapping technique; if you are using set associative mapping technique you have a TAG you have a SET you have a WORD. So, you need to match that TAG with the number of blocks that you are having. When you are matching this TAG the CPU generates a logical address, the logical address gets translated into physical address and then that matching takes place.

Instead once we know the logical address, from the logical address if we can extract the TAG and then can directly start this matching, the hit time can be reduced.

(Refer Slide Time: 15:07)

(a) Use Larger Block Size

- Increasing the block size helps in reducing the miss rate.
 - See plot on the next slide.
- Larger blocks also reduce compulsory misses.
 - Since larger blocks can take better advantage of spatial locality.
- Drawbacks:
 - The miss penalty increases, as it is required to transfer larger blocks.
 - Since the number of cache blocks decreases, the number of conflict misses and even capacity misses can increase.
 - The overheads may outweigh the gain.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

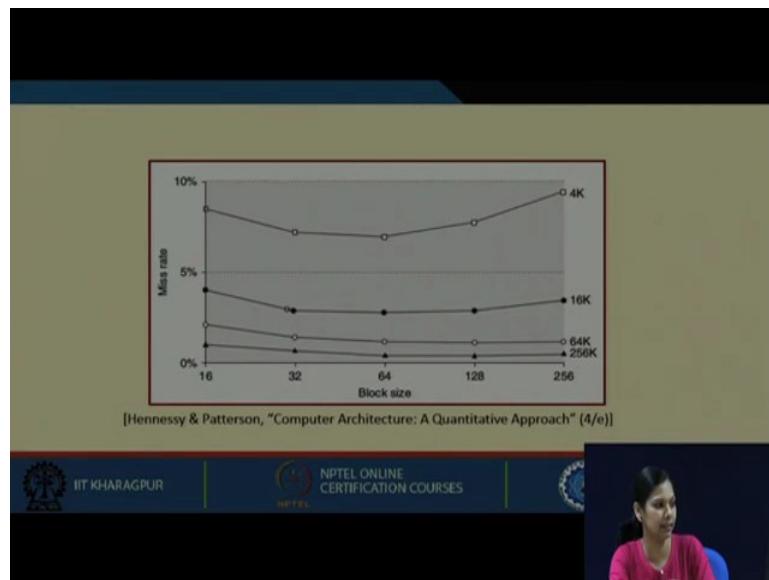
Let us see each one of the points in some detail. When we use larger block size what happens? Increasing the block size helps in reducing the miss rate; obviously, if you have a larger block size the entire program you can bring to the same block and then you can access in a faster fashion using that.

We have various kinds of misses that also we have seen in the previous lecture. Larger block size will also reduce the compulsory misses. Every time you bring a data there is a miss, but if you want to bring a data you bring a larger block where all the data are now brought in. And now you will be just hitting the cache to get the subsequent one, the time will be spent in bringing the entire data from the lower level of the memory to higher level of the memory, but then you can access it very easily, but what will be the drawback. The miss penalty increases as it is required to transfer a large block.

So, for bringing a large block, there will be more number of words in that particular block. You have to bring all the words into your cache. In that particular case you have to wait until the entire block is brought in. So, the miss penalty will increase, but in turn the hit time will increase. Since the number of cache blocks decreases the number of conflict misses and even capacity misses can increase. So, conflict miss will happen we know for direct mapping and for set associative mapping, where different blocks of main memory are mapped to a same block of cache memory.

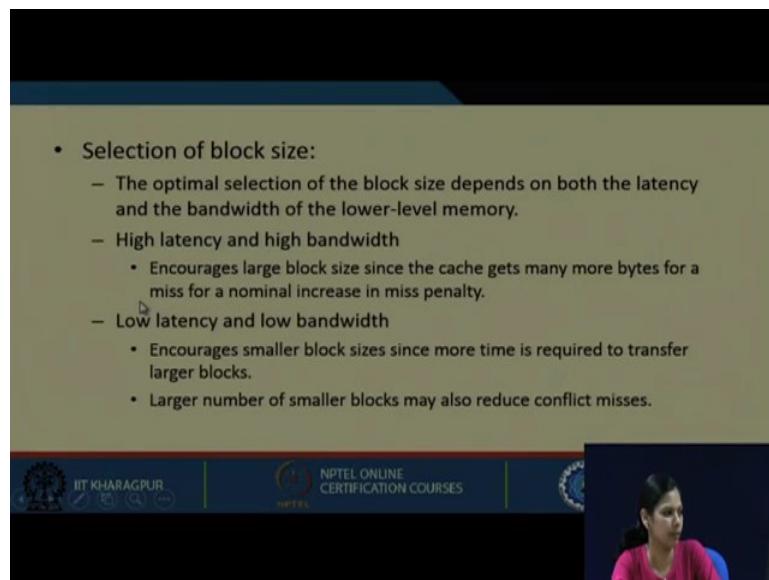
In that case this conflict will be more because you have smaller number of blocks because the block size you have increased. If the cache size is limited and the block size is more, smaller number of blocks will be there. So, if you want to bring two blocks together then it will be a problem. So, it can be seen that if you have a larger block size the overhead that we are getting might outweigh the gain, the gain which we were expecting you might not get, but the overhead becomes much more.

(Refer Slide Time: 18:11)



Let us see this particular diagram. This is the miss rate, this is the block size and this is the cache size. When the cache size is small the miss rate decreases, but after certain point the miss rate again increases, whereas if you have a larger block size you can see that the miss rate decreases and there is an increase, but not so much here.

(Refer Slide Time: 19:03)



We have seen that if you increase the block size the miss rate will decrease. In such case, other overheads will also be there; conflict misses and capacity misses will also be there,

then how do we select the block size? The optimal selection of block size depends on both the latency and the bandwidth of the lower level memory.

If we have high latency and high bandwidth this encourages large block size, since the cache gets many more bytes for a miss for a nominal increase in miss penalty. So, with nominal increase in miss penalty we get more bytes from a miss. If you have low latency and low bandwidth this encourages smaller block size since more time is required to transfer large blocks of course, more time will be required to transfer larger blocks.

Larger number of smaller blocks may also reduce conflict misses if you have more number of blocks, then this conflict miss can get reduced.

In such situation what can happen is that if you have large number of smaller blocks, you will have various opportunities of mapping different blocks. So, the conflict misses can even reduce.

(Refer Slide Time: 21:11)

(b) Use Larger Cache Memory

- Increasing the size of the cache is a straightforward way to reduce the capacity misses.
- Drawbacks:
 - Increases the hit time since the number of TAGs to be searched in parallel will be possibly larger.
 - Results in higher cost and power consumption.
- Traditionally popular for off-chip caches.

Next is use larger cache memory. So, if you have larger cache memory, then it will be easy. So, let us see how it can improve. Increasing the size of the cache is straightforward. It is a straight way straightforward way to reduce capacity misses, but what is the drawback then? Increases the hit time since the number of tags to be searched in parallel will be possibly large.

So, one of the parameter is increasing. So, we are decreasing the misses the miss rate is decreasing, but the search time for the tag in the cache memory will increase, that is, the hit time result in higher cost and power consumption of course. We know that cache memory is built using static RAM that requires roughly 6 transistors for one bit. So, it is much more costlier compared to data memory. If you are increasing the cache memory you are increasing the cost as well. So, all these are related. So, which one you will reduce to how much such that you will get a better gain in turn needs to be analyzed and understood.

(Refer Slide Time: 22:38)

(c) Use Higher Associativity

- For N -way associative cache, the miss rate reduces as we increase N .
 - Reduces conflict misses, as there are more choices to place a block in cache.
- General rule of thumb:
 - 8-way set associative cache is as effective as fully associative for practical scenarios.
 - A direct mapped cache of size N has about the same miss rate as a 2-way set associative cache of size $N/2$.
- Drawbacks:
 - Increases the hit time as we have to search a larger associative memory.
 - Increases power consumption due to higher complexity of associative memory.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NPTEL

We can use higher associativity. So, for a N -way associative cache the miss rate reduces as we increase N . This reduces the conflict misses as there are more choices to place a block in a cache. As the size of this N increases what we can do is that more number of blocks from main memory can be placed in same set in that particular cache.

In that way the conflict misses will definitely reduce. So, there is a general rule of thumb: 8-way set associative cache is as effective as fully associative for practical scenario. So, we need not have to pay the cost for a fully associative cache.

So, direct map cache of size N has about the same miss rate as a 2-way set associative cache of size $N/2$. So, if we have a $N/2$ size 2-way set associative cache, it will be same as direct mapping. The direct mapping will be easy to implement as well. So, what is the

drawback of this? Increases the hit time and as we have to search a larger associative memory.

(Refer Slide Time: 25:05)

(d) Use Multi-level Caches

- Here we try to reduce the miss penalty, and not the miss rate.
- Performance gap between processors and memory increases with time.
 - Use faster cache to keep pace with the speed of the processor?
 - Make the cache larger to bridge the widening gap between processor and MM?
- We can use both in a multi-level cache system:
 - The L1 cache can be small enough to match the clock cycle time of the fast processor.
 - The L2 cache can be large enough to capture many accesses that would go to MM, thereby reducing the miss penalty.

This is another way use of multi level caches; here we try to reduce the miss penalty and not the miss rate. So, we are not reducing the miss rate rather we are trying to reduce the miss penalty, that is the time to bring the data from your main memory to cache memory and then to processor.

The performance gap between processor and memory increases with time that we already know, use faster cache to bridge the widening gap between processor and main memory. Multiple number of levels of cache will be helpful because we are bringing the data from main memory and we are keeping in those multi level caches, and whenever it is required by the processor and if it is not present in L1 cache, it is looking into L2 and L3 to find out the data and in most cases they are getting it from L1 and L2 and they do not have to go to main memory to access the data. The L1 cache can be small enough to match the clock cycle time of the fast processor, the L2 cache can be large enough to capture many accesses that would go to main memory thereby reducing the miss penalty.

So, what we are trying to say here is that L1 cache will be small enough and it will be much faster, and L2 cache will be little larger such that most of the data from main memory will be there in L2 cache, and whenever it is not present in L1 cache I am just

getting it from L2 cache and not from main memory. So, we are getting some advantage out here.

(Refer Slide Time: 27:44)

- Consider a 2-level cache system, consisting of L1 cache and L2 cache.
- The average memory access time can be computed as:
$$AMAT = HitTime_{L1} + MissRate_{L1} \times MissPenalty_{L1}$$
where $MissPenalty_{L1} = HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2}$
- Thus,
$$AMAT = HitTime_{L1} + MissRate_{L1} \times (HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2})$$
- The second-level miss rate $MissRate_{L2}$ is measured on the leftovers from the first-level cache.

So, let us consider a 2-level cache system consists of L1 cache and L2 cache, the average memory access time can be computed as shown.

(Refer Slide Time: 28:35)

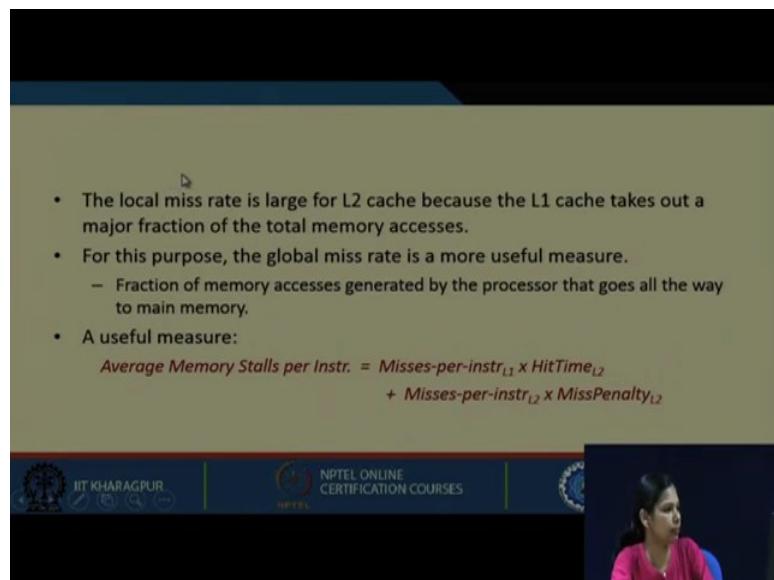
- We define the following for a 2-level cache system:
 - Local Miss Rate
 - This is defined as the number of misses in a cache divided by the total number of accesses to this cache.
 - For the first level, this is $MissRate_{L1}$
 - For the second level, this is $MissRate_{L2}$
 - Global Miss Rate
 - This is defined as the number of misses in a cache divided by the total number of memory accesses generated by the processor.
 - For the first level, this is $MissRate_{L1}$
 - For the second level, this is $MissRate_{L1} \times MissRate_{L2}$

We define the following for the 2-level cache system here. One is call local miss rate; this is defined as the number of misses in a cache divided by total number of accesses to this cache. So, for the first level this is miss rate of L1, and for the second level this is

miss rate of L2. Now what do you mean by global miss rate? This is defined as the number of misses in the cache divided by total number of memory accesses generated by the processor.

So, we are not separately taking miss for L1 and miss for L2; rather we are taking in terms of the total. This is in terms of total number of memory accesses generated by the processor. For the first level this is miss rate of L1, but for the second level it will be miss rate of L1 x miss rate of L2.

(Refer Slide Time: 29:51)



The local miss rate is large for L2 cache because the L1 cache takes out a major fraction of the total memory access, because we are separately calculating for this purpose the global miss rate is more useful measure. So, we generally use this global miss rate, fraction of memory access is generated by the processor that goes all the way to the main memory.

A useful measure that can be used is average memory stalls for instruction, defined as shown.

(Refer Slide Time: 30:50)

Example 1

- Suppose that in 1000 memory references there are 60 misses in L1-cache and 15 misses in L2-cache. What are the various miss rates?
Assume that MissPenalty_{L2} is 180 clock cycles, HitTime_{L1} is 1 clock cycle, and HitTime_{L2} is 12 clock cycles.
What will be the average memory access time? Ignore the impact of writes.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Now, let us see this example. Suppose that 1000 memory references are there and there are 60 misses in L1 cache and 15 misses in L2 cache, what are the various miss rates. So, total is 1000 out of which 16 misses are there in L1 and 15 misses are there in L2. Assume that miss penalty is 180 clock cycles, hit time of L1 is 1 clock cycle, and hit time of L2 is 12 clock cycle, what will be the average memory access time ignore the impact of writes?

(Refer Slide Time: 31:34)

Solution:

- MissRate_{L1} = 60 / 1000 = 6 % (both local or global)
- LocalMissRate_{L2} = 15 / 60 = 25 %
- GlobalMissRate_{L2} = 15 / 1000 = 1.5 %

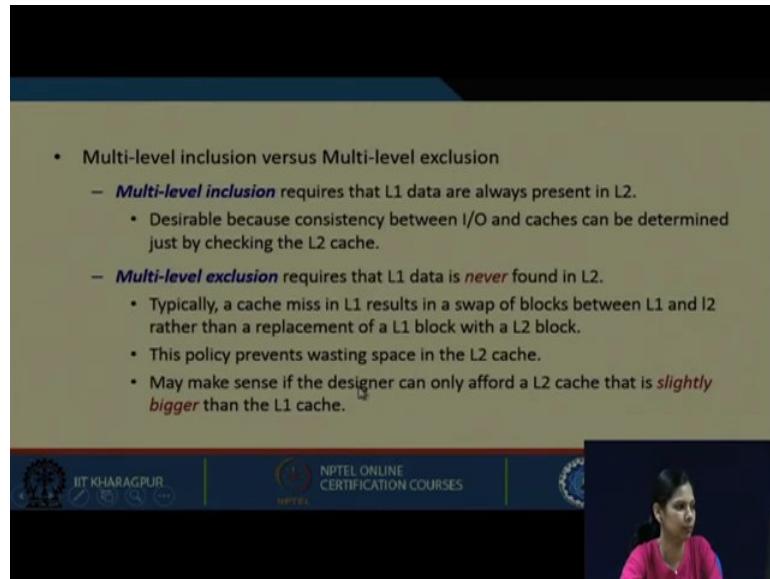
$$\begin{aligned} \text{AMAT} &= \text{HitTime}_{L1} + \text{MissRate}_{L1} \times (\text{HitTime}_{L2} + \text{MissRate}_{L2} \times \text{MissPenalty}_{L2}) \\ &= 1 + 6\% \times (12 + 25\% \times 180) \\ &= 1 + 6\% \times 57 = 4.42 \text{ clock cycles} \end{aligned}$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

The miss rate of L1 will be $60 / 1000$, because total we are considering that is 6% both for local or global.

Miss rate of L2 means how many misses total in L1 is 60; out of 60 there are 15 misses. So, $15 / 60 = 25\%$, and global miss rate for L2 will be $15 / 1000 = 1.5\%$.

The calculation of AMAT is shown (Refer Slide Time: 32:41)



Let us see another thing; multi level inclusion versus multi level exclusion. Multi level inclusion requires that L1 data are always present in L2 cache, inclusion means it is included. So, in L1 we have certain data and L2 cache is the next level cache where we are saying that whatever data is present in L1 is also there in L2, that is called multi level inclusion. This is desirable because for the consistency between IO and caches can be determined just by checking the L2 cache.

The IO can just check the L2 cache and find out the data. What is multi level exclusion? This requires that L1 data is never found in L2; that means, L1 data are certain data which is there in L1, and L2 data are certain data that is present in L2, but there is no common data in between. That means, whatever is there in L1 is not there in L2. Typically a cache miss in L1 result in a swap of block between L1 and L2, rather than replacement of L1 block with a L2 block.

So, there is no replacement rather you are bringing a block from L2 to L1. This policy prevents wasting space in L2 cache.

(Refer Slide Time: 34:50)

(e) Giving Priority to Read Misses Over Writes

- The presence of write buffers can complicate memory accesses.
 - The buffer may be holding the updated value of a location needed on a read miss.
- Simplest solution is to make the read miss to wait until the write buffer is empty.
 - As an alternative, check the contents of the write buffer for any conflict; and if none, the read miss can continue → *reduces read miss penalty*.
 - Most desktops and servers follow this approach, giving priority to reads over writes.

Now, giving priority to read misses over writes. We know that reads are more frequent, the presence of write buffers can complicate the memory accesses, the buffers may be holding the updated value of the location needed on a read miss. So, whenever there is a read miss the buffer may be holding an updated data. The simplest solution that is there is to make the read miss to wait until the write buffer is empty.

As an alternative what can be done is check the content of the write buffer for any conflict, if there is any conflict we have to check the write buffer and if none the read miss can continue.

(Refer Slide Time: 36:05)

(f) Avoiding Address Translation during Cache Indexing

- Even a small and simple cache must cope with the translation of a virtual address to a physical address to access memory.
- An idea to make the common case fast:
 - We use virtual addresses for cache, since hits are much more common than misses.
 - Such caches are termed as *virtual caches*.
- Drawback:
 - Page level protection is not possible.
 - Context switching and I/O (that uses physical addresses) further complicates the design.

Another is avoiding address translation during cache indexing. As I said that whenever CPU is generating a logical address, it is first translated into physical address and then we extract the TAG bits that are matched. Finally, we see that whether the data or instruction is present in cache memory or not. Even a small and simple cache must cope with the translation of virtual address to physical address to access memory. So, this has to be done.

As in an idea to make the common case first, what can be done we use virtual addresses for cache since hits are much more common than misses. Such caches are termed as virtual caches. We will have some virtual caches in place to make the common case fast, but what is the drawback? We are saying that we will not be having a translation rather we will be extracting the address from the logical address only.

Page level protection is not possible, this is a drawback and context switch and IO that uses physical address further complicates the design, because this IO and the context switching; that means, switching between two processes is required often. So, that process becomes more complicated.

(Refer Slide Time: 37:43)

Some Additional Cache Optimizations

1. Use small and simple first-level caches to reduce hit time
2. Way prediction to reduce hit time
3. Pipelined cache access to increase cache bandwidth
4. Multi-banked caches to increase cache bandwidth
5. Critical Word First and Early Restart to reduce miss penalty
6. Compiler optimizations to reduce miss rate
7. Prefetching of instructions and data to reduce miss penalty or miss rate

These are some additional cache optimizations like to use small and simple first level caches to reduce hit time, way prediction to reduce the hit time, pipeline cache access to increase cache bandwidth, multi-banked caches to increase the cache bandwidth, critical word first and early restart to reduce the miss penalty, compiler optimization to reduce miss rate and prefetching of instruction, and data to reduce miss penalty or miss rate. So, these are some more cache optimization through which we can reduce hit time, miss penalty and miss rate.

There are also some more methods through which miss rate, hit time, and miss penalty can be further reduced.

We are at the end of lecture 32 and we are done with memory system. So, in this week what we have seen are various kinds of techniques through which we can reduce the memory access time, such that it can bridge the speed gap between the processor and the memory.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 33
Design of Adders (Part I)

So in this course we have so far seen various aspects of computer organization and architecture. So, if we recall we looked at the instructions and architecture of a typical processors. As a case study we looked at the MIPS 32 processor, then you looked at the design of the data path and how we can design the control unit of a typical processor or machine, then we looked at various aspects of memory system design.

Today starting from this lecture we shall be discussing various aspects of designing the arithmetic logic unit of a computer system. So, as you know a computer system essentially is meant to do some computation and in that sense the ALU or the arithmetic logic unit forms the basic heart of the system.

So, we shall be starting by looking at how we can design the different kinds of circuits for implementing addition, multiplication, division etc. So, the topic of our lecture today is design of adders.

(Refer Slide Time: 01:40)

Introduction

- Computers are built using tiny electronic switches.
 - Typically made up of MOS transistors.
 - The state of the switches are typically expressed in binary (ON/OFF).
- To design arithmetic circuits for use in computers, we need to work with binary numbers.
 - How to represent numbers in binary?
 - How to carry out various arithmetic operations in binary?
 - How to implement them efficiently in hardware?

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

As you know that computers are built using tiny electronic switches the switches are typically in the form of MOS transistors now the way this MOS transistors work in a digital circuit or digital system is that they can be turned on or off, they can be in 1 of 2 states. So, quite naturally such a system can be used to implement or model a binary number system where the state of the switch can be mapped to a binary digit 0 or 1.

So, the essential idea here is that when you are talking about designing arithmetic circuits we need to work with binary numbers although in practice we are more familiar with the decimal number system the way we work we calculate on paper, but here with respect to computers we need to work with binary numbers.

To recall you can represent binary numbers in either an unsigned form or in signed form; with respect to signed number representation we have seen the sign magnitude 1's complement and the 2's complement representations. What we will see now is that how we can carry out various arithmetic operations in binary and how to implement these operations efficiently in hardware; this will be the main objective of the next few lectures.

(Refer Slide Time: 03:43)

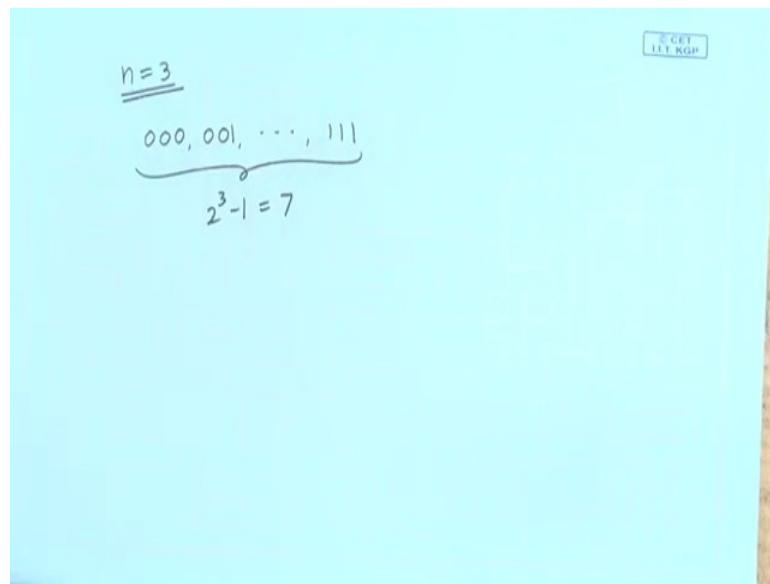
Representation of Integers

- Unsigned integer number representation
 - For n-bit binary, range is 0 to (2^n-1) .
- Signed integer number representation
 - For n-bit 1's complement, range is $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$.
 - For n-bit 2's complement, range is -2^{n-1} to $+(2^{n-1}-1)$.
 - For both the representations, subtraction can be done using addition.
 - 2's complement representation is most widely used.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, we make a quick recapitulation of the representation of integers. You may recall that with respect to unsigned number representation. Let us say we are representing integers. So, in n-bits we can represent numbers from 0 up to $2^n - 1$.

(Refer Slide Time: 04:10)



So, let us say suppose $n = 3$. So, for $n = 3$ you can represent numbers starting from 0 0 0 up to 1 1 1. So, there are a total of $2^3 - 1$ or 7 possible combinations.

This will be the range of the numbers, but when you talk about signed representation as it said there are several alternative methods out of them 1's complement and 2's complement are the most common. For 1's compliment representation in n -bits the range of the numbers that can represented is $(-2^{n-1} - 1)$ to $(2^n - 1)$, but for 2's complement representation when on the negative side we can represent 1 extra digit number extra number $(-2^{n-1} - 1)$.

The reason is that for 1's compliment representation there are 2 alternate representations of 0 this we have already seen earlier, but in 2's complement form 0 has a unique representation. The main advantage of this 1's and 2's complement forms is that we really do not need a subtract circuit in our ALU in both these representation subtraction can be done using addition alone, but we will see for reasons that we will be discussing that out of these two methods, again 2's complement representation has a distinct advantage and therefore, it is most widely used.

(Refer Slide Time: 06:03)

Subtraction Using Addition :: 1's Complement

- How to compute $A - B$?
 - Compute the 1's complement of B (say, B_1).
 - Compute $R = A + B_1$
 - If a carry is obtained after addition is '1':
 - Add the carry back to R (called *end-around carry*).
 - That is, $R = R + 1$.
 - The result is a positive number.
 - Else
 - The result is negative, and is in 1's complement form in R .

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

So, let us first look at how we can carry out subtraction using 1's complement representation. The idea is fairly simple well if I subtract a number B from A , let us say $A - B$ this is what we are trying to compute what we do we first compute the 1's complement of B let us call it B_1 the 1's complement of B , then we add this 1's complement of B to A ; that means, what we calculating A plus B_1 .

Now after this addition if we find that there is a final carry out if the carry of 1 is coming out then what we do we make a correction, we take this carry back and add this 1 to R . That means, we are effectively doing R equal to R plus 1 as a corrective step whenever there is a carry of 1 coming out and in this situation the final result will be a positive number, but; however, if there is no carry coming out this will imply that the result is negative and it is already in 1's complement form in this variable or register R . Let us see an example let us say we are trying to subtract 2 from 6 in 4 bit representation.

(Refer Slide Time: 07:31)

Example 1 :: $6 - 2$

1's complement of 2 = 1101

$\begin{array}{r} 6 :: 0110 \\ -2 :: 1101 \\ \hline 1\ 0011 \end{array}$ <p>End-around carry → 1 0100 = +4</p>	<p>Assume 4-bit representations. Since there is a carry, it is added back to the result. The result is positive.</p>
--	--



IIT Kharagpur Spring Semester | NPTEL ONLINE CERTIFICATION COURSES Programming and Data Structures | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA



So, 6 in 4 bit is 0110 and 2 is 0010. So, the 1's complement will be 1101; so we are actually adding the 1's complement of 2; that means, minus 2 to 6 directly. So, 0 and 1 is 1 no carry 1 and 0 is 1, 1 and 1 is 0 with a carry of 1, 1 and 1 is 0 with a carry of 1. So, there is a final carry out. So, what we are saying is that this final carry out we are bringing it back and adding to this intermediate sum. So, 0 0 1 1 plus 1 this becomes 0 1 0 0 this is the final result plus 4. This is the how subtraction is carried out when the result is positive.

(Refer Slide Time: 08:34)

Example 2 :: $3 - 5$

1's complement of 5 = 1010

$\begin{array}{r} 3 :: 0011 \\ -5 :: 1010 \\ \hline 1101 \end{array} = -2$	<p>Assume 4-bit representations. Since there is no carry, the result is negative. 1101 is the 1's complement of 0010, that is, it represents -2.</p>
--	--



IIT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES Programming and Data Structures | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us take another example where the result is supposed to be negative 3 minus 5. So, 3 is 0 0 1 1 and the 1's complement of 5 is 1 0 1 0 you just add them up 0 1 is 1, 1 1 is 0 with a carry of 1, 0 0 1 is 1, and 1 0 is 1 but no carry. So, when there is no carry, our first conclusion is that the result is negative and whatever is remaining here this will be the result in 1's complement form, you recall 1 1 0 1 in 1's compliment representation of the number -2.

(Refer Slide Time: 09:25)

Subtraction Using Addition :: 2's Complement

- How to compute $A - B$?
 - Compute the 2's complement of B (say, B_2).
 - Compute $R = A + B_2$
 - If a carry is obtained after addition is '1':
 - Ignore the carry.
 - The result is a positive number.
 - Else
 - The result is negative, and is in 2's complement form in R.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, this is what has been summarized here and what we get here. Now let us look at the 2's complement representation. So, in 2's complement also the idea is fairly similar. So, when you are trying to subtract the number B from A, we take the 2's complement of B.

So, you recall the 2's complement of order of a number is the 1's complement plus 1 you add an additional 1 to the number to get the 2's complement form. So, in this case this number which you are trying to subtract you take the 2's complement of that number. So, let us call it B_2 and we add B_2 to a this is how we carry out subtraction, and after this addition step if you see that a carry is coming out simply ignore the carry and your conclusion is that the result is a positive number, but if there is a no carry your conclusion will be the result is negative and it is already in 2's complement form.

So, you see here in 2's complement form that additional corrective step is not required that end around carry you are bringing it back and adding to the partial sum that step is not required here.

In a sense 2's complement representation is more efficient in terms of the effort of calculation. So, when you are talking about implementing in hardware this will also be directly responsible for the decision that will be taking that 2's complement will get better than 1's complement. So, let us take some examples again here.

(Refer Slide Time: 11:11)

Example 1 :: 6 - 2

2's complement of 2 = $1101 + 1 = 1110$

$6 :: 0110$ $-2 :: 1110$ \hline $10100 = +4$	Assume 4-bit representations. Presence of carry indicates that the result is positive. No need to add the end-around carry like in 1's complement.
--	--

Ignore carry

Let us take this $6 - 2$. So, for 2 the 1's complement is 1 1 0 1 you add 1 to it you get the 2's complement it is 1 1 1 0. You simply add 6 and -2 0 0 is 0 1 1 is 0 with a carry of 1, 1 1 is 1 with a carry of 1, 1 0 1 is 0 with a carry of 1. So, there is a carry coming out you simply ignore this carry and 0 1 0 0 whatever remains that is your final result plus 4 as I said here you do not need to add the end around carry.

(Refer Slide Time: 12:00)

Example 2 :: 3 - 5

2's complement of 5 = $1010 + 1 = 1011$

3 :: 0011
-5 :: 1011
 $\underline{1110} = -2$

Assume 4-bit representations.
Since there is no carry, the result is negative.
1110 is the 2's complement of 0010, that is, it represents -2.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

When the result is supposed to be negative like the same example 3 - 5 you take the 2's complement of 5 this is the 1's complement plus 1, this is the 2's complement you add them up 1 1 is 0 with a carry of 1, 1 1 1 is 1 with a carry of 1, 1 0 0 is 1, 0 1 is 1 no carry. So, your conclusion is that your result is negative and 1 1 1 0 is the 2's complement representation of the result. Now you can check 1 1 1 0 is nothing, but minus 2. So, you are getting the correct result.

So, the idea is that for 2's complement representation whether you are adding a larger number from a smaller number or a smaller number from a larger number your addition mechanism is identical. Subtraction or an addition are no different when you want to subtract you represent a negative number in 2's complement form and you add it. So, you only carry out addition no subtraction is required right.

(Refer Slide Time: 13:07)

Addition of Two Binary Digits (Bits)

- When two bits A and B are added, a sum (S) and carry (C) are generated as per the following truth table:

Inputs		Outputs	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$\begin{array}{l} 0+0=00 \\ 0+1=01 \\ 1+0=01 \\ 1+1=10 \end{array}$$

HALF ADDER

$$\begin{aligned} S &= A'B + A'B' = A \oplus B \\ C &= AB \end{aligned}$$

JIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

So, when we talk about addition of 2 binary digits let us say. So, when you are adding 2 bits A and B we generate a sum S and a carry C as per the truth table that is shown here. If the input bits are 0 0 this sum is 0 as well as carry is 0, if they are 0 1 or 1 0 then there will be a sum of 1, but no carry, but if they are both 1 and 1 then sum will be 0 and carry will be 1. So, this is the rule as I said sum plus these two numbers A and B this will be generating a sum and a carry. So, this is actually shown here 0 0 0 1 in that order first carry on then sum 0 1 and 1 0. So, this kind of a circuit which implements this truth table is called a half adder, and in a block diagram form we can show it like this A and B are the 2 inputs and this sum and carry are the 2 outputs.

So, if you just write down the expression from this truth table expression for sum will be $AB' + A'B$ this is nothing, but the exclusive or of A and B($A \oplus B$). and C will be 1 when AB both are 1 that is AB . So, in terms of implementation a half adder will consist of an exclusive OR gate and a AND gate.

(Refer Slide Time: 14:50)

Addition of Multi-bit Binary Numbers

$\begin{array}{r} 0010110 \\ 0101011 \\ + 0001001 \\ \hline 0110100 \end{array}$	$\begin{array}{r} 1111110 \\ 0111111 \\ + 0000001 \\ \hline 1000000 \end{array}$
Carry Number A Number B Sum S	Carry Number A Number B Sum S

- At every bit position (stage), we require to add 3 bits:
 - 1 bit for number A
 - 1 bit for number B
 - 1 carry bit coming from the previous stage

WE NEED A FULL ADDER

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now, when we talk about addition of multi bit numbers normally we do parallel addition like this is a number this is another number. So, when you add them we start from the least significant bit. So, initially the carry input is 0. So, this row shows the carry. This 1 will result in a sum of 0 with a carry of 1 this one 1 1 and 0 added again sum is 0 and carry of 1. 1 0 0 will give a sum of 1 no carry. 0 1 1 will generate sum of 0 carry of 1, 1 0 0 will give 1 no carry this again 1 no carry 0 0 0 0.

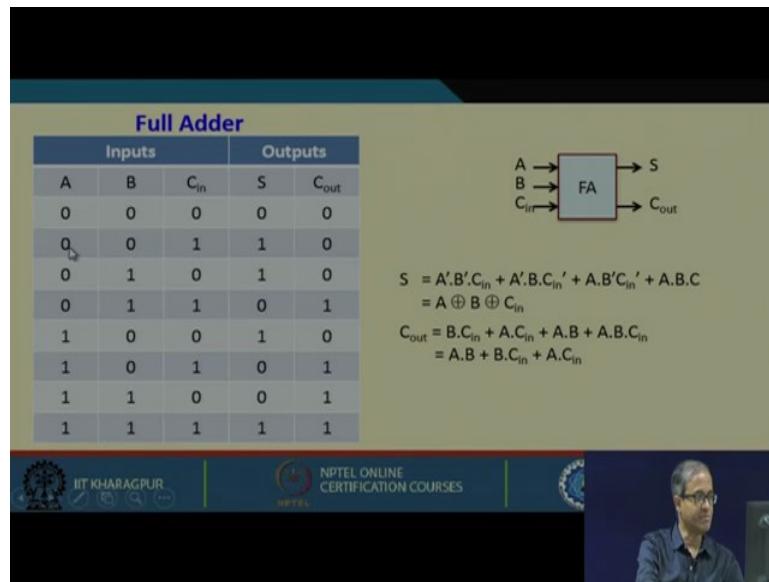
So, you get the sum you take another example here where this is an extreme case of something that I am trying to demonstrate. So, the first number is like this 0 1 1 1 1 and the second number is only a 1 less zeros. So, you add the first two binary digits get a 0 with a carry of 1, 1 and 1 will be 0 with a carry of 1, the same thing is repeating 0 with a carry of 1 0 with a carry of 1 and finally, this carry will be generating the last bit of sum.

So, what I tend to illustrate in this example is that the carry is propagating from the least significant stage to the most significant stage. After adding the first digit there is a carry generated, this carry again generates the carry, this carry again generates a carry. So, there is a rippling effect of the carry from one stage to the other. So, it is something like that from the least significant stage to the most significant stage the carry will be moving stage by stage in that fashion.

So, you will have to wait until all the carry has rippled through the different stages of addition and finally, after that you get the result. So, the observation here is that for

adding multi bit number for every stage or bit position we need to add three digits, two digits for the numbers A and B and one digit from the carry which is being generated from the previous stage. So, essentially we require to add three bits, one bit for A, one bit for B and one carry bit this kind of an adder which adds three bits is called a full adder.

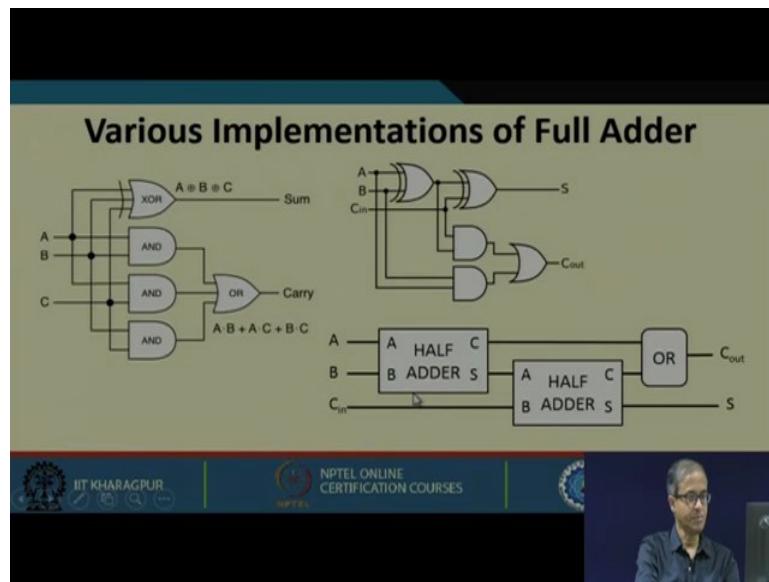
(Refer Slide Time: 17:34)



Let us now see how a full adder looks like. This is the truth table of a full adder where I have 3 input bits A and B and sum and Cout output bits. So, in inputs there can be 8 combinations. So, if it is 0 0 0 sum will be 0 carry will be 0; for 0 0 1 sum will be 1 carry will be 0; same for 0 1 0, but if that two of them are 1's then sum will be 0 and carry will be 1; similarly and for the last case 1 1 1, sum will be 1 carry will also be 1.

A full adder can be conceptually in a block diagram form expressed like these 3 inputs and 2 outputs, and this sum and carry expressions can be written like this. Sum expression if you just consider the output of the sum is 1 these 4 terms and the main terms will be $A' B' C_{in}$, $A'BC_{in}'$, $AB'C_{in}'$ and ABC . So, if you do a simplification of these, you will find that this is nothing, but the exclusive or of A B and C ($A \oplus B \oplus C$). Similarly for C_{out} as the four places where it is 1 it will be $A'BC_{in}$, $AB'C_{in}$, ABC_{in}' and last one is ABC . So, this again if you minimize it will be just AB or BC or AC ($AB+BC+CA$). So, these are the final expression for the sum and the carry out .

(Refer Slide Time: 19:15)



So, when you implement full adder there are various ways of implementing this circuit the first circuit directly implements this function. S is the exclusive or and C is $AB + BC + AC$. So, it directly implements the C_{in} . C is the C_{in} .

So, an exclusive-or to generate the sum, and a 2-level and-or network to generate the carry; now there is some scope for optimization if you break this 3 input XOR into 2 input XOR gates and if you take the output from the first XOR gate to this 2 level and or circuit here also you can generate C_{out} and your circuit complexity becomes a little less this is also an alternate design. And the third diagram that I am showing is that if you have half adders as your building blocks you can combine 2 half adders plus AND OR gate to design of full adder these are the alternate designs.

(Refer Slide Time: 20:22)

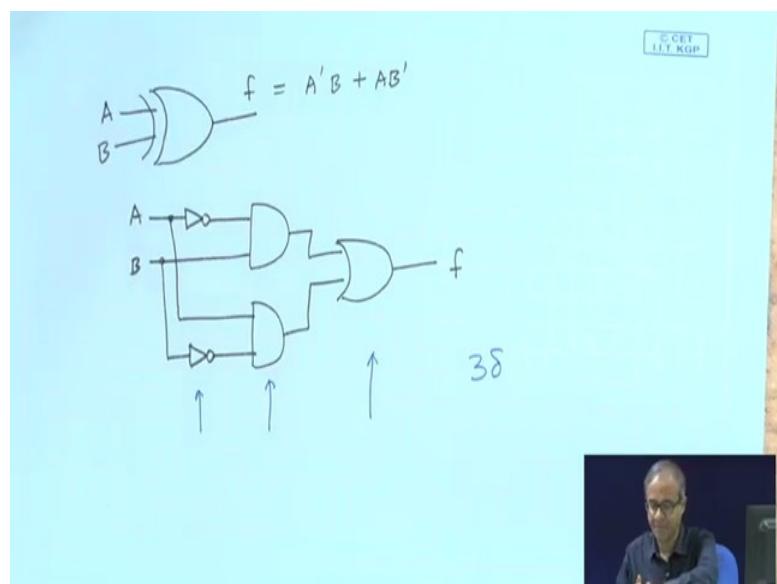
- Delay of a full adder:
 - Assume that the delay of all basic gates (AND, OR, NAND, NOR, NOT) is δ .
 - Delay for Carry = 2δ
 - Delay for Sum = 3δ
(AND-OR delay plus one inverter delay)

The logic diagram shows a full adder with three inputs A, B, and C. The sum output is labeled $A \oplus B \oplus C$, and the carry output is labeled $A \cdot B + A \cdot C + B \cdot C$. The diagram uses AND and OR gates to implement the sum and carry logic.

Logos at the bottom: IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, let us look at the design like this let us make a simple delay analysis. So, we assume that the delay of the basic gates as 1. So, what are the basic gates? Let us assume AND, OR and NOT are the basic gates.

(Refer Slide Time: 20:47)



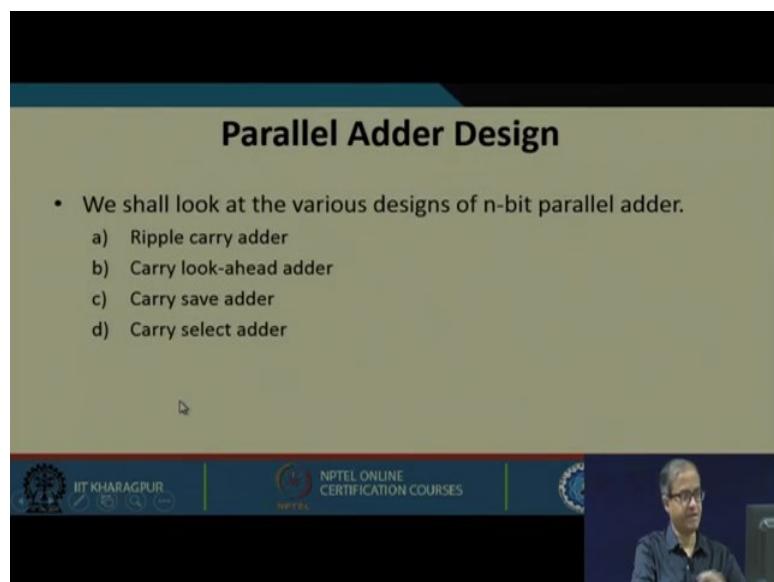
But what about XOR gate let us look at a scenario of a 2 input XOR gate. Let us say I have 2 inputs A and B and an output of f. So, what does f represent $A'B$ or AB' bar. So, if I can implement it using 2 level AND OR this will be my f. The first input will be fed with A' B. So, if this is A there will be a NOT gate to generate A' and then AND with B.

Second one will be AB' . A will be fed as it is and there will be B' . So, here if you look from the input to the output you will see that there are 3 level of gates one is to negate the gates then these AND gates and then these OR gate.

So, the equivalent delay of a 2 input XOR will be thrice delta if delta is the delay of a basic gate. This is what is mentioned here that for the XOR gate for this sum the delay will be thrice delta and for the carry this circuit I am already showing here it will be 2 level gate delay.

Student: (Refer Time: 22:18).

(Refer Slide Time: 22:16)



Now let us see how we can design a parallel adder. Parallel adder means we are trying to add more than 1 bits together. For n-bit numbers we have already seen how we can this. The same concept you can extend in the first adder design that we shall be seeing there are several main types of adders that we will be exploring, ripple, carry look ahead, carry save, carry select adders etc.

(Refer Slide Time: 22:56)

Ripple Carry Adder

- Cascade n full adders to create a n- bit parallel adder.
- Carry output from stage-i propagates as the carry input to stage-(i+1).
- In the worst-case, carry ripples through all the stages.

$$\begin{array}{r} \textcolor{red}{1} \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 & \text{Carry} \\ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 & \text{Number A} \\ + \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 & \text{Number B} \\ \hline 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 & \text{Sum S} \end{array}$$

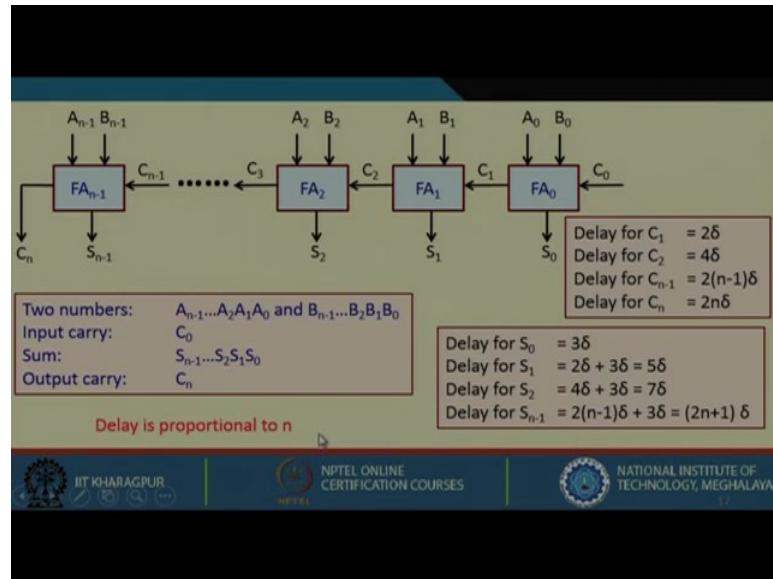
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

But let us look at it 1 by 1; ripple carry adder is the simplest which models the way we do addition using a paper and pencil method, just to recall again you take this example once more. So, the 2 numbers A and B are given we add the least significant digit of the numbers with an initial carry of 0 no carriers there you get a sum.

Student: (Refer Time: 23:24).

You get a carry in the next stage you again add these 3 digits, you get a sum and a carry you again add these 3 digits and this way you repeat. So, if there are n digits in my binary number, we require n such adders.

(Refer Slide Time: 23:48)



Let us see this n - full adder stages will be like this.

Student: (Refer Time: 23:54).

So, my input numbers are fed bitwise in this form is A_0, B_0 to the least significant stage a A_1, B_1 here a A_2, B_2 here and A_{n-1} and B_{n-1} will here; and as you can see that the carry out of the first stage the is going as a carry input to the next stage, carry out of this stage is going as a carry input to next stage and so on. This is a parallel adder that I have designed just using a cascade of several full adders. So, I have several full adders and every full adder is capable of adding 3 bits. So, I am adding 3 bits like that and there will be 1 full adder at every stage which will be handling the addition of those 3 bits; and when the addition is carried out the full adder is generating a sum and it will be generating the carry for the next stage.

So, the example that I have shown in the worst case the carry might be propagating from the least significant stage to the most significant stage for ripple carry adder circuit. So, what we assumed is that the two numbers are A and B both are n -bit numbers the input carry is C_0 this is the carry in of the last stage, sum is S_0 to S_{n-1} for n -bits and there will be a carry out C_n .

Now if you want to calculate the delay it is fairly simple. See for a full adder we have already calculated the delay, and we have seen that the delay for the carry out is twice

delta; so for the first full adder when I apply A_0 B_0 and C_0 . So, after 2 delta C_1 will be generated. So, it is only after 2 deltas in the worst case this A_1 B_1 C_1 will be available. So, you wait for another 2 delta you will be getting C_2 . So, for C_2 will be getting at four delta C_3 will be getting at 6 delta and so on.

C_{n-1} ; so if you extend it will be after $(2n - 1)x\delta$, and the final carryout will be generated after $2n\delta$. So, for generating the carries your maximum delay is $2n\delta$; let us now look at this sum. See for sum as you recall it is an XOR gate. So, delay will be 3δ . So, S_0 will be generated in 3δ . Let us look at S_1 , this carry C_1 is generated at 2δ time. So, starting from 2δ this will require another 3δ , that will be 5δ , C_2 is generated at 4δ time. So, starting at 4δ you take another 3δ for FA_2 . So, S_2 will be generated at $4\delta + 3\delta = 7\delta$ time. Look at the last stage; C_{n-1} is generated at this time $(2n - 1)\delta$ so that plus another 3δ , which is $(2n + 1)\delta$.

So, you see out of these $(2n + 1)$ delta is larger. So, the worst-case delay of the ripple carry adder is $(2n + 1)$ delta. This grows linearly as the value of n increases and this is mainly due to the ripple effect of the carry. So, I have already shown the worst-case scenario that when you apply an input carry the worst case the input carry will be rippling through all the stages and to the final output fine.

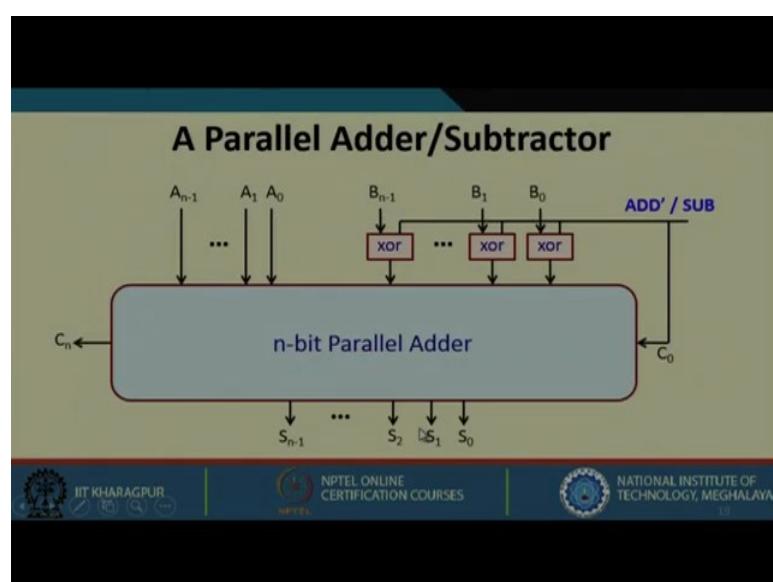
(Refer Slide Time: 28:09)

How to Design a Parallel Subtractor?

So, this is how a ripple carry adder works; delay is proportional to n . So, how could we design a parallel subtractor, I have already shown you that how to design a parallel adder. Now already we have seen how 2's complement subtraction is done you take this 2's complement of the number you want to subtract and then add it. So, it is fairly simple you see I am just showing you a schematic diagram let us say for every bit of this number B , let x_i denote B_i' . So, what we do we take our normal ripple carry adder just like we have shown?

The first number A we apply as usual; second number instead of B you apply this x . So, what is x ? X is essentially the 1's complement of B --- you are complementing or doing a NOT of all the bits, and another change you are doing, carry input see in the initial cases this carry input initially would be 0, but here we are saying that the carry input is set to 1. So, essentially I am applying 1's complement and adding a carry input of 1 which effectively means I am adding a 2's complement number. I am doing a 1's complement plus another extra 1 that makes it 2's complement. So, what this adder will be computing is nothing, but the sum of A and the 2's complement of B right. So, what will be getting in the final sum output will be the final result A minus B . So, how you can design a general adder - subtractor?

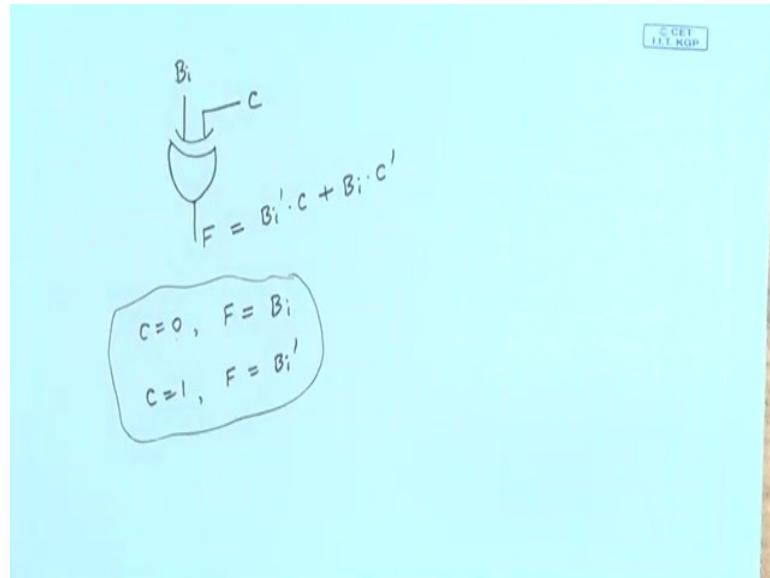
(Refer Slide Time: 29:52)



Then let us have a circuit like this we have a n -bit adder I apply the first number here A_0 to A_{n-1} . Well I apply the second number not directly, but through a sequence of XOR

gates. So, how are these XOR gates connected? One of the input to the XOR gate is connected directly from an external control input add/sub, and the other input the second input of this XOR gates are fed with the bits of the second number B0 B1 Bn-1.

(Refer Slide Time: 30:40)



Suppose I have an XOR gate. Let us say I have 1 input as let us say B_i , I have a second input C . If this is F , let us say if C is 0, then you can easily check F will be same as B_i ; if C is 1, F will be same as B_i' . Because if you write down the exclusive or function it will be immediately clear you put $C = 0$, B_i will remain; you put $C = 1$, B_i' will remain. So, you can use this XOR gate as a controlled inverter; if C is 0 no inversion, if C is 1 then there will be NOT. So, essentially here this XOR gates are being used as a controlled inverter, if this control signal is equal to 1 which means the 1's complement of this number B will be fed into this input of the adder, and this C_0 will also be 1 same input is will fed here.

So, now, we will be doing a subtraction, but if this control input is 0, so the carry input is 0 as well as B will be directly coming to the input of the adder. So, now, it will be acting as an adder. So, this simple circuit just by the addition of a few exclusive OR gates we can have a combination where you can implement both an adder and a subtractor using this same hardware circuit.

We shall see some adder designs which are faster than the ripple carry adder that we have already seen in our next lecture, it means we have also seen one thing that how we

can combine addition and subtraction. For subtraction we really do not need a separate piece of hardware we can use an adder directly to carry out subtraction as well. So, this was the main topic of our discussion in this lecture.

So, with this we come to the end of this lecture as I said in our next lecture we would be looking at some other kind of adder designs, which are in some sense better or faster than the ripple carry adder that we have seen.

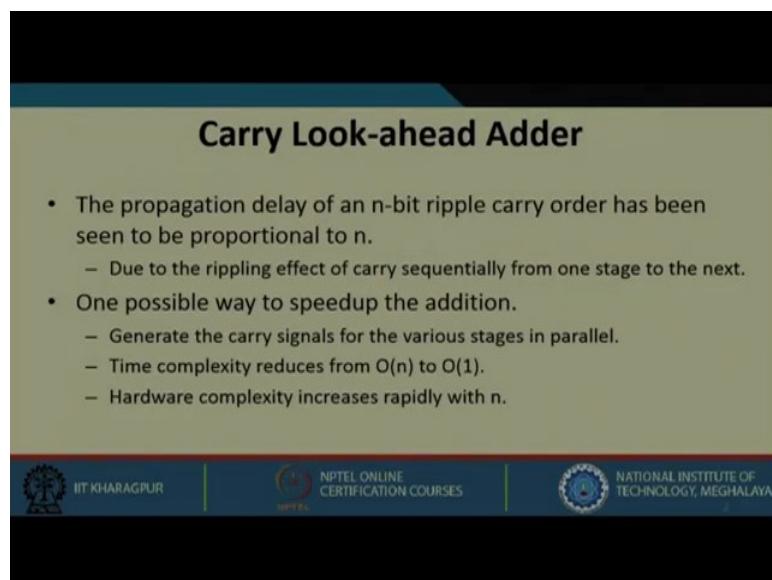
Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 34
Design of Adders (Part II)

We continue with our discussion on Adders. If you recall in our last lecture we looked at one kind of adder the so called ripple carry adder. And in the ripple carry adder have observed one very distinct characteristic; that in the worst case the carry input ripples through all the stage up to the final carry output, this results in an overall worst case delay of the adder which is proportional to the number of bits or the number of stages.

(Refer Slide Time: 00:59)



Carry Look-ahead Adder

- The propagation delay of an n-bit ripple carry order has been seen to be proportional to n.
 - Due to the rippling effect of carry sequentially from one stage to the next.
- One possible way to speedup the addition.
 - Generate the carry signals for the various stages in parallel.
 - Time complexity reduces from $O(n)$ to $O(1)$.
 - Hardware complexity increases rapidly with n.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, we continue with our discussion. So, this is what I just now mentioned that the ripple carry adder that we have seen so far the total propagation delay in the worst case has been seen to be proportional to n and the primary reason was the rippling effect.

Suppose I consider the full adder at any particular stage, now that full adder is unable to start the addition until or unless its carry input is available. Now in the ripple carry adder this carry input is coming sequentially through a rippling effect, because of that the different full adders was having to wait till the carry input comes. So, this second approach is another kind of adder that we look here is called carry look ahead adder. So,

as the name implies we are doing some kind of a look ahead to determine the values of the carry.

If you can do that, then the total time complexity of the adder can reduce from order n which is roughly proportional to the number of stages to order 1 which means addition can be done in constant time independent of the number of stages, but this is possible if somehow we can generate the carry bits in parallel. If you can do that then all the full adders can perform their addition simultaneously. Here there is no need to wait for the previous full adder to complete; this is the basic idea behind the carry look ahead adder. But the only trouble here is that the hardware complexity the amount of gates that will be required will increase rapidly with the value of n. This is the drawback of carry look ahead adder.

(Refer Slide Time: 03:18)

- Consider the i-th stage in the addition process.
- We define the *carry generate* and *carry propagate* functions as:

$$G_i = A_i \cdot B_i$$

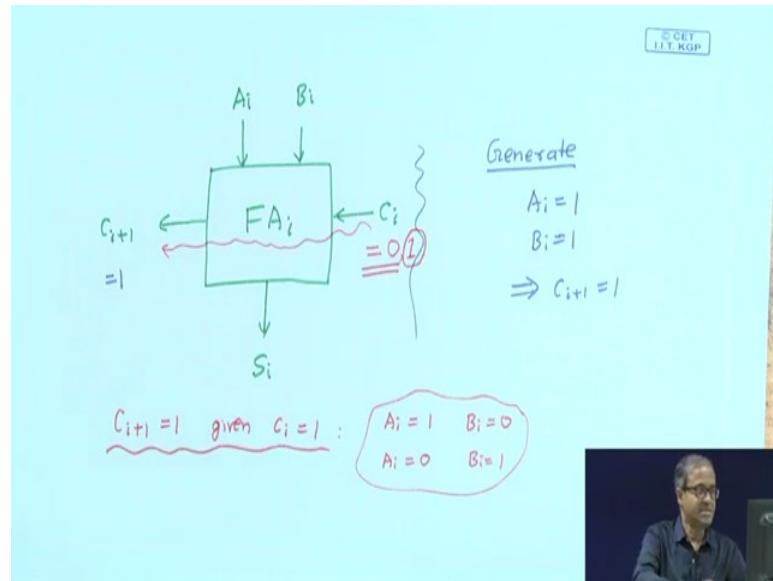
$$P_i = A_i \oplus B_i$$
- $G_i = 1$ represents the condition when a carry is generated in stage-i independent of the other stages.
- $P_i = 1$ represents the condition when an input carry C_i will be propagated to the output carry C_{i+1} .

$$C_{i+1} = G_i + P_i \cdot C_i$$

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

So, let us see the basic idea, we look at a full adder in terms of how it works let us consider the full adder in the i-th stage of a ripple carry adder say where the inputs are A_i , B_i and C_i and the outputs are S_i and C_{i+1} . Now we define 2 different kinds of events one we call as the carry generate, other we call the carry propagate. The idea is similar to full adder.

(Refer Slide Time: 04:01)



Let us consider full adder at stage i . So, the inputs are A_i and B_i , sum is S_i and input carry is C_i and the output carry is C_{i+1} . Carry generate means that this full adder will be generating a carry; meaning C_{i+1} will be 1 independent of what has happened in the previous stage.

So, I do not care what the value of C_i is; C_i can be 0, C_i can be 1 whatever it depends on the previous stages, but this A_i and B_i are the present inputs from the numbers which are coming. So, we can say if A_i is 1 and B_i is 1 then definitely there will be a carry out this will imply C_{i+1} will be 1. This is the scenario for carry generate a carry is generating in the i -th stage itself irrespective of what has happened in the previous stage. So, you see the generate function G_i is defined as A_i and B_i .

The second scenario is called carry propagate. Suppose my input carry has come as 1. So, my question is under what condition this input carry will be propagating to the output; that means, C_{i+1} will be equal to 1 given C_i equal to 1 what are the other conditions for that? This carry will be propagated under the condition when A_i is 1, B_i is 0 or A_i is 0, B_i is 1 because you add this 1 to 1 and 0 it will generate a carry you add this 1 to 0 and 1 it will also generate a carry. So, this condition actually talks about the exclusive OR function.

So, the carry propagate function is defined as $A_i \oplus B_i$. So, this is what I have explained so far G_i equal to 1 represents the condition when a carry is generated in this stage itself

independent of the previous stages, and propagate function will be 1 when an input carry C_i will be propagated to the output carry C_{i+1} . Now with respect to this G_i and P_i sub functions we can write down an expression for C_{i+1} , we can write the carry output will be 1 if either a carry is generated or the propagate function is true and there was an input carry. So, if either of these two conditions are true then C_{i+1} will be 1. So, I can simply write down the expression for this C_{i+1} in terms of G_i and P_i like this.

(Refer Slide Time: 08:29)

Unrolling the Recurrence

$$\begin{aligned}
 C_{i+1} &= G_i + P_i C_i = G_i + P_i (G_{i-1} + P_{i-1} C_{i-1}) = G_i + P_i G_{i-1} + P_i P_{i-1} C_{i-1} \\
 &= G_i + P_i G_{i-1} + P_i P_{i-1} (G_{i-2} + P_{i-2} C_{i-2}) \\
 &= G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + P_i P_{i-1} P_{i-2} C_{i-2} = \dots
 \end{aligned}$$

$$C_{i+1} = G_i + \sum_{k=0}^{i-1} G_k \prod_{j=k+1}^i P_j + C_0 \prod_{j=0}^i P_j$$

Now, this is like a recurrence relation, I am defining the carry of $(i+1)$ in terms of the input i . So, I can recursively expand this function like what I can write is C_{i+1} is $G_i + P_i C_i$ in a similar way this C_i we can expand as $(G_{i-1} + P_{i-1} C_{i-1})$, again I have a C_{i-1} , expand the C_{i-1} as G_{i-2} , plus $P_{i-2} C_{i-2}$ multiply them out you continue this multiplication process till you reach C_0 . So, in the last term where there is C_0 will be containing the product of all the p 's.

And there will be several other terms with G_{i-1} , there will be a single P_i , G_{i-2} there will be 2 P_i 's, G_{i-3} there will be 3 P_i 's and so on. So, it can be expressed in the form of expression like this. So, if you work this out you will see that this expression and this generalization they mean the same thing. So, actually I am working this out for a specific case for a 4 bit adder.

(Refer Slide Time: 10:06)

Design of 4-bit CLA Adder

$C_4 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3$	4 AND2 gates
$C_3 = G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2$	3 AND3 gates
$C_2 = G_1 + G_0 P_1 + C_0 P_0 P_1$	2 AND4 gates
$C_1 = G_0 + C_0 P_0$	1 AND5 gate
$S_0 = A_0 \oplus B_0 \oplus C_0 = P_0 \oplus C_0$	1 OR2, 1 OR3, 1 OR4
$S_1 = P_1 \oplus C_1$	and 1 OR5 gate
$S_2 = P_2 \oplus C_2$	4 XOR2 gates
$S_3 = P_3 \oplus C_3$	

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

A video of a professor speaking is visible on the right side of the slide.

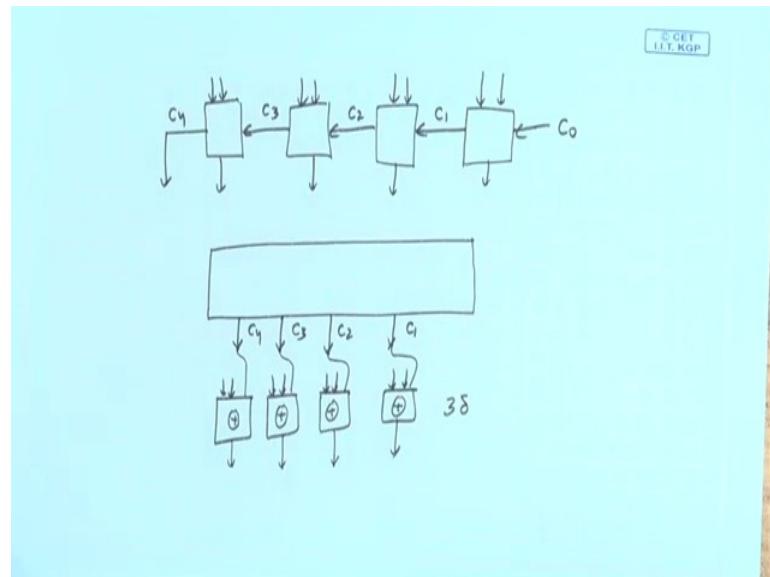
Let us do it. So, let us consider a 4 bit carry look ahead adder where using the same expressions whatever we have written down here. So, using the same expression C_4 can be written as G_3 plus $G_2 P_3$, $G_1 P_2 P_3$, $G_0 P_1 P_2 P_3$ and $C_0 P_0 P_1 P_2 P_3$ this actually follows from this way expression this you can check C_3 can be similarly expressed like this, C_2 can be expressed like this, C_1 can be expressed like this and some for a full adder is nothing.

But the \oplus XOR gate of the 3 inputs now already the carry propagate function we have defined carry propagate function is nothing, but the XOR of the inputs. So, this $A_0 \oplus B_0$ I can replace by P_0 similarly S_1 I can write $P_1 \oplus C_1$ S_2 I can write $P_2 \oplus C_2$ and so on. So, I have derived some expressions for C_1 C_2 C_3 C_4 in terms of the carry generate and carry propagate. Suppose I want to implement this how many gates I will required just check. So, how many 2 input and gates you record is called n_2 n_2 means 2 input and gates I need 4 the why 4 this is 1 2 input.

And function $G_2 P_3$, $P_1 P_2$, $j_0 P_1$ and $C_0 P_0$ I need 3 3 input and gates worth in here there is 1 here there are 3 terms you have to do end 1 here, 1 here 3 we need 1 5 input and gate this is 1 5 input and gate 1 2 3 4 5 we need 2 4 input and gate 1 here and 1 here and we need 1 2 input or gate for this we need 1 3 input or gate for this 1 4 input or gate and 1 5 input or gate and for this some unit 4 XOR gates now in addition you recall

for generating the carry generate and carry propagate signals you need n XOR gates and n AND gates.

(Refer Slide Time: 12:46)



Now, if you do this what you are actually gaining let us try to understand in normal ripple carry adders that have seen earlier.

So, the inputs were coming carry in was there was this or this some outputs, and they carry outputs were rippling through right. So, if this was your input carry C 0. So, C 1 C 2, C 3 and C 4 now because of the rippling effect of the carry this full adder they had to wait till the carry input was available then only it can do the computation and generate the sum. Now you see what we are saying is that we propose to design a separate combinational circuit that will be directly generating my C 1, C 2, C 3 and C 4 values and then I will be having 4 full adders they will be fed with the ab inputs as usual, and if third carry input can be directly fed from here. So, you see there is no ripple effect between the full adder stages.

So, after this carries are available. So, these you need just an XOR function here just an XOR function; that means, just a 3 delta delay. So, when all your sum will be available and this will be independent of the value of n, this is what we try to show in these expressions. So, if we use these expressions to generate a combinational circuit we can directly generate C 1 C 2 C 3 C 4 and if we can do that we can directly feed those carry inputs. So, that the 4 full adders can directly we do not need the full functions of the full

adder we need just an XOR gates to generate the sums and this can be done in parallel no rippling effect ok.

So, this expression can be simplified you just look at it one thing this sub expressions G 2 G 1 P 2 this 1 is appearing here also if you take P 3 common from these 4 terms you will see that this same sub expressions appearing G 2, G 1 P 2, G 0 P 1 P 2, C 0 P 0 P 1 P 2.

(Refer Slide Time: 15:39)

Design of 4-bit CLA Adder

$C_4 = G_3 + C_3 P_3$ $C_3 = G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2$ $C_2 = G_1 + G_0 P_1 + C_0 P_0 P_1$ $C_1 = G_0 + C_0 P_0$	4 AND2 gates 2 AND3 gates 1 AND4 gates 1-AND5-gate 1 OR2, 1 OR3, 1 OR4 and 1 OR2 gate
$S_0 = A_0 \oplus B_0 \oplus C_0 = P_0 \oplus C_0$ $S_1 = P_1 \oplus C_1$ $S_2 = P_2 \oplus C_2$ $S_3 = P_3 \oplus C_3$	4 XOR2 gates

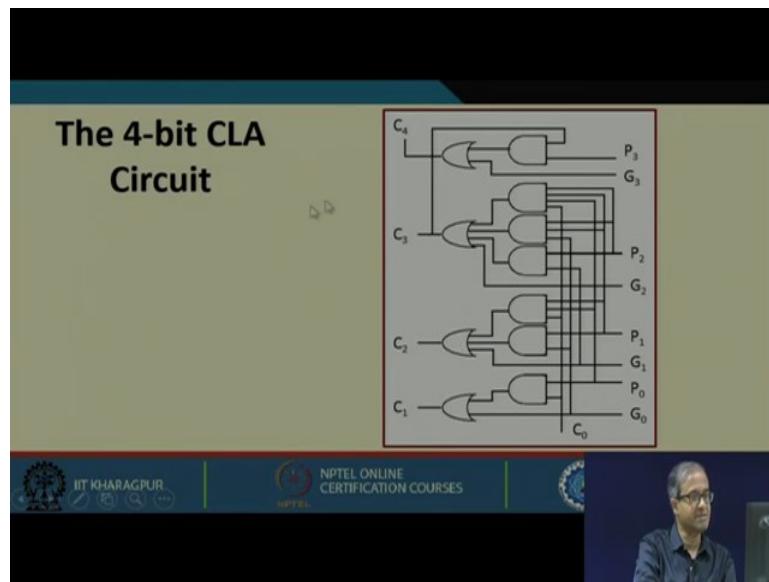



NPTEL ONLINE CERTIFICATION COURSES



So, if you substitutes C 3 here you this expression gets simplified to C 3 P 3 and if you do the simplification the advantage you gain is that see here you needed 3 and 3 gates and 2 and 4 gates here the number of gates are reducing this and 5 gate you do not need and earlier you required some you needed an or 5 gate 5 terms or odd, but now you need A 2 input or gate just ok.

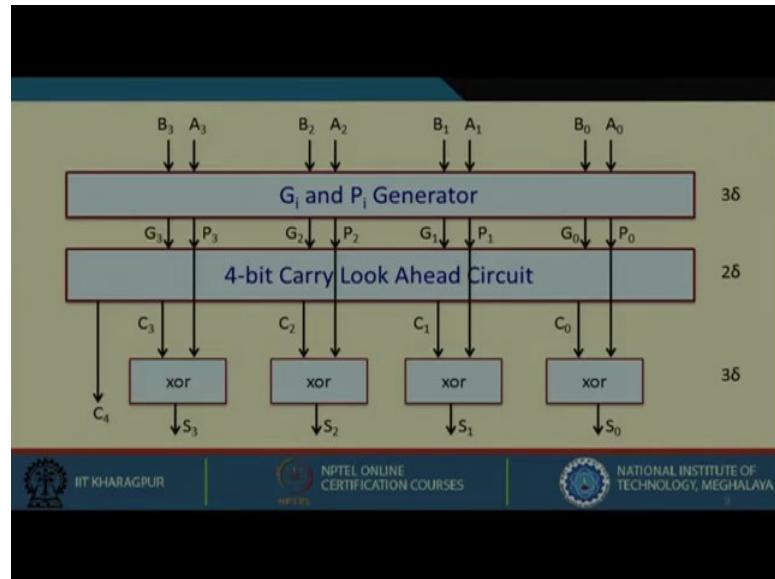
(Refer Slide Time: 16:11)



So, your circuit becomes simpler. So, if you simply realize these functions your circuit will look like this. So, suppose I have my P_i G_i functions, then using P_i G_i just following these expressions you can generate the carries using this kind of a 2 level circuit and you recall that P_i is nothing, but the XOR of the inputs; that means, 1 XOR gate and G_i is nothing, but the end of the input; that means, 1 and gate. So, you need this much hardware, but one problem is that if we look at this recurrence relation as the number of n increases this the complexity of this function $C_1, C_2, C_3, C_4, C_5, C_6$ they will go on increasing.

So, the number of gates will also increase the number of inputs of the gates will also increase. So, it will become increasingly difficult to implement or realize the function. So, you may know that as the number of inputs the size of a gate increases the delay also increases very rapidly. So, you need to control the number of inputs limited to a small value 2 or 3 let say. So, this is one drawback.

(Refer Slide Time: 17:38)



So, the circuit of this 4 bit carry look ahead adder will overall look like this there will be 1 circuit which will be generating G_i and P_i .

(Refer Slide Time: 17:55)

$$\begin{aligned}
 P_i &= A_i \oplus B_i & 3\delta \\
 G_i &= A_i \cdot B_i & \delta
 \end{aligned}
 \quad] \quad 3\delta$$



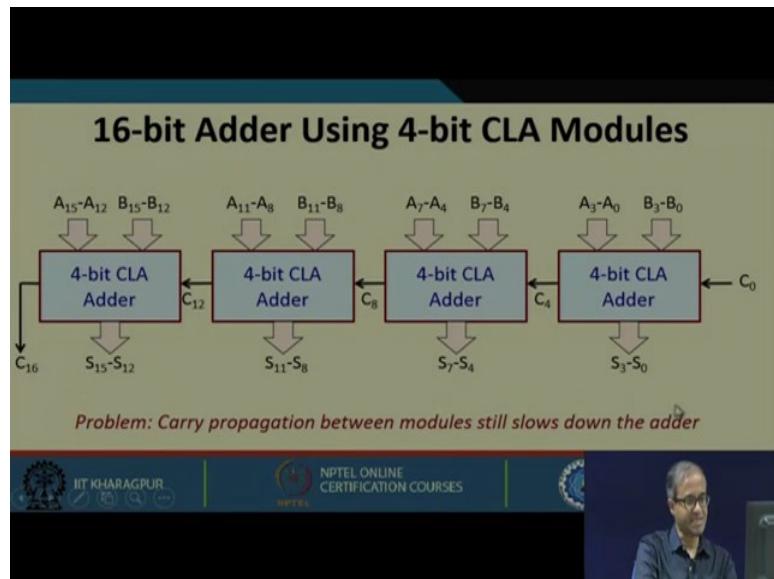
How just as I have said G_i is nothing, but P_i is nothing, but XOR. So, how much delay will take? This XOR takes a delay of thrice delta as I said earlier 3 gate delay.

And gate is a single gate delay. So, the worst case gate delays 3 delta. So, this G_i and P_i generator requires the time of 3 delta to generate all the G_i P_i values right and this for this 4 bit carry look at circuit will look like this you say this is just a 2 level and or circuit

simple 2 level and or circuit, but if you implement this like this the no delay increases, but you implement it in the original form the larger form then toul, but here this C 3 you are bringing it back. So, it becomes worst case delay of 4 for generating C 4, but I am assuming the previous approach where you are not doing that simplification. So, it will be 2 delta, but using the simplification this will become 4 delta right.

And once you have done this after this time all the carry values are available in parallel. So, you simply do an XOR with the P values to generate this sums and this XOR again will require 3 delta. So, you see after a constant time of eight delta you have generate this some bits in parallel. So, that is why we mentioned earlier that we are aiming to reduce the addition time from order n to order one; that means, proportional to n to constant time addition right. So, carry look ahead adder helps us in doing this.

(Refer Slide Time: 19:48)



But 4 bit is too less a number let us say we want to design a 16 bit adder, well having a single 16 bit carry look ahead adder will become impractical as I have just now said; because the expression for the carries as you would go up to the higher stages will become exceedingly complex.

So, the product terms will become larger in sizes number of terms will also become very large requiring a very large or gates. So, 1 alternative may be you can have as a basic building block 4 bit carry look ahead adder modules just like the once you have designed they will be taking eight delta eight delta times for addition, and you change them just

like a ripple counter between these stages. So, this is a mix of carry look ahead and ripple carry adder. So, every 4 bit you can do in parallel, but this C 4 will be generated only after you just recall this carries will be generated only after 3 delta plus 2 delta 5 delta time. So, you will have to wait that time for the carry to be generated similarly C 8 similarly C t12 similarly C 16.

So, unless C 4 is available you cannot start addition here, unless C eight is available you cannot start addition here, but of course, you can do this G i and P i generation in parallel, but for this carry look ahead circuit you have to wait to lambda and only after that carry look ahead circuit C 4 will be generated C 8 will be generated C 12 will be generated right. So, carry propagation between modules is still slowing down the adder here.

(Refer Slide Time: 21:48)

- Solution:
 - Use a second level of carry look-ahead mechanism to generate the input carries to the CLA blocks in parallel.
 - The second level of CLA generates C4, C8, C12 and C16 in parallel with two gate delays (2δ).
 - For larger values of n, more CLA levels can be added.
- Delay calculation of a 16-bit adder:
 - a) For original single-level CLA: 14 δ
 - b) For modified two-level CLA: 10 δ

So, what we propose, we propose that we use a second level of carry look ahead like you see here we have 4 bit carry look ahead modules, and inside this modules we have a circuit like this. So, we already have G i P i generator and carry look ahead circuit.

So, what we are saying is that at a higher level let us have another carry look ahead circuit for which you can derive the equations in a similar way that will be generating C 4, C 8, C 12 and C 16 in parallel in constant time that is quite possible. So, in that case you can have a 2 level carry look ahead adder module this is what we are saying here use a second level of carry look ahead mechanism to generate the input carries for the carry

look ahead blocks in parallel, and this second level block will be generating the carries C 4 C 8 C 12 and 16 together in constant time, but if n is larger maybe you will be requiring more number of levels.

Let us say I need a 64 bit adder, I build 4 bit 4 bit modules 4 such 4 bit modules are combined to build a 16 bit adder, 4 such 16 bit adders with another 4 bit carry look ahead module I can use to build up 64 bit adder that is how so, called fast adders are built in many of the recent computer systems. So, delay calculation is his fairly simple for original single level carry look ahead adder for 16 bit, the delay was 14 delta, but for 14 delta means I am talking of this circuit because this carry generation will take 2 delta additional time because of this carry look ahead time 2 delta because this G i P i can be generated in parallel by only blocks then this 2 2 2 2 this time will come and it will become 14 delta.

If you do a calculation, but if you do this modified form then you need only 10 delta be a faster. So, I am not showing the detail of this 2 level multi-level design the conceptually the idea I mentioned talking of a general k bit adder, I am showing a quick comparison between a ripple carry adder.

(Refer Slide Time: 24:18)

Delay of a k-bit Adder		
n	T _{CLA}	T _{RCA}
4	8δ	9δ
16	10δ	33δ
32	12δ	65δ
64	12δ	129δ
128	14δ	257δ
256	14δ	513δ

$$T_{CLA} = (6 + 2\lceil \log_4 n \rceil) \delta$$

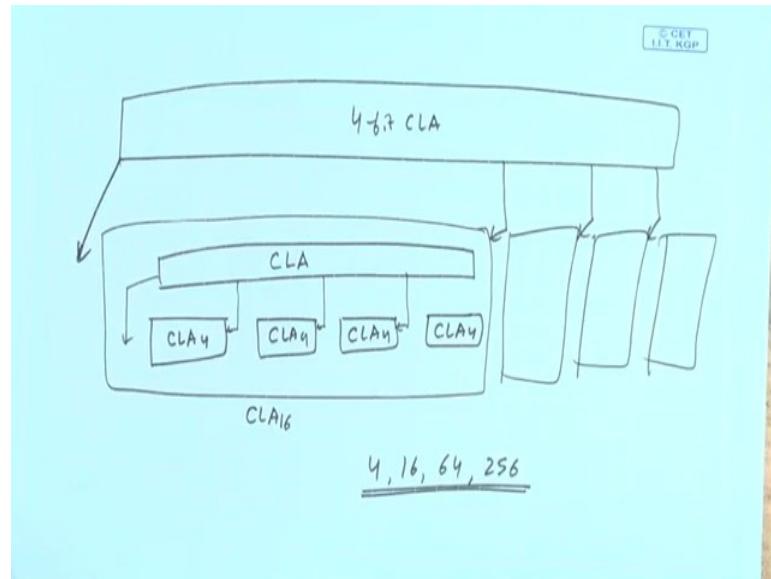
$$T_{RCA} = (2n + 1) \delta$$


IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES



And a carry look ahead adder with this kind of 4 bit 4 bit look ahead. 4 bit look ahead means I am just repeating once more that my idea is like this.

(Refer Slide Time: 24:37)



So, I will be using carry look ahead adder 4 CLA 4 blocks; that means, 4 bit carry look ahead adder. So, I use a higher level carry look ahead module for this will be parallelly generating this carries. So, what I have now a block which I can call as CLA 16 16 bit adder.

So, I can have 4 such blocks 16 bit adders, and above that I can have another higher level 4 bit CLA module. So, these will be generating the carry inputs for these high level adders. So, in multiples of 4 I can do, in 4 bits then 16 bits then 64 bits then 256 bits and so on well means assuming that 4 is a reasonable number beyond 4 it will be too expensive to construct a carry look ahead adder this is our assumption. So, from this calculation the delay for a carry look adder comes like this 6 plus $2 \log n$ to the base 4 sealing of that if you make a simple calculation it will come. The 6 comes 3 for generating P i and G i and 3 for the final XOR to generate the sum, and 2 into this is for the carry generations.

And this will be the number of stages look ahead stages if any 64 for example, then this term is equal to 3. 3 into 6, 6 and 6, 2 12 and in contrast for a ripple carry adder the delay as you know is $2 n$ plus 1 delta. So, if I compare you see therefore, ripple carry adder as n increases the delay increases rapidly, but for carry look ahead adder it does not increase that much appreciably it remains more or less in the same order 10 12 14 fine.

(Refer Slide Time: 27:01)

Carry Select Adder

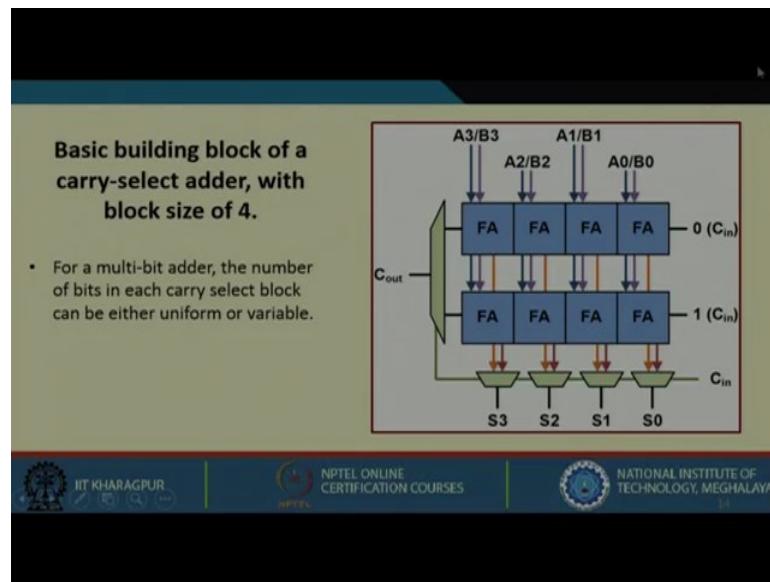
- Basically consists of two parallel adders (say, ripple-carry adder) and a multiplexer.
- For two given numbers A and B, we carry out addition twice:
 - With carry-in as 0
 - With carry-in as 1
- Once the correct carry-in is known, the correct sum is selected by a multiplexer.

The slide footer includes the IIT Kharagpur logo, the NPTEL logo, and the text "NPTEL ONLINE CERTIFICATION COURSES". A video feed of a speaker is visible on the right side of the slide.

Now, let us look at an alternate kind of a adder called carry select adder, this is more like a philosophy which can make or addition faster. Because the hardware is quite inexpensive today here we are using 2 adders for every addition, let us say we use 2 ripple carry adders and in addition we use a multiplexer.

So, why you are using 2 adders? Because the idea is that we do not know the input carry let us say. So, it will take some time for the input carry to be generated. So, I take 2 adders the 2 adders are parallelly adding the numbers first one is assuming that the carry is 0 second 1 is assuming that the carry is one. So, we are generating the 2 sums later on when the carry is known. So, one of the 2 sums can be selected by using a multiplexer only after a multiplexer delay that is quite fast this is the basic idea.

(Refer Slide Time: 28:10)

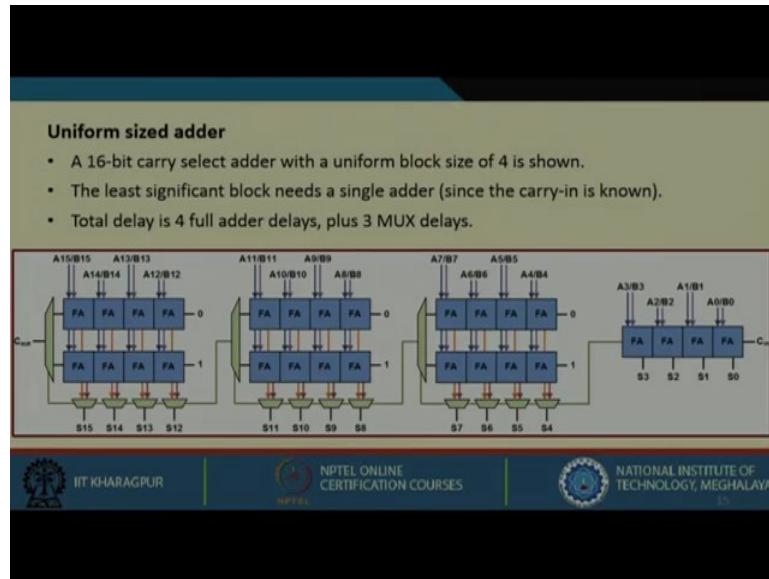


So, I am showing a schematic diagram of a 4 bit can select adder you can see, here I am assuming there is a ripple carry adder kind of a thing 4 full adders there is another adder the first 1 is assuming carry in is 0 second 1 is assuming carry in is one.

So, the 2 fall full adders are generating the sums the first one is this orange this is this brown. So, the 2 additions are fed to a chain of 2 to 1 multiplexers. Later on when the actual value of key you see in gets known by that time the full adder operations are already done. So, you can simply select the multiplexer and take out one of the 2 additions sums that you have calculated similarly the carry out.

So, either you take carry out from here or carry out from here depending on the value of seen. So, this is what carry select adder means basically that you are trying to save time by carrying out 2 additions bone with a carry in of 0 other with a carry in of 1 and then later on you select one of the 2 sums based on the actual carry which is generated.

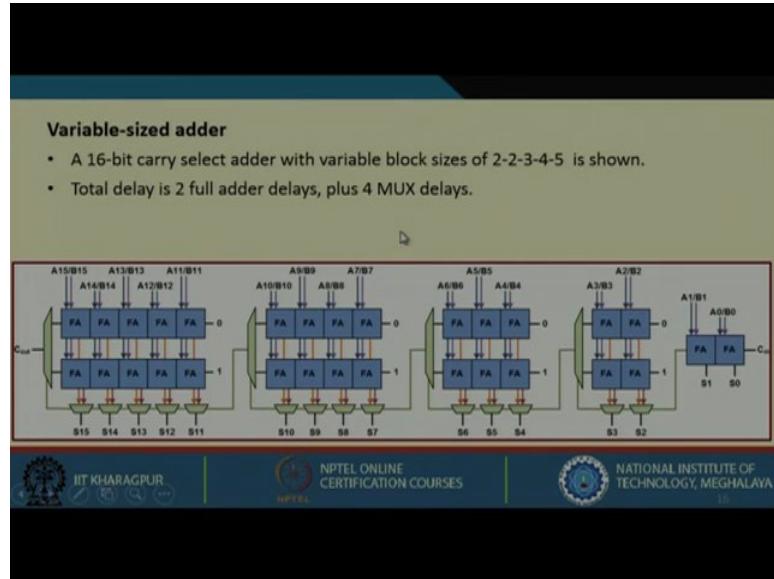
(Refer Slide Time: 29:35)



So, let us take 2 example cascades. So, here we are showing how a 16 bit carry select adder can be generated by a cascade of 4 such 4 bit adders. Now the least significant stage you do not need a carry select adder because you know the carry in already. So, there is no need of any speculation it can be 0 it can be one. So, the last stage there will be a single ripple carry adder, but in the only other stages there will be a module like to be 1 we have explained. Now you see so after the addition time of a 4 bit adder is done. So, all these adders are ready with their some outputs. So, this adder is also ready with its carry output.

So, once it is done you select the multiplexer take out the correct sum select this multiplexer take out the correct sum select the multiplexer take out the correct sum. So, it is just a chain of multiplexers it is quite fast multiplexer means may be to get delays. So, the total time will be the addition time of a 4 bit adder plus the total delay of this multiplexers after that time you will be getting all these some bits in parallel. Now there is an alternate way which can make the performance even better you can make these adders of variable sizes like the last 1 2 bit.

(Refer Slide Time: 31:09)



Two bit 3 bit 4 bit 5 bit the idea is that there is no point in having all 4 bits; you see 2 bits is a minimum time you need for this carry to propagate to this multiplexer. Now by the time this multiplexer propagates the signal this 3 bit addition is also done by the time this multiplexer propagates the signal this 4 bit addition is already done.

So, like this you have to make a calculation of the full adder delays and also the multiplexer delays and come up with an optimum configuration this is a so called 2 2 3 4 5, 2 2 3 4 5 configuration for a 16 bit adder now by you by optimizing the sizes of this adders you can actually minimize the overall addition time right.

(Refer Slide Time: 32:09)

Carry Save Adder

- Here we add three operands (say, X, Y and Z) together.
- For adding multiple numbers, we have to construct a tree of carry save adders.
 - Used in combinational multiplier design.
- Each carry save adder is simply an independent full adder without carry propagation.
- A parallel adder is required only at the last stage.

 IIT-KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, the last kind of adder that we look at today is carry save adder. So, the concept is very simple, in a carry save adder we are adding 3 numbers together I mean if there are more than 3 numbers we to add then you have to construct a tree of such adders. Now what is the carry save adder basically carry save adder is basically an independent array of full adders and only in the last stage we need a parallel adder I will just explain.

(Refer Slide Time: 32:44)

• An illustrative example:

X: 1 0 0 1 1	X: 1 0 0 1 1	X: 1 0 0 1 1
Y: + 1 1 0 0 1	Y: + 1 1 0 0 1	Y: + 1 1 0 0 1
Z: + 0 1 0 1 1	Z: + 0 1 0 1 1	Z: + 0 1 0 1 1
<hr/>	<hr/>	<hr/>
C: 1 1 0 1 1	S: 0 0 0 0 1	C: 1 1 0 1 1
		Sum: 1 1 0 1 1 1

A set of full adders generate carry and sum bits in parallel The sum and carry vectors are added later (with proper shifting)

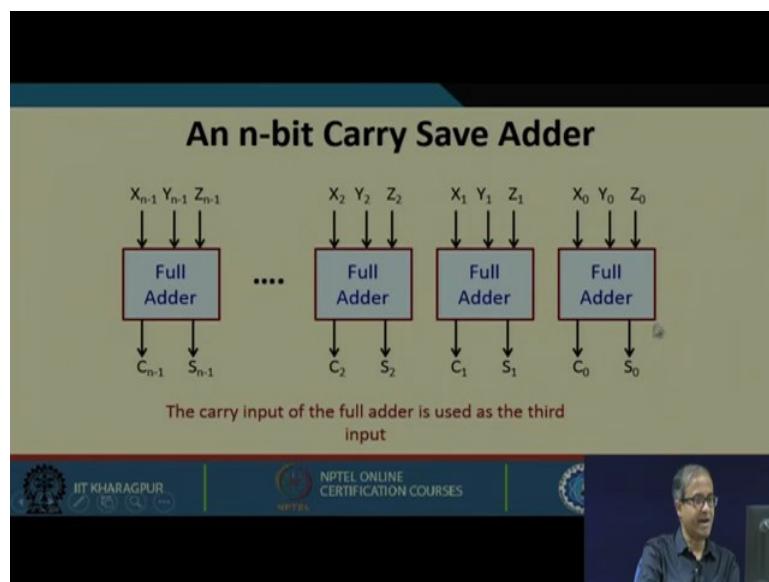
 IIT-KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES | 

Let us take 2 examples that we want to add 2 numbers x and y let us say we add it twice here we are generating only the sum with without any carry propagation independently like the 3 numbers x y z we are adding together.

So, x y z 1 1 1 will generate a sum of 1 one 0 zero will generate a sum of 0 independent no carry propagation 0 1 1 0 also and also here carry we are generating. So, again independently this 1 one 1 will generate a carry of 1 10 1 will generate a carry of 1 0 0 0 will generate a carry of 0 and so on.

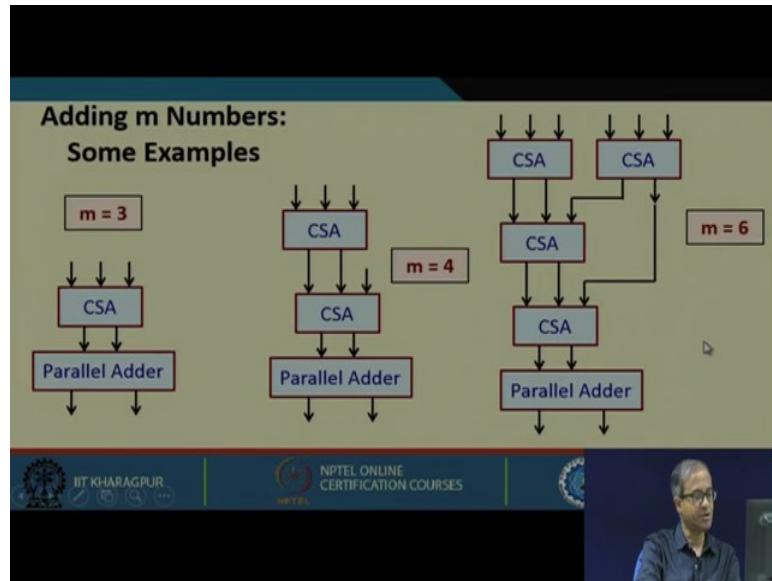
Later on what we do this sum and carry which you have generated like this we do a shift of this carry and add them up together this sum and carry will be getting the final correct sum. So, we are saying is that we are not unnecessarily carrying out carry propagation, we are living carry and sum as 2 separate words and later on we are adding them together after proper shifting like 1 bit shift carry will be shifted by 1 place you can add them together.

(Refer Slide Time: 34:03)



So, a carry save adder technically is just a set of independent full adders without an interconnection because there are 3 inputs a b and a carry in. So, technically a full adder is able to add 3 numbers x y z and is generating sum and carry as the outputs.

(Refer Slide Time: 34:24)



So, some examples I am showing here for m equal to 3 you can use a single carry save adder to add the 3 numbers generate this sum and carry, the last stage use a parallel adder to add them up to get the final sum. Let us say m equal to 4 numbers you want to add you use 1 carry save adder to generate 2 to add 3 numbers sum carry and the fourth number add it with this sum.

And carry this is a parallel adder to finally, get the sum let us say m equal to 6 take 2 CSAs in the first stage to add the 3 numbers and 3 numbers sum carry some carry use another CSA to add 3 of them, again sum carry take the forth and bring it here again some carry it mean parallel adder will get the final sum. So, you can have a tree of carry save adders the last stage you need a parallel adder, the advantage is that the total addition time can be fast and you can add many such numbers together in parallel.

So, with this we come to the end of our discussion on adders. So, in our next lecture we shall be starting some discussion on the design of multipliers.

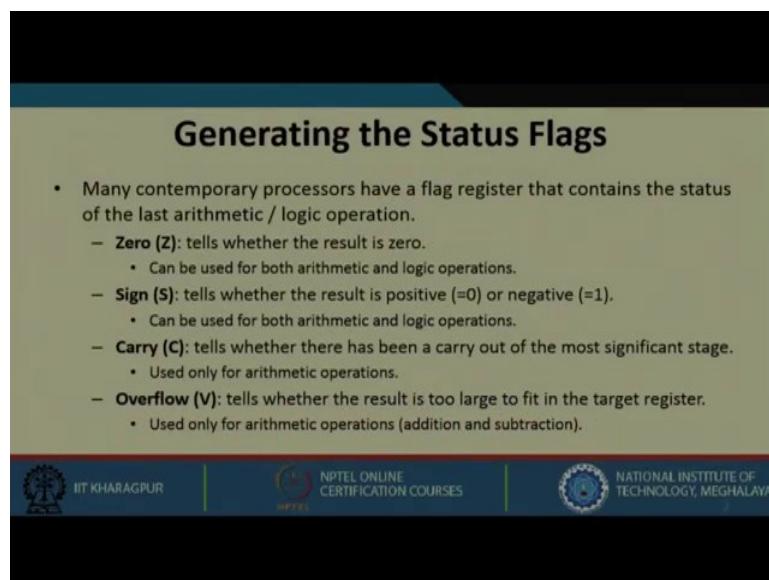
Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 35
Design of Multipliers (Part I)

In the last couple of lectures we had looked at various kinds of Adders. So, how they can make designed implemented, and some we can say comparative study among them. In this lecture first we shall be looking at some aspect about the have of the procedure addition which you have not talked about yet; namely generating some so called conditional flags and then we shall be moving on to design of multipliers.

(Refer Slide Time: 00:57)



Generating the Status Flags

- Many contemporary processors have a flag register that contains the status of the last arithmetic / logic operation.
 - **Zero (Z)**: tells whether the result is zero.
 - Can be used for both arithmetic and logic operations.
 - **Sign (S)**: tells whether the result is positive ($=0$) or negative ($=1$).
 - Can be used for both arithmetic and logic operations.
 - **Carry (C)**: tells whether there has been a carry out of the most significant stage.
 - Used only for arithmetic operations.
 - **Overflow (V)**: tells whether the result is too large to fit in the target register.
 - Used only for arithmetic operations (addition and subtraction).

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

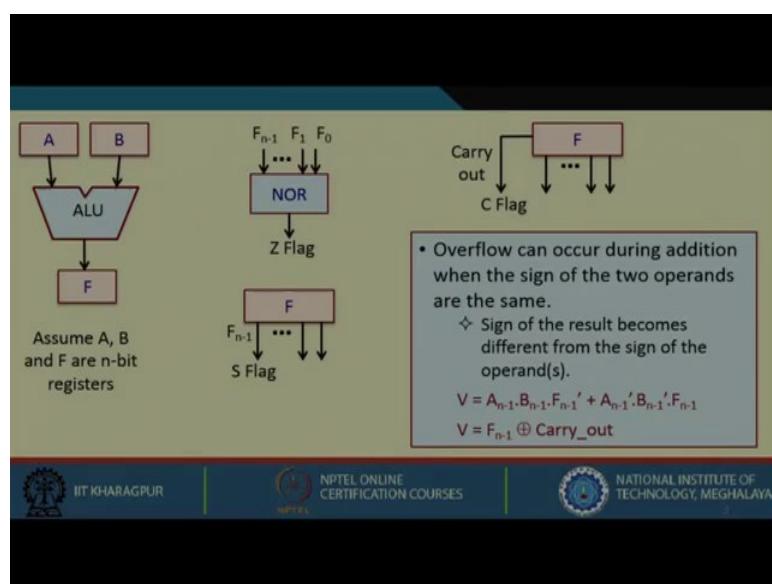
So, in this lecture we shall be stating with generating this status flags; now what is a status flag now many conventional processors if you look at the instruction set or if you look at the instruction set architecture, we will find that they have some specific condition flags like sign flag, 0 flag, over flow flag and so on. These flags are automatically set as a result of some arithmetic or logic operations; like for example, you can do an addition after addition you can compare whether the result is negative or positive this you can do by checking a whether the sign flag is 0 or 1 or you can check if the result is 0 this you can do by checking whether the 0 flag is 0 or 1.

So, the concept of condition flag these, this there are setup flip flops or one bit register you can say they are the flags they are automatically set or reset depending on the result of some previous operation. So, after that you can use a conditional branch instruction for example, to check the status of the flag and take some decision accordingly fine.

So, here we shall be talking about 4 of the commonly use flags, then several other flags also used in some machine; this 0 flag tells whether the result is 0 or not. Normally this flag is set by both arithmetic and logic operations; sign flag tells whether the result is positive or negative. So, for twos complemented representation a 0 will remain positive; 1 will remain negative.

That means, the most significant bit of the result is the same as the sign flag and this sign flag is set for both arithmetic and logic operations typically carry already we have seen adder of we have seen how carry is generated from one addition stage to another. This flag will tell whether there has been a carry out of the final out the most significant stage. And this carry information is used only for arithmetic operations; and lastly there is a flag called overflow which tells whether the result is too large to fit in the register often operation or not. Say used for arithmetic operations typically arithmetic addition and subtraction sometimes also multiplication.

(Refer Slide Time: 04:05)

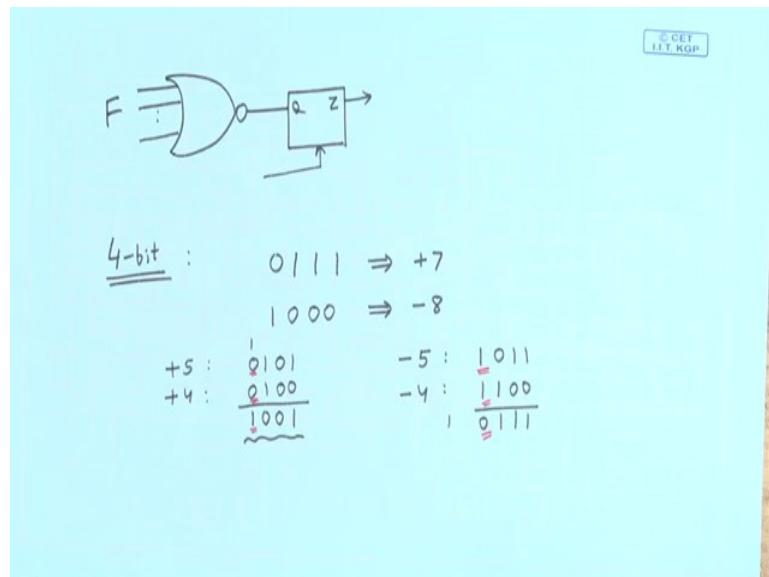


So, will see these things one-by-one; so generation of this flags is not difficult rather easy let us consider very simple typical scenario like this, where we have an arithmetic

logic unit which takes as inputs from two registers A and B, and the result it is depositing into another register F. Let us assume all of them are n bit registers. Firstly, the 0 flag can be very easily generated by using a single n input NOR gate. So, what is the NOR gate? The output of NOR gate will be one if all the input are 0. So, if all the bits of the result are 0 the output NOR gate will be one that will indicate the 0 flag.

So, actually the circuit will be like this, you have a large NOR gate.

(Refer Slide Time: 05:03)



So, the input from the result register F are faded here, and the output here fading to a flip flop this is a z flip flop or flip flop let us call this input Q. So, this will be loaded by some control signal and once loaded this 0 flag will be available simply just an NOR gate. Similarly the sign flag can be generated directly from the most significant bit of the result; just you take this fn minus one if feed it to S flip flop that will be a S flag register very simple. So, generating of this sign flag is also very easy. Carry flag well you know that for addition subtraction you already seen that there is already a carry out signal coming out of the adder.

So, this ALU will also be having a carry out signal, that carry out signal will be directly going into the C flag flip flop this we call load directly right. So, this is all can generate carry. Now talking about over flow see here we are talking about over flow during additions and subtraction. See over flow can occur during addition only when the sign of the two operands are the same. Now if one of the number is positive and the other

number is negative you can never have a overflow. Let me take an example let us take a very simple example of a 4 bit representation those complement.

So, in 4 bit what are the maximum numbers again represent in the positive side I can represent maximum 0 1 1, which means plus 7, in a negative side I can represent 1 0 0 0 which means minus 8. So, let us say I am adding plus 5 and plus 4 let say I am adding these two numbers plus 5 will be 0 1 0 1, plus 4 will be 0 1 0 0. Now quiet naturally the sum of the numbers will not fit in 4 bits because maximum you can store is 7. So, let us say what happens of after addition 1 0 0 with the carry of 1 1. So, my result is a 1 0 0 1.

So, one observation is the sign of my numbers where positive, but the result has become negative. Same thing you look for two negative numbers same thing will happen let us take minus 5 and minus 4 let us say; minus 5 will be 1 0 1 1 this is minus 5 minus 4 will be 1 1 0 0 just add them of 11 1 0 with a carry of one. Now here also we see that your sign bit of your original numbers were 1 and 0, but the sign bit of the result has become different right these are the condition for over flow detection.

So, the condition is sign of the result becomes different from the sign of the operands. So, you can write down in terms of expression like this, sign of the two numbers are minus and B n minus suppose A 1 1, but F n minus 1 is 0 or A minus 1bar B n minus 1 bar they are 0 0, but this is 1; and this expression can be simplified like this also simply take an XOR of F n minus 1 and the final carry out they will actually mean the same thing. So, either a whichever easier to implement you can implement over flow flag.

Now one thing we are not considering over flow for division operation at present because division or multiplication let us say. Because in multiplication normally when we add two n bit numbers the product is stored in A 2 n bit register double this size.

(Refer Slide Time: 09:46)

- The MIPS architecture does not have any status flags.
- Why?
 - MIPS ISA is designed for efficient pipeline implementation.
 - Several instructions can be in various stages of execution in the pipeline.
 - Flag registers result in **side effects** among instructions.
- MIPS stores information about the flags temporarily in a GPR.

slt	\$t0, \$s1, \$s2
beq	\$t0, \$zero, Label

So, there can never be an over flow there. So, for that purpose for the time being or where assuming that over flow is important only for addition and subtraction not for multiplication.

Now, again talking of the flag register the MIPS architecture that we have been studying, does not have any status flags at all why; because the instruction set on the architecture of MIPS was designed primarily for very efficient pipeline implementations. Now in a pipe line will be studying this in much more detail later, that when there are several instructions which are running in a pipeline they are in various stages of execution. Now if there is a single set of flag register there can be confusion; the first register may be setting the 0 flag, the second register may also be trying to set the 0 flag. So, there can be some confusion. So, for a pipeline implementation existing of the flags create something called side effects which are undesirable.

So, MIPS uses an entirely different philosophy, they do not use any status flags rather they temporarily stored flag information in a general purpose register. I am showing simple example there is an instruction called set less than. Set less than what it does it will check if S 1 is less than S 2 or not these two registers? If they are less S 1 is less than S 2 then the target register will be set to one otherwise this target register will be set to 0. So, you are storing either a 0 or a one in the target register. So, as if the whole target register your using as a flag something like that, and immediately after that you can have

an instruction like a conditional branch if equal to 0 with 0 your checking whether the result is 0 or not; 0 means this was not less than right then you jump to a label.

So, earlier what you do? Earlier you do branch if not zero jump to a level, but now since you do not have a flag register you have to use one of these set instructions first, then use a conditional branch instruction checking the value of the target register whether is 0 or nonzero fine.

(Refer Slide Time: 12:22)

Multiplication of Unsigned Numbers

- Multiplication requires substantially more hardware than addition.
- Multiplication of two n-bit number generates a 2n-bit product.
- We can use shift-and-add method.
 - Repeated additions of shifted versions of the multiplicand.

1 0 1 0	Multiplicand M (10)
1 1 0 1	Multiplier Q (13)

1 0 1 0	
0 0 0 0	
1 0 1 0	
1 0 1 0	

1 0 0 0 0 0 1 0	Product P (130)

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Let us now come to multiplication. Talking about multiplication of unsigned numbers the first thing is that the amount of hardware require for multiplication is substantially greater as compare to addition. Multiplication of two n bit numbers can generate to n bit product let us look at an example to less treat it. So, we are following a method which is very similar to the pen and pencil method we are familiar with. So, when you multiplied two numbers using this so called decimal number system, we follow a very similar approach. So, what we do? Let us say this is our multiplicand this is our multiplier all expressed in binary they unsigned numbers not twos compliment. So, 1 0 1 0 means 10 1 1 0 1 means 13.

So, I check the first digit of the multiplier; if it so, normal multiplication what we do? We multiply this by this you do the same thing multiply one by this you get 1 0 1 0, then shift one plus slight left multiply is 0 with this we get 0 0 0 0, multiply one with this shift multiply one with this shift this, then finally you add all the bits up 0 1 0 1 1 is 0 with a

carry of 1 0 with a carry of 1 0 with a carry of 1 0 with a carry of 1. So, the product is one thirty you see that the product requires 8 bits. So, this one example shows that when you multiply two 4 bit numbers, your result may become double of that 8 bits.

So, the product can be 2^n bits and this method have a addition is called shift and add. So, we do repeated additions of shifted versions of the multiplicand; because here you are only multiplying by 1 or 0. So, one may think this same multiplicand will be appearing 1 0 1 0 1 0 1 0 1 0 after various number of shifting. So, either we can write down all these so called partial products you can say and then add them of together or you can continuously go on adding as the next one is generated that is also possible.

(Refer Slide Time: 15:05)

A General Case

A_3	A_2	A_1	A_0
B_3	B_2	B_1	B_0
A_3B_0	A_2B_0	A_1B_0	A_0B_0
A_3B_1	A_2B_1	A_1B_1	A_0B_1
A_3B_2	A_2B_2	A_1B_2	A_0B_2
A_3B_3	A_2B_3	A_1B_3	A_0B_3

- Each $A_i B_j$ is called a partial product.
- Generating the partial products is easy.
 - Requires just an AND gate for each partial product.
- Adding all the n -bit partial products in hardware is more difficult.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

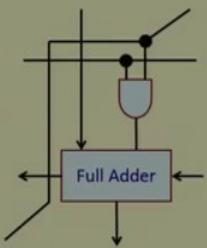
So, in a general case I am just writing this in a very general term this A_i and B_j can be 0 or 1. So, B_0 it can be 0 or 1 is multiplied to all this 4 bits. B_0 into A_0 , B_0 into A_1 , B_0 into A_2 , B_0 into A_3 . Similarly B_1 , B_1 , A_0 , B_1 , A_1 , B_1 , A_2 , B_3 , A_3 similarly B_2 with B_3 the is exactly what we doing. Now this each of these product of that have shown here this is called a partial product; now for adding to 4 bit numbers if you can just check there are 16 partial products.

So, for multiplying to n bit numbers there will n^2 partial products right to generate every partial product you need just an and gate just you and the corresponding bits you get this partial product right. So, for multiplying to n bit numbers for generating all the partial products will be needing n^2 .

(Refer Slide Time: 16:37)

Design of a Combinational Array Multiplier

- We can directly map the multiplication process as discussed to hardware.
 - We use an array of cells to generate the partial products.
 - Instead of adding the partial products at the end, we add the partial products at every stage of the multiplication.
- The required multiplication cell is as shown.
 - Combines capabilities of partial product generation and also addition of partial products.



IIT Kharagpur NPTEL ONLINE CERTIFICATION COURSES

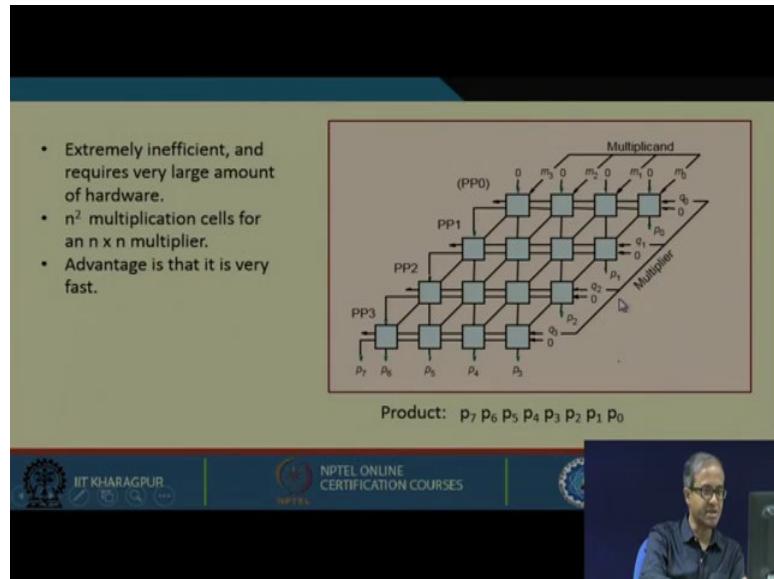
A video player showing a person speaking.

Such and gets and then you add them in a suitably you will be needing a complex add a circuit to adds so many partial products together. So, you need a lot of hardware this is what I wanted to say.

So, just from this basic principle let us try to come off with the simple hardware implementation of an adder, this is called a combinational array multiplier. So, what we are doing? We are using something called a multiplication cell. So, whatever the process we are just now shown here shift and add generating the partial products, you are directly trying to map it in hardware how we are doing? We are using an array of cells like this many cells each of the cells will look like this. So, let us was explain this first.

So, we are not waiting till the end to add all the partial products, we are adding the partial product at every stage. Now for thus this indicate? This and gate which is shown this generates the partial products and this full adder will be adding this partial product with another partial product coming from the previous stage after shifting, and this full adder will be getting a carry input it will generating a carry output, and this will be a generating a some.

(Refer Slide Time: 18:06)



It will clear if you see this schematic diagram how it is done. Here for 4 by 4 multiplication I am showing, there will be 16 such cells. So, diagonally from the right the bits of the multiplicands are coming m_0, m_1, m_2, m_3 ; that means, here from here and that is same bit is also coming here going to the next stage. So, the m_0 is also going to the next stage, m_1 is also going to the next stage, m_2 and m_3 and on the other side the quotient bits are applied q_0, q_1, q_2, q_3 it is from this side.

So, the quotient and the multiplied if you just add them you will get the partial product so that partial product are getting added you have a ripple carry adder. These adder full adders are all connected in cascade. So, you have ripple carry adder. So, initially the input is one input is 0 0 0 0 0; if first partial product is getting added to 0 you get the partial product here.

Next one: so the way this multiplier is designed there is automatically a one bit left shift you can see. Next stage is computing the partial products can adding to the previous one next one is compare a next partial product after again one bit shift adding to the previous one in this why it goes run. So, finally, at the end whatever you get will be the product see p_0 will be generated here P_1 here P_2 here then P_3, P_4, P_5, P_6, P_7 this is have you can also see from here because a first bits are generated earlier the last bits will be generated together the last stage the same thing is happening here.

So, this method is very simple whatever you are doing by hand you have directly mapped it to hardware, but the problem is that this kind of a multiplied is extremely inefficient, it requires extremely large amount of hardware; that means, you need n square such cells, but the advantage is that it is much faster. As compare to the other multiplication methods that will see that will be some kind of sequential multiplication method that will be fine.

(Refer Slide Time: 20:36)

Unsigned Sequential Multiplication

- Requires much less hardware, but requires several clock cycles to perform multiplication of two n -bit numbers.
 - Typical hardware complexity: $O(n)$.
 - Typical time complexity: $O(n)$.
- In the “*hand multiplication*” that we have seen:
 - If the i -th bit of the multiplier is 1, the multiplicand is shifted left by i bit positions, and added to the partial product.
 - The relative position of the partial products do not change; it is the multiplicand that gets shifted left.

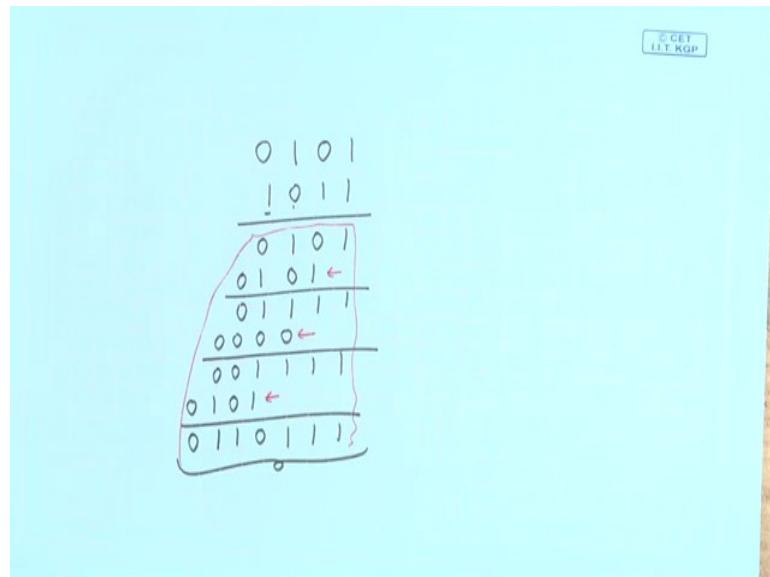
IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

So, we have seen the multiplication method which is called combinational array multiplier. So, it is a combinational circuit I apply multiply may quotient. So, after some delays my result will be generated. Now we look at some multiplication methods which will be requiring much less amount of hardware, but will be sequentially nature means you will have to run the circuit for several clock cycles, to complete the multiplication ok.

So, let us look at unsigned sequential multiplication first. So, exactly what we have said is that requires much less hardware, but requires several clock cycles. Much less hardware means what? Earlier for combinational array multiplier we are using n square cells. So, the hardware complexity was order n square, but here we are saying would be needing of hardware which will be proportional to the number of bits being multiplied and also the time require you also be order n proportional to n .

Now, in the just in the hand multiplication step what we have seen? You have seen that if be its bit of the multiplier is one we shift the multiplicand by one bit position, and add to the partial product like I am just showing on example here.

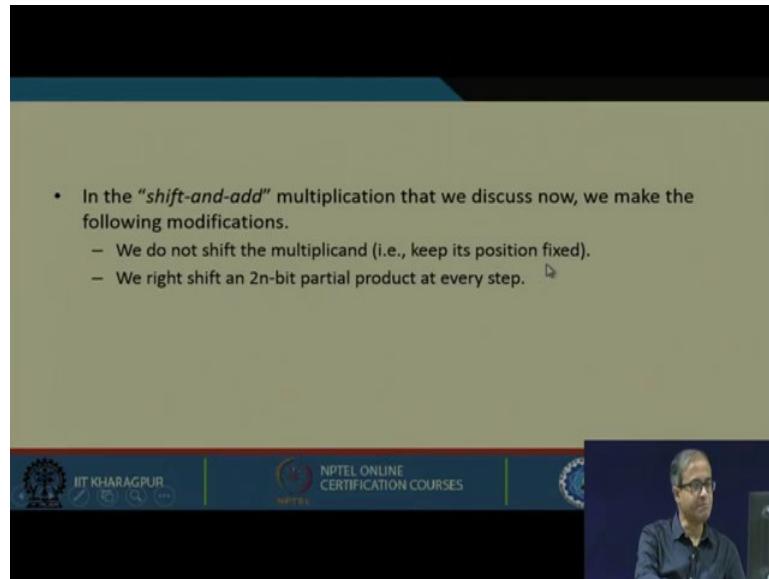
(Refer Slide Time: 22:10)



Let us multiply 0 1 0 1 with 1 0 1 1 first with this one line multiply this becomes 0 1 0 1, then with this one and multiply with one shift 0 1 0 1 I immediately add them then comes this 0 0 means he again shift and add 0 just immediately add them last 1 0 1 0 one just again another shift 0 1 1 0 this is your final product.

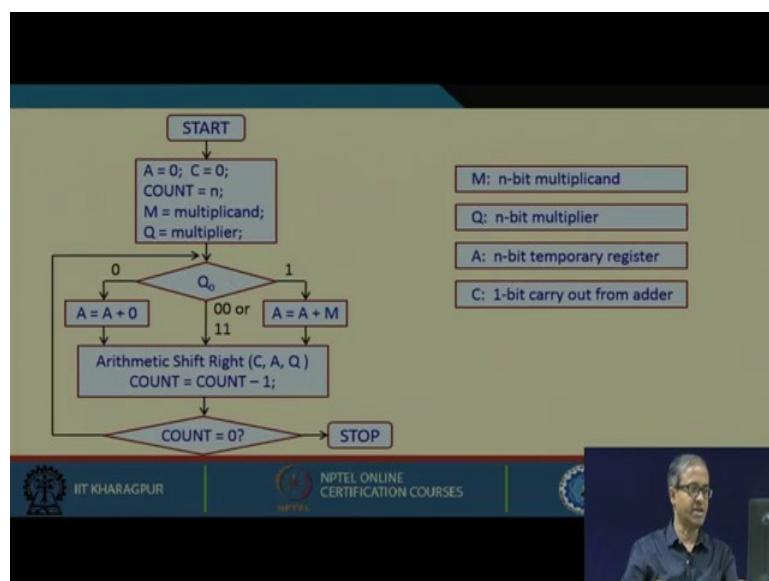
So, your shifting multiplicand by variable amounted every carrier shifting here shifting again here shifting again here, but the relative position of the partial product you are not changing this bits bit positions are changing kept in the same position, but for actually implementation will be doing something else, instead of doing this will be making a small change.

(Refer Slide Time: 23:33)



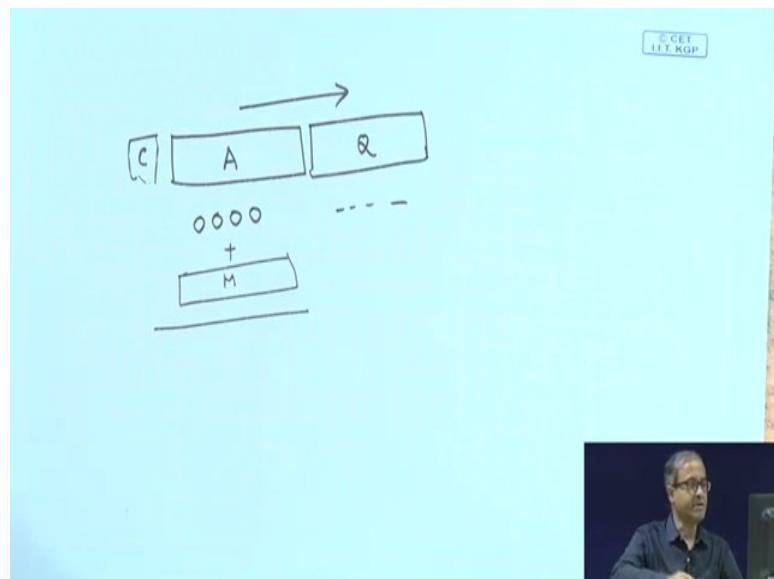
So, we do not shift the multiplicand will keep the position fixe, rather the partial product will be shifting right at every step. Like here you are keeping the partial product same and shifting the multiplicand left every step, what was saying will keep the multiplicand in the same place partial product will shifting right. So, means these two are equivalent. So, either you keep this in the same place and shift this left or keep this in the same place and shift this right. So, we are following the second order because it is easier to implement in hardware right ok.

(Refer Slide Time: 24:15)



This shift and add multiplication let us look at it first we talk about the overall steps in the forms of flow chart has shown here. See here the idea is as follows we are computing something like a partial product.

(Refer Slide Time: 24:40)



You see we are using some registers A M and Q. Normally this registers A and Q they are consider together, initially A will be loaded with all zeros and Q will be loaded whatever the quotients right and will be shifting this to the right and whenever we have to just add the multiplicand will be adding at here, the M will be adding into this position and this whole thing will be shifting right this will be the principle will be following ok.

So, you see this. So, we start by initializing this A register to 0, C is a carry bit 2 0, count is the number of times will be looking for n M is multiplicand Q is multiplier. So, I am assuming M and Q are both n bit multiply n bit registers, A is a n bit temporary register initialize to 0 and C is the carry out form an adder. So, what we do we check the last bit of the quotient Q 0.

So, if it is one we have to add if it is 0 we are adding A 0 right in the shift and add method let us look at this. So, you look at the last bit Q 0. So, if the Q 0 bit is one you add the multiplicand to a just like as your seen your adding a multiplicand to a if the Q 0 bit is 0 for adding 0 to a right and after adding what I am I told here after adding will be shifting right a and Q not only a and Q after adding they will be a carry which will be coming here this will be c. So, C A Q together will be shifting right. So, we are do

exactly that we are doing a arithmetic shift right C A Q and where decrementing the count by one. We are checking the weather count is 0 or not if it is not 0 we go back if it is 0 we stop right this is the method.

(Refer Slide Time: 27:13)

C	A	Q	
0	0 0 0 0 0	0 1 1 0 1	Initialization
0	0 1 0 1 0	0 1 1 0 1	$A = A + M$ Step 1
0	0 0 1 0 1	0 0 1 1 0	Shift
0	0 0 1 0 1	0 0 1 1 0	$A = A + 0$ Step 2
0	0 0 0 1 0	1 0 0 1 1	Shift
0	0 1 1 0 0	1 0 0 1 1	$A = A + M$ Step 3
0	0 0 1 1 0	0 1 0 0 1	Shift
0	1 0 0 0 0	0 1 0 0 1	$A = A + M$ Step 4
0	0 1 0 0 0	0 0 1 0 0	Shift
0	0 1 0 0 0	0 0 1 0 0	$A = A + 0$ Step 5
0	0 0 1 0 0	0 0 0 1 0	Shift





So, I am illustrating with help of an example let us consider two 5 bit numbers 10 and 13. 10 in 5 bits is this, 13 in 5 bits is this. So, the product is expected to be 130. So, initially my A is 0, Q contains the quotient 0 1 0 1 0 0 1 and the carry bit is 0. So, what we do I check whether Q is 0 or 1 I say it is 1. So, I have to add multiplicand M to this. So, I add I do A equal to A plus M. So, A become it was 0 A will become 0 1, 0 1, 0 and I do a right shift there is no carry I do a right shift. So, after right shift what happens the quotient the last bit you already seen after right shift the next bit comes here. So, I will inspecting the next bit after that this bit.

Because I have to check the bits one by one. So, again I check this, this is 0; so now, will have to add 0. So, there is no change in a no change and again we do a shift right again the shift right. So, the next bit comes here next bit of the quotient comes here one you add m again to it. So, 0 1 0 1 0 plus 0 0 0 1 0 it becomes 0 1 1 0 0 this you can check and you do again you do a shift. So, this one comes here finally. So, you check this. So, you again add m to this 0 0 1 1 0 it becomes 1 0 0 0 0. So, again do a shift last time we check this bit the last bit 0. So, you add 0 no change in shift this is your final result final product.

So, you see the quotient which I had add loaded initially in this register has slowly shifted out and is replaced by the product finally. So, at the end I had made 5 shifts. So, the quotient bits are all they have gone out. So, the result does not contain the quotient bits any more. So, after addition shifting, addition shifting, addition shifting they will slowly fail up this whole 10 bit register right this is the product.

(Refer Slide Time: 29:48)

	C	A	Q	
Initialization	0	0 0 0 0 0	1 0 1 0 1	
Step 1	0	1 1 1 0 1	1 0 1 0 1	$A = A + M$
	0	0 1 1 1 0	1 1 0 1 0	Shift
Step 2	0	0 1 1 1 0	1 1 0 1 0	$A = A + 0$
	0	0 0 1 1 1	0 1 1 0 1	Shift
Step 3	1	0 0 1 0 0	0 1 1 0 1	$A = A + M$
	0	1 0 0 1 0	0 0 1 1 0	Shift
Step 4	0	1 0 0 1 0	0 0 1 1 0	$A = A + 0$
	0	0 1 0 0 1	0 0 0 1 1	Shift
Step 5	1	0 0 1 1 0	0 0 0 1 1	$A = A + M$
	0	1 0 0 1 1	0 0 0 0 1	Shift

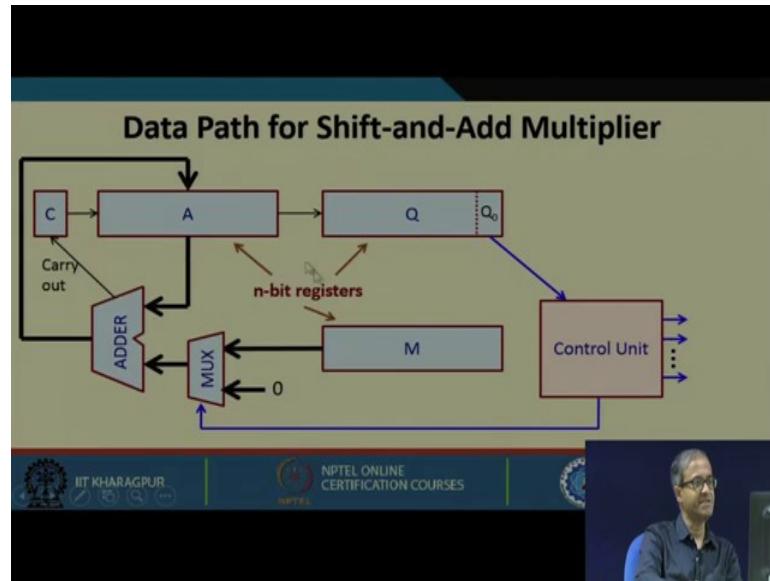
Example 2: $(29) \times (21)$
Assume 5-bit numbers.
M: $(11101)_2$
Q: $(10101)_2$
Product = 609
 $= (1001100001)_2$

Now, in this example the carry bit was never one, let us take another example where the carry bit can be one. Let say we multiply 29 and 21 these are the two numbers the product is supposed to be 609. This same way you look at one you add the multiplicand 11101 then do a right shift next bit is 0. So, you add 0 no change then do a right shift 1. So, 0011 one will be added to 1110 one this will be generating this product with a carry these you can check.

So, after shift this carry will get shifted in here right then you check again the next bit 0 no addition; that means, you are adding 0 no change again shift next bit is one, the last bit you again add M to it you again get a carry out of 1, again shift right this one will again go in and this will be your last and final result 609.

So, this is how the basic shift can add multiplication works.

(Refer Slide Time: 31:05)



Now talking about the hardware circuit for (Refer Time: 31:12) it is very simple. So, what we shown in the example we need exactly that we need a register for A we need a register for Q, we need a carry flip flop and they will be connected as a shift register, we need an adder with one of the inputs coming from M, and the other inputs either coming from the multiplicand or the 0.

And there we will have these are all n bit registers A Q and M the last bit of Q is Q 0 and there will be a control unit which will be checking Q 0 at every step. And will be selecting a multiplexer appropriately either it is selecting 0 or m. And also it will be generating control signal for the other part. So, means after addition the shifting will go on, it will keep track of how many times it will be repeating. So, control unit will be doing that right. So, the data path as you can see is very simple other than the registers you need just an adder and a multiplexer, and just one flip flopper carry.

So, with this we come to the end of this lecture in the next lecture, we shall be looking at some other methods of multiplication particularly signed multiplication, how we can multiply to sign numbers and some improvements they are in.

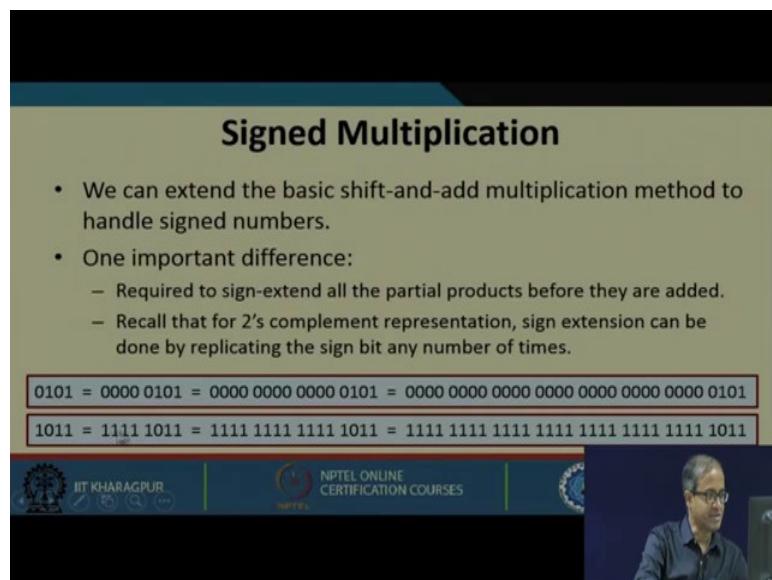
Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 36
Design of Multipliers (Part 2)

In the last lecture we have seen how to carry out unsigned multiplication. So, we continue our discussion to the present lecture. So, today we shall see how we can carry out signed multiplication, multiplication of two numbers which can be either positive or negative specifically in those compliment form. So, how we can do that?

(Refer Slide Time: 00:45)



Signed Multiplication

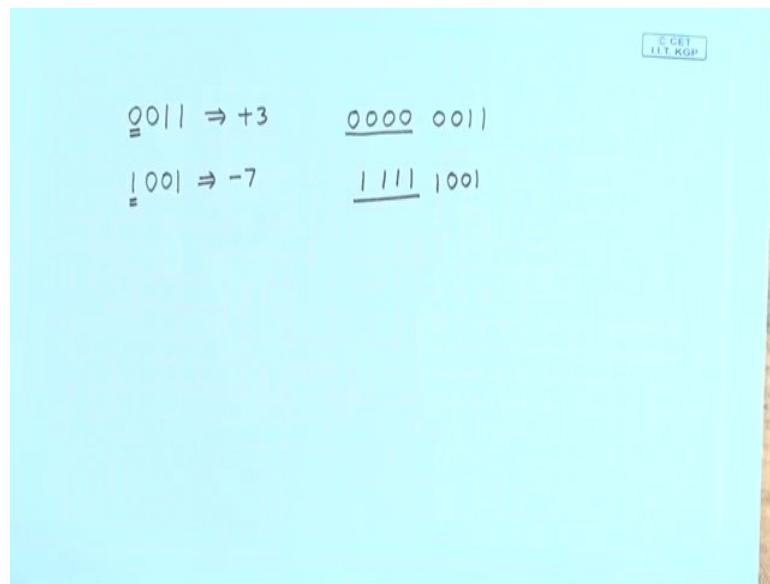
- We can extend the basic shift-and-add multiplication method to handle signed numbers.
- One important difference:
 - Required to sign-extend all the partial products before they are added.
 - Recall that for 2's complement representation, sign extension can be done by replicating the sign bit any number of times.

0101 = 0000 0101 = 0000 0000 0000 0101 = 0000 0000 0000 0000 0000 0000 0000 0101
1011 = 1111 1011 = 1111 1111 1111 1011 = 1111 1111 1111 1111 1111 1111 1111 1011

IIT KHARAGPUR **NPTEL ONLINE CERTIFICATION COURSES**

So, our topic of discussion today's signed multiplication. So, the first approach that we talk about is a simple extension of the basic shift and add multiplication method that have already seen, but here there is one important difference like depending on the sign of your multiplier or the multiplicand. So, you will have to sign extend all the partial products. Now you recall earlier being it was discussed that any signed number can be sign extended to any number of bits.

(Refer Slide Time: 01:50)



So, what is the rule for sign extension for those compliment numbers? If the number is positive you can fill the number with as many 0's you want if the number is negative you can fill it with any number of once you want like for example, 0 0 1 1 represents plus 3 in 4 bits suppose you required to represented in 8 bits you simply add four 0's in the beginning; that means, you are simply extending the sign of the number to this additional bits.

Let us see another example 1 0 0 1 this represents minus 7 in those compliment; if you again want to represent it in 8 bits. So, what you do similarly you look at this sign it is one, you replicate this sign bit hear this is called sign extension. So, the value of number remains same in this process right. So, this is actually what you have to do. So, the example is shown here also let us say number 0 1 0 1 this is the positive number, you can sign extend it to 8 bits sign extend it to 16 bits sign extend to thirty two bits like this similarly a negative number sign extend to 8 bits to 16 bits to 32 bits just replicate the sign bit as many times you want.

(Refer Slide Time: 03:15)

An Example: 6-bit 2's complement multiplication

Note: For n-bit multiplication, since we are generating a 2n-bit product, overflow can never occur.

1 1 0 1 0 1	(-11)
x 0 1 1 0 1 0	(+26)

0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
1 1 1 1 1 1 1	1 0 1 0 1
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0
1 1 1 1 1 0 1 0	1 0 1
1 1 1 1 0 1 0 1	1 0 1
0 0 0 0 0 0 0	0 0 0 0 0 0

1 1 1 0 1 1 1 0 0 0 1 0	(-286)

So, in the shift and add extension for sign numbers, we just use this principle just look at it. So, we are multiplying minus 12th is the multiplicand which is a negative number, with a multiply which is positive let say plus 26. So, we represent them by 6 bit numbers. So, the result is supposed to be 12 bits twice of that. So, when the first bit is 0 0 multiplied by this will be 0. So, we are sign extending it by adding 6 0's in the beginning to make it 12 bit partial product.

Next is one. So, we add this, this is negative that is why we use a sign extension to it all once next again 0 0 sign extension next is 1, 1 this multiplicand sign extension, next one again one multiplicand sign extension (Refer Time: 04:25) 0 0 sign extension. So, if you add these bits of now, you will see that what you have got is this if you convert it to decimal you see that this is indeed minus 286 which is the product.

So, this simple shift and add we can extend by using this some extension concept the extending this sign extension. So, we will be getting correct result means in terms of the sign of the product and another thing I just mention earlier also. So, you are also assuming one thing that in gets of multiplication since you are assuming that 2 n bit numbers are being multiplied to generate at 2 n bit product. So, overflow can never occur here because the product can be at most 2 n bits right not more than that.

(Refer Slide Time: 05:25)

Booth's Algorithm for Signed Multiplication

- In the conventional shift-and-add multiplication as discussed, for n-bit multiplication, we iterate n times.
 - Add either 0 or the multiplicand to the 2n-bit partial product (depending on the next bit of the multiplier).
 - Shift the 2n-bit partial product to the right.
- Essentially we need *n additions and n shift operations*.
- Booth's algorithm is an improvement whereby we can avoid the additions whenever consecutive 0's or 1's are detected in the multiplier.
 - Makes the process faster.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL



Now, we look at some improvement to this basic algorithm that we have just now seen.

Now, in this shift and add method if you just recall the basic mechanism what you are doing. Our multiplier in the multiplicand are both n bit numbers we are repeating the process n times inspecting one bits of the multiply every time, you are either adding the multiplicand or adding the zero. So, we need n number of addition steps and n number of shifting steps that is required in the basic shift and add method. Now what we discuss here is that essentially these is also a type of a shift and add multiplier, but we are trying to reduce the number of additions steps by using some kind of a you can say bit encoding technique.

So, the method we talk about now is called Booth's algorithm; and Booth's algorithm can be used for signed numbers also to it is compliments multiplication. So, as I just now said in the conventional shift and add multiplied for n bit multiplication, we have to repeat the process for every bit n times, you either add 0 or the multiplicand to the partial product at every iteration and also you shift. So, we need n additions and n shift operations and as I said in Booth's algorithm we are trying to avoid additions forever possible very specifically whenever there are consecutive 0's or consecutive 1's in the multiplier we avoid additions; this can make the process faster.

(Refer Slide Time: 07:30)

Basic Idea Behind Booth's Algorithm

- We inspect two bits of the multiplier (Q_i, Q_{i-1}) at a time.
 - If the bits are same (00 or 11), we only shift the partial product.
 - If the bits are 01, we do an addition and then shift.
 - If the bits are 10, we do a subtraction and then shift.
- Significantly reduces the number of additions / subtractions.
- Inspecting bit pairs as mentioned can also be expressed in terms of *Booth's Encoding*.
 - Use the symbols +1, -1 and 0 to indicate changes w.r.t. Q_i and Q_{i-1} .
 - $01 \rightarrow +1$, $10 \rightarrow -1$, 00 or $11 \rightarrow 0$.
 - For encoding the least significant bit Q_0 , we assume $Q_{-1} = 0$.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Let us look at the basic idea, see here again we look at the numbers in the same way we assume.

(Refer Slide Time: 07:36)

A

Q

$Q_{n-1} \dots Q_2 Q_1 Q_0$

$Q-1 = 0$

That we have a temporized called a; we have a question register Q. So, initially the quotients register is loaded with the quotients. So, it will be $Q n$ minus 1dot dot Q 2 Q 1 and Q 0. Now here we are adding another single bit register a flip flop, this where calling as Q minus 1. If I design that will see very shortly if Booth's algorithm we do not look at one bit of the multiplier at a time, but rather we look at pairs of bits $Q i$ and $Q i$ minus 1

for all i . So, you see earlier we are looking at Q_0 first then Q_1 then Q_2 , but now we shall be looking at pairs $Q_0 Q_u$ minus 1 that say this additional bit you have kept then $Q_1 Q_0$ then $Q_2 Q_1$ then $Q_3 Q_2$ and so, on pairs of bits right this is the basic idea because of this pairing, we need to add another flip flop at the end which is initialize to 0 right.

Now, the rule of multiplication is very simple, we look at this pairs of bits. So, if the bits are the same either 0 0 or 1 1 no need to do any addition subtraction, just only shift the partial product. If the bits are 0 0, then you do an equal to a plus M do an addition and then do a right shift. If the bits are 1 0 do a subtraction a equal to a minus M and then the right shift see here I am not going into the formal proof of the Booth's algorithm, the method looks very simple. So, if you workout you can also verify that whatever you are doing here is actually carrying out multiplication, but here I am not going into the formal proof I am just taking the Booth's algorithm and how it can be implemented right.

So, by doing this whenever we find 0 0's or 1 1's in the bit pairs we avoid addition or subtraction right. So, this can significantly reduce the number of additions or subtraction. Now in some textbooks you will see that instead of looking at this bit pairs they have used an alternate notation called Booth's encoding, they have used the symbols plus 1 minus 1 and 0 to indicate this status of these bit pairs whether they are changing or not changing.

If the bits are 0 and 1 you code it as plus 1 if it is 1 0 you call it minus 1. If it is 0 0 or 1 1 it does not change you call it is 0 and this already I have said that for coding the last bit we assumed that that extra flip flop Q minus 1 that you added here, this is initialize to 0 right this is initialized to 0 fine. So, let us take some examples of this booth encoding technique suppose my multiplier was this.

(Refer Slide Time: 11:21)

• Examples of Booth encoding:

- a) 01110000 :: +1 0 0 -1 0 0 0 0
- b) 01110110 :: +1 0 0 -1 +1 0 -1 0
- c) 00000111 :: 0 0 0 0 +1 0 0 -1
- d) 01010101 :: +1 -1 +1 -1 +1 -1 +1 -1

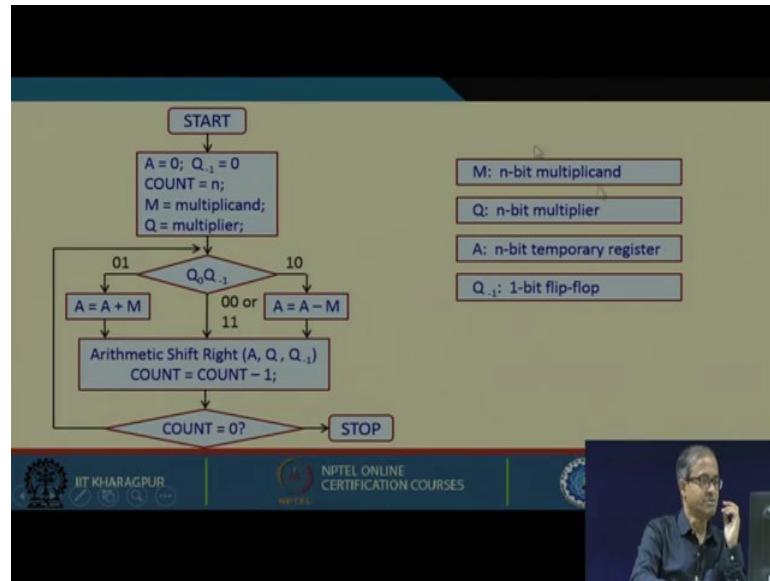
• The last example illustrates the worst case for Booth's multiplication (alternating 0's and 1's in multiplier).

- In the illustrations, we shall show the two multiplier bits explicitly instead of showing the encoded digits.

So, there will be an additional 0 here that Q minus 1. So, 0 0 is coded as 0, this 0 0 is coded as 0 this 0 0 0 0 two more zeros, 1 and 0 will be coded as minus 1, 1 1 will be coded as 0, 1 1 will be coded as 0, 0 1 is coded as plus 1. So, is a essentially looking from right to left 0 means in this encoded encodings scheme 0 means that we need only shifting no addition or subtraction minus 1 means we need a subtraction plus 1 means we need an addition another example. So, last two bits are 0 0 this is 0 1 0 is minus 1, 1 1 0, 0 1 is plus 1 1 0 is minus 1 1 one 1 1 0 0 plus 0 1 is (Refer Time: 12:23).

Now, the last example this is the worst case for Booth's algorithm alternating 0's and ones. So, we will find here you have not got any 0's all are plus 1 minus 1 plus 1 minus 1 alternately like that. So, we will have to always do either an addition or subtraction alternative step. So, this is the worst case of Booth's algorithm, but on the average you may have to do additions which are for example, for the c case you need only two addition or subtraction rest are all zeros.

(Refer Slide Time: 12:58)



So, the Booth's algorithm goes like this in the flowchart form. So, I have already shown you that a is a register n bit, this M holds the multiplicand Q holds the multiplier these are all n bits and Q minus 1 is a one bit flip flop initialized to zero. So, you inspect Q 0 and Q minus 1 every time.

So, if it is 0 1 then you have to add, if it is 1 0 you have to subtract, but if it is 0 0 or 1 1 you do not have to add or subtract. So, what we will have to do? You will have to shift anyway shift what shift this Q and this Q 1 minus 1 all together shift all this 3 things together right. So, the earlier Q 0 will go into Q 1, Q 1 will go into Q 0. So, every time you will be checking Q 0 and Q minus 1 only. So, after shifting this will ensure that you are inspecting all the bit pairs and after doing the right shift a Q and Q minus 1 decrement the count it is initialized to n you check whether you have checked all the n times you have done if it is 0 now if it is yes you stop.

(Refer Slide Time: 14:44)

A	Q	Q ₋₁	
0 0 0 0 0	0 1 1 0 1	0	Initialization
M: (1 0 1 1 0) ₂	0 1 0 1 0	0 1 1 0 1	A = A - M
-M: (0 1 0 1 0) ₂	0 0 1 0 1	0 0 1 1 0	Shift Step 1
Q: (0 1 1 0 1) ₂	1 1 0 1 1	0 0 1 1 0	A = A + M Step 2
Product = - 130 = (1 1 0 1 1 1 1 1 0) ₂	1 1 1 0 1	1 0 0 1 1	Shift
	0 0 1 1 1	1 0 0 1 1	A = A - M Step 3
	0 0 0 1 1	1 1 0 0 1	Shift Step 4
	0 0 0 0 1	1 1 1 1 0	Shift Step 5
	1 0 1 1 1	1 1 1 0 0	A = A + M
	1 1 0 1 1	1 1 1 1 0	Shift





So, the algorithm is very simple. So, let us work out the some examples. So, as I said if there consecutive 0's and 1's no additional subtractions are needed. So, you can skip over those runs of 0's and ones, let us take an example. So, multiplicand is negative minus 10. So, minus ten is this 13, this is 13 this M is 1 0 1 1 0 minus M this is 2's complement of being 0 1 0 1. So, what I will do whenever I have to subtract I will actually add minus M that is equivalent to subtracting m. So, I am showing both M and also minus M.

So, here the product will be sorry it will be minus 130, x minus 130 the product is minus 1 thirty. So, this is our initial thing you loaded with the quotient A is 0, Q 1 is also 0 you check the bit pair it is 1 0. So, 1 0 means we have to subtract A equal to A minus M. So, you are adding minus M to a minus M is 0 1 0 1 0 it become 0 1 0 1. So, on then you do a shift.

Now the next bit pair is 0 1, 0 1e is you have to add you add M to 0 0 1 0 one. So, the result will be this 1 1 0 1 1 then you do a shift again now the next two bits are again one zero. So, 1 0 means you have to subtract. So, add 0 1 0 0 again to it becomes this. So, again do a shifting next bit pair is 1 1.

So, just skip no addition or subtraction just shifting next bit pair is 0 1. So, (Refer Time: 16:49) do an addition. So, add M to this it becomes 1 0 1 1 one then do a shift. So, whatever you have finally is the result product this is minus 130. So, this is the process

of bush multiplication. So, it is very similar to shift and add the only decision is that you have added an addition bit Q minus 1 with checking two bits at a time.

(Refer Slide Time: 17:15)

Example 2:
 $(-31) \times (28)$
 Assume 6-bit numbers.
 M: $(100001)_2$
 $-M: (011111)_2$
 Q: $(011100)_2$
 Product = -868
 $= (110010)_2$
 $011100)_2$

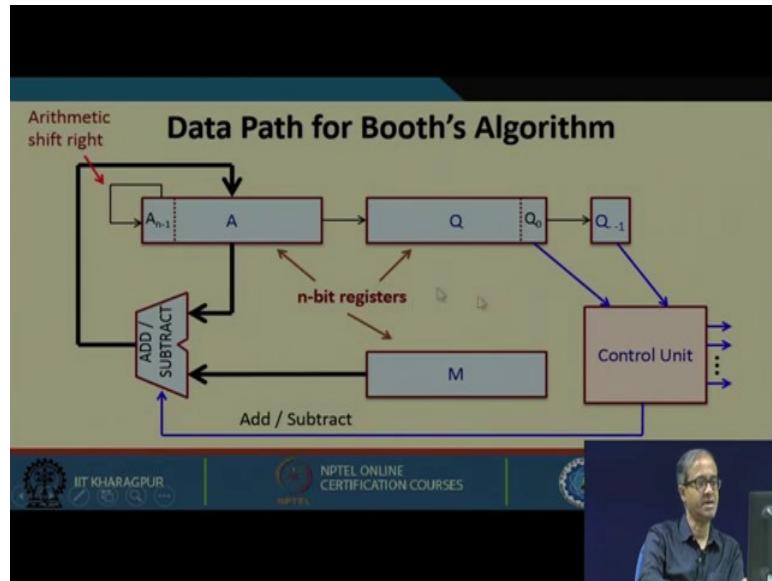
A	Q	Q ₁	
0 0 0 0 0 0	0 1 1 1 0	0 0	Initialization
0 0 0 0 0 0	0 0 1 1 1	0 0	Shift Step 1
0 0 0 0 0 0	0 0 0 1 1	1 0	Shift Step 2
0 1 1 1 1 1	0 0 0 1 1	1 0	$A = A - M$ Step 3
0 0 1 1 1 1	1 0 0 0 1	1 1	Shift Step 4
0 0 0 1 1 1	1 1 0 0 0	1 1	Shift Step 5
1 0 0 1 0 0	1 1 1 0 0 0	1	$A = A + M$ Step 6
1 1 0 0 1 0	0 1 1 1 0 0	0	Shift

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

And sometimes you are skipping addition or subtraction step doing only shift. Let us take another example minus 31 and 28, this is your quotient and this is your multiplicand minus M is this.

So, the product is supposed to be again it should be minus I missed minus sign minus 868 which is this. So, you check the last 2 bits 0 0. So, only shift no addition next two bits are again 0 0. So, only shift 1 0 means are subtraction. So, we add minus M then shift again 1 1 no addition subtraction only shift. So, so again 1 1 is only shift. Finally, you have 0 1 means addition add M to this, you will get this and then a shift. So, you are done this will be your final result minus 868. So, you see the Booth's algorithm is simple in this example you have seen that you are able to skip 1 2 3 4 times only twice you needed to do some addition subtraction.

(Refer Slide Time: 18:39)



So, this method is significantly faster as compared to be previous shift and add multiplication method. In terms of the hardware requirement it is very similar you need a register a Q register and one additional Q minus 1 flip flop here, and earlier you needed only an adder, but now we have an adder or substructure because sometimes you also need to add and here while you are doing a right shift like in the previous example, when you are doing a right shift you are doing a arithmetic right shift. So, the sign bit would be replicated right.

So, that sign bit it is coming back when you are right shifting this is for arithmetic right shift and the control unit will be checking both Q 0 and Q minus 1 together, and we will decide whether to add or subtract or whether to skip this step for the control signals will be generated accordingly right fine.

(Refer Slide Time: 19:38)

Design of Fast Multiplier

a) Bit-Pair Recoding of Booth's Multiplication

- A technique that halves the maximum number of summands; derived directly from the Booth's algorithm.
- If we group the Booth-coded multiplier digits in pairs, we observe:
 - $(+1, -1): \quad (+1, -1) * M = 2 * M - M = M$
 - $(0, +1): \quad (0, +1) * M = M$
- We need a single addition instead of a pair of addition & subtraction.
 - Other similar rules can be framed.
 - Shown on next slide.

NPTEL ONLINE CERTIFICATION COURSES

IIT KHARAGPUR

NPTEL

Now, let us look at how we can improve this speed of the multiplier even further, we have already seen how both multiplier can reduce a number of additions or subtractions. Now we look at a modification of Booth's; Booth's multiplier here, this is called bit pair recoding of Booth's multiplier. So, this method effectively halves the maximum number of addition or subtraction you see the idea is like this.

So, in Booth's algorithm we have seen for that 0 1, 0 1, 0 1, 0 1 multipliers scenario, in the worst case you need n multiplications or divisions multiple or subtractions sub addition subtraction right sorry. So, in this new method modified bit by decoding what we are saying is that the worst case maximum number of addition and subtraction will be only M by 2, 50 percent of what was there earlier. So, we are reducing the worst case complexity of addition subtraction to almost half 50 percent.

So, the observations are like this. So, in the original booth encoding what we have seen let us say we have two symbols plus 1 minus 1 one after the other; plus 1 means for addition minus 1 means for subtraction of M . Now plus 1 coming before means that I have to make one shift left shift M left shift of M means 2 into M and then we subtract M . So, plus 1 minus 1 both operating on M means effectively 2 into M minus M ; because this plus 1 will be shift left by 1 plus effectively it means multiplied multiplying by 2 and minus 1 will be lower significant minus M . So, effectively this means M you also

observe if we have a paired 0 plus 1, 0 plus 1 also means M because you multiplied by M in the lower place in a higher place it is 0 you do not do anything it remain same.

So, the final result of these two are equivalent same. So, wherever you have plus 1 minus 1 you can as well replace them by 0 plus 1, which means you have brought in an additional 0 which means one less additional subtraction there are other similar rules you can frame.

(Refer Slide Time: 22:16)

The slide contains a table and a list of rules:

Original Booth-coded Pair	Equivalent Recoded Pair
(+1, 0)	(0, +2)
(-1, +1)	(0, -1)
(0, 0)	(0, 0)
(0, 1)	(0, 1)
(+1, 1)	--
(+1, -1)	(0, +1)
(-1, 0)	(0, -2)

- Every equivalent recoded pair has at least one 0.
- Worst-case number of additions or subtractions is 50% of the number of multiplier bits.
- Reduces the worst-case time required for multiplication.

At the bottom of the slide, there are logos for IIT Kharagpur and NPTEL, and a video feed of a speaker.

So, we are showing the rules here in the next slide. So, just like that we have shown here you can similarly try this out you can also prove these rules similarly. Plus 1 and 0 they can be encoded as 0 plus 2 means here you are multiplying by 2. So, this new symbol plus 2 and minus 2 comes in this new method plus 2 means shift by 1 position this minus 1 plus 1. If there are you replace them as 0 minus 1 0 0 does not change 0 1 does not change plus 1 1 (Refer Time: 23:20) you cannot do that this, this is not a valid occurrence plus 1 minus 1 can be replaced by 0 plus 1 this already we have seen and minus 1 0 can be replaced by 0 minus that if you see. So, in the modified encoding at least 1 0 is there which means at most 50 percent times will be doing additional subtraction, rest to all times you will be doing shifting.

(Refer Slide Time: 23:36)

0	0	1	1	0	1
.	-1	.	-1	.	-2

1	1	1	1	1	0 0 1 1 0
1	1	1	1	1	0 0 1 1
1	1	1	1	0	0 1 1

1	1	0	1	1	1 0 0 0 1 0
A red box highlights the final result: 1 1 0 1 1 1 0 0 0 1 0. To the right, a list of results is shown:

- M = 001101 (+13)
- 1 * M = 110011
- 2 * M = 100110

The footer of the slide includes logos for IIT Kharagpur, NPTEL, and NITI Aayog, along with the text 'NPTEL ONLINE CERTIFICATION COURSES'. On the right side of the slide, there is a video feed of a person speaking."/>

Example: (+13) X (-22) in 6-bits.

Original: Multiplier -- 1 0 1 0 1 0
Booth: Multiplier -- -1 +1 -1 +1 -1 0
Recoded: Multiplier -- 0 -1 0 -1 0 -2

0 0 1 1 0 1
. -1 . -1 . -2

1 1 1 1 1 1 0 0 1 1 0
1 1 1 1 1 1 0 0 1 1
1 1 1 1 0 0 1 1

1 1 0 1 1 1 0 0 0 1 0

• M = 001101 (+13)
• -1 * M = 110011
• -2 * M = 100110

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NITI AAYOG

So, this reduces the worst case time required by the Booth's multiplication algorithm. So, let us take an example plus 13 multiplied by minus 22. So, the original multiplier minus 22 was this, this is minus 22 in 2's complement. If you do booth and this encoding this is 0 0 0 1 0 is minus 1, 0 1 is plus 1, 1 0 is minus 1, 0 1 plus 1, 1 0 is minus 1. Now according to this new rules whatever I have shown in the table you replace like for example, minus 1 0 this table shows minus 1 0 can be replaced by 0 minus 2, minus 1 0 replaced by 0 minus 2, you look at pair wise minus 1 plus 1, minus 1 plus 1 you replace by 0 minus 1 replace 0 minus 1 plus 1 replace by 0 minus 1.

So, now this is your new booth recoded coding. So, 0 minus you have to (Refer Time: 24:33) not been addition subtraction. So, here I am showing them by dots the zeros, these are the places where to do additional subtraction. So, you need only 3 mean addition steps required here minus 2 you do minus 2 M shift and then multiply, multiply on the shift minus 1 just the operand you see plus thirteen 1 0 0 1 1 minus 1 1 0 0 1 1, but here because it was minus 2. So, after 1 0 1 1 we have done a left shift left shift means multiplied 2. So, 1 0 1 1 was minus 1 if you left shift by one it becomes minus 2.

So, you see the process of multiplication becomes much simpler here only 3 steps required the other 3 steps you can skip. So, this is what I have said minus 1 into M means this minus two into M means shift left by one position. So, a 0 goes 0 here fine

(Refer Slide Time: 25:41)

b) Carry Save Multiplier

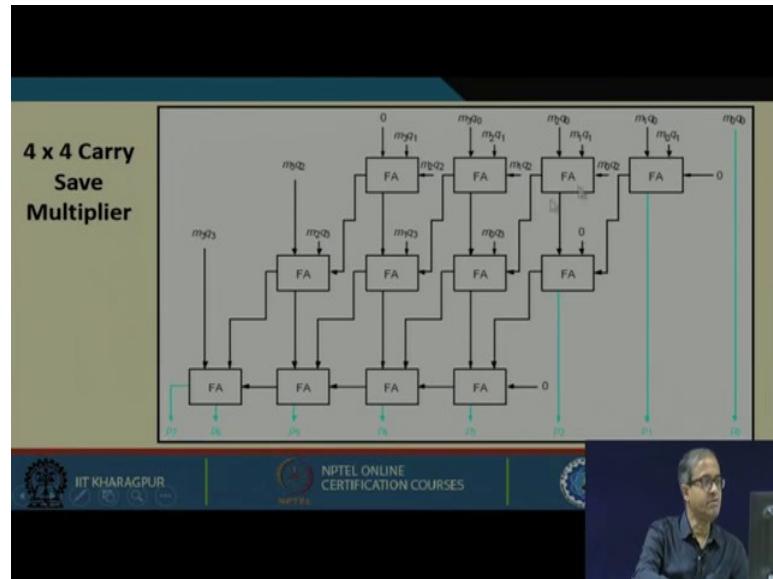
- We have seen earlier how carry save adders (CSA) can be used to add several numbers with carry propagation only in the last stage.
- The partial products can be generated in parallel using n^2 AND gates.
- The n partial products can then be added using a CSA tree.
- Instead of letting the carries ripple through during addition, we *save* them and feed it to the next row, at the correct weight positions.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

And the last kind of multiplier that we talk about here is called carry save multiplier, see earlier we looked at carry save adder. We have seen that we can use carry save adder to add several numbers mean without any carry propagation we are adding the carry only in the last stage and also we have seen, in the combinational array multiplier that the partial product generation requires n square AND gates.

So, what we say here is that we can use a tree of carry save adders to multiply the n partial products to add them up to add the n partial products we need a carry save adder tree. Now see what we saw for carry save adder is that a single carry save adder can add up to 3 numbers, if we want to add more number of numbers like 4 5 6 you will require several carry save adders. Now for multiplication also there will be n partial product you will have to add all those n partial products. So, you can have a similar carry save adder tree where all those n partial products you are adding and finally, at the end you get the final product.

(Refer Slide Time: 27:10)



So, here we show an example for a 4 by 4 multiplier. So, these are independent full adders are every step the first two stages indicate carry save adders there no connection in between. So, you see that all the partial product generated by the AND gates are fed here this is M_1Q_0 , M_0Q_1 , M_0Q_2 , M_1Q_1 , M_2Q_0 .

So, we just if you can check you will see that exactly the same thing is happening and when the carry of the carry output is going to next stage you are doing a one bit shift and then connecting. So, that the shifting can implement and some of the product comes a generated directly for example, M_0Q_0 means the LSB of the product. The first full adder can generate P_1 one the next bit this full adder can generate P_2 , but in the last stage you will be needing a regular adder with carry propagation. So, here I have shown a ripple carry adder this can also be a carry look at adder no problem.

(Refer Slide Time: 28:34)

- Wallace Tree Multiplier
 - A Wallace tree is a circuit that reduces the problem of summing n n -bit numbers to the problem of summing two $\Theta(n)$ -bit numbers.
 - It uses $n/3$ (floor of) carry-save adders in parallel to convert the sum of n numbers to the sum of $2n/3$ (ceiling of) numbers.
 - It then recursively constructs a Wallace tree on the $2n/3$ (ceiling of) resulting numbers.
 - The set of numbers is progressively reduced until there are only two numbers left.
 - By performing many carry-save additions in parallel, Wallace trees allow two n -bit numbers to be multiplied in $\Theta(\log_2 n)$ time using a circuit of size $\Theta(n^2)$.

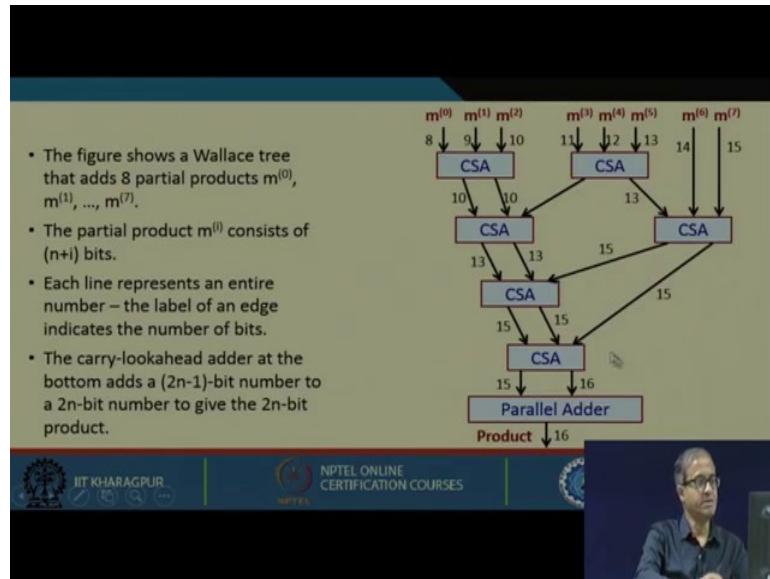
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL



So, this will be generating the remaining 5 bits of the product. So, if we have a circuit like this you can carry out multiplication much faster using carry save adder. So, you have an alternate method called Wallace tree Wallace tree is very similar in concept. So, Wallace tree is a circuit it is a like a tree in generate it is a graph, that reduces the problem of summing n numbers to the problem of summing two numbers, which are of size theta of n . So, this also uses carry save adders it uses the floor of n by 3 so many carry save adders to convert the sum of n numbers to the sum of ceiling of $2 n$ by 3 numbers. So, we are not going into the detail of this mathematics, we shall just show some example Wallace tree formal formulation for a particular multiplier.

So, the advantages that finally using many carry save addition in parallel Wallace tree will allow $2 n$ bit numbers to be multiplied in theta $\log n$ time, this is important you are having a logarithmic time multiplier, but the circuit complexity will become n square theta of n square.

(Refer Slide Time: 29:53)



So, this is one Wallace tree example that I am showing, which is adding 8 partial products M_0, M_1 up to M_7 . Suppose we are doing 8 by 8 multiplication. So, the final result should be 16 bit. So, these numbers beside these edges will indicate number of bits that are being generated here this will also be 13. So, is 13 this also 13. So, these are the partial products, these are the carry save adder tree you generate, some of the carry save adders are working in parallel you see these are in parallel these two are in parallel, but these two are in sequential and finally, you have a parallel adder this is so, called Wallace tree multiplier that also uses carry save added tree.

So, what Wallace tree says- it is a particular way of arranging the carry save adders, so that maximum amount of parallelism can be exploited and the depth of the tree is reduced. So, how many carry save adders maximum can be used this is just an example I have shown fine. So, with this we come to the end of this lecture.

So, we have so far looked at the design of adders and multipliers. So, one thing is left among the basic arithmetic operations namely division. So, in the next lecture we shall be looking at some algorithms for division and how they can be implemented.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 37
Design of Dividers

In the last few lectures we had seen how we can implement adders and multipliers. In this lecture we shall be looking at the design of dividers the various algorithms which if you can use and how we can implement them in hardware. So, the topic of our discussion today is design of dividers.

(Refer Slide Time: 00:46)

The slide has a dark blue header bar with the title 'Introduction' in white. Below it is a light blue section containing text and a table. On the left, there is a bulleted list:

- Division is more complex than multiplication.
- Example: Typical values in Pentium-3 processor.
 - Not easy to construct high-speed dividers.
- The ratios have not changed much in later processors.

On the right, there is a table with three columns: 'Instruction', 'Latency', and 'Cycles / Issue'.

Instruction	Latency	Cycles / Issue
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Floating-point Add	3	1
Floating-point Multiply	5	2
Floating-point Divide	38	38

At the bottom of the slide, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Before we start let us look into a basic problem that is there in the division operation; because the first thing is that division is more complex than multiplication because of various reasons that we shall be seeing during the course of this lecture.

Now, to appreciate this point that division is indeed more difficult than multiplication or any other arithmetic operation, let us look at some typical facts and figures which are present in one of the commercial processors of the year Pentium 3. So, here in a table we are showing for some typical instructions or in arithmetic operations 2 things one is called the latency other is called the cycles per issue we shall be coming back to this. So, just one thing we have we can see from this table that the values for divide operations

either integer or a floating point whatever is much higher as compared to the values for load store multiply addition this kind of operations.

Now, these are the values for Pentium 3 and even in the very recent processors the relative differences they still remain. So, we shall be coming back to this slide once more first let us see what do you mean by latency and cycles per issue.

(Refer Slide Time: 02:26)

- Latency:
 - Minimum delay after which the first result is obtained, starting from the time when the first set of inputs is applied.
- Cycles/Issue:
 - Whenever a new set of inputs is applied to a functional unit (e.g. adder), it is called an *issue*.
 - Pipelined implementation of arithmetic unit reduces the number of clock cycles between successive issues.
 - For non-pipelined arithmetic units (e.g. divider), the number of clock cycles between successive issues is much higher.
 - Next input can be applied only after the previous operation is complete.

Let us look at this first latency refers to the minimum delay starting from the time when the first set of input is applied and up to which the first result is obtained. So, latency talks about the just the initial delay, the delay for the first time suppose I have a circuit it can be an adder it can be a multiplier divider whatever I apply a set of inputs.

Now, the question is after how much time I get my result this time duration is called latency. So, minimum after how much time I can get back my first result. Now there is another thing. Well, you see I apply an input I get back an output after some time that is equal to the latency. Now the second point is that how frequently can I apply the inputs do I have to wait until the operation is complete before I apply the second input or I can overlap my operation in some way.

Well, if I have to wait for the operation to be complete before I apply the next input then this so called cycles per issue this is called cycles per issue after minimum how much gap I can apply my second input that will be the same as latency. But if we are allowed

to have some kind of an overlap means before the first operation is complete I am allowed to feed in the next input or the next to next input, this is what happens in a typical pipeline processor which you shall be looking into detail much later during the course of these lectures.

Now, in a pipeline implementation we can apply inputs much faster although the latency value remains basically the same I apply 1 input after how much time the result comes in comes out that remains same, but the inputs that I apply 1 after the other the gaps between them that can be less. So, this second pointer parameter here is cycles per issue. So, let us try to explain this. So, whenever we apply a new set of inputs to some functional unit like an adder or a multiplier, we call it an issue that the inputs have been issued.

Now, as I have said that if we have pipelined implementation, then we can reduce the number of clock cycles between successive issues, but for a circuit like divider it is very difficult to have a pipeline implementation, most of the dividers are non-pipelined. So, for such circuits the number of clock cycles between successive issues is much higher next input can be applied only after the previous operation is complete. So, let us go back to the previous slide and see, for load store kind of instructions latency is 3 clock cycles while cycles per issue is one that means, I can issue 1 such instruction every clock cycle there is an overlap possible for integer multiply again latency is 4.

So, I can complete multiplication in 4 cycles, but I can apply new sets of inputs every 1 clock cycle floating point addition again 3 is the latency one is the clock cycles per issue and for floating point multiply it is 5 and 2, but you see if I divide latency and cycles per issue values are same 36 here and for floating point 38 this means that division units are not pipelined they actually work as a single non pipeline block you apply an input get the output after it is finished only then you apply the second input. So, cycles per issue are equal to the latency.

(Refer Slide Time: 07:06)

The Process of Integer Division

- In integer division, a *divisor* M and a *dividend* D are given.
- The objective is to find a third number Q, called the *quotient*, such that
$$D = Q \times M + R$$
where R is the *remainder* such that $0 \leq R < M$.
- The relationship $D = Q \times M$ suggests that there is a close correspondence between division and multiplication.
 - Dividend, quotient and divisor correspond to product, multiplicand and multiplier, respectively.
 - Similar algorithms and circuits can be used for multiplication and division.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Let us now look into the process of integer division. So, what you really do when you divide 2 numbers. So, for integer division we have something called a divisor something called a dividend we divide the dividend by the divisor D is greater than M. So, the objective is to find a quotient that is the result of the division and of course a remainder; remainder R should be less than this divisor M, and this M DQ and R they are related by an equation like this, quotient multiplied by the divisor plus the remainder will give you the dividend fine.

Now, just if we ignore the remainder for the time being D is Q multiplied by M. So, you see there are basically 2 things which you are multiplying and you get back D. So, there is an analogy you can draw between division and multiplication the operations look quite similar. So, here we are talking about dividend quotient and divisor and in multiplication we talked about product multiplicand and multiplier. So, there is a correspondence between dividend and product quotient and multiplicand divisor and multiplier.

Now, this correspondence will be clear when we look at the basic data part or the circuits that we use for division and if you compare this circuit with what we had used for multiplication, you can see that see this correspondence immediately fine. So, because of this correspondence very similar circuits and very similar kinds of algorithms can be used for multiplication as well as division.

(Refer Slide Time: 09:10)

The slide contains the following text and diagram:

- One of the simplest division methods is the sequential digit-by-digit algorithm similar to that used in pencil-and-paper methods.

Divisor M **1 1 0**

$D = 37 = (100101)_2$ $M = 6 = (110)_2$ Quotient Q = 6 Remainder R = 1	$0 \ 1 \ 1 \ 0$ $\overline{1 \ 0 \ 0 \ 1 \ 0 \ 1}$ $\underline{- \ 1 \ 1 \ 0}$ $-----$ $1 \ 0 \ 0 \ 1 \ 0 \ 1$ $- \ 1 \ 1 \ 0$ $-----$ $0 \ 1 \ 1 \ 0 \ 1$ $- \ 1 \ 1 \ 0$ $-----$ $0 \ 0 \ 0 \ 1$ $\underline{1 \ 1 \ 0}$ $-----$ $0 \ 0 \ 1$	Quotient $Q = Q_0Q_1Q_2Q_3$ Dividend $D = R_0$ $Q_0.M$ (<i>Does not go; $Q_0 = 0$</i>)
---	---	---

R_1
 $Q_1 \cdot 2^{-1} \cdot M$ (*Does go; $Q_1 = 1$*)

R_2
 $Q_2 \cdot 2^{-2} \cdot M$ (*Does go; $Q_2 = 1$*)

R_3
 $Q_3 \cdot 2^{-3} \cdot M$ (*Does not go; $Q_3 = 0$*)

$R_4 = \text{Remainder } R$

At the bottom right, there is a video frame showing a person speaking.

So, let us just work out a simple example division using the traditional approach which is sequential digit by digit, I will shift an add kind of thing.

So, which is quite similar to what we do using the so called pencil and paper approach. So, the example we take is for a dividend D which is 37, in decimal it is 1 0 0 1 0 1 divisor is 6 1 1 0 we are dividing D by M. So, D we are writing here this is 37, 1 0 0 1 0 1 and this is my divisor 1 1 0 which is 6. So, what we do in a normal division step we see if my divisor goes with the first 3 digit here it is 1 0 0, if it goes I can subtract right, but here we see that 1 1 0 is greater than 1 0 0 which means it does not go because it does not go what to do we set the next quotient bit to 0 if it does not go.

So, the next quotient bit is set as 0. So, this is one step. So, because it does not go we do not make any change to the dividend it remains the same next step. The divisor we shift by 1 place and repeat the same process, we right shift by one step and we compare it with this 1 0 0 1. Now we see 1 0 0 1 is greater than this; that means, we can subtract which means it does go and the next quotient bit will be one. So, because it goes now we can do the actual subtraction 1 minus 0 is 1 this is 1 this is 0 that is it.

So, you repeat the same process divisor again you shift by another position and again check 1 1 0, 1 1 0 again it goes. So, next quotient bit is also equal to one. So, again you subtract 0 0 0 and this 1 remains last step the quotient is shift with this divisor is shift it again 1 place and again you check, but here it does not go because 0 0 1 is smaller. So, it

does not go. So, the next quotient bit is 0. So, I do not make any change. So, what it finally remains here this will be my remainder and I have already found out my quotient. So, my quotient is 6 in decimal my remainder is 1 in decimal.

So, you see here the steps which I have shown side by side. So, the quotient bit that we are generating for example, Q 1 equal to 1 and this divisor M I am shifting it right by 1 position I am expressing it as 2 to the power minus 1 into M 2 to the power minus 1 means divided by 2 and you recall divided by 2 means shifting the number right by 1 position. So, we are doing exactly that here, similarly here we are shifting M right by 2 positions 2 to the power minus 2 means dividing by 4, here we are shifting with right by 3 positions 2 to the power minus 3 into M.

So, actually whatever we are trying to subtract it is actually the quotient bit multiplied by 2 to the power some minus I into M this partial remainders you can say R 1 starting with R 0, R 1, R 2, R 3 and finally, R 4 you get we subtract these values from this R i's this is what we do here.

(Refer Slide Time: 13:16)

- In the example, the quotient $Q = Q_0Q_1Q_2\dots$ is computed one bit at a time.
 - At each step i , the divisor shifted i bits to the right (i.e. $2^i \cdot M$) is compared with the current partial remainder R_i .
 - The quotient bit Q_i is set to 0 (1) if $2^i \cdot M$ is greater than (less than) R_i ,
 - The new partial remainder R_{i+1} is computed as:

$$R_{i+1} = R_i - Q_i \cdot 2^i \cdot M$$

Now,. So, just to recall exactly what we saw. So, we were computing the bits of the quotient one at a time, and at each step what you are doing the divisor was being shifted I bits to the right you see here it was 1 bit, 2 bits, 3 bits. So, I goes from 1 2 3, 2 to the power minus 1, minus 2, minus 3 like this.

So, we shift I bits to the right; that means, we compute 2^I minus M and this compared with the current partial remainder whether it goes if it goes then we set Q i equal to 1 if it does not go we set Q i equal to 0 right, and the new partial remainder is computed by subtracting. So, if it does not go Q i is 0. So, actually this is 0 anyway. So, we do not make any change R i minus 0, but if it is 1 we subtract the shifted divisor 2^I minus i M from R i right this is what we do, but in the actual machine implementation.

(Refer Slide Time: 14:24)

- Machine implementation:
 - For hardware implementation, it is more convenient to shift the partial remainder to the left relative to a fixed divisor; thus

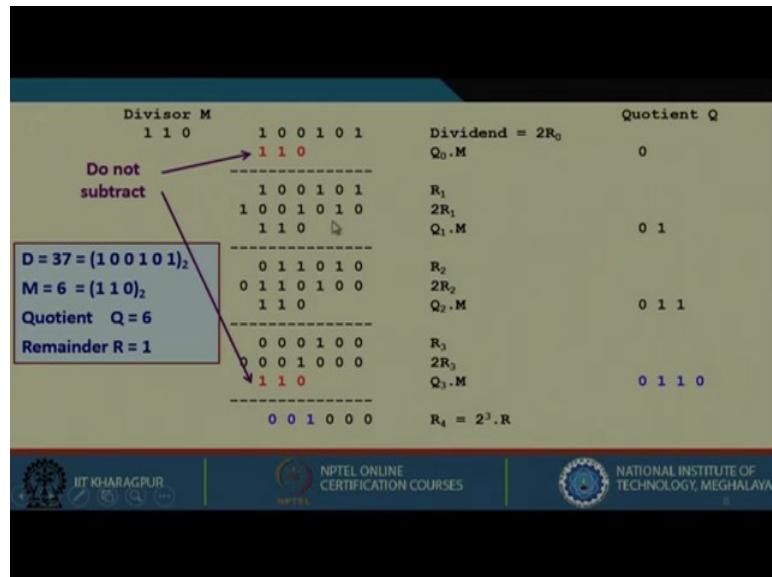
$$R_{i+1} = 2R_i - Q_i M \quad (\text{instead of } R_{i+1} = R_i - Q_i 2^I \cdot M)$$
 - The final partial remainder is the required remainder shifted to the left, so that $R = 2^j \cdot R_0$ (see next slide).

If we want to implement by hardware just like in multiplication algorithm we were keeping the partial product shifted and the multiplicand we are keeping in the same position, we are following the same principle here the partial remainders we shall be shifting, but the divisor will be keeping fixed in one position that will help us in implementing it in hardware.

So, here it is more convenient to shift the partial remainder to the left, but we keep the divisor in a fixed location, which means in the earlier case we are doing something like this right we were shifting the divisor to the right and we are subtracting from the partial remainder, but now we shall be shifting the partial remainder to the left; that means, multiplying by 2^I twice R_i , and we will be subtracting this divisor which is at a fixed location. So, there is no 2^I minus M here just Q_i multiplied by M if the next quotient bit is 0 we do not subtract anything if it is 1 we subtract the divisor

But here the only one change is there because we are not shifting this. The final partial remainder that remains actually to get the remainder from there the final remainder we have to shift it right by 3 places this is the only correction or the note we have to make.

(Refer Slide Time: 16:05)



So, here we have an example worked out that same example 37 divided by 6. Here you see 37 is the dividend here we expressed in 6 bits and divisor is we see that whether it goes 110 divisor is in fixed place you see 110 is in fixed place. So, it does not go. So, we said the next quotient bit to 0 and no subtraction.

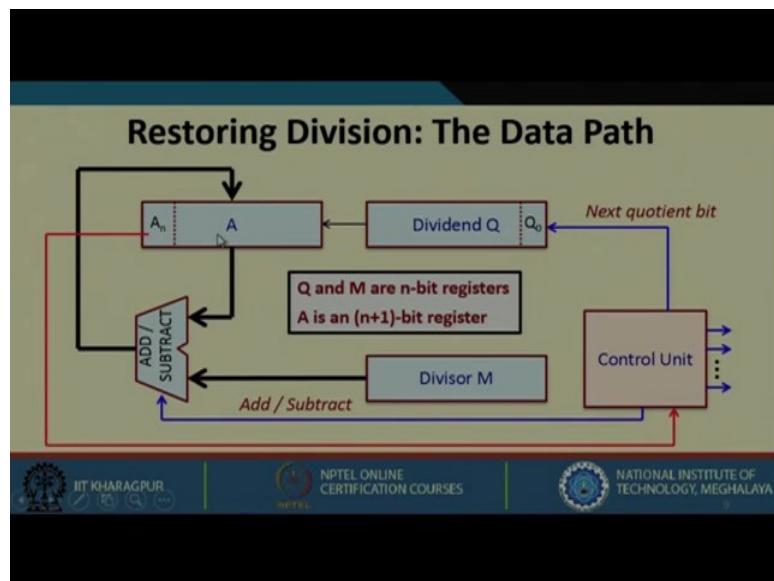
So, the partial remainder remains the same this is R 1 we shift R 1 to the left by 1 position shift left. So, this becomes 100, 101 and a 0 goes in. Now you again try to subtract 110 from here, here it goes. So, the next quotient bit will be 1 you do an actual subtraction 11 and 0 and these 3 bits remain this is my second partial remainder R 2 shift it left again twice R 2. So, again you subtract the divisor it again goes 110110. So, next quotient bit is again one. So, you subtract this is R 3 shift it left divisor try to subtract you see that it does not go.

So, next quotient bit is 0, and you do not subtract and whatever was there it remains. So, remainder is this. So, remainder actually to get the remainder will have to shift it right by 3 positions that was actually mentioned here. Remainder multiplied by 2 to the power 3 is actually what you are getting here. So, this will be my quotient and this will be my remainder. So, one thing you have seen from this example is that at every step we are

checking whether the divisor goes; that means, whether we can subtract it or not, but in an actual hardware circuit how we shall check that whether it goes or does not go we can do it by making a trial subtraction.

Let us do a subtraction and see that the result is becoming negative or not, if the result is becoming negative then you can say that it does not go if the result remains positive then it is fine because we are talking about unsigned numbers no negative number. So, far, but if you find it does not go and you have already subtracted. So, we will have to add the divisor back to restore the correction, restore the original value. So, a restoring step may be required as the correction.

(Refer Slide Time: 18:59)



So, here as it said we do not subtract here we do not subtract here these 2 places, and the concept that we are saying that we make a trial subtraction and then you can add it back to make the correction step when we see that it does not go, that can be very easily implemented by using a hardware circuit or a data path as we can see in this diagram. So, this diagram looks very much similar to the multiplier data path that is why we said that there is a very strong correspondence. So, we have a temporary register A which will be initializing to 0, the dividend we stored here and the divisor we store in another register and this A is an n plus 1 bit register one extra bit because we have to check this sign after trial subtraction.

So, what we do from a at every step we do a trial subtraction of the divisor we do a subtraction and check whether the result of the subtraction; that means, the sign bit is 0 or 1. So, the control unit will be checking that. So, if it sees that the result is negative then it will again activate an addition step.

So, the divisor will be added back to a to restore back the value; that means, I have done a subtraction wrongly I added back to restore the previous value this is basically what we are trying to do and this whole process we repeat and in every step we shift dividend and this a register left by one position, this was exactly what we are doing here right we are shifting this every step by one position, one position, one position we do exactly the same thing here right in hardware.

(Refer Slide Time: 20:54)

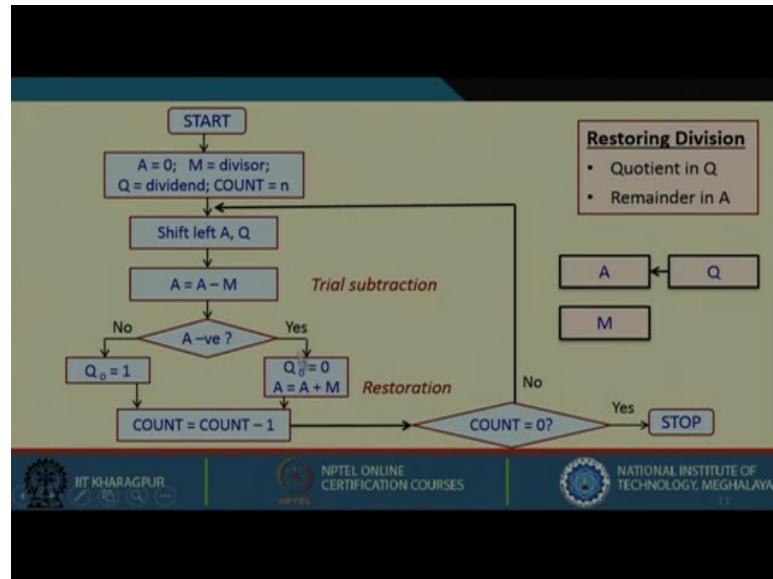
Basic Steps

Repeat the following steps n times:

- Shift the dividend one bit at a time starting into register A.
- Subtract the divisor M from this register A (*trial subtraction*).
- If the result is negative (*i.e. not going*):
 - Add the divisor M back into the register A (*i.e. restoring back*).
 - Record 0 as the next quotient bit.
- If the result is positive:
 - Do not restore the intermediate result.
 - Record 1 as the next quotient bit.

So, in terms of the steps; so we have this A register Q register and the divisor in another register M. So, we shift the dividend 1 bit at a time into register A. So, both a and Q we are shifting left subtract divisor M from A; that means, M is aligned to a we always subtract M from A this is our trial subtraction. If the result is negative which means it is not going then we add M back to register a this is the restoring step and because it was not going the next quotient bit; that means, last bit of Q we record as 0, but if the result is positive; that means, it goes you do not do any restoration or addition and record one as the next quotient bit ok.

(Refer Slide Time: 21:53)



This can be depicted in the flowchart form as follows the same thing which we said this you see we load a with 0 M contains the divisor Q contains the dividend.

So, we start by shifting left AQ; that means, the first bit of the dividend gets into a and we make a trial subtraction, this is your trial subtraction after shifting you make a trial subtraction then you check after trial subtraction whether a is becoming negative or not. If you say a is negative which means it does not go in that case the last bit of QQ 0 you set to 0 and you add M back to a this is a restoration step, but if you see a is not negative is positive after subtraction then it is fine you simply set the quotient bit to one and repeat this n times count was initialized to n decremented by 1 as long as it does not reach 0 you repeat this loop if it reaches 0 you stop right.

This is the basic restoration division algorithm. Now you see from this flowchart that you are looping n times, now every time you are starting by doing a subtraction and depending on the sign of a negative or positive you are either doing an addition or not doing an addition. So, we can say that on the average we shall be doing the corrective addition 50 percent of the time; that means, n by 2 time on the average.

(Refer Slide Time: 23:39)

- Analysis:
 - For n-bit divisor and n-bit dividend, we iterate n times.
 - Number of trial subtractions: n
 - Number of restoring additions: $n/2$ on the average
 - Best case: 0
 - Worst case: n

So, this is the analysis. So, for n bit divisor and n bit dividend we iterate n times. So, every time we carry out a trial subtraction; that means, there are n trial subtractions and on the average number of restoring additions will be $n/2$, because in the best case you will not need any restoration 0 worst case it will be n.

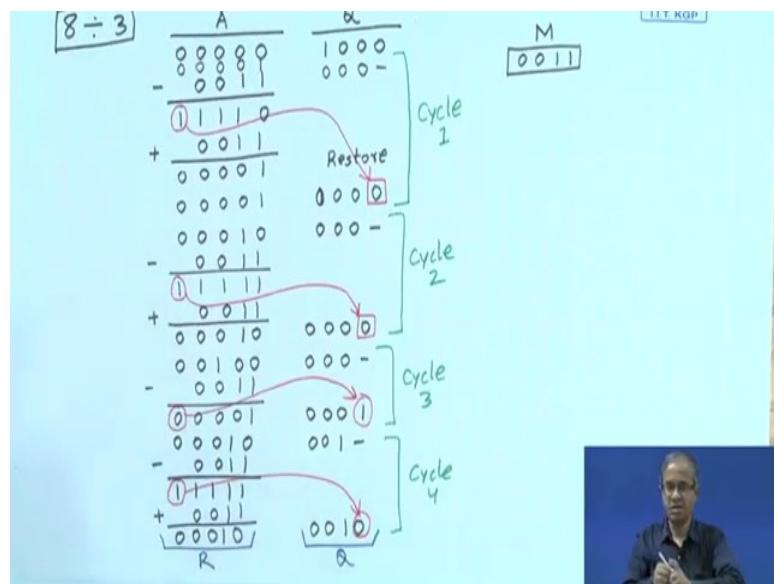
So, average will be $n + 0$ by $2n$ by 2 . So, the total number of addition subtraction will be $n + n$ by 2 here ok.

(Refer Slide Time: 24:13)

A Simple Example: 8/3 for 4-bit representation (n=4)			
Initially:	0 0 0 0 0	1 0 0 0	
Shift:	0 0 0 1 1		
Subtract:	<u>0 0 0 0 1</u>	0 0 0 -	
Set Q ₀ :	(1) 1 1 1 0		
Restore:	<u>0 0 0 1 1</u>		
Shift:	0 0 0 0 1	0 0 0 (0)	
Subtract:	<u>0 0 0 1 0</u>	0 0 0 -	
Set Q ₀ :	(1) 1 1 1 1		
Restore:	<u>0 0 0 1 1</u>		
	0 0 0 1 0	0 0 0 (0)	
Shift:	0 0 1 0 0	0 0 0 -	
Subtract:	<u>0 0 0 1 0</u>	0 0 0 -	
Set Q ₀ :	(0) 0 0 0 1		
Restore:	<u>0 0 0 0 0</u>		
Shift:	0 0 0 1 0	0 0 1 -	
Subtract:	<u>0 0 0 1 0</u>	0 0 1 -	
Set Q ₀ :	(1) 1 1 1 1		
Restore:	<u>0 0 0 1 1</u>		
	0 0 0 1 0	0 0 1 (0)	
			Remainder Quotient
		0 0 0 1 0 = 2	0 0 1 0 = 2

There is an example which is worked out here I shall actually just work out this example on paper just to show. So, I am illustrating this step says the same example I shall be working out. Finally, you get the remainder and quotient let us work out this example. So, we want to carry out a division operation 8 divided by 3.

(Refer Slide Time: 24:43)



So, what we do we start like this we have this a register A is a n plus 1 bit register suppose we are representing everything in 4 bits. So, a will be a 5 bit register initialize to 0 and side by side there is a Q register Q will contain the dividend 8 1 0 0 0 and the multiplicand here M is 3. So, M in binary is 0 0 1 1. So, you start the operation. So, you will have to make a trial subtraction from a first step. So, you subtract 0 0 1 1 from a this is a subtraction you are doing. So, after subtraction if you do a subtraction the result will be 1 1 1 1 0. So, what you see after subtraction is that your result is negative; that means, your sign bit is 1, because your sign bit is 1 you will have to restore it back; that means, you add 0 0 1 1 again to restore back the original value of course, means one thing I just missed out.

So, you have to start by making shifting. So, you first make a shift and then do this addition. So, this will be a shift 0 0 0 dash and then you do the addition subtraction. So, the 0 0 0 0 1 minus this will be this now it is correct. So, if you do a restoration it will become 0 0 0 0 1; that means, you get it back this is your restore step, because you are

restoring what you will happen is that this was your content of a and the Q content this bit will be 0.

So, the quotient bit will also become 0 because this bit is 1, this 1 will result in this bit also being 0; that means it is not going and this is your first step or the first cycle you can say cycle 1. So, you repeat this. So, repeat the same process. So, you shift it again left by 1 position. So, it becomes 0 0 0 1 0 this will be 0. So, this will be 0. So, this will be 0 0 0 dash. So, again we check whether this 0 0 1 1 is going or not. So, we make a trial subtraction again 0 0 1 1. So, so if you do a subtraction the result will be 1 1 1 1 1.

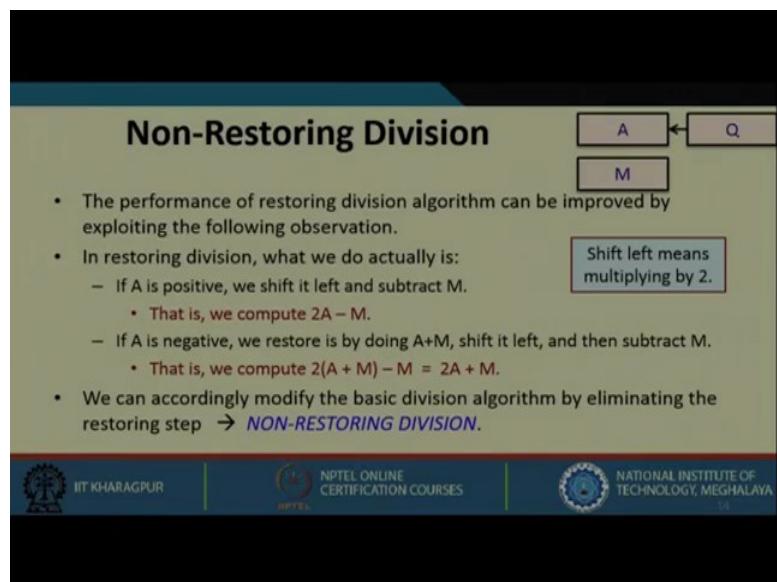
So, here again you see that your sign bit is one; that means, it is not going. So, you restore it back restore 0 0 1 1 add it again you will get back 0 0 0 1 0 and here 0 0 0 was there this last bit also will get 0 because this was also not going because of this 1 this bit will be 0. And this will be your cycle 2 this will have to repeat 4 times in the third cycle you continue the same process again shift it left 0 0 1 0 0 this is 0 you make a trial subtraction minus 0 0 1 1 now it goes 1 0 0 is greater than 0 0 1 1.

So, this trial subtraction if you make the result will be 0 0 0 0 1. So, now, this sign bit is 0 that means it goes; so you do not need any restoring addition here. So, what you do you simply this bit you fill up with 1 because it is going, the quotient bit has to be because of this 0 this bit will become 1 this will be your cycle 3. And the last cycle you again shift it left 0 0 0 1 0 you shift it left this will become 0 0 1 dash, again do a trial subtraction minus 0 0 1 1 if you do a subtraction it will be a 1 1 1 1 1 you see. So, here again the result is become negative it does not go right.

So, we will have to do a restoration 0 0 1 1 added back. So, you get 0 0 0 1 0 back and the last bit will become 0 again. So, this bit is 0 and this 1 determines it. So, you see this is your cycle number 4. So, you have finished 4 cycles and at the end of 4 cycles you get your remainder here and you get your quotient here 2 and 2 8 divided by 3 generates a remainder of 3 and a quotient also remainder 2 and a quotient of 2 right. So, this is how you can work out the algorithm step by step by hand.

Let us look into an improvement now, the restoration division algorithm that you have talked about here what we have seen is that we are doing subtraction every time in the iteration and on the average half a time we are also doing a corrective addition.

(Refer Slide Time: 31:32)



The slide is titled "Non-Restoring Division". It features a diagram showing three boxes: A (top), Q (middle), and M (bottom). An arrow points from Q to A, indicating the quotient is stored in A. To the right of the boxes is a note: "Shift left means multiplying by 2." Below the diagram is a bulleted list of observations:

- The performance of restoring division algorithm can be improved by exploiting the following observation.
- In restoring division, what we do actually is:
 - If A is positive, we shift it left and subtract M.
 - That is, we compute $2A - M$.
 - If A is negative, we restore it by doing $A+M$, shift it left, and then subtract M.
 - That is, we compute $2(A + M) - M = 2A + M$.
- We can accordingly modify the basic division algorithm by eliminating the restoring step → **NON-RESTORING DIVISION**.

At the bottom of the slide, there are logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Now, let us see if you can reduce the number of addition and subtraction our objective is to carry out only one operation every iteration total addition subtraction will be n only. So, we make an observation we observe that in the restoring division algorithm that you have already seen what you are doing is if means after the trial subtraction a was positive, we shifted left and subtract m; that means, shift left means we are doing 2 a then we are subtracting M shift left means multiplying by 2, but if a was negative of what we were doing we were first restoring it by adding back M a plus M, then shifting and then again subtracting M subtracting M for the next iteration.

So, this is this was done in the current iteration and again you go back next iteration you again do a subtract. So, if you combine the 2 what it means is that this a plus M is done shifted left means twice of that then subtract M minus M. So, this is 2 A plus M. So, effectively in one case you have been $2A - M$, and in the other case we are doing $2A + M$. So, if we can modify our algorithm by making this observation then we can reduce the number of operation this method is called non-restoring division.

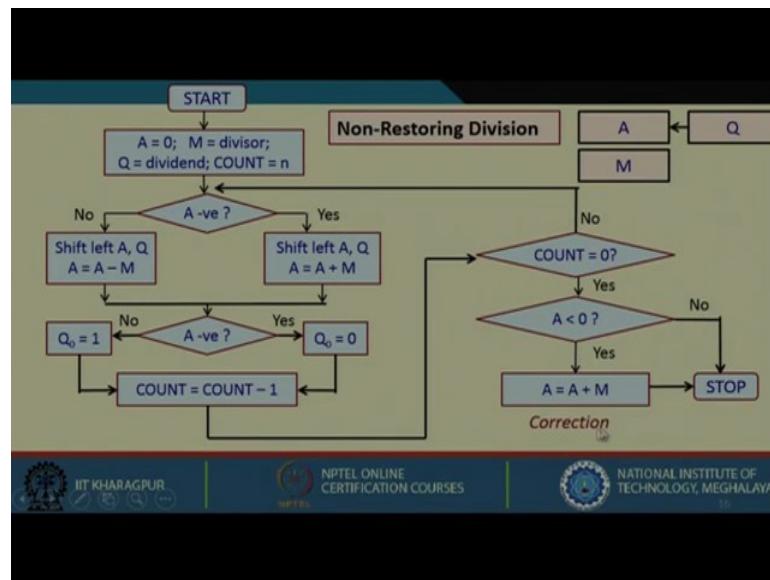
(Refer Slide Time: 33:26)

- Basic steps in non-restoring division:
 - a) Start by initializing register A to 0, and repeat steps (b)-(d) n times.
 - b) If the value in register A is positive,
 - Shift A and Q left by one bit position.
 - Subtract M from A.
 - c) If the value in register A is negative,
 - Shift A and Q left by one bit position.
 - Add M to A.
 - d) If A is positive, set $Q_0 = 1$; else, set $Q_0 = 0$.
 - e) If A is negative, add M to A as a final corrective step.

So, let us see what we do here we similarly start by initializing register a to 0 and repeat this steps bc and D n number of times. The method is simple if a is positive you shift a and Q left by 1 position and subtract M from A, but if it is negative you add M to a this was the observation in one case we subtract M other case you add M then you check if after this addition or subtraction a is positive or negative if it is positive set the quotient bit to 1 else set the quotient bit to 0.

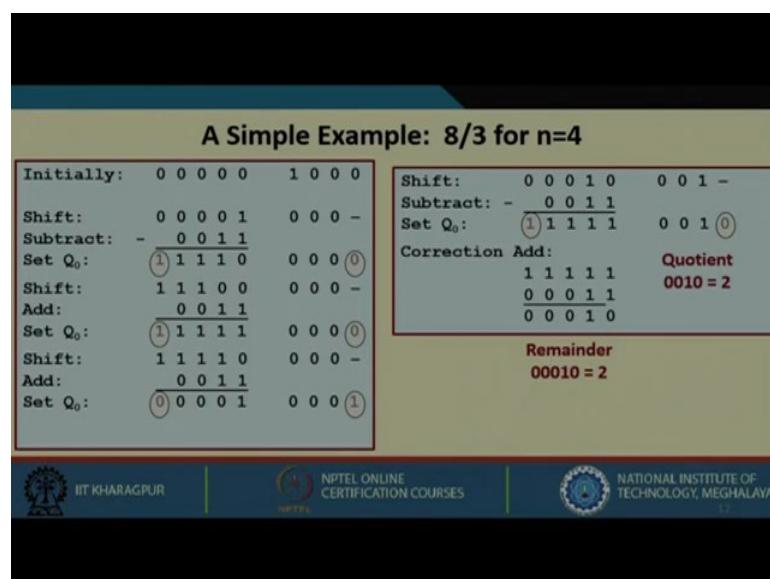
Now, see here we are including one additional subtraction which is there in the next iteration with this operation. So, for the last time there may be an addition subtraction operation which we have carried out. So, we may have to carry out a corrective addition only at the very end. So, that we are doing here if at the very end a is negative; that means, we have carried out a wrong last subtraction we have to make a corrective addition at the end right.

(Refer Slide Time: 34:44)



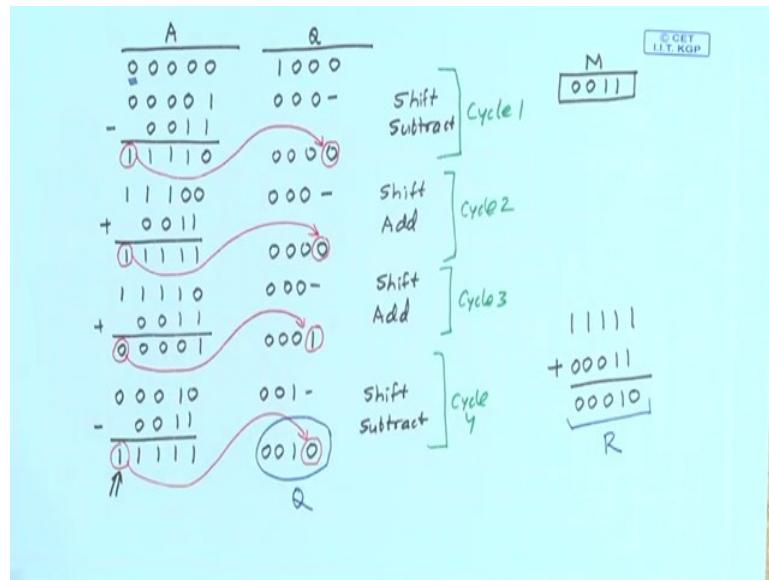
So, in terms of flow chart, so whatever we just wrote here I just showing it diagrammatically like this we initialize the registers as earlier, we check a is negative or not if yes we shift left and add if no shift left and subtract then we check whether a is still negative or positive if it is negative the quotient bit is set to 0 or quotient bit set to 1. This thing you repeat n times and after you complete n times you check finally, whether a is still negative or not, if it is still negative you have a corrective addition step.

(Refer Slide Time: 35:39)



This is what the algorithm is. So, here again I shall be working out this simple example 8 by 3 by hand. So, the same example I will be taking I am just showing this animation first fine. So, let us work this out again.

(Refer Slide Time: 35:59)



So, here just exactly similar to what we did for the restoring division we start with this register a containing all zeros and register Q containing the dividend and you recall our divisor was 3 which was 0 0 1 1 this was M. So, according to the algorithm let us go back to the algorithm, we shall be just looking into we are starting with by checking whether a is negative or not accordingly we do a shift and then addition or subtraction.

So, we are doing that same thing. So, we are seeing that the number is positive. So, we will be doing a shifting and then a subtraction, if you do a shifting this becomes 0 0 0 0 this 1 comes here this will be 0 0 0 dash this is your shifting then you do a subtraction subtract M from this minus 0 0 1 1. So, this result will be 1 1 1 1 0 and this 0 0 you see after subtraction your sign is one. So, your next bit here will be 0. So, this one will decide the next quotient with 0 you see the flowchart once more that here we do a subtraction and after subtraction we check whether a is negative or not when if it is negative we set Q 0 to 0 exactly that is what we are doing.

So, because the negative we are setting Q 0 to 0. So, let us continue with this. So, we have a shift step we have a subtract step this is your cycle 1. So, you repeat this step shift left again 1 position now this is 1 that is how you have to shift, and add if it is 0 we will

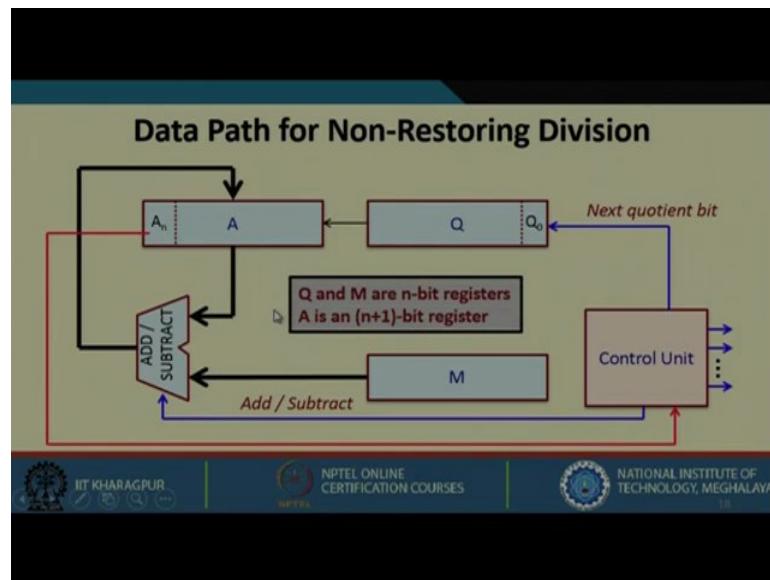
be subtracting if it is 1 we will be adding. So, we do a shift first it will be a 1 1 1 0 0 this 0 will come in 0 0 0 dash and we will be doing addition here plus 0 0 1 1. So, this will be 1 1 1 1 1 and 0 0 0 see here again you see that the sign is one. So, the next bit that will be coming in the quotient will be 0 just like in the previous case.

This 1 will be deciding this quotient bit and this is your cycle 2. So, in cycle 2 we have done a shift we have done an add now this is plus. So, again we have to do an add shift an add if it is 0 we will do a minus if it is a 1 we will do a plus. So, first again shift left. So, it will be a 1 1 1 1 0 0 0 0 dash this is shift then add 0 0 1 1. So, after adding we will be getting 0 0 0 0 1 and the next quotient bit will be a one because this sign bit is now 0. So, if the sign bit is 1 quotient will be 0 if it is 0 it will be a 1.

So, we are doing a shift and then we are doing an add this is your cycle 3, and in the last cycle you again continue with the same thing this is 0. So, now, I will be doing a shifting and then subtracting because it is 0. So, we do a shift first 0 0 0 1 0 this will be also shifted 0 0 1 dash. Now we will be doing the subtraction minus 0 0 1 1. So, if you do the subtraction this will be will become all ones and because it is 1 this bit will become 0. So, again this is 1 this 1 will decide this is becoming 0.

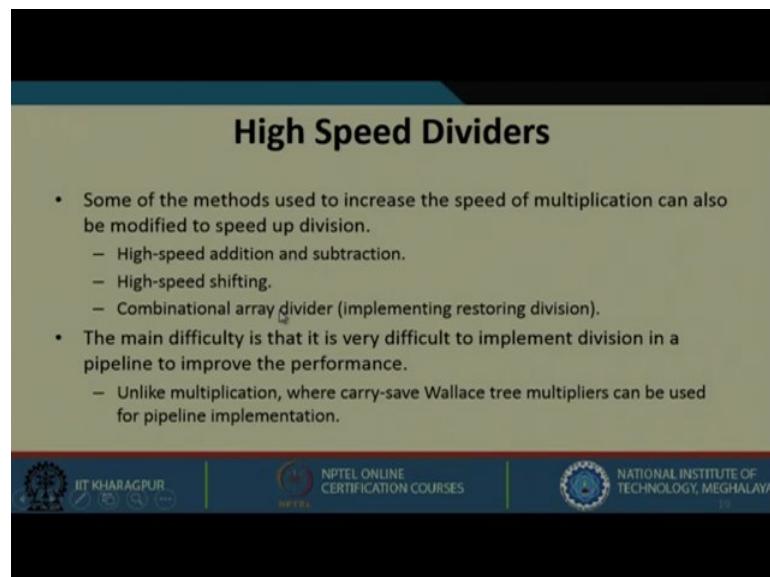
So, we have completed 4 cycles right. So, in the last cycle we have done a shift and we have done A subtract. So, after completing 4 cycles we see that our sign bit of A is still 1 which means we need that correction step. So, 1 1 1 1 1 we need to have a corrective addition step we add this 1 1 to this, if you add this you will be getting 0 0 0 1 0 which will be your final remainder this will be a remainder, and already whatever is there in the Q register this will be your quotient 2 and 2. So, you see this step by step we have seen how the restoring division algorithm works.

(Refer Slide Time: 42:12)



So, regarding the data path, data path is exactly identical to what we saw for the restoring algorithm. So, there is no difference the same hardware can be used only your control unit will be different the advantage is that we need only n number of addition or subtraction steps here right.

(Refer Slide Time: 42:32)



Now, one thing means if we want to go for high speed division some of the methods that we have already seen to design high speed multipliers, can be used here also like you

can use high speed adders and subtractors using carry look at adders carry select adders etcetera.

We can use high speed shifters like barrel shifters, we can use combinational array dividers which are similar to combinational array multipliers and they means actually can implement the restoring division algorithm, but the problem with division algorithm is that you can do it, but it is extremely uneconomical and means ineffective to build a combinational array kind of a divider, we service what we could do for a for a multiplier for multiplying 2 numbers using carries evaders. The other thing is that the including signed sign in the division process; that means, signed division algorithm is not easy normally it is done using a separate step by separately checking the sign bits and making a corrective step at the end to correct the sign of the product or the divisor remainder in the quotient.

So, sign division is again a problem. So, the main difficulty that you have seen earlier in the very beginning of the lecture we said that for division the latency as well as the number of cycles per issue are very high, that is mainly because of the difficulty in implementing the division algorithm in a pipeline multiplication can be very effectively implemented in a pipeline in a Wallace tree kind of a multiplier that is how you have seen that the number of cycles per issue can be multiple for multiplications can be as low as 1 every cycle you can feed a new data to a multiplier and the total latency will be 3 it takes 3 cycles to complete the multiplication.

But for division it is not so, division still remains the bottleneck. So, as a programmer whenever you are developing some applications or programs you should keep this in mind division is an expensive operation, you should replace division by other arithmetic operations wherever possible that can increase the effectiveness of your program or application in terms of the speed and performance.

So, in the next lectures we shall be moving on to. Firstly, the floating point operations. So, how you can extend whatever we have learned so far to handle floating point numbers, numbers with decimal points and suddenly we shall be looking at other logical operations how everything can be integrated within the arithmetic logic unit. So, we come to the end of this lecture.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 38
Floating - Point Numbers

In the last few lectures we have seen various arithmetic algorithms and techniques, using which we can carry out various operations on integer numbers. Specifically we looked at the design of adders, subtractors, multipliers and also dividers.

There we had seen basically how operations on integer quantities can be carried out. Today we shall be extending that concept and in this lecture we shall be specifically talking about something called floating-point numbers where we can handle fractional quantities as well. So, the topic of this lecture is floating point numbers.

(Refer Slide Time: 01:13)

Representing Fractional Numbers

- A binary number with fractional part
$$B = b_{n-1} b_{n-2} \dots b_1 b_0 . b_{-1} b_{-2} \dots b_{-m}$$
 corresponds to the decimal number
$$D = \sum_{i=-m}^{n-1} b_i 2^i$$

If the radix point is allowed to move, we call it a **floating-point representation**.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, ROURKELA

Let us see the motivation first; well here as we said we want to represent a binary number with a fractional part. In general any binary number with a fractional part can be represented like this where these b_0, b_1, \dots are the binary digits; this dot is the binary radix point.

So, this side you have the integer part and this side you have the fractional part. If you want to convert this binary number into equivalent decimal, you recall that every binary

position has a weight. b_0 has a weight of 2 to the power 0, b_1 as a weight of 2 to the power 1, and so on. Similarly on the fractional side b_{-1} has a weight of 2 to the power -1, b_{-2} has a weight of 2 to the power -2, and so on.

So, you can write the decimal equivalent of this number as this the digit multiplied by the weight 2 to the power i ; where i can vary from $-m$ up to $n-1$. This kind of a representation is also called fixed point numbers because the position of this radix point is fixed in the number representation, but if we allow this radix point to move (not fixed in a particular position), then we refer to this number representation as a floating point number representation.

This is the basic idea behind the so called floating point number representation. Let us look at some examples.

(Refer Slide Time: 03:02)

Some Examples

1011.1	$\rightarrow 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1}$	$= 11.5$
101.11	$\rightarrow 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$	$= 5.75$
10.111	$\rightarrow 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$	$= 2.875$

Some Observations:

- Shift right by 1 bit means divide by 2
- Shift left by 1 bit means multiply by 2
- Numbers of the form $0.11111\dots_2$ has a value less than 1.0 (one).

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY ROURKELA

You look at a binary numbers with fractional part, 1011.1. If you want to converted into decimal you will have to multiply each digit position by its weight 2 to the power 3, 2 to the power 2, 2 to the power 1, 2 to the power 0, and for the fractional part 2 to the power -1. So, the value becomes 11.5.

Similarly this number 101.11 we will have a value again multiplied by the weights; this will be 5.75 .

If you take a number 10.111, the weights will be this.

The value will be 2.875. Now one thing you see across the examples I have taken that the decimal point is actually shifting left by one position from this number to this, this number to this. So, when the decimal point or the radix point shifts left by one position if you look at the corresponding value, you see that this means a division by 2. 11.5, if you divide by 2, you get 5.75. 5.75, if you divide by 2, you get 2.875.

So, shifting right by 1 bit; that means, if you move this radix point to the left or right this will mean divide by 2, or multiply by 2. If you move the radix point to the left it will be divide by 2; if you move the point to the right it will be multiply by 2; and another point to notice that if you have a number of this form 0.11111 this will have a value less than 1.

(Refer Slide Time: 05:19)

A handwritten derivation of the sum of a geometric progression (GP) series. It starts with the binary fraction $0.\overline{11111\dots}$. This is shown as a sum of powers of 2: $2^{-1} + 2^{-2} + 2^{-3} + \dots$. This is then equated to a GP sum formula: $= \frac{\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots}{1 - \frac{1}{2}}$. The denominator is simplified to $= \frac{\frac{1}{2}(1 - \frac{1}{2^n})}{1 - \frac{1}{2}}$. The term $\frac{1}{2^n}$ is circled in blue and has an arrow pointing to 0, indicating it approaches zero as n increases. The final result is given as $| - \epsilon \rightarrow |$.

Why this, you see if I write such a number 0.111... say there are several binary digits. If you look at the weights it is 2 to the power minus 1, then 2 to the power minus 2, then 2 to the power minus 3, and so on. You see this is a GP series; this is actually half plus one fourth, plus one 8th plus like this. So, if you calculate the sum of this series we get 1 minus 1 by 2 to the power n, where n is the number of digits.

You see as the value of n increases, as we increase this number of digits 1 1 1 1 1, the value of this second quantity tends to 0 because 2 to the power n increases rapidly. So, this is a fraction; you can write as 1 minus a very small value epsilon. This actually tends

to 1 as n tends to infinity. This is exactly what we are meaning here; a number of this form we will always have a value less than 1, in the limit it will approach 1.

(Refer Slide Time: 06:57)

Limitations of Representation

- In the fractional part, we can only represent numbers of the form $x/2^k$ exactly.
 - Other numbers have repeating bit representations (i.e. never converge).
- Examples:

$3/4 = 0.11$	
$7/8 = 0.111$	
$5/8 = 0.101$	
$1/3 = 0.10101010101 \dots$	
$1/5 = 0.001100110011 \dots$	
$1/10 = 0.0001100110011 \dots$	

- More the number of bits, more accurate is the representation.
- We sometimes see: $(1/3)*3 \neq 1$.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, TRIVANDRUM

Now, in the representation there are some limitations let us look at this now. In the fractional representation in binary if you look little carefully you will see that only numbers which can be represented in the form x divided by 2^k , where the denominator is some power of 2, those numbers can be precisely represented in binary; like $3/4$ where 4 is a power of 2 --- you can express as 0.11.

$7/8$, 8 is a power of 2 as 0.111. $5/8$ as 0.101, but if the denominator is not a power of 2 like here if I take examples of 3, 5 and 10. So, if you go on expanding decimal to binary you will see that it will never terminate. So, this $1/3$ this 1 0 1 0 1 0 alternates. $1/5$ uses 0 0 1 1 0 0 1 1 alternates. $1/10$ also you will see after some point 0 0 1 1 0 0 1 1 keeps alternating. So, more the number of bits you use for the representation more accurate will be our representation, but one thing we should also remember that when you are representing $1/3$ in a computer like this, we have a finite number of bits to represent. So, we can never represent $1/3$ exactly, it will be very close to $1/3$.

The point is that you are not able to represent some of the fractions in a precise way; there is an error in the representation. By virtue of that what might happen is that suppose in a program you are diving $1/3$ and the result you are again multiplying by 3. You expect that the final result should be 1, but you might see that the final result is not

1, but close to 1; and this is because of this truncation error. You can see we have finite number of bits to represent; this 1/3 you cannot represent precisely.

(Refer Slide Time: 09:22)

Floating-Point Number Representation (IEEE-754)

- For representing numbers with fractional parts, we can assume that the fractional point is somewhere in between the number (say, n bits in integer part, m bits in fraction part). → *Fixed-point representation*
 - Lacks flexibility.
 - Cannot be used to represent very small or very large numbers (for example: 2.53×10^{-26} , $1.7562 \times 10^{+35}$, etc.).
- Solution :: use floating-point number representation.
 - A number F is represented as a triplet $\langle s, M, E \rangle$ such that
$$F = (-1)^s M \times 2^E$$

Now, let us come to the so called floating point number representation, which is around for quite some time. IEEE has come up with the standard. So, whatever we shall be discussing it will be basically based on this standard.

Let us try to see what this standard says. Firstly, we already talked about the fixed point representation earlier. So, there we had said that if we have a number with a fractional part, then we can have n bits in the integer part and m bits in the fractional part. So, the radix point is fixed, but in this representation we have limited amount of flexibility. Why? Because we cannot represent for instance very small or very large numbers, which occur very frequently in scientific computations. Like for instance I want to represent number 2.53×10^{-26} or $1.7562 \times 10^{+35}$, these kind of numbers where either the number is very large or number is very small, after the radix point there are many zeros, after that only the significant digits will start. So, these kinds of numbers you cannot represent in a finite number of bits, where the radix point is fixed in a location. For this you need something else.

So, what is the solution? The solution is to use this so called floating point number representation. Here a number F is represented as a triplet $\langle S, M, E \rangle$, but the value of the number is minus 1 to the power S into M multiplied by 2 to the power E .

(Refer Slide Time: 11:39)

F = $(-1)^s M \times 2^E$

- s is the *sign bit* indicating whether the number is negative (=1) or positive (=0).
- M is called the *mantissa*, and is normally a fraction in the range [1.0,2.0].
- E is called the *exponent*, which weights the number by power of 2.

Encoding:

- Single-precision numbers: total 32 bits, E 8 bits, M 23 bits
- Double-precision numbers: total 64 bits, E 11 bits, M 52 bits

S	E	M
---	---	---

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MADRAS

Coming again to this representation as I said s is the sign bit. This indicates whether the number is negative (for that s is 1) or it is a positive (s is 0). This M is conventionally known as the mantissa and I will explain a little later why this mantissa is usually a fraction, which is in the range 1.0 to 2.0.

And E is called the exponent, which is weighted as you can see by a power of 2. 2 to the power E; and in the IEEE floating point format you can have either single precision numbers or double precision numbers. The general format looks like this. The number starts with the sign bit, followed by some number of bits for the exponent, and finally some bits for the mantissa. For single precision numbers the total size is 32 bits, for E you have 8 bits and for M you have 23 bits.

(Refer Slide Time: 13:09)

Points to Note

- The number of *significant digits* depends on the number of bits in M.
 - 7 significant digits for 24-bit mantissa (23 bits + 1 implied bit).
- The *range* of the number depends on the number of bits in E.
 - 10^{38} to 10^{-38} for 8-bit exponent.

How many significant digits? $2^{24} = 10^x$ $24 \log_{10} 2 = x \log_{10} 10$ $x = 7.2 \text{ -- 7 significant decimal places}$	Range of exponent? $2^{127} = 10^y$ $127 \log_{10} 2 = y \log_{10} 10$ $y = 38.1 \text{ -- maximum exponent value}$ 38 (in decimal)
--	--

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, ROURKELA | Prof. Arun Kumar Ray

But for double precision numbers you have 64 bits in total, where E is 11 bits and M is 52 bits. Now just going back once the number of significant digits in your representation will depend on how many bits you are using for the mantissa. This is the number of significant digits, it will depend on how many bits you have in M.

Now, although for single precision number we have kept 23 bits in the mantissa, but actually the idea is as follows. The mantissa is represented as a number that always starts with a 1.

(Refer Slide Time: 13:44)

A handwritten note on a grid background. At the top right is a small box containing "© CET I.I.T. KGP". Below it, a binary mantissa is shown as "1. 0101100..." with a bracket underneath labeled "23" and an arrow pointing down labeled "Implied". To the right, the exponent is listed as "EXP: -128 to +127" and "-127 to +127". Below that, the exponent range is given as "E: 0 to 254".

And there is an implied radix point here and after that you can have anything, 0 1 0 1 1. The first digit is always 1. We assume that this is an implied bit and because it is always 1 you do not store this; you store only the remaining number of bits. So, here for single precision there are 23 bits here, but if you also count this 1 it becomes 24. So, the number actually is a 24-bit number, but because it always starts with a 1 you are actually storing 23 bits. So, when you are calculating you will be assuming 24-bit mantissa because that implied 1 bit is also there. Now how do you calculate number of significant digits is very simple. In binary you have 24 bits, in decimal how many bits?

You use this; this equation 2 to the power 24 equal 10 to the power x . You take logarithm on both sides, and get $x = 7.2$. This means in decimal you can have 7 significant digits, but if you think of double precision number where for mantissa we have 52 bits. 52 plus 1 will be 53; 53 multiplied by $\log 2$. So, it will be much larger --- 15 or 16 digits you will have for significant digits.

Similarly for exponent in single precision number you have 8 bits, and this exponent can be either positive or negative 2 to the power plus something or 2 to the power minus something. This 8 bit is actually you can regard it that is the 2 's compliment signed integer; range will be -128 to +127.

If you want to find out the range in decimal you follow a similar calculation, 2 to the power 127 is equal to 10 to the power y . Finally y comes to about 38 point something. So, in decimal equivalently you can have maximum exponent value as 38 which means in single precision you can represent up to 7 significant decimal places and the range of the number can be 10 to the power 38 to 10 to the power -38.

(Refer Slide Time: 16:50)

“Normalized” Representation

- We shall now see how E and M are actually encoded.
- Assume that the actual exponent of the number is EXP (i.e. number is $M \times 2^{EXP}$).
- Permissible range of E : $1 \leq E \leq 254$ (the all-0 and all-1 patterns are not allowed).
- **Encoding of the exponent E :**
 - The exponent is encoded as a biased value: $E = EXP + BIAS$
where $BIAS = 127$ ($2^{8-1} - 1$) for single-precision, and
 $BIAS = 1023$ ($2^{11-1} - 1$) for double-precision.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY WARANGAL

Now in floating point number there is an interesting concept called normalization; let us look at the encoding part of it that how the exponent and mantissa are actually stored or encoded. Let us assume that the actual exponent of the number is EXP. The number is M multiplied with 2 to the power EXP. We are encoding this EXP in some way and we get E. Now the value of E can range from 1 up to 254; it is an 8 bit number you know for unsigned number the range is 0 up to 255, but here we are saying 1 up to 254, which means we are leaving out 0 and 255.

So, the all 0 and all 1 patterns are left out; they are not allowed. So, how we are encoding E, you have your actual exponent EXP; we are adding a bias to it. You see EXP can be either negative or positive, but after we add this bias E becomes always positive. So, how much is the bias? For 8-bit exponents for single precision you take the bias as 127. So, because you see I mean say again if you look at the value of the actual exponent it can range from -128 to +127. Now this one combination you leave out let us say it is from -127 to +127. So, if you add a bias 127 to it, $-127 + 127$ become 0, and $127 + 127$ becomes 254. So, this is positive.

The bias is 127 for single precision, and it is $2^{11-1} - 1$ which is 1023 for double precision, because in double precision we have 11 bits for the exponent.

(Refer Slide Time: 19:31)

- **Encoding of the mantissa M:**
 - The mantissa is coded with an implied leading 1 (i.e. in 24 bits).
 $M = 1 . xxxx...x$
 - Here, xxxx...x denotes the bits that are actually stored for the mantissa. We get the extra leading bit for *free*.
 - When xxxx...x = 0000...0, M is minimum (≈ 1.0).
 - When xxxx...x = 1111...1, M is maximum ($= 2.0 - \epsilon$).

Now, for the mantissa as had said earlier we encode the mantissa in such a way that it always starts with a leading 1 and whenever you encode mantissa in this way we say that the number is normalized. If the mantissa starts with 0 we say that it is not normalized; and this x x x denote the bits that are actually stored because the first bit is always 1; we do not store it explicitly. We are getting this extra bit for free, we are storing only the remaining 23 bits.

When the value of x is all 0's the value of M is minimum 1.0 0 0 0 0 the value is 1. When x is all 1's; that means 1.11111. As we saw earlier point 1 1 1 1 1 is the value which is very close to 1. The value of this number becomes very close to 2; 2 minus very small value epsilon. You recall we mentioned earlier that the mantissa M will be having a value between 1 and 2, and this is why it is so.

(Refer Slide Time: 20:54)

An Encoding Example

- Consider the number $F = 15335$
 $15335_{10} = 1110111100111_2 = 1.110111100111 \times 2^{13}$
- Mantissa will be stored as: $M = 110111100111 0000000000_2$
- Here, EXP = 13, BIAS = 127. $\rightarrow E = 13 + 127 = 140 = 10001100_2$

0 10001100 1101111001110000000000 466F9C00 in hex

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY WARANGAL

Let us look at some encoding examples. We consider a number F let us say 15335. If you convert this into binary it is like this, and if you expressed it in fractional form, if you put a decimal point here the radix point here. So, if you count the digits into 2 to the power 13 there are 13 binary bits here.

So, if you express the number like this you see the mantissas already normalized. It starts with 1; so you are not storing this 1 you are storing the remaining bits. So, the mantissa will be stored as this. How many bits will be there for the mantissa? 23 bits and exponent is 13. So, bias for single precision is 127. So, the actual value of E will be $13 + 127 = 140$. This number will be stored as sign positive, followed by exponent, followed by mantissa. If you divide this number into 4 4 bits and convert into hexadecimal you will see you will start is 0 1 0 0 which is 4, then 0 1 1 0 which is 6 ,again 0 1 1 0 which is 6, 1 1 1 1 which is F, and so on.

(Refer Slide Time: 22:24)

The slide has a yellow header with the title "Another Encoding Example". Below it is a bulleted list:

- Consider the number $F = -3.75$
 $-3.75_{10} = -11.11_2 = -1.111 \times 2^1$
- Mantissa will be stored as: $M = 111000000000000000000000_2$
- Here, EXP = 1, BIAS = 127. $\rightarrow E = 1 + 127 = 128 = 1000000_2$

Below the list is a binary representation in a box:
1 1000000 11100000000000000000000000000000
40700000 in hex

The footer contains logos for IIT Kharagpur, NPTEL, and National Institute of Technology, Tiruchirappalli, along with a video player showing a person speaking.

Let us take another example where the number is negative -3.75, which in binary you can write like this. If you express in normalized form push the decimal point here it becomes like this. So, we will be storing the mantissa as only this 1 1 1 this one we do not store and the remaining zeros and exponent EXP is 1, you add the bias it becomes 128 which is this.

So, the number will be represented as sign is one, negative exponent and mantissa which in hexadecimal is C070.

(Refer Slide Time: 23:19)

The slide has a yellow header with the title "Special Values". It lists two cases:

- When E = 000...0
 - M = 000...0 represents the value 0.
 - M ≠ 000...0 represents numbers very close to 0.
- When E = 111...1
 - M = 000...0 represents the value ∞ (infinity).
 - M ≠ 000...0 represents *Not-a-Number* (NaN).

Callout boxes provide additional information:

- Zero is represented by the all-zero string.
- Also referred to as *de-normalized* numbers.
- NaN represents cases when no numeric value can be determined, like uninitialized values, $\infty * 0$, $\infty - \infty$, square root of a negative number, etc.

The footer contains logos for IIT Kharagpur, NPTEL, and National Institute of Technology, Tiruchirappalli, along with a video player showing a person speaking.

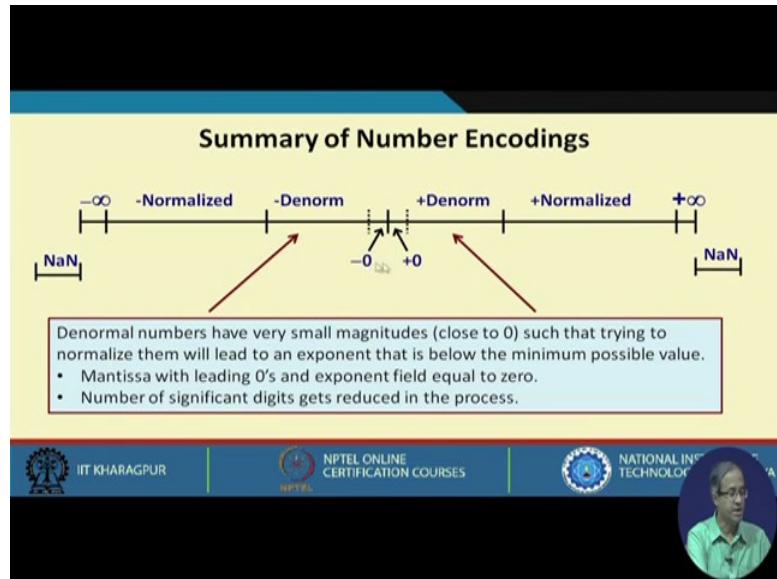
Now, some special values are supported. When we saw earlier we mentioned that the values of E all 0's and all 1's are not allowed in actual numbers, but what happens when the value of E is all 0's or all 1's. So, these are the so-called special values. When E is all 0's then if M is also all 0's this combination represents this special number 0. So, you see the number 0 is represented as an all 0 string. It can be easily checked by hardware. And for E all 0's if we have a mantissa which is not all 0's this represents some numbers which are very close to 0, because see $E = 0$ means it is already in biased format means 2 to the power minus 127.

After adding 127 you are getting 0. So, the magnitude of the number is really very small. Here we are representing numbers which are very close to 0 when E is all 0 these are sometimes called de-normalized numbers because if M is all 0, you do not have a 1 at all you cannot make the first bit 1.

So, for these de-normalized numbers your exponent has to be all 0's meaning that this is a special kind of a number where the mandatory requirement of the first bit of the mantissas 1 is not to be assumed in this case, and 0 is represent with all 0 string.

When E is all 1's then the all 0's combination of the mantissa represents the value infinity; this is our representation in the IEEE format, and any value which is not equal to 0 this represents an invalid number which means not a number sometimes it is called NaN. Why it is required? Because you see there is certain cases where you do not know the value of a number like you have defined some variables, but not initialized them.

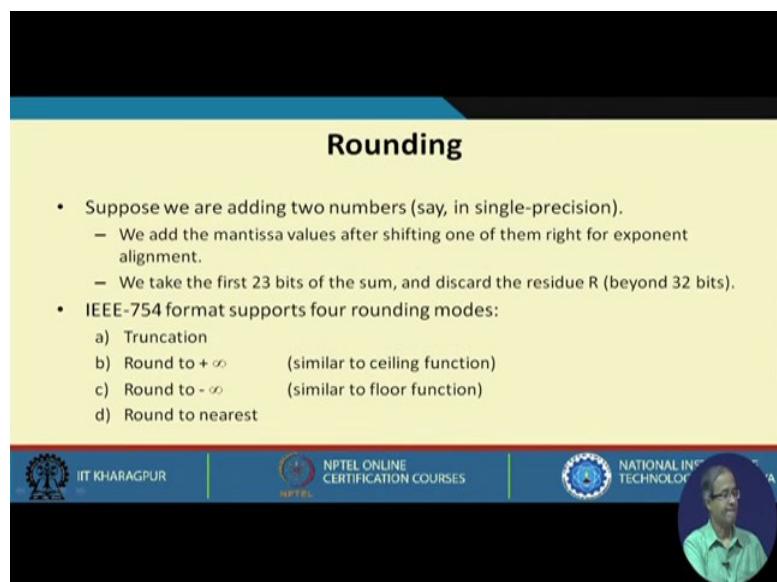
(Refer Slide Time: 26:07)



In this scale this summary of number encodings you can see that in between when you have the exponent and mantissa values all 0, you have the number 0. So, there are two representations of 0, +0 and -0. And for very small numbers you can have de-normalized numbers; and when you have normalized numbers it will be beyond that and you can go up to plus infinity, plus infinity is a special representation; minus infinity is also special representation.

Beyond that you have invalid numbers which are NaN.

(Refer Slide Time: 26:54)



Now, there is another feature of the IEEE format, which is also very much useful this called rounding. Suppose when we add two numbers in single precision. Normally we add the mantissa value after aligning the exponents. What do you mean by exponent alignment? Suppose my one number is 1 . 1 0 0 into 2 to the power 2.

(Refer Slide Time: 27:18)

$$\begin{array}{r} 1.100 \times 2^2 \\ 1.011 \times 2^1 \Rightarrow \\ \hline \end{array}$$

$$\underline{1.100} \times 2^2$$

Let us say other number is 1 . 0 1 1 into 2 to the power 1. So, when we want to add we cannot straight add because the exponents and not same. What we do this second number we make it 2 to the power 2, and for that we shift the decimal point to the left by one position.

So, it becomes 0.1011 then we can add 1.100 with this, and we get the result. This is what is mentioned here. We add the mantissa value after shifting one of them. We shall be coming to this again. After adding, the first 23 bits of this sum is taken and the remaining bits are discarded; this is called the residue. Now IEEE format supports four different rounding modes, one is truncation --- beyond 23 bits you just discard the remaining bits this is truncation, rounding to plus infinity is the second mode what it says is that you look at r if you see where there r is greater than or equal to 0.5. If r is greater than or equal to 0.5 just like ceiling function, you increase mantissa by 1. Similarly if it is less than 0.5 we will rounding to minus infinity you move it to the next lower integer value, and second one is a rounding.

So, depending on the value of the r here you are either moving into the next higher or the next lower, and rounding means if it is 0.5 or higher you move it up, otherwise you move it down.

(Refer Slide Time: 29:14)

The slide contains a bulleted list of points related to rounding:

- To implement rounding, two temporary bits are maintained:
 - Round Bit (r):* This is equal to the MSB of the residue R .
 - Sticky Bit (s):* This is the logical OR of the rest of the bits of the residue R .
- Decisions regarding rounding can be taken based on these bits:
 - $R > 0:$ If $r + s = 1$
 - $R = 0.5:$ If $r.s' = 1$
 - $R > 0.5:$ If $r.s = 1$ \therefore // ' + ' is logical OR, ' . ' is logical AND
- Renormalization after Rounding:
 - If the process of rounding generates a result that is not in normalized form, then we need to re-normalize the result.

The footer of the slide includes logos for IIT Kharagpur, NPTEL, and National Institute of Technology, Tiruchirappalli, along with a video player showing a speaker.

This is rounding to the nearest. To implement this two temporary bits are used in the representation, one is called round bit r which is the most significant bit of the remaining residue R , and sticky bit s which is actually the logical OR of the remaining bits of R other than the MSB.

(Refer Slide Time: 30:24)

The slide title is "Some Exercises". Below the title is a numbered list of exercises:

- Decode the following single-precision floating-point numbers.
 - 0011 1111 1000 0000 0000 0000 0000 0000
 - 0100 0000 0110 0000 0000 0000 0000 0000
 - 0100 1111 1101 0000 0000 0000 0000 0000
 - 1000 0000 0000 0000 0000 0000 0000 0000
 - 0111 1111 1000 0000 0000 0000 0000 0000
 - 0111 1111 1101 0101 0101 0101 0101 0101

The footer of the slide includes logos for IIT Kharagpur, NPTEL, and National Institute of Technology, Tiruchirappalli, along with a video player showing a speaker.

Here are some exercises for you to work out.

We have seen some representations of floating point numbers and in particular the IEEE format that is almost universally used nowadays in almost all computer systems. We have seen how numbers are represented, how some special numbers are represented that are very useful during computations and also how we can do rounding of the numbers.

With this we come to the end of lecture number 38. In the next lecture we shall be starting discussion on how we can carry out arithmetic using floating point numbers. In this lecture you have looked at just the representation, later on we would be seeing how we can carry out addition, subtraction, multiplication and division on floating point numbers.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 39
Floating - Point Arithmetic

In this lecture we shall be starting or discussion on Floating - Point Arithmetic. We have seen earlier how a floating point number can be represented. Now we shall see how with this representation, we can carry out addition, subtraction, multiplication, division and for doing that what kind of hardware is required.

(Refer Slide Time: 00:44)

Floating Point Addition/Subtraction

- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$, where $E1 > E2$ (say).
- Basic steps:
 - Select the number with the smaller exponent (i.e. $E2$) and shift its mantissa right by $(E1-E2)$ positions.
 - Set the exponent of the result equal to the larger exponent (i.e. $E1$).
 - Carry out $M1 \pm M2$, and determine the sign of the result.
 - Normalize the resulting value, if necessary.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

We start with floating point addition and subtraction. We consider them together because the process is very similar, and the same hardware can be used for performing both addition and subtraction. Let us assume that we have two numbers; for the time being we are not showing this sign, sign is also there.

The first numbers is $M1 \times 2$ to the power $E1$, and the other number is $M2 \times 2$ the power $E2$, and we assume that the exponent value of the first number is larger than that of the second, $E1 > E2$. For addition or subtraction the basic steps will be as follows.

Here I have assumed that the second number has the smaller exponent. Select the number of the smaller exponent, here it is $E2$ and shift its mantissa right by $E1 - E2$ positions,

such that this E2 becomes equal to E1. That means, we are aligning the mantissa such that the exponents are becoming equal. Suppose this was 2 to the power 10 and this was 2 to the power 5; you shift M2 by 5 positions such that this also becomes 2 to the power 10.

So, both the exponents will become equal. So, after this we will be adding or subtracting M1 and M2 straight away, because M2 is already shifted right. We have already aligned the mantissa.

After shifting we simply carry out addition or subtraction depending on which operation we are trying to carry out. Also we can see the result of addition and subtraction, and you can calculate the sign of the result and after addition or subtraction we may need to normalize the result because the result may not start with a 1. For normalization we have to shift it left, so that the first bit of the mantissa always becomes 1, so that we can represent the mantissa in a suitable way excluding that implied 1.

(Refer Slide Time: 03:14)

Addition Example

- Suppose we want to add $F_1 = 270.75$ and $F_2 = 2.375$

$$F_1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$$

$$F_2 = (2.375)_{10} = (10.011)_2 = 1.0011 \times 2^1$$
- Shift the mantissa of F2 right by $8 - 1 = 7$ positions, and add:

1000 0111 0110 0000 0000 0000	1 0011 0000 0000 0000 0000	<hr/>
$\begin{array}{r} 1000 1000 1001 0000 0000 0000 \\ \hline 1000 1000 1001 0000 0000 0000 \end{array}$		

Residue

- Result: 1.00010001001×2^8

Logos at the bottom:
 IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, ROURKELA
  

Let us take an example. Suppose we are adding two numbers F1 and F2, values are 270.75 and 2.375. 270.75 if you express in binary it becomes this and in a normalized binary floating point representation, you can write it as this. So, it becomes this multiplied the 2 to the power 8. Similarly F2 can be represented in binary as this. So, if you shift this point by one position it becomes this. So, it becomes 2 the power 1.

Now for the two numbers, one of them is 2 to the power 8, other is 2 the power 1; second one is smaller. So, what I do we shift the mantissa of the number with the smaller exponent; that means, F2 by $8 - 1 = 7$ positions. So, it becomes like this. Then we add the two manissas.

So, you take the first 24 bits of the sum, this will be your actual result, and these remaining bits will be the residue. So, your actual result will be this where it this number is already normalized it starts with 1.

(Refer Slide Time: 05:32)

Subtraction Example

- Suppose we want to subtract $F_2 = 224$ from $F_1 = 270.75$

$$F_1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$$

$$F_2 = (224)_{10} = (11100000)_2 = 1.111 \times 2^7$$
- Shift the mantissa of F_2 right by $8 - 7 = 1$ position, and subtract:

1000 0111 0110 0000 0000 0000
<u>111 0000 0000 0000 0000 0000</u>
0001 0111 0110 0000 0000 0000 000

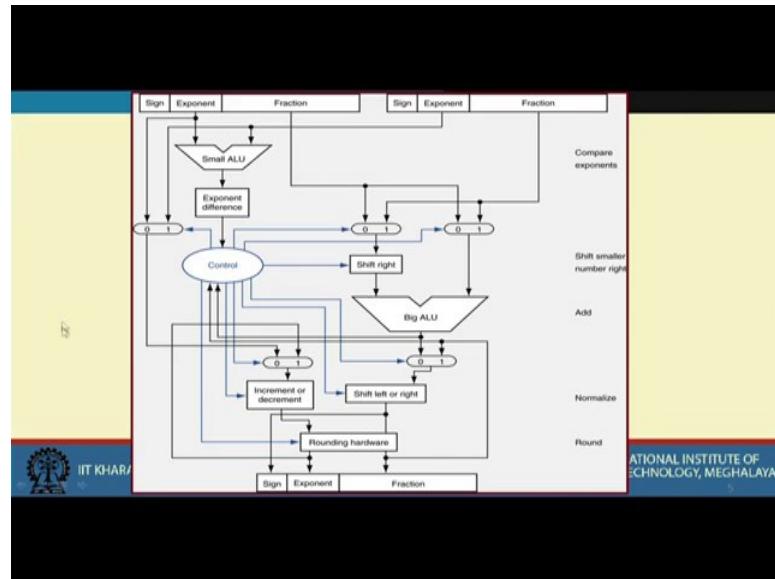
- For normalization, shift mantissa left 3 positions, and decrement E by 3.
- Result: 1.01110110×2^5

 IIT KHARAGPUR |
  NPTEL ONLINE CERTIFICATION COURSES |
  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Similarly, let us take a number 224 and 270.75. We can represent the numbers as follows. Similarly we will have to shift the mantissa of F_2 by one position. Then you subtract the mantissas. You need a normalization step here and the result will be this. This adjustment is required at the end.

This is the process of addition or subtraction of floating point numbers; what are the steps that are involved. From this you can directly generate the circuit.

(Refer Slide Time: 07:16)



You look at this schematic. These are the two numbers, this is the first number, this is second number; sign exponent fraction and sign exponent fraction. This is a small 8 bit ALU; this is actually subtracting the two exponents and finding out which one is smaller; also it is calculating the exponent difference because later on we will have to shift the mantissa by that amount.

So, there is a control circuit here. This information is going to the control depending on which exponent is smaller. There is another ALU where this smaller of the exponents will be coming to the first input; there is a multiplexer here. Either this fraction or this fraction will be selected; control will be generating the signal.

That will be shifted right by so many positions (exponent difference). After shifting you do the actual addition or subtraction; these two multiplexers are selected by the control unit. After you do the addition or subtraction again similarly you will have to look at the process of normalization. Like in the previous example you have seen that the result was not normalized; you had to shift it left by 3 positions. You do the same thing, you may have to shift it and you see by how many positions you have to shift.

Accordingly you make the adjustment and for IEEE format you need an additional step for rounding. So, if you have enabled rounding, after everything is done the mantissa will be rounded depending on those two bits r and s and you get the final result.

The concept of floating point addition subtraction is very simple because already you have seen the steps involved. You have to compare the exponents, you have to align the mantissa by shifting one of them, then you have to do addition or subtraction, then we have to carry out and normalization of the result, and finally, if required, rounding.

(Refer Slide Time: 09:50)

Floating-Point Multiplication

- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$
- Basic steps:
 - Add the exponents $E1$ and $E2$ and subtract the *BIAS*.
 - Multiply $M1$ and $M2$ and determine the sign of the result.
 - Normalize the resulting value, if necessary.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Next let us come to multiplication; floating point multiplication and division are relatively easier in concept, because it involves less number of steps. Suppose I am trying to multiply two numbers like this $M1 \times 2$ to the power $E1$, $M2 \times 2$ to the power $E2$.

(Refer Slide Time: 10:17)

The image shows a handwritten calculation for floating-point multiplication. On the left, two numbers are given in scientific notation: 1.2×10^5 and 1.2×10^6 . These are multiplied to yield 1.44×10^{11} , indicated by a diagonal line through the intermediate result. On the right, the biased exponents E1 (+127) and E2 (+127) are added together. The result is -127 , which is then subtracted from the sum of the biased exponents to give the final biased exponent E.

Suppose I have a number 1.2×10 to the power 5, and there is another number 1.2×10 to the power 6.

When you multiply the two numbers, there is no question of doing any adjustment of mantissa; you simply multiply the two mantissa. So, 1.2×1.2 will become 1.44. And you simply add the two exponents; it will be 11, and we get the result. But in floating point number one thing you have to remember that is E1 and E2 are both biased; that means, for single precision number already we have done +127. So, for multiplication if you are adding these two numbers whatever you get, do not forget to subtract a 127 because you have counted 127 twice.

So, you have to subtract it once. What I have mentioned is written here. You add the two exponents and subtract the bias that is 127; multiply M1 and M2 and also determine the sign of the result, and only at the end you need to normalize result if it is required.

So, it is much simpler than addition.

(Refer Slide Time: 12:00)

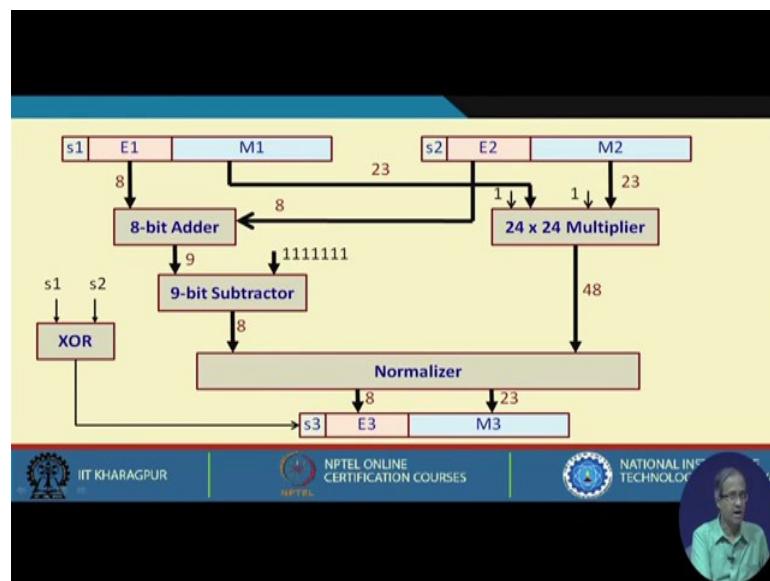
Multiplication Example

- Suppose we want to multiply $F_1 = 270.75$ and $F_2 = -2.375$
 $F_1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$
 $F_2 = (-2.375)_{10} = (-10.011)_2 = -1.0011 \times 2^1$
- Add the exponents: $8 + 1 = 9$
- Multiply the mantissas: 1.01000001100001
- Result: $1.01000001100001 \times 2^9$



So, let us take an example. Suppose we want to multiply $F_1 = 270.75$ and $F_2 = -2.375$; both are normalized. The steps of multiplication are shown.

(Refer Slide Time: 13:08)



The multiplier hardware will be conceptually simpler. You have the first number here, and the second number here. There is an adder which will be adding E_1 and E_2 ; you will have to add the two exponents and these are 8 bit numbers, temporarily you generate an 9-bit sum because you will have to subtract the bias 127.

After subtracting the bias you get the final 8-bit result which is the final exponent. On the other side you have a 24×24 multiplier, because the mantissas are all 23 bit numbers, but there is an implied 1 bit which is hidden. So, the multiplier will also have to be fed with that extra 1 bit, and then the two numbers multiplied.

So, we get 48 bit product; you look at the product whether it starts with 1 or not; if not you will have to do a normalization, shift it and also adjust the exponent accordingly. After that the final exponent will go into the result. The final mantissa 23 bits will go to here, and the sign will be the exclusive or of s1 and s2.

So, if one of the numbers is positive and other is negative then only the result will be negative; otherwise the result will be positive. So, this is the multiplication hardware; floating point division is quite similar.

(Refer Slide Time: 15:02)

The slide has a yellow background and a black header. The title 'Floating-Point Division' is centered in bold black font. Below the title is a bulleted list of steps:

- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$
- Basic steps:
 - Subtract the exponents $E1$ and $E2$ and add the *BIAS*.
 - Divide $M1$ by $M2$ and determine the sign of the result.
 - Normalize the resulting value, if necessary.

At the bottom, there are logos for IIT Kharagpur, NPTEL, and National Institute of Technology, Raipur, along with a small video player showing a person speaking.

For division let us say we have two numbers $M1 \times 2^{E1}$, and $M2 \times 2^{E2}$. For division, instead of adding the exponents we are subtracting the exponents, but you see in both the exponents we have added a bias. So, when you subtract them, the bias also gets canceled out. So, you will have to additionally add that bias after the subtraction, then you divide $M1$ and $M2$, and finally, normalize.

(Refer Slide Time: 15:40)

Division Example

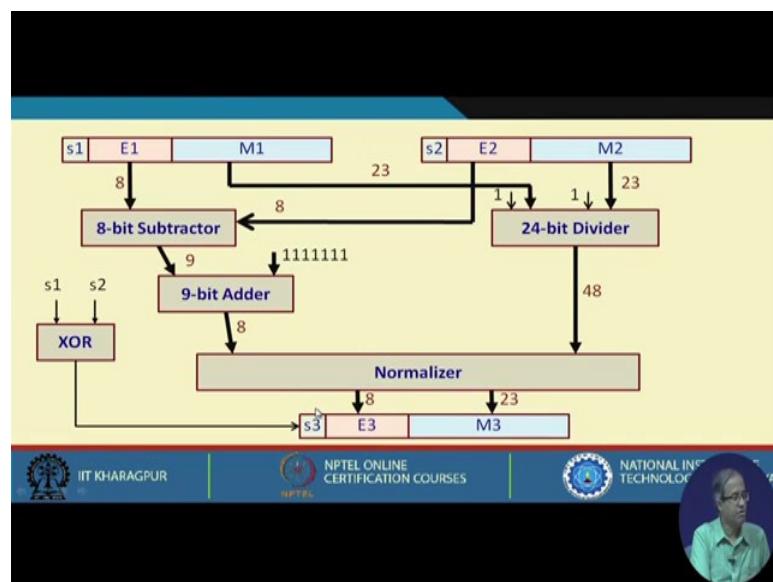
- Suppose we want to divide $F_1 = 270.75$ by $F_2 = -2.375$
 $F_1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$
 $F_2 = (-2.375)_{10} = (-10.011)_2 = -1.0011 \times 2^1$
- Subtract the exponents: $8 - 1 = 7$
- Divide the mantissas: 0.1110010
- Result: 0.1110010×2^7
- After normalization: 1.110010×2^6



So, let us take an example; suppose I want to divide this number by this number.

The steps of division are all shown.

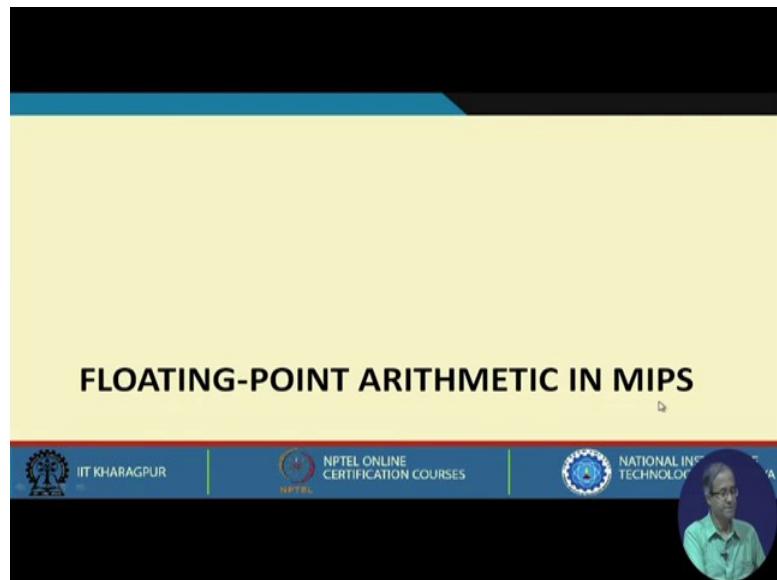
(Refer Slide Time: 16:41)



The hardware will be very similar to multiplication. The only difference is that here instead of addition you do a subtraction; you subtract the two exponents, you get a 9 bit result, and you add the bias to get the final value of the exponent.

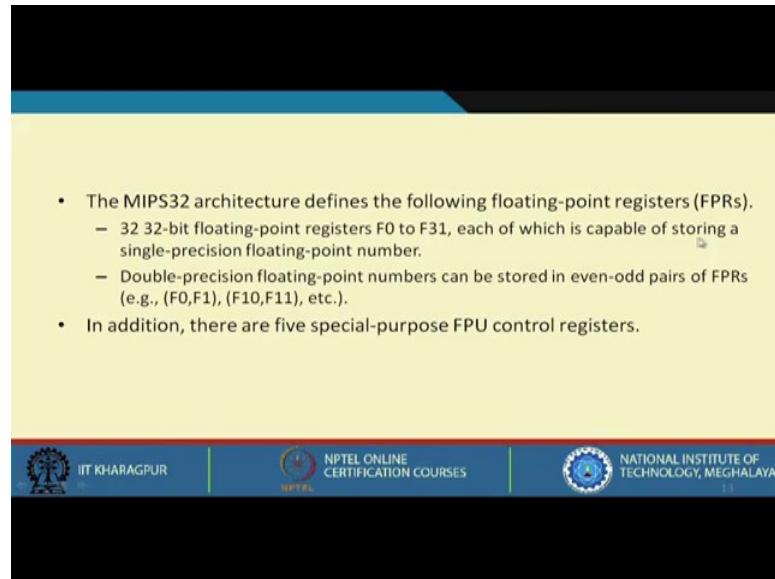
Similarly, you have a 24-bit divider. These 23 bit mantissas are coming and the implied 1 bit are here. The divider will be generating the bits of the result. Again you may have to normalize as the previous example showed, and after normalization you take the corrected value of exponent. The first 23 bits of the result will go to mantissa, and just like multiplication for division also you take the XOR of the s1 and s2 that will give you this sign of the result.

(Refer Slide Time: 17:42)



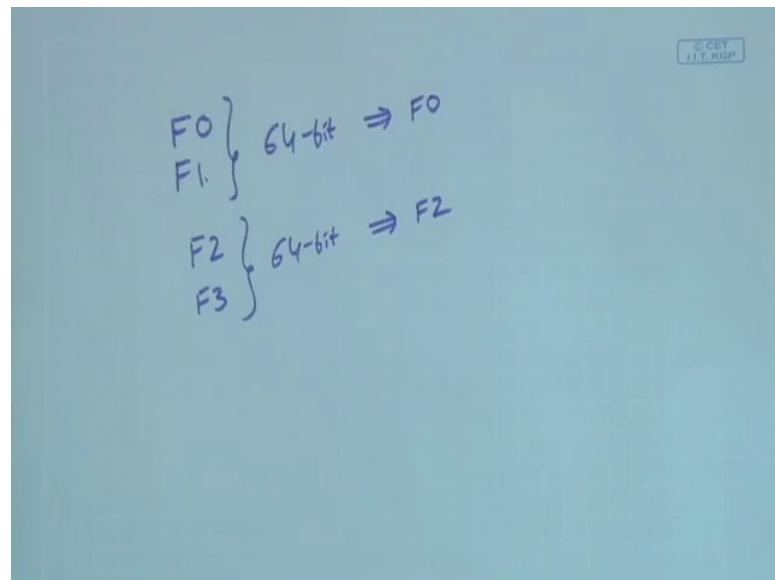
Now, for MIPS32 instructions set architecture there are also floating point arithmetic, which we have not talked about earlier. Very briefly let us look at what are the features that are available in MIPS. The first thing is that in the MIPS architecture we have some floating point registers called F0 up to F31; just like for integer registers we had R0 to R31.

(Refer Slide Time: 18:25)



These registers are all capable of storing a single precision number because they are 32-bit numbers. Now additionally you can also operate on double precision numbers which are 64 bits in size, but how you do it? You will be actually using register pairs (F0, F1), (F2, F3), etc.

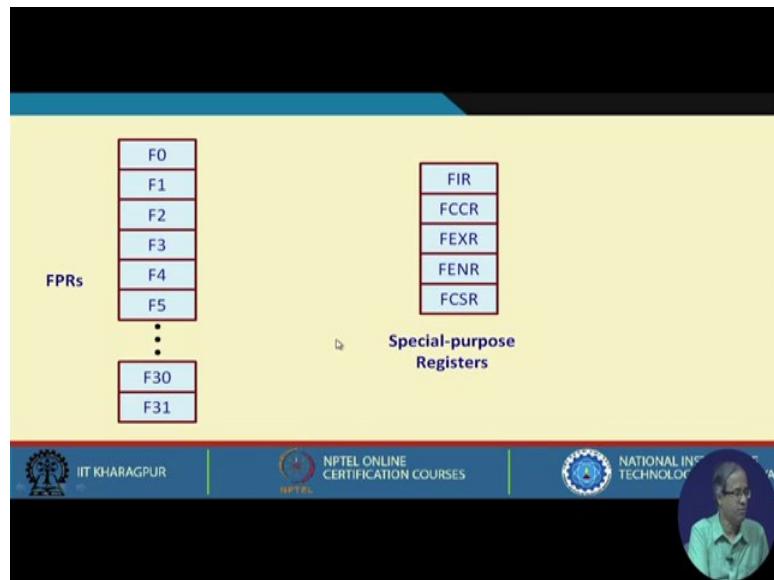
(Refer Slide Time: 18:49)



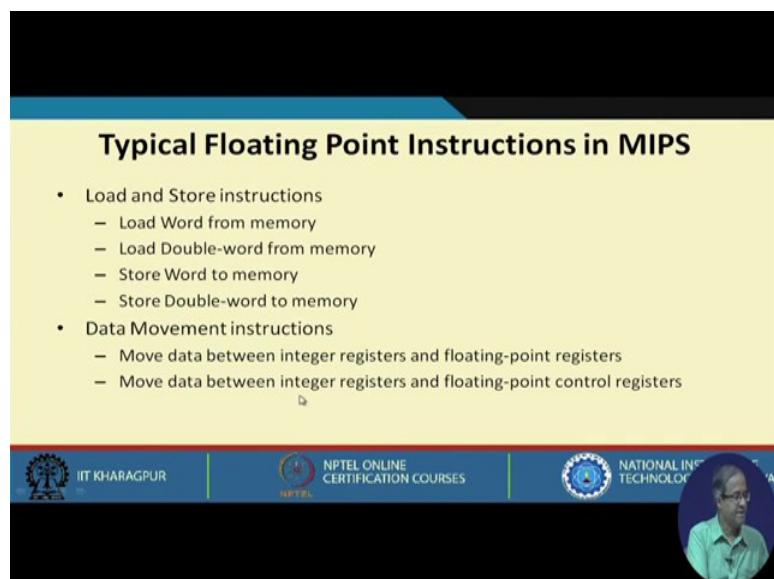
If you are using F0 and F1, together this will be a 64-bit register, if you look at F2 and F3 together this will also be a 64-bit register. But when you use it in a program you just refer to as F0 double precision, F2 double precision, etc.

So, when you are into double precision F0 means (F0, F1), F2 means (F2, F3). It is implied that even and odd register pairs will be taken. In addition to this there are some special purpose registers also, but here we are not going into details of them because these are not really required.

(Refer Slide Time: 19:43)



(Refer Slide Time: 19:54)



And the instructions that relate to the floating point registers are as follows. There are Load and Store instructions; we can load a single precision floating point number from

memory or we can also do a load double. If you have a load double let us say to F0, it will be loaded into F0 and F1.

Similarly, there is a store word, store double word data movement instruction. You can move data from an integer register to a floating point register or vice versa, or into some control registers. Also the arithmetic instructions add, subtract, multiply, divide.

(Refer Slide Time: 20:34)

The slide has a black header and footer. The main content area is yellow. It lists various arithmetic instructions:

- Arithmetic instructions
 - Floating-point absolute value
 - Floating-point compare
 - Floating-point negate
 - Floating-point add
 - Floating-point subtract
 - Floating-point multiply
 - Floating-point divide
 - Floating-point square root
 - Floating-point multiply add
 - Floating-point multiply subtract

The footer contains logos for IIT Kharagpur, NPTEL, and National Institute of Technology, along with a portrait of a man.

There are some other instructions also. You can calculate the absolute value of a number, you can compare two numbers, you can negate; that means, given x you find out $-x$, square root of a number. Multiply add, multiply subtract means you multiply two numbers and add a third number $a \times b + c$. This multiply accumulate kind of instructions or multiply subtract are quite useful in various DSP applications, where this kind of computations appear quite frequently.

(Refer Slide Time: 21:27)

- Rounding instructions:
 - Floating-point truncate
 - Floating-point ceiling
 - Floating-point floor
 - Floating-point round
- Format conversions:
 - Single-precision to double-precision
 - Double-precision to single-precision

There are some instructions to round the numbers. Here you can specify what kind of rounding you are doing: truncation, ceiling, floor, or just round. I mentioned these 4 types are there, and also you can sometimes convert from single to double precision or vice versa; those instructions are also there.

(Refer Slide Time: 21:51)

Example: Add a scalar s to a vector A

```
for (i=1000; i>0; i--)  
    A[i] = A[i] + s;
```

Loop:

```
L.D      F0, 0(R1)  
ADD.D   F4, F0, F2  
S.D      F4, 0(R1)  
ADDI   R1, R1, -8  
BNE    R1, R2, Loop
```

R1: initially points to A[1000]
(F2,F3): contains the scalar s
R2: initialized such that 8(R2) is the address of A[1]
We assume double precision (64 bits):

- Numbers stored in (F0,F1), (F2,F3), and (F4,F5).

I just take a simple example of adding a scalar to a vector. Like in C let us say I have a loop like this. The MIPS32 assembly language code is also shown.

This is a very simple example that shows how we can use the floating point registers and instructions. It is quite simple, just an extension of the integer instructions.

So, with this we come to the end of this lecture. If you recall in this lecture we had looked at how we can carry out floating point operations, in particular the addition, subtraction, multiplication and division, and also we had discussed the corresponding hardware requirements.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 40
Basic Pipelining Concepts

In this lecture will shall be talking about some Basic Pipelining Concepts. Now you may wonder why we are discussing pipeline in the context of floating point arithmetic and floating point; I mean hardware implementation of arithmetic functions.

I would like to tell you here is that, pipelining is a very commonly used and widely used technique to enhance the performance of a system without significantly investing. In hardware if I want to make something faster I can always replicate some functional units. So, instead of one adder I can use 5 adders; my addition time will speed up 5 times. But here the philosophy is different; we are not replicating hardware, we are using a different kind of a philosophy by which we can promote something called overlapped execution whereby the performance improvement can be quite close to what we can achieve by actually replicating the hardware.

We are getting good return of investment without making any significant enhancement in the hardware. This is the basic concept of pipelining. The reason we are discussing pipelining here is that, we shall see how we can use pipelining to improve the performance of arithmetic circuits that you have already seen.

Later on we shall also see how we can enhance the performance of the processor, that is called instruction level pipelining; how instructions in the MIPS32 data path can be executed faster by implementing a pipeline there. But before that you will have to understand what is a pipeline, how it benefits the designer, and what kind of speed up you are expected to get. So, here in this lecture we shall be talking about the basic pipelining concepts.

(Refer Slide Time: 02:57)

What is Pipelining?

- A mechanism for overlapped execution of several input sets by partitioning some computation into a set of k sub-computations (or stages).
 - Very nominal increase in the cost of implementation.
 - Very significant speedup (ideally, k).
- Where are pipelining used in a computer system?
 - Instruction execution: Several instructions executed in some sequence.
 - Arithmetic computation: Same operation carried out on several data sets.
 - Memory access: Several memory accesses to consecutive locations are made.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

As I had said pipelining is nothing but a mechanism for overlapped execution of several different computations. Several different computation means they are possibly carrying out some computation on several input data sets. The basic principle is that we try to divide our overall computation into a set of k sub-computations, called stages.

As I had said if we do this there is a very nominal increase in the cost of implementation. Again I shall be illustrating it with a real life example. But we can achieve very significant speedup that can be very close to k . This is really fantastic; we are not really investing in additional hardware, just we are dividing the hardware into smaller functional sub-blocks, and by doing that we get an architecture where we can equivalently get a speedup of up to the number of sub-functional units that you have divided our overall computation into.

This kind of pipelining can be used in several different areas of computer design. For instruction execution we shall see later that several instructions can be executed in some sequence, because you have already seen earlier that when instruction is executed in a processor basically there is a fetch and execute cycle. During fetch we fetch an instruction, then we decode it; depending on the type of instruction we can do the appropriate actions or operations to complete the execution. Now the idea is that when an instruction is being decoded and it is being executed, why not fetch the next instruction

and try to decode it at the same time; this is what I mean by overlapped execution. This is what is there for instruction pipelining.

Similarly, for arithmetic computation we shall be seeing how we can speed up arithmetic computations by implementing pipeline. Similarly for memory access also we shall see later again that we can use the concepts of pipelining to speedup accesses from consecutive locations in memory.

(Refer Slide Time: 06:16)

A Real-life Example

- Suppose you have built a machine M that can wash (W), dry (D), and iron (R) clothes, one cloth at a time.
 - Total time required is T .
- As an alternative, we split the machine into three smaller machines M_W , M_D and M_R , which can perform the specific task only.
 - Time required by each of the smaller machines is $T/3$ (say).

$\xrightarrow{\quad W + D + R \quad}$
 $\xleftarrow{\qquad T \qquad}$
 For N clothes, time $T_1 = N \cdot T$

$\xrightarrow{\quad W \quad} \cdots \xrightarrow{\quad D \quad} \cdots \xrightarrow{\quad R \quad}$
 $\xleftarrow{\qquad T/3 \qquad} \xleftarrow{\qquad T/3 \qquad} \xleftarrow{\qquad T/3 \qquad}$
 For N clothes, time $T_3 = (2 + N) \cdot T/3$

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY WARANGAL | 

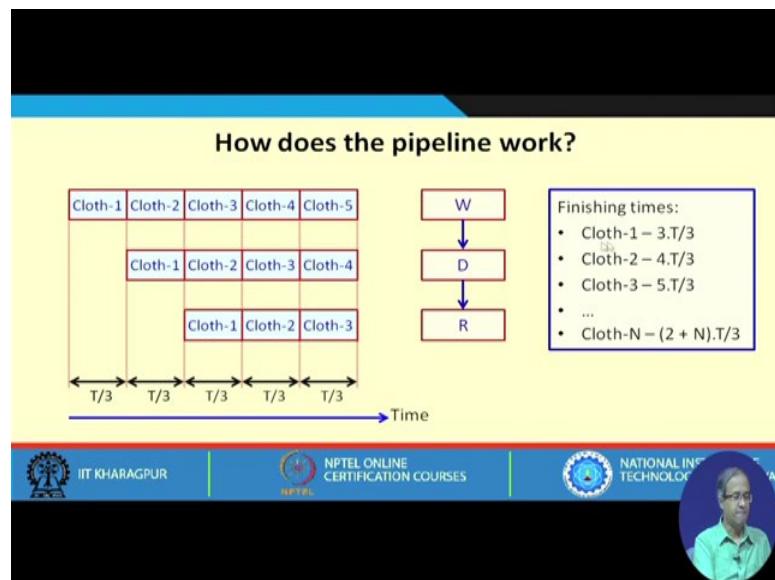
Let us take a real life example say of washing. Suppose we have a requirement where we need to wash clothes, dry them and iron them. Suppose I have a machine, which can do washing, which can do drying, which can do ironing. So, there are 3 stages of the machine. I feed my clothes to the machine. The clothes will be moving through the 3 stages and then I can feed the next cloth or next set of clothes whatever.

Let us assume that this whole thing takes a time T . So assuming that I can feed one cloth at a time, if there are N clothes I need to do washing, drying and ironing, the total time will be $N \times T$. Now as an alternative what we are saying that we are not buying 3 such machines, but rather we are breaking the machine into 3 smaller parts. There will be one machine which can do only washing, there will be one machine which can do only drying, and one machine which can do only ironing. So, roughly the total cost remains approximately equal to the total cost of the original machine. Assuming that they take

equal time, let us assume that earlier it was taking a time of T , now each of these will be taking a time of $T/3$.

We shall see very shortly that if I do this then for washing, drying and ironing N clothes, I need a time $(2 + N) \cdot T/3$. So, if N is large you can ignore 2; it is approximately $N \cdot T/3$, which means I have got a speed up of approximately 3 (equal to the number of pieces I have broken my original machine into). This is the essential idea behind pipelining; I can get a significant speedup.

(Refer Slide Time: 09:01)



Let us see, washing - drying - ironing; after washing is completed I will be giving it to the dryer, after drying is completed I shall giving it to ironing.

To start with, during the first time slot let us say the time slots are all $T/3$, $T/3$ each, the first cloth reaches the washer. It finishes washing at the end of this time slot. Then this cloth-1 will be moving to D. What will happen in the next time slot? This cloth-1 will be moving to D, but now the washer will be idle. So, I can feed this second cloth to the washer.

Now, the washer and dryer are working on two different clothes in an overlapped way. So, at the end of this time slot, dryer has finished with cloth-1 and washer has finished with cloth-2. So, what will happen next? Cloth-1 will come here to the ironer, cloth-2 will come here, and cloth-3 will come to washer.

Now you see you see all these 3 machines are busy. Now at the end of this time what will happen? Cloth-1 is done and will be taken out; at the next time cloth-2 will come here, cloth-3 here, and next cloth will come here.

You see after this initial two periods of time that is required for this pipeline to be filled up, after that in every time slot I am getting one output; that means, one cloth is finishing and I can take them out. That is why I talked about the total time as $(2 + N) \cdot T/3$.

(Refer Slide Time: 11:36)

Extending the Concept to Processor Pipeline

- The same concept can be extended to hardware pipelines.
- Suppose we want to attain k times speedup for some computation.
 - Alternative 1: Replicate the hardware k times → cost also goes up k times.
 - Alternative 2: Split the computation into k stages → very nominal cost increase.
- Need for buffering:
 - In the washing example, we need a tray between machines (W & D, and D & R) to keep the cloth temporarily before it is accepted by the next machine.
 - Similarly in hardware pipeline, we need a *latch* between successive stages to hold the intermediate results temporarily.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

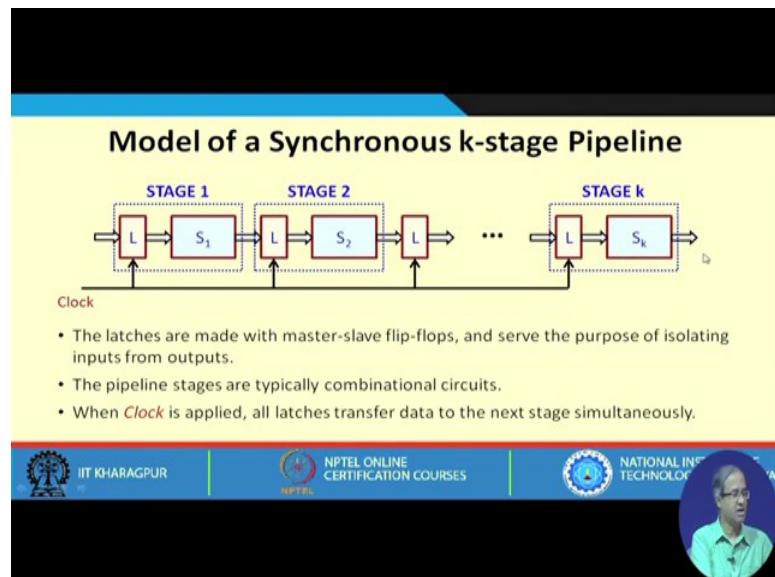
We can extend the concept discussed so far to actual processor designs. Suppose for some computation we want to achieve a speedup of k . The two alternatives are with us. We can use k copies of the hardware, this will obviously give us a speedup of k , but the cost will also go up k times because we are replicating the hardware.

Alternative 2 is pipelining. We are splitting the computation into k number of stages. This will involve very nominal increase in cost, but potentially it can give you a speedup of k . But one thing is required which we ignored; you see if you think of the washing example – washing, drying and ironing, see after one machine finishes and a cloth comes out and goes to the input of the other machine, I need some kind of a tray or a buffer in between. The cloth will be temporarily stored in that buffer before it can be fed to the next machine. Because it may so happen then when the first machine finishes with washing, the dryer is still working on the previous one, and it is still not free.

So, only when it is free then only the cloth can be given to the dryer. Thus, I need some kind of a buffering mechanism in between the stages. This is what we talk about here – the need for buffering. For the washing example is it said we need a tray between the machines to keep the cloth temporarily before the next machine accepts it. In exactly the same way when we want to implement pipeline in hardware, we need something similar to a buffer. It is nothing but a latch or a register between successive stages, because one stage finish some calculation and prior to giving it to the next stage, maybe the next stage is still not finished.

So, that value is temporarily kept in the latch, so that the next stage whenever it is free can take it from the latch. This is the idea behind hardware pipelining.

(Refer Slide Time: 14:41)



I am showing a schematic diagram of a k stage hardware pipeline. This is called a synchronous pipeline because there is a clock which is synchronizing the operation of all these stages. So, as you can see there are k number of stages. Each stage involves some computation, S_1, S_2 to S_k , and also there is a latch. The latch will temporarily store the result of the previous stage before the next stage is ready to accept it. The clock will be generating active signals periodically and clock period will be large enough, so that all these stages can finish their computation and also it should take care of the delay of the latch. This we shall come a little later how the clock frequency or the clock period can be calculated. The stages S_1, S_2, \dots are typically combinational circuits. So, whenever the

next active edge of the clock comes, the next data is fed to S1, the output of S1 goes to S2, S2 goes to S3, and so on. There is a shift that goes on whenever the clock signal appears. In synchronism with the clock the pipeline shifts result from one stage to the other in a lock step fashion.

(Refer Slide Time: 16:34)

Types of Pipelined Processors

- Can be classified based on various parameters:
 - a) Degree of overlap
 - Serial, overlapped or pipelined
 - b) Depth of the pipeline
 - Shallow or Deep
 - c) Structure of the pipeline
 - Linear or Non-linear
 - d) How the operations are scheduled in the pipeline?
 - Static or Dynamic

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

You can classify or categorize pipelined processors based on various parameters like degree of overlap, depth of the pipelining, structure of the pipeline, and how we schedule the operations.

(Refer Slide Time: 16:57)

(a) Degree of Overlap

- Serial
 - The next operation can start only after the previous operation finishes.
- Overlapped
 - There is some overlap between successive operations.
- Pipelined
 - Fine-grain overlap between successive operations.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

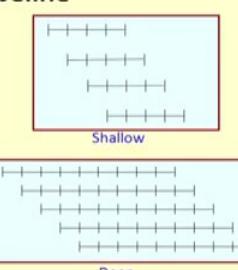
Let us very briefly look at these. When you talk of the degrees of overlap, one extreme can be serial that is a strictly non-pipeline implementation. The next operation can start only after the previous operation finishes. So, here I shown it schematically like this, this is an operation let us say which takes 1 2 3 4 5 steps. So, only after the first operation is completed only then the second operation can start. It is strictly sequential, with no overlap. There can be partial overlap as the second diagram shows. Partial overlap naturally results in a speedup because earlier it was taking so much time, but now it is taking 2 time units less. In the extreme case you can have something called pipeline as I said. There is almost complete overlap; when the first operation has finished with the first step it moves to the second step, and the second operation can start with its first step. This is called fine-grained overlapped execution.

Here naturally the time required is much less. So, depending on the degree of overlap you can classify how deep or how efficient your pipeline implementation is. Ideally you should have something like this.

(Refer Slide Time: 19:12)

(b) Depth of the Pipeline

- Performance of a pipeline depends on the number of stages and how they can be utilized without conflict.
- Shallow pipeline is one with fewer number of stages.
 - Individual stages more complex.
- Deep pipeline is one with larger number of stages.
 - Individual stages simpler.



Shallow

Deep

 IIT KHARAGPUR
 NPTEL ONLINE CERTIFICATION COURSES
 NATIONAL INSTITUTE OF TECHNOLOGY WARANGAL



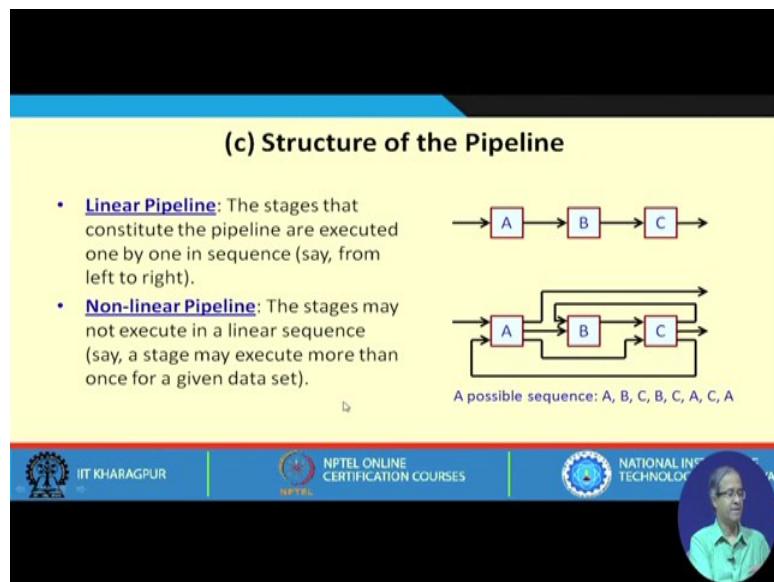
Then comes the depth of the pipeline. We have seen earlier that if there are k number of stages, then we can have a potential speedup of up to k . Then you can argue why do not have a very large value of k so that we can get a very large speedup. But there are some limitations to increase the value of k ; we cannot break up a computation into smaller

pieces beyond a limit. Beyond a limit it does not make sense, maybe the delay of the latch will be more will become more than the delay of the computation.

So, the depth of the pipeline is an issue. You can either have a shallow pipeline with a few number of stages, or you can have a deep pipeline with large number of stages. If you have 9 stages then potentially you can have a speedup of 9. Depth of pipeline is a design issue. But it is also very important to evaluate that you can increase the depth all right, but you must also see in what way you can allow the computations to proceed so that you can have overlapped execution without any conflict. We shall be seeing later that conflicts in a pipeline are very important and there are so many techniques to handle them.

For a shallow pipeline because we are making the number of stages smaller, individual stages are more complex. But if I have a very large number of stages, each stages will become simpler.

(Refer Slide Time: 21:18)



Next comes structure of the pipeline. We can have a linear pipeline that is most conventional. Linear pipeline means I have divided the computation into a number of stages that are supposed to be executed linearly one after the other. That means, there is some kind of a straight-line sequence; for every data input first A, then B, then C, and then you take out the result. This is the order of execution for every input data. But for a non-linear pipeline, see for very complex pipeline implementation you can have this kind

of non-linear pipelining. What non-linear pipeline says is that the stages do not execute in this kind of a straight-line or linear sequence. You see this is a pipeline again with 3 stages A B C, of course I have not shown the latches.

The arrows indicate the connection between stages. Like you can see from A there is a connection to B, from A you can also move to C, and also you can take a result out from B, you can go to C, from C either you can take a result out or you can move it back to B, or we can move it back to A, and new data is coming always to A. Here in this example a possible sequence can be this A B C B C A C A. You can have some other sequence also, say A B C B C.

This is so-called non-linear pipeline where the stages may not execute in a linear sequence. As this example shows a particular stage may execute more than once for a given data set.

(Refer Slide Time: 24:10)

(d) Scheduling Alternatives

- Static Pipeline:
 - Same sequence of pipeline stages are executed for all data / instructions.
 - If one data / instruction stalls, all subsequent ones also gets delayed.
- Dynamic Pipeline:
 - Can be reconfigured to perform variable functions at different times.
 - Allows feedforward and feedback connections between stages.

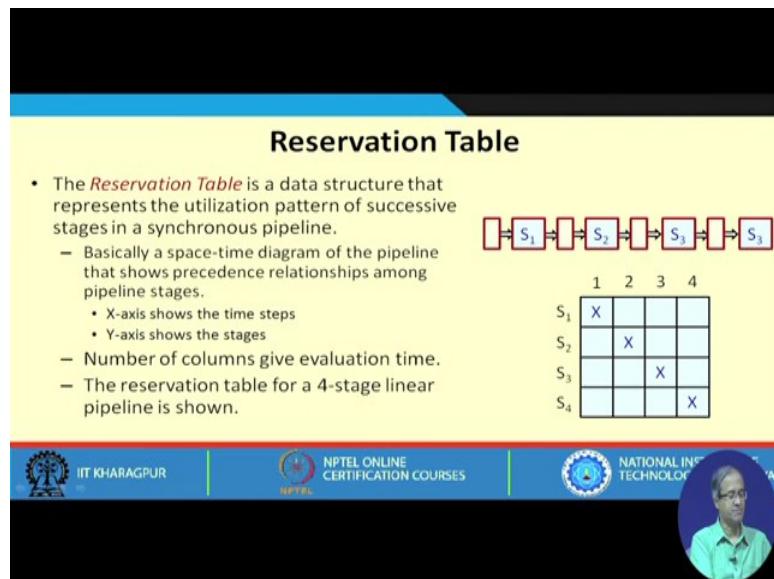
IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY ROURKELA

The last classification depends on how we are scheduling the pipeline; it can be either static or it can be dynamic. Static means you have the pipeline stages and in the previous example you have seen that either I can execute them linearly or I can execute them in some particular order by feedback, etc. Static says same sequence of pipeline stages are executed for all data sets, say you can have non-linear piping all right, but the sequence of stages you are executing will always remain the same, that will not change with time.

So, if one data or instruction stalls, I cannot feed the new data. Stall means due to some ongoing computation, I have to stop the pipeline temporarily. Here it says if I stall a particular data, then all the subsequent data sets also gets stalled.

In contrast we can have a dynamic pipeline where with time the pipeline can be configured, so that different sequence of pipeline stages can get executed. This allows feedforward and feedback connections as the previous example showed. This is a feed forward connection and this is a feedback connection. One example of a dynamic pipeline is one that is able to carry out both floating point addition and multiplication.

(Refer Slide Time: 26:27)

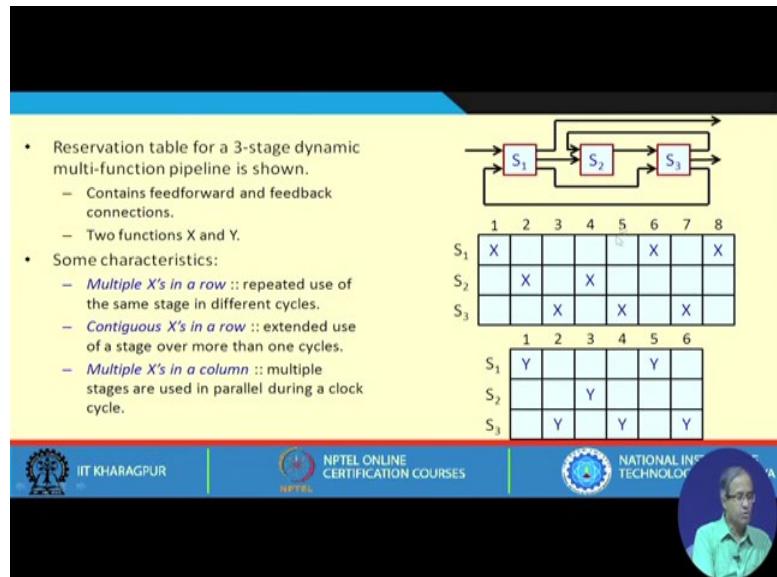


When you are doing addition then certain sequence of stages will be executed, when you are doing multiplication some other sequence of stages will be executed. This is example of dynamic pipeline scheduling. Reservation table is a very commonly used data structure that represents the utilization pattern of successive stages. Let us take a simple example. Suppose I have a linear pipeline S1 S2 S3 S4. When a data comes, in the first cycle it goes to S1, second cycle it goes to S2, third cycle to S3, fourth cycle to S4.

You see the reservation table represents exactly that. On one side I show the stages other side I show the time steps. First time step we use S1, second time step we use S2, third time step S3, fourth time step S4. So, this is basically a space-time diagram that shows precedence relationship among the pipeline stages. The x-axis shows the time steps and

y-axis shows the stage. Number of columns indicates the total time required by the pipeline to evaluate the result. Let us look at a more complex reservation table.

(Refer Slide Time: 27:51)



Here we look at that non-linear pipeline example we showed earlier.

For the same pipeline example I am showing two possible reservation tables. One computation X is here other computation Y is here. So, what it shows? It says that for computation X we need 8 steps: first time step S₁, second time step S₂, third time step S₃, fourth time step again S₂; that means, from S₃ again you go back to S₂, fifth time step again S₃, sixth time step S₁. So, from S₃ you go back to S₁, then again S₃. You follow this path to S₃ then again S₁ again and you are finished. The Y computation is like this: first time step S₁ then S₃. So, you follow this path straight away then S₂, then S₃, then S₁, S₃ again S₁ again, from here you come out.

So, X comes out of here and Y result comes out of here. For the reservation table although all of these are not being shown in these examples, the first one is shown. Multiple cross marks in a row means that for this computation this stage S₁ will be used in time cycles 1, 6 and 8; stage S₂ will be used in time cycles 2 and 4; S₃ will be used in 3, 5 and 7. So, multiple X's in a row means repeated use of the same stage in different time cycles. Sometimes you may need a stage for an elongated period of time.

You can have two consecutive X's side by side, which will mean extended use of a stage over more than one cycle. Sometimes it may be required you may need for a computation stages required for two cycles. So, there will be two X's side by side. It is not shown in this example, you can also have multiple check marks in a column. That means, in a particular time step you can have a check mark here as well as here, which means both S1 and S2 are working together on the same data set.

(Refer Slide Time: 30:58)

Speedup and Efficiency

Some notations:

- τ :: clock period of the pipeline
- t_i :: time delay of the circuitry in stage S_i
- d_L :: delay of a latch

Maximum stage delay	$\tau_m = \max \{t_i\}$
Thus,	$\tau = \tau_m + d_L$
Pipeline frequency	$f = 1 / \tau$

– If one result is expected to come out of the pipeline every clock cycle, f will represent the maximum throughput of the pipeline.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY TRIVANDRUM

So, multiple stages are used in parallel during a clock cycle. In general you can have all these characteristics in a reservation table. Let us make a quick calculation how we can compute the speed up and efficiency of a pipeline. We define some notations; tau is the clock period of the pipeline, t_i denotes the time delay of the stage i circuitry, and d_L denote the delay of a latch. There are k numbers of stages S_1 S_2 to S_k . So, the delays will be t_1 t_2 to t_k . The maximum stage delay I am denoting as τ_m which is max of t_i . The minimum clock period should be satisfying this criteria, it must not be less than $\tau_m + d_L$; that means, the maximum stage delay plus the delay of a latch. The pipeline frequency f would be the reciprocal of τ .

f will represent the maximum throughput of the pipeline. That means, in this rate the output results will be coming out of the pipeline. If you have a linear pipeline you are expecting one result to come out every cycle, but for a non-linear pipeline things will

become more complex as we will see later, you may not be generating a result every clock cycle.

(Refer Slide Time: 32:41)

- The total time to process N data sets is given by
$$T_k = [(k - 1) + N] \cdot \tau \quad (k - 1) \tau \text{ time required to fill the pipeline}$$
$$1 \text{ result every } \tau \text{ time after that} \rightarrow \text{total } N \cdot \tau$$
- For an equivalent non-pipelined processor (i.e. one stage), the total time is
$$T_1 = N \cdot k \cdot \tau \quad (\text{ignoring the latch overheads})$$
- Speedup of the k -stage pipeline over the equivalent non-pipelined processor:
$$S_k = \frac{T_1}{T_k} = \frac{N \cdot k \cdot \tau}{k \cdot \tau + (N - 1) \cdot \tau} = \frac{N \cdot k}{k + (N - 1)}$$
As $N \rightarrow \infty$, $S_k \rightarrow k$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

The total time to process N data sets is given by this expression.

Now if you have a equivalent non pipelined processor where we do not have pipelining, the total time will be $N \times k \times \tau$.

Here of course I made an assumption that all stage delays are equal and the latch overheads are ignored, but still this will help us to make a very fair comparison. Let us see how much improvement we are getting by pipelining. As N becomes very large this speedup tends to k .

(Refer Slide Time: 34:57)

- Pipeline efficiency:
 - How close is the performance to its ideal value?

$$E_k = \frac{S_k}{k} = \frac{N}{k + (N - 1)}$$

- Pipeline throughput:
 - Number of operations completed per unit time.

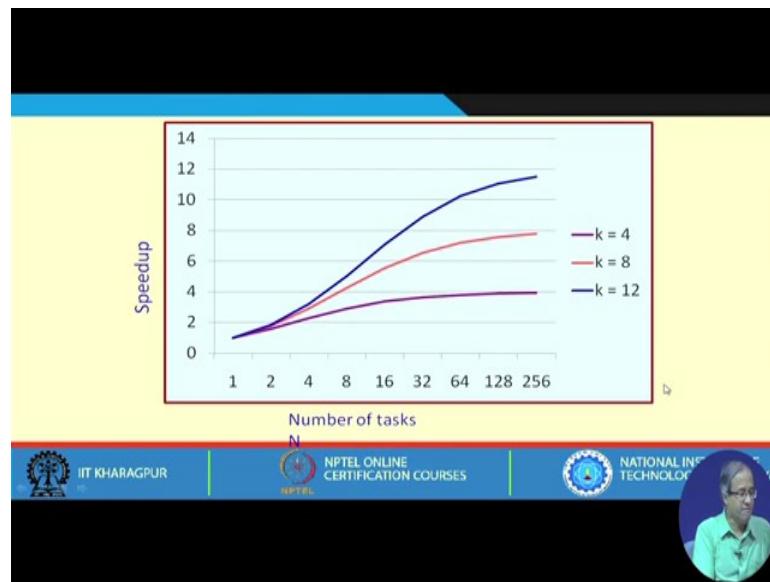
$$H_k = \frac{N}{T_k} = \frac{N}{[k + (N - 1)] \cdot \tau}$$

This simple expression tells you that we can achieve a pipeline speedup of maximum k as n tends to infinity; and we can define a term called pipeline efficiency which is defined as how close the actual pipeline performance is towards ideal value. You see the actual speed up is S_k just as we had expressed, and the ideal speed up is k.

So, I can divide S_k by k. How close this fraction is to 1; if it is 1 which means I have got the ideal value maximum efficiency. But in practice the denominator is greater than the numerator. So, efficiency is a little less than 1.

The pipeline throughout can be defined as number of operations completed per unit time.

(Refer Slide Time: 35:59)



This is a graph I am showing. As you vary the number of tasks n the speedup also increases. But for a 4-stage pipelining it levels up to 4; it cannot cross 4. For 8 stages it levels to 8, for 12 stages it approaches 12. So, it can never cross k.

(Refer Slide Time: 36:42)

Clock Skew / Jitter / Setup time

- The minimum clock period of the pipeline must satisfy the inequality:
$$\tau \geq t_{\text{skew+jitter}} + t_{\text{logic+setup}}$$
- Definitions:
 - Skew*: Maximum delay difference between the arrival of clock signals at the stage latches.
 - Jitter*: Maximum delay difference between the arrival of clock signal at the same latch.
 - Logic delay*: Maximum delay of the slowest stage in the pipeline.
 - Setup time*: Minimum time a signal needs to be stable at the input of a latch before it can be captured.

Lastly I talk about the clock period. Actually the clock period depends on few other things. The minimum clock period must satisfy an expression like this.

When the clock signal is generated it goes to all points in the pipeline latches. Because of unequal distance of the wires the clocks may not reach all the latches at the same time,

and there will be small time difference. This is called clock skew. Clock jitter means because of noise is on the neighboring lines due to capacitive affects, the delay of a signal line can vary slightly from one time to other; maybe for the same latch the period of the first one was τ for the next one it is a little less than τ , for the next on it will be little greater than τ , there can be small variations this is called jitter.

So, skew is the maximum delay difference between the arrival of clock signal at the stage latches, and jitter denotes maximum delay difference between the arrival of clock signal at the same latch. Designers always try to lay out the clock signal in such a way that skew and jitter is minimized, but still there will be a worst case value.

This is the more accurate expression that gives you some lower bound of the clock period that can be used.

With this we come to the end of this lecture. If you recall what we saw in this lecture, we have basically tried to convince you that pipelining is a concept where you can have significant speedup without making a significant investment. In our next lectures we will see that there are some complexities in the pipeline; for example, for non-linear pipeline you may be using the same stage more than once for the same data. So, you are not allowed to feed the input data at every clock; if you do it, there can be some conflicts or clashes in some stages.

Something called pipeline scheduling to decide when I need to feed my next data is very important. We shall be looking into pipeline scheduling aspects and then we shall be seeing how some of the arithmetic operations in particular the floating point operations can be implemented in a pipeline.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

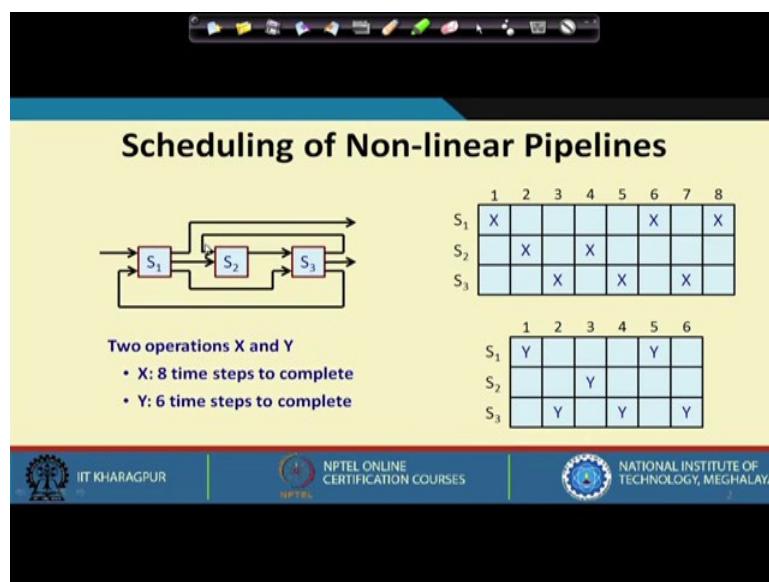
Lecture - 41
Pipeline Scheduling

We have seen so far what is a pipeline, how a pipeline works. In this lecture we shall be talking about Pipeline Scheduling, which means how we can feed the inputs to a pipeline such that no conflict or so called collision occurs in any one of these stages.

Now for a linear pipeline where the stages are connected as a linear chain this problem does not arise because we can potentially apply one new set of inputs every clock cycle. But if there are feed forward and feedback connections in a pipeline in general for a non-linear case, then the calculation of the schedule is not so easy or trivial.

So, here we shall be basically talking about Pipeline Scheduling.

(Refer Slide Time: 01:16)



We take the same example of a non-linear pipeline that we saw earlier. This is a simple enough example that you can illustrate. There are three stages: S₁ S₂ S₃. Here we are not showing the inter-stage latches that are also there, but this is only for illustration. This arrow shows how the data flows from one stage to the other. And here we are assuming that not only the pipeline is non-linear in the sense that there can be data flow from say

S3 to S1, S1 to S3 and so on, but it is a multifunction pipeline; which means, we have the same pipeline which can either compute a function X or compute a function Y.

You can take an analogy for an arithmetic pipeline where you can have a general pipeline where both multiplication and addition can be carried out. But of course, all of the stages will not be used for the two cases. We can selectively use some cases, some of the stages may be using for more than one clock cycles and so on.

So, the operation X that is represented by this reservation table, requires 8 cycles to complete and this is the sequence of stage occupancy. First the input data goes to S1, then to S2, then to S3, then again to S2; you see S3 to S2 there is a connection, from S2 again to S3, S3 to S1; you see S3 to S1 also there is a connection. Then S1 to S3 there is a feed forward connection like this and finally S3 again to S1 then it finishes. So, from S1 it finishes or it goes out.

Similarly for the function Y it again starts with S1 it goes to S3, S2, 3, S1, and finally S3 where it finishes; so from S3 it exits. So, X will exit from here, Y will exit from here. With the help of this example let us explain how scheduling of this pipeline can take place. This means, say the first input which during computation maybe in S2, the second input is also trying to go into S2 at the same time. So, it cannot happen that we know, so one of them has to wait.

(Refer Slide Time: 03:59)

The slide has a yellow header bar with the title 'Latency Analysis'. Below the title is a bulleted list of points:

- The number of time units between two initiations of a pipeline is called the *latency* between them.
- Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a *collision*.
- The latencies that can cause collision are called *forbidden latencies*.
 - Distance between two X's in the same row of the reservation table.

Below the list are two reservation tables:

	1	2	3	4	5	6	7	8
S ₁	X				X	X		
S ₂		X	X					
S ₃			X	X	X		X	

A red box highlights the entries 'X' at positions (S₁, 4) and (S₂, 3), with the text 'Forbidden latencies: 2, 4, 5, 7' written below the box.

	1	2	3	4	5	6	
S ₁	Y						
S ₂				Y	Y	Y	
S ₃			Y	Y	Y	Y	

A red box highlights the entries 'Y' at positions (S₂, 4) and (S₃, 3), with the text 'Forbidden latencies: 2, 4' written below the box.

At the bottom of the slide, there are three logos: IIT Kharagpur, NPTEL, and National Institute of Technology. There is also a small video player window showing a person speaking.

We shall be doing something called latency analysis. Latency is actually the delay between two successive data inputs or initiations that are fed to the pipeline. Our objective is to calculate the latency value or the sequence of latency values that will not result in collision in any of the stages.

Latency is basically defined as the number of time units or clock cycles between two successive initiations; two inputs applied to the pipeline what is the minimum number of time steps we need to give between them. And as I had said if we are not careful enough then two or more initiations may try to use the same stage at the same time that results in a collision.

Some of the latencies will definitely result in a collision, and such latencies are referred to as forbidden latencies; we cannot use those latencies. How you can estimate the forbidden latencies? You can simply calculate the distance between two check marks in the rows of the reservation table. Let us take the example reservation table for X. You see here there are so many check marks, there are check marks at a gap of 2, there are check marks at a gap of 4, there are check marks at a gap of 5, and gap of 7.

These are all forbidden latencies; distance between any two pairs of check marks. For this reservation table for X, the forbidden latencies are 2, 4, 5, and 7. For the other function Y, here you see 2 and 4, here also it is 4. Here the forbidden latencies are only 2 and 4. This is the first step in our analysis to list or enumerate the forbidden latencies.

(Refer Slide Time: 06:33)

The slide has a dark blue header bar with various icons. Below it is a yellow section containing text and two tables. At the bottom are three logos: IIT Kharagpur, NPTEL, and National Technology.

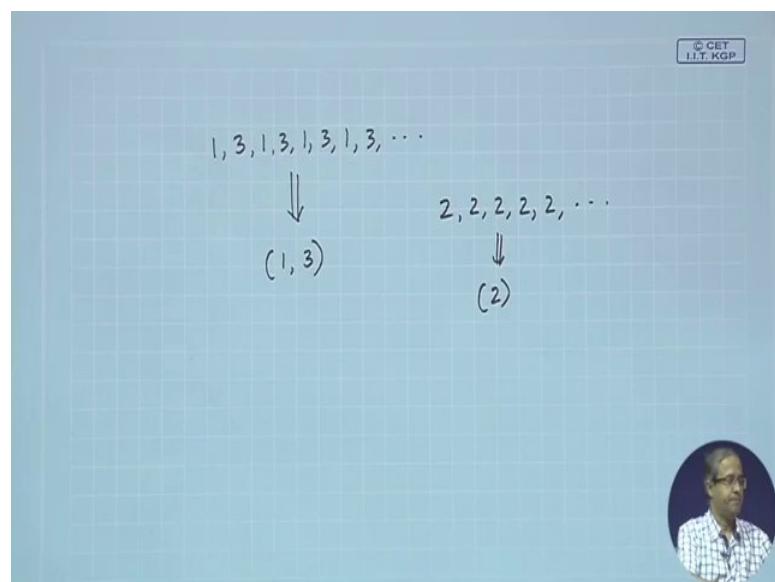
- A *latency sequence* is a sequence of permissible non-forbidden latencies between successive task initiations.
- A *latency cycle* is a latency sequence that repeats the same subsequence.

Function X	Function Y
<ul style="list-style-type: none">• Forbidden latencies: 2, 4, 5, 7• Possible latency cycles: (1, 8) = 1, 8, 1, 8, ... (average latency = 4.5) (3) = 3, 3, 3, ... (average latency = 3.0) (6) = 6, 6, 6, ... (average latency = 6.0)	<ul style="list-style-type: none">• Forbidden latencies: 2, 4• Possible latency cycles: (1, 5) = 1, 5, 1, 5, ... (average latency = 3.0) (3) = 3, 3, 3, ... (average latency = 3.0) (3, 5) = 3, 5, 3, 5, ... (average latency = 4.0)

Now a latency sequence is defined as a sequence of permissible non-forbidden latencies that do not result in a collision. I can say that I am using a latency sequence of 1 2 1, which means between the first and second input I give a gap of 1, between second and third input I give a gap of 2, between third and fourth input I again give a gap of 1.

This sequence can have any arbitrary delay gaps that you can specify. And if this sequence repeats in time then we call it is a latency cycle. Latency cycle is nothing but a latency sequence where the same subsequence is repeated.

(Refer Slide Time: 07:44)



Let us take an example: suppose we use a latency sequence like this: 1, 3, 1, 3, 1, 3, and this repeats indefinitely. We can write in a compact notation that it is a latency cycle (1, 3); this 1, 3 will be repeating indefinitely. Similarly if I have a case for the same latency 2, 2, 2, 2, 2 is to be used, then I can write it as a latency cycle containing a single latency (2).

Talking about the two function: the first function X we have already seen that the forbidden latencies are 2, 4, 5, and 7. Here we are just listing some of the forbidden latency cycles or the some of the latency cycle that you can use, but how to get this we shall be showing shortly. Let us say one of the possible latency cycles that do not result in collisions is (1, 8). So, 1, 8, 1, 8 like that you proceed. This will result in an average latency of 4.5, because (1 + 8) is 9 divided by 2.

If it is a 3, 3, 3, like this here average latency is 3 because 3 is a non forbidden latency and multiples of 3's are also not there. Similarly, (6) is also a feasible latency cycle 6, 6, 6, with average of 6. In this case the minimum latency is resulting for the second case.

Similarly, for the function Y where the forbidden latencies are 2 and 5 we have several alternatives. We can use 1, 5, 1, 5, this kind of a cycle, because 1 is not forbidden, 5 is also not forbidden; this results in an average of 3. We can use 3, 3, 3, this also results in average of 3, also you can use 3, 5, 3, 5 alternating; for this average will be 4.

Now the cycles where there is no change, the same latency have to be used, are referred to as constant cycles. Like here (3), (6) and here (3), these are examples of constant cycles.

(Refer Slide Time: 10:04)

Collision Free Scheduling

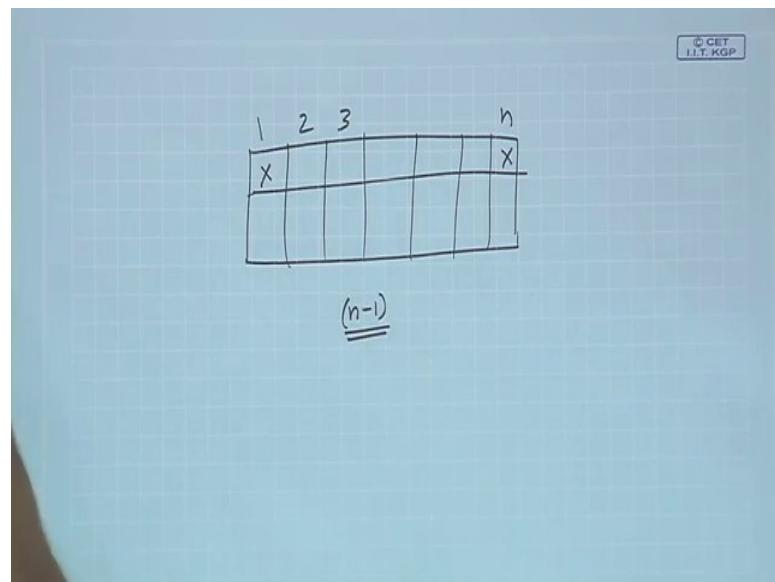
- Main objective:
 - Obtain the shortest average latency between initiations without causing collisions.
- We define a *collision vector*.
 - If the reservation table has n columns, the maximum forbidden latency is $m \leq n-1$.
 - The permissible latencies p will satisfy: $1 \leq p \leq m-1$.
 - The collision vector is an m -bit binary vector $C = (C_m \ C_{m-1} \dots \ C_2 \ C_1)$, where $C_i = 1$ if latency i causes collision, and $C_i = 0$ otherwise.
 - C_m is always 1.

Function X: $C_X = (1011010)$
 Function Y: $C_Y = (1010)$

Now, our objective is to do something called collision free scheduling. How to schedule the pipeline, how to determine the latencies dynamically? Dynamically means when the data comes should be able to automatically find out whether we can feed the next data or not; that is called scheduling. We shall be looking into this.

Scheduling such that there is no collision; well of course, the main objective is to obtain the shortest average latency. Well we start by defining a data structure or vector you can call collision vector. It is a bit vector. Collision vector is defined as follows: if there are n numbers of columns in the reservation table then it is like this.

(Refer Slide Time: 11:04)



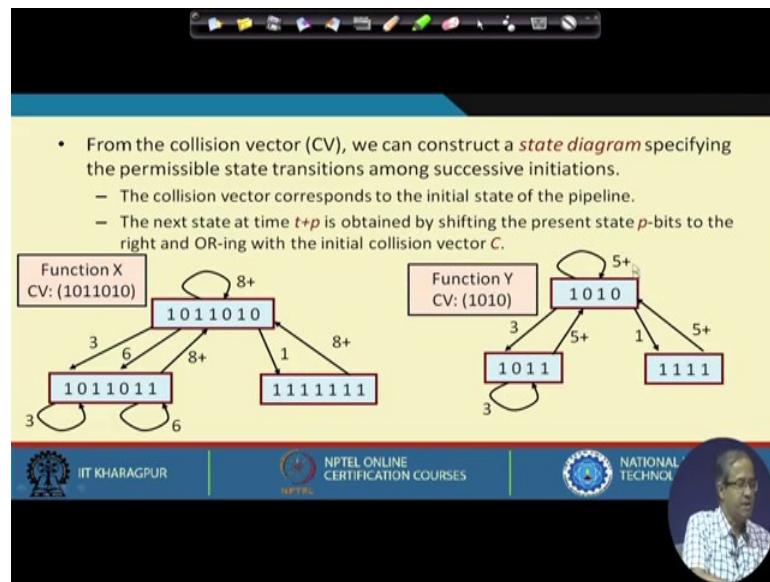
If you have a reservation table there are rows, there are many columns, let us say there are n number of columns. In the worst case there can be a check mark here and check mark here that will result in a forbidden latency of $n - 1$, this can be the maximum forbidden latency.

Here we have mentioned the same thing; the maximum forbidden latency m cannot be greater than $n - 1$. And the permissible latencies that we shall be using for scheduling this obviously has to be less than or equal to $m - 1$, because more than m does not make sense, because effectively if there are 5 stages in the pipeline and I am saying that I have to give a gap of 6 then there is no use in having the pipeline. Anyway, I will be waiting for 6 time cycles to feed the next data. So, that is not a good way of scheduling. We will say that the permissible latencies p will be satisfying this inequality, maximum forbidden latency is $m - 1$, and minimum is of course 1.

The collision vector is an m -bit vector, where m is this maximum forbidden latency, C_1, C_2 to C_m where a particular bit is 1 if latency i causes a collision, and that bit is 0 if the latency i does not cause a collision. And because m is the maximum forbidden latency by definition C_m is always 1, because m is the maximum forbidden latency.

So, for the function C_x , the forbidden latencies were 2, 4, 5, and 7. The collision vector is shown. And function Y from the right side again 2 and 4, these are the forbidden latencies. So, the collision vectors for the two cases are defined like this.

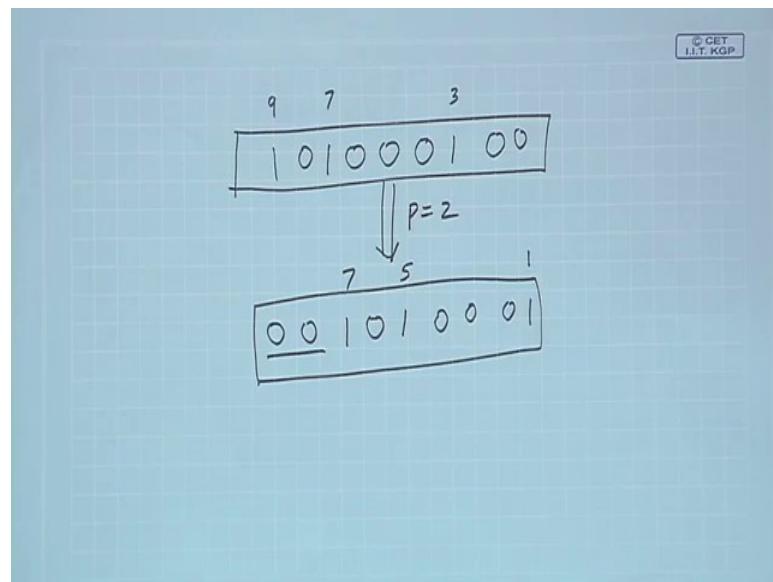
(Refer Slide Time: 13:23)



Now from the collision vector we construct a state diagram where we define the states as the status of collisions at a particular point in time. That means which time gaps or latencies can result in a collision at that point in time; that is one state. And the state transition diagram can contain multiple states; you can go from one state to the other.

And the initial state is the collision vector. And from the collision vector from any state you can move to some other state by advancing the time by some number p . Suppose you are currently in time t . If you advance the time by p , what you do. Whatever is the present state you shift it right by p positions.

(Refer Slide Time: 14:31)



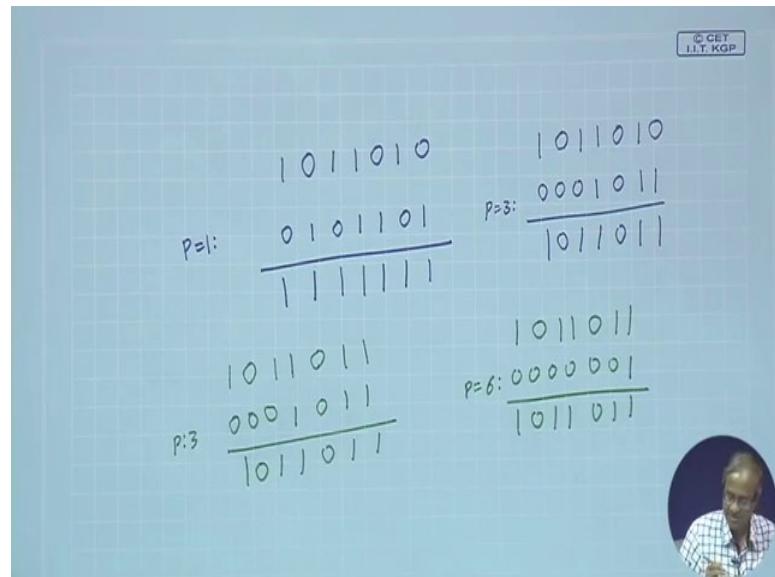
See what is the meaning? Suppose I am in a present state where my bits are something like this, which means that from time 1, 2, 3, 4, 5, 6, 7, 8, and 9; 3, 7, and 9 these are my forbidden times where I cannot feed the next input --- these are forbidden latencies. But now say if I advance my time by 2 steps then whatever was 3 now after two steps this will become 1, 7 will become 5, 9 will become 7. So, actually we are shifting this right by 2 positions. So, this 1 will come here this 0 0 0 1 0 1 and two zeros will be shifted in. This will be my new status after time 2. Now the collisions will occur at time 1, 2, 3, 4, 5, and 7; this is the idea.

So, the next state is obtained by shifting it right, and of course the initial collision vector that was there that lists the forbidden latencies. The latencies that are forbidden will always remain forbidden. So, whatever you have obtained by right shifting you will have to do a bit by bit logical OR with the initial collision vector, so that whenever there is a 1 in the initial collision vector it will remain a 1 in all the states, because those will always be forbidden you cannot use that latency at any point in time.

Let us take that same example. This is the state diagram for function X. Let us see how we have obtained this. We have said that for this function the collision vector was this; it indicates the forbidden latencies of 2, 4, 5, and 7. So, this will be my initial state. Now from my initial state I can advance my time only by that amount where I have 0's that

are not forbidden, which means time 1, 3, 6 or 8, or more. If I advance it by 1 you see my collision vector was 1 0 1 1 0 1 0.

(Refer Slide Time: 17:04)



If I advance it by time 1, I will be doing a right shift 1 0 1 1 0 1 and 0 will come in; then you do a bit by bit OR.

So, from this initial state that represents the collision vector if I go or advance time by 1, I go to this state just like I calculate it. If I advance it by 3, I go to this state, if I advance by 6, this you can check you will also arrive at the same vector.

But if I advance it by 8; that means, if you shift it by 8 everything will become 0, 0 OR with the initial vector will be the initial vector itself. Similarly from here there are no zeros, so you can only have 8 or more; shift it by 8 positions all will be zeros or with this you get back this. So, this will be an arrow back. And so on.

(Refer Slide Time: 20:47)

- From the state diagram, we can determine latency cycles that result in *minimum average latency (MAL)*.
 - In a *simple cycle*, a state appears only once.
 - Some of the simple cycles are *greedy cycles*, which are formed only using outgoing edges with minimum latencies.

Function X:	Function Y:
<ul style="list-style-type: none">• Simple cycles: (3), (6), (8), (1,8), (3,8), (6,8)• Greedy cycles: (3), (1,8)	<ul style="list-style-type: none">• Simple cycles: (3), (5), (1,5), (3,5)• Greedy cycles: (3), (1,5)

MAL = 3 for both X and Y

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL TECHNOLOGY TRANSFER CENTER

From the state diagram we can determine the latency cycles such that the minimum average latency is minimized. When we talk about a simple cycle a state appears only once. This is of course the kind of cycle that you want, because we do not want a cycle like this. for I am telling 3, 3, 8, 3, 3, 8, --- 3 is appearing twice. I shall only be considering simple cycles. And greedy cycles are those where from every state we shall only be taking the outgoing edges with minimum label.

(Refer Slide Time: 22:33)

```
Load collision vector (CV) in a shift register R;
If (LSB of R is 1) then
begin
    Do not initiate an operation;
    Shift R right by one position with 0 insertion;
end
else
begin
    Initiate an operation;
    Shift R right by one position with 0 insertion;
    R = R OR CVorig;           // Logical OR with original CV
end
```

The Scheduling Algorithm

Original CV (CV_{orig})

OR Gates

Shift Register R

0: safe
1: collision

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL TECHNOLOGY TRANSFER CENTER

Now, scheduling algorithm goes as follows. Suppose I want to build a control circuit that will automatically tell me whether it is safe to feed the next data or not. We load collision vector in a shift register. This is my original collision vector, and I load my collision vector in the shift register initially.

At every point in time I do a right shift, where I feed a 0 and I check whether is 0 is coming out or a 1 is coming out. If a 1 is coming out it means that next time instant is forbidden, you cannot feed a data at that time. If the last bit that is coming out is 1 then do not initiate an operation simply shift the register by 1 bit with 0 insertion, and in the next iteration again check whether you are allowed to feed or not.

But if you find that it is a 0 what does that mean with respect to the state diagram; that here 0 means you are going to the next state. Next state means you have to shift right and OR with the collision vector to get the new state. The same thing we are doing here. Here you see: if the bit is 0 initiate an operation then we shift it right and then we OR-ed with the collision vector; whatever is the shifted value we OR-ed with the collision vector and load back into the shift register, and this process repeats.

If you just think, whatever we are doing here is exactly the process by which we were computing the state diagram. At every step we are trying to shift it by a number of bits that is allowed. We go on shifting till the next 0 comes, then we OR it with the collision vector. And we always move when you get the first 0; that means we are looking only for greedy cycles. From every state we are moving out following the smallest or the least value of the cycle.

(Refer Slide Time: 25:16)

The slide is titled "Optimizing a Pipeline Schedule". It contains the following content:

- We can insert non-compute (dummy) delay stages into the original pipeline.
 - This will modify the reservation table, resulting in a new collision vector.
 - Possibly a shorter MAL.

Below the text is a diagram of a pipeline with three stages, S₁, S₂, and S₃. Stage S₃ has a feedback loop. To its right is a reservation table:

	1	2	3	4	5
S ₁	X				X
S ₂		X		X	
S ₃			X	X	

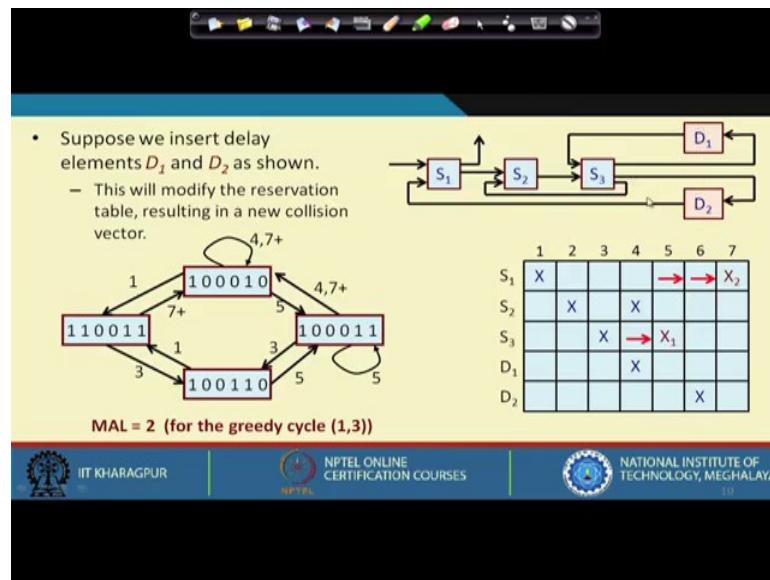
Below the table is a state diagram with nodes labeled 1, 2, 3, 4, 5+, and 3. The connections are: 1 to 2, 2 to 3, 3 to 4, 4 to 5+, and 5+ to 3. Below the state diagram is the text "MAL = 3".

At the bottom of the slide are logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

Now, there are ways to optimize a pipeline schedule. I am showing you one method with the help of an example. Suppose we have a non-linear pipeline like this where the reservation table is as follows. You see in some of the time steps like 4, two of the stages are simultaneously used. And here the trial forbidden latencies are 1 2 and 4; so 1 0 1 1 is the CV. The state diagram will be very simple; you can either shifted by 3 or 5 or more; obviously, 3 is the minimum average latency.

Now what you are exploring here is that: can we insert some delay elements in the pipeline, some dummy stages which do not compute anything just it take up some clock cycles? You may say that well if you insert dummy stages that do nothing simply eat up some cycles, then my time will be increasing. Well your time maybe increasing for a single computation, but when you are feeding many data one after the other earlier there were lot of collisions, but if you insert those delays maybe the number of collisions will become less; that is the idea.

(Refer Slide Time: 26:42)



So for this example, suppose I insert delays like this. This was my original pipeline; I insert delays in two of the branches: one delay here one delay here. That means, while going from S₃ to S₁ I put a delay, and S₃ to S₃ also I put a delay. So, earlier my reservation table was this, now in the reservation table there will be two additional dummy stages introduced. And because of the delay some of the check marks will get shifted: this will get shifted here, this will get shifted here. And see here after S₃ you first go to D₁, then you are going back to S₃. Earlier you are directly going to S₃. Similarly for this from S₃ you are going to D₂, from D₂ then you are going to S₁. Now for this one again it is a slightly more complex here the forbidden latencies are 2 and 6.

If you just compute the state transition diagram in a similar way you can check this. This is the state transition diagram that is obtained. And if you compute the greedy cycles in this case this will be the best cycle (1, 3); 1 3 1 3 1 3 that will be result in an average of 2. Earlier it was 3 now by inserting the delays we have achieved an average of 2. So, by inserting this kind of dummy delay stages the performance of a non-linear pipeline can be improved, this is something we have to keep in mind.

(Refer Slide Time: 28:34)

Exercise 1

- For the following reservation tables,
 - What are the forbidden latencies?
 - Show the state transition diagram.
 - List all the simple cycles and greedy cycles.
 - Determine the optimal constant latency cycle, and the MAL.
 - Determine the pipeline throughput, for $\tau = 20$ ns.

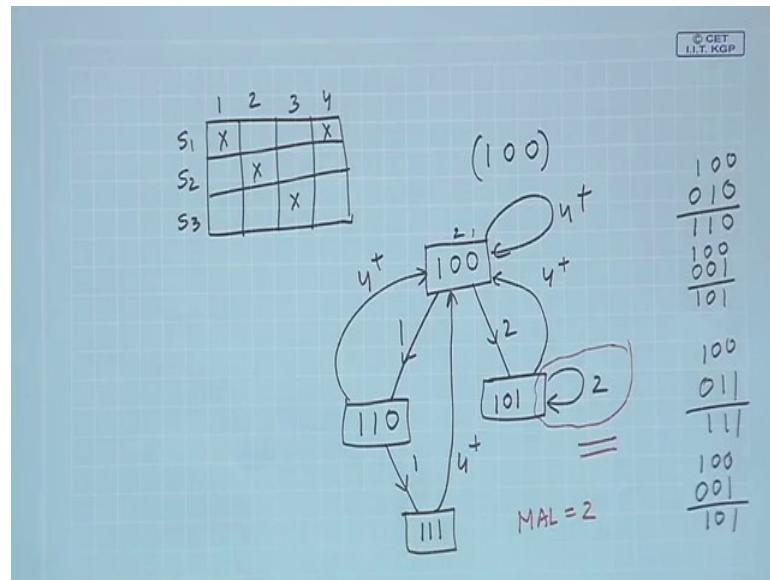
	1	2	3	4
S ₁	X			X
S ₂		X		
S ₃			X	

	1	2	3	4	5	6	7
S ₁	X		X				X
S ₂		X		X			
S ₃				X	X		

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us work out couple of examples. Let me work out one of them, and one of them I leave as an exercise for you. Let us take the smaller one.

(Refer Slide Time: 28:48)



Here we have a reservation table, where there are three stages: S₁, S₂ and S₃; and there are four time steps which are required for a computation. This is a very simple example where my only forbidden latency is 3. So, my collision vector will (1 0 0). If I draw the state transition diagram this will be my initial state. Now I can shift it by either 1 or 2. If I shift it by 1 what will happen; I will shift it right by 1 position it will be 0 1 0, I OR it

with the original collision vector I get 1 1 0. So, my new state will be 1 1 0. If I shift it by 2 positions it will be 0 0 1, again if I OR it with the original collision vector I get 1 0 1.

Now once you are here the only 0 is here, if we shift it by 1 position it will become 0 1 1 and OR it with original CV it will become 1 1 1. So, if you shift it by 1 you land up in another state 1 1 1. And here 2 is only forbidden. If you shift by 2 positions it will be 0 0 1, OR with 1 0 0; it will remain in 1 0 1. So, if you have 2 it will remain here.

If we analyze this state transition diagram we see that the minimum cycle is (2). So, minimum average latency will be 2. So, like this given any reservation table you can first calculate your state transition diagram, then from here you have to find the cycle that results in minimum average latency; that will be your minimum.

(Refer Slide Time: 31:38)

The image shows a screenshot of a presentation slide. At the top, there is a toolbar with various icons. Below the toolbar, the title 'Exercise 2' is centered in bold black font. The main content consists of a bulleted list of questions. The list starts with a general statement about two processors, X and Y, followed by two specific questions labeled 'a)' and 'b)'. At the bottom of the slide, there is a footer bar with three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya. The footer bar is blue with white text and logos.

Exercise 2

- A non-pipelined processor X has a clock frequency of 250 MHz and an average CPI of 4. Processor Y, an improved version of X, is designed with a 5-stage linear instruction pipeline. However, due to latch delay and clock skew, the clock rate of Y is only 200 MHz.
 - a) If a program consisting of 5000 instructions are executed on both processors, what will be the speedup of processor Y as compared to processor X?
 - b) Calculate the MIPS rate of each processor during the execution of this particular program.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us work out another example. This directly do not relate to instruction scheduling, but a problem of pipelining. Let us see what this states. It says a non-pipelined processor X where there is no pipelining has a clock frequency of 250 MHz and a CPI of 4. Because there is no pipelining it takes 4 clock cycles to execute one instruction. There is an improved version Y that is having a 5-stage linear pipeline, but because of the latch delays and other overheads the clock frequencies reduce to 200 MHz.

Now, two things are asked: that if I am running 5000 instructions on both the processors what will be the speedup. How fast will Y be with respect to X? And secondly, what will be the MIPS rating of the two processors?

(Refer Slide Time: 32:47)

The image shows handwritten calculations for two processors, X and Y. Processor X has an instruction count of 5000, a CPI of 4, and a clock cycle time of 250 microseconds. Processor Y has an instruction count of 5000, a CPI of 1, and a clock cycle time of 200 microseconds. The notes show the execution time for 5000 instructions for each, the speedup ratio, and the MIPS rating for each second.

$X: \quad XT = IC \times CPI \times CCT$ $XT_x = 5000 \times 4 \times \frac{1}{250} \text{ usec} = 80 \text{ usec}$ $XT_y = 5000 \times 1 \times \frac{1}{200} \text{ usec} = 25 \text{ usec}$ $\therefore \text{Speedup} = \frac{XT_x}{XT_y} = \frac{80}{25} = 3.2$	$X: \quad \begin{array}{rcl} 80 \text{ usec} & \xrightarrow{\hspace{1cm}} & 5000 \text{ instrs.} \\ \therefore 1 \text{ sec} & \xrightarrow{\hspace{1cm}} & \frac{5000}{80} \text{ MIPS} = 62.5 \text{ MIPS} \end{array}$
$Y: \quad \begin{array}{rcl} 25 \text{ usec} & \xrightarrow{\hspace{1cm}} & 5000 \text{ instrs.} \\ \therefore 1 \text{ sec} & \xrightarrow{\hspace{1cm}} & \frac{5000}{25} \text{ MIPS} = 200 \text{ MIPS} \end{array}$	

Let us work out for processor X. Let us calculate the execution time. We recall execution time is given by the instruction count multiplied by the cycles per instruction multiplied by the clock cycle time. So, for the first processor execution time comes to 80 microseconds.

Similarly, the execution time of the pipelined processor comes to 25 microseconds.

So, the speed up will be $80 / 25 = 3.2$.

Now, second part of the problem concerns the MIPS rating. Again for the machine X you see for executing 5000 instruction you require 80 microseconds. You can say that in 80 microseconds you are able to execute 5000 instructions. Therefore, in 1 second how many you execute? This comes to 62.5 million instructions per second.

Similarly, for Y it comes to 200 MIPS.

With this we come to the end of this lecture. In this lecture we talked about the various ways in which we can calculate the permissible latencies in a general pipeline. You see,

normally if it is just a linear pipeline all these calculations will not come into the picture. But if it is a more complicated kind of a pipeline with non-linear feed forward and feedback connections, then this latency calculation and latency analysis becomes important.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 42
Arithmetic Pipeline

In this lecture we shall be talking about arithmetic pipelines. Now, in a computer system if you see the way pipelines are used or utilized, you will find that they are used in broadly two areas or in two ways. One to speed up the execution of the instructions that is called instruction pipelining, and second to speed up the arithmetic operations that is called arithmetic pipelining.

So, today in this lecture, we shall be looking at some examples of the design of arithmetic pipelines.

(Refer Slide Time: 01:09)

Fixed Point Addition Pipeline

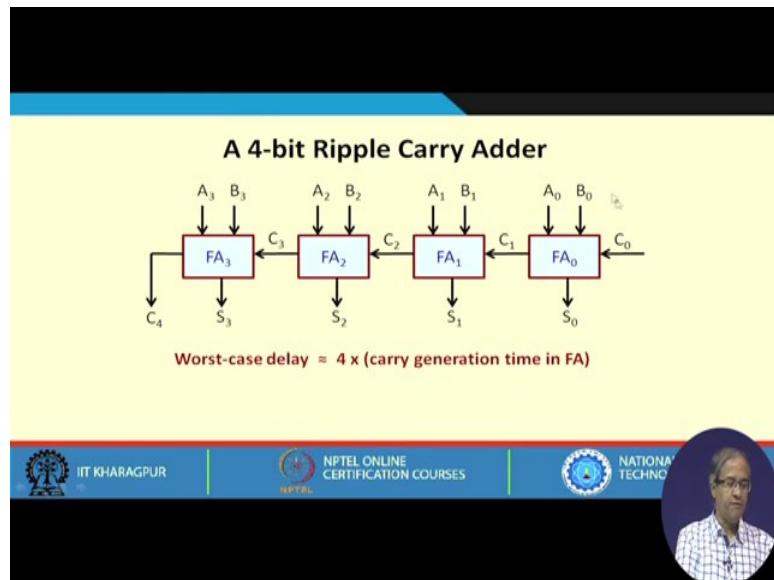
- We have seen how a ripple-carry adder works.
 - Rippling of the carries gives it a bad worst-case performance.
- We explore whether pipelining can improve the performance.
- *Assumption:* delay of a latch is comparable to the delay of a full adder.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

We start with a very simple example. We have already studied fixed-point addition; we have seen how a ripple carry adder works. In a ripple carry adder we use a cascade of full adders. Every full adder is carrying out the addition of a pair of bits, it generates sum, it also generates a carry, and the carry goes to the input to the next stage. So, this carry ripples like this and that is why the name is ripple carry adder. Earlier we have seen that due to the rippling nature of the carry, the performance of a ripple carry adder is not good. The worst case delay is pretty high in fact.

So, it has a bad worst case performance, but now let us see can how we use pipelining to improve the performance of a ripple carry adder. Well, one assumption we shall make. We shall be requiring latch or registers if we want to make a pipeline. Our assumption is that the delay of a latch will be comparable with the delay of a full adder.

(Refer Slide Time: 02:35)

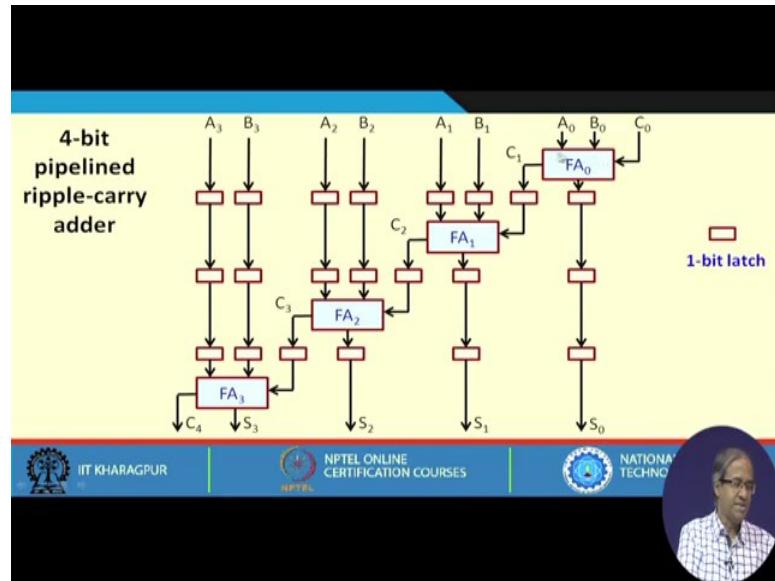


Let see this is the full adder design that we have seen earlier, this is a 4-bit ripple carry adder. In fact, there are 4 full adders. As I have said each of the full adders will be adding a pair of bits of the input numbers A and B, generating the sum bit and generating the carry bit for the next stage.

So, worst-case delay will be the first full adder will be generating a carry and because of that this carry will be generated, and again this carry will be generated. So, 4 multiplied by carry generation time in full adder, this will be roughly the worse case delay. Of course, there will be one more because of this sum generation. Anyway I am looking at the carry only. So, roughly speaking because it is a 4-bit ripple carry adder, the worst case delay will be four times the carry generation time of a single full adder.

Now, let us try to do one thing. Let us try to stretch this ripple carry adder in a skewed way, let us move the different stages of the ripple carry adder in the different time steps. Let us make them different stages in the pipeline.

(Refer Slide Time: 04:19)



What we are doing is that whatever ripple carry adder was here like this, we are making it like this. We are stretching it, you see we just ignore these pink boxes for the time being.

You see that the full adder is still a full adder, but what I have done is that we have drawn it in a skewed way, and these are the four stages of the pipeline we are defining, and the small pink boxes will be our latches. See in the first stage only, the last full adder will be working and the other input bit pairs which are coming; they will be simply stored in the latches. They will be used later not now.

After the first stage is finished, this carry C_1 has been generated and they will be stored in this latch, and this sum will be stored here. So, now stage 2 comes into the picture. In stage 2, only the second full adder is active. The others are simply dummy. These values are copied into the next lecture. Simply this sum is copied.

So, again this carry is copied, this sum is copied in the third stage again. There is a full adder here. These sums are copied, these bits are copied and in the last stage there is a full adder. So, you see we are using many latches, but when the input data is transferred to the second stage, we can parallelly feed the next set of data to the first stage. So, there can be the possibility of overlapped execution just like in a pipeline. This is the advantage we gain here.

(Refer Slide Time: 06:14)

- Delay of a full adder = t_{FA}
- Delay of a 1-bit latch = t_L
- Clock period $T \geq (t_{FA} + t_L)$
- After the pipeline is full, one result (sum) is generated every time T .
 - Convenient for vector addition kind of applications.

```
for (i=0; i<10000; i++)
    a[i] = b[i] + c[i];
```

Here is a simple calculation. If we just assume that the delay of a full adder is t_{FA} and a delay of a latches t_L , so the clock period T has to be greater than equal to this full adder delay plus latch delay. After the pipeline is full, we can expect one result in every time T . See earlier in a full adder roughly we are getting one result every time $4 \times$ delay of a full adder, but now in the pipeline implementation, we will be getting one output every clock cycle, and that is the advantage.

Suppose I have a program like this, where I am doing some addition a large number of times. Let say there are 10,000 additions. I can use this kind of a pipeline to advantage. After my pipe is full, I can get one output every cycle. Instead of waiting for one addition to be complete before starting the next addition, if we have a pipeline, it can definitely give you speedup for this kind of vector.

So, just like this if you are adding two arrays and number or the elements are being fed to the pipeline in sequence one by one, then this kind of arithmetic pipeline can be used to benefit. Basically wherever there is something called vector kind of arithmetic, pipelining can give you great benefit. This again we shall come back to later.

(Refer Slide Time: 08:22)

Floating-Point Addition

- Floating-point addition requires the following steps:
 - a) Compare exponents and align mantissas.
 - b) Add mantissas.
 - c) Normalize result.
 - d) Adjust exponent.
- Subtraction is similar.

Example:
A = 0.9504 × 10³
B = 0.8200 × 10²

Align mantissa: 0.0820
Add mantissa: 0.9504 + 0.0820 = 1.0324
Normalize: 0.10324
Adjust exponent: 3 + 1 = 4
Sum = 0.10324 × 10⁴

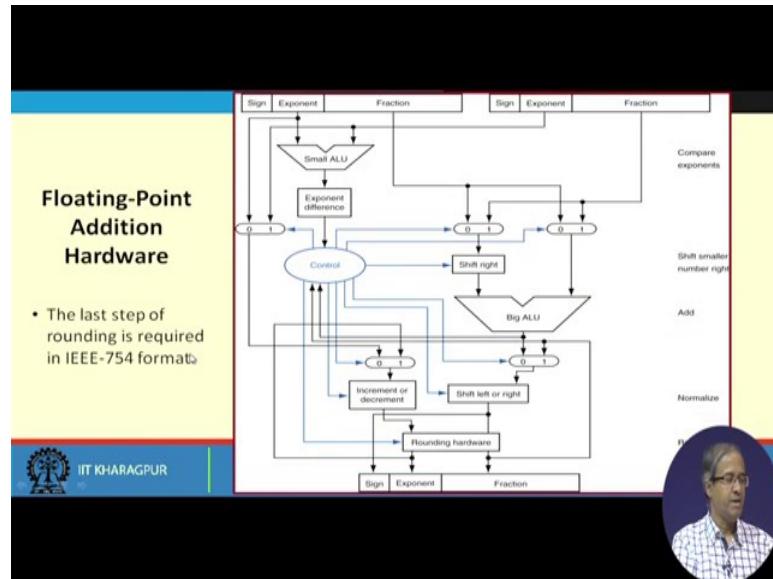
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us now come back to floating point arithmetic. That is how it can be pipelined. Let us look at floating-point addition, this we have already seen in detail. Earlier floating-point addition requires these steps of comparison of exponents and alignment of mantissas. Then, addition of mantissas, normalization of the result, and because of normalization, you may need to adjust the exponent. Subtraction will be very similar. Instead of addition we will be subtracting; an example for two decimal numbers are shown here.

Suppose I have two numbers A and B; with exponents 2 and 3. I make the second one also 3. This is called alignment of mantissa. After alignment I add 0.9504 and 0.0820. I get this let say normalization means it starts with 0. So, here I do a normalization. That means, I shift right by one position.

So, exponent was 3, but because I have shifted right, I have to adjust the exponent and I have to make it 4. So, my final result will be this into 10 to the power 4. These steps can easily be mapped into the various pipeline stages.

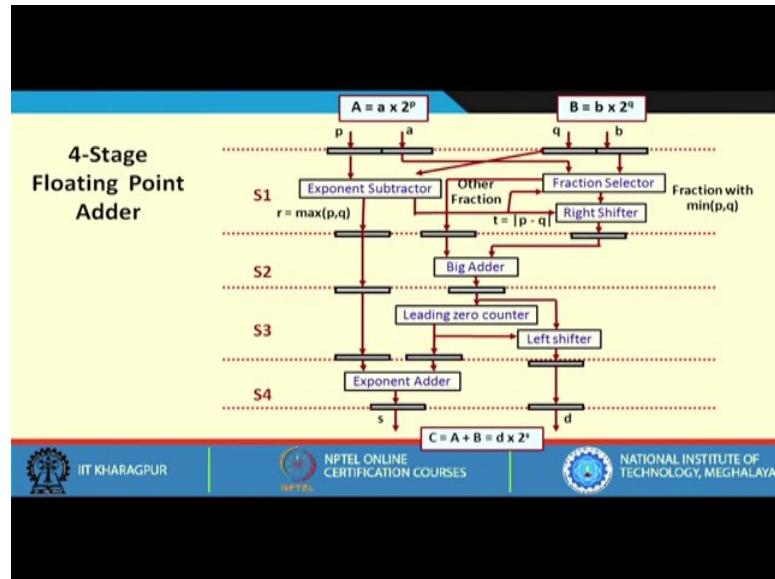
(Refer Slide Time: 10:02)



This is the floating-point addition hardware that we saw earlier. This is a non-pipelined version. Just a quick recap: the two numbers to be added over here, sign, exponent and the mantissa of the fraction, first we are comparing the exponent by using a subtraction using a small ALU, and after subtraction you see the difference in the exponents will have to this, smaller number fraction by that many bits. So, there will be a shifter. After shifting, there will be multiplexer to select the correct fraction, there is an ALU which will be carrying out addition and after addition, we will have to do normalization. We will have to check whether you have to do increment or decrement, according the shifting left or right and for IEEE format, you may have to do a rounding in the last step. So, that step is also shown.

This already we have seen earlier. Now, these basic steps you can easily break into as pipeline stages. I am showing one simple implementation of a pipeline like this. Let say the two numbers are $A \times 2^P$ and $B \times 2^Q$.

(Refer Slide Time: 11:14)



Here I am not showing the sign, just the mantissa and the exponent. This is P and this is A, this is Q and this is B. In the first stage S1, I am comparing the exponent. There is an exponent subtract. This P is coming and this Q is coming. That is subtracting.

The output will be R which will indicate which one is greater. That means, R is the greater of the two, P or Q. That also is known and after doing this, you select the fraction which one is smaller. So, either A or B, one of them is selected and after subtraction whatever is the result that many time P - Q, these many positions you are doing a right shift. This is your mantissa alignment. So, after mantissa alignment, you take the other fraction here and this shifted fraction here, you do an addition in stage S2. You are adding the mantissas.

In stage S3, after adding you have to do normalization. You will have to have a circuit that will count the number of leading 0's and depending on that, it will be shifting the mantissa left by that many positions. This is stage S3 and depending on how many positions you have shifted your exponent will have to be added by that number. So, this exponent correction, well here I am not showing normalization. Just simple floating point addition as a pipeline and these small rectangular boxes are the latches. So, finally you get the result, and it is a 4-stage simple pipeline.

(Refer Slide Time: 13:38)

Floating-Point Multiplication

- Floating-point multiplication requires the following steps:
 - Add exponents.
 - Multiply mantissas.
 - Normalize result.
- Division is similar.
A last step of rounding is required in IEEE-754 format.

Example:
 $A = 0.9504 \times 10^3$
 $B = 0.8200 \times 10^2$

Add exponents: $3 + 2 = 5$
Multiply mantissa: $0.9504 \times 0.8200 = 0.7793$
Normalize: 0.7793 (no change)
Product = 0.7793×10^5



IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, floating-point multiplication is a little simpler because here you do not need to align the mantissas before the operation. You simply add the exponents and multiply the mantissas. Then, normalize like if you have two numbers like this. You add the exponents $3 + 2 = 5$. You multiply the mantissas straightforwardly and it is 0.7793. It is already normalized. No change and see your result is this into 10 to the power 5.

Division is similar to multiplication. Here you will be doing subtraction, here you will be doing division and again for IEEE format at the end, you may need a step of rounding.

(Refer Slide Time: 14:31)

A MULTIFUNCTION PIPELINE FOR ADDITION AND MULTIPLICATION

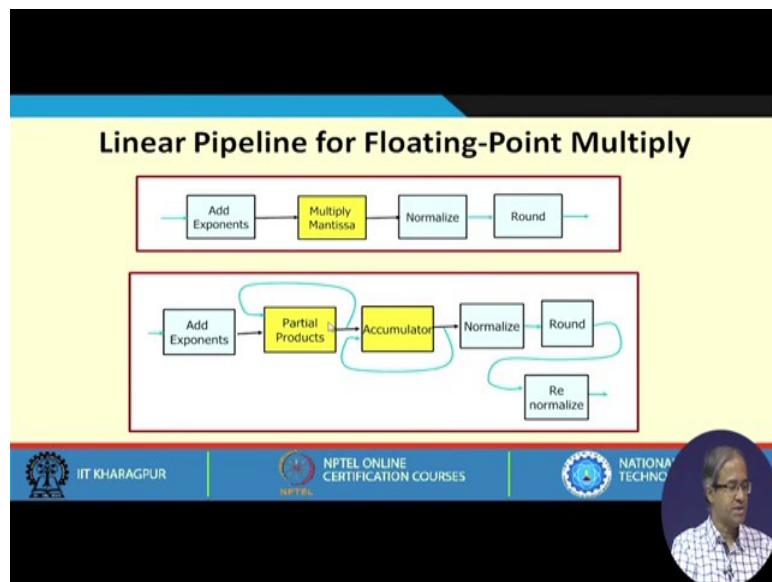


IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA



For multiplication and division we have already seen earlier how we can implement that. Now, we will be seeing a multifunction pipeline for addition and multiplication together, because already we have seen non-linear pipelines in our last lecture. This is a simple pipeline for floating point multiply.

(Refer Slide Time: 14:49)

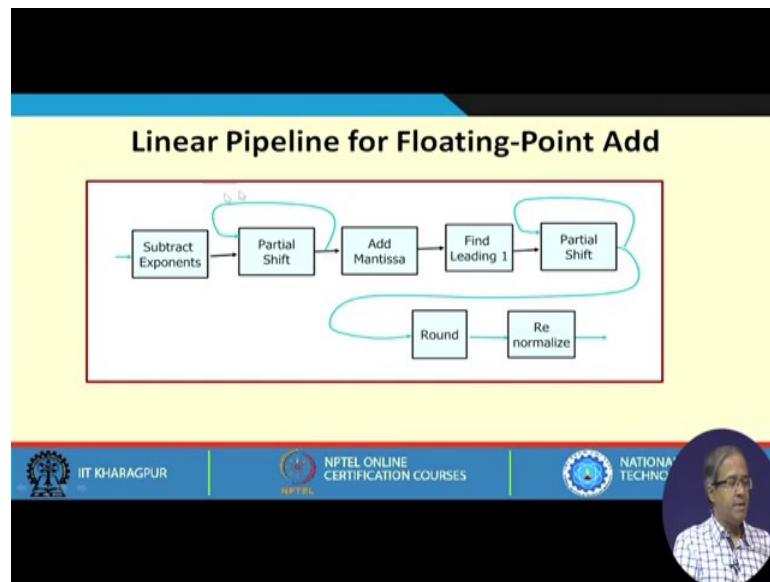


I am showing a simplified diagram. You do not add exponents, multiply mantissa, normalize rounding, but the thing is that when I am saying add exponents, multiply mantissa for example, or normalize these may require multiple steps because multiplication and addition are not of the same complexity. Multiplication takes more time than addition. You may have to compute the partial products, then add them. It may need multiple clock cycles.

So, technically more correct visualization will be something like this multiply mantissa. You can break it up into two boxes. One is the generation of partial products. This also will be an iterative process, and for every iteration, you will be doing an addition, you will be accumulating the partial result sum.

So, both these two stages may be required to execute it for multiple clock cycles and normalize and rounding and at the end, you may need to do renormalizations after rounding because the number might become again un-normalized.

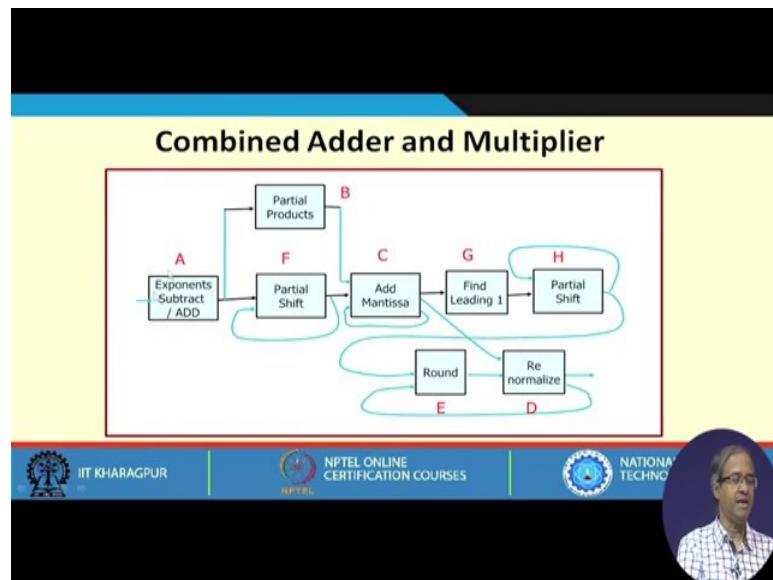
(Refer Slide Time: 16:37)



This is a more technically correct version of a floating point multiply pipeline with the IEEE format, because for IEEE format, you need this last step. And if you look at the floating point add, you need a mantissa alignment initially. For mantissa alignment, you have to shift the mantissa certain number of times. This again can be an iterative step; after the shifting, then you do an addition, you find the number of leading ones, then you will have to do a right shift or left shift.

Here left shift by that many position, this also can require multiple number of cycles, the rounding and for IEEE format renormalize. So, you see with respect to the previous diagram, there are some similarities.

(Refer Slide Time: 17:29)



We can combine both the diagrams into a single pipeline like this. This is a combined adder and multiplier. You see this partial shift will be required for addition, partial product will be required for multiplication, but for both, these cases mantissa addition is required. Similarly partial shift will be required for addition, but for multiplication you can straight away go from add mantissa to renormalize. You can skip these two steps because leading one and partial shift will be required only for addition. From multiplication you can straightaway skip G and H; you can straightaway come to D here.

So, if you have a multifunction pipeline like this, I am not going into much detail on this. I will just give you an idea and then, you can similarly construct the reservation tables for addition.

(Refer Slide Time: 18:28)

The slide shows a reservation table for multiply operations. The table has 8 rows labeled A through H and 7 columns labeled 1 through 7. Red 'X' marks indicate reserved resources. Row A has an 'X' at column 1. Row B has 'X's at columns 2 and 3. Row C has 'X's at columns 3 and 4. Row D has an 'X' at column 5. Row E has an 'X' at column 6. The table is used to determine forbidden latencies and collision vectors.

	1	2	3	4	5	6	7
A	X						
B		X	X				
C			X	X			
D					X		X
E						X	
F							
G							
H							

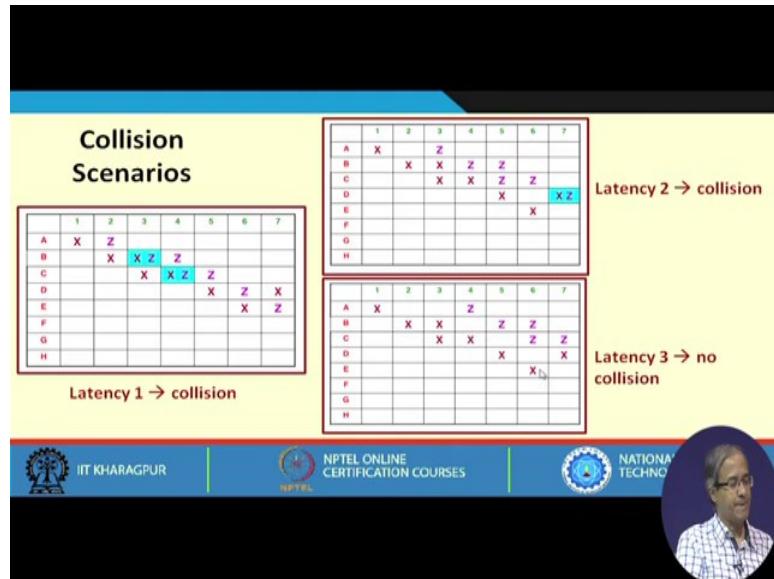
- Forbidden latencies: 1, 2
- Collision Vector: (0 0 0 0 1 1)
- MAL = ?

The slide also features logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, along with a portrait of a man.

For multiply operations, you will be needing ABCD and what we are saying is that steps B and steps C will be requiring little more time. We are seeing that it is taking two time steps because partial shifting or the partial products and the add mantissa, this will take more time, and because of that I am assuming that this stage is B and C are being used for extended period.

For this reservation table, you see that the forbidden latencies are 1 and 2. So, the collision vector will be this. So, intuitively you can say that your minimum average latency will be 3, because 1 and 2 are both forbidden. So, 3 we can use definitely here. So, let us see for latency 1, there will be collision.

(Refer Slide Time: 19:43)



The slide shows two reservation tables illustrating collision scenarios. The top table is labeled 'Latency 1 → collision' and the bottom one is labeled 'Latency 2 → no collision'. Both tables have rows for stations A through H and columns for time slots 1 through 7.

Latency 1 → collision:

	1	2	3	4	5	6	7
A	X	Z					
B		X	X	Z	Z		
C		X	X	Z	Z		
D			X	Z	X		
E				X	Z		
F					X	Z	
G						X	Z
H							X

Latency 2 → collision:

	1	2	3	4	5	6	7
A	X	Z					
B		X	X	Z	Z		
C		X	X	Z	Z		
D			X	Z	Z		
E				X	Z	X	
F					X	Z	
G						X	Z
H							X

Latency 3 → no collision:

	1	2	3	4	5	6	7
A	X	Z					
B		X	X	Z	Z		
C		X	X	Z	Z		
D			X	Z	Z		
E				X	Z	X	
F					X	Z	
G						X	Z
H							X

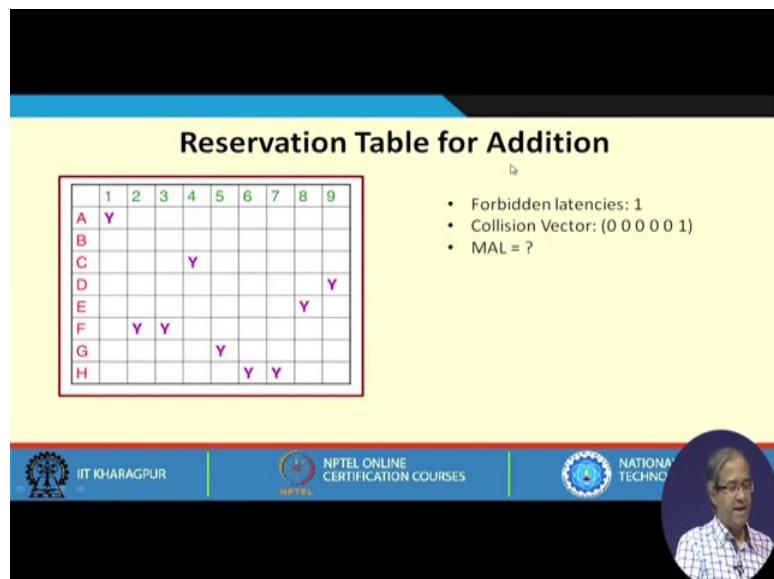
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL TECHNOLOGY



You see this was the reservation table. So, this X is the first data. Suppose I am feeding the second data, I mean it is after one cycle. So, Z has shifted here, Z has shifted here, X as you see X and Z will collide in B and C. So, latency 1 will lead to a collision. So, X and Z refer to consecutive data items which are fed, Z is fed earlier X is fed now.

Now, if I use latency 2 that will also lead to a collision. Latency 3 does not lead to collision.

(Refer Slide Time: 20:53)



The slide shows a reservation table for addition operations. The table has rows for stations A through H and columns for time slots 1 through 9. The table is labeled 'Reservation Table for Addition'.

	1	2	3	4	5	6	7	8	9
A	Y								
B									
C			Y						
D								Y	
E									Y
F	Y	Y							
G			Y						
H				Y	Y				

Forbidden latencies: 1
Collision Vector: (0 0 0 0 0 1)
MAL = ?

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL TECHNOLOGY



For multiplication you can feed the data with a latency of 3 without any problem. Similarly for addition, the reservation table will look like this. So, here again this F and H are been used extended period here. There is only one forbidden latency, 1. Here minimum average latency will be equal to 2. So, here you can similarly justify that.

(Refer Slide Time: 21:17)

The slide has a black header and a yellow body. The title 'Summary' is centered in the yellow area. Below it is a bulleted list of three items. At the bottom, there is a footer bar with logos for IIT Kharagpur, NPTEL, and National Technology Engineering and Research Institute (NTERI), along with a video player showing a person speaking.

- Arithmetic pipeline is a standard feature in modern-day processors.
- Mandatory for vector processors, which are designed specifically to operate on vectors of data.
- We shall see the impact of arithmetic pipeline in MIPS32 instruction pipeline implementation later.

To summarize this arithmetic pipeline is a very standard feature in modern day processors. We shall see later in some detail about vector processors. Vector processors are computer systems that are specifically designed to operate on vectors of data very fast. We shall of course first be discussing later in the next week that how we can build or how we can convert our MIPS architecture that we have seen earlier into a pipeline, how we can execute the instructions faster, but then we shall see that how we can also augment that pipeline with arithmetic pipeline concepts, so that vector arithmetic operations can also be made faster.

What we have seen today? We have seen some very simple examples of some arithmetic pipelines, and one instance of a multifunction pipeline of multiplier and added together. We have seen that we can speed up operations by implementing arithmetic circuits as a pipeline, provided we have continuous stream of data to be fed to that pipeline. That is possible only if we have vectors of data to be operated on.

We shall see the impact of arithmetic pipeline, we shall see later that arithmetic pipeline can complicate the design of the MIPS32 pipeline, but this we shall be seeing only later.

With this we come to the end of this lecture. There are a few things we shall be discussing after this; we shall be looking at the input-output process in a computer system, how peripheral devices are interfaced to a computer system, what are the different characteristics of the peripheral devices that makes the interfacing somewhat more complicated or complex as compared to interfacing memory devices.

And also, you shall be looking at some of the commonly used bus standards that we see everywhere today, and of course, after that we shall be looking at how we can make our computers faster by incorporating pipeline at the instruction level, and also as I had said at the arithmetic level. So, with this we stop for today.

Thank you.

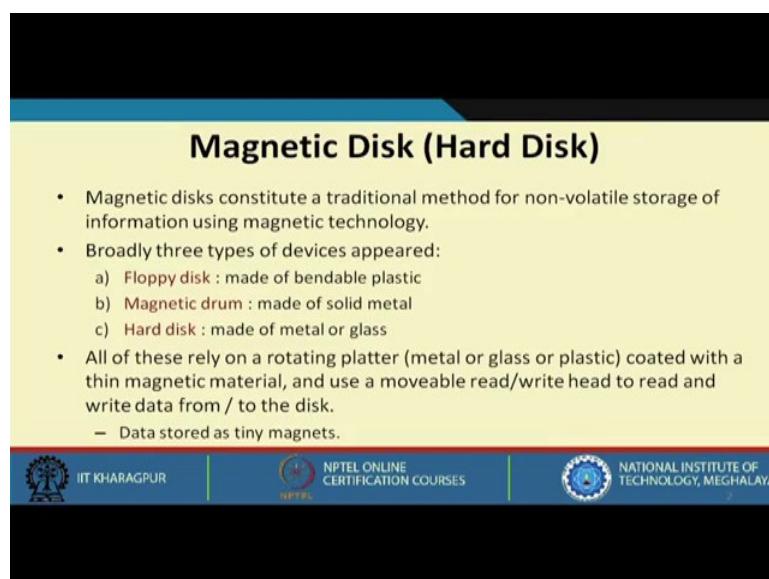
Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 43
Secondary Storage Devices

In this week we shall be starting our discussion on input output devices and the ways in which you can interface such devices to the computer system. Now when we talk about interfacing IO devices the most important kind of IO device interfacing that we encounter is that of the secondary memory, which traditionally has been the hard disk.

Well of course, very recently there is a new technology which has come up; the so called solid state disks that are based on something called flash memory technology. They are also very fast replacing the hard disk technology. The topic of our discussion in this lecture is secondary storage devices.

(Refer Slide Time: 01:15)



Magnetic Disk (Hard Disk)

- Magnetic disks constitute a traditional method for non-volatile storage of information using magnetic technology.
- Broadly three types of devices appeared:
 - a) Floppy disk : made of bendable plastic
 - b) Magnetic drum : made of solid metal
 - c) Hard disk : made of metal or glass
- All of these rely on a rotating platter (metal or glass or plastic) coated with a thin magnetic material, and use a moveable read/write head to read and write data from / to the disk.
 - Data stored as tiny magnets.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

We start with our discussion on magnetic disks which has been around for several decades already. Now actually how does a magnetic disk work? Magnetic disks, as the name implies, rely on some kind of magnetic technology. There is some kind of ferromagnetic material coated on the surface of the disk, and with the help of some externally applied magnetic field we can change the orientation of tiny magnets on the surface.

Depending on the orientation we can say that we are storing either 0 or 1, this is the basic idea. Now because of the accuracy of positioning those electromagnets on the surface of the disk it is possible to store a very large number of zeroes and ones in the form of small magnets on the surface of the disk.

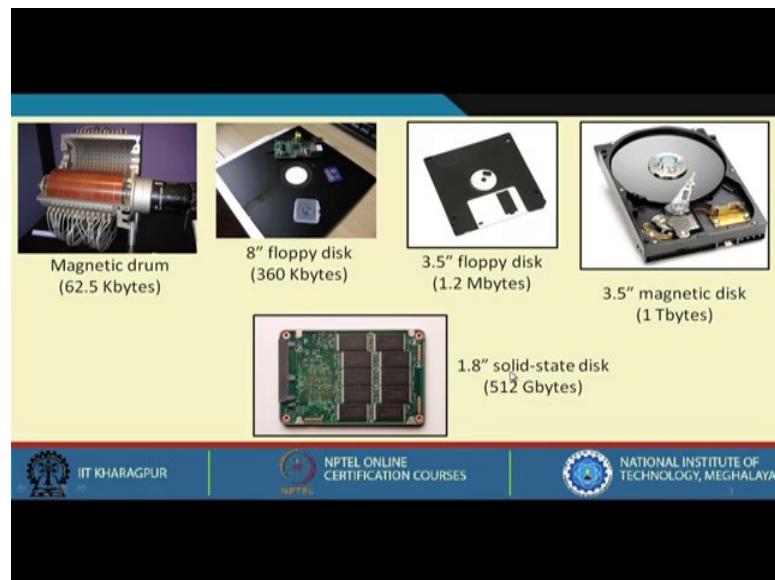
This is the traditional method that has been used since many decades. One characteristic of this kind of a storage media is that it is non volatile. Non volatile means whenever the power is withdrawn or the power is switched off, whatever you are storing will remain; that will not go off because it is based on magnetic technology, the magnets will remain even when we withdraw the power. So, this is non-volatile and it uses magnetic technology.

Now, over the years there are broadly three kinds of magnetic storage devices that have appeared. Magnetic drum was the first to appear, then floppy disks and hard disks. Magnetic drum traditionally was built out of solid metal on the top of which some ferromagnetic material was coated. Floppy disk was similar, but it was made of plastic, which was soft and you could bend the surface of the floppy disk, that is why the name was floppy.

In contrast the hard disk that you see today are really hard; they are made of metal or glass, you cannot bend the disks without breaking. Now all these different devices which I have talked about they are all relying on some kind of rotating platter. Either it is a wheel which is rotating or it is a drum which is rotating. So, the idea is that something is rotating and there is some kind of a sensor or an actuator we call it the read write head. The read write head is sitting in one place and the surface is rotating or moving under that read write head. So, you can read or write the bits on the surface.

The rotating platter as I had said are made of either metal or glass or plastic. They are coated with a very thin magnetic material and as I had said there is a read write head, for some of the devices the read write head is fixed like in magnetic drum, but for floppy and hard disks they can also move. And as I had said the data are stored as tiny magnets.

(Refer Slide Time: 05:06)



Here we have some pictures of the different kinds of devices which had appeared in the market over the years. You see this was the first kind of device to appear in the late 60s and early 70s; the magnetic drum. As you can see there was a cylindrical kind of a thing which was rotating and there were read write heads positioned along the axis. All these read write heads were able to read or write data bits on the surface, and in this kind of a device the total capacity was only about 62.5 kilobytes; it was pretty less in those days.

Later on in the 1980s floppy disks appeared. The first kind of floppy disks were of 8 inch size. The diameter was 8 inch; this kind of a disk was able to store 360 kilobytes of data. Subsequently we had a smaller version of the floppy disks 3.5 inch, which some of you may still be using today. Of course, these have become almost obsolete, their capacities are 1.2 megabytes, then came the hard disks.

Well here the picture that I am showing is the picture of a 3.5 inch magnetic disk, where the capacity can be as high as one terabytes or even more. But believe me the first hard disks which appeared in the market were pretty large in size. The diameter was more than a foot. So, it is more than 12 inches in diameter they were really big, but over the time it has become smaller; now we can have three and half inch hard disks.

Now, I am showing another kind of a device here. This is technically not a hard disk, but this is replacing hard disk, this is the so called solid state disk. Here there is something which is rotating, and there is a read write head which moves on the surface. So, there is

some kind of mechanical movement here; because of mechanical movement the overall life of the device will be relatively less.

In contrast the solid state disk does not have any moving parts, they are based on integrated circuit technologies. They are chips, there are no mechanically moving parts and secondly, they are also smaller in size. This is an example which is an 1.8 inch disk, the capacity is 512 gigabytes.

(Refer Slide Time: 07:59)

• Since the platters in a hard disk are made of rigid metal or glass, they provide several advantages over floppy disks:

- They can be larger.
- Can have higher density since they can be controlled more precisely.
- Has a higher data rate because it spins faster.
- No physical contact with read/write head as it spins faster.
 - The read/write head floats on a cushion of air (few microns separation).
 - Requires dustless environment.
 - Results in higher reliability.
- More than one platters can be incorporated in the same unit.

IIT KHARAGPUR | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

Now, let us come back to hard disk. In a hard disk as I had said the platters are made of metal or glass which are rigid. There are some advantages because in floppy disks the disk surface is soft it can bend. Because the hard disks are rigid they can be made larger because you can more accurately position the read write head on a solid surface. If it is soft it can bend, and a slight bend can lead to a misalignment of the read write head; you can read the wrong data by accident. Because of that hard disks can have higher density and they can be controlled more accurately or precisely.

And in a floppy disk the read write head was actually physically coming in contact with the disk and the disk was rotating. In contrast in a hard disk the disk is rotating at a much higher speed and the read write head is not touching the surface, there is a thin layer of air, the read write head floats on top of the surface. Because of that the wear and tear on the surface is also much less.

Because its spins faster the data rate can also be higher because you can read write at a higher rate, there is no physical contact as I had said. So, the read write head actually floats on a thin layer of air which is few microns thick. Because of this even if there is a single spec of dust on the surface of the disk the read write head will crash on the dust because the layer of air is even thinner than the spec of dust. So, the disk has to be put in a dust free enclosure.

This obviously results in higher reliability and for higher capacity you can have more than one disks in the same unit. There can be several disks, you can put them one on top of the other; several disk can rotate at the same time. They will all be having separate read write heads something like this.

(Refer Slide Time: 10:34)

Organization of Data on a Hard Disk

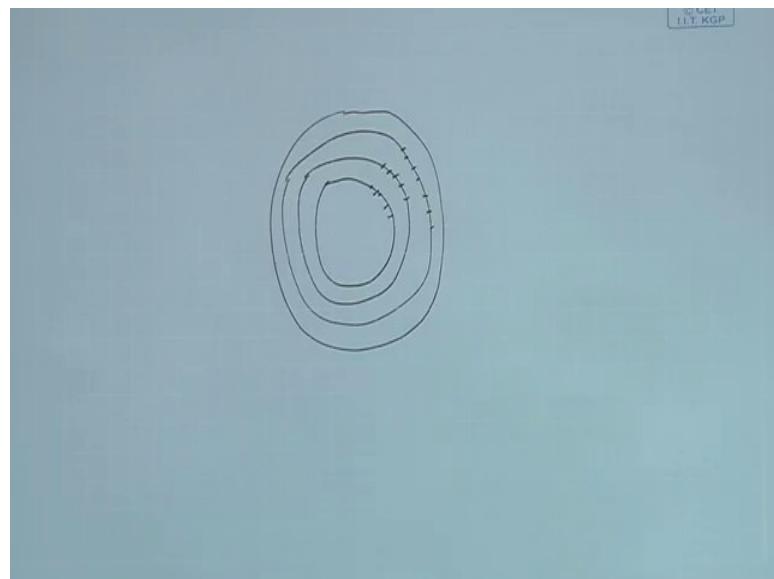
- The hard disks consists of a collection of *platters* (typically, 1 to 5), which are connected together and can spin in unison.
 - Each platter has two recording surfaces, and comes in various sizes (1 – 8 inches).
 - The stack of platter typically rotates at a speed of 5400 to 7200 rpm.
 - Each disk surface is divided into concentric circles called *tracks*.
 - The number of tracks per surface can vary from 1000 to 5000.
 - Each track is divided into a number of *sectors* (64 – 200 sectors/track).
 - Typical sector size: 512 – 2048 bytes.
 - Sector is the smallest unit that can be read or written.
 - The disk heads for all the surfaces are connected and move together.
 - All the tracks under the heads at a given time on all surfaces is called a *cylinder*.

Let see how data are organized on a hard disk. As I had said each disk is called a platter, platter is a circular hard material which is magnetically coated typically on both the sides. Now a magnetic disk unit as I had said can contain more than one platter which rotates all simultaneously.

So, there is a collection of platters, typically the number of platters will be 1 to 5 which are connected together, and as I had said they spin together. Each platter has two recording surfaces that are magnetically coated on both sides, and the platters can come in various sizes. There are many disks of about one inch diameter also available, but the larger ones are about 8 inches in diameter.

But the more commonly used ones today are typically three and half inch diameter, there are larger ones also. And these platters rotate at speed typically that range from 3600 revolutions per minute (RPM), to 5400 or 7200 RPM.

(Refer Slide Time: 12:17)



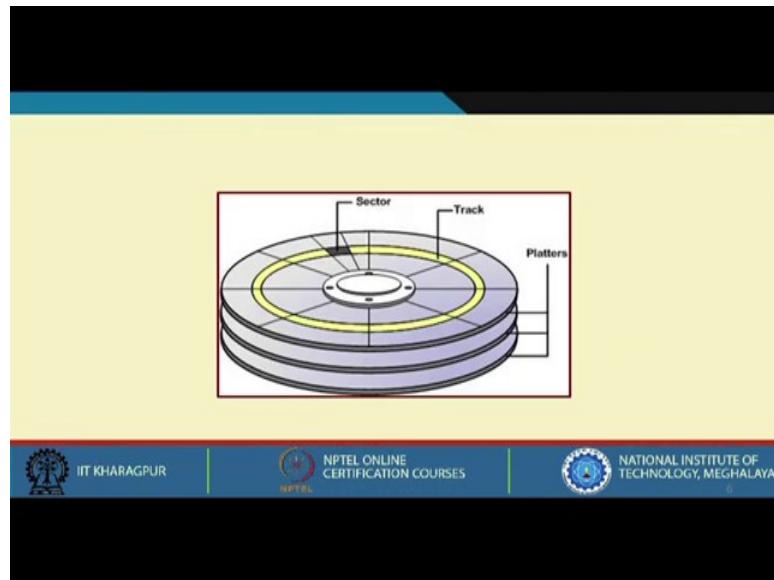
Each disk's surface is divided into concentric circular regions called tracks. If this is the top view of a platter, then you can have the circular regions which are concentric circles. These are called tracks and we store the individual data bits along these tracks. This is unlike record player, for those of you have seen how a record player works. In a record player, you put the playback pin on one side and it is a helix kind of a thing, it slowly moves towards the centre; that is how a record player stores the data. But in a disk they are concentric circles called tracks, and typically the number of tracks can vary from 1000 to 5000.

And again each track is divided into a number of logical sectors. There can be 64 to 200 sectors per track, and each sector has size typically ranging from 512 to 2048 bytes. And with the disks the point to notice that sector is the smallest unit of data that you can read or write. This means you cannot read a single byte or you cannot write a single byte; you will have to read an entire sector or you will have to write an entire sector, this is something you have to remember.

Sector is the minimum unit of data that can be transferred to and from the disk, and the disk heads for all the surfaces as I had said are connected together and move. And if

there are multiple platters all the tracks under the heads at a time is called a cylinder. So, there are several platters on each of the platter there is a track. Track number 1 on all the surfaces taken together form something called a cylinder. Similarly track 2 they form another cylinder, track 3 they form another cylinder, like that the same track on all the surfaces or the platters constitute a cylinder.

(Refer Slide Time: 15:01)



Pictorially you can see these are the platters. This yellow ring is the track, this is the concentric circle and the track is broken up into a number of small sectors, one of the sector is shown here. There are many sectors. And for individual disks or platters there are magnetic coatings on both sides. So, if there are three disks or platters, then there can be six recording surfaces.

(Refer Slide Time: 15:39)

Disk Access Time

- There are three components to the access time in hard disk:
 - a) Seek time:
 - The time required to move the head to the desired track.
 - Average seek times are in the range 8 – 20 msec.
 - Actual average can be 25 – 30% less than this number, since accesses to disks are often localized.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, when you talk about the access time of a disk there are three components to it. First is called the seek time. You see a disk contains a number of tracks, right now the read write head may be located somewhere. Suppose you want to read data from track number 10. The first thing is that you have to move the head to track number 10; that movement time is referred to as the seek time.

Seek time is the time required to move the read write head to the desired track, and this seek time on the average can range from 8 to 20 milliseconds. So, you see these times are of the order of milliseconds, but actually this number on the average can be 25 to 30 percent less, because for every stored data on the disks we normally do not have to move across many tracks, normally the related data are stored in nearby tracks. So, we have to move the read write head by smaller amounts that will be requiring less time. If you move the head across many tracks the time required will be more. So, this is seek time.

(Refer Slide Time: 17:08)

b) Rotational delay:

- Once the head is on the correct track, we must wait for the desired sector to rotate under the head.
- The average delay or latency is the time for half the rotation.
- Examples:
 - For 3600 rpm, average rotational delay = $0.5 \text{ rotation} / 3600 \text{ rpm} = 8.30 \text{ msec}$
 - For 5400 rpm, average rotational delay = $0.5 \text{ rotation} / 5400 \text{ rpm} = 5.53 \text{ msec}$
 - For 7200 rpm, average rotational delay = $0.5 \text{ rotation} / 7200 \text{ rpm} = 4.15 \text{ msec}$

At the bottom, there are logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

Next comes the rotational delay. Suppose we have reached the correct track. On the tracks suppose I want to read data from sector number 5. So, I will have to wait until sector number 5 rotates under the head. That is the rotational delay; I will have to wait for the disk to rotate, so that the data I want to read or write comes under the head. We must wait for the desired sector to rotate under the head.

On the average this delay will be half the rotational delay, because in the best case you may already be there, or in the worst case you may have to wait for an entire rotation. So, on the average it will be half. Some example value of rotational delay for 3600 rpm disk is shown. Half rotation per 3600 rpm, if you divide this and if you calculate you will get 8.3 milliseconds. For 5400 rpm the delay becomes 5.53 milliseconds, and for 7200 it becomes even less 4.15 milliseconds.

(Refer Slide Time: 18:55)

The slide has a yellow header and footer area. The main content area is black. In the yellow header, there is a question: "c) Transfer time:". Below it is a bulleted list:

- The total time to transfer a block of data (typically, a sector).
- Transfer rates are typically 15 MB/sec or more.
- Transfer time depends on:
 - Sector size
 - Rotation speed of the disk
 - Recording density on the tracks

In the yellow footer, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

The data transfer rate in bytes per second or kilobytes per second whatever you say, is the total time required to transfer a block of data, say a sector or a track, in a disk. The typical transfer rates are typically of the order of 15 megabytes. The transfer time depends on several things, the sector size, rotation speed of the disk because faster the disk is rotating my speed will be higher, and also recording density on the tracks.

Are we recording the zero and ones very sparsely or very densely? If you are recording very densely then the reading and writing you can also do faster because the disk is rotating at the same speed. So, all these things will determine how fast we can read or write the data.

(Refer Slide Time: 20:00)

Example 1

- Consider a disk with sector size 512 bytes, 2000 tracks per surface, 64 sectors per track, three double-sided platters, and average seek time of 10 msec.
 - What is the capacity of the disk?
 - If the disk platters rotate at 7200 rpm, and one track of data can be transferred per revolution, what is the transfer rate?
 - Bytes/track = $512 \times 64 = 32K$
 - Bytes/surface = $32K \times 2000 = 64,000K$
 - Bytes/disk = $64,000K \times 3 \times 2 = 384,000K$
 - Transfer rate = Capacity of a track / average rotational delay
 $= 32K / 4.15 \text{ msec} = 7,711 \text{ Kbytes/sec}$

Let us work out a simple example. Let us consider a disk, where the sector size is 512 bytes, there are 2000 tracks per surface 64 sectors per track, and there are 3 platters double-sided, so there are 6 surfaces, and average seek time is 10 millisecond. The first question is; what is the capacity of the disk and the second question is if the disk rotates at 7200 rpm and we assume that one track of data can be transferred per revolution what is the transfer rate?

In the first question, capacity of the disk calculation is shown.

The transfer rate calculation is also shown.

(Refer Slide Time: 22:26)

Some Recent Advancements

- a) Most of the modern-day disk units include a high-speed cache directly in the disk unit.
 - Allows fast access of data that was recently read between transfers requested by the CPU.
- b) In conventional disks, each track contains the same number of bits.
 - Outer tracks record data at a lower density than inner tracks (circumference of a circle is proportional to its radius).
 - An alternate scheme uses *constant bit density*, where the outer tracks store more bits than the inner tracks.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Some recent advancements in hard disk technology they allow two things. Firstly, if you break open a disk sometime you will see that other than the disk surface there are lot of circuits inside. Among other things they include a high speed memory that is called as disk cache directly inside the disk unit. Whenever data are transferred from the disk, they are first stored in the cache, and whenever cpu requests they can be directly accessed from the disk cache. So, it will much faster that way. This use of the disk cache speeds up the data transfer of the average.

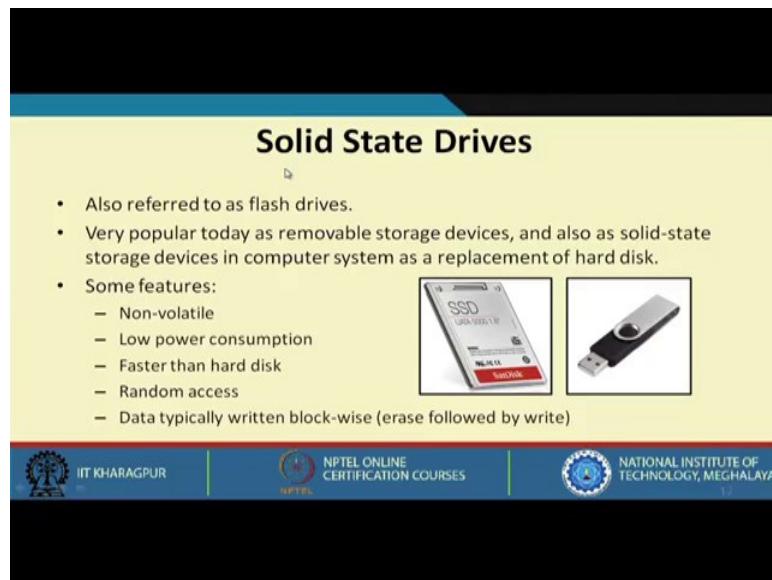
And the second advancement is that in the conventional earlier disks. So, each of the tracks has the same capacity means if one track content 32 kilobytes of data, the innermost track which is much smaller will also contain 32 kilobytes of data. So, irrespective of the circumference of the tracks the capacity was the same. But, imagine the tracks which are outside the circumference is much larger. So, potentially they can store more data.

In the earlier disks it was not done that way. Now you can have a scheme called constant bit density scheme where the outer tracks can store more bits than the inner tracks, this is in contrast with the traditional or conventional way where the outer and inner tracks all stored the same amount of data, which means the outer tracks recorded data at a lower density.

(Refer Slide Time: 24:34)

Solid State Drives

- Also referred to as flash drives.
- Very popular today as removable storage devices, and also as solid-state storage devices in computer system as a replacement of hard disk.
- Some features:
 - Non-volatile
 - Low power consumption
 - Faster than hard disk
 - Random access
 - Data typically written block-wise (erase followed by write)



The slide has a yellow header bar with the title 'Solid State Drives'. Below it is a white content area containing a bulleted list of features. To the right of the list are two small images: one of an SSD (Solid State Drive) and one of a pen drive. At the bottom of the slide is a dark footer bar with three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Now, let us come to the other competing technology which is now threatening to replace the hard disks, the so called solid state drives. We are already using solid state drives in our daily life in the form of the pen drives, pen drives are everywhere. Pen drive is one form of solid state drive that is storage where there is no moving part, it is electronic, it is non-volatile, and it is very convenient. Also it is faster than hard disk this is another very important characteristics.

Solid state drives are very popular today as removable storage devices. You can see a picture of a traditional solid state hard disk, and a pen drive which you all know. These devices are already replacing hard disks in many computer systems like laptops. Some of the features are unlike hard disks, they are non volatile, they have extremely low power consumption, they are faster than hard disk; at least 10 times faster, random access and data are written blockwise. One thing I did not mention; this device does not have any moving parts like a hard disk, that is another advantage. So, the life is also higher.

(Refer Slide Time: 26:08)

The slide has a dark blue header and a light yellow main content area. At the top center, it says 'Floating-Gate MOSFET'. Below that is a bulleted list of points. At the bottom, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Floating-Gate MOSFET

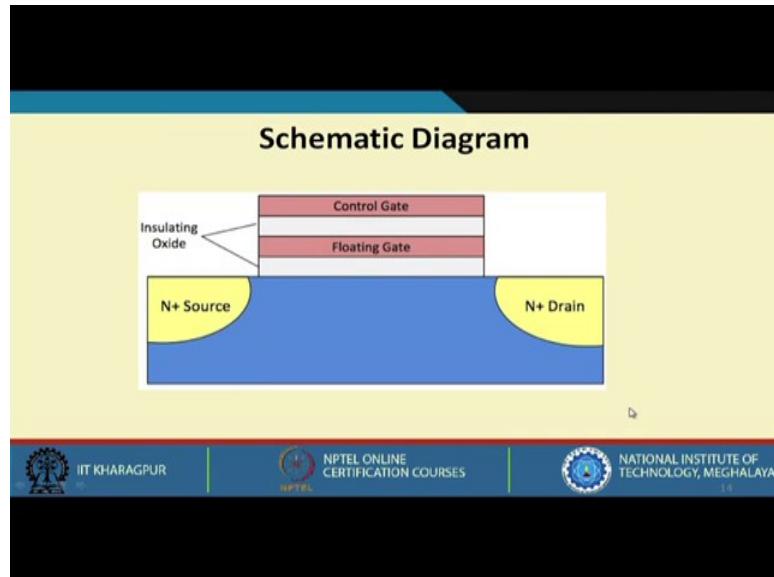
- The floating-gate MOSFET is a semiconductor device whose structure is similar to a conventional MOS transistor.
- The gate of the transistor is electrically isolated, and is referred to as floating gate (FG).
 - Since FG is surrounded by highly resistive material (insulator), the charge contained in it remains intact for long periods of time.
- By applying a suitable voltage on the control gate, the charge in FG can be controlled.
 - Presence or absence of charge can indicate binary states (0 or 1).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

The basic technology that is used to built this kind of flash or solid state memory devices is called a floating gate MOS transistor. The floating gate MOSFET is a device whose structure is somewhat similar to a conventional MOS transistor, which is used in CMOS technology for building chip today.

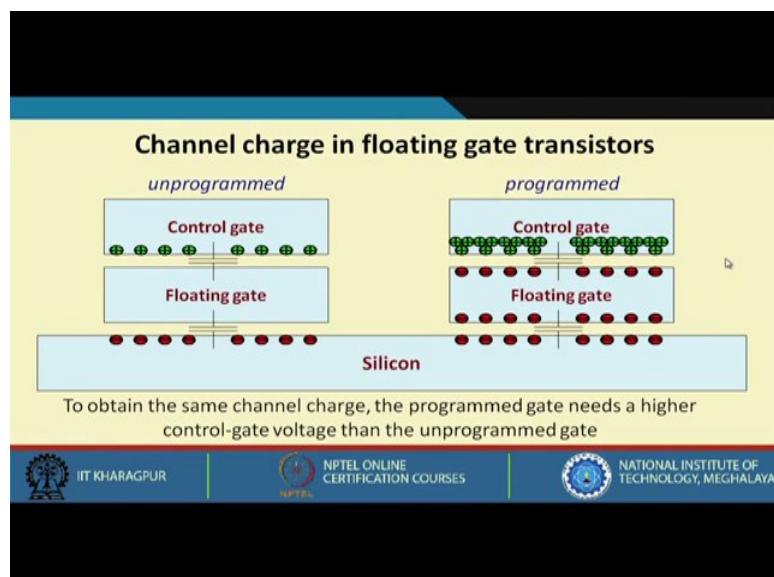
The difference is that the gate of the transistor is electrically isolated, and this is referred to as floating gate. Electrically isolated means there is an insulating material and inside that insulating material the gate is residing or is hidden inside. If you can somehow put some charge on the gate there is no path for the charge to leak out, the charge will remain. This is the basic idea how it can memorize some information; you put or force some charge in the gate, and because of the insulation the charge cannot leak out and it can be retained for quite long amount of time.

(Refer Slide Time: 27:45)



By applying a suitable voltage on a control gate you can control the charge and the presence or absence of the charge can be representative of the binary state 0 or 1. This is how you can store. Pictorially the floating gate transistor looks like this. Just like a normal MOS transistor we have the n-type regions. One is the source other is drain; there is a gate, but in between there is another floating gate which is covered by insulating oxide.

(Refer Slide Time: 28:18)



By applying some voltage on the control gate you can inject some electrons or some charges within the floating gate like this. Suppose when there is no charge on the floating gate, when you apply some normal voltage on the control gate some electrons will be attracted on the channel just like a normal transistor and between the source and drain a current will be flowing.

But when you program this gate like you inject some charge in the floating gate; that means, already some electrons are injected then to have this channel you have to put in more voltage on the control gate to create the channel. So, just like checking this that how much voltage is required for a current to flow you can check whether this gate is charged or not charged, whether it is 1 or 0.

(Refer Slide Time: 29:12)

The diagram illustrates a 2D grid of transistors. It consists of two sets of parallel lines: red lines labeled 'Control gate lines' and blue lines labeled 'drain lines'. The intersections of these lines form a grid where each intersection point represents a single transistor. The grid is composed of several rows and columns of these intersections.

- Reading a bit means:
 - Apply a voltage V_{read} on the control gate.
 - Measure the drain current I_d of the FG transistor.
- The transistors are laid out on a 2-dimensional grid.

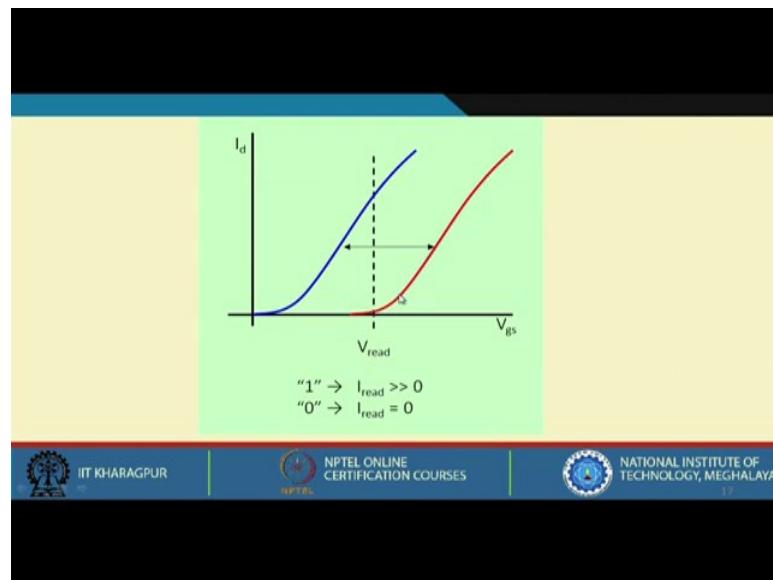
IIT KHARAGPUR

NPTEL ONLINE CERTIFICATION COURSES

NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

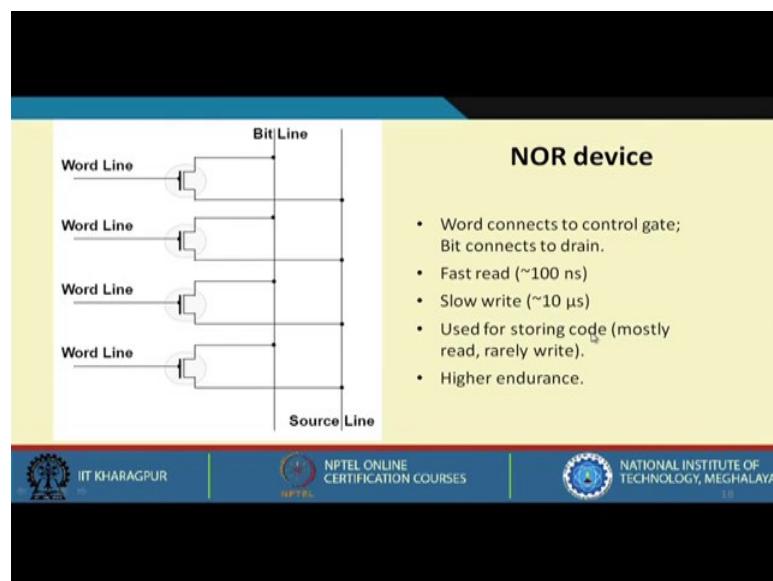
Now, when you talk about reading a bit it means you apply a voltage on the control gate and measure the drain current whether the current is flowing or not; and typically this transistors are laid out on a two dimensional grid. These red and blue lines are on two different levels they are not touching, and the transistors are fabricated at the junctions; one is the drain other is the source, this is the gate and transistors connected like this.

(Refer Slide Time: 29:45)



These are the characteristic curves. As you can see for one state the current will be very less, and for the other state the current will be quite high. So, just by measuring the current you can sense that in which state your device is in, 1 or 0.

(Refer Slide Time: 30:10)



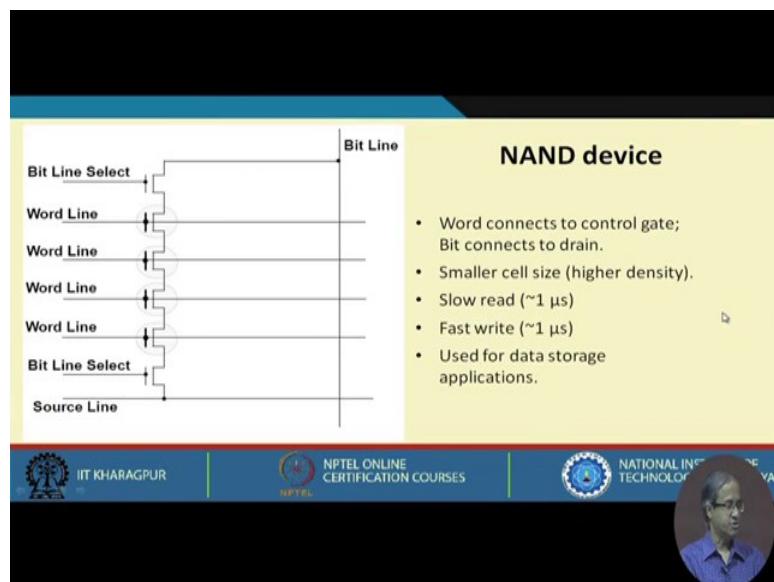
Now, the solid state device can be fabricated in two ways, one is called NOR flash or a NOR device, other is called NAND device. A NOR device looks like this. Here the floating gate transistors are fabricated like this, so all the sources are made common and the drains are connected to the bit line and the word lines are connected to the gate. So,

when you enable a transistor, whatever you apply you can measure the current between source and bit line, and you can check whether you are storing 1 or 0. By applying a high voltage you can inject some charge on the floating gate; this is how NOR devices are laid out.

Now, the characteristics of NOR devices is that reading of data is very fast, of the order 100 nanosecond. So, you can compare the time with that of hard disk where the seek time itself takes around 10 milliseconds plus rotational delay another 10 milliseconds. But write is relatively slow or because in write you have to apply a higher voltage and have to wait for a longer time for the charge to leak inside the floating gate; that takes about 10 microsecond.

So, this is a memory which can be read fast, but writing is slower. So, you can use it in applications where you are storing some program code which you are mostly reading, but rarely writing. This kind of NOR devices have higher endurance, but the capacity is smaller because layout wise it occupies more area.

(Refer Slide Time: 32:17)



The other device is NAND device. You can see here there are two wires coming out of each transistor that is why it takes more area, but for a NOR device the transistors are connected in series like this, there is a bit select line on two side I am not going to detail. The source is here and bit line if they are activated, when you are selecting this line there will be a current flowing you can sense it.

In case of NAND device the cell size is smaller resulting in higher density. All these solid state drives that you see today are built using this kind of NAND technology, because it has higher density. Now here reading and writing times are almost similar, slower read as compared to NOR, but as compared to NOR writing is faster. So, read and write are almost the same time and this is used for data storage applications like pen drives, solid state drives etc.

(Refer Slide Time: 33:24)

The slide has a yellow header bar with the title 'Some Characteristics of NAND Flash'. Below the title is a bulleted list of characteristics:

- Typical operations supported:
 - Read / Write a page (typical page size = 512 bytes)
 - Erase a block (set of pages)
- A block must be erased before it can be written.
- Wear leveling – an important consideration.
 - Maximum number of erases/writes per cell is typically ~1M.
 - Reliability of the cells decrease over time.
 - Wear leveling tries to evenly distribute cell accesses over the entire array.
 - Write page may mean copy-and-write.

At the bottom of the slide, there are logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Rourkela, along with a small portrait of a man.

Some characteristics of NAND flash are as follows. Some typical operations that are supported are read or write a page. Again you cannot read or write a byte or word. You can write an entire page which is of the order 512 bytes, because you see in a NAND so many things are connected in series. So, when you do an operation you have to operate on all of them, and by applying a voltage you can erase all the blocks in a page; means all the cells in a block. A block is defined as a set of pages and another thing is that when you write into such a device first you have to erase the block only then you can write.

Another important consideration in this kind of flash is something called wear levelling. You see the life of the devices specified by the manufacturer as 1 million writes, that the manufacturer tells you, but if by accident you are writing most frequently in a few of the cells, those few cells will go bad very quickly because they will reach 1 million, but the others may be still fresh.

Wear levelling means you consciously try to distribute the writes across all the cells. If you see that you are writing into one cell many times you will explicitly make a copy of that cell to some other cell and write it there. So, lot of copies and writes are required due to wear levelling considerations. Wear levelling tries to evenly distribute cell accesses, write page may mean copy to some other position and then write.

With this we come to the end of this lecture where we tried to give you a brief idea about the technologies that are used to built a hard disk or a solid state disk. Due to their characteristics and differences whenever you are designing or building a memory hierarchy or a computer system you will be able to take a more informed decision based.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 44
Input - Output Organization

In this lecture we start our discussion on Input Output Organization. Now, we shall be starting to talk about how input output devices can be interfaced to the computer system and what are the different ways in which you can transfer data to and from the input output devices.

(Refer Slide Time: 00:47)

The slide has a black header bar. Below it, a teal bar contains the title 'Introduction'. The main content area is yellow and lists reasons why interfacing I/O devices is complex:

- Interfacing input/output devices is more complex as compared to interfacing memory systems.
- Why?
 - Wide variety of peripherals (keyboard, mouse, disk, camera, printer, scanner, etc.).
 - Widely varying speeds.
 - Data transfer rate can be regular or irregular.
 - Sizes of data blocks transferred at a time varies widely (few bytes to Kbytes).
- Slower than processor and memory.

At the bottom, there is a blue footer bar with three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Let us see why this input output interfacing is so important. The first thing to note is that interfacing I/O devices is more complex as compared to interfacing memory systems. When we talk about interfacing memory systems which you saw earlier, there can be static RAM there can be dynamic RAM, well static RAMs are faster than dynamic RAM you know, but when you interface the speeds of the memory systems are known to us, they are not that widely varying. You see normally static RAMs are used to build the cache memory, dynamic RAMs are used to build the main memory.

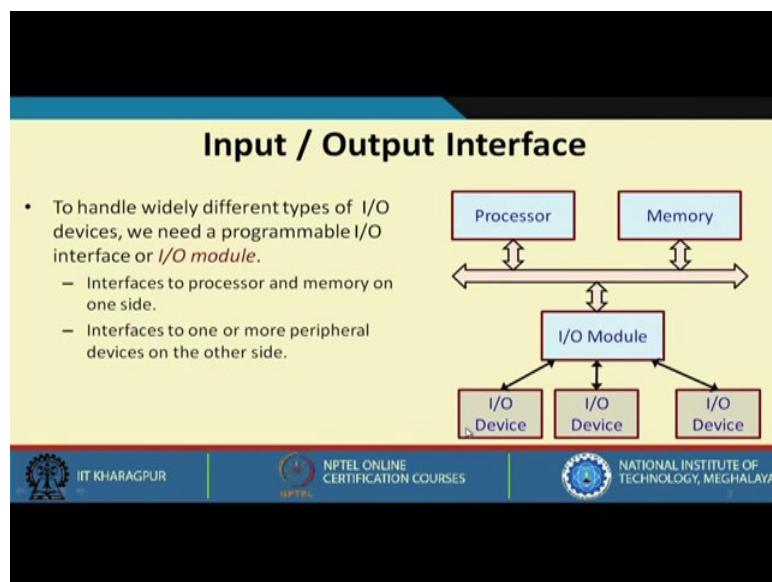
So, when you are designing the interface for the main memory, it is all dynamic RAMs, their speeds are all very uniform. The way CPU transfers data to and from memory are also very similar. But you think of the I/O devices we have very slow devices like a

keyboard or a mouse where the computer has no idea when the next input will be given, as a user you can press the next key immediately or you can even press it after one hour. But when you are interfacing a disk or a some kind of high speed I/O devices, then the data might be coming at a much faster data, e.g. scanner or camera. So, the data transfer speed varies widely from one device to another, and their characteristics are also widely different.

That is what is mentioned here. The main reason is the wide variety of peripherals that are existing starting from keyboard and mouse very slow ones, disk camera the faster ones, printer is also not very fast, it comes in between scanner. There are so many devices, they are widely varying in speeds. Not only that the data transfer characteristics can be regular or irregular; like when you think of a camera which is continuously sending you data, the data transfer is in some way regular.

But for the other kind of device like scanner, its scans one page then it stops, then after a while it scans another page then stops. So, it is irregular in some sense, and the size of the data block depends very much on the type of the I/O device.

(Refer Slide Time: 04:33)



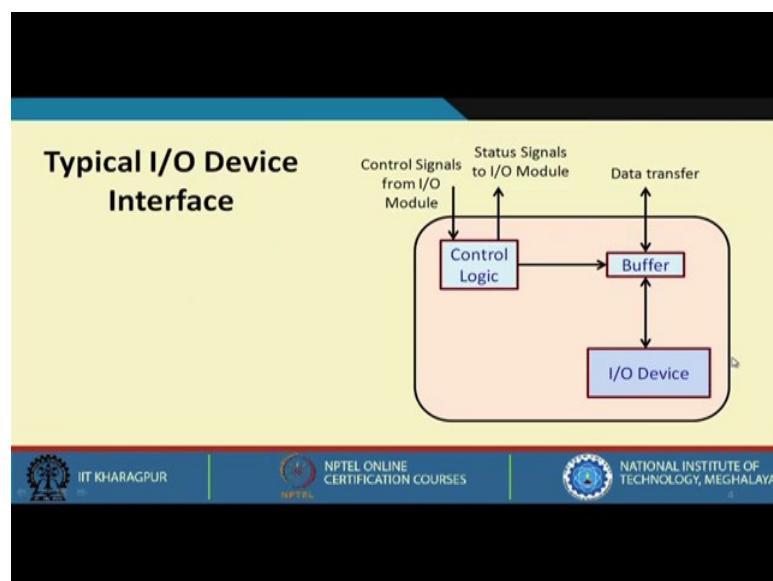
Size of data block transferred can vary widely, it can be a few bytes it can even be few kilobytes. Let us say a disk it can transfer a sector or a block (collection of sectors). In general I/O devices are slower than both processor and memory.

Just a schematic diagram we are showing here which tells you something about the input output interface.

You see here we have the main processor bus that connects the processor with the memory. This is the most important and the highest speed components in the system, then we have the interface called I/O module. You need a separate I/O module; you cannot afford to connect all the I/O devices directly to this bus, because of the reason that you mentioned these I/O devices have so widely varying characteristics. This I/O module in some way can take care of these variations, and it can present a uniform interface to this bus.

The I/O module on one side interfaces to the processor and memory through this bus, and on the other side it interfaces to one or more peripheral or I/O devices.

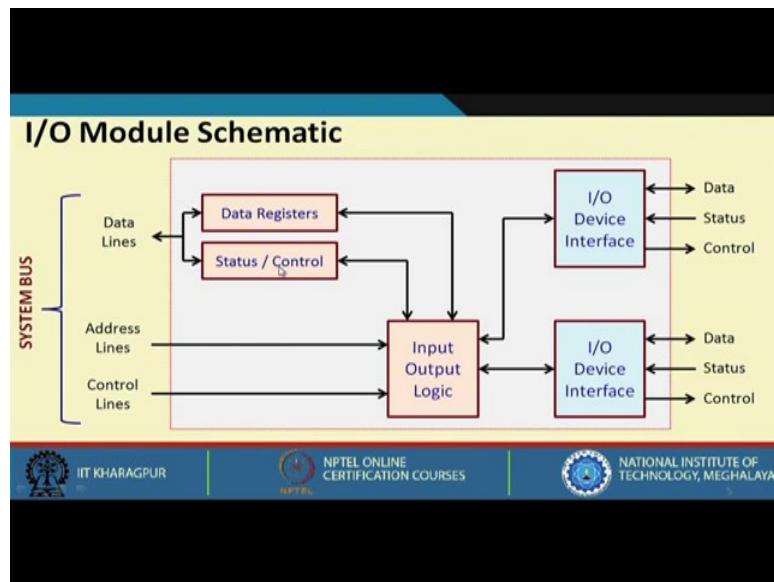
(Refer Slide Time: 05:55)



In typical I/O device interface shown in this diagram, when you say this I/O device is connected to the I/O module, exactly how the interface is like. See the I/O module signals are here. The I/O module can send some control signals, it can receive some status signals, or it can do some data transfer. In the previous diagram these arrows which I show will carry all these three kinds of signals; control signals, status signals and data transfer.

Now, inside the I/O device interface in addition to the I/O device we have a buffer which temporarily stores the data that is been transferred, and there is a control logic, which controls the buffer and there will be other circuitry also. This status signal will tell you whether the I/O device is ready, whether the next data word or data byte is ready, and so on and so forth. This is a typical I/O device interface.

(Refer Slide Time: 07:06)



Now, the I/O module if you look at ,overall it looks like this. On one side you have the I/O devices. For each of the I/O devices if you again look at the previous diagram, three kind of signals are required: control, status and data. Each of the I/O device interface has control, status and data. And other side you have the system bus, where there is an address bus, data bus and also the control bus. The address bus will provide you with the address of the I/O device, which I/O device you are trying reading or writing. So, the I/O logic will select that appropriate device and the control lines will tell you exactly what is the operation being carried out.

Depending on that the data line will be carrying either some data or some status or control. The CPU may want to control the device, send some control signal, or it may want to read the status of the device. This is during the configuration phase and during the data transfer phase the same line will be sending or receiving data.

(Refer Slide Time: 08:36)

The slide has a dark blue header and footer. The main content area is yellow. The title 'Typical Steps During I/O' is centered at the top of the yellow section. Below the title is a list of six steps:

- a) Processor requests the I/O Module for device status.
- b) I/O Module returns the status to the processor.
- c) If the device is ready, processor requests data transfer.
- d) I/O Module gets data from device (say, input device).
- e) I/O Module transfers data to the processor.
- f) Processor stores the data in memory.

At the bottom of the slide, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

This is roughly how the I/O module looks like. Now, typical steps during the I/O operation. The first step can be that the processor requests the I/O module for device status, because the processor wants to do some kind of I/O. The processor is not sure whether the device is ready to carry out the operation.

After sometime the I/O module will return back the status to the processor, whether the device is ready or not. Once the processor knows the status of the I/O device, if it is ready then the processor initiates the data transfer, read or write. Suppose it is a input device. Now, this input data transfer will be initiated and I/O module will be getting the data from the device, this data will be sent to the processor, which will store the data in memory. This process will repeat. This is how typically the data transfer takes place between the I/O module and the I/O devices via the processor.

So, you see the processor needs to periodically check the status of the I/O device, that is very important. Whether the device is ready; if it is ready only then I will try to read or write, otherwise I will wait because you have to understand the device is much slower than cpu. cpu may be requesting very fast, but the I/O device does not have that speed, it is much slower. So, cpu will have to wait till the I/O device is ready, only then it can read or write the data.

(Refer Slide Time: 10:38)

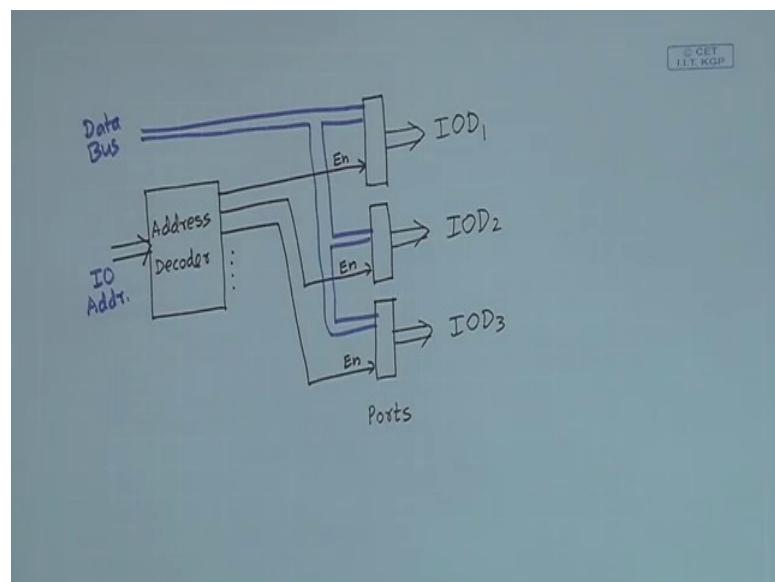
How are I/O devices typically interfaced?

- Through input and output ports.
- Output port:
 - Basically a PIPO register that is enabled when a particular output device address is given.
 - The register inputs are connected to the data bus, and the register outputs are connected to the output device.
- Input port:
 - Basically a parallel tristate bus driver that is enabled when a particular Input device address is given.
 - The driver outputs are connected to the data bus, while the inputs are connected to the input device.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let see how the I/O devices are typically interfaced. The interface is through something called input or output ports. Now what is an output port? I will give an example in next slide. Output port is nothing but a parallel-in parallel-out register which is enabled when a particular output device address is given. The idea is suppose I have several I/O devices.

(Refer Slide Time: 11:21)



Let us say all are output devices. I will be having several I/O ports. On this side I am connecting the I/O devices, this is I/O device 1, I/O device 2, I/O device 3, and I am

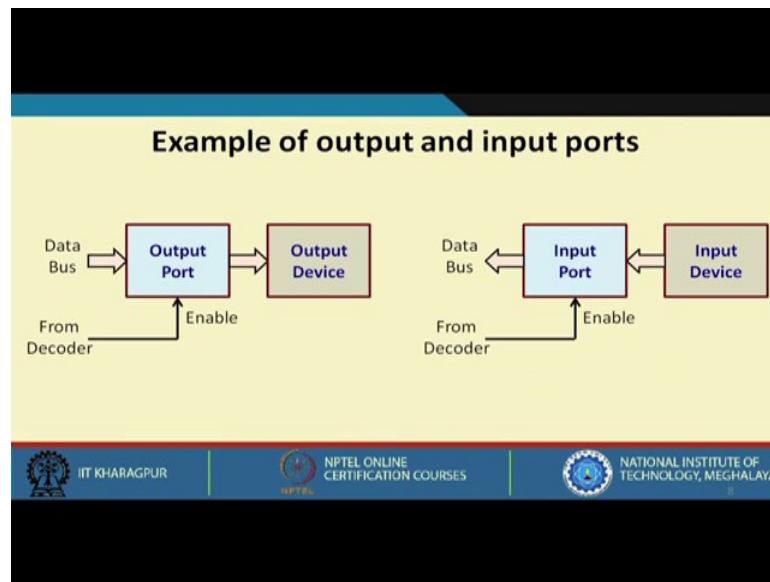
selecting one of these ports at a time. How I am selecting? There will be some kind of an address decoder just like you used an address decoder for selecting memory modules, this is something very similar to that. Some address of an I/O device will be applied, the address decoder output will be selecting one of these I/O ports, these are the enable lines.

One of the I/O port will be enabled and on the other side you have the data bus connection. Roughly speaking an output port looks like this. Data bus is there, this is your I/O address, the address of the I/O device. The address decoder decodes the address, it will be selecting one of the ports, and the port which is selected the data on the data bus will get stored there, and the I/O device will be getting that if it is an output device. So, this is about the output port.

The output port are basically a register that is enabled when the particular output device is given. Now you may ask why you need a register ? We need a register because the I/O devices are much slower. The cpu will be writing something into the port, it will be stored in that register, and the device can take its own sweet time to take the data from that register. But if the CPU just writes it and then withdraws, the device may not be able to read the data within that short time, that is why we need a register to store the data.

On the other hand for input devices you can have input ports. For an input port you do not need a register, you need basically a tristate bus driver. Just a set of drivers with tristate control which will be enabled when a particular input device address is given, using that same kind of an address decoder.

(Refer Slide Time: 14:48)



The driver outputs are connected to the data bus, inputs are connected to the input device. For an output port the diagram I showed this is for one device. There is one output port, which is connected to one output device, and from the address decoder the enable of the output port is activated. For an input port it is similar, but it is not a register it is just a tristate buffer. Input device is sending the data and address, decoder again is decoding, and enabling it. When it is enabling this data will be available on the data bus because it is a buffer.

You see for an input port you do not need a register, a tristate buffer is sufficient. Why? Because the I/O devices are much slower. So, whenever the I/O devices are ready, the input device will be providing the data to the input side of the input port, and whenever CPU wants to read it, it will be enabling the tristate buffer. The data will come on the data bus it will read from the data bus. So, the device need not store it in a register. The buffers are much simpler to build than register, that is why you can save some hardware.

(Refer Slide Time: 16:13)

Memory-Mapped and I/O Mapped Device Interface

- Memory-mapped device interface:
 - The same address decoder selects memory and I/O ports.
 - Some of the memory address space is occupied by I/O devices.
 - All data transfer instructions to/from memory can be used to transfer data to/from I/O devices.
 - The processor need not have separate instructions for I/O, nor it need to specify whether an address generated by the CPU is a memory address or an I/O address.

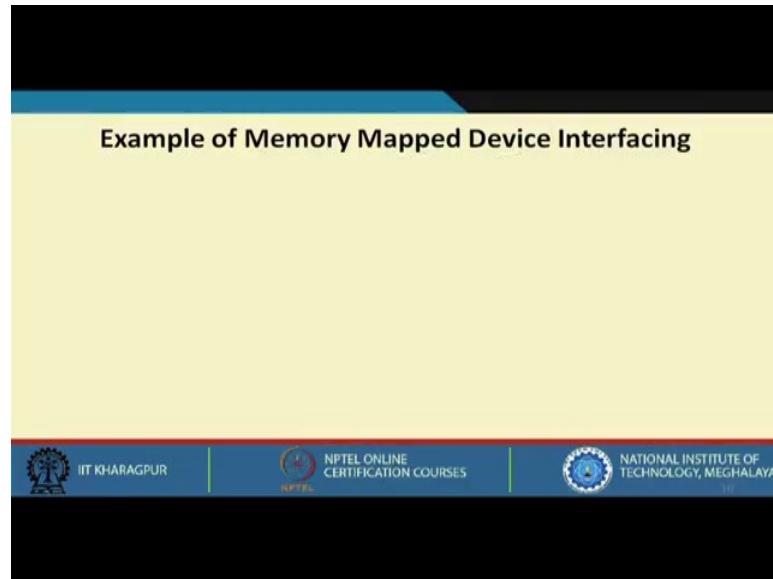
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us look at two alternate ways of connecting the I/O devices to the address and data buses. There is something called memory-mapped device interface, there is something called I/O-mapped device interface. What is memory mapped device interface? Here the same address decoder is selecting both memory and the I/O ports and there is no separate decoder for memory and I O devices. The same single decoder is there and some of the memory address space is eaten up by the I/O devices. I/O devices occupy some of the memory addresses, and you cannot use the full capacity of memory here, some of the memory space is occupied by the I/O devices.

And for data transfer from the input devices or the output devices on the processor, you do not have separate instructions. You treat the I/O devices as memory. For example, in the MIPS32 architecture you can use load and store instructions to load data from the memory or store data to the memory. In a same way if we use a memory address that represents an I/O device, you can load data from that address means I am reading the data, or you can store data into that address means I am writing the data to the device.

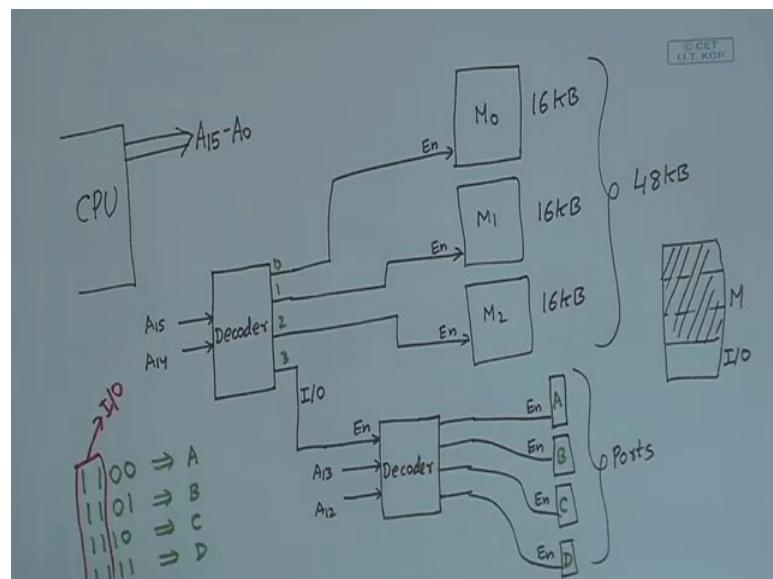
So, reading data from the input device or writing data into the output device you can do using the same load and store instructions. Here as I had said the processor need not have separate instructions for I/O, the processor need not specify whether the address is for memory or I/O, because the decoder will take care of it.

(Refer Slide Time: 18:39)



Let us explain through an example. We shall work out an example.

(Refer Slide Time: 18:43)



Suppose I have some memory modules. These are three memory modules, let us call M0, M1 and M2. Suppose on this side I have the processor which is generating a 16-bit address A15 to A0. So, the total addressable memory will be 2 to the power 16, which means 64 kilobytes.

Let us assume that the memory modules are each 16 kilobytes in size. So, that the total memory that I am connecting is 48 KB. What I am doing is I am using an address

decoder. The address decoder will be fed with the two highest orders bits of the memory address, there will be 4 outputs, the first 3 will be selecting one of these 3 memory modules, this will be the enable.

What we do with the fourth one? What we are saying is that the fourth one we are reserving for I/O, we are saying that in our total memory map if I say this is our total memory map if I divide up into 4 parts, the first three parts we are keeping for memory the remaining quarter here we are reserving for I/O. So, what you do to this I/O? Let us use another decoder here, this is also 2 to 4 decoder.

I am using the next two bits of the address, let us say A13 and A12, and this one I am using to enable this decoder. Decoder also has an enable input, this will enable the decoder. I am having 4 ports for I/O, these are the ports through which I can connect I/O devices, and this decoder will be selecting one of these ports.

You see here when a particular address combination is given, then this will be selected. suppose this is your 0, 1, 2 and 3. So, when the address combination is 3, means A15 A14 should be 1 1, then 3 will be selected let us call the 4 ports as A B C D. If the next 2 bits A13 A12 are 0 0, this means port A is selected; if the first 4 bits are 1 1 0 1 then B is selected, if it is 1 1 1 0 then C is selected, if it is 1 1 1 1 then D is selected.

You see here I am using the same decoding logic which is selecting both memory and the I/O ports, just by looking at the address I can tell whether it is memory or I/O. So, if I see that my address starts with 1 1, then I can immediately say that this is an I/O address. This decoder will take care of this.

This is the example of a memory mapped I/O interfacing, where the I/O devices are considered as an extension of the memory system, and the same instructions that we used to access memory, we can use to access the I/O devices also. Just we need to remember what is the address of my input devices, what is the address of my output devices, I will have to load and store from those addresses.

(Refer Slide Time: 24:16)

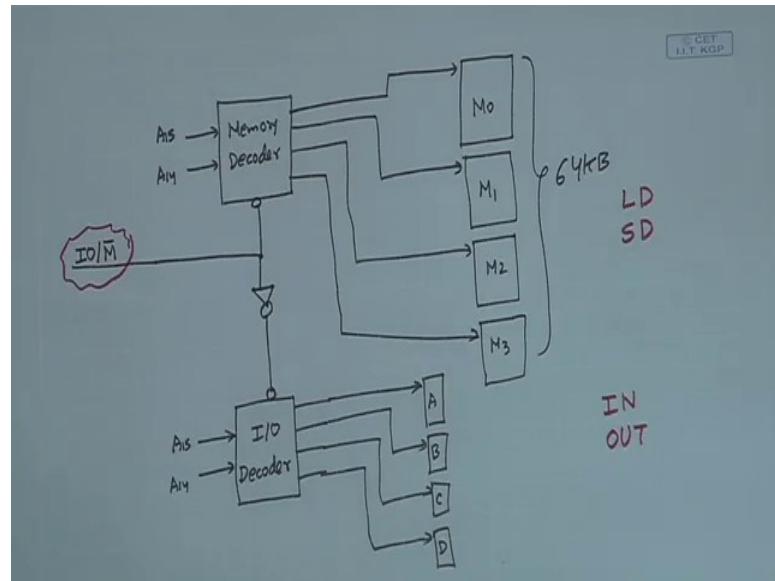
- I/O mapped device interface:
 - Separate instructions for I/O data transfer (say, IN and OUT).
 - A processor signal identifies whether a generated address refers to a memory location or an I/O device.
 - Separate address decoders for selecting memory and I/O ports.
 - The complete memory address space can be utilized.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us look at the I/O mapped device interface now. It is a little different from memory mapped. Here we are saying that there are separate instructions for I/O data transfer, let us say IN and OUT. So, IN instruction will be reading some data from an input port, and OUT instruction will be writing some data into an output port.

And there will be also a processor signal which will clearly identify to the circuitry outside whether the address which is cpu is generating, is a memory address or an I/O address. Here we need to use separate address decoders for memory and I/O. The one advantage is that we can utilize the complete memory address space.

(Refer Slide Time: 25:26)



Let us quickly work out that same example we took earlier. Here we consider that I have 4 memory modules, and here we do not need to compromise; we can connect the whole memory; so 16 kilobytes. So, total I have 64 kilobytes, and I have a memory decoder where I am decoding the 2 highest bits, and this will be selecting one of the memory modules.

I assume that the CPU is generating another signal, let us say IO/M' , this signal means if the signal is 0 then it is memory address, if it is 1 it is I/O address. If it is 0 I will be enabling the memory decoder.

Now, let us see what will happen for I/O. For I/O in the same way we will be having a I/O decoder, this will be enabled when IO/M' is 1. This same signal I connect a NOT gate and connect it to the enable of this. So, here again I can connect A15 and A14 and the 4 outputs that I get, I can select one of the 4 I/O ports A B C and D.

You see here just by looking at the address you cannot say which is memory which is I/O, because same address can refer to a memory or an I/O device also, but the difference is the way you are accessing memory and I/O. When you are accessing memory, let us say you are using load and store instructions, but whenever you are using I/O, you will be using IN and OUT instructions. What these instructions will do? These instructions will activate IO/M' in a proper way; if it is a load or store instruction then it will be 0, if it is an IN or OUT instruction it will be 1.

So, the appropriate decoder will be selected and the corresponding I/O port will get activated. This explains how the I/O mapped device interfacing works. In our next lecture we shall be looking at what are the different ways in which you can actually transfer data from the I/O devices to the processor, and vice versa. There are several techniques which depends on the characteristics of the I/O devices and also the environment what exactly you want to do.

We shall be looking into those methods in some detail. With this we come to the end of this lecture number 44.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 45
Data Transfer Techniques

If we recall in our last lecture we talked about how we can select the IO device interfaces in the form of the IO ports for carrying out input and output for selecting the devices. In this lecture and the next few lectures we shall be talking about in some detail how the actual data transfer takes place between the IO devices and the processor and what are the different variations that are possible.

So, the topic of today's lecture is data transfer techniques.

(Refer Slide Time: 01:03)

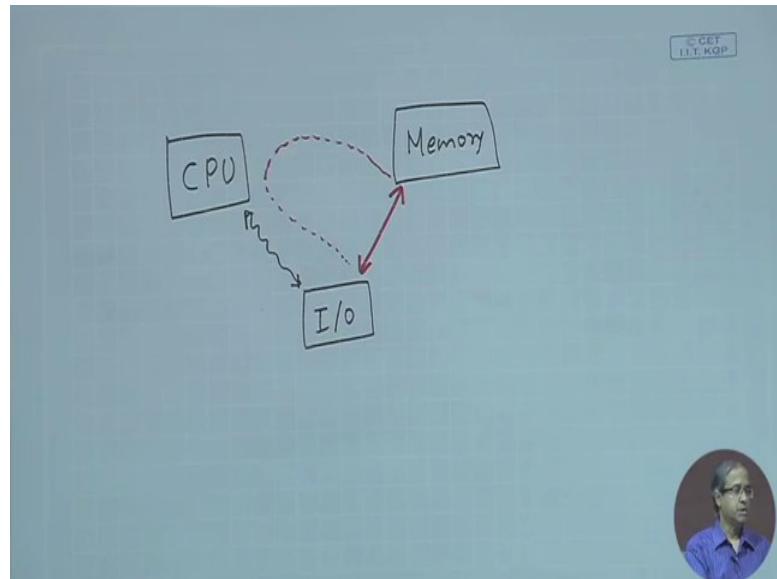
Data Transfer Techniques

1. Programmed: CPU executes a program that transfers data between I/O device and memory.
 - a) Synchronous
 - b) Asynchronous
 - c) Interrupt-driven
2. Direct Memory Access (DMA): An external controller directly transfers data between I/O device and memory without CPU intervention.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Broadly speaking when we talk about data transfer techniques we basically talk about a scenario like this; we have the CPU, we have the memory, and we have the IO devices.

(Refer Slide Time: 01:12)



When we talk about data transfer you may be tempted to think that we are actually thinking about data transfer between CPU and IO, but it is not exactly that. Data transfer on input output transfer actually takes place between IO and memory devices. Because suppose it is an input device. the CPU after reading it will write into memory, and if it is an output device CPU will be reading some data from the memory and write it into that, because CPU is just a mediator, whatever data it is transferring that will be there in memory.

Under the data transfer technique the first broad approach can be called as programmed data transfer. Programmed means as the name implies CPU will be executing a program that transfers data between IO device and memory, meaning suppose it is an input device. Let us say we are using memory mapped kind of device interfacing, then CPU will have to execute a program that will be loading some data from the memory from that device, and it will be storing it into some memory location. Suppose there are 100 data this has to be done in a loop 100 times. So, there will be a program that will be running, and will be actually reading the data from the input device and will be writing the data into the memory.

Under programmed IO again there can be three variations, synchronous, asynchronous or interrupt driven. We shall be looking into the detail of this later. The other alternative is direct memory access or DMA; well here the CPU has very minimal intervention. CPU

does not execute any program, rather there is a separate external controller that will directly transfer data between IO device and memory without disturbing the CPU; CPU may be continuing with whatever it was doing. So, it will be directly transferring the data between the IO device and the memory. This is what DMA is; external controller directly transfers data between IO device and memory without CPU intervention.

(Refer Slide Time: 04:46)

(a) Synchronous Data Transfer

- The I/O device transfers data at a fixed rate that is known to the CPU.
- The CPU initiates the I/O operation and transfers successive bytes/words after giving fixed time delays.
- Characteristics:
 - During the time delay, CPU lies idle.
 - Not many I/O devices have strictly synchronous data transfer characteristics.
- A flowchart for synchronous data transfer from an input device is shown on the next slide.

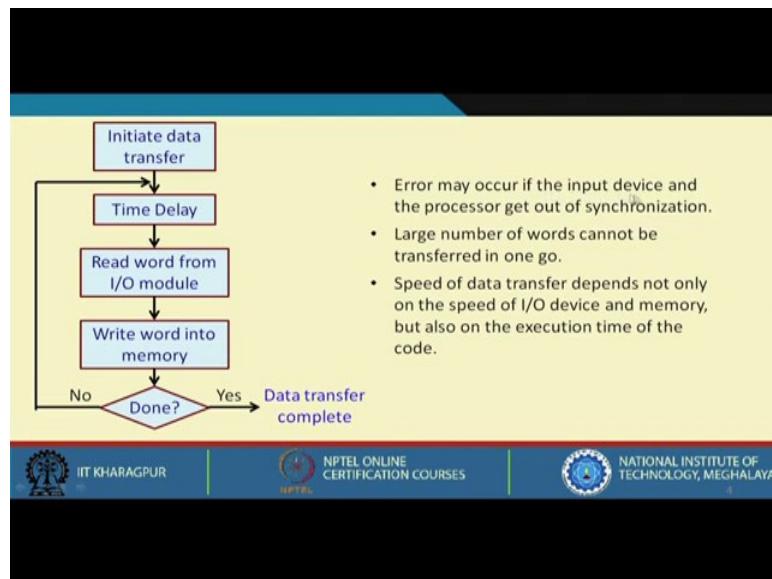
At the bottom of the slide, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

Let us look into these methods one by one. Synchronous data transfer ... as this name implies that there is some kind of a transfer going on, there is a sender, there is a receiver and both of them are synchronized. Synchronized roughly means both of them know what is the speed of data transfer. Suppose I know some data is coming to me, I know what is the speed with which it is coming to me. So, I can pick up the data one by one with that speed.

This is what is mentioned here --- the IO device transfers data at a fixed rate that is also known to the CPU. So, here the CPU will be initiating the IO operation whenever it wants to, and after it does the CPU will know that the data will get transferred at fixed rate. Giving some fixed time delay the CPU can read or write the bytes or words one by one. Now there is some characteristic in this method that while the CPU is waiting with the time delay, it does not have anything else to do. CPU will lie idle; this is one major problem that you are having leading to very poor CPU utilization.

Second thing is that you will find very few IO devices have this kind of synchronous behavior. The data always coming at fixed speed is very rare you know. So, you will not find many devices like this.

(Refer Slide Time: 06:57)



How this works I am showing it in a flowchart form here. The steps are roughly like this. The CPU initiates the data transfer, after initiating CPU will know that now the data will be coming. Suppose it is an input device; data will be coming at a fixed rate. So, CPU will give that amount of time delay, and it will read the next word from the IO device, it will write that word into memory, then it will check whether all the words have been transferred or not. If not it will again go back, again wait for a time delay, and read the next word. This will repeat and if it is done, the CPU will know that data transfer is complete it will continue with the next step.

Now, you see here the CPU is reading the data after giving fixed time delays because it knows the rate. If there is a mismatch in the speed, say due to some problem the IO device is not able to send the data at that fixed rate, then obviously CPU will be starting to read the wrong data because CPU is expecting the data to come every time t , but if the device is a little late, it may be reading the old data.

So, some error may occur if the input device and the processor get out of sync. This is one problem; because of the same reason large number of words cannot be transferred in one go, because chance of the two devices getting out of synchronization will be high.

And third thing is that you see there is a program that is executing. You are reading from some IO module, and writing into memory. If we think of the speed of data transfer, what is the maximum speed you can achieve? The maximum speed of data transfer does not depend only on the speed of IO device and memory, but also in the execution time of this code. This is true for all the programmed I/O techniques; the basic code where you are reading a word from I/O module and writing into memory, this will take some time and this will limit the maximum data transfer rate.

(Refer Slide Time: 09:39)

(b) Asynchronous Data Transfer

- The CPU does not know when the I/O module will be ready to transfer the next word.
- CPU has to check the status of the I/O module to know when the device is ready to transfer the next word.
 - Called *handshaking*.
- Characteristics:
 - While the CPU is checking whether the I/O module is ready, it cannot do anything else.
 - Wasteful of CPU time for slow devices like keyboard or mouse.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us move on to the next method asynchronous data transfer that is more practical, because you will not find many devices in real life that are strictly synchronous in nature. So, in asynchronous what are the characteristics?

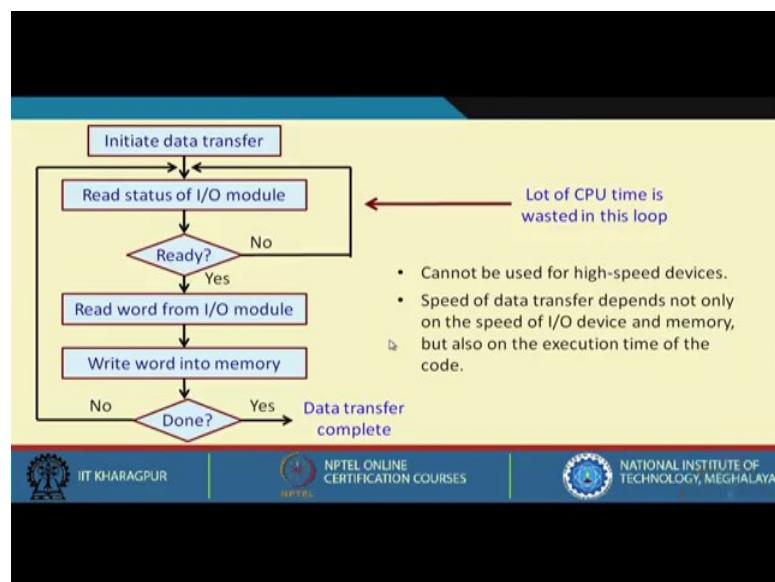
Here it is assumed that the CPU does not know or need not know when the IO module will be sending the next data. So, CPU does not know beforehand when the next data will be coming; next millisecond or may be after one hour. So, the CPU has to continuously check the status of the IO device whether it is ready. It is continuously asking the device are you ready are you ready are you ready, and as soon as the IO device says yes I am ready, the data transfer takes place.

You can see the CPU is wasting its time just asking the status of the device; are you ready are you ready. During this process the CPU is doing nothing useful, it is simply checking the status. CPU has to check the status of the IO module because CPU does not

know when the next data is coming; it will continuously have to check the status of the IO module and this process is called handshaking. So, CPU and the IO module are doing some kind of handshaking by virtue of which it comes to know when the device is ready to transfer.

While the CPU is checking the status of the IO module, it cannot do anything else it is just waiting. This is obviously wasteful of CPU time, particularly for slow devices like keyboard or mouse, because for the entire period this CPU is waiting to check the status.

(Refer Slide Time: 12:01)



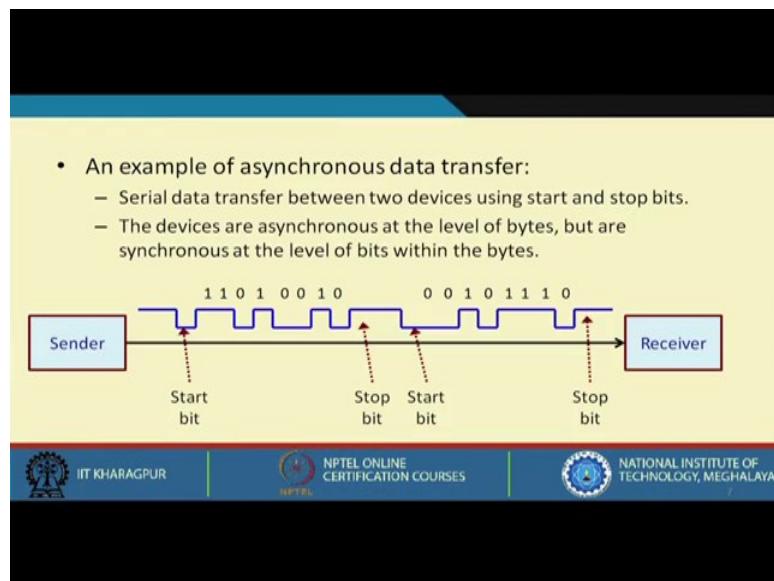
In terms of a flowchart the asynchronous data transfer works as follows. CPU will initiate data transfer, then CPU will read the status of the IO module to check whether it is ready or not; if it is not ready it will go on repeating this, it will continually check the status until it is ready. Now, as soon as it is ready it can read the word from the IO module and write the word in memory; the data transfer can be done and then it checks whether all the data have been transferred, if not it again goes back and starts this process, and if it is yes the transfer is complete and it proceeds with the next step.

Now, in this loop lot of CPU time is wasted because during this time the CPU is not doing anything useful just checking the status of the IO module whether it is ready. Clearly this method cannot be used for high speed devices and just like the previous method the speed of transfer depends on the execution time of the code and not only on the speed of IO device and memory.

So, synchronous and asynchronous data transfer methods we have seen they are quite similar, in one case the CPU does not wait for the IO device because CPU knows when the next data will be available. Suppose you take an analogy. I have asked you to do some work I know that you will take 10 minutes. I just come after ten minutes and take it from you, I do not ask you anything. And in asynchronous mode, I give you a task, and I continuously come every ten seconds and ask you whether you have done, you have done, you have done. I am continually engaged here; this is the main difference between these two.

Now let us move to something that is more practical, but before that let us see some examples of this asynchronous method. This asynchronous method or handshaking that we have talked about, this can be used not only for data transfer between CPU and some IO device, but also for data communication purposes.

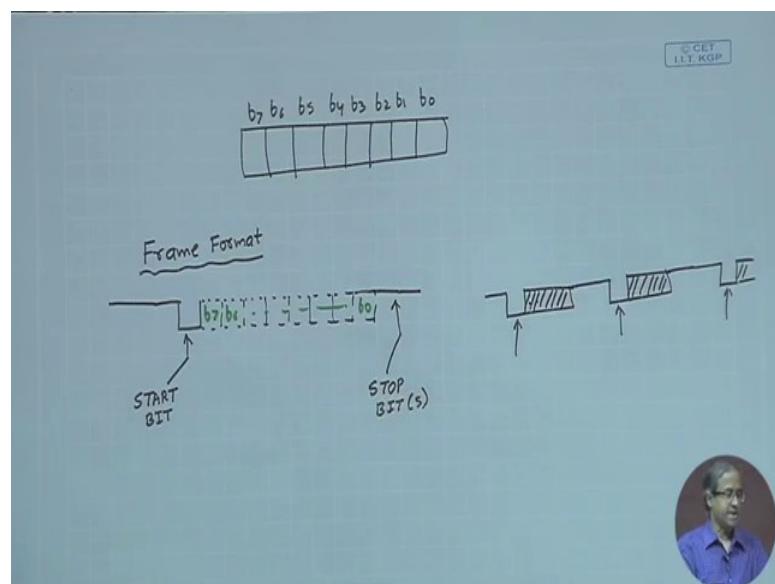
(Refer Slide Time: 15:00)



This example that we are showing is an example of asynchronous data transfer. Let me explain what is the meaning. There is a sender, there is a receiver --- sender will be sending a sequence of bytes to the receiver sequentially. We are assuming that the devices are asynchronous at the level of bytes, meaning the receiver does not know when the next byte will come; it can come immediately, it can come after five minutes, it can come after 10 minutes.

So, it is asynchronous at the level of the bytes, but once a byte is coming the bits within the bytes are synchronous. Because once a byte has started to come, within that byte the bits are coming with fixed delays that are synchronous. This is a very realistic assumption in communication systems.

(Refer Slide Time: 16:23)



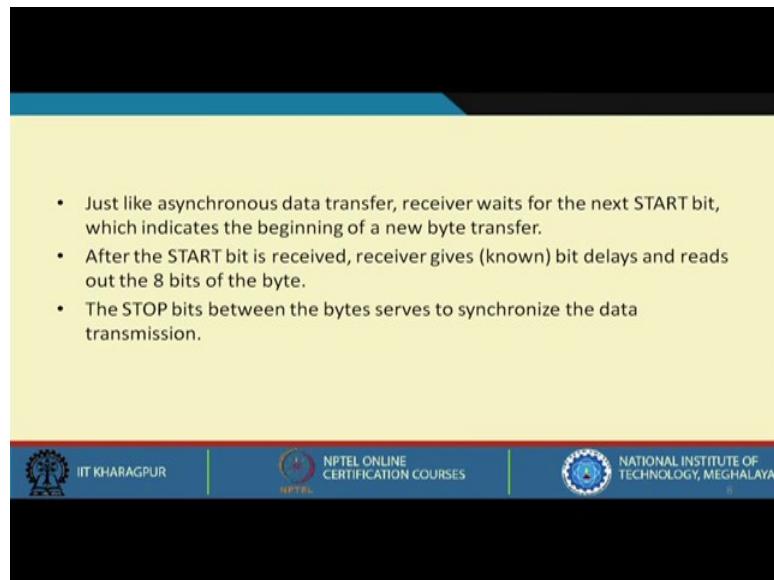
Now, let us see how this data transfer is taking place. Suppose I have to transmit a byte; there are 8 bits, let us call them b7, b6, b5, b4, b3, b2, b1, b0. Now we define something called a frame format. What is a frame format? What we are saying is that this signal that we are transmitting will go zero at the beginning, and this zero is called the start bit. So, whenever this signal goes zero, the receiver will know that a new byte is coming. Once it has done this,, the eight bits can be transmitted one after the other. So, here the 8 bits are transmitting let us say b7 first, then b6 ... up to b0, and at the end I am leaving the line high for 1 or 2 cycles, I am calling this as stop bit, or stop bits.

When I am transmitting a number of such bytes there will be a start bit, there will be a stop bit. How will it look like? It will begin with a start bit, then the bits of the byte will come, then the stop bit, then again a start bit will come then again the byte then again the stop bit, and so on. The receiver will always be looking for the start bits again. So, once a start bit is found, it will know that immediately the next 8 bits of the data bits are coming in fixed intervals of time. So, after some fixed delays it can read them out.

Exactly this thing is shown in the slide. The sender is transmitting two bytes one by one.

The start bit is a way of indicating that a new byte is coming. So, the receiver continuously checks whether the start bit is coming. Once it is found that it will simply read the next 8 bits of the byte. This is a very classical example where asynchronous IO is used.

(Refer Slide Time: 20:41)



This is what I have just said. The receiver will wait for the next start bit that will indicate the beginning of the next byte transfer and once the start bits is received, the receiver knows the data rate, it will give appropriate bit delays and read the 8 bits. And the stop bits between the two bytes serve the purpose of synchronization, because if you do not give the stop bits then one byte and the next byte might be joined together and the receiver might get confused. So, it is always good to give some gap in between, and again resynchronize at the next start bit. So, sender and receiver will get resynchronized.

(Refer Slide Time: 21:32)

(c) Interrupt-Driven Data Transfer

- The CPU initiates the data transfer and proceeds to perform *some other task*.
- When the I/O module is ready for data transfer, it informs the CPU by activating a signal (called *interrupt request*).
- The CPU suspends the task it was doing, services the request (that is, carries out the data transfer), and returns back to the task it was doing.
- Characteristics:
 - CPU time is not wasted while checking the status of the I/O module.
 - CPU time is required only during data transfer, plus some overheads for transferring and returning control.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

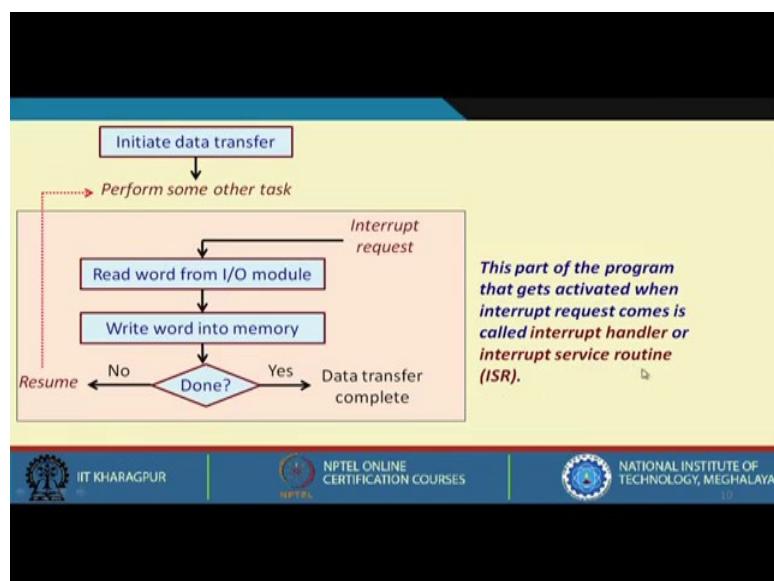
The third and most important kind of data transfer technique that we discuss is called interrupt driven data transfer. Let me first tell you the analogy. Say I come and give you a task, I ask you do this, but I am not asking you continuously that whether you have finished whether you have finished. I give you a task, I go back and I start doing my own work; something else I am doing. You finish your task and as soon as you are done, you come to me and tell me that you have finished. Whatever I was doing I will keep it, I will see what you have done, then I will again come and resume whatever I was doing. This is the basic idea behind interrupt driven data transfer or interrupts in general.

Whatever I have said in the terminology of computers it is like this. The CPU initiates the data transfer and proceeds to perform some other task. It is not sitting idle, it is doing some other task. When the IO module is ready for data transfer, it informs the CPU by activating a signal called interrupt request; like someone is coming to the CPU and telling that well I am ready.

When the interrupt request comes, the CPU that was doing some other task will suspend that task temporarily, service the request -- whatever interrupt request is coming. In this case it is for a data transfer, so it will do the data transfer and it will return back to that some other task it was doing. It is similar to a subroutine or a function call and return; whenever interrupt comes I go somewhere, I finish it and then again I come back.

Some of the characteristics here are that the CPU time is not wasted while checking the status of the IO module, because we need not check the status. The device itself will come and tell me that I have finished. CPU time is required only to carry out the data transfer plus of course, some overhead for transferring and returning of the controls. Stopping the process, going there and again coming back this extra overhead is there, but in addition to that the CPU time can be utilized in a much better way. CPU can do something more fruitful, some other task.

(Refer Slide Time: 24:36)



Pictorially let us see how it looks like CPU initiates the data transfer and after initiating the data transfer CPU is performing some other task. The device when it is ready, the IO interface will send an interrupt request to the CPU, interrupt request will tell the CPU that I am finished. Then CPU will temporarily suspend whatever it was doing, it will read word from IO module, write or do the data transfer, check whether all the words have been transferred, if it is yes done - if no it will resume this other task whatever it was doing and it will be waiting for the next interrupt request to come. Next interrupt request means another data transfer fine.

The part of the code that gets activated whenever interrupt request comes, in this case a simple data transfer, is called interrupt handler or interrupt service routine (ISR). Here since we are more concerned about IO transfer, we are talking about interrupts for IO, but interrupt is a general concept. Interrupts can come for other reasons also, like

interrupt may be coming from a timer in a time sharing operating system, interrupt may be coming from internal to the processor like there is a division by zero, or there is a power failure; the power is failing there is a sensor; there are so many sources of interrupts.

Depending on that exactly what you do inside your interrupt handler or ISR may be different, but here because we are concerned only for IO transfers. So, the ISR's sole tasks will be to transfer the data.

(Refer Slide Time: 27:12)

Some Features of Interrupt-Driven Data Transfer

- How is ISS different from a normal subroutine or function?
 - A function is called from well-defined places in the calling program.
 - Only the relevant registers need to be saved on entry to the function, and restored before return.
 - The ISS can get invoked from *anywhere* in the program that was executing.
 - Depends on when the interrupt request signal arrived.
 - So potentially all the registers that are used in the ISS needs to be saved and restored.

Some features of interrupt driven data transfer are as follows. There is some similarity with function or a subroutine call; when an interrupt request comes you stop what you are doing, you go somewhere else, and then again come back. In a function call also you do something like that. You go to a function, do it and come back, but there is one difference. For a function call you actually make a function call from somewhere in the program. You know that in a program from this place are making a function call. But here you do not know when the interrupt request will come; it can come here, it can come here, it can come here. So, while your program is executing your interrupt request can come potentially anywhere. Wherever it comes you will have to stop, you will have to go there, finish it and then again come back.

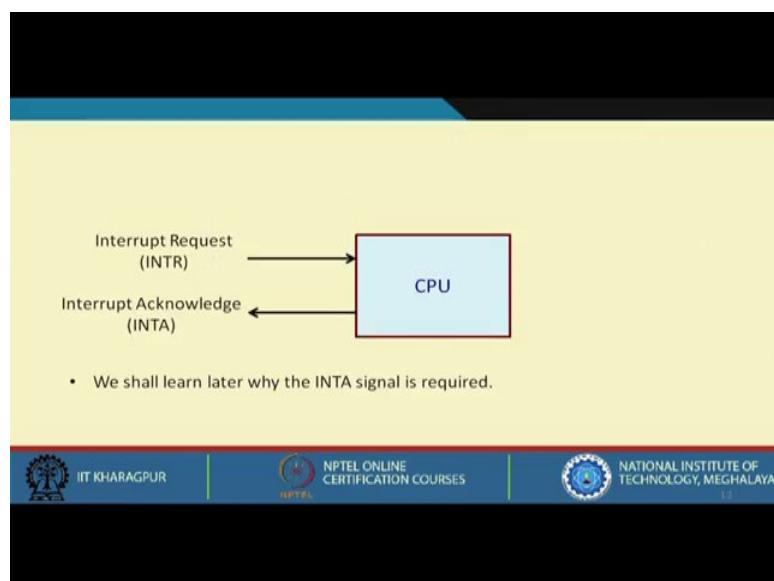
Just as I said for a function it is called from well defined places in the calling program that is actually coded and only the relevant registers need to be saved. Whatever registers

you are using you need to save only those registers and return, because the function might be destroying some of the registers your main program or the calling program was using. So, you should make sure that those registers are not destroyed.

But in contrast the ISS or ISR can get invoked from anywhere in the program that was executing --- this anywhere means it depends on exactly where the interrupt signal arrived. So, potentially all the registers that are used in the service routine needs to be saved and restored.

What I mean to say is that because you do not know where in the program your interrupt might come, and in the program you may be requiring a large number of registers in different parts, so potentially you may have to save and restore all those registers. The overhead of saving and restoring registers in interrupt can be much higher.

(Refer Slide Time: 29:51)



Pictorially it looks like this. This is a processor, the interrupt request comes from some device or some external source. Processor generates an interrupt acknowledge and why this is required we shall be discussing later, because so far we have not talked about interrupt acknowledge. Now there are some challenges in interrupts, and these we shall be dealing with in some detail in our subsequent lectures.

(Refer Slide Time: 30:22)

The slide has a dark blue header and footer. The main content area is yellow. The title 'Some Challenges in Interrupts' is centered at the top of the yellow section. Below the title is a bulleted list of challenges:

- For multiple sources of interrupts, how to know the address of the ISR?
- How to handle multiple interrupts?
 - While an interrupt request is being processed, another interrupt request might come.
 - Enabling, disabling and masking of interrupts.
- How to handle simultaneously arriving interrupts?
- Sources of interrupts other than I/O devices.
 - Exceptions, TRAP, etc.

At the bottom of the slide, there are three logos with their respective names: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

First thing is that if there are multiple devices interrupting, how to know who has interrupted and how to know what will be the address of the interrupt service routine. Because the service routine for the disk may be different from the service routine of the keyboard, which may be different from the service routine for the mouse, so I have to know which device has interrupted and where to go and how to know this.

Secondly how to handle multiple interrupts. What I am trying to say is that while an interrupt request is being processed let us say another interrupt request has come, so what do you do?

There are many alternatives we shall be discussing. This may require enabling, disabling or masking of the interrupt system. Next there can be simultaneously arriving interrupts, so how to handle. Here you have to do something regarding prioritization of the interrupts; you have to define interrupt priorities. And lastly there can be interrupts that can come from some other device, not only IO devices, like exceptions, etc.

With this we come to the end of this lecture. In our next lecture we shall be continuing with our discussion on interrupt handling and interrupt processing. We will look into the different details exactly what are the issues and what are the possible solutions we can use there.

Thank you.

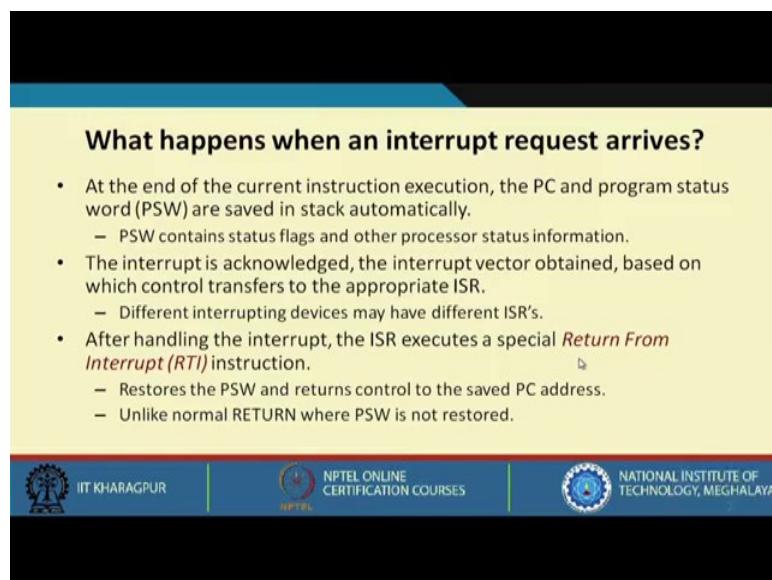
Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 46
Interrupt Handling (Part I)

In our last lecture we were talking about the various IO transfer techniques specifically under the programmed IO category. If you recall we looked at synchronous method of data transmission, then you looked at asynchronous or handshaking, and finally we were discussing the concept of interrupt driven data transfer.

In case of interrupt driven data transfer what you saw was the processor is doing something, whenever the IO device is ready to transfer some data it will be interrupting the CPU by sending a interrupt request signal. The CPU when it receives the interrupt request signal will know that some IO device is now ready to transfer data, so it will be jumping to some ISR where the data transfer will take place, and after that it will again return back to the program which it was executing. So, here while the device is getting ready for transfer, the CPU is able to do something else; CPU is not getting tied up.

(Refer Slide Time: 01:48)



What happens when an interrupt request arrives?

- At the end of the current instruction execution, the PC and program status word (PSW) are saved in stack automatically.
 - PSW contains status flags and other processor status information.
- The interrupt is acknowledged, the interrupt vector obtained, based on which control transfers to the appropriate ISR.
 - Different interrupting devices may have different ISR's.
- After handling the interrupt, the ISR executes a special *Return From Interrupt (RTI)* instruction.
 - Restores the PSW and returns control to the saved PC address.
 - Unlike normal RETURN where PSW is not restored.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

We shall be continuing our discussion on interrupt driven data transfer, in particular the topic of interrupt handling.

Let us see what exactly happens when an interrupt request reaches the CPU. The first thing that happens is that the CPU was executing some instruction when the interrupt request came. We shall be seeing that some exceptions can be there, but typically the instruction which was being executed will first complete and only after that the interrupt request will be acknowledged.

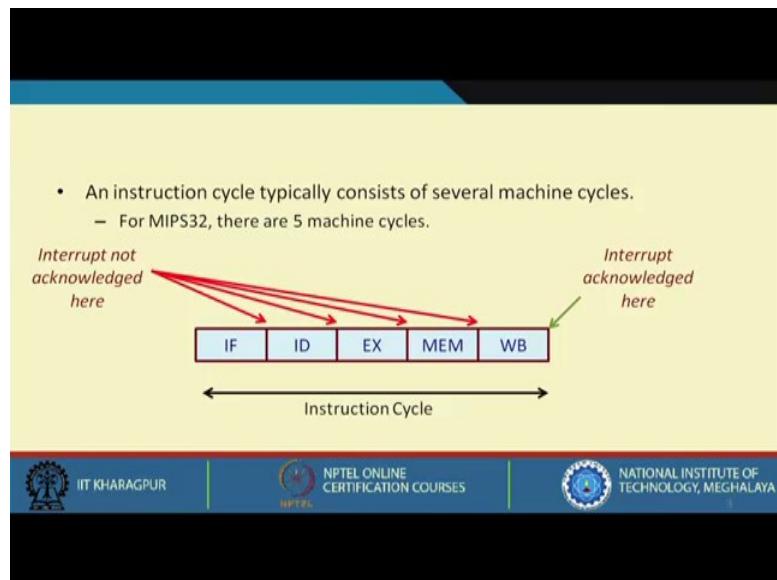
So, what we are saying is that at the end of the current instruction execution, during the interrupt acknowledge cycle the program counter and the program status word or PSW are saved in a stack. You may recall that for normal subroutine or function call and return only the return address; that means, the value of the PC is saved in stack, but here we are saving not only the PC, but also some status information, which depends on the processor, the ISA.

Typically in computer systems there are status flags. PSW will contain status flags and some other processor status information regarding which level of privilege it is working on, and so on and so forth. Now you may recall in MIPS32 processor we do not have any such status flags. So, saving of the PSW becomes much simpler there.

After this is done the interrupt is acknowledged, which means the interrupt acknowledge signal is activated and the external interrupting device or the IO module can supply the interrupt vector. The interrupt vector will identify the device that has interrupted; using that information you can compute the address of the interrupt service routine and transfer control to that.

Different devices may be having different service routines. You need to get the address of the correct service routine and then jump to that. Now after the interrupt handling is completed the interrupt service routine executes a return, but this is a special kind of a return instruction not a normal return instruction that you use to return from a subroutine or a function where only typically the PC is popped from the stack and is loaded into PC. But here you have to restore not only the value of the PC, but also the program status word which was also saved. So, this is a special kind of instruction, let us say RTI. This will restore the PSW and return control to the saved PC address. As I had said this is slightly different from the normal return instruction where PSW is not saved and returned, only the PC is restored.

(Refer Slide Time: 05:23)



Now, let us look at an instruction cycle. In MIPS32 you recall the instruction cycle consist of five steps, IF ID EX MEM and WB. These are typically called machine cycles. An instruction cycle comprises of typically more than one machine cycles. These machine cycles are executed in order to complete the execution of an instruction.

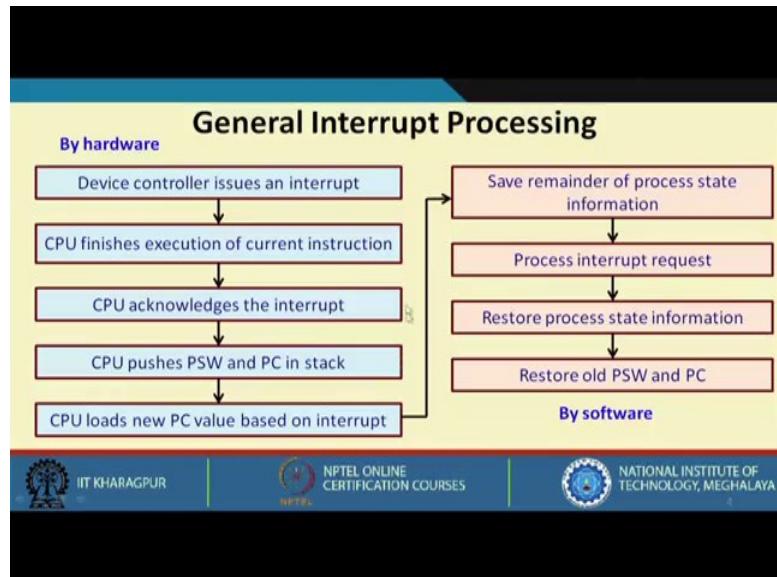
When the instruction execution is complete only then interrupts are acknowledged, but in between the machine cycles interrupts are not acknowledged even if the interrupt may appear earlier. But the processor will wait till the instruction execution is complete, and only then the interrupt system will get acknowledged.

Now, in this context you may recall one thing that we talked about the machine cycles and instruction cycle, and said that interrupts will be acknowledged only at the end of the instruction cycle not in between, but there is another method of IO transfer which we have not discussed yet that is direct memory access.

DMA is one method using which the IO device can directly transfer some data to memory or back without intervention of the CPU. Now here there is slight difference in terms of the handling of the DMA requests. Whenever a device wants to make such DMA transfer, we can stop the processor not only at the end of the instruction cycle, but in between also. At the end of a machine cycle we can stop, at the end of IF we can stop, at the end of ID or EX or MEM we can stop. This is possible because while the transfer

is going on the CPU status is not changing; it is just like a pause. While the transfer is going on CPU will pause, and then it will again continue.

(Refer Slide Time: 08:08)



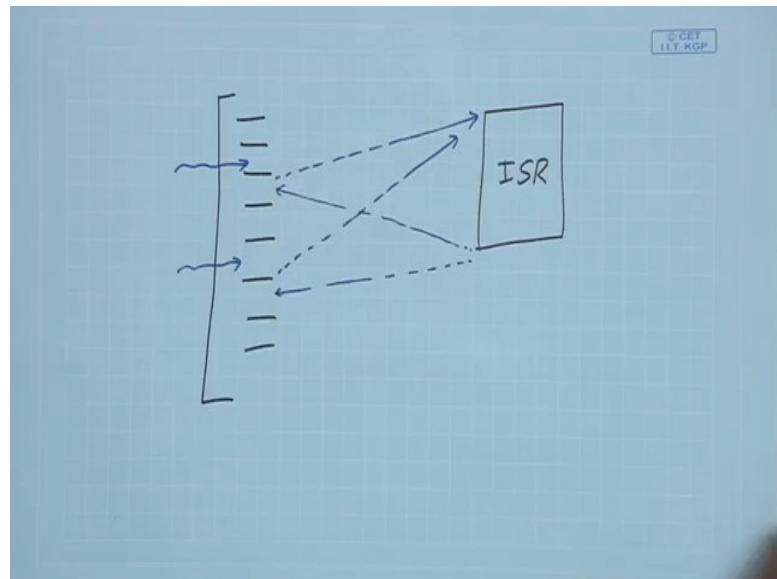
The general interrupt processing can be summarized by this flow diagram. While the portion marked in blue are performed in the hardware unit in the CPU, while the pink part are performed by software as part of the interrupt service routine. Let us see what happens. The device controller to which the IO device is connected issues an interrupt request. CPU receives that, CPU will finish the execution of the current instruction, and only after that the CPU will save the PC and the PSW, it is after that it will acknowledge the interrupt and it will save PSW and PC in the stack.

After acknowledging the interrupt as the CPU will try to know that which device has interrupted. Here the interrupt vector concept comes in, whenever interrupt acknowledge is coming the external device controller will be pushing some kind of a interrupt vector or a device ID on the data bus. CPU can read it and can identify the device.

Now, depending on the device CPU will be jumping to the appropriate interrupt service routine, which means it will loading the PC with the address of the corresponding ISR. After this is done control will jump to the ISR; this part is done by the CPU automatically whenever the interrupt comes, but here when ISR assumes control there is a program code that is written. It will be saving some other processor information, if required may be some registers it will be saving, and it will be processing the request, if

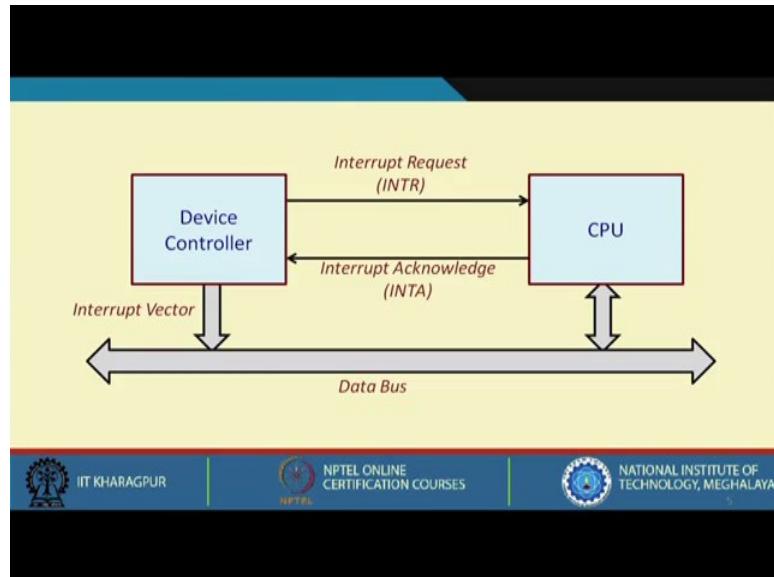
it is data transfer it will be actually carrying out the transfer, and whatever registers were saved will restore them back, and it will return back by restoring the old PSW and PC.

(Refer Slide Time: 10:58)



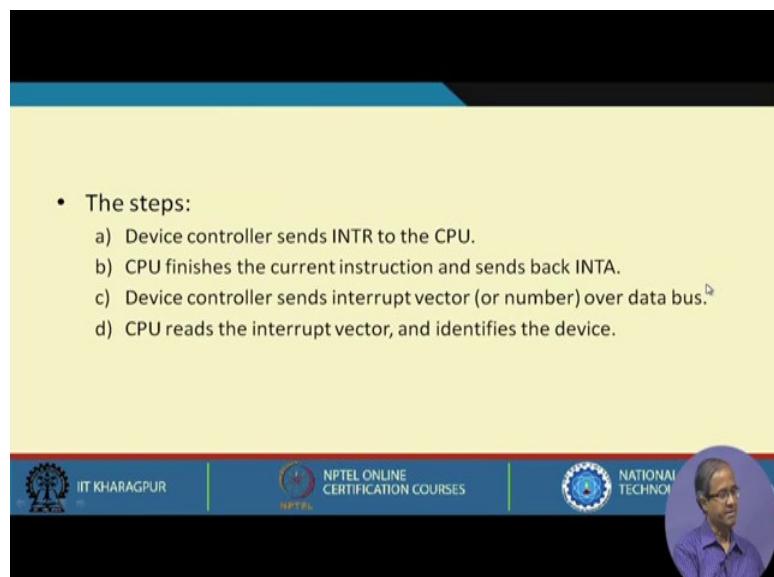
Now the point to note is that suppose this was a program which was executing. There are many instructions and let us say this is my ISR. Now you cannot predict before exactly where the interrupt will come; maybe the interrupt will arrive here, then after the execution of this instruction you will be jumping to the ISR. It will finish and then it will return back. Maybe the interrupt has come here, then after execution of this control will jump it will get executed and it will be returning here.

(Refer Slide Time: 11:49)



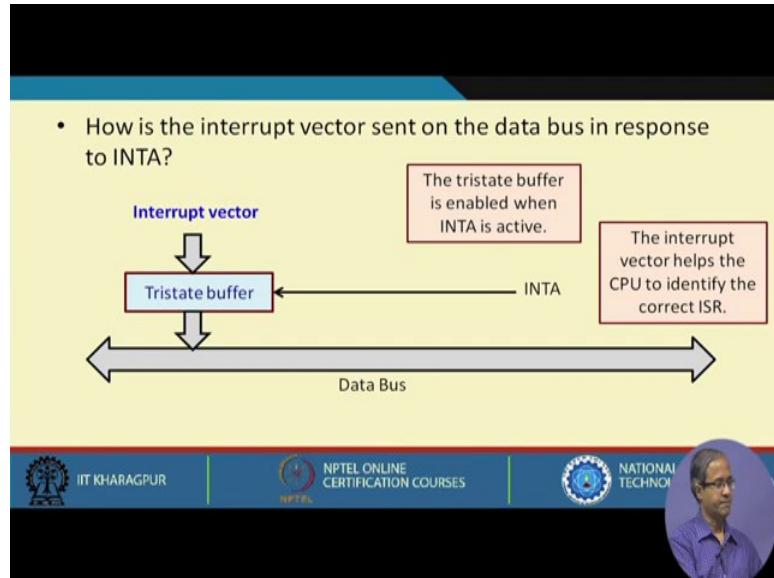
So, it can actually jump from anywhere in the program depending on where the interrupt request has actually come. This is the picture that shows the CPU, the device controller and the bus. The device controller will be sending an interrupt request to the CPU, CPU at the end of the current instruction execution will be sending back an interrupt acknowledge.

(Refer Slide Time: 12:25)



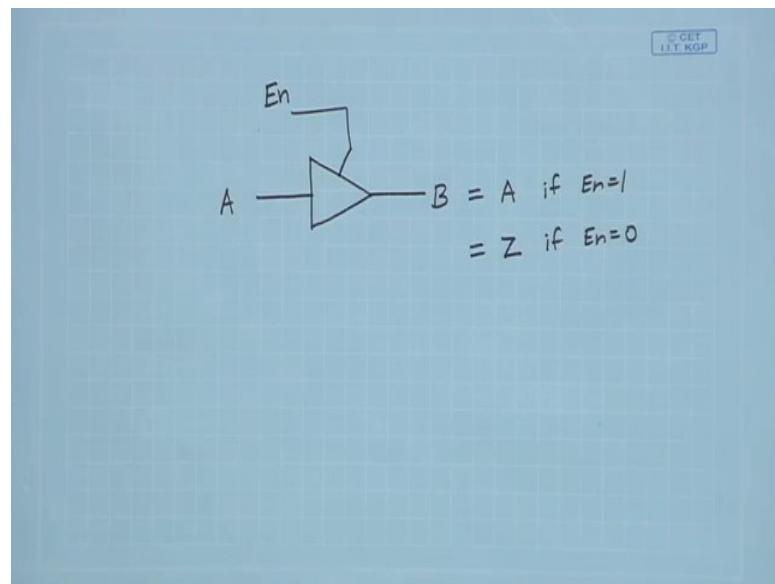
In response to that, device controller will be pushing an interrupt vector on the data bus. CPU will be reading that interrupt vector and will be able to identify which device has interrupted.

(Refer Slide Time: 12:55)



Now, the question is how is the interrupt vector send to the data bus in response to INTA? We have said that whenever this INTA signal is coming, the device will be putting the interrupt vector on the data bus, but how does it happen automatically? It is very simple. Here actually what we require is the tristate buffer. On the input side of the tristate buffer we have the interrupt vector, the output of which is connected to the data bus and the tristate buffer is enabled by the interrupt acknowledgement signal.

(Refer Slide Time: 13:46)



You recall what is a tristate buffer. A tristate buffer is a circuit which has an enable. Suppose I have an input A and the output B. If this buffer is enabled, then B will be equal to A, but if it is not enabled then the output will be electrically disconnected or in the high impedance state. So, this tristate buffer can either connect or disconnect a signal from a destination point; in this case the destination point is the data bus.

Whenever INTA is active the interrupt vector will appear on the data bus. When it is not active, tristate buffer will be making the output lines in high impedance state. The interrupt vector which is pushed on the data bus can be read by the CPU. This will help the CPU to identify which device had interrupted, and accordingly it can identify the correct interrupt service routine.

(Refer Slide Time: 15:20)

Multiple Devices Interrupting the CPU

- A common solution is to use a priority interrupt controller.
 - The interrupt controller interacts with CPU on one side and multiple devices on the other side.
 - For simultaneous interrupt requests, interrupt priority is defined.
 - The interrupt controller is responsible for sending the interrupt vector to CPU.

The diagram illustrates the architecture for handling multiple interrupt requests from various devices. On the left, a blue box labeled 'CPU' has two lines extending from it: a red line labeled 'INTR' pointing to the right, and a yellow line labeled 'INTA' pointing to the left. In the center, a blue box labeled 'Priority Interrupt Controller' receives the 'INTR' line and sends the 'INTA' line back to the CPU. From the right side of the PIC, four parallel lines extend to the right, each labeled with a pair of interrupt codes: 'INTR0 and INTAO', 'INTR1 and INTA1', 'INTR2 and INTA2', and 'INTR3 and INTA3'. This indicates that the PIC can handle up to four simultaneous interrupt requests by prioritizing them based on their arrival sequence and sending the appropriate interrupt vector to the CPU.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us consider the scenario where multiple devices can interrupt the CPU. Let us look at a scenario like this. Suppose there are several devices connected on this side. Each of these devices will be having an interrupt request and interrupt acknowledge; let us say there are 4 such sets.

When there are multiple devices that needs to interrupt the CPU, one common solution is to have a device which is called a priority interrupt controller. A priority interrupt controller works exactly like shown in this diagram. On one side it will be having multiple sets of interrupt request and acknowledge lines through which various device controllers can be connected, but on this side of the CPU there is a single interrupt line and a single interrupt acknowledge line. The interrupt controller interacts with CPU on one side and multiple devices on the other side.

Here the idea is whenever any one of the devices will be sending an interrupt request, the priority interrupt controller will automatically generate an interrupt request to the CPU, and when the CPU sends back the acknowledgement the interrupt controller knows that which device has sent interrupt. Whenever acknowledgment comes, this interrupt controller will be putting the appropriate interrupt vector on the data bus corresponding to that interrupting device.

So, interrupt controller is responsible for sending the correct interrupt vector to the CPU, but just one thing. If suppose two or more interrupt lines are activated simultaneously

then what will happen? Then this priority comes into the picture. Let us say you define a priority where this line 0 has higher priority than 1, 1 has higher priority than 2, and 2 has higher priority than 3. So, if interrupt request 0 and interrupt request 2 are activated simultaneously, then this interrupt request 0 will be processed and 2 will be ignored for the time being.

Later on, after finishing the processing of interrupt request 0, if the interrupt request 2 is still active it will be processed and handled. This is a typical way using which multiple interrupts can be handled.

(Refer Slide Time: 18:34)

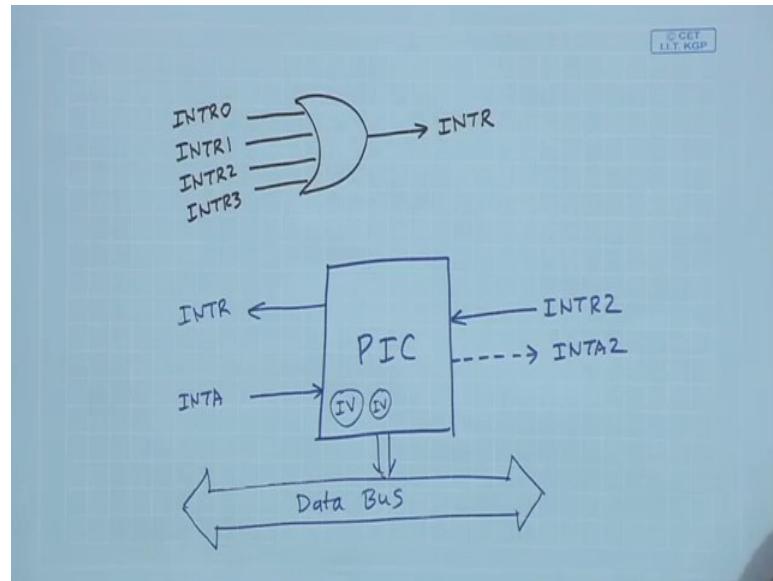
The slide has a yellow background and a black header bar. The title 'How it works?' is at the top left. Below it is a bulleted list of points about how interrupt requests work. In the bottom right corner, there is a circular inset showing a man with glasses and a purple shirt, likely the speaker.

- How it works?
 - The INTR line is made active when some of the device(s) activate their interrupt request line.
$$INTR = INTRO + INTR1 + INTR2 + INTR3$$
 - When the CPU sends back INTR, the interrupt controller sends back the corresponding acknowledge to the interrupting device, and puts the interrupt vector on the data bus.
 - The interrupt controller is programmable, where the interrupt vectors for the various interrupts can be programmed (specified).
 - For more than one interrupt request simultaneously active, a priority mechanism is used (e.g. INTRO is highest priority, followed by INTR1, etc.).

At the bottom, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Technology.

The interrupt line as I had said is activated when some of the devices activate their interrupt request line.

(Refer Slide Time: 18:55)



How it can be implemented? You use a simple 4 input OR gate, where the 4 interrupt request lines are connected to the 4 inputs and the output will be generating the consolidated interrupt request signal to the CPU. Then the CPU sends back interrupt acknowledge INTA. When the CPU sends back INTA, the interrupt controller will send back the corresponding acknowledgement to the interrupting device and puts the interrupt vector on the data bus.

Suppose I have the priority interrupt controller here. Let us suppose device number 2 has interrupted. In response the CPU got this INTR. Later on when the CPU generates the acknowledgement, the PIC will be generating the corresponding acknowledgement for this device 2, INTA 2, this will be done automatically. It also has another responsibility; it is also connected to the data bus. Whenever this interrupt acknowledge is sent by the CPU, the corresponding interrupt vector for this particular device will be pushed on the data bus, so that this CPU can read the value and know which of the devices had actually interrupted. This is the second step and the third point to note is that the interrupt controller can be storing the interrupt vectors for the different devices.

The interrupt controller is in some sense programmable, programmable means before you are using it you can specify that these will be the interrupt vectors for the 4 devices. The CPU can initialize some internal registers within that PIC, where the values of the

interrupt vectors can be programmed or specified. If more than one requests are activated simultaneously we can use a priority mechanism.

Let us say by convention interrupt 0 will be having the highest priority followed by interrupt 1 followed by interrupt 2 and then interrupt 3 .

(Refer Slide Time: 22:14)

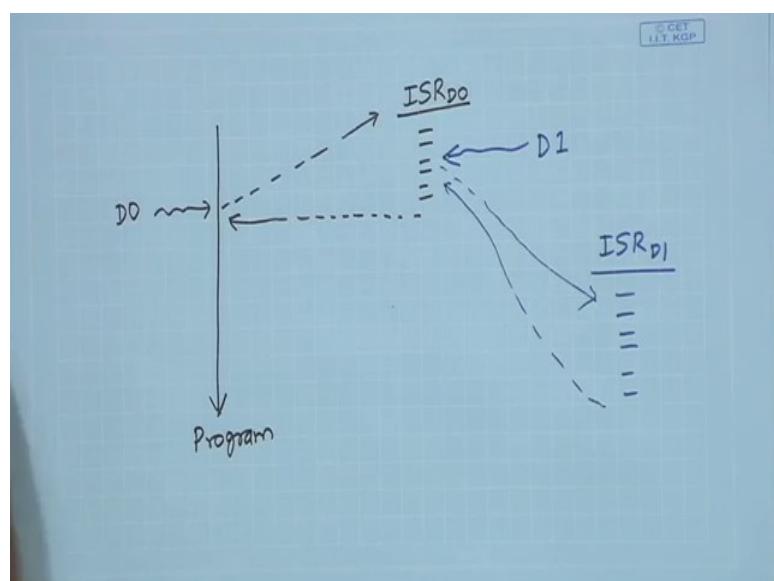
How is interrupt nesting handled?

- Consider the scenario:
 - a) A device D0 has interrupted and the CPU is executing the ISR for D0.
 - b) In the mean time, another device D1 has interrupted.
- Two possible scenarios here:
 - D1 will interrupt the ISR for D0, get processed first, and then the ISR for D0 will be resumed. → *CREATES PROBLEM FOR MULTI NESTING*
 - Disable the interrupt system automatically whenever an interrupt is acknowledged so that handling of nested interrupts is not required.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Another issue is nesting of interrupts. Let us consider a scenario where a device D0 had sent an interrupt to the CPU, and the CPU is currently executing the corresponding ISR. While this ISR is being executed, let us assume that another device D1 has interrupted.

(Refer Slide Time: 22:47)



What I am saying is that some program was executing in between there was an interrupt from device D0. Because of this interrupt the control had transferred to the ISR corresponding to device D0. After finishing it is supposed to come back like this and resume execution.

But what we assuming is that while this ISR D0 was executing there is another interrupt which has arrived from D1. So, now you may argue that there is also an ISR for device D1. So, should we stop this go here process this and then come back and resume here. This is called nesting that before something is finished you are going to somewhere else. This is nesting of interrupt requests; before the handling of interrupt request D0 is finished, another interrupt has come. For nesting there can be two possible scenarios.

The first one is what I have just now illustrated. D1 can interrupt the interrupt service routine for D0, get processed first and then the ISR of D0 will be resumed. So, when the D1 interrupt comes, the ISR of D1 will be processed first, then it will come back, and ISR of D0 will be resumed, and then it will be finished. This will involve nesting of interrupts and theoretically means any arbitrary number of such nestings can happen because interrupts may appear one after another. Some important interrupt request that came earlier might get delayed for its processing.

The other thing is that you can disable the interrupt system automatically whenever an interrupt is acknowledged, so that handling of nested interrupt is not required. Because I mean if you disable the interrupt system, then if some interrupt request comes it will not be considered, it will be ignored. This is another solution.

(Refer Slide Time: 25:38)

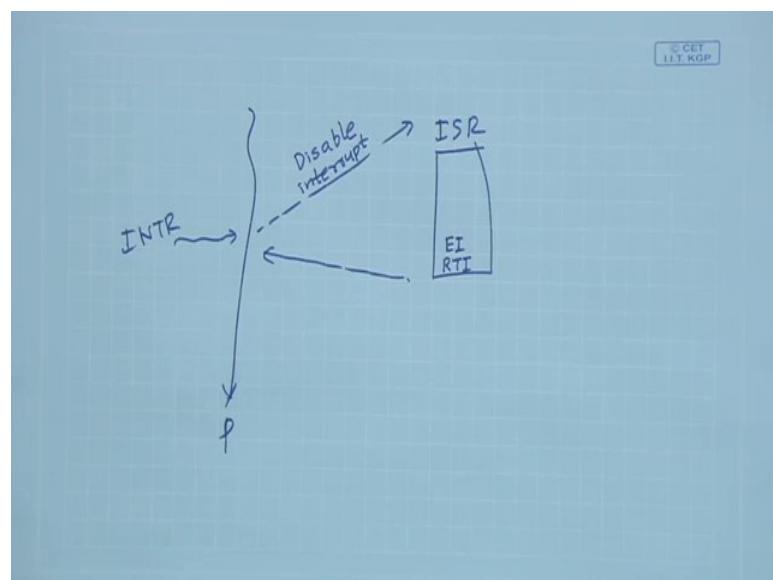
- Typical instruction set architectures have the following instructions:
 - EI : Enable interrupt
 - DI : Disable interrupt
- For the second scenario as discussed, the ISR will give an EI instruction just before RTI.
 - Some ISA combine EI and RTI in a single instruction.
- The DI instruction is sometimes used by the operating system to execute atomic code (e.g. semaphore wait and signal operations).
 - Nobody should interrupt the code while it is being executed.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Whenever you are processing an ISR you disable the interrupt system. So, nesting will not happen.

For this typical machine instructions are there for enabling and disabling interrupts, like EI and DI. So, for this second scenario the interrupt service routine will give an EI instruction just before return from interrupt.

(Refer Slide Time: 26:17)



Now, we are having a design where whenever an interrupt comes before jumping to the ISR interrupt is disabled. Like what I am saying is that a program was executing and

interrupt comes. Before you jumping to the ISR, before you are jumping to the ISR the hardware is automatically disabling the interrupt. So, execution of the ISR will not be interrupted. And in the ISR before the RTI instruction that is supposed to be the last instruction, there will be an explicit EI instruction so that before returning back you enabling interrupt again, so that any future interrupt if it comes should be acknowledged.

Some instruction set architecture combine EI and RTI in a single instruction. So, when you do a return, for means return from interrupt, interrupt will be automatically enabled and you do not need a separate EI.

(Refer Slide Time: 28:28)

Cases that make interrupt handling difficult

- For some interrupts, it is not possible to finish the execution of the current instruction.
 - A special RETURN instruction is required that would return and *restart* the interrupted instructions.
- Some examples:
 - a) Page fault interrupt: A memory location is being accessed that is not presently available in main memory.
 - b) Arithmetic exception: Some error has occurred during some arithmetic operation (e.g. division by zero).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

We have seen different cases of interrupts. There are some cases that can make the interrupt handling more difficult. Like we have assumed so far that whenever an interrupt is coming we have to finish the current instruction, then only we can acknowledge the interrupt. But there are some cases where you cannot finish the current instruction that makes interrupt handling more difficult. For some kinds of interrupt it is not possible to finish the execution of the current instruction.

For such cases you need a special return instruction that would be returning from the ISR, but after returning it will be restarting the same instruction which was interrupted; not that it will be returning to the next instruction it will be restarting the interrupted instruction. An example is page fault; this happens in operating system in the memory management when a memory location is accessed that is not presently loaded in

memory. You will have to load the requested memory location page and come back and again restart the instruction, so that it can access the memory location correctly this time.

And you can also think of arithmetic exceptions like division by 0, square root of a negative number. In such cases you are not able to finish the instruction. Such cases can make interrupt handling more difficult.

With this we come to the end of this lecture. In the next lecture we shall be continuing with our discussion on interrupts, and we shall see some more issues particularly regarding multiple devices sending interrupts and the different types of interrupts.

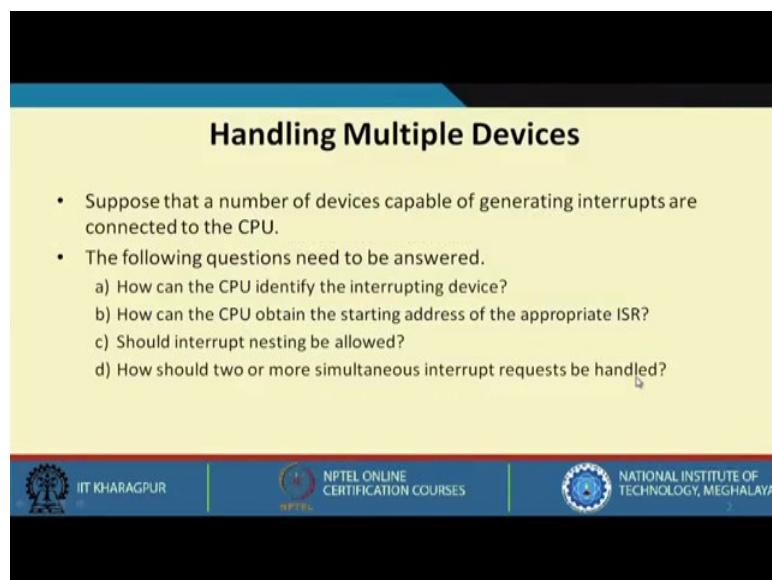
Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 47
Interrupt Handling (Part II)

We continue with our discussion on Interrupt Handling. In this lecture, we shall first be looking at some issues that occur whenever multiple devices are allowed to send interrupt signals to the processor. So, this is the Part II of Interrupt Handling.

(Refer Slide Time: 00:40)



Handling Multiple Devices

- Suppose that a number of devices capable of generating interrupts are connected to the CPU.
- The following questions need to be answered.
 - a) How can the CPU identify the interrupting device?
 - b) How can the CPU obtain the starting address of the appropriate ISR?
 - c) Should interrupt nesting be allowed?
 - d) How should two or more simultaneous interrupt requests be handled?

The slide footer features logos for IIT Kharagpur, NPTEL, and National Institute of Technology Meghalaya.

We start with this problem of handling multiple devices. We have already talked about this earlier in our last lecture, but there are something else we shall be discussing here.

Here we assume because there are multiple devices a number of devices can potentially generate interrupts to the CPU. Now, there are four questions we need to answer. We need to have solutions to all four of these. Some of these we have already answered earlier. First is how can the CPU identifies the interrupting device? Well, earlier we suggested one method that is using the interrupt vector concept. The interrupting device itself can send an interrupt vector through which the CPU can identify who has interrupted.

This is the way in which the device itself is identifying that well I have interrupted. So, CPU can know that.

The second important question is how can the CPU obtain the starting address of the ISR? Once the CPU has identified the device, this is easy because the address of the ISR can be stored in a table. After knowing which device has interrupted, the CPU can consult the table, get the address and can jump to that appropriate address.

The third question is should we allow interrupt nesting to happen because we saw earlier that interrupt nesting can create some problems, or we should be disabling the interrupts while interrupts have been processed? And lastly, for simultaneous interrupt requests, how we should handle them?

(Refer Slide Time: 02:49)

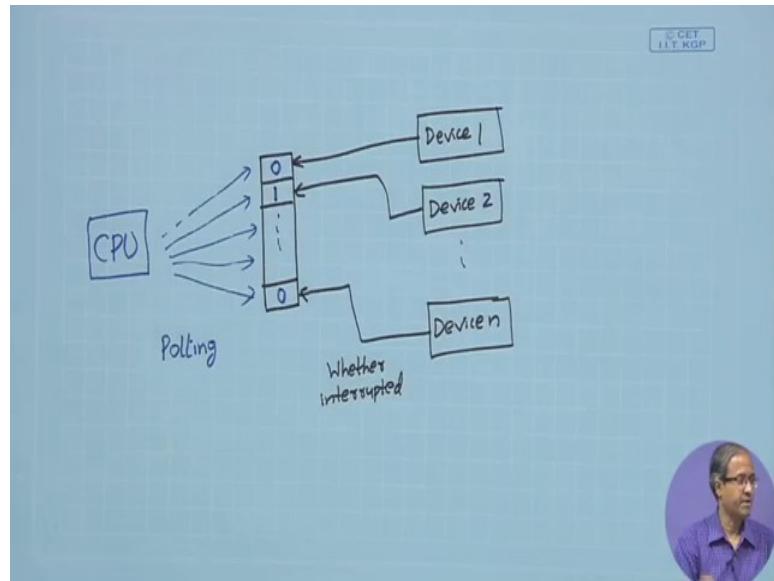
(a) Device Identification

- Suppose that an external device requests an interrupt by activating an INTR line that is common to all the devices. That is,
$$INTR = INTR_1 + INTR_2 + \dots + INTR_n$$
- Each device can have a status bit indicating whether it has interrupted.
 - CPU can *poll* the status bits to find out who has interrupted.
- A better alternative is to use the interrupt vector concept discussed earlier.
 - The interrupting device sends a special identifying code on the data bus upon receiving the interrupt acknowledge.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL TECHNOLOGY

Let us look at these one by one. First device identification. This has already been talked about earlier. The first thing is that there are several external devices which may be having different interrupt signals INTR1, INTR2 up to INTRn. So, if anyone of them generates an interrupt, you can make a logical connection and you can generate the consolidated interrupt request which will be sent to the CPU. So, CPU will get interrupted.

(Refer Slide Time: 03:36)



Now, in addition each device can have a status bit indicating whether it has interrupted. What we are saying is something like this. Let us say we have the different devices device1, device2 and devicen. Let us say there is a port, an input port, where each of the device can set one of the bits.

Now, what these bits will indicate? These will indicate whether interrupted or not. Let us say device 2 has interrupted. So, device 2 will be setting this particular bit to 1, and all other bits will remain 0. So, what the CPU will do? CPU can read this word and can check bits one by one which of them is 1. It can identify which of the bits is 1. This process is sometimes called polling. Polling means scanning one by one to find out which device has sent the interrupt.

This is exactly what is mentioned here in the slide. Each device can have a status bit indicating whether it has interrupted, and CPU can go on polling those bits one by one to find out that which device had interrupted. A better alternative will be to use interrupt vector concept that we talked about earlier, because when the CPU is doing polling, some additional time will be required because there will be program which will be running which will be shifting the bits one by one and will be checking which bit is 1, which bit is 0. That will take some time, but if the device itself sends an identifying code, the interrupt vector on the data bus, CPU can immediately read that and directly find out

which device has interrupted. There is no need of any polling. So, this is a better alternative.

(Refer Slide Time: 06:18)

(b) Find Starting Address of ISR

- For a processor with multiple interrupt request inputs, the address of the ISR can be fixed for each individual input.
 - Lacks flexibility.
- If we use the interrupt vector scheme discussed earlier, the device is able to identify itself to the CPU.
 - CPU can then lookup a table where the ISR addresses for all the devices are stored.
 - The interrupt latency is somewhat increased, since we are not immediately jumping to the ISR.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

The second question is to find starting address of ISR. Here there can be multiple ways. There can be processors where multiple interrupt requests pins are there. This you would see mostly in some older processors, but in most of the modern processors, you do not see many. You will see that there was multiple interrupt input lines and each of these input interrupt request lines was associated with a particular address of ISR, which means that if an interrupt comes over this line 1, it will jump to a particular address. If an interrupt comes on line 2, it will jump to another particular address.

So, when you write your ISR, you will have to write those ISRs in those particular addresses. This was one of the concepts which was used in some of the processors. For multiple interrupt request lines the address of the ISR can be fixed for each individual input by hardware. If an interrupt comes on a particular line, the control will jump to a particular address. So, ISR will have to be loaded there, but this method lacks flexibility because if there are four interrupt inputs, you can have only four ISRs, but in general the number of devices you are interfacing can be more than four also. So, flexibility is less in this approach.

Again if you use the interrupt vectors scheme, this is much more flexible. There can be many devices which are connected to the CPU. Whoever has interrupted will send the

corresponding interrupt vector on the data bus. So, there is no restriction as to how many devices can be connected. It can be 2, 5, 10, or 100. The interrupting device will have the responsibility of sending the interrupt vector on the data bus, and CPU can read it and uniquely identify the device who has sent the interrupt.

Once the CPU has identified the device, there can be a look up table where all the ISR addresses will be stored. The CPU can do a look up in that table and find out the corresponding ISR address, and then it will be jumping to the corresponding ISR. The only difficulty here is that you need to do some processing, searching a table and so on. So, a little more time is required for interrupt handling. This we are calling as the interrupt latency; the delay between an interrupt request signal is arriving and when the interrupt acknowledge is finally going out. This delay will be increased a little bit because here you need to do a little more processing to find out the address of the ISR.

(Refer Slide Time: 09:44)

(c) Interrupt Nesting

- A simple approach:
 - Disable all interrupts during the execution of an ISR.
 - This ensures that the interrupt request from one device will not cause more than one interruptions.
 - ISR's are typically short, and the delay they may cause in handling a second interrupt request is often acceptable.
- Interrupt priority:
 - Some interrupting devices may be assigned higher priorities than others.
 - *Example:* timer interrupt to maintain a real-time clock.
 - Higher priority interrupt may interrupt the ISR of lower priority ones.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Here we are not immediately jumping to the ISR. We are searching for the address first and then jump. Now, regarding interrupt nesting, we told earlier that we can have a simple approach of disabling all interrupts while the execution of an ISR is going on. This is of course simple. This will ensure that whenever an interrupt comes, there will not be any more interruptions. The ISR will be executing to completion, it will come back, and then if another interrupt comes, there will be another interruption.

There will be at most one interruption per interrupt request, but the point to note is that for most of the devices, the ISRs are typically very short. Just transfer a few bytes or words of data that is the task of the ISR. So, the total time taken for the ISR to finish or execute is typically very small. So, you can afford to use this approach that you can disable the interrupt because you know that ISR will be finished very quickly even if you have disabled. So, after finishing the interrupt can be handled which, has come after that. So, this will not be that much of a problem.

A better method may be to use interrupt priority, with some interrupting devices assigned higher priority than the others. The idea is that here you are not ruling away interrupt nesting all together. Here you are saying that you are allowed to do interrupt nesting, but only higher priority interrupts can interrupt the ISR of a lower priority interrupt. Suppose there is a timer which is generating an interrupt every 1 millisecond, which you are using to maintain a real time clock which is very accurate.

You cannot afford to delay this timer interrupt because this real time clock is a very important thing. Many subsystems of your computer may be using this real time clock for various purposes. When this real time clock interrupt comes, if the CPU sees some other ISR is currently being executed, it will interrupt that because this real time clock interrupt is a very high priority interrupt. A high priority interrupt can interrupt the ISR of lower priority devices or interrupts. This is the concept.

(Refer Slide Time: 12:54)

(d) Simultaneous Requests

- Here we consider the problem of simultaneous arrivals of interrupt requests from two or more devices.
 - CPU should have some mechanism by which only one request is serviced while the others are delayed or ignored.
 - If the CPU has multiple interrupt request lines, it can have a *priority scheme* where it accepts the request with the highest priority.

INTA_i has higher priority than INTR_j, where i < j.

IIT Kharagpur | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

By using interrupt priority, you can have an arrangement where high priority interrupts can interrupt the ISR of lower priority ones. Lastly comes simultaneous requests. Here we are assuming that simultaneous interrupt requests can arrive from two or more devices. Obviously CPU should have some mechanism to resolve that.

The first approach can be a simple priority scheme which we talked about earlier using a interrupt priority controller. Conceptually there is a CPU, there can be multiple devices; so each of them having a request and an acknowledgement line.

If more than one requests are coming together, the device or the interrupt input that has the higher priority will be acknowledged, while the others will be temporarily ignored. It will be handled later. Here we are assuming, INTR_i will be having higher priority than INTR_j, if $i < j$. So, INTR1 will be having higher priority than INTR2, INTR2 will be having higher priority than INTR3 and so on.

(Refer Slide Time: 14:19)

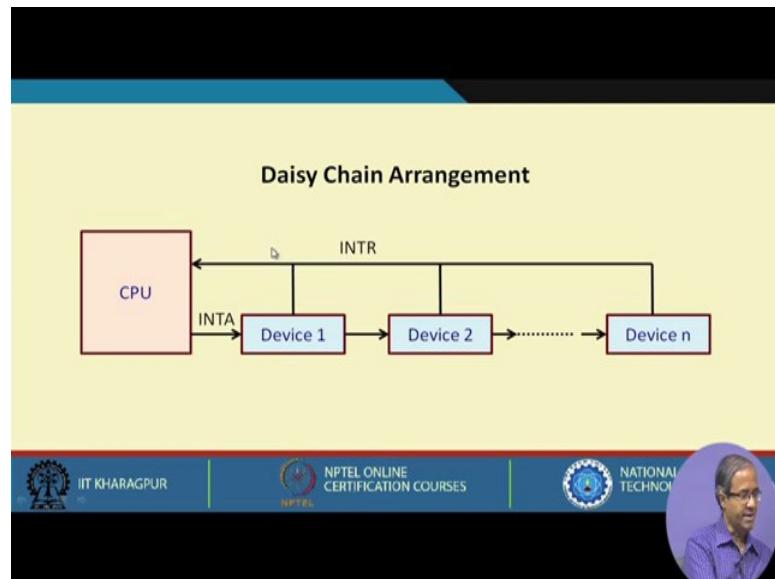
- Another way to assign priority is to use polling using *daisy chaining*.
 - In polling, priority is automatically assigned based on the order in which the devices are polled.
 - In daisy chain connection, the INTR line is common to all the devices, but the INTA line is connected in a daisy chain fashion allowing it to propagate serially through the devices.
 - A device when it receives INTA, passes the signal to the next device only if it had not interrupted. Else, it stops the propagation of INTA, and puts the identifying code on the data bus.
 - Thus, the device that is electrically closest to the CPU will have the highest priority.

There is another way to assign priority. Instead of fixed fixing, there is another way in which this priority is implemented in practice. This is a method called daisy chaining. The idea of this daisy chaining, we will be explaining with an example. The concept is fairly simple. In the method of polling you are checking the devices one by one. Now, there you can regard that priority of the devices are automatically assigned based on the order in which you are checking or you are polling.

Suppose there are 8 devices and you are checking the status of the devices one by one. Now, the order in which you are checking the devices that will determine the priority; the first device that you are checking well, of course we checked first. If it has interrupted, it will be handled first. So, it is having the higher priority. So, if the first device has not interrupted, then only you are going to the next that is having a lower priority. So, the order in which you are checking that implicitly defines the priority.

For daisy chain connection, the interrupt request line is common to all the devices, but the interrupt acknowledge line is connected in a particular fashion. We will be showing this. This is called a daisy chain which propagates serially through the device.

(Refer Slide Time: 16:08)



All the devices having the interrupt request are all connected together. Any device interrupting will be generating an INTR, but INTA is not connected directly to all the devices. INTA is connecting to device 1, device 1 is generating an INTA signal to device 2, device 2 is generating an INTA signal to device 3, and so on.

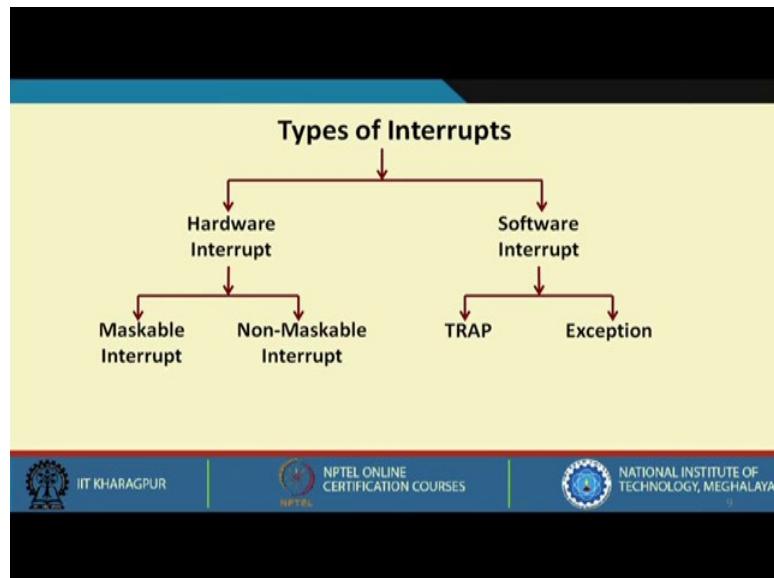
Now, the way it happens is like this. After an interrupt is generated, this interrupt could have come from any of the devices. The CPU generates an acknowledgement and it first comes to device 1. If device 1 had interrupted, then it will send the interrupt vector on the data bus and it will stop this INTA from propagating any further, but if device 1 had not interrupted, and then it will allow this INTA to go forward to device 2. Device 2 will do the same thing. It will check whether it has interrupted or not. If yes, then it will put

the interrupt vector on the data bus and if not, it will allow INTA to propagate to the next device and so on.

The order in which the device is connected will define the priority. The device which is nearest to the CPU will be having the highest priority, and the device farthest will be having the lowest priority. So, a device when it receives INTA, passes the signal to the next device only if it had not interrupted. Else if it had interrupted, it will stop the propagation of INTA and will put the interrupt vector or the identifying code on the data bus.

So, as I had said the device that is electrically closest to the CPU will have the highest priority. This is how daisy chain works.

(Refer Slide Time: 18:29)



Now, let us look at the different types of interrupt. This diagram shows you the different types. Broadly speaking you can classify interrupts as hardware interrupt and software interrupt. Well, the name comes from the source. Hardware interrupt is something which is generated by the hardware, and software interrupt is something which is generated because of the execution of an instruction.

That is why it is called a software interrupt and hardware interrupt again in turn can be either maskable or non-maskable, and under software interrupt, you can make a

categorization, you can have something called trap or you can have something called exception. Let us look at these nomenclatures what these are actually.

(Refer Slide Time: 19:27)

The slide has a yellow background with a black header and footer. The title 'Hardware Interrupt' is at the top. The main content lists two types of interrupts:

- **Hardware Interrupt:**
 - The interrupt signal is coming from a device external to the CPU.
 - Example: keyboard interrupt, timer interrupt, etc.
- **Maskable Interrupt:**
 - Hardware interrupts that can be masked or delayed when a higher priority interrupt request arrives.
 - There are processor instructions that can selectively mask and unmask the interrupt request lines of the CPU.

At the bottom, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

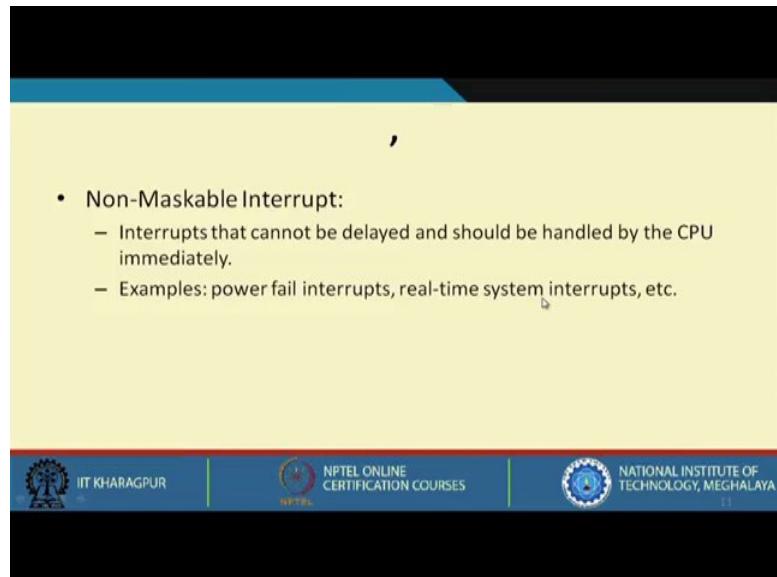
Coming to hardware interrupt the interrupt signal is generated by the hardware. It is coming from a device which is external to the CPU from some hardware unit. It is coming from memory, coming from IO device; some examples are keyboard interrupt, timer interrupt, etc. Suppose I am maintaining the time of the day, the tick is coming every millisecond. So, whenever a tick comes, you increment the value of the clock. These are examples of hardware interrupt.

Now, maskable interrupt means well you are allowed to mask some of the interrupts. Mask means temporarily stop the handling of those interrupts. This means masking. That means, you are allowed to delay the handling or the processing of the interrupts. These kind of interrupts are called maskable interrupts.

The hardware interrupts that can be masked or delayed when a higher priority interrupt request arrives are called maskable interrupts. Suppose there are multiple interrupts. You are handling one of them. If a higher priority interrupt comes and you can afford to delay the current interrupt which has been handled, then we say that it is a maskable interrupt. You are able to mask or pause the handling of that interrupt temporarily.

In systems where this kind of a maskable interrupts are allowed, there are instructions using which you can mask or unmask the various interrupts.

(Refer Slide Time: 21:30)



Non-maskable interrupt is one where the interrupts cannot be delayed. They should be handled by the CPU immediately. Some examples are some kind of power fail interrupts. You detect that the power supply of your computer is failing; the voltage level is going down. May be there is a power failure. So, if there is an interrupt which comes, now I may save some of my important data to a non-volatile part of the memory. This is a very important or high priority interrupt.

Similarly, there can be some real time system interrupts. Real time system interrupt means some interrupt request is coming and the interrupt request has to be serviced within well-defined time limit. So, if you are not able to do that, may be your processing will be wrong or the objective for each of design the system will be lost.

Well, some examples may be medical information system where you are monitoring a patient in real time depending on some health status. You are taking some corrective actions. So, if there is something anomalous, you must take some action immediately. You think of an industrial control plant. You are sensing temperature, pressure, humidity, so many things. So, if you see that some parameter values has gone beyond the desired range, you may have to immediately take some corrective action.

Well, for a defense system you see that an enemy aircraft or a missile is heading towards you, you should immediately start the intercepting mechanism. So, these are some examples where interrupts have to be handled immediately and we have to use non-maskable interrupt for such cases. These interrupts cannot be masked or delayed. These are extremely important.

(Refer Slide Time: 23:45)

The slide has a yellow background with a black header and footer. The header contains the title 'Software Interrupts' and the subtitle 'What is Software Interrupt?'. The footer contains logos for IIT Kharagpur, NPTEL, and National Institute of Technology Meghalaya, along with the text 'NPTEL ONLINE CERTIFICATION COURSES'.

- Software Interrupt:
 - They are caused due to execution of some instructions.
 - Not caused due to external inputs.
- TRAP:
 - They are special instructions used to request services from the operating system.
 - Also called *system calls*.

Coming to software interrupts, they are caused due to execution of some instructions. They are not caused by some external hardware. One type is called Trap. They are also sometimes called system calls. These are nothing but special instructions which are used to request services from the operating system.

These we studying in more detail in the operating system course, but for the time being let us assume that a trap or a system call is a kind of an instruction which behaves similar to an interrupt; so when it is being executed, some instructions are saved and then, you jump to a routine. That routine gets executed and that may be part of the operating system. Then, you come back. That is what a trap is.

(Refer Slide Time: 24:45)

- Exception:
 - These are unplanned interrupts generated while executing a program.
 - They are generated from within the system.
 - Examples: invalid opcode, divide by zero, page fault, invalid memory access, etc.

The second kind of software interrupt is called exception. Exception is unplanned while trap is planned. That means you have an instruction and you are executing it. This is the planned call, but an exception is unplanned. They are generated within the system, but you do not know when they will come. They might be generated because of something which is beyond your control or due to some error this has happened. Like you may encounter due to some error, let say some memory error, some opcode has become invalid ,and you are trying to decode an instruction and find out that the opcode is not a valid opcode. So, you may have to interrupt immediately and tell that well there is something wrong and I have to stop my program.

Again, divide by zero, page fault, invalid memory access. You are trying to access a memory which you are not supposed to access. It is part of some other program or process. There is memory protection hardware which will prevent you from accessing that. If you try to access a memory location which is beyond your allowable limit, then a interrupt will be automatically generated.

Whenever such a thing happens, you generate something called an exception. These are also software interrupts. Later on we shall see when we look at the pipeline implementation of MIPS32 processor ISA, interrupt handling and exception handling is a real problem and we shall of course also suggest some of the approaches which we can use to handle interrupts and exceptions in MIPS32 pipeline implementation.

We have come to the end of this lecture, and in the next lecture we shall be continuing with some other IO transfer techniques, namely direct memory access which we have not yet discussed following which we shall be discussing some of the commonly used standards for data transfer, some bus standards and so on.

Thank you.

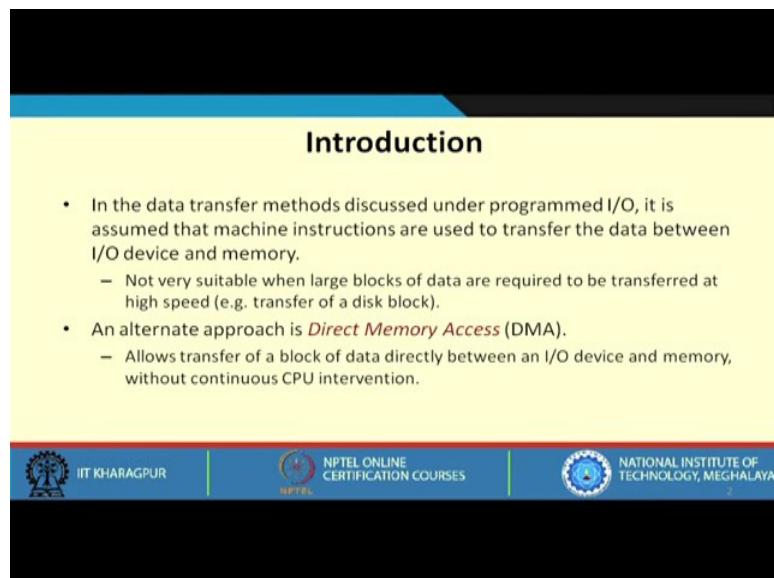
Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 48
Direct Memory Access

In the last few lectures we have discussed the programmed I/O technique, using which the processor can transfer data to and from an external I/O device. Now one thing to note is that during the actual transfer of data, the CPU is executing some instructions, which is responsible for transferring the data; may be from an input port to a CPU register, and from register to the memory, or vice versa.

So, it is the responsibility of that program to transfer the data and this is true for synchronous, asynchronous or interrupt driven I/O whatever you think of. Of course, in interrupt driven I/O the overhead will be less because the CPU is not spending time to check the status of the device whether it is ready or not.

(Refer Slide Time: 01:38)



Introduction

- In the data transfer methods discussed under programmed I/O, it is assumed that machine instructions are used to transfer the data between I/O device and memory.
 - Not very suitable when large blocks of data are required to be transferred at high speed (e.g. transfer of a disk block).
- An alternate approach is *Direct Memory Access* (DMA).
 - Allows transfer of a block of data directly between an I/O device and memory, without continuous CPU intervention.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Today we shall be looking at an alternate way to transfer data between I/O device and memory -- this is called direct memory access. In short, we call it DMA. Let us see what this method is all about. As I have just now mentioned under the programmed I/O technique, whatever three methods we have discussed, there are machine instructions that are actually executed and they are responsible for the transfer of the data. But one

problem with these methods is that whenever you use a program to transfer a data, the instruction execution time will come in as a bottleneck.

Suppose you need to execute a minimum of 8 or 10 instructions to transfer a data. So, the execution time of those 10 instructions will act as a bottleneck, you cannot have a data transfer speed which will be faster than that time. This is one drawback in these schemes. These methods will not be very suitable when we have a high speed device that is transferring data not only in large chunks (which means there are large blocks of data which are transferred in one go) but also at a high speed. This is quite characteristic of block I/O devices like a disk, like a CD or a DVD, these kind of devices they will transfer a block of data at a given time.

An alternate approach that we exploring now is called direct memory access. Here there is no execution of instructions for transferring of the data. Here by having some hardware mechanism we can allow the transfer of data, may be a block of data also, directly between I/O and memory without CPU intervention during that time.

When I say continuous CPU intervention, it means you recall in the programmed I/O technique for every byte or every word of data that are transferred CPU has to execute a program. But here, CPU has to initiate the DMA operation and after that the CPU can proceed and do something else, CPU need not have to interfere during the time the data transfer are taking place. And again at the end when the data transfer is over, CPU may have to do some book keeping operations; so only at the beginning and at the end. Thus, CPU does not require continuous intervention.

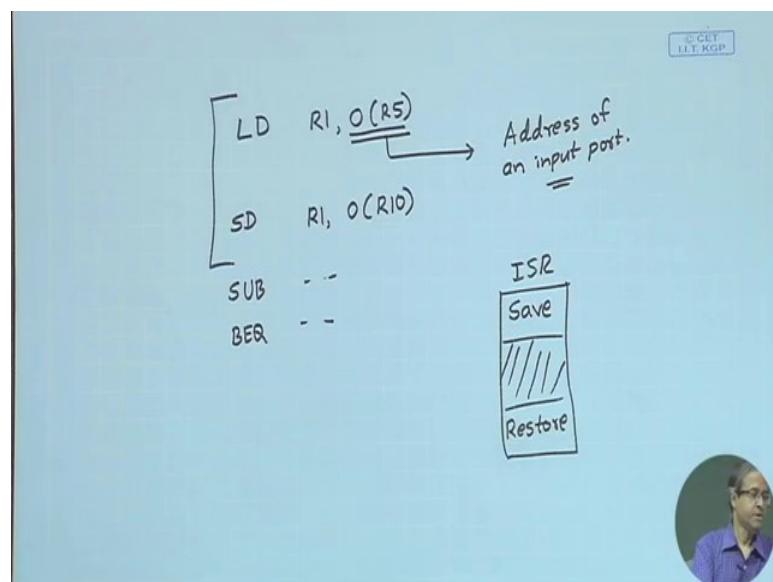
(Refer Slide Time: 04:50)

- Why programmed I/O is not suitable for high-speed data transfer?
 - a) Several program instructions have to be executed for each data word transferred between the I/O device and memory.
 - Suppose 20 instructions are required for each word transfer.
 - The CPI of the machine running at 1 GHz clock is 1.
 - So, 20 nsec is required for each word transfer → maximum 50 M words/sec
 - Data transfer rates of fast disks are higher than this figure.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Again looking back into that issue; why programmed I/O is not suitable for high speed data transfer? We cite 2 reasons; first one I have already mentioned just sometime back -- several program instructions may have to be executed, for each word that is transferred between I/O device and memory.

(Refer Slide Time: 05:30)



So, typically what kind of instructions will be executed? If we look at the MIPS instruction format, you may be loading some data into a register R1. The address is stored in some other register, let us say R5, this 0(R5) may indicate the address of an

input port. For memory mapped I/O you can load like this and after loading you can store the data into memory wherever you want to store this data, you can store let us say, R10 is pointing to memory.

So, basically you need 2 instructions, but there will be other instruction like you will be decrementing some pointer, I mean you will be doing some subtract, then you will be using a branch-if-equal etc. There will more instructions involved in general. In addition to that if you are thinking of an interrupt driven data transfer then within the ISR, the interrupt service routine, you need to have some additional code like saving the status and at the end restore the status. So, these additional instructions will also be there.

Let us take am example. Suppose for each word transfer we require 20 instructions to execute which is quite realistic. And let us assume in the ideal case that the CPU is running with a CPI of 1, and the clock is 1 GHz, which means every one nanosecond one instruction is getting executed. So, for the 20 instructions the total time required will be 20 nanoseconds. For transferring every word we require 20 nanoseconds. If we just calculate $1 / 20$ nsec, maximum data rate will be 50 million or mega words per second. This is the maximum you can achieve. But if you see the fast disk unit that are available today the sustained data transfer speeds can be more than 50 mega words per second, which means such programmed I/O techniques will not be suitable for such devices.

(Refer Slide Time: 08:17)

b) Many high speed peripheral devices like disk have a synchronous mode of operation, where data are transferred at a fixed rate.

- Consider a disk rotating at 7200 rpm, with average rotational delay of 4.15 msec.
- Suppose there are 64 Kbytes of data recorded in every track.
- Once the disk head reaches the desired track, there will be a sustained data transfer at rate $64 \text{ Kbytes} / 4.15 \text{ msec} = 15.4 \text{ MBps}$.
- This sustained data transfer rate is comparable to the memory bandwidth, and cannot be handled by programmed I/O.

The second reason is there are many high speed peripheral devices; again disk is a classic example, which have a synchronous mode of operation. And during this mode of operation data are transferred at a fixed rate. We mentioned that the disk access time consist of several components. First is a seek time. You will have to give some time, for the read write head to move under the required track from where you want to read or write, second part is rotational delay you will have to wait till the desired sector where you want to access rotates under the read write head. And third once when you are there depending on the speed with which the disk is rotating and the density of the data with which it is recorded on the surface, data will get transferred at a fixed rate. Why fixed rate? Because the disk is rotating at a very accurate and fixed rate, 3200 rpm, 5200, 5400 rpm or 7200 rpm.

Once you are on the track and the data transfer has started, after that it is something like synchronous data transfer -- data will be available at fixed time intervals. That is what is mentioned here. Let us take an example again; say, we have a disk which is rotating at 7200 rpm which earlier we saw that for this the average rotational delay will be 4.15 milliseconds.

Suppose that there are 64 kilobytes of data in every track. So, once the disk head reaches the desired track and the starting sector is under it, then data can be transferred at a sustained rate because the disk is rotating with the average rotational delay of 4.15 milliseconds. In 4.15 millisecond time the disk will be rotating once and during this rotation 64 kilobytes will be transferred. The data transferred rate will be 64 divided by this, which comes to about 15.4 megabytes per second.

Now, this data transfer bandwidth is the sustained rate once the read write head is there, because you see movement of the read write head requires several milliseconds, that there will be a long delay, but once you are on the place, the disk is rotating pretty fast and the data transfer rate will be high.

If you look at the typical memory bandwidths that are also in the order of tens of megabytes per second. So, it is comparable. If there was some mechanism with which you can directly transfer the data to memory from this device, then a speed match could have been obtained. This data rate obviously, cannot be handled by programmed I/O as we had seen earlier.

(Refer Slide Time: 11:57)

DMA Controller

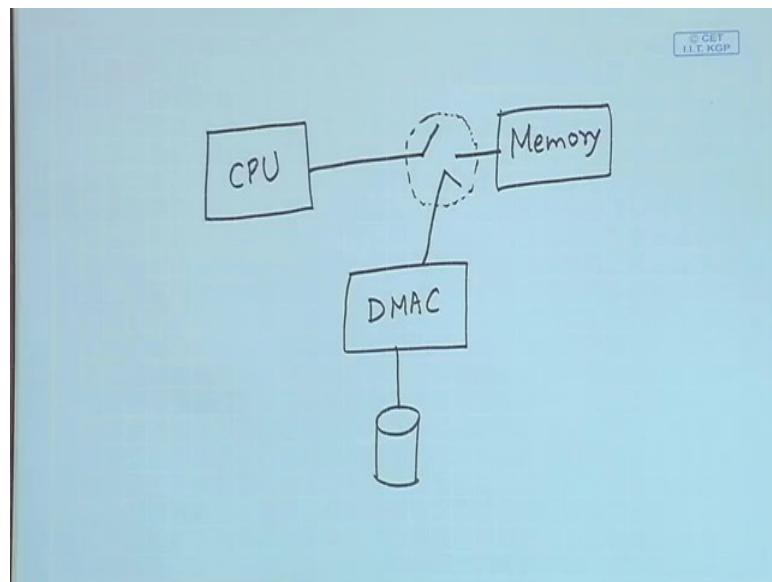
- A hardwired controller called the DMA controller can enable direct data transfer between I/O device (e.g. disk) and memory without CPU intervention.
 - No need to execute instructions to carry out data transfer.
 - Maximum data transfer speed will be determined by the rate with which memory read and write operations can be carried out.
 - Much faster than programmed I/O.

↳

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

DMA controller is a hardware device we require to carryout DMA transfer. Now let us see what is a DMA controller and what are it is functions. DMA controller basically enables direct data transfer between I/O device and memory without CPU intervention.

(Refer Slide Time: 12:24)

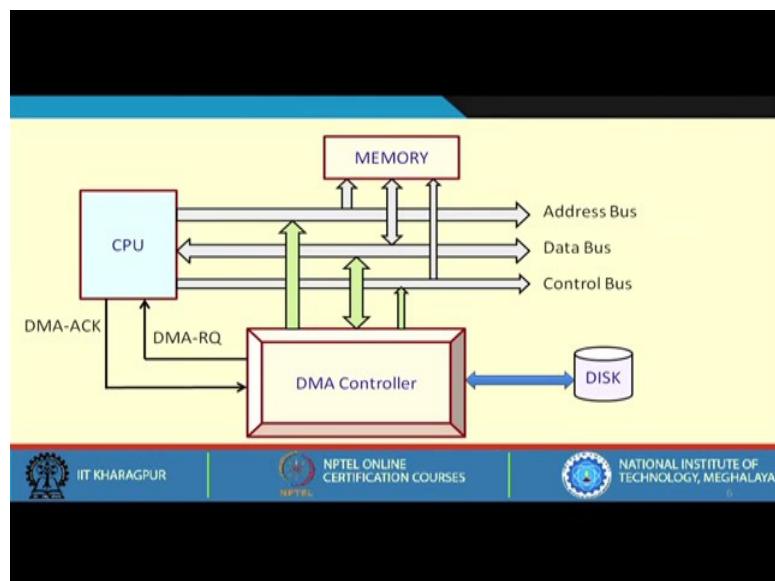


Now, pictorially speaking, we have CPU, we have memory, and now we have a third entity the DMA controller. Now CPU will be normally accessing memory for fetching instructions and also accessing data.

Now, DMA controller will also be accessing memory because there will be some devices, let us say a disk is connected. Here this disk will be connected to the DMA controller and DMA controller will try to transfer data directly to memory. Now, there will be something like a switch, there will be a 3-way switch. So, either the CPU will be connected to the memory or the DMA controller will be connected to the memory. There has to be a mechanism with which this switch can be controlled in a proper way. Because under no circumstances both CPU and DMA controller should turn on the switches, both are accessing to memory because there will be a clash, there will be bus conflict.

If the DMA controller is able to carry out direct data transfer between the device and the memory, then obviously, there is no need to execute instructions for the purpose. And the maximum data transfer speed will be determined not by instruction execution time, but by the memory read write times and also the I/O data transfer times. And this method will be much faster than the programmed I/O techniques.

(Refer Slide Time: 14:19)



Pictorially DMA controller is connected like this. You see when you have the processor or the CPU connected to the memory, there are 3 different busses: one is of course, the bus carrying the address, address bus, second bus is the data bus, and third bus will contain the different control signal like read write enable all these things.

Now, when CPU is interfaced with memory, so all these buses are connecting to the memory. And CPU is in full control CPU generates an address, issues a read signal, and

the data will be read from the memory -- the data comes from the data bus. Similarly when it wants to write it puts an address, puts a data, then puts a write signal -- the data will be returning to memory.

But now there is a third person the DMA controller also setting and wanting to use the bus. I am using wanting to because while CPU is using DMA controller cannot use the bus. What will it do is -- it will set all the signals to the high impedance or tristate. Putting the signals in high impedance state means they are electrically disconnected. Normally CPU and memory are connected and they are working like that.

But now suppose the disk or the CPU wants to transfer some data, from the disk and memory. So, now the DMA controller will wake up. What the DMA controller will do? It will send a DMA request signal to the CPU; this signal will tell the CPU that the DMA controller or some device is wanting access of this bus so that data can be transferred through memory.

The CPU will be receiving this after some time After sometime what the CPU will do – it will relinquish control of the bus; means now CPU will make its own bus tristate. CPU will be disconnected. And after doing that it will be sending back a DMA acknowledge signal.

When the signal reaches DMA controller, DMA controller knows that CPU has disconnected itself on the bus. So, now, DMA controller will enable its own bus and will directly access memory. This is roughly how it works. So, the DMA controller and the CPU has a handshaking mechanism via the DMA request and the DMA acknowledge signal, by virtue of which exactly one of the 2 devices are having their bus signals enabled, the other one is having them in the high impedance state they are disabled. Under normal situation CPU is accessing, and while transfer is going on DMA controller is accessing.

(Refer Slide Time: 17:54)

Steps Involved

- When the CPU wants to transfer data, it initializes the DMA controller.
 - How many bytes to transfer, address in memory for the transfer.
- When the I/O device is ready for the transfer, the DMA controller sends DMA-RQ signal to the CPU.
- CPU waits till the next DMA breakpoint, relinquishes control of the bus (i.e. puts them in high impedance state), and sends DMA-ACK to DMA controller.
- Now DMA controller enables its bus interface, and transfers data directly to/from memory.
- When done, it deactivates the DMA-RQ signal.
- The CPU again begins to use the bus to access memory.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, ROURKELA

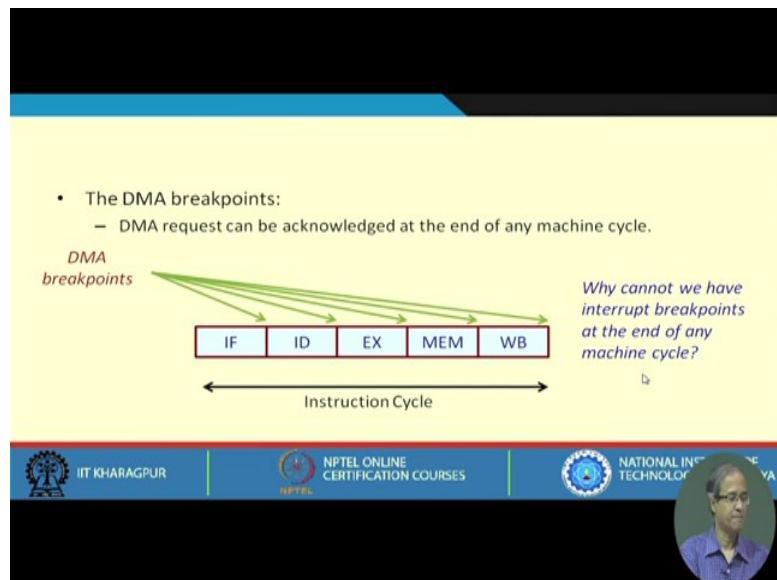
The steps involved as I had said, when the CPU wants to transfer data, it initializes the DMA controller. Because it has to tell something to the DMA controller. Like how many bytes to transfer what is the size of the block. So, if it is read or write whatever, where from in the memory means we have to transfer the data, so addressing memory.

Second when the I/O device is ready for the transfer, the DMA controller will send the DMA request signal to the CPU. CPU will wait for the next DMA breakpoint, I will explain this a little later. DMA break point means the time step where CPU can release the bus. CPU will wait for that and once the next DMA breakpoint comes it relinquishes control of the bus; that means, puts them in the high impedance state. And after doing this it will be sending DMA acknowledge signal to the controller.

DMA controller is now having control, it will now enable its own bus interface and will transfer data directly from the memory and the device. When the DMA controller has finished transferring the data it will deactivate the DMA request signal, but of course, before that it will be disable its own bus lines.

Now here one thing I have not mentioned is there is also an interrupt signal which is coming into the picture. After the DMA controller has finished it can send an interrupt to the CPU telling that when the transfer is complete.

(Refer Slide Time: 20:17)



Now, talking about the DMA breakpoints; you see I mentioned earlier when we discussed interrupts that the instruction execution cycle can be divided into 5 machine cycles IF, ID, EX, MEM and WB. Now you recall for interrupts we were acknowledging the interrupt only at the end, but for DMA we are saying that we can have DMA acknowledged at the end of a machine cycle also. Now why this difference let us try to understand.

Well in interrupt processing what was happening, whenever there is an interrupt you jump to an interrupt service routine, you save the status registers and other information execute the ISR restore the status, and then come back to the point from where you were interrupted. This is how interrupt works.

Now, you see if we allow interrupts to be acknowledged in between the machine cycle also. Then the only trouble is that you will have to store or save much more information than just the registers. Suppose you are stopping the CPU at the end of the ID cycle and instruction had completed ID instruction decode that time you are stopping interrupting and you are jumping to the ISR.

So, here you will also have to store all the intermediate registers, the instruction that was fetched in IR. So, after decoding that sign extended value. All those things also will have to be saved. Unnecessarily you are complicating the saving and restoring part the process, but in DMA there is nothing to save and restore because you are not interrupting

the CPU. You are just putting it in the pause mode. You tell the CPU just please wait for a while let me do it after that you resume. It is not like interrupt that the CPU is interrupted the CPU jumps to another program, ISR executes it and then comes back, in DMA it does not happen that way. CPU is just going into the pause mode.

Why cannot we have interrupt breakpoints at the end of any machine cycle? Because we will have to store lot of status information.

(Refer Slide Time: 23:05)

- For every DMA channel, the DMA controller will have three registers:
 - a) Memory address
 - b) Word count
 - c) Address of data on disk
- CPU initializes these registers before each DMA transfer operation.
- Before the data transfer, DMA controller requests the memory bus from the CPU.
- When the data transfer is complete, the DMA controller sends an interrupt signal to the CPU.

IIT Kharagpur | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

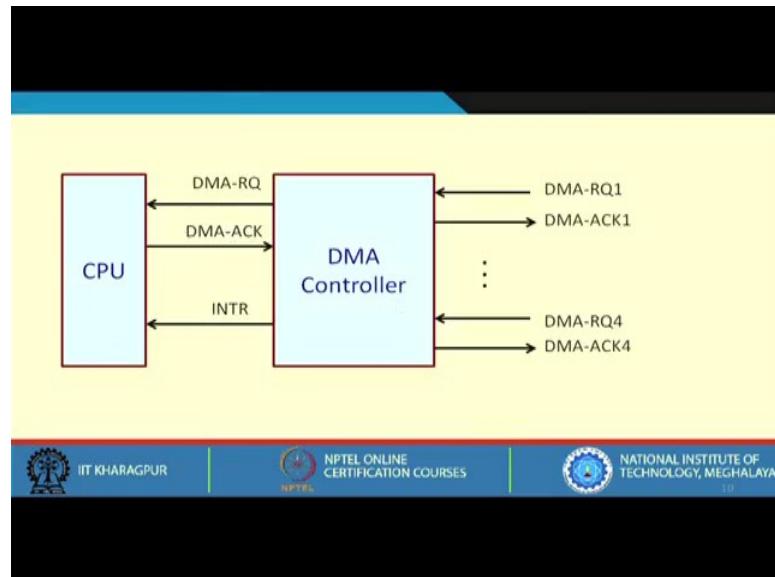
Now for the DMA controller for every I/O device that is connected to it, we call it a DMA channel. There are 3 registers which are there inside the DMA controller. Memory address register, word count register and another register, which stores the address of data on the disk.

Before a DMA operation starts CPU has to initialize all these 3 registers. Because CPU has to tell for example, for a write operation this is the memory address where the data is stored. It will also tell how many words to write, that is the data count. And lastly it will also have to tell in disk where to write which sector number which track number etc.

This is one thing I did not mention in the previous diagram, before the data transfer DMA controller requests the memory bus from the CPU by sending the DMA request signal. But when the transfer is complete typically there is also an interrupt signal which is activated. Because when the transfer is complete it is not just that CPU can be brought

out of the pause mode, you also tell the CPU that well you had initiated an I/O operation, now the I/O operation is complete, it is in memory if it is an input operation. Now you can do whatever processing you want to do on that data.

(Refer Slide Time: 25:00)



Pictorially speaking, this will be the interface. DMA controller will be interfacing with CPU with DMA request and acknowledge and in addition the interrupt request. And on the other side for every device or group of devices there will be a separate request and acknowledgement.

Whenever any of the devices sends a request DMA controller in turn will inform the CPU by activating DMA request. CPU will be relinquishing the bus, send back acknowledgement and then DMA controller will be using the register values corresponding to this device to carry out the transfer. Because for all these devices there will be different sets of registers.

(Refer Slide Time: 25:53)

DMA Transfer Modes

- DMA transfer can take place in two modes:
 - a) DMA cycle stealing
 - The DMA controller requests for a few cycles 1 or 2.
 - Preferably when the CPU is not using memory.
 - DMA controller is said to steal cycles from the CPU without the CPU knowing it.
 - b) DMA block transfer
 - The DMA controller transfers the whole block of data without interruption.
 - Results in maximum possible data transfer rate.
 - CPU will lie idle during this period as it cannot fetch any instructions from memory.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

DMA transfer broadly speaking can take place in 2 different modes. The first mode is called DMA cycle stealing mode. Now cycle stealing mode conceptually is like this, you see CPU is normally working it is accessing memory because for every instruction execution it has to fetch.

Now, what the DMA controller is doing? DMA controller will be grabbing the bus from the CPU by activating DMA request, but only for a very short time, only for the transfer of one or 2 words of data. It will not continuously transfer the whole block. It will transfer one or 2 words and then it will again give bus back to the CPU.

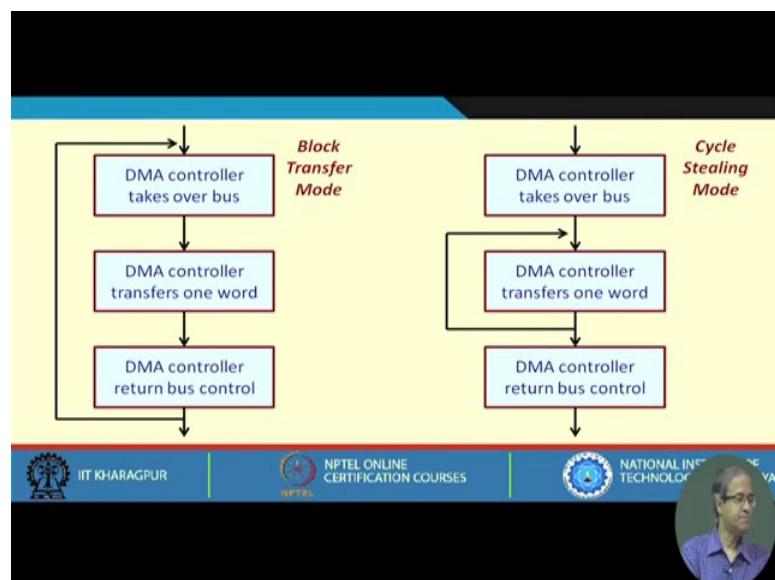
Well this serves 2 purposes, first the CPU need not have to be paused for very long. Because if it is a long block to be transferred in one go, then the CPU will have to wait for the whole time. Secondly, CPU does not necessarily access memory in all the machine cycles. For the MIPS if you see memory is accessed only during IF and MEM.

Suppose when DMA request came CPU was in the ID cycle. So, internally CPU execution can proceed because ID does not require memory, ID can proceed, EX does not require memory EX can also proceed. So, some of the cycles which CPU is not using the DMA controller is time to steal it from the CPU. CPU is also not getting hampered due to that; this is the basic idea behind cycle stealing.

The other alternative is you transfer the whole block in one go. The DMA controller transfers the whole block of data without interruption. This of course, will result in very high data transfer rate -- maximum possible rate, but CPU will have to lie idle during the period of the data transfer.

Now, depending on the kind of devices you may decide on block transfer or cycle stealing. So, if you see that if it is a continuous stream of data which is coming at a very high speed then you may have to go for block transfer mode, but if you see intermittently data are coming where the speed is not that high you can go for cycle stealing mode.

(Refer Slide Time: 29:25)



So, when you are initializing the DMA controller for a particular device you can also specify which mode of DMA transfer you want.

The transfers pictorially can be represented as follows.

(Refer Slide Time: 30:52)

The slide has a black header and footer. The main content area is yellow. The title 'Others Applications of DMA' is centered at the top of the yellow section. Below the title is a bulleted list:

- Other than data transfer to/from high-speed peripheral devices, DMA can be used in some other areas as well:
 - High-speed memory-to-memory block move.
 - Refreshing dynamic memory systems, by periodically generating dummy read requests to the columns.

At the bottom of the slide, there are three logos with their respective names: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

DMA is used not only for transfer of data between I/O devices and memory. There are other interesting applications also. Like, you think of a high speed memory to memory block move. Suppose you have an application where you want to transfer let us say, large block of data from one part of the memory to another.

The normal way to write program that will be reading the data one word at a time and store in the other place, read and store like that. So, again this will involve instruction execution and the data transfer of the block will be slower. But you can use the DMA controller for this purpose also. Here DMA controller instead of interacting with the device it is interacting with memory on both sides. It will read from memory, it will write into memory. And these 2 things are done by the hardware so no need of any instruction execution.

And the other applications where it is used is for dynamic memory system. You see dynamic memory system requires periodic refreshing. And for refreshing you have to issue some dummy read signals to the rows or columns depending on how the memory is organized. The DMA controller can be used to periodically generate such dummy read requests so that the dynamic memory systems are getting refreshed. Well here of course, nowadays the DRAMs that are available are intelligent enough -- this refreshing circuitry is also built into those boards or the cards or the chips. So, you need not have to do it separately.

With this we come to the end of this lecture. Over the last few lectures we had looked at various data transfer techniques between the CPU and the external input and output devices that are also called peripheral devices. In the next lecture we shall be looking at two simple devices, and look at some issues regarding the interfacing or how the data transfer can take place.

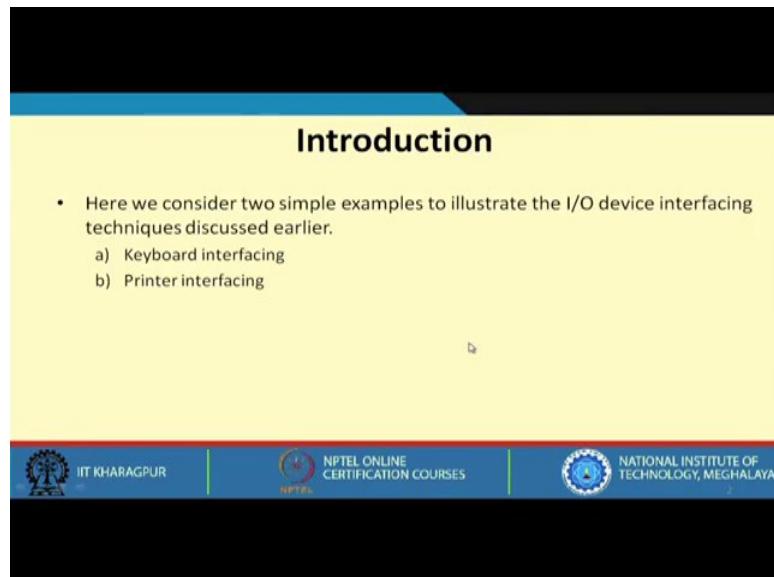
Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 49
Some Example Device Interfacing

In this lecture we shall be looking at two examples for device interfacing. Deliberately we have taken some examples which are simple enough so that you can actually understand what is happening inside. And some of the I/O transfer techniques that we have discussed earlier you can see how we can use them here.

(Refer Slide Time: 00:44)



The title of the lecture is some example device interfacing. As I had said that we will take 2 simple examples. One is the problem of interfacing a keyboard, and the second one is the problem of interfacing a printer. Let us look into this one by one.

(Refer Slide Time: 01:03)

Keyboard Interfacing

- What is a keyboard?
 - A set of pushbutton switches (keys) interfaced to a computer.
 - Typically arranged in the form of a two-dimensional matrix.
 - A key is connected to a row line and a column line at every junction.
 - Results in minimization of the number of port lines required.



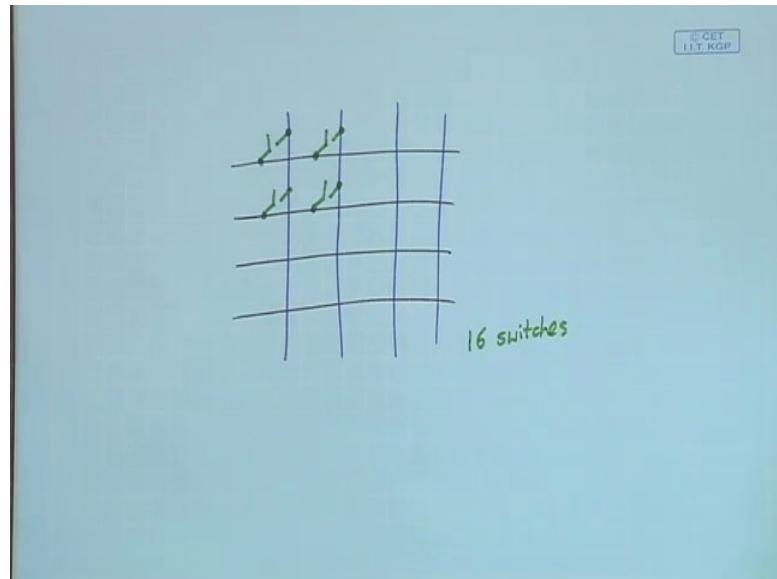
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Keyboard interfacing: well, all of us use keyboards in our computer systems, in our mobile phones -- wherever we have a mechanism for entering something. Even in our microwave oven we have a version of the keyboard, even with our TV remote there is a keyboard. With our air conditioning machine remote control there is a keyboard. So, any device which allows user to give some input via pressing of switches is a kind of keyboard.

A keyboard in generic sense is nothing but a set of pushbutton switches that are called keys. They are interfaced to some system, computer or microcontroller. This is an example of a small keyboard where you can see that there are 16 keys, and this is a standard computer keyboard, which is also conceptually very same.

Now, here the small example; as it shows that the 16 keys we have organised in 4 rows and 4 columns. Now logically speaking we arrange the keys in this fashion only. Why? See there are rows and columns we connect a key at the junction of the rows and column.

(Refer Slide Time: 02:43)

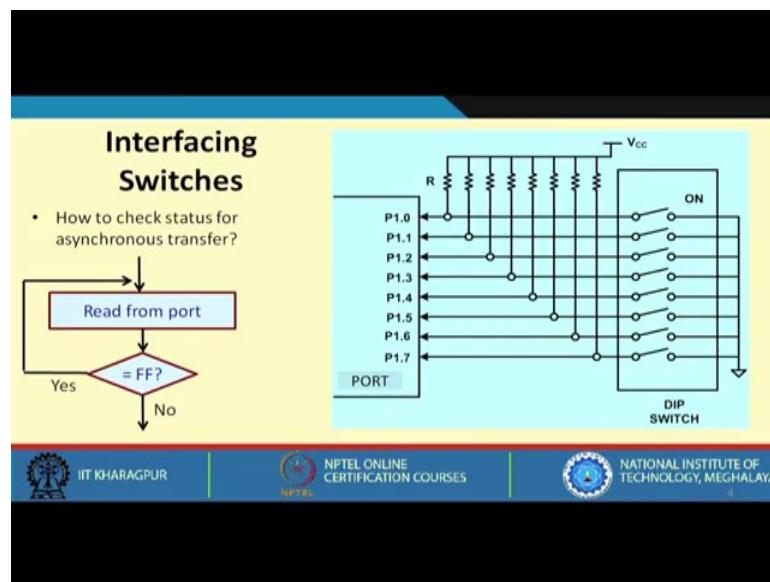


Suppose I have some row lines and also there are some column line, but they are not connecting they are on a different layer there are column lines.

Now, what we do? We connect a switch between every row and column. There is one switch connected here, there is one switch connected here, one switch connected here like this. So, in every junction there will be a switch. So, for a 4 by 4 thing there will be 16 such switches. Whenever you press a switch the corresponding row and column will be connected, this is electrical property of the keyboard.

Now, the computer keyboard that you see may look like a little longer, but inside if you see the layout the keys are again laid out logically in a matrix form. As we will see later, the number of interconnection lines gets reduced if you have a 2 dimension matrix realization.

(Refer Slide Time: 04:07)



Let us come into the problem of interfacing of switches. Let us start with the simplest scenario where we do not have matrix kind of a layout. We have a set of 8 switches.

Let us see how we have connected them. There are 8 switches which are shown here. One end of the switches is connected to ground, and the other end is connected to the lines of an input port. This is an input port of the computer and the switch can be regarded as an input device. Whenever you are pressing a switch the state of the switch will be coming as an input data.

Now, you see here whenever you press a switch because it is connected to ground that line will definitely become 0. But if you do not press it is open then this line will be floating. So, you really do not know what voltage will be coming here. So, to be sure that you are having some voltage here you connect some resistances on all the lines, this is these are called pull up resistances. And they are connected to a positive voltage, which can indicate logic 1.

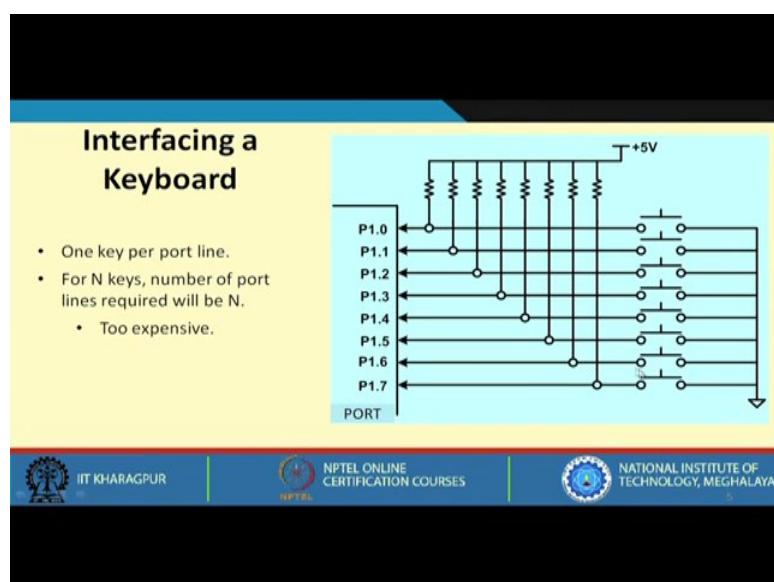
So, now what will happen if the keys not pressed, then this V_{cc} will come and appear here this will be treated as a 1. So, if I have not pressed any keys and if I read this input port all these lines will be, 1, 1, 1, 1, 1, 1, 1, 1, but if any of the key is pressed the corresponding input will become 0.

Let us see how we can check this status for asynchronous transfer. Because you recall for asynchronous data transfer or hand shaking we mention that the CPU will be continuously checking the status of the device, are you ready, are you ready? If it is not it will go back and check again. When the device is ready, then only it will come out and read the actual data.

So, here checking for this status is very easy. If you have not pressed any key, which means there is no input; if you read from the port you will get all 1's. In hexadecimal it will be FF. So, if it is FF it means that no key has been pressed there is no input, you go back and continue checking. But as soon as you see that it is not FF, which means one of the switches is 0, then you come out. Then first thing you will have to check which of the bits is 0, that you can easily check there are shift instructions there are bitwise and or operations you can check the bits which bits 0, if the bit number 0 is 0 then you do something, bit number 1 is 0 you do something else depending on what those indicate.

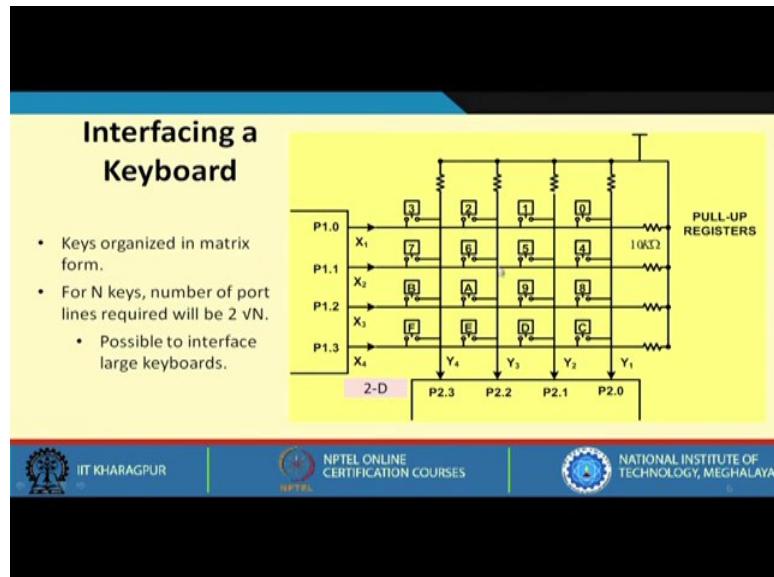
If you connect switches like this it is very simple. But there is one problem. Here you see we have 8 switches. So, one 8 bit port you can connect them directly. Now I tell you that well I have hundred switches. Where do you connect them? You will be needing hundred lines of ports, large number of ports, not only that -- hundred connections. As the number of switches increases the complexity of connections and the number of ports will also increase, this is one drawback.

(Refer Slide Time: 08:14)



This is too expensive to use and to lay out, that is why people do not use although this is very simple -- the way you check whether a key is pressed or not is very easy, but because of the complexity of wiring people do not use this.

(Refer Slide Time: 09:16)



We use a matrix keyboard, just the kind of keyboard I mentioned. Here I have shown an example of a 4×4 matrix. There are 4 row lines, there are 4 column lines, there are these are small switches connected between a row and a column.

Once you connect this 16 switches in a 4×4 matrix, you do not require 16 wires. You need 4 wires in the rows and 4 wires in the columns. And as you can see that there are resistances to a positive voltage the pull-up registers connected to the rows as well as columns.

Let us see how it works. The first thing is that, number of wires getting reduced. So, for n number of keys if you can lay them out in a perfect square if n is a square let us say 64 keys, then you can layout as 8×8 . So, we will be needing 8 wires on this side and 8 wires on this side. This is twice square root of n . For a 64 keys you need only 8×8 , 16 port lines, not more than that.

(Refer Slide Time: 11:21)

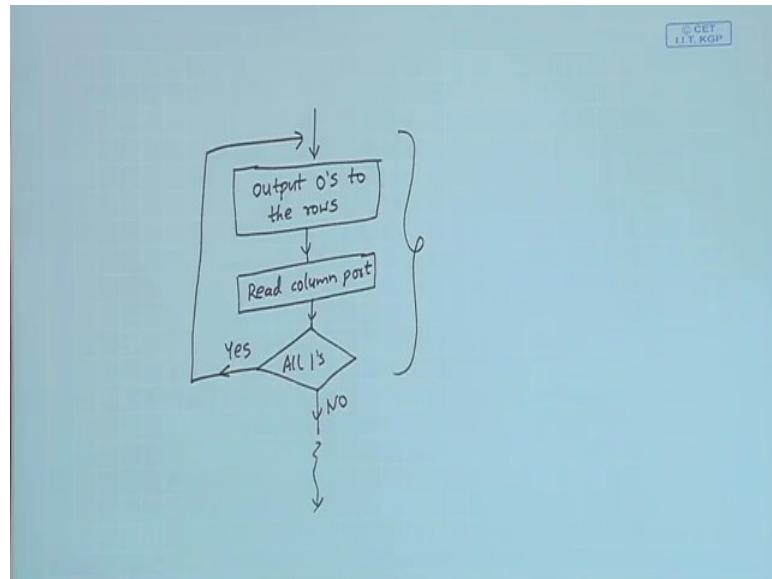
- How to detect the status of the device (i.e. whether any key has been pressed)?
 - Output all-0's to the rows.
 - Read the column port, and check whether all the bits are 1.
 - If any of the bits is 0, it means a key has been pressed.
- Allows asynchronous mode of transfer.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let see how we can check whether the device is ready to transfer; that means, some key is pressed or not. How we can check that? Well, there is a simple way -- you output all zeros to the rows then read the column port and check whether all the bits are 1, if any of the bits is 0, it means a key has been pressed.

Look at this diagram again. You output 0 0 0 0 on this port line. Now suppose the keys are not pressed. The column lines will all be connected to the plus voltage. So, they will get 1, 1, 1, 1. Now suppose this key A is pressed. So, this row and this column will get connected. So, this was 0 this 0 will propagate and this bit will be 0. So, if I read and if I see that this is not 1 1 1 1, which means at least one of the key has been pressed.

(Refer Slide Time: 13:01)



I will show in the flow chart of the code; output, 0's to the rows. Read column port then you check here whatever you reading are they all 1's? If you see it is all 1's, it means none of the keys pressed. So, in that case we have to go back and check again. And if any of the bits are 0 some keys pressed, you can go ahead to the next step.

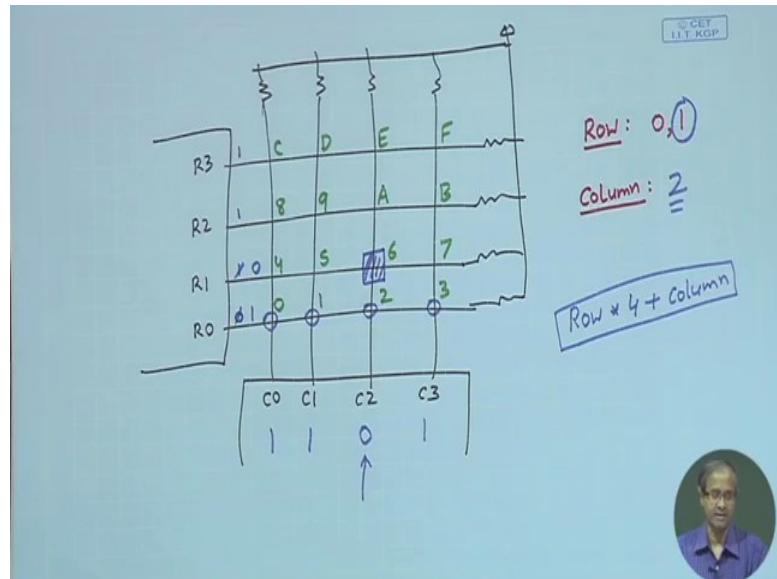
(Refer Slide Time: 14:14)

- How to detect which key has been pressed?
 - Requires a process called keyboard scanning.
 - One of the rows is made 0 at a time, and the column bits are checked.
 - We basically check whether some key in that particular row has been pressed.
- We find out both the row number and the column number of the key that has been pressed.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

The next question is, how to determine exactly which key has been pressed? For this purpose we need to do something called keyboard scanning. Here one of the rows is made 0 at a time and the column bits checked, like I will illustrate with an example.

(Refer Slide Time: 14:47)



Suppose I have 4 rows and 4 columns. There is one port connected on this side, one port connected on this side. And there are pull-up resistances. They are all connected to the positive supply voltage.

Now, I want to check whether a particular key is pressed or not. I can identify some row numbers and column, let say let us follow convention this is my row number 0, row 1 row 2 and row 3. And this is my column 0, column 1, column 2 and column 3.

What I do is as follows. I keep 2 counters, one I call as the row counter, and another the column counter. Initially I said the row counter to 0. I activate row 0. Activate row 0 means, I output 0 on this row, but all others are 1. Then I read the columns and see that whether any of the bit is 0. If I see any of the bit is 0 it will mean that one of the keys which are connected to the row number 0 must have been pressed, because of that this 0 is coming here.

But if I see it is all 1 1 1 1 which means in row 0 no key has been pressed, then what we do? Then we increment row to 1 and activate the next row. So, now, we make this as 0 and make the others as 1. So, now, you do the same thing, you check whether the columns any of these are 0 or not. Suppose I had pressed this key, this key was pressed. So, in row 1 it will get detected. So, I am reading something which will be 1 1 0 1, I get this data, because this was being shorted to 0.

So, now I can identify that there has been a key pressed in row number 1. Now in whatever I have received I will check which bit is 0. I find bit number 2 is 0. So, I set column number to 2. So, I have got row number and column number. So, I can define a key code like this (row \times 4 + column). This is typically done in a 4 \times 4 keyboard. So, how the keys will be mapped? Row number is 0, 0 multiplied by 4 plus column. So, 0 plus 0 this will be 0 1 2 3; then row 1, 1 into 4 plus column. This will be 4 5 6 7 8 9 10 in hexadecimal it is A and then 11, it is B 12 13 14 and 15.

Once you identified both the rows and columns, then we can identify which key has been pressed. But before you start this process you have to do this. You have to first check whether a key has been pressed or not. If the key has been pressed, then you move on to the keyboard scanning. This is actually how the thing works.

(Refer Slide Time: 19:29)

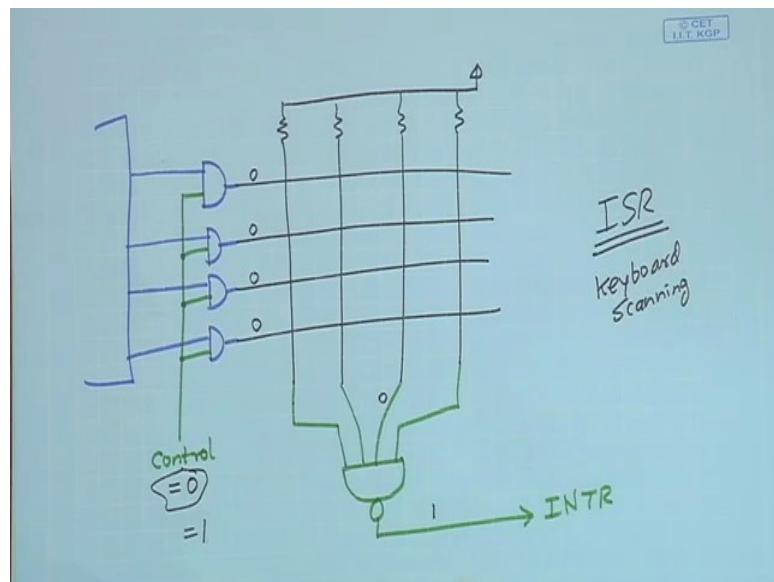
The slide has a yellow background with black text. At the top, there is a decorative bar consisting of a black triangle pointing right, a blue horizontal bar, and another black triangle pointing left. Below this is a solid black bar. The main content area is yellow with black text. It contains two bullet points:

- How to interface a keyboard in interrupt driven mode?
 - Normally all the rows of the keyboard are connected to ground; possibly through a set of AND gates with a control input that is made 0.
 - The column lines are connected to the inputs of a NAND gate, the output of which is connected to the INTR input.
 - The output of the NAND gate will become 1 whenever any key is pressed.
- Inside the ISR:
 - The control inputs of the AND gates are set to 1.
 - Normal keyboard scanning is carried out to identify the key pressed.

At the bottom of the slide, there is a footer bar with three logos and text: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

Now you think that you can make some small changes and make the keyboard interface work in an interrupt driven mode also. See in the earlier case what we saw is that just for checking whether I mean you are pressing any key or not, you are outputting all 0's and you are checking whether the column is all 1's or not.

(Refer Slide Time: 20:10)



Now, here I am again drawing the diagram, but I am not showing the switches and the resistances just the rows and columns I am showing 4 rows and 4 columns. So, what I am saying is that we will be connecting 4 AND gates and the output of the AND gates will be driving the rows. And the port will be driving one of the inputs of the AND gate. The other input of the AND gate will be made common, this we can call some kind of control. This will also be controlled by the CPU; this will be connected to some other output port.

So, this is a change you do, and the another change is that you use a NAND gate. And connect the columns to the NAND. And the output of this NAND you connect to the interrupt input of the processor. Just think what happens here. You recall these are all pulled up to positive voltage, these are all positive voltage, and when you are checking whether the key is pressed. So, the under normal conditions control will be set to 0.

Control 0 means what? One of the input of the AND gate will be 0. So, these lines will always be 0 0 0 0. Now suppose any of the key is pressed. Well, suppose it is not pressed. If it is not pressed, then all these lines will be 1 1 1 1, the output of the NAND will be 0. So, there is no interrupt, but if any keys is pressed because you are applying all 0 by default that particular bit will become 0, and correspondingly the output of the NAND gate will become 1 and an interrupt will be generated. The

processor will know that there is an interrupt; some key has been pressed; now it will jump to the ISR. And inside the ISR that keyboard scanning routine will be there.

When you are inside the ISR you are doing keyboard scanning, you will have to change the control to 1 during that time. Because now whatever you are sending over the port that should come to the rows. So, with this hardware modification you can have interrupt driven keyboard interface.

(Refer Slide Time: 23:42)

The slide has a black header bar at the top. Below it is a yellow section containing the title 'Printer Interfacing'. Underneath the title is a bulleted list of points:

- Older printers have serial and parallel ports for interfacing to computer systems.
 - RS-232C serial data interface.
 - LPT parallel data interface (8 data lines).
- Modern-day printers support the much higher speed Universal Serial Bus (USB) interface.
 - Almost all devices today have USB interfaces.

At the bottom of the slide, there are three logos with their respective names: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

Now, let us very briefly look at means a printer interfacing problem, how some of the I/O transfer method we discussed earlier can be used there. Well, if you think of the older printers which are available may be 5, 10 years back. They had either serial or a parallel interface it was called LPT. This parallel interface was almost universally used for printer interfacing.

Now, in the parallel interface there are 8 parallel data lines, but nowadays in the computers you will not see the serial ports and parallel ports anymore. They have become obsolete, but in the old computers you may still see find some serial ports RS 232c, and parallel LPT ports available.

Now, almost all printers today support, an interface called universal serial bus or USB. Now not only printers USB has become really universal, almost all the devices today,

they have USB interfaces including our mobile phones, cameras, mouse, keyboard everything.

(Refer Slide Time: 25:22)

- The LPT port used a 25-pin connector:
 - 8 data lines
 - STROBE
 - BUSY
 - ACK
- After sending the data, the CPU activates the STROBE input to inform the printer that data is ready.
 - The printer will activate BUSY and start printing; once done, it will send back ACK to the CPU.
- The interface allows asynchronous data transfer using handshaking.

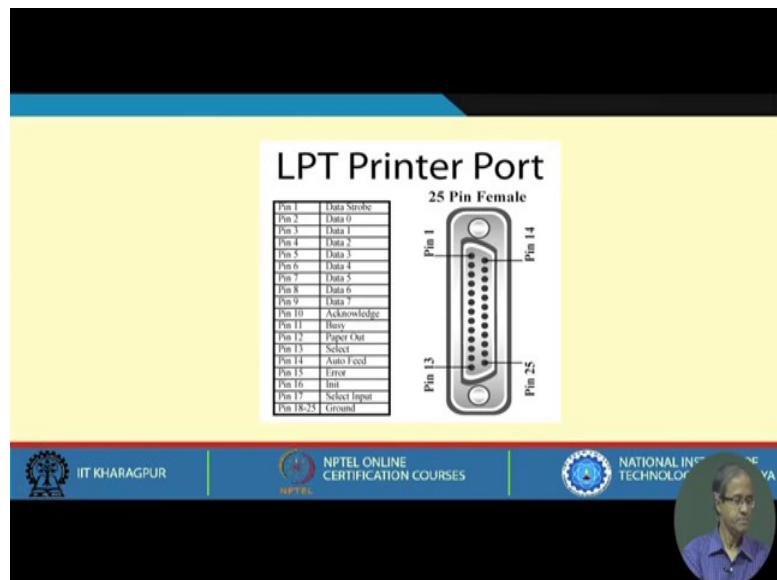
 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

This LPT port used to have a 25-pin connector. All the 25 pins are not used of course, there are 8 data lines. There were power supply lines, positive supply, negative supply, and a ground. And there are 3 interesting signal lines: a strobe, a busy, and an acknowledgement, using which you can carry out asynchronous data transfer with handshaking.

Well how it was done? Well, the printer is an output device. CPU will be writing some data to the printer right. It could send one byte at a time. CPU will activate the strobe input indicating to the printer that well I have sent one byte of data you please read it. The printer can read 8 bits of data, and during the reading the printer will activate the busy signal, that is going back to the CPU.

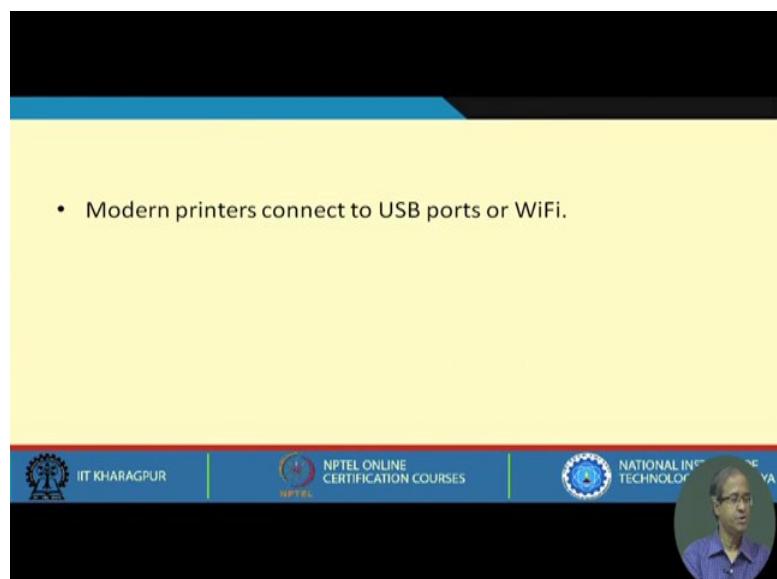
Now, after the printer is done, and it is ready to accept the next data, the printer will be sending back the ack signal. Just using the signal you can see here you can have handshaking you can have asynchronous data transfer. There are some additional signals like busy that was specifically printer specific, which of course, in a normal handshaking acknowledgement is enough, but this strobe and busy are required because this allows 2 way handshaking, because CPU knows that the printer is ready printer will also know that the CPU has sent a data, both ways.

(Refer Slide Time: 27:23)



The printer port actually looks like this. You see 25 pins -- you see pin number 1 is a strobe then there are 8 lines, then there is an acknowledgement, there is a busy. And there are some other pins which are very printer specific like there is one pin indicates paper out, if the printer is out of paper this pin will be active, then auto feed if auto feed is active this pin is active. If there is some error in the printer this pin is active and there is some ground.

(Refer Slide Time: 28:08)



So, this was interface which was quite popular in the early days, but as I had said today if you look at the printers, they have either USB ports or the more sophisticated ones they have Wi-Fi connection. They can connect to the wireless and from a computer system you can print wirelessly. What actually I have tried to show through these 2 examples is that well any kind of devices you see, will have some built in features or you will have to find out a way to do it, to carry out some kind of handshaking on asynchronous kind of transfers.

Now, you see the computer keyboards, now you may wonder where is the matrix who does this scanning, another thing? Well, if you break open the keyboard some day you will see inside there is some circuits. There is a processor or the CPU sitting inside your keyboard only, and it is that processor which is doing this keyboard scanning. It is not the CPU of your computer who is doing this, the processor sitting inside the keyboard is doing the scanning and whenever a new key is pressed, it will be sending an interrupt signal to our main processor. The main processor will know that well a new key has been pressed.

You see newer technologies and newer standards are evolving, one of the main objective is to make the interconnection simpler. Earlier the parallel ports carried 25 watts, but now the USB connections if you see that there are only few connections few wires, not many we will be looking into the details of the USB interface later with the next lecture. But here our objective was to look at a couple of examples, but if you take any other device any other example, you will see that some similar features or facilities are there.

With this we come to the end of this lecture. In the next lecture we shall be working out some examples following which we shall be looking into some standard bus interfaces like USB.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No – 50
Exercises On IO Transfer

In this lecture we shall not be learning anything new, rather I shall be taking some exercises on input output transfer and try to work them out with you. I think if you work out some exercises, you will have a better understanding regarding some of the concepts which we have already discussed during the last few lectures.

(Refer Slide Time: 00:45)



This lecture is titled exercises on IO transfer.

Let us look at the exercises one by one and let us try to solve them.

(Refer Slide Time: 00:54)

Example 1

- Suppose we want to read 2048 bytes in programmed I/O mode of transfer. The bus width is 32 bits. Each time an interrupt occurs, it takes 4 μ sec to service it (i.e. transfer 32 bits). How much CPU time is required to read 2048 bytes?

↳

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Just look at the first example. This says that we want to read 2048 bytes in programmed IO mode of transfer, the bus width is 32 bits. This means that we can transfer 4 bytes or 32 bits each time an interrupt occurs. It takes 4 microsecond to service it.

So, to transfer four bytes, we need 4 microsecond. The question is how much CPU time is required to read all 2048 bytes. It is fairly simple because it is already mentioned that for interrupt processing, you need 4 microsecond and what you are doing is, you are transferring 32 bits.

(Refer Slide Time: 02:03)

Transfer 4 bytes in 4 μ sec
∴ " " " " | "
∴ " 2048 " " 2048 μ s
= 2.048 ms

IIS GET
IIT KGP

You can say that we are transferring 4 bytes in 4 microseconds. Therefore, we will transfer 1 byte in $4 / 4 = 1$ microsecond. Our problem is to transfer 2048 bytes. So, it will be $2048 \text{ microseconds} = 2 \text{ milliseconds}$.

So, let us move on to next problem.

(Refer Slide Time: 02:52)

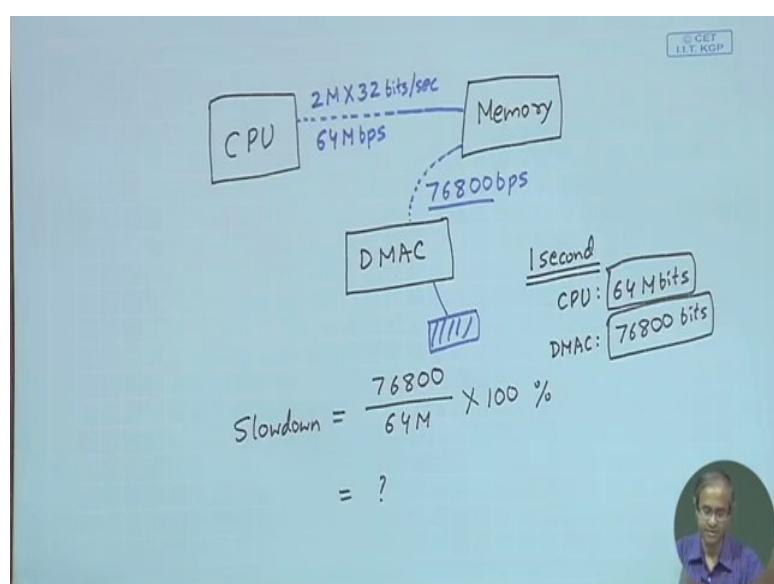
Example 2

- A DMA module is transferring bytes to main memory from an external device at 76800 bps. The CPU can fetch instructions at a rate of 2 million instructions per second. Assume instruction size is 32 bits. How much will the processor be slowed down due to DMA activity?

The slide footer includes logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Rourkela, along with a video thumbnail of a professor.

Here there is a DMA module. The DMA module is transferring bytes to main memory from an external device, where the data transfer rate is given. Let us show pictorially whatever we are reading out.

(Refer Slide Time: 03:24)



Iff you recall in this schematic, there are three part is involved, i.e CPU, memory and DMA controller. The problem statements says that DMA module, there can be some device connected, some IO device is connected to DMAC. That device is trying to transfer bytes to main memory. It will try to do this and the speed is specified as 76800 bits per second. This is the required IO transfer rate.

Now, the problem says again that the CPU can fetch instructions at a rate of 2 million instructions per second. You assume that instruction size is 32 bits. What does this mean? If you look at CPU memory connection, we are saying that 2 million instructions which means $2 \text{ million} \times 32 \text{ bits per instruction}$, so many bits per second.

So, these many bits per second is what IO is trying to transfer and these many bits per second is what CPU memory connection actually can go up to $2 \times 32 = 64$ megabits per second. The question is how much will the processor be slowed down due to DMA activity.

If you look at a one second window, the CPU normally was transferring 64 megabits of data and the DMA controller is transferring data at 76800 bits per second. So, what will be the processor slow down? The calculation is shown.

Let us move on to the third problem.

(Refer Slide Time: 08:10)

Example 3

- A DMA controller transfers 32-bit words to memory using cycle stealing. The words are assembled from a device that transmits bytes at a rate of 2400 bytes per second. The CPU is fetching and executing instructions at an average rate of 1 million instructions per second. By how much time will the CPU be slowed down because of the DMA transfer?

IIT Kharagpur | **NPTEL** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

The third problem is a similar problem using DMA controller. This says DMA controller transfers 32 bit words to memory using cycle stealing and the device from where the data transferred, they transfer bytes at a rate of 2400 bytes per second. This is exactly similar to the previous problem.

The DMA controller transferring data in 32 bits of word. The words are coming at the speed 2400 bytes per second and the CPU is fetching and executing instructions, one million instruction, let say one instruction is one word. The calculation is shown.

(Refer Slide Time : 09:21)

A handwritten calculation on a light blue background. At the top right is a logo for "CET IIT KGP". The text "1 second" is written above a horizontal line. Below the line, "CPU : 1 M words" is written. To the right of this, "DMA : $\frac{2400}{4} = 600 \text{ words}$ " is written. Below these, the formula "Slowdown = $\frac{600}{1M} \times 100\% = ?$ " is written. A hand is visible at the bottom right corner of the page.

The CPU is transmitting one million instruction per second. So, CPU will transmit one million words and the IO device is transferring 2400 bytes which means it is transferring 4 bytes at a time. So, it will be 600 words. It will be $2400 / 4 = 600$ words. So, you can calculate how much is the slowdown. It will be a small fraction of 1%.

Let us look at the next problem.

(Refer Slide Time : 10:44)

Example 4

- Consider a system employing interrupt-driven I/O for a device that transfers data at 8 KB/s on a continuous basis. The interrupt processing takes about 100 μ sec and the I/O device interrupts the CPU for every byte. While executing the ISR, the processor takes about 8 μ sec for the transfer of each byte. What is the fraction of CPU time consumed by the I/O device?

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Here, we have a problem of interrupt. This problem says, there is a system which employs interrupt driven IO for a device that transfers data at 8 kilobytes per second on a continuous basis. The device is continually transferring data and the interrupt processing takes 100 microseconds. For every interrupt, the CPU will be spending 100 microseconds as overhead for saving status, restoring status, jumping to ISR, everything taken together.

Another thing is mentioned here is that IO device will interrupt the CPU for every byte. There are two things. There is a device which is transferring data at 8 kilobytes per second and for every byte the device will interrupt the CPU, and for the interrupt, the overhead is 100 microseconds per interrupt processing and 8 microsecond for actual transfer of each byte. So, it will be 108.

You can easily calculate here the overhead.

(Refer Slide Time : 12:15)

Overhead for every byte = $100 \text{ msec} + 8 \text{ msec}$
= 108 msec

1 second
8 KB \Rightarrow Total overhead = $8000 \times 108 \text{ msec}$
 $\approx 800,000 \text{ msec}$
 $= 800 \text{ msec} = \underline{\underline{0.8 \text{ sec.}}}$

\therefore CPU time consumed = $\frac{0.8}{1} \times 100 = 80\%$



For every byte transfer, the overhead will be 100 microseconds + 8 microsecond. It is 108. Now, the device is transmitting data at the rate of 8 kilobytes per second on a continuous basis. So, the question is what is the fraction of CPU time consumed by IO device?

Let's take the 1 second window again. In 1 second, the total data that is being sent is 8 kilobytes. This 8 k means let say let call it 8000 for simplicity of calculation. Actually it will be 8192, i.e. 8000 into 100 and 8, so many microseconds. It will be 8 lakh microseconds, which means 800 microseconds or 0.8 seconds.

You see out of 1 second 0.8 second is spent as overhead. You are left with only 0.2 second for useful work. If you multiply by 100, you will be getting the percentage which is 80 %. So, you see about 80 % of CPU time is getting wasted here. This is a characteristic of interrupt driven I/O, where the data rate is very fast and you have to do interrupt processing very frequently, and the overwrite can be very high.

Let us move on to the next problem.

(Refer Slide Time : 14:50)

Example 5

- Consider a disk drive with 16 surfaces, 512 tracks per surface, and 512 sectors per track, 1024 bytes per sector, and a rotation speed of 3600 rpm. The disk is operated in cycle stealing mode whereby whenever one 4-byte word is ready, it is sent to memory. Similarly for writing, the disk interface reads a 4-byte word from memory in each DMA cycle. The memory cycle time is 40 nsec. Find the maximum percentage of time that the CPU gets blocked during DMA operation.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY WARANGAL

This is a problem that involves a disk and DMA. We have a disk drive with 16 surfaces. There are 16 surfaces. That means 8 platters, there are 512 tracks per surface.

(Refer Slide Time : 15:23)

16 surfaces
512 tracks /surface
512 sectors /track
1024 bytes /sector
3600 rpm
Every DMA cycle — 4 bytes
Memory cycle time — 40 nsec
Assume: one track data is transferred at a time
Capacity of track = 512 kB
In $\frac{1}{60} \text{ sec}$, transmit 512 kB
∴ $1 \text{ sec} = 512 \times 60 \text{ kB} = 30 \text{ MB}$
 $\% \text{ Blocked} = \frac{30}{100} \times 100 = 30\%$

3600 — 60 sec
 $\therefore 1 = \frac{1}{60} \text{ sec.}$

40 nsec — 4B
 $\therefore 1 \text{ sec} = \frac{4}{40} \text{ sec} \approx 0.1 \text{ sec} = 100 \text{ MB}$

Then, there is 512 sectors per track and 1024 bytes per sector. These are some data which is specified and of course, it is specified that the rotation speed is 3600 revolutions per minute. The disk rotates at this speed.

Now, the question is the disk is operated in cycle stealing mode whereby whenever 4 byte word is ready, it is sent to memory. Similarly, for writing the disk interface reads 4

byte word from memory in each DMA cycle. What it actually says is that in every DMA cycle, we transfer 4 bytes of data. Another thing is mentioned that the memory cycle time is 40 nanoseconds.

The question is what is the maximum percentage of time the CPU gets blocked during DMA operation? Assume that data from one track can be transferred at a time. Let us assume although they are 16 surfaces, we assume that one track data is transferred at a time.

Let us see what is the capacity of a track. In a track, there are 512 sectors and each sector, it is 1 kilobyte. So, it will be 512 kilobytes. This is the capacity of a track and let us see what is the rotational speed. It is 3600 rpm. To make a calculation like this 3600 rotations in 1 minute, let say we do it in seconds, in 60 seconds. Therefore, one rotation in $1 / 60$ seconds.

When the disk is rotating once, this much data will be transmitted. It is 512 kilobyte in $1 / 60$ second. So, you can say in $1 / 60$ second, you transmit 512 kilobytes of data. Therefore, in one second you will be transmitting $512 * 60$ KB. If you multiply by 2, it becomes 30 megabytes. So, in one second, you are transmitting 30 megabytes.

Let us now look at the memory. We have only looked at the disk. Memory cycle time is 40 nanosecond. In 40 nanosecond, we can read or write one word or 4 bytes. We say that in that 40 nanosecond, you can transfer 4 bytes.

Let us see in one second how much data can memory support? It is $4 / 40$ nano = 0.1 gig which means 100MB in one second. The memory can support 100 megabytes of data transfer and your disk sending data at 30 megabytes per second. So, percentage blocked will be 30 percent. It is actually 30 by 100 multiplied by 130 percent. This is how you calculate.

Let us move on to the next problem.

(Refer Slide Time : 21:50)

Example 6

- A hard disk is connected to a 50 MHz processor through a DMA controller. Assume that the initial set-up time for a DMA transfer takes 2000 clock cycles for the processor, and also assume that the handling of the interrupt on DMA completion requires 1000 clock cycles for the processor. The hard disk has a transfer rate of 4000 KB/s and average block size transferred is 8 KB. What fraction of the processor time is consumed by the disk, assuming that data are transferred only during the idle cycles of the CPU?

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

This problem says that this hard disk is again connected to 50 megahertz processor.

(Refer Slide Time : 22:07)

50 MHz \Rightarrow CCT = 20 nsec (CET IIT-KGP)

$$\frac{1}{50 \times 10^6} \approx 20 \times 10^{-9}$$

DMA setup: 2000 clocks = $2000 \times 20 \text{ nsec}$
= 40 μsec

DMA completion: 1000 clocks = $1000 \times 20 \text{ nsec}$
= 20 μsec

4000 kB is transferred in 1 sec
 $\therefore 1 \text{ kB } " " " \frac{1}{4000} \text{ sec} = 2 \text{ msec}$
 $\therefore 8 \text{ kB } " " " \frac{8}{4000} \text{ sec} = 2 \text{ msec}$

\therefore Total time for 8 kB = $60 \mu\text{s} + 2 \text{ msec}$
= $2.060 \mu\text{sec}$ || overhead =

50 megahertz means the CCT will be 1 by 50 or 20 nanoseconds. Now, it says that the initial setup time for DMA transfer takes 2000 clock cycles and handling of interrupts on DMA completion requires 1000 clock cycles. What does this mean? We have already calculated clock cycle time. For DMA transfer it is 2000×20 nanoseconds = 40 microseconds, and DMA completion is 1000×20 = 20 microseconds.

Now, it says that the hard disk has a transfer rate of 4000 kilobytes per second and the average block size transferred is 8 kilobytes. For DMA setup and DMA completion, you have the overheads. You can see for initialization, you need 40 microseconds. At the end, you need 20 microseconds, but after that you are transferring data whose block size is 8 kilobytes, where the transfer rate is 4000 kilobytes per second. So, you can calculate like this 4000 kilobytes is transferred in one second.

Now, you calculate one block of 8 kilobytes which is mentioned so how much time will take for that block to get transferred. Average block size is 8 kilobytes. 8 kilobyte will be transferred in $8 / 4000$ seconds = 2 milliseconds. For this entire data transfer, what you have? We have DMA setup of 40 microsecond. The actual block transfer takes 2 milliseconds and DMA completion 20 microsecond. Therefore, total time is 2.060 microseconds.

The question is what fraction of the processor time is consumed by the disk? Assuming that the data transfers only during the idle cycles of CPU, we assume that when this data transferred, only the idle cycles are being utilized. You can actually find out total 8 kilobytes is being transmitted in this much amount of time. I will leave it an exercise for you.

In this way we have calculated how much time will be taken for the block transfer of 8 kilobytes. Now, that block transfer is done, it is assumed that it is done in cycle stealing mode and CPU is not disturbed, and this is the total time that is taken for that.

Now, the last problem let us move on to.

(Refer Slide Time : 27:55)

Example 7

- A device with transfer rate of 20 KB/s is connected to a CPU. Data is transferred byte wise. Let the interrupt overhead be 6 μ sec. The byte transfer time between the device interface register and CPU or memory is negligible. What is the minimum performance gain of operating the device under interrupt-driven mode?

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

A device with transfer rate of 20 kilobytes per second is connected to CPU. The interrupt overhead is assumed to be 6 microsecond. The byte transfer time between device interface register and CPU or memory is negligible.

You see data is transferred byte wise. You have a transfer rate of 20 kilobytes per second.

(Refer Slide Time : 28:34)

20 kB/sec
Interrupt overhead = 6 μ sec

1 second
$$\frac{20 \text{ kB}}{20 \text{ kB} + 6 \mu\text{sec}} = 120 \text{ msec}$$

Perf. Gain:
$$\frac{1000 - 120}{1000} \times 100 \%$$



It is said that the interrupt overhead is 6 microsecond and you are transferring byte wise. So, every byte there will be an interrupt. You look at a 1 second window again. In 1 second window, total 20 kilobytes will be transmitted and for every byte there will be an overhead of 6 microsecond. The total overhead will be $20k \times 6$ microsecond = 120 milliseconds.

The question is what is the minimum performance gain? Performance gain you can calculate like this that initially the time was one second, total time was 1000. Out of that 120 is being just wasted as an overhead divide by 1000, but if you do not use interrupt driven, your entire time would have been wasted. So, this will be your performance gain. If you multiply it by 100, you will get the resulting percentage. This is your performance gain.

In this lecture we worked out some problems on I/O interfacing. Most of this problems are of the standard of GATE examination that you may be aware of. In the next lectures, we shall be starting our discussions on some common I/O bus standards like USB for example. What are the silent features there, how they are used, what are the peak speeds, advantages and so on and so forth.

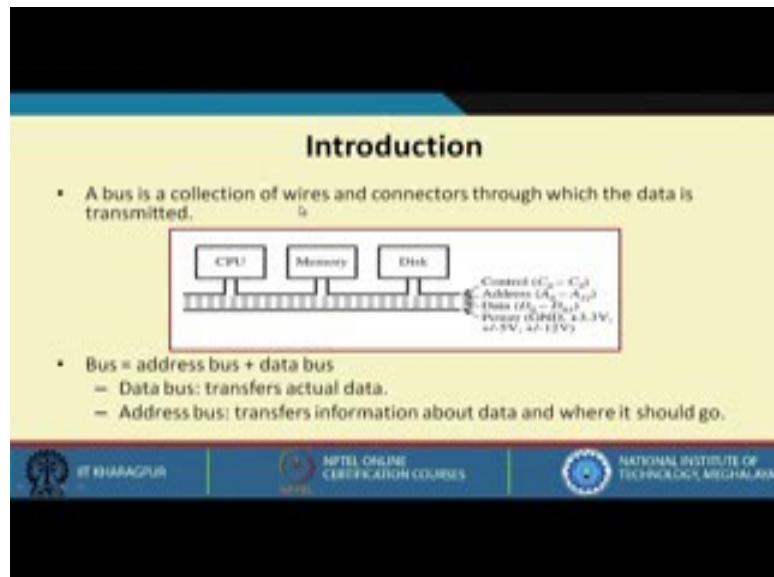
Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 51
Bus Standards

In this lecture we shall be discussing on some of the bus standards and protocols which are used inside a computer system, not necessarily only for interfacing the peripheral devices. We shall be looking at the overall picture first, and then we shall be looking at one very popularly used bus standard USB.

(Refer Slide Time: 00:52)



The topic of today's lecture is bus standards. Let us start by defining a bus. A bus roughly refers to a common shared path. Now, in a city when you board a bus, you see the bus takes a group of people from one point to the other. So, it is like a common shared path. Many people are using the same communication facility, unlike a private car that is dedicated to a person. A bus means a common shared path that can be used by more than one entities for communicating between two end points.

Basically the bus is a collection of wires. As this diagram shows, you can have your CPU, your memory, and some secondary memory like disk. This is a simplified diagram and this is your bus. This will be a collection of wires and connectors through this bus data will flow.

Now, in this bus what are the basic things that should be there? There should be the address lines. When CPU is sending some data, it will also mention to whom that data is meant because there can be multiple devices on the bus. How many data can be sent at a time depends on the width of the bus. How many wires are there, then there can be some control signals and the control will tell you exactly what kind of operation CPU is trying to do. Whether CPU is trying to read the status of the device or wants to send some data or receive some data and so on. And in addition, there can be a set of power supply lines, various voltages, ground, so that some of the devices that are connected can directly draw power from those lines.

(Refer Slide Time: 03:24)

- **Bus Protocol:**
 - Rules determining the format and transmission of data through bus.
- **Parallel Bus:**
 - Data transmitted in parallel.
 - Advantage: It is fast.
 - Disadvantage: High cost for long distance communication, inter-line interference at high frequency.
- **Serial Bus:**
 - Data transmitted serially.
 - Advantage: low cost for long distance communication, no interference.
 - Disadvantage: Slow.

Let us look at some of the terminologies. When you say bus protocol, what does it mean? If you look at the dictionary meaning, protocol is a set of rules or conventions which both the end systems should comply or follow, so that faithful communication can take place. Similarly, in a bus or for any communication system, there has to be a protocol. Both the end systems should be following the same set of rules, so that data communication can take place without any problem.

So, bus protocol essentially constitutes the rules that determine the format and transmission of data through the bus. Now, the bus can be either parallel or serial. For a parallel bus, the data are transmitted in parallel means, I can have 8 or 16 or 32 parallel lines in the bus through which data can be transmitted multiple bits per cycle.

The advantage here you can see is obvious because there are so many lines with which data can flow. It is fast, but the disadvantage is that your cable or the bus will be very thick. There will be so many wires. So, for a long distance communication, the length of the cable will be large, the cost of the cable will also be higher and because there are large number of lines that are stacked in the same cable, there can be inter-line interference, particularly when data communication is taking place at higher frequencies, and because of this, parallel bus cannot be used for longer distances.

In fact, you see that most of the communication that we carry out today is based on some kind of serial bus. In our earlier systems like that printer port, I talked about some parallel bus standards were there, but today you will find very few.

In contrast the serial bus has a single line for sending and a single line for receiving. So, when you have a single line for sending, then you can have the serial communication protocol I talked about earlier using start bits and stop bits to synchronize between the sender and receiver. That kind of a format can be used. Well, there are other ways also. Some synchronous kind of communication can also be done, where the sender and receiver knows the exact speed of data transfer. Those are also possible.

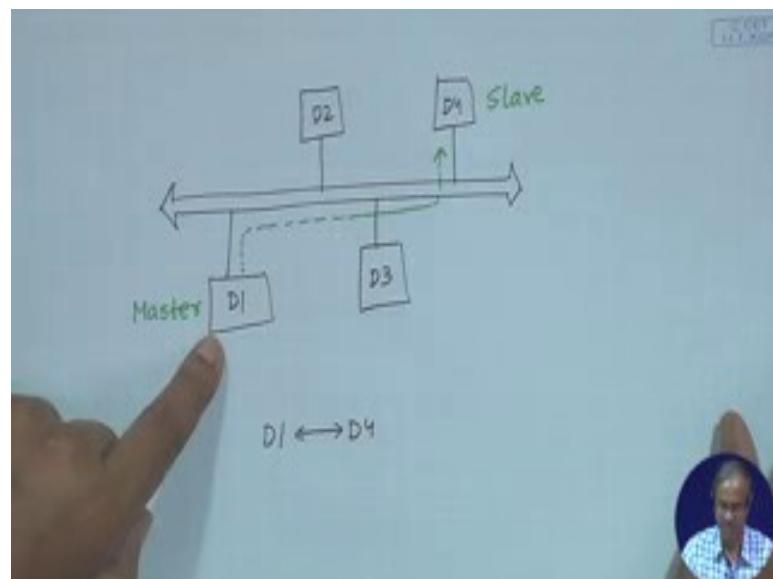
Because the number of lines are very few, the advantage is that the cost of the cable will be very low, and so for long distance communication, it will be very suitable, and because there is a very few lines, interference will be very less. The disadvantage is that because you are sending the data serially bit by bit, it will be relatively slow as compared to the parallel bus.

(Refer Slide Time: 06:53)

- Bus Master and Slaves:
 - The device that controls the bus is called master; others are slaves.
- Local or System Bus:
 - Bus that connects CPU and memory.
- Front-Side Bus:
 - Original concept: connects CPU to components.
 - Modern Intel architecture: connects CPU to NorthBridge chipset.
- Back-Side Bus:
 - Connects CPU to L2 cache.
- Memory Bus:
 - Connects NorthBridge chipset to memory.

In a bus as it is said there can be many nodes or devices connected to it.

(Refer Slide Time: 07:03)



Let us say here we have a bus and we denote the bus like this. There can be several devices that are connected to the bus. The devices may not all be of the same type, they can be different. One can be more powerful, one can be very simple. Just I am calling them D1, D2, D3 and D4.

Now, by calling the bus master and slave what I mean to say is that suppose D1 and D4 want to communicate, now depending on the bus protocol and the capabilities of the

device interfaces, there can be a scenario like this where this device D1 will be designated as the master and D4 will be designated as the slave. What does this mean? D1 being the master will be able to initiate the data transmission, but D4 on its own cannot do so because it is the slave. The master is always responsible for initiating every data transfer over the bus.

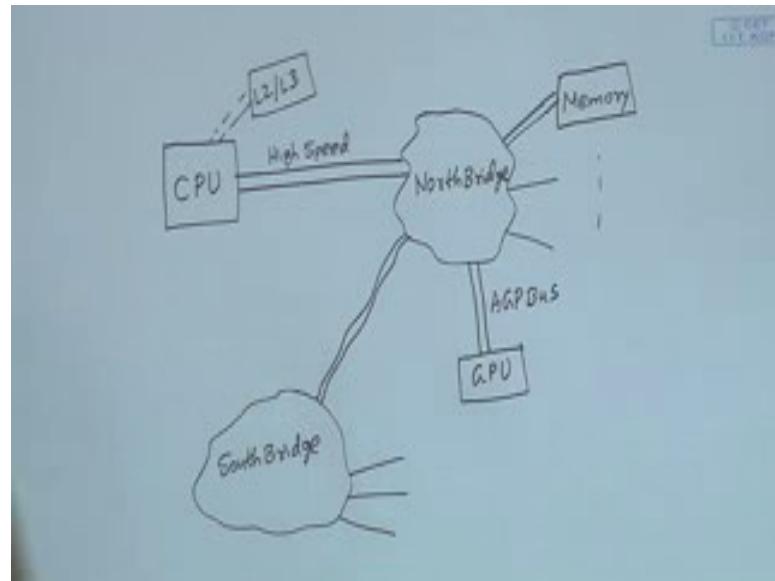
There are many protocols where there is a master-slave relationship between the various devices and it is the master which takes care or initiates the communication with the other devices which are slaves.

Now, we are talking about a bus or the buses which are there inside a computer system, if you just open your desktop or a laptop, you will see a lot of circuitry inside, but schematically or architecturally how does the devices get connected inside there say on the mother board. Here we are talking about that.

Local or system bus, this is a term we used to refer to a bus that connects CPU and memory. Now, you know that in a computer system, the CPU and the memory are the fastest two components and the communication between CPU and memory is the most crucial in determining the overall performance of the system. This bus that connects the CPU and the memory is referred to as the local bus or the system bus.

Following the nomenclature of Intel, we can distinguish the buses as front side or back side. Front side bus means the bus that connects CPU to the other components. This was the original concept, but if you look at the modern mother boards, front side bus refers to a bus that connects CPU to the north bridge chipset.

(Refer Slide Time: 10:30)

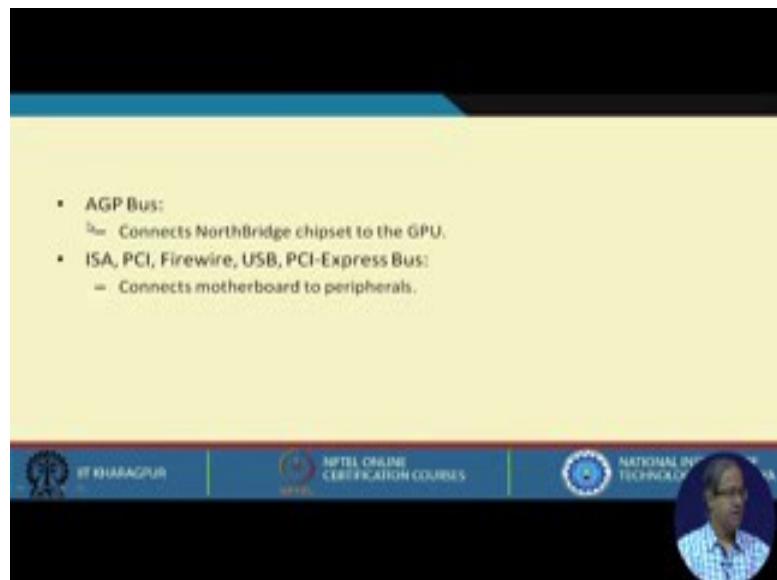


Here we are talking about a connection like this. There is CPU, there is something which is called North Bridge and the CPU is connected to the north bridge by a bus which is a high-speed bus and North Bridge may be connected to various high speed devices like memory.

There can be a backside bus that can connect CPU to the level 2 cache. If it is outside the processor or level 3 cache, there can be a memory bus which connects north bridge chipset to the memory. So, here this link I talked about North Bridge is connected to the memory.

There are different kinds of devices that are connected there. There can be a backside bus also where the CPU is connected to L2 or L3 cache. That bus is also there depending on whether L2 is inside the CPU or outside.

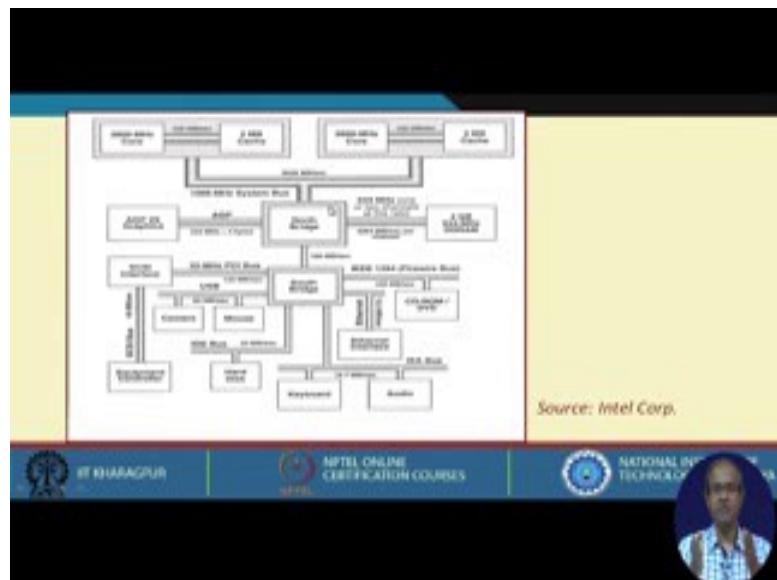
(Refer Slide Time: 11:58)



You have AGP bus, which refers to the bus through which the graphics cards are connected. Here the north bridge chipset is connected to GPU. So, here there is one connection with the north bridge that connects with the GPU unit. These buses refer to AGP bus, and in addition there are several other buses inside CPU. There is another device there which is called the South Bridge and that is also there. The north bridge is connected to the south bridge and the south bridge is relatively much slower in speed. South bridge typically connects devices that are not that high speed.

So, you can see there are so many buses inside the system and the devices are connected to one or more of these buses. Now, talking about some of the standards that evolved over the years, these are standards through which you can connect some peripherals to a computer system. It can be motherboard, it can be some connectors that are connected to the mother board. ISA, PCI, Firewire, PCI express and today you have USB, these are all examples of some bus standards that connect peripherals to the mother boards. In some way, they vary in the speeds and other capabilities.

(Refer Slide Time: 13:51)



Let us look at a typical architecture diagram of a mother board inside a modern day PC. At the top I am assuring that this is a dual core processor. There are two cores which are shown here. You see there is 3800 MHz core. The core is running at 3.8 GHz, there is another core and inside the core, inside the chip there is 2 megabyte cache. This is the IL1 cache and here you can see is the north bridge.

Chipset actually refers to the north bridge and the south bridge combination, what are its capabilities, what are its speeds and so on. A chipset will determine that. This north bridge and south bridge taken together, this is the chipset you can refer to.

You see north bridge how it is connected. In one side, it is connected to the core, the processors. This is the high speed bus. As you can see this runs at 1066 MHz. So, it is about 1 GHz speed and the data bandwidth is about 8528 megabytes per second. So, it is pretty fast. North Bridge connects to main memory via the memory bus. In this example memory bus is running at 533 MHz and it can transfer data 4264 megabytes per second. This is DDRAM at 533 MHz.

From the other side of the north bridge through AGP bus, you have the graphics card. Now, the AGP can be inside and North Bridge is connected to the south bridge through a relatively lower speed link. This is 100 megabytes per second.

Now, south bridge connects to the other devices like it can have IEEE 1394 interface that is sometimes called firewire. Of course, now firewire has become obsolete. In the modern computers, you do not see firewire any more. So, you can connect some devices like CD ROM, DVD through this. You can have a network adapter card internet interface that also can be connected through it. You can have the slow systems like keyboard and audio. This can have a much slower bandwidth. You can have a hard disk connected through either IDE interface or SCSI or here you can have USB interfaces, where you can connect various sorts of devices.

In the modern systems as the standards evolve, this USB standard has become more and more widely used and faster. There are USB versions that instead of being connected to the south bridge, they are connected directly to the north bridge because of their much higher speeds.

(Refer Slide Time: 18:12)

Some Features of a Bus

- **Bus width:**
 - Number of wires available in the bus for transferring data.
- **Bus bandwidth:**
 - Total amount of data that can be transferred over the bus per unit time.

IT KHARGPUR | NIT DURGAPUR | NIT MEGHALAYA

Talking about a bus, some of the characteristic features will be bus width meaning how many data lines are available, how many bits of data you can transfer in every cycle and the speed of the bus or the bandwidth. It is total amount of data in bits per second or bytes per second that can be transferred over the bus.

Now, let us look at these features, bus width and bandwidth for some of the commonly used bus standards.

(Refer Slide Time: 18:44)

Bus	Width (bit)	Bandwidth (MB/s)
16-bit ISA	16	15.9
EISA	32	31.8
PCI	32	127.2
64-bit PCI 2.1 (66 MHz)	64	508.6
AGP Rx	32	2,133
USB 2	1	Slow-Speed: 1.5 Mbit/s Full-Speed: 12 Mbit/s Hi-Speed: 480 Mbit/s
Firewire 400	1	400 Mbit/s
PCI-Express 16x, version 2	16	8,000

ST KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

You see the ISA bus was 16 bit bus which carried 16 watts and the bandwidth was 15.9 megabytes per second. This was extended to ISA EISA 32 bits 31.8 MHz. PCI was also 32 bits, but the speed was about four times. Then, 64 bit PCI came version 2.1 where the bus width was 64. So, the speed was again increased significantly.

The parallel buses exist inside a system within the mother board, but today when you connect an external peripheral with the computer because of the cost of the cabling, the data communication is mostly serial in nature, but inside the mother board as you can see, these buses they are all carrying 16, 32, 64 bits in parallel at a time.

For the graphics AGP bus of 32 bit, it runs much faster 2133 megabytes per second, but USB2, well this is a gold standard. There are newer standard we will see later. USB 3.1 here data communication is serial and the data communication speed can be in one of three mode; slow, full and high. In the high speed mode, it can go up to 480 megabits per second, but nowadays with a modern USB versions, you can work to several gigabits per second also.

Firewire was supposed to be one of the fast standards. This is also serial. This can go up to 400 megabits per second and PCI expresses one of the fastest buses available. It carries 16 wires and the speed is 8000 megabytes per second.

(Refer Slide Time: 21:05)

The slide has a dark blue header and footer. The main content area is yellow. The title 'Synchronous versus Asynchronous Bus' is at the top. Below it is a bulleted list comparing the two types of buses.

- **Synchronous Bus:**
 - There is a common clock between the sender and the receiver that synchronizes bus operation.
- **Asynchronous Bus:**
 - There is no common clock.
 - Bus master and slave have to *handshake* during the process of communication.

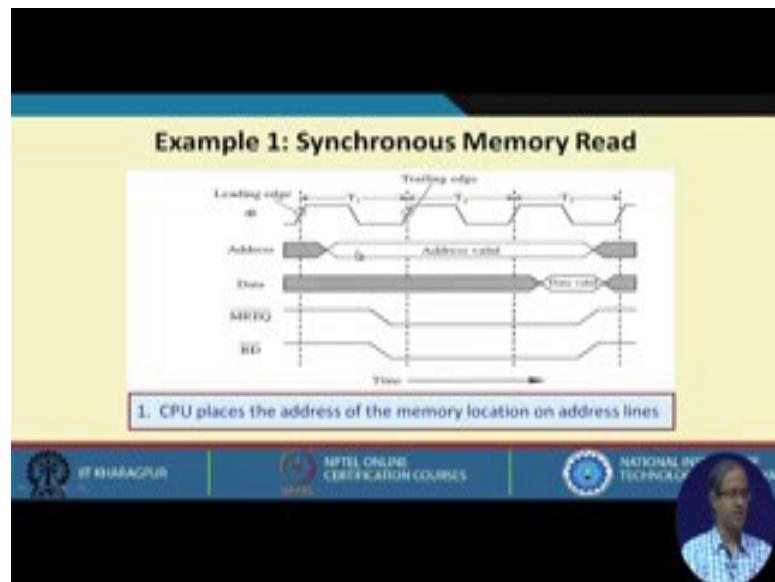
At the bottom, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

Just to have an idea, buses can also be synchronous and asynchronous. I will take an example to illustrate the difference between synchronous and asynchronous. With respect to IO transfer, you have already seen. The concepts are very similar for a synchronous bus. There will be some kind of a common clock between the sender and the receiver that will synchronize all data transfer operation over the bus.

In contrast an asynchronous bus does not have any common clock and just like in asynchronous I/O we had to use handshaking. Here also there will be a number of hand shaking signals which the master and slave will be sending each other, so that the data communication can be completed and both sides will be knowing about it.

Let us take an example.

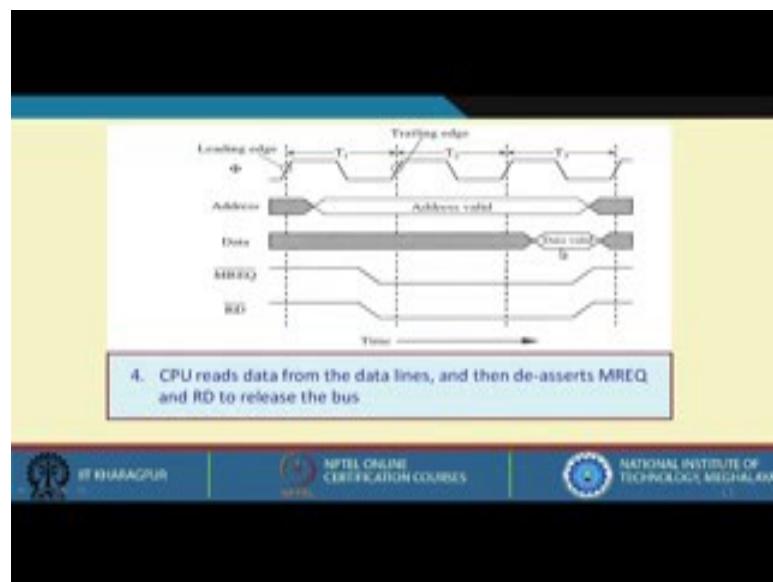
(Refer Slide Time: 21:57)



This is an example of a bus that connects CPU and memory. The first example is that of a synchronous memory read. Here we are showing some signal timing diagram. First one shows the clock and these are the clock states T1, T2, T3. This is the address bus, data bus. These are two control signals, memory request and read.

The first step is CPU places the address of the memory location in the address lines. It is here in the first clock. After the clock rises high, there will be some delay. After that delay, the CPU puts the address of the memory location in the address line. If the first step, none of the memory request and read lengths are active. These are bar means they are active low. They will be active when they are put to low or 0. So, first step is to put the address.

(Refer Slide Time: 23:06)



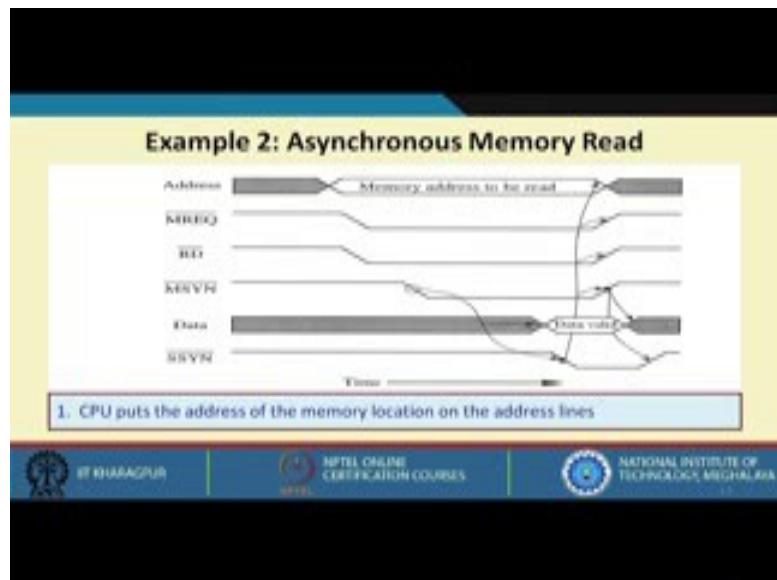
After the address is put, second step is CPU will assert the memory request and read lines. Memory request line will be set to low and read line will also be set to low, so that the memory system will now know that it is a memory request and this CPU is requesting read. At the end of T1, these two signals have been asserted. This address is already valid.

Third step, memory controller will be accessing memory location and load the data on the data lines. Now, memory will be having some access time. Let us say the read signal has activated here and it will take so much time for the valid data to come on the data bus. So, we are giving sufficient time for the memory to access the contents.

The data will be loaded on the data line here somewhere in T3. When the data has already arrived, sorry the last step will be CPU will be reading the data because it is already on the data line and it will de assert memory request and read lines because it is already done. So, memory request will again be set to high, read will again be set to high.

This is synchronous because CPU knows exactly how much time memory will take to read the data. It is waiting for exactly that much time. It will expect the data is already there. It will read from the data bus.

(Refer Slide Time: 25:36)

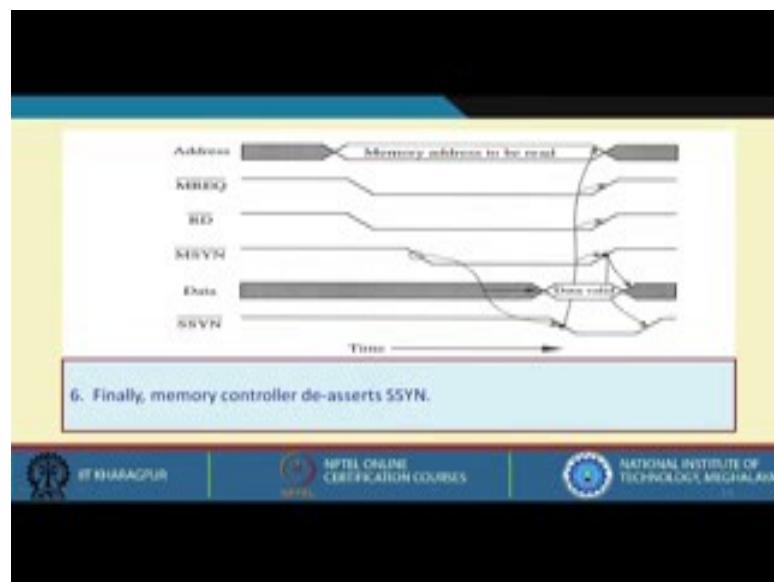


The example that we take here is that of asynchronous memory read. I am not showing the clock signal because earlier everything was happening in synchronism with the clock. We are assuming both CPU and the memory system were having access to the same clock, but for asynchronous system here, we are assuming that there is no clock. There are other handshaking signals using which CPU and the memory, both will be knowing exactly what is going on.

There are address lines, data lines and there are some other signals. You see memory synchronization signals.

The first step as usual will be for CPU to put the addressable memory location on the address lines. This CPU is putting the address on the address bus. So, a valid address is available on the address lines.

(Refer Slide Time: 27:01)



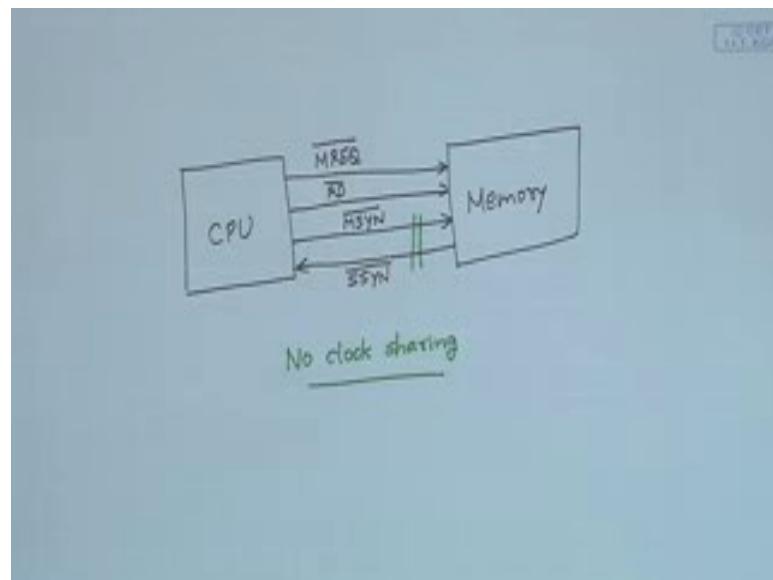
After the address lines have become stable, this symbol means it has become stable. CPU will assert memory request and read lines. These are two signals that CPU is sending to the memory system. The memory request line is activated and read line is activated. This will be after a little delay, after the address lines have been stabilized. Once these two have been activated, the memory read process starts.

Third step, what will happen is CPU will assert MSYN line. After memory request and read, now MSYN this is the handshaking signal. This MSYN will tell the memory system that CPU is now expecting some data from the memory.

Then, memory controller will take some delay depending on the access time. After some delay, the data will come on the data line and once it is valid, the memory controller will activate the SSYN. The signal that comes from the memory controller to the CPU and when it sees that SSYN is activated, it will know that data is available on the data bus.

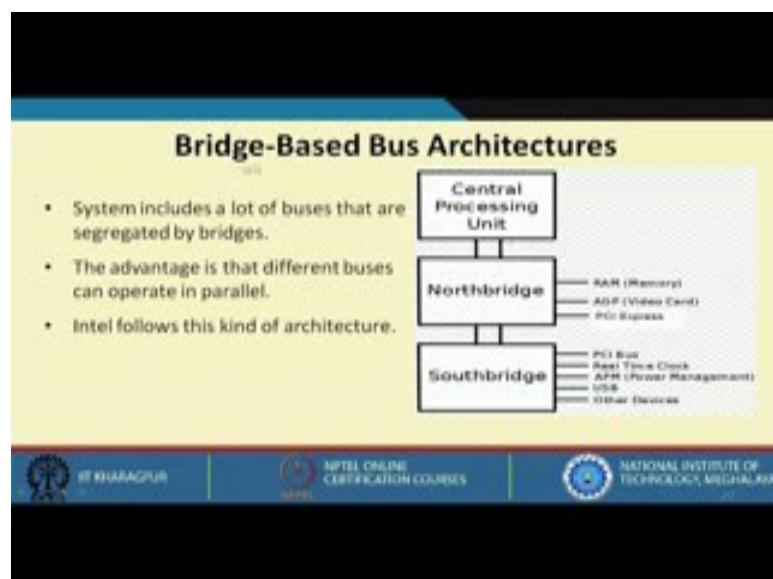
So, now what CPU will read the data from the data bus and it will de-assert memory request read and MSYN and lastly, memory controller will de-assert SSYN indicating that the operation is over.

(Refer Slide Time: 29:08)



Essentially in this method, you have CPU and you have the memory system. So, actually the memory controller is interacting. For memory read the signals that are used are memory request read. Then, there are two handshaking signals, MSYN and SSYN. Using these handshaking signals, data transfer can take place even without CPU and memory sharing a clock. This is the point to be noted.

(Refer Slide Time: 30:13)



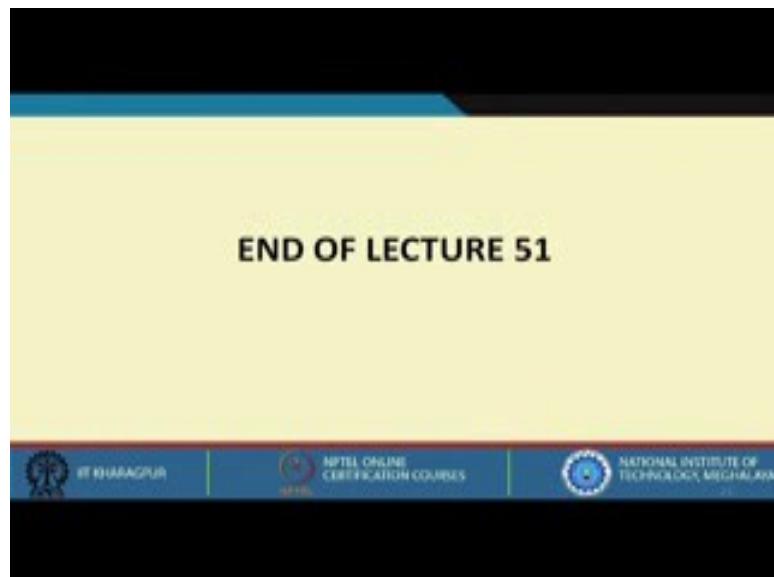
I have given the example of Intel mother board bus architectures that are typically based on the bridges. Many of the other manufacturers are also going by that. The system

includes a lot of buses that are not all connected together. They are segregated by bridges like you recall I mentioned the north bridge will be connecting RAM, memory, video card, AGP, PCI express, the highest speed buses.

The south bridge will typically connect the PCU that is the lower speed, PCI clock, USB and other devices like keyboard disk etc. So, Intel and also other companies have started following this kind of bridge based bus architecture.

So, this is the diagram that we showed earlier. This is an example of a bridge based bus architecture, north bridge south bridge. This is a very commonly used architecture that is available in the desktops and laptops that we mostly see today around us.

(Refer Slide Time: 31:22)



With this we come to the end of this lecture. In this lecture, we have talked about the need of buses inside a computer system, why we have so many different buses because each bus has a specific requirement, their speeds can be varying widely. So, instead of all devices connected to the same bus, it is always better to segregate the buses and by isolating them using bridges, you can have a better management.

In the next lecture, we shall be looking at one specific bus standard in some detail which is really becoming universal in today's context that is Universal Serial Bus or USB.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 52
Universal Serial Bus (USB)

In this lecture, we shall be talking about the Universal Serial Bus or USB that has become a real kind of an universal bus standard today. Almost all the devices that we want to connect to the computer system, we will see that there is USB interface available. You compare this scenario with the computer systems that were available about let say 15-20 years back. There you would see there will be so many different kinds of connectors at the back plane of your PC, the keyboard will be connected using a special kind of a connector, mouse will be connected using another kind of a connector, then the serial port will be RS233 serial port connector, the printer LPT will be connected to your printer, etc.

So, there will be a whole lot of connectors, but this USB tries to integrate or consolidate all these bus standards into a single bus standard, which is acceptable to almost all different kinds of application, so that you can have a computer system with a single kind of a port. If you look at a modern laptop, you will see that it has only USB port and nothing else. So, everything else you want to connect, you have to connect through USB and only that is what is happening today.

(Refer Slide Time: 01:55)

The slide has a yellow header bar with the title 'Universal Serial Bus (USB)'. Below it is a white content area containing a bulleted list. At the bottom is a dark footer bar with three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Universal Serial Bus (USB)

- USB is the most popular external bus standard in use today.
 - = Allows connection of almost all types of peripheral devices.
 - = USB interfaces exist today in keyboard, mouse, printer, scanner, mobile phones, disks, pen drives, camera, etc.
- Facilitates high-speed transfer of data.
 - = USB 1.1 (1998): up to 12 Mbps
 - = USB 2.0 (2000): up to 480 Mbps
 - = USB 3.0 (2008): up to 5 Gbps
 - = USB 3.1 (2013): up to 10 Gbps

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

The topic of today's lecture is Universal Serial Bus or USB. So, as it said USB is the most popular external bus standard. It means, it is not used inside a computer system, but rather this is used to connect some device to the peripheral system, external to the system outside the computer.

USB standard allows connection of almost all types of devices. This can include keyboard, mouse, printer, scanner, mobile phones, disk, pen drive, camera. Anything you name, you will find that there is an USB interface available.

The advantage of USB is that it facilitates high speed transfer of data. Now, the standard has evolved over the years. Some of the standards I am showing: version 1.1, 2.0, 3.0 and 3.1 and these are the years where they were proposed.

The first USB 1.1 version, it worked up to a speed of 12 megabits per second. At that time it was considered to be good enough, but USB 2.0 which came in 2000, the speed was jacked up to 480 megabits per second, but this 3.0 and 3.1, they have brought it an altogether different level, 5 and 10 gigabits per second.

With the modern day USB interface, you can interface virtually any kind of device because 10 gigabits per second is really fast. There are newer standards available that can go even higher 40 Gbps also.

(Refer Slide Time: 04:09)

History of USB

- A group of 7 companies (Compaq, DEC, IBM, Intel, Microsoft, NEC, Nortel) initiated the development of the USB standard in 1994.
- Main Goal:
 - Simplify the problem of connecting external devices by replacing the variety of connectors that were available earlier.
 - Simplify software configuration of the connected devices.
- The first USB version 1.0 appeared in 1996, which was followed by many other generations, with USB 3.1 being the latest.

 IIT KHARAGPUR  NPTEL ONLINE CERTIFICATION COURSES  NATIONAL INSTITUTE OF TECHNOLOGY AND INDUSTRIAL INNOVATION



USB was a standard that was jointly formulated by seven companies that included Compaq, Digital Equipment Cooperation which of course today is no longer there. Compaq is also no longer there. Then IBM, Intel, Microsoft, NEC and Nortel. They sat down together and felt that there is a need and necessity to come up with a new standard for connecting peripheral devices.

The main goals were two fold. First was there was a big problem of connecting external devices to the computer. Earlier there were so many different standards and that is why the computer systems came up with so many different sets of connectors. Depending on the device, you will be connecting them to one of the available connectors.

Instead of having so many different kinds of connectors, why not have a single kind of a connector that can connect almost all devices, and the second problem is also important to simplify software configuration of the devices you are connecting.

Now, with USB there is a mechanism. If you connect a device through the USB port, the system software will automatically try to find out what kind of device it is and it will automatically try to locate and install the device driver. You might have experienced when you connect various devices like pen drives and other devices to a computer system, but earlier it was not like that. So, when you connect a new device, you will have to put in CD or DVD containing the device driver and you will have to manually install the device driver. Then only the device will work. So, this USB made the task easier. As

it said the first USB version 1.0, this appeared as early as in 1996. There are many other versions in between. The latest version is 3.1.

Now, let us look a little bit inside.

(Refer Slide Time: 06:37)

- How are data transmitted?
 - Data are transmitted serially using differential NRZI encoding.

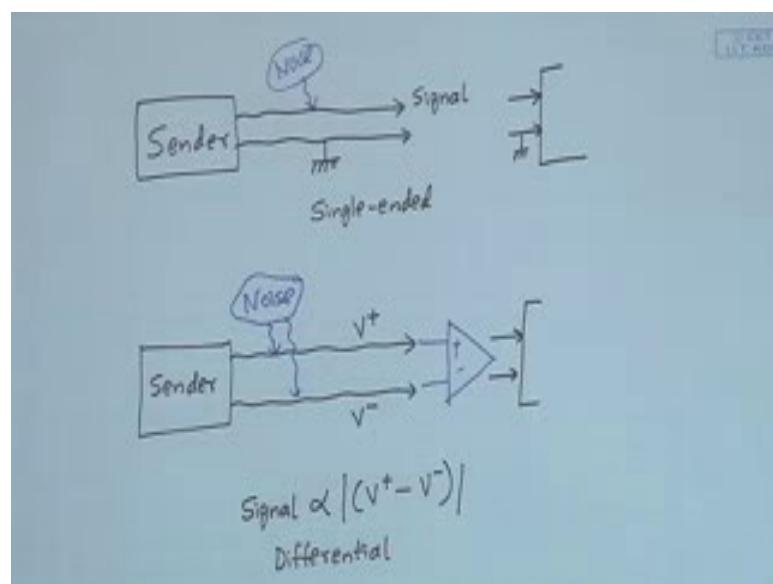
Bit to be sent	Previous line state	New line state
0	0	1
0	1	0
1	0	0
1	1	1

– Bit stuffing is used to ensure minimum bit toggle frequency during communication.

- A 0 is inserted whenever a sequence of 6 1's is encountered.
- 1101 1111 1101 0100 → 1101 1111 10101 0100

USB how it works? First is that it is a serial bus. Data are transmitted serially and uses differential NRZI encoding. Now, let me first tell you what is meant by differential. When you carry out communication suppose I am the sender.

(Refer Slide Time: 07:02)



There are two ways in which you can send data. You can have two wires, one wire can carry your signal and the other wire can be ground. When you collect the signal at the receiver, receiver should also know that this is your signal and this is your ground. So, you have to connect the signal in the proper order. This is sometimes called single ended connection.

There is an alternate way of connecting where we do not have a separate ground connection, rather signal is sent over two lines. One we call it as V+, we call it as V- and the receiver receives these two signals. The idea is that the signal that is transmitted is actually represented by the difference of V+ and V- and modulus of that and this mode of transmission is called differential mode of transmission.

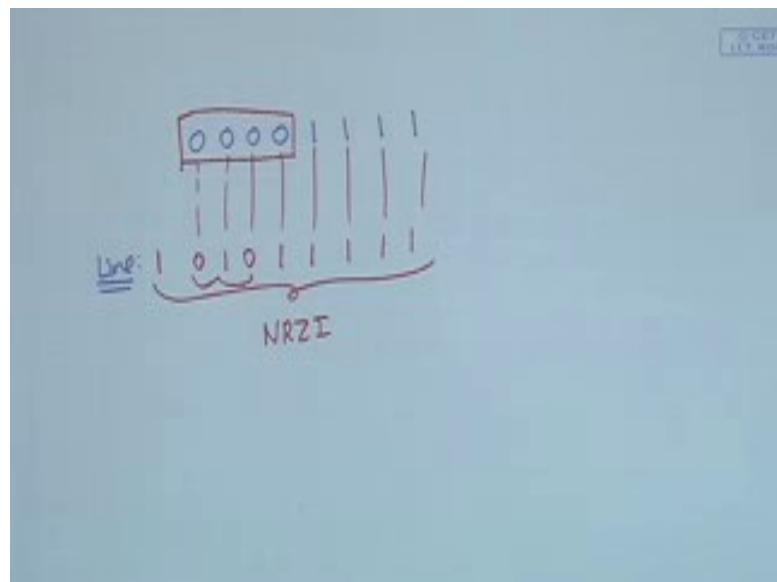
The advantage of differential mode of transmission is that suppose when there is the communication link like this, there can be some nearby sources of noise like power supply lines and other things which will be injecting noise on the communication lines. Now, if it is the pair of wires, the same noise will be injected on both the lines and if you subtract them, the noise will be cancelled out.

So, at the receiving end, you will be having some kind of a differential amplifier. It will be finding the difference of the signals amplified and feed it to the receiver, but on the single ended case since one line was grounded, grounded line is never infected by noise. Noise will only affect the line carrying the signal. So, the receiving signal will be degrading in quality.

So, differential signal has this advantage. The way the data bits are encoded, there is a standard way NRZ immediate (NRZI). Depending on the bit you are sending and the previous state of the line you do not send, the bit 0 0 or 1 1 directly depending on what was in the previous state.

Suppose if the bit is 0, the earlier the line was 0, then you send 1. Well, if the bit is 0 and earlier the line was 1, you send 0. If 1 0 0 1 1 1 let us take an example and see what happens.

(Refer Slide Time: 10:28)



Suppose my data that I am transmitting is 0 0 0 0 1 1 1 1 and let say my line was initially the line was at 1. Let us see what will happen here. The previous state was 1 and now this line is 0. You look at this table. Previous line state is 1 bit to be sent is 0 and the new line state will be 0.

So, new line state 0 bit to be sent is 0. You make it 1 1 new line state. Previous state is 0. Make it 0. So, in this way new 1 and 1 and 1, it will remain 1 1 and 1 and 1 again. Previous lines it is 1, this one like this. So, you do a kind of encoding before you are actually sending the data. This is called NRZI encoding.

NRZI encoding has some good properties that you always inject some transitions 0 1 0 1. Well, even if your line is continuously at 0, the input is 0, a long stream of zero, but still in the line there will be some transitions.

Now, transitions on the line are always good because you can synchronize the receiver. If the signal is changing, receiver can synchronize itself with the transitions. It is always desirable in a serial communication link, there should be a sufficient number of transitions on the line.

There is another way to ensure that long streams of zeros and ones will never be there. There is a method called bit stuffing that is also used in USB. Bit stuffing says that whenever there is a sequence of six consecutive ones, then the transmitter will forcibly

insert a 0. So, let say this is bit stream to be communicated here, you see 7 1's are there consecutive.

So, after 6 ones, a zero is getting inserted and this stream is transmitted. The receiver will receive the same thing. Whenever it will receive six consecutive 1's, it will drop the next bit. The advantage of bit stuffing is that you will never have a scenario where the line will be having continuous stream of long stream of 1s. This NRZI ensures that zeros will not be there. Long stream of zeros and bit stuffing will ensure that long stream 1's will not be there. So, there will be guaranteed transitions in the lines. This is ensured by these two methods.

(Refer Slide Time: 13:38)

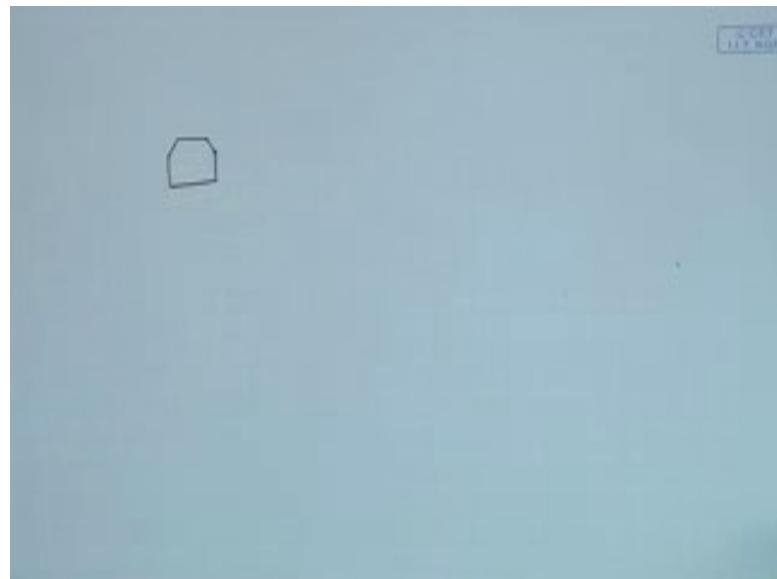
USB Connectors

- There are two pre-defined connectors in any USB system.
 - The *Type-A plug* has an elongated cross-section, inserts into a Type-A receptacle on a downstream port on a USB host or hub, and carries both power and data.
 - The *Type-B plug* has a near square cross-section with the top exterior corners beveled. As part of a removable cable, it inserts into an upstream port on a device (e.g. printer).
- For connecting smaller devices like mobile phones and digital cameras, mini and micro USB connectors have also been developed.

Another interesting thing is about USB connectors. You have seen USB connectors yourself because you must have handled all these devices. Well, traditionally the USB connectors came in two kinds, Type A plug and Type B plug. Type A plug has an elongated cross section that inserts into A type receptacle. Receptacle means a Type A plug can go inside A Type A receptacle only. So, there will be like a male female connector. It is called a connector will be going inside another connector. A type can go inside A type only, but the two connectors are different. The receptacle normally when you say a computer system or a laptop, the connectors which are available those are the receptacle connectors and a device when you connect that is your normal Type A connector which goes inside that receptacle, right.

The Type B plug looks different. So, it is not elongated and flat. It looks like a square with a top external corners beveled.

(Refer Slide Time: 15:10)



Beveled means it has a shape like this, right square, but the top two corners have a little curved. Now, this Type B plug traditionally was used to insert into an upstream port or a device let say printer.

Earlier when you used to connect a printer or a plotter to a computer system, you have a USB cable wherein one side you had a Type A plug and on the other side, you had Type B plug. Though Type B plug will go inside the printer and Type A plug will go inside the computer system. This means with the developments in the standards, this difference is no longer required. You can have a cable with the same kind of connector both ends. You can connect the cable in any order you want. It is not that Type B will go into the printer and Type A will go into the computer and not the reverse.

I am showing you some pictures here.

(Refer Slide Time: 16:50)



This picture shows you the Type A connector. You see the Type A connector is the one that you see mostly. It is a little elongated and flat. This is your normal connector and this is your receptacle. The normal connector will go inside the receptacle and Type B looks like this. This will be your Type B connector and it will go inside a receptacle that looks like this.

Say for example, if you have a printer with a Type B connector, so on the printer you will be having a port like this and on the cable, you will be having a port like this, and this standard USB connector looks like this and the corresponding mini connector looks like this. So, you see it is less than half the size.

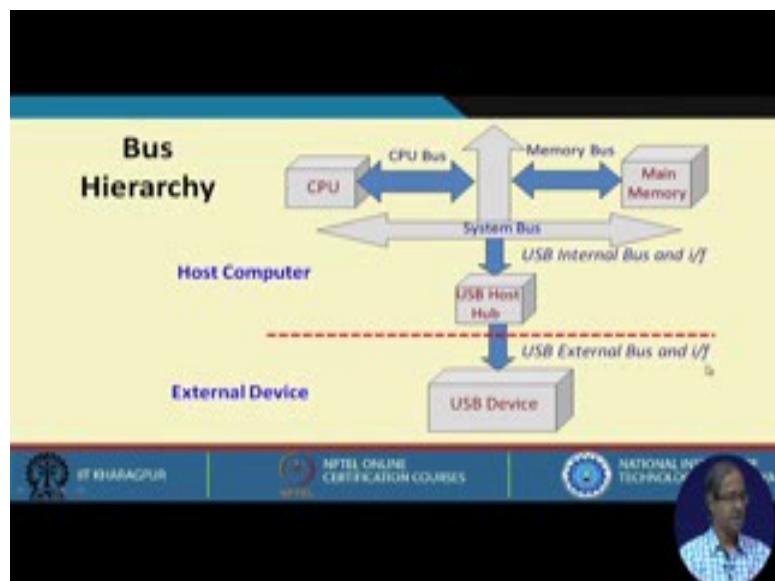
Micro USB connector width wise it is almost same, but thickness wise it is smaller and you see the connectors that we have on our mobile phones today.

(Refer Slide Time: 17:54)



Let us see that the connector which we have in the back, this one, this is an example of a micro USB connector. The cable that you connect to it that is a micro USB cable.

(Refer Slide Time: 18:11)



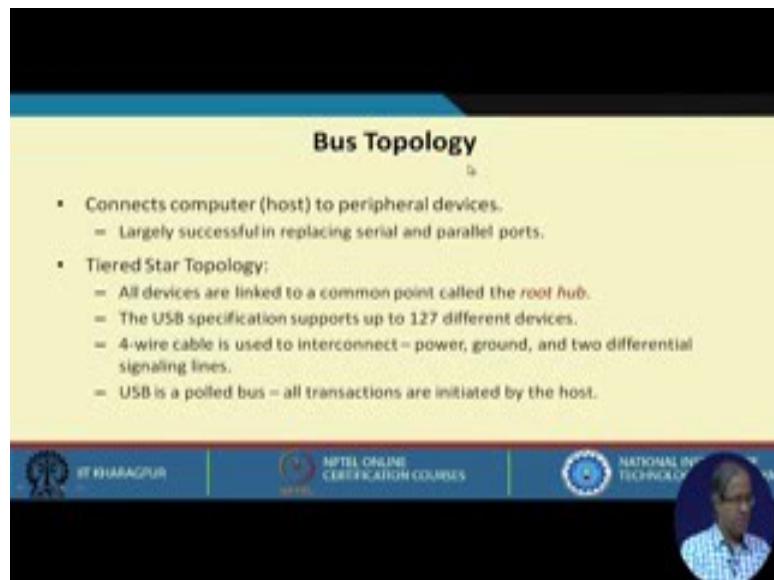
In USB there is a concept of a bus hierarchy, the way the USB devices are connected. Let us look at this picture. On the top part we have a host computer. This can be PC, desktop or laptop. I am not showing the north bridge and south bridge separately. I am just showing it in abstract sense. There is CPU, there is a memory, you have CPU bus, you have a memory bus. May be your north bridge is setting here, then you have a

system bus may be this is your south bridge. This is connected to a chipset that is your USB host hub. USB host hub -- this chipset is available that also is inside your computer mother board.

From the USB host hub, you will have all the USB connectors on your computer on external device. You may be having a pen drive or any other device you want to connect it to the USB host hub, so you have the connector here that will be connecting to one of the host computer connector. The two buses that you have here, system bus and USB host hub, they are connected through the so-called USB internal bus and interface and when you connect an external device through an external cable, that is referred to as USB external bus and interface.

Now, bus topology.

(Refer Slide Time: 19:52)



By bus topology what I mean is that how these external devices are connected. See inside the computer system, it is fixed. Now, how you can expand the capability after that you can connect several devices outside. You must have seen there are some products called USB extenders. You plug a device to the USB port and on the other side, you will be getting four USB ports.

So, you have a USB connection topology that is sometimes called a bus topology. I will be illustrating with the example. This bus topology is used to connect the computer

system that is sometimes called the host to the peripheral devices. Now, again you think of the master slave relationship I talked about earlier, when say you connect a computer with a peripheral device like a pen drive. A computer works as the master and your pen drive works as a slave.

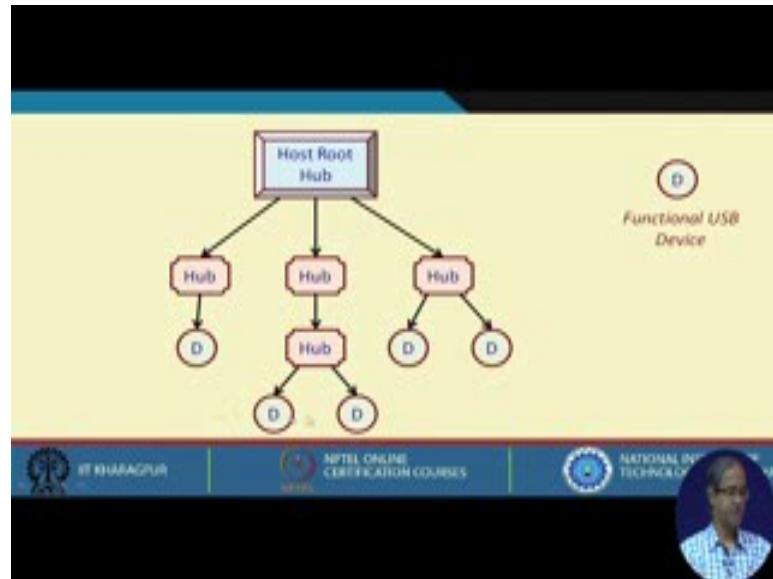
Well, a few years back when you connect a mobile phone to your laptop, your mobile phone also can be used only in the slave mode, but nowadays with the advancements in the operating system and the hardware of the mobile phone, for example this is a modern mobile phone here. You can use this mobile phone as the master as well like here you have a micro USB connector. You can connect an adapter through that adapter, you can connect a pen drive to this mobile phone also. There this mobile phone will be acting as the master and the pen drive will be acting as the slave.

This bus topology is typically a star topology. It is a multi-tiered architecture and not a single level. There can be multi-level of hierarchy and it looks like a star. There is no loop or feedback. We will take an example. The devices are all connected to a common point which actually is called a root hub that refers to the master and in the star topology, the USB specification is such that you can connect up to 127 devices and each USB connection is a 4-wire cable; just a power supply, a ground and two differential signaling line V+ and V- as I said.

With the development in the USB technology, the power and the ground lines are also such that sufficient power can be delivered over the lines. Nowadays, the external hard disk drives that are available, you can straightway connect them to the USB port without any need to separately connect any power supply.

Now, you recall in an external hard disk, there has to be some motors inside. So, there will be significant power requirement there, so that entire power can be drawn from the USB power lines and lastly, as I had said USB is called a polled bus like all transactions are initiated by the host that is the root.

(Refer Slide Time: 23:33)



The bus topology this tiered architecture looks something like this. You have a host root hub, this is your host connection and you can connect this D circular things are the devices, these are your USB devices. If you want you can connect a device directly through a hub. If you want to extend the capability, you can connect several levels of hubs and there are hubs that can be having several different connections.

Suppose this has four connections, you can connect four devices. Like this you can develop a tree or a star kind of a connection with which you can connect up to 127 USB devices.

(Refer Slide Time: 24:27)

- **USB Host:**
 - It is a device that controls the entire system (usually a computer).
 - It processes data arriving to and from the USB port.
 - It contains a sophisticated set of software drivers.
 - Drivers schedule and compose USB transactions.
 - Access individual devices to obtain configuration information.
 - Software dependence of USB systems make it difficult to use on stand-alone systems (with no OS support).
 - The physical interface to *USB Root Hub* is called the *USB Host Controller*.

Here you have host, you have hub, you have the devices. Talking about the host it is a device that controls the entire system and is usually a computer or a laptop. Some mobile phones can also act as the host. It will be processing data arriving from the USB port and it contains a sophisticated set of software drivers. Depending on the kind of USB devices that are connected, the appropriate drivers are selected and they are executed, so that data transfer can take place.

Because of this kind of an operation, USB is not so easy to use in small standalone systems, where there is no operating system support. You need to install the driver software and then only you can use your USB in a proper way.

So, the interface to the USB root hub is sometimes called the USB host controller.

(Refer Slide Time: 25:43)

- USB Hub:
 - It checks for new devices and maintains status information of child devices.
 - It serves as *repeater*, boosting strength of upstream and downstream signals.
 - It *electrically isolates* devices from one another, thus allowing an expanded number of devices.
 - Allows malfunctioning devices to be removed.
 - Allows slower devices to be placed on a faster branch.
 - Can be purchased as stand-alone devices.

Talking about the USB hub, it checks the new devices that you are connecting to it and maintains status information like when you connect a new device to an USB port, that port will check what kind of device it is and depending on that it will try to locate the driver that needs to be activated. If the driver is not available, the host will try to install that new driver.

The USB hub also acts as a repeater like it can boost signal strength for both way connection, upstream and downstream. Not only that, it serves the purpose of electrically isolating devices from one another and this allows you to expand to a larger number of devices. You can remove a device if you want to if not working properly. You can have any kind of connection because of those hubs. You can have several hubs and a hub will be isolating the devices that are under it from the rest of the system. Just like a network switch or a network hub we use for connecting computer system, the same concept is used here.

This USB hubs are available as separate systems that you can purchase and connect your system to expand the number of ports and the USB devices.

(Refer Slide Time: 27:07)

• **USB Devices:**

- All functional USB devices are *slaves*; only responding to data reads or writes, never initiating any.
- May indicate a need to transmit or receive data through polling.
- Contain registers that identify relevant configuration information.
- Exist in conjunction with corresponding set of software drivers inside the host system.

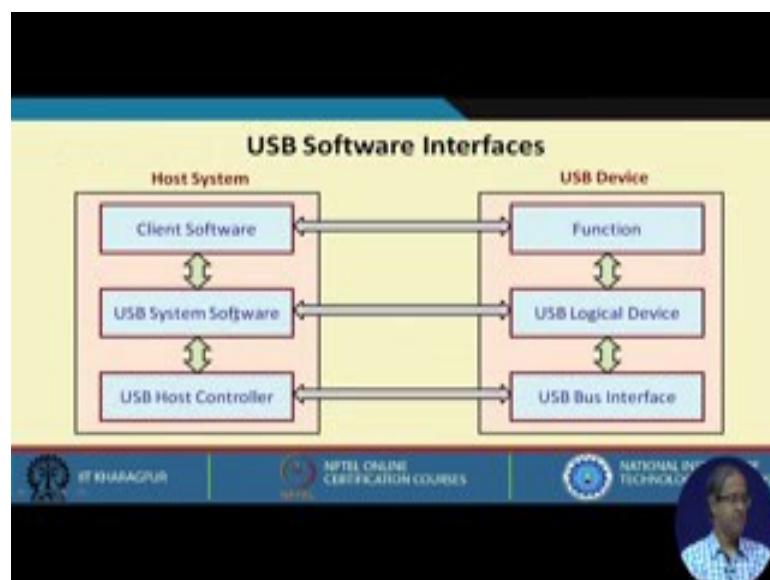
NPTEL ONLINE CERTIFICATION COURSES

NATIONAL INSTITUTE OF TECHNOLOGY KARUR

Conventionally all of them work as slaves. They cannot initiate any operations, they can only respond to operations that are initiated by the host. There are some registers inside the USB device that can identify what kind of device it is and what kind of drivers it required and so on.

This device can work only when the corresponding set of software drivers are there and are installed in the host system. This is a requirement.

(Refer Slide Time: 27:46)



Pictorially for the host system and USB device, the connection is like this. You see in the host system, you have the USB host controller that is the connector and in a software part, you have the USB system software and in a highest level, you have the client software and in other side, they interact with the USB bus interface because host controller directly connects to the USB bus interface by a cable and these are virtual connections. USB system software interrupts with USB logical device and client software interacts with the USB device function whatever it is.

(Refer Slide Time: 28:25)

The slide has a yellow header bar with the title 'USB Host Controller' in white. Below the header is a large blue rectangular area containing a list of bullet points. At the bottom of the slide is a dark footer bar with three logos and text: IIT Kharagpur, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY ROURKELA. On the right side of the footer bar is a circular video player showing a man speaking.

- The Client Software determines what transactions are required with a given device.
 - What data is to be transferred?
- Scheduling and configuration of data transfers is completed in *USB System Software* level.
 - When and how often data is to be transferred?
- Data transfers are composed and regulated at the *USB Host Controller* level.
 - How are data to appear to the functional device?
 - How does system keep track of data sent and received?

The client software determines what transactions are required with the device, what data is to be transferred, scheduling and configuration data transfer in completed USB system software level. All data transfer gets completed here logically. When and how often data is to be transferred that is decided there and either lowest level USB host controller data transfers are composed and regulated.

(Refer Slide Time: 29:18)

Future of USB

- USB Type-C Plug:
 - They are about the same size as micro USB connectors.
 - Can deliver power output of 20 volts and 5 amps (100 watts).
 - Can be used for charging laptops and phones.
- Thunderbolt 3 port uses the same port type as USB-C.
 - Peak speed up to 40 Gbps.
 - Available on Apple machines.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY

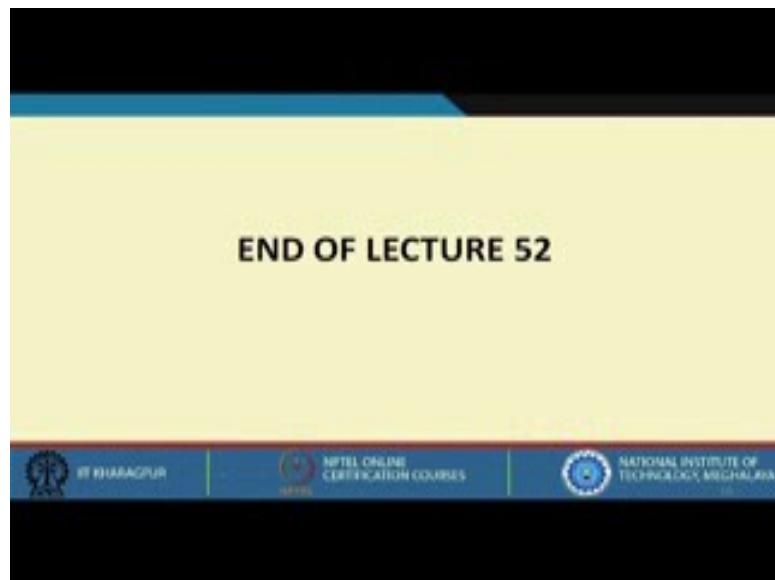


Talking about the future of USB, you might have seen a third kind of USB plug that is available on the Macbook Air. The recent versions that are there, they have come up with a new USB port called Type C port. Type C port looks a little circular. They are symmetrical. The plug that you are putting in, you can reverse it also in putting without any problem. So, the direction is not important. You can put the plug in any way you want.

This is the type C plug and you can see a small version of that same size is micro connector. You can put it in any direction you want and they can deliver pretty high powers. You can see 20 volts and 5 amperes -- it comes to 100 watts. You can use these ports as your charging points also. You can connect these ports to a charger for charging laptops and phones as well.

Now, Thunderbolt is a standard that is also very specific to the apple systems. Thunderbolt 3 port also uses the same kind of USB C connectors, where the speed can go up to 40 gigabits per second. You see in the future you have the same kind of USB ports with smaller feature factors coming up with higher and higher speeds. Almost any kind of applications that demands high speed communication, they can be handled by this kind of interfaces.

(Refer Slide Time: 31:09)



With this we come to the end of this lecture, where we talked about various bus standards and one of the very important bus standard which is there with us today, the USB.

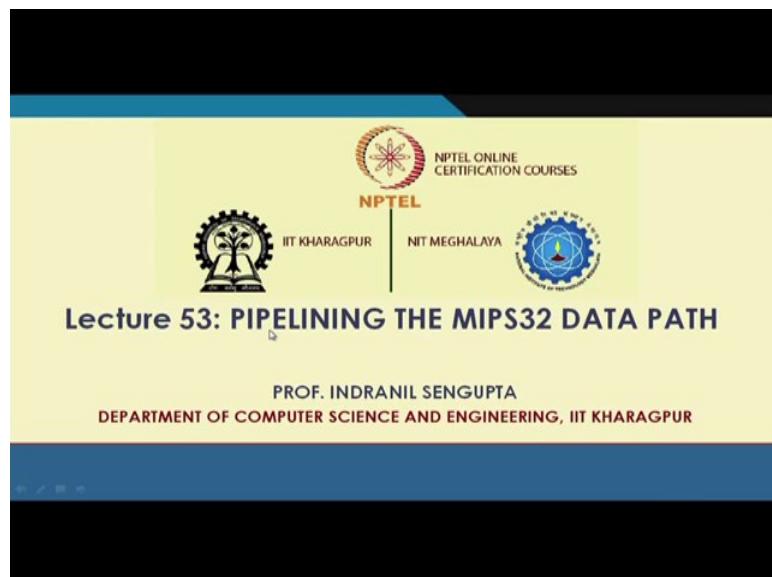
Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 53
Pipelining the MIPS32 Data Path

In this week we shall be starting our discussion by looking at how we can create a pipelined version of the MIPS32 data path. Earlier we have seen how we can have a non-pipelined data path for the MIPS32 instruction set architecture. You have seen because of the simplicity of the instruction set, because of the regularity, because of the simple instruction encoding, the required hardware was very simple. If you recall, there are 5 steps overall in the MIPS32, instruction fetch, instruction decode, execute, memory operation and write back. We start from there in this lecture.

(Refer Slide Time: 01:32)



We shall be exploring the pipelining of the MIPS32 data path.

Let us look at the basic requirements first.

(Refer Slide Time: 01:42)

Introduction

- Basic requirements for pipelining the MIPS32 data path:
 - We should be able to start a new instruction every clock cycle.
 - Each of the five steps mentioned before (IF, ID, EX, MEM and WB) becomes a pipeline stage.
 - Each stage must finish its execution within one clock cycle.
- Since execution of several instructions are overlapped, we must ensure that there is no conflict during the execution.
 - Simplicity of the MIPS32 instruction set makes this evaluation quite easy.
 - We shall discuss these issues in detail.

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL TECHNOLOGY

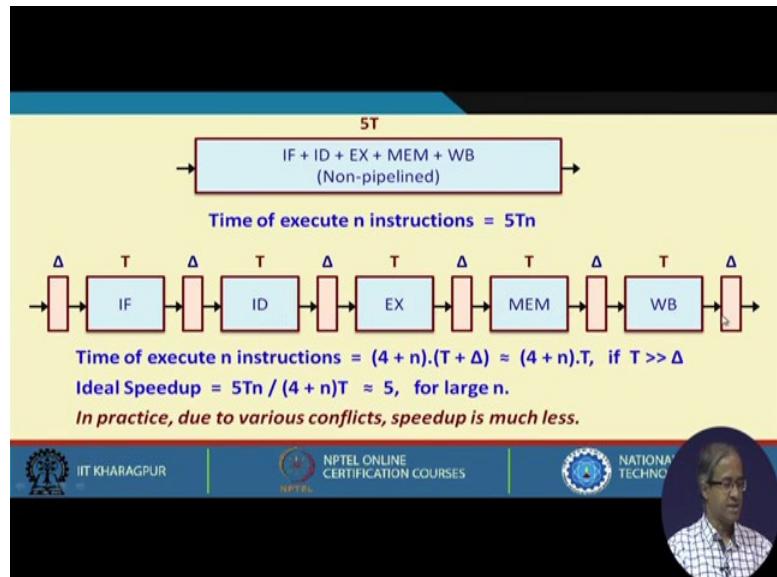
If we want to do pipelining, what are the basic requirements? The first requirement of course, is we should be able to start a new instruction every clock cycle. See if we talk about instruction pipelining we are saying that instructions are being pipelined, we are feeding the instructions to a pipeline one by one. So, instructions are flowing through the pipeline stages one by one, and they are getting executed. In the ideal case we would expect that one instruction would be completing every cycle, which is the beauty of pipeline. Because we are getting one instruction completed every cycle we should also be able to start a new instruction every cycle.

This is one requirement, and earlier the 5 steps that were mentioned, these 5 steps will become pipeline stages. This is another modification we would be doing. And the stages will be such that, they must be finished within one clock cycle. The clock period must be chosen to be large enough such that every stage execution must be finished by that time. This is a mandatory requirement that the clock cycle time must be large enough such that every stage should finish their executions.

Now, there are complications that will come in we shall see, as executions of several instructions will overlap now. We have these stages: IF ID EX MEM WB. When one instruction is in ID, next instruction will try to come into IF, and so on. We must ensure that there is no conflict of any sort during the execution of these instructions. We shall also see that there will be conflicts, but the analysis and some solutions to avoid them

becomes quite easy because of the simplicity of the MIPS32 RISC instruction set. We shall be discussing these issues in some detail subsequently.

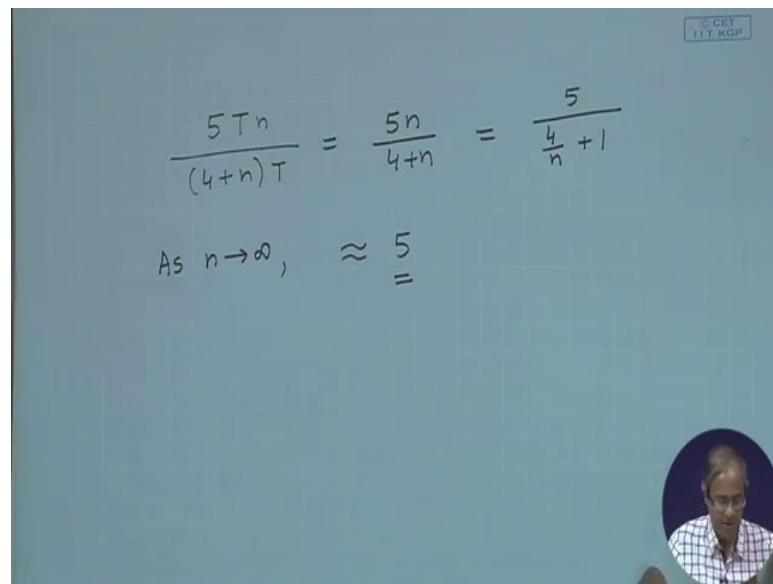
(Refer Slide Time: 05:03)



This is our non-pipelined MIPS32 data path, where the 5 steps are all together. Let us assume that the time taken by this entire thing is $5T$. Every instruction requires a time $5T$, it finishes, and then the next one comes. So, if there are n instructions the total time will be $5Tn$. Now for the pipelined version we do something like this. We break these 5 steps into 5 stages, and we insert latches between stages. Let us assume that each of the stages take time T and latches have a delay of Δ . We have already seen that how we can analyze a pipeline for its execution performance. You can similarly say that for n instructions what will be the total time, 4 clock cycles will be required to fill the pipe, and after that we will be getting one output every clock cycle.

The total number of clock cycles required will be $(4 + n)$. And what is the clock cycle time? It must be at least equal to $(T + \Delta)$. Now if T is large as compared to Δ , we can ignore this Δ , and this becomes approximately equal to $(4 + n)T$. So, ideal deal speedup will be $5Tn / (4 + n)T$.

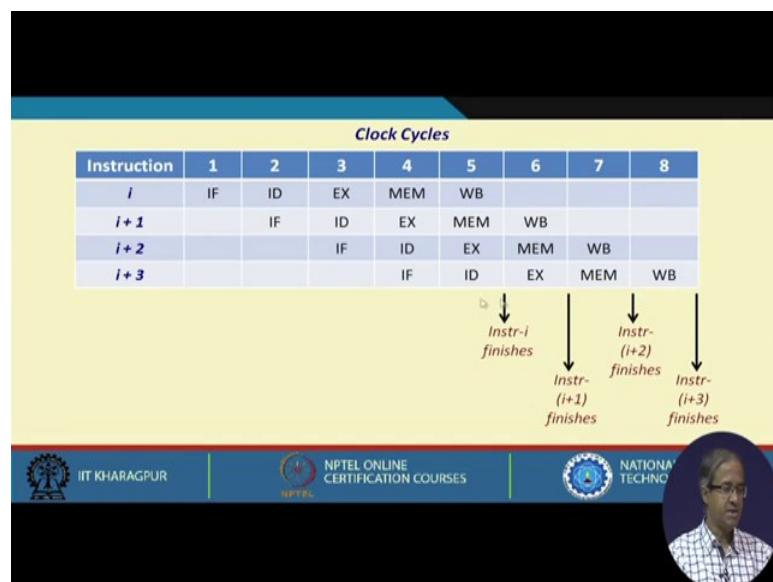
(Refer Slide Time: 07:20)



As n becomes large, this value will be approximately equal to 5. This will be the ideal speedup as n becomes large. So, ideal pipeline speedup will be equal to the number of stages.

We will see later that things are not that rosy in an instruction pipeline. There will be various kinds of conflicts that will appear, and speed up will be significantly less than 5.

(Refer Slide Time: 08:51)



Let us see how the pipeline works. I am showing the time steps or clock cycles and let us say instructions are coming one by one; it is instruction $i+1$, $i+2$, $i+3$ like this.

So, in clock cycle 1 the first instruction i enters the IF stage; that means, the instruction is fetched. After it is fetched it goes to the ID in step 2. While it is being decoded the next instruction can be fetched. Thus, there is some overlap. After this is done first instruction will go to the EX phase, second instruction to the ID phase and third instruction can go into the IF. In this way after fourth step the pipe is full. Now this way it will go on. The first instruction will finish here, second instruction will finish here, third instruction will finish here, like this.

It is after time step 5 instruction i will finish. Then time step 6 instruction $i+1$ will finish, time step 7 $i+2$ will finish, and time step 8 $i+3$ will finish. After the initial delay for the pipe to get filled up, in the ideal case you will be getting one instruction completion every clock cycle. That is a beauty of pipelining.

(Refer Slide Time: 10:46)

Clock Cycles								
Instruction	1	2	3	4	5	6	7	8
i	IF	ID	EX	MEM	WB			
$i+1$		IF	ID	EX	MEM	WB		
$i+2$			IF	ID	EX	MEM	WB	
$i+3$				IF	ID	EX	MEM	WB

Some examples of conflict:

- IF & MEM: In clock cycle 4, both instructions i and $i+3$ access memory.
 - Solution: use separate instructions and data cache.*
- ID & WB: In clock cycle 5, both instructions i and $i+3$ access register bank.
 - Solution: allow both read and write access to registers in the same clock cycle.*

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now you can see that it is not that simple as there can be various kinds of conflicts. In this diagram now I am showing two kinds of conflicts.

First is between IF and MEM, which is shown in red. Suppose the first instruction was a load or store kind of instructions, during the MEM stage it will be accessing memory. But at the same time instruction $i+3$ is also trying to read from memory in IF. So, there is a conflict for memory both instructions i and $i+3$ are trying to access memory at the same time. Now for this particular case, we can suggest a solution that will let us use separate instruction and data cache. See now you can find a logic why people actually use

separate caches for instruction and data. If we do this then IF will be using the instruction cache and MEM will be using the data cache. So, the conflict that was there is apparently removed or avoided.

Similarly, there is another conflict here, suppose first instruction is a register type instruction, say add. After everything is done it will be writing the result into the register in the WB phase. Now if we recall the micro operations that we have seen earlier for the different stages, you remember that in the ID phase the register operands are pre fetched. So, again here marked in blue there is a conflict in time cycle 5. The instruction $i+3$ is trying to read from the register bank, and instruction i is trying to write into the register. This is also some kind of a conflict. What is our solution here? Our solution is we shall be allowing both read and write to continue in the same clock cycle for the registers. We can read from the registers also you can write into the register in the same clock cycle.

(Refer Slide Time: 13:54)

Advantages of Pipelining

- In the non-pipelined version, the execution time of an instruction is equal to the combined delay of the five stages (say, $5T$).
- In the pipelined version, once the pipeline is full, one instruction gets executed after every T time:
 - Assuming all state delays are equal (equal to T), and neglecting latch delay.
- However, due to various conflicts between instructions (called *hazards*), we cannot achieve the ideal performance.
 - Several techniques have been proposed to improve the performance.
 - To be discussed.

So in the practical scenario because of various conflicts that can arise between instructions, which are commonly known as hazards, we cannot achieve the ideal pipeline performance. In the ideal case we have seen that our speedup can approach 5, but because of hazards it can be less than 5.

However, various techniques have been proposed by the designers to improve the performance to the extent possible. We shall be looking at several of the techniques.

(Refer Slide Time: 15:27)

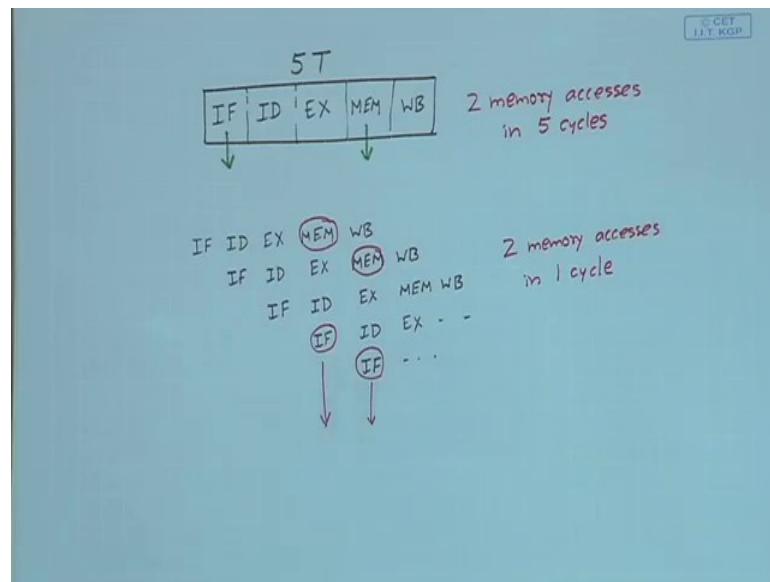
Some Observations

- a) To support overlapped execution, peak memory bandwidth must be increased 5 times over that required for the non-pipelined version.
 - An instruction fetch occurs every clock cycle.
 - Also there can be two memory accesses per clock cycle (one for instruction and one for data).
 - Separate instruction and data caches are typically used to support this.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL TECHNOLOGY

Let us look at some of the observations which we discussed a while back. Let us look at the memory constraint. Earlier, when you had a non-pipelined version of the MIPS processor, an instruction was getting executed in 5 steps, in say 5T time.

(Refer Slide Time: 16:01)



So the total time was 5T in non-pipelined version. Now if you consider memory access in this 5T time there was one memory access here and there can be another memory access here. Of course, all instructions are not load and stores, but in the worst case there

will be to memory accesses here. So, for the non-pipelined version there will be 2 memory access maximum in 5 cycles.

But for the pipelined case, the instructions will be overlapping. So what happens here? In the worst case in every cycle there can be potentially 2 memory accesses. Earlier there were 2 memory accesses in 5 cycles, but now for the pipeline version there can be maximum 2 memory accesses in every cycle. This simply means that the memory bandwidth requirement is increasing 5 times, and this is a big issue.

This has been summarized here. To support overlapped execution the peak memory bandwidth must be increased 5 times over that required for non-pipeline version. One solution we have already said that in the L1 cache we use separate instruction and data caches. Most of the time because of the locality of reference data reference by the CPU will be found in the caches.

So, there will be no conflict; one instruction can fetch from I-cache other instruction can read or write from the D-cache, but of course, when there is a cache miss there will be delay, but this is a simple solution and explains why we have separate instruction and data caches.

(Refer Slide Time: 20:28)

b) The register bank is accessed both in the stages ID and WB.

- ID requires 2 register reads, and WB requires 1 register write.
- We thus have the requirement of 2 reads and 1 write in every clock cycle.
- Two register reads can be supported by having two register read ports.
- Simultaneous reads and write may result in clashes (e.g., same register used).
 - Solution adopted in MIPS32 pipeline is to perform the write during the first half of the clock cycle, and the reads during the second half of the clock cycle.

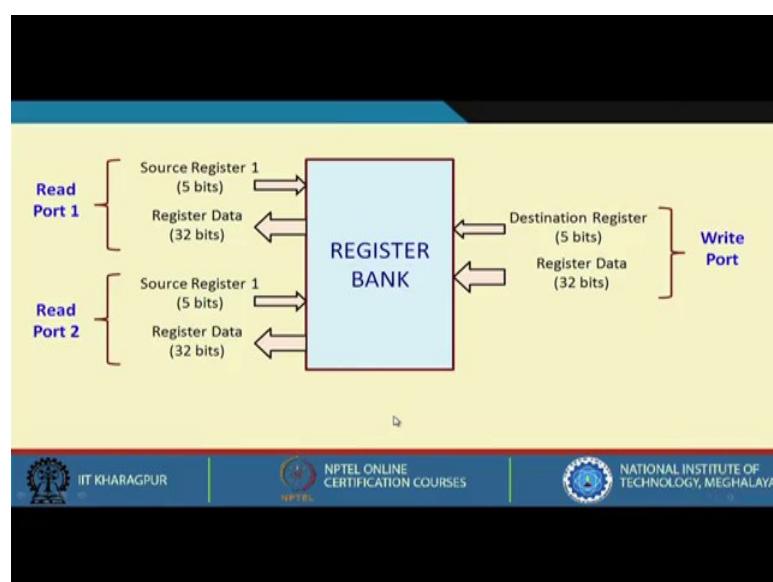
This second is that register bank issue. We mentioned earlier that some instruction can read a register bank during ID, some other instruction can write into a register during

WB. We just made a loose statement that we shall allow both read and write to happen in the same clock, but how is that really possible? Let us find a solution here. In the ID stage we are pre-fetching both the register operands source-1 and source-2. So, this ID requires 2 register reads.

And in WB we require at most one register write. So, in a pipeline our requirement is that in every clock cycle we should support 2 register reads and one register write. How we can have 2 register reads? Well if our register bank has two separate read ports then reading 2 registers parallelly or concurrently is not that much of problem, but the trouble will be when someone is trying to read register R1, but someone else is trying to write into register R1, then what will happen? Simultaneous reading and writing will result in clash if same registers are used. Here a very simple trick to solve this problem.

What is done in the MIPS pipeline is that we divide the clock cycle time into two halves. We are saying that in the leading edge of the clock we do all the register writes and the falling edge of the clock we do all the register reads, because you see register access is pretty fast. And clock cycle time is pretty long because all the stage executions must finish by that. In case that kind of a thing happens that someone is writing into R1 and some other instruction is reading from R1, then first the write will happen then the read of that new data will happen.

(Refer Slide Time: 23:52)



Our register bank picture will look like this. There will be 2 read ports. Source register numbers will be fed as register addresses 5 bits, 5 bits and the 2 register values will be read out just like reading memory. And on the other side there will be a write port that also will specify the register number where to write, and also the data that is to be written. As I had said write will happen in the first half of the clock and the 2 reads will happen during the second half of the clock. By using this simple convention and this modification to the register bank we can avoid this conflict also.

(Refer Slide Time: 24:42)

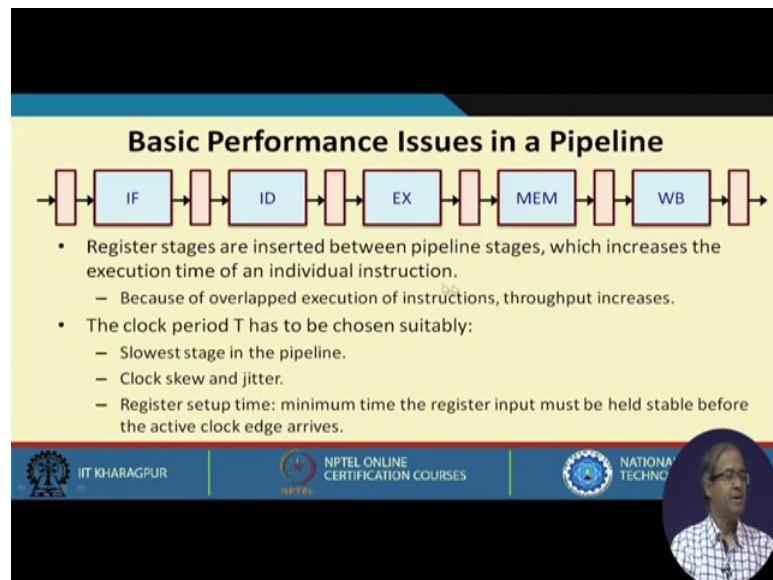
c) Since a new instruction is fetched every clock cycle, it is required to increment the PC on each clock.

- PC updating has to be done during IF stage itself, as otherwise the next instruction cannot be fetched.
- In the non-pipelined version discussed earlier, this was done during the MEM stage.

The third important issue is that you see earlier, the micro-operations for the non pipelined MIPS there in the IF stage we were not incrementing the PC what we are doing is that we are using a temporary register called NPC, we were storing $PC + 4$ into NPC. Later on during the MEM stage we were transferring that NPC into PC if it was not a branch.

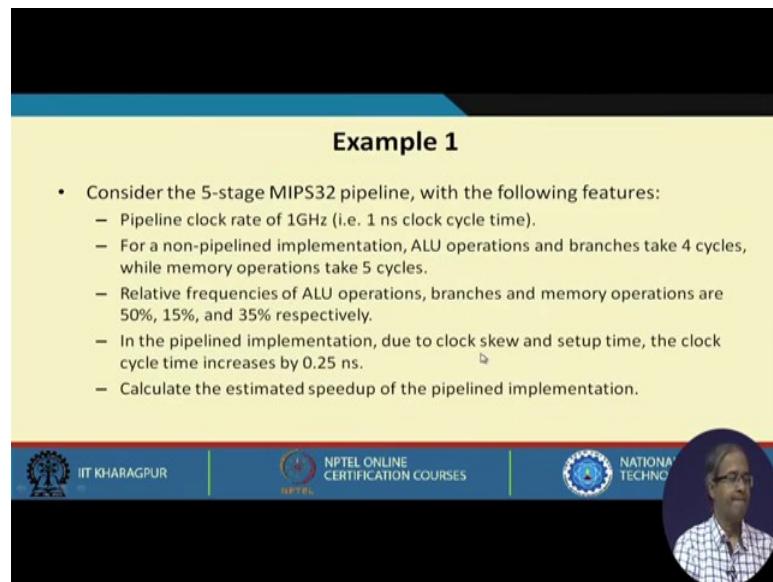
That was happening in a non-pipeline version, but now we are doing a pipeline can we afford to do that? In every clock we need to fetch a new instruction. So, the PC should be ready with the address of the next instruction at the end of every clock. At the end of every clock in IF itself we have to increment the PC, because if we do not do that we cannot fetch the next instruction in the next clock.

(Refer Slide Time: 26:17)



Some basic performance issues can be discussed here. I am showing the MIPS pipeline stages with the latches. The register stages or latch stages are inserted, as you know these are required for correct pipeline operation, because if you do not use these latch stages then the output of one stage may disturb the input of the next stage, so that the computation can become wrong. And again the clock period has to be chosen suitably, this was also discussed earlier. Because clock period T will depend on the slowest stage of the pipeline, plus also the clock skew and jitter and the setup time of the latches. All these times taken together will determine the minimum clock period that will ensure correct operation of the circuit.

(Refer Slide Time: 27:26).



The slide has a yellow header bar with the title "Example 1". Below it is a white content area containing a bulleted list of requirements for a MIPS32 pipeline. At the bottom is a dark blue footer bar with logos for IIT Kharagpur, NPTEL, and National Technology Engineering Institute, along with a circular portrait of a man.

Example 1

- Consider the 5-stage MIPS32 pipeline, with the following features:
 - Pipeline clock rate of 1GHz (i.e. 1 ns clock cycle time).
 - For a non-pipelined implementation, ALU operations and branches take 4 cycles, while memory operations take 5 cycles.
 - Relative frequencies of ALU operations, branches and memory operations are 50%, 15%, and 35% respectively.
 - In the pipelined implementation, due to clock skew and setup time, the clock cycle time increases by 0.25 ns.
 - Calculate the estimated speedup of the pipelined implementation.

Let us take an example. We considered a 5-stage MIPS pipeline with clock rate of 1 GHz, which means one nanosecond clock cycle time.

Now, in a non-pipelined implementation let us assume that ALU and branch operations take 4 cycles to finish, while memory operations because they also need memory access will take 5 cycles. Also assume that the frequencies of ALU, branch and memory operations are 50%, 15% and 35% respectively. We also have a equivalent pipeline implementation where because of the register stages, clock skew, etc the clock cycle time increases by 0.25 nanosecond; it becomes 1.25 nsec. The question is what will be the estimated speed up of the pipeline implementation?

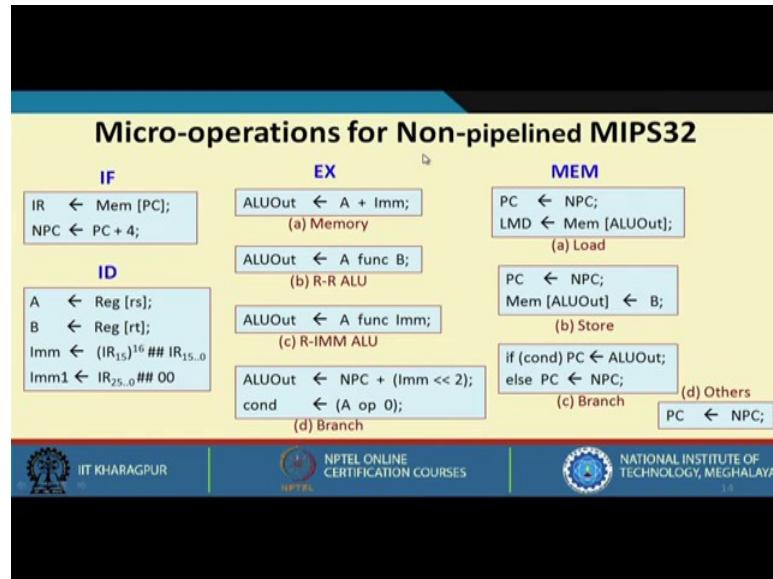
(Refer Slide Time: 28:48).

- **Solution:**
 - a) For non-pipelined processor:
 - Average instruction execution time = Clock cycle time x Average CPI
= $1 \text{ ns} \times (0.50 \times 4 + 0.15 \times 4 + 0.35 \times 5) = 4.35 \text{ ns}$
 - b) For pipelined processor:
 - Clock cycle time = $1 + 0.25 = 1.25 \text{ ns}$
 - In the steady state, one instruction will get executed every clock cycle.
 - Speedup = $4.35 / 1.25 = 3.48$

For the non-pipelined processor we can calculate the average instruction execution time by multiplying the clock cycle time with the average cycles per instruction. Now clock cycle system it was 1 nanosecond, and average cycle time you see it was mentioned 50% are ALU, 15% are branch, and 35% are memory, and they take 4, 4, and 5 cycles respectively.

So, if you calculate it becomes 4.35, but for the pipeline processor as I had said that the clock cycle time is slowing down it becomes 1.25 nsec. Now assuming that in the steady state one instruction gets executed every clock cycle, the average instruction execution time will be equal to 1.25 nanosecond. So, speedup will be $4.35 / 1.25 = 3.48$.

(Refer Slide Time: 30:12).



Let us have a quick recap on the non-pipelined MIPS32 micro-operations that you have seen earlier. See in the IF we are doing this; we are fetching the instruction, incrementing the PC, but storing it in NPC. See NPC to PC is been transferred only during MEM. In ID we are fetching the registers, we are also pre fetching the immediate data and doing sign extension. In the EX stage we are doing some arithmetic operation, but depending on the type of instructions it can vary; for memory we are just adding Imm to A to calculate the effective address of the memory, for ALU operation we simply operate on A and B, for Immediate operation we operate on A and Imm, for branch operations we are calculating the branch target address, and also we are evaluating the branch condition whether it is a taken or non-taken branch. And again during MEM operation if it is a load instruction we do a memory read, if it is a store instruction we do a memory write, if it is a branch instruction we simply check the condition, and based on that we update the PC.

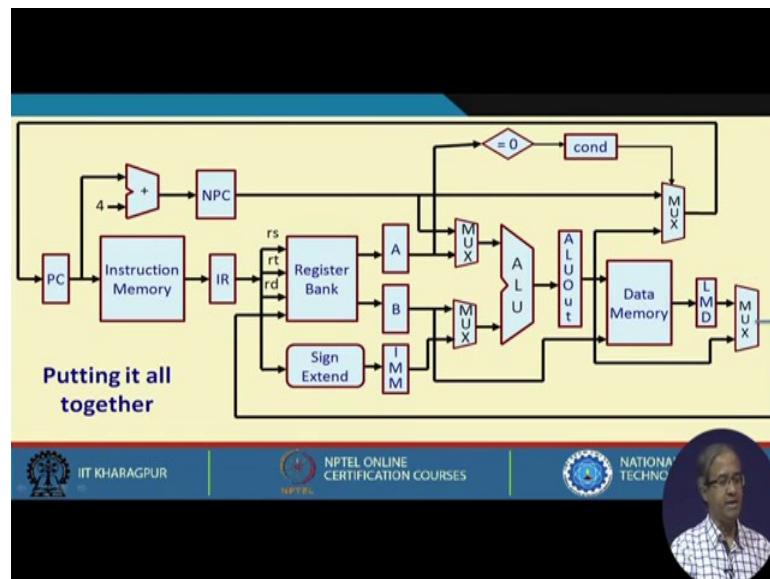
So, PC is getting updated much later here this is one observation.

(Refer Slide Time: 32:07).



And the WB stage we are storing either the ALU output, or for load instruction the output from the memory into the register.

(Refer Slide Time: 32:20).



Now we have seen that all the things taken together the data path looks like this, and next what we will see is that if we want to make this into a pipeline, what are the changes that are required? These are a few things we shall be seeing in our subsequent lectures.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 54
MIPS32 Pipeline (Contd.)

In our last lecture we had identified some of the requirements that need to be satisfied or fulfilled in order have a pipelined version of our MIPS32 architecture. Now today we shall be looking into that very specifically, how we can modify the non pipeline data path of the MIPS32 that we have seen earlier into an equivalent pipeline version which works. Of course, subsequently we shall try to identify what are the problems or conflicts that can come in, and how we can solve them.

(Refer Slide Time: 01:10)

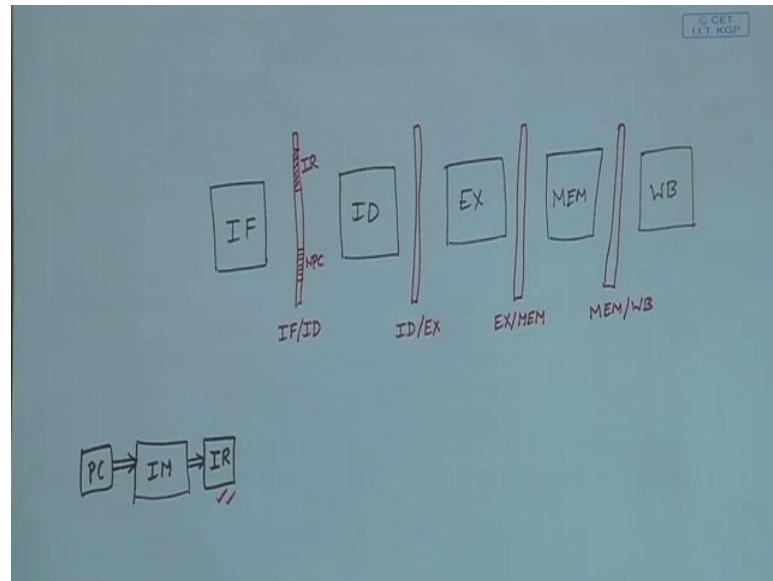
Micro-operations for Pipelined MIPS32

- Convention used:
 - Many of the temporary registers required in the data path are included as part of the inter-stage latches.
 - IF/ID: denotes the latch stage between the IF and ID stages.
 - ID/EX: denotes the latch stage between the ID and EX stages.
 - EX/MEM: denotes the latch stage between the EX and MEM stages.
 - MEM/WB: denotes the latch stage between the MEM and WB stages.
- Example:
 - ID/EX.A means a register A that is implemented as part of the ID/EX latch stage.

IIT KHARAGPUR | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

We shall continue with our discussion on MIPS32 pipeline. First we shall be defining some conventions.

(Refer Slide Time: 01:34)



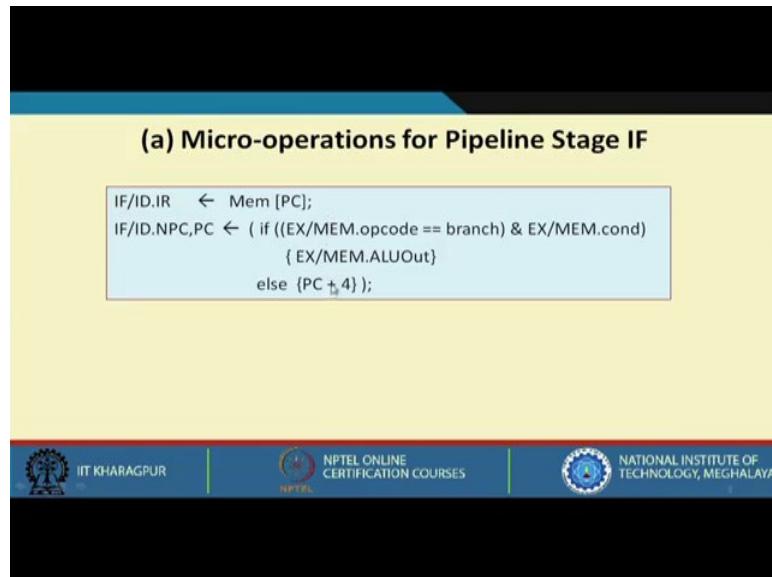
The conventions are like this. You see the five stages; IF, ID, EX, MEM and WB. Now what I am saying is that, you also require some latch stages in between, some register stages. The convention that I am talking about is, see each of these latch stages we are giving some specific names. For example, this latch stage between IF and ID, we call them as IF/ID. This latch stage, we call as ID/EX, this one as EX/MEM and this one as MEM/WB. And another thing, in the non-pipelined version if you recall in the IF stage, what we are doing? We are saying that there is an instruction memory. This is being accessed by some address from PC and some data is read, it is loaded into a register called IR; instruction register.

But here what we are saying is that, we need not use separate registers like IR for example, in every stage. Rather in this IF/ID stage itself we can earmark a portion of it. Say these are register type, these consists of certain number of bits. We can here mark certain number of bits that can be our IR; for example. We can here mark certain number of bits here that can our NPC; for example. The idea is that we shall not be using these registers separately within these stages, rather many of these registers we shall be integrating along with this latches.

Let us come back to the conventions. I have mentioned IF/ID, ID/EX, etc. are the names of the latches and the naming convention is like this. If I write ID/EX.A, this means that

I am referring to a register A, which is implemented as part of the ID/EX latch. This is the naming convention that shall be followed.

(Refer Slide Time: 04:46)



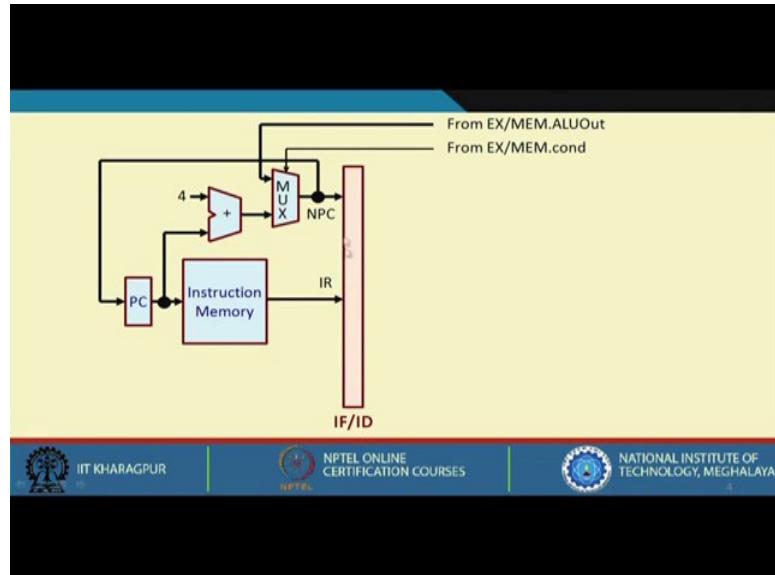
Let us look into the stages one by one. In the IF stage, one thing we mentioned that we need to increment PC by 4 here itself.

We cannot wait till MEM. So, our micro operation can be like this, we are doing an instruction fetch -- after fetching where we are storing? We are storing into IR, means IF/ID.IR. IR is part of the IF/ID latch. Similarly in the IF/ID.NPC, there is a register NPC as part of this while also there is separate register PC. PC is a separate register, “comma PC” is how I write. This is loaded with either this ALUOut from EX/MEM or PC + 4. Depending on some condition see, if it is a branch instruction and the branch condition is true then you will have to load PC with EX/MEM.ALUOut. If it is a branch instruction you have no way to know the branch target address; right now in IF, you will have to wait, but if it is a not branch instruction you can proceed. You can proceed with PC = PC + 4, and we are ready with the next stage, when the next instruction can be fetched.

We are doing exactly that. We are making a check if it is a branch instruction and condition, then we do this. You see this checking is not that difficult because we are already fetching it here, and this opcode equals to branch means we are checking a few bits in this IR. Whether it is a branch condition? Of course, these we are not checking here; we are checking EX/MEM.opcodes, so later. By default in IF will be going to the

else part PC + 4, but if it is a branch instruction and if the condition is a taken branch, then something else will be loaded. So, how will our IF stage look like now?

(Refer Slide Time: 07:31)



This is our IF and this is the IF/ID latch. We have a register here, this is IF/ID.IR. I am just showing IR by the side it, actually it is here. Similarly NPC is another register inside IF/ID.NPC, and PC is a separate register of course. PC is used to access memory and this multiplexer selects either PC + 4 or this ALUOut, which is coming from EX/MEM. By default PC + 4 will be loaded. This is how we are making the modification the IF stage so that PC = PC + 4 is done here itself, but if it is a branch instruction there will have to be some corrections to be made.

(Refer Slide Time: 08:35)

(b) Micro-operations for Pipeline Stage ID

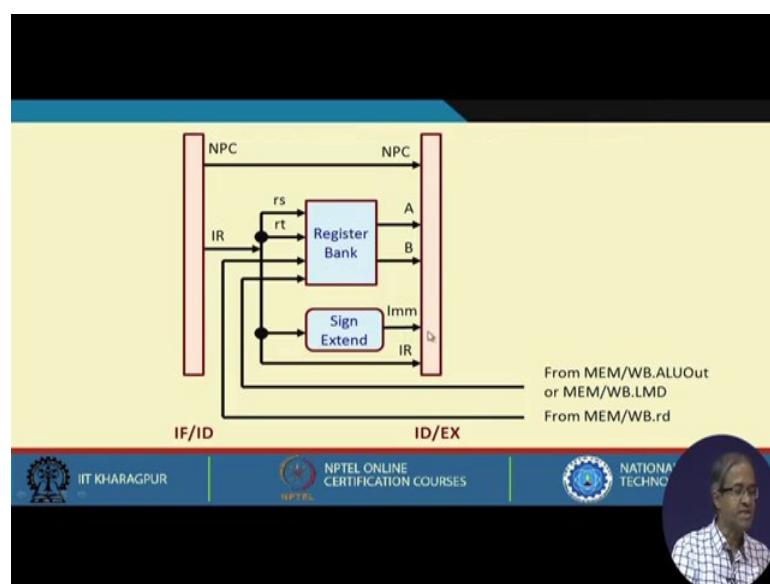
$$\begin{aligned} \text{ID/EX.A} &\leftarrow \text{Reg [IF/ID.IR [rs]]}; \\ \text{ID/EX.B} &\leftarrow \text{Reg [IF/ID.IR [rt]]}; \\ \text{ID/EX.NPC} &\leftarrow \text{IF/ID.NPC}; \\ \text{ID/EX.IR} &\leftarrow \text{IF/ID.IR}; \\ \text{ID/EX.Imm} &\leftarrow \text{sign-extend (IF/ID.IR}_{15..0}\text{)}; \end{aligned}$$

NPTEL ONLINE CERTIFICATION COURSES



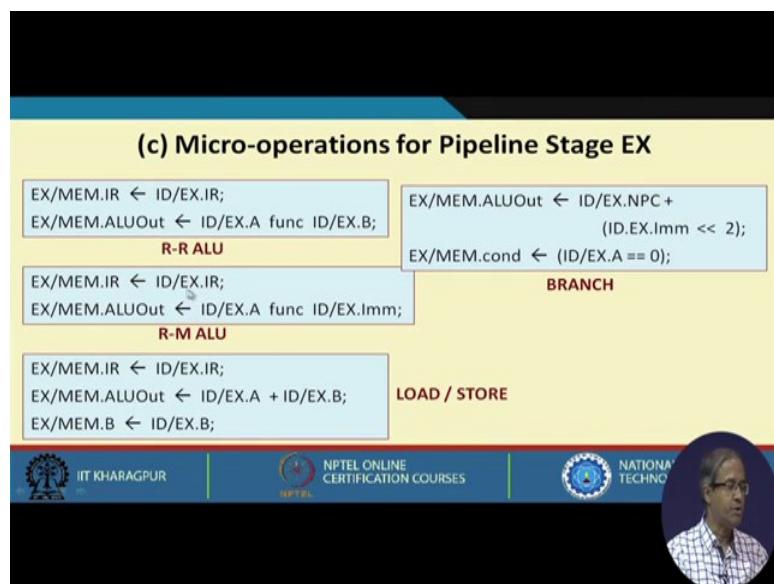
Now, for the ID stage what were the operations? We are doing some register prefetch. You see in the previous one already this IR is in IF/ID. So, from there we are checking the different bits; source registers and Imm. That is why IF/ID.IR; the rs field of that, the rt field of that. We are prefetching the two registers and storing them in A and B registers that are part of ID/EX latch. NPC you are not doing anything, we are forwarding this NPC from IF/ID to ID/EX because this will be required later. Similarly IR we are doing the same thing; from IF/ID we are simply forwarding it to ID/EX. And Imm, IFID.IR, we are doing sign extension and we are storing in ID/EX.Imm.

(Refer Slide Time: 09:48)



Whatever I have written here can pictorially be explained like this. You see in the previous IF/ID stage, I have this NPC and IR, but in the next one ID/EX, I have NPC, A, B, Imm, IR. So, IR I am simply coping, NPC I am simply coping, this A, B I am reading from the register bank, this Imm I am also using sign extension. You see this is exactly what we have done. Fetch, sign extension these are simple copies and these two arrows that are shown, these are actually for the registers. This will come from later stages. Here only we have shown register read but because the register bank is located in stage ID, the register write signal should also come here, from MEM/WB.rd and MEM/WB.LMD or MEM/WB.ALUOut.

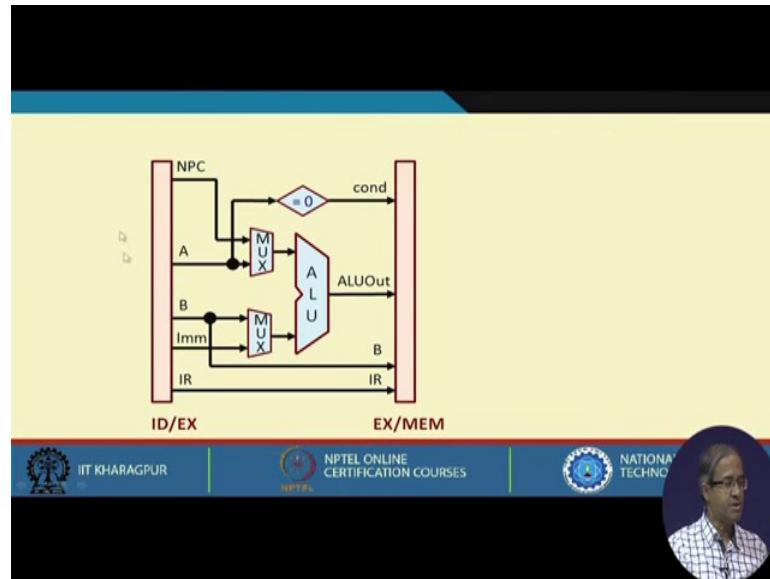
(Refer Slide Time: 10:54)



Now, let us come to the EX stage. Here the operation is different depending on the kind of instruction. If it is register to register ALU, then we are doing some operation on A and B and storing it into ALUOut that is part of EX/MEM, and IR again we are forwarding to the next latch.

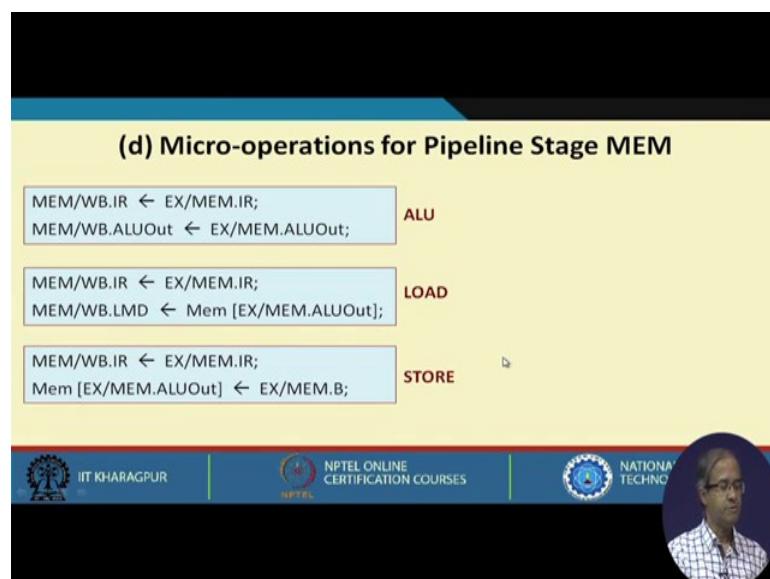
For register memory ALU, one is register and other is immediate data. We are actually adding a register content or subtracting some operation depending on the function to an immediate data. And for LOAD and STORE we will have to calculate the effective address. So, you are forwarding it to this ALUOut you are calculating. And for branch, again you are doing that same thing; calculating the branch address and checking the conditions.

(Refer Slide Time: 12:31)



In terms of the pipeline implementation, this EX stage look like this. See here the operation may look quite complex, but in terms hardware it is very simple. In the ID/EX stage you had NPC, A B, Imm, IR. This IR gets copied and B gets copied because you need B for some instruction. ALUOut gets stored and cond also gets stored. This is the EX stage, just an ALU, some multiplexers and a zero condition checker.

(Refer Slide Time: 13:19)

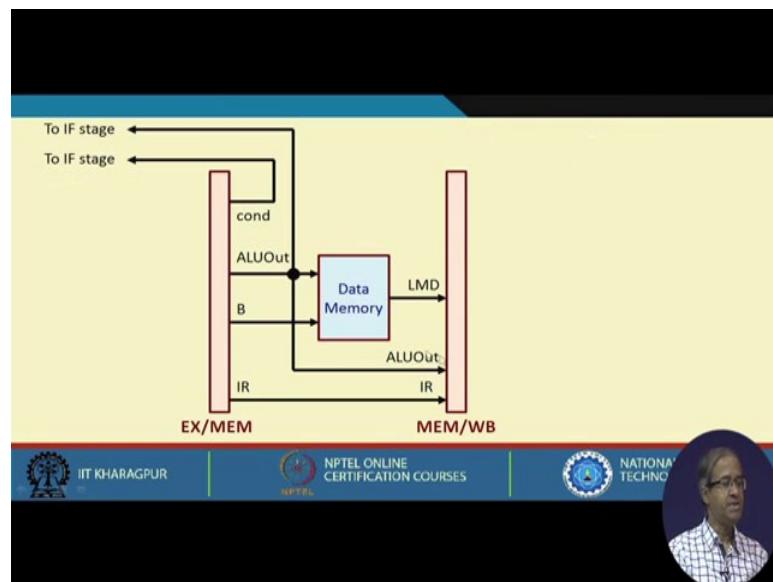


Then when you come to the MEM stage, for ALU operation you again simply forward IR to the next latch and also you forward ALOut here because for ALU instruction MEM

does not do anything. It is simply forwards it, for load operation you are doing a memory read and IR is forwarded, for store operation you are doing a memory write and IR is forwarded.

Here memory address is in EX/MEM.ALUOut, EX/MEM.ALUOut. So, while reading you are reading, storing it in LMD and while writing you are taking it from EX/MEM.B.

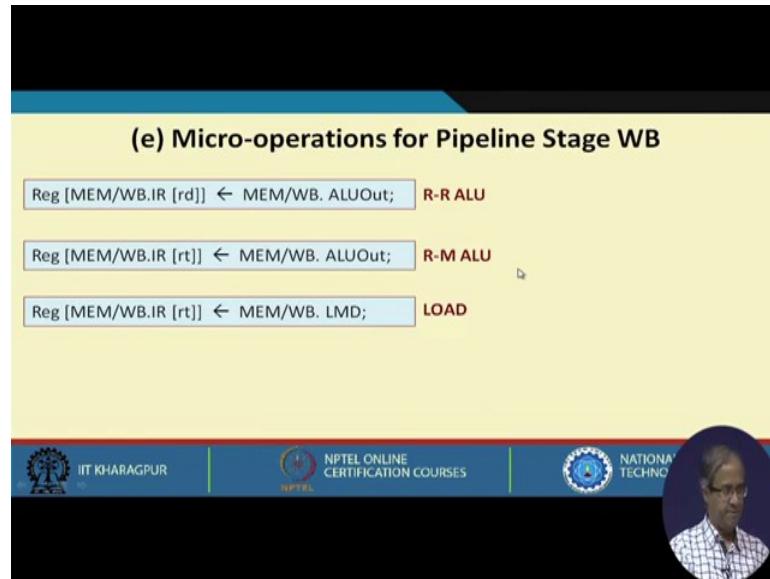
(Refer Slide Time: 14:04)



Your MEM stage is simply like this, it consists of a memory. Your address is coming from here, data is coming here. For reading, you read into LMD, for writing it will come from B and IR is forwarded, ALUOut is forwarded. Because you are reading, these values will be required in the next stage, that is why you have to forward these. And one thing just to recall that, here this ALUOut and the cond signals that are generated in MEM, these are fed back to the IF stage because here at this stage you have known for sure whether it is a branch instruction or not and you whether the condition is true or false.

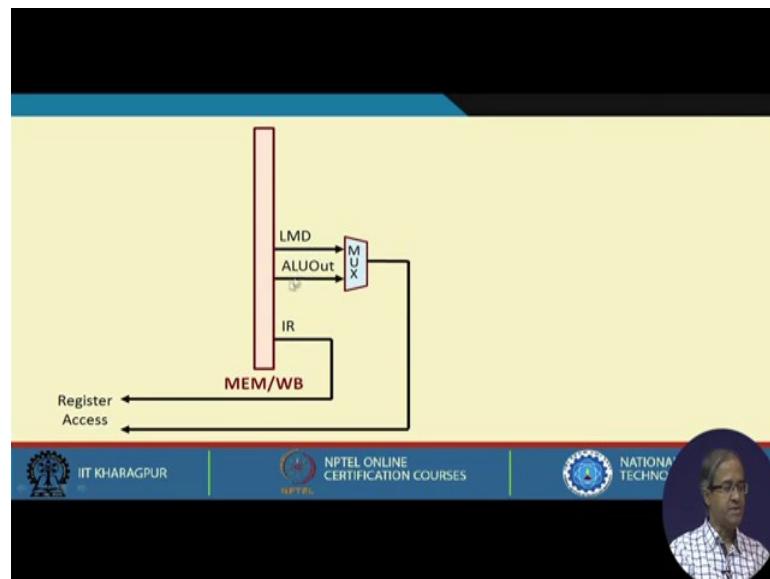
So, you send them to the IF stage indicating the condition of the branch instruction and what is the target address of the branch so that PC can be updated accordingly.

(Refer Slide Time: 15:12)



Lastly in the in WB stage you are doing some write into the register banks depending on R-R, R-M or LOAD instruction whatever you are doing.

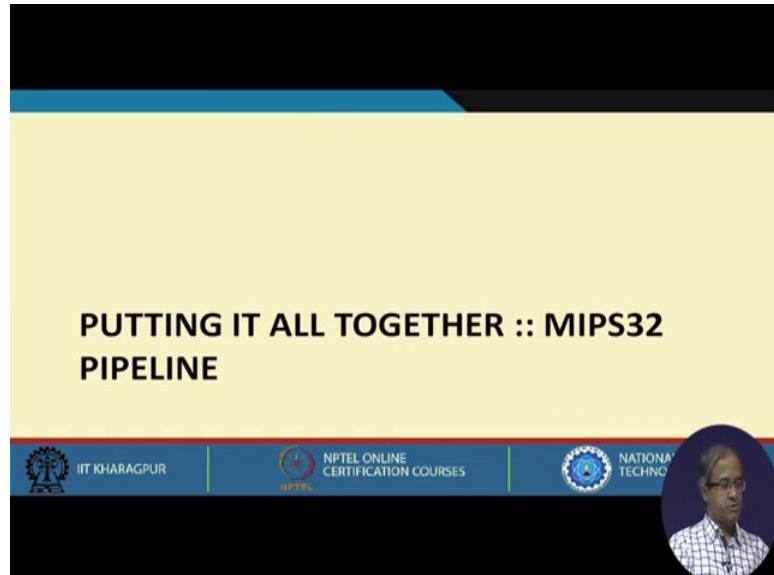
(Refer Slide Time: 15:35)



This stage is fairly simple, it takes the LMD and ALUOut from here, and IR. Why does it require IR? Because you see depending on instruction you need these fields rd or rt because they have to be written into the register. So, these signals will have to go back to the register bank. You see here these signals are coming from later in the WB stage. In

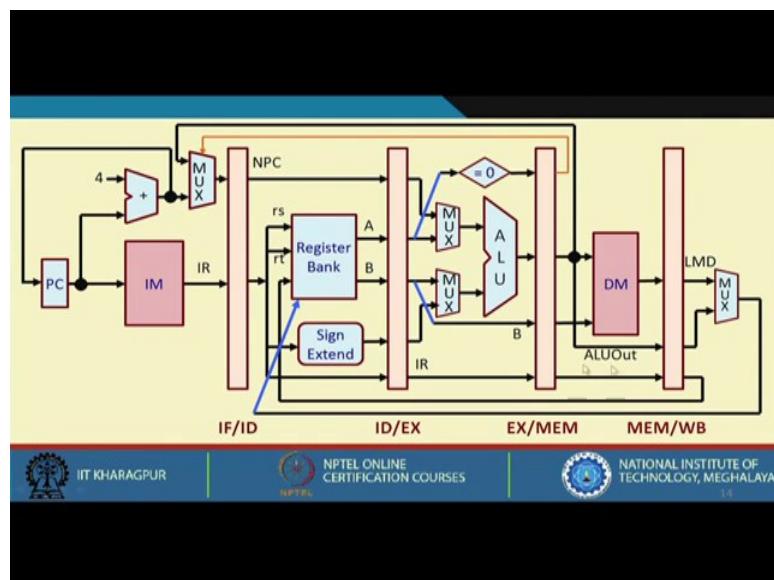
the ID stage and this MUX selects either LMD or ALUOut depending on the type of instruction. It is either LMD or ALUOut; one of them is going to be written.

(Refer Slide Time: 16:44)



The control circuit will be selecting this multiplexer. Putting all the things together the MIPS pipeline looks something like this.

(Refer Slide Time: 16:49)



Well this is a little complex; I have tried fit it into a slide. You see, this is your IF. Well here I have shown this PC from here, but actually this can be from the output of the MUX also. Well you can either do it always + 4 you load it, or you can make it as

default, it does not matter really. Earlier I have shown that this connection is coming from here, but here I am showing that directly $PC = PC + 4$ you are doing here. But if it is a branch, later it is known then from the output of the MUX you have to make some changes. For that change this connection has to be taken from the output of the MUX. Here just for simplicity I have not shown the exact connection, this you remember and this is our instruction memory and this is your ID stage; register bank, two reads and one write, and sign extension. These rs , rt are for reading and the third one for writing, and this is the data to be written to the register, And this is the EX stage; MUX selects either NPC for branch instruction, or A for other instruction, A or sign extension for immediate operations. This depends on instruction, MUX will be selecting one of the inputs, it goes here. So, cond is getting generated here, this cond will be selecting this MUX.

Actually this will be fed here; data memory MUX. So, this is how the total MIPS32 pipeline implementation looks like. You see it is not at all complex; it is very similar to our earlier non-pipeline diagram with some minor changes. Just these four register stages included, but we shall look at some other issues or problems called hazards and you should see how some of these hazards can be overcome or tackled.

So, we are stopping now, but we will be looking at just assuming that we have this base MIPS32 pipeline available with us. We shall be exploring the various hazard situations and try to come up with some solution. How to solve them? We shall be continuing with the hazard detection and avoidance in our subsequent lectures.

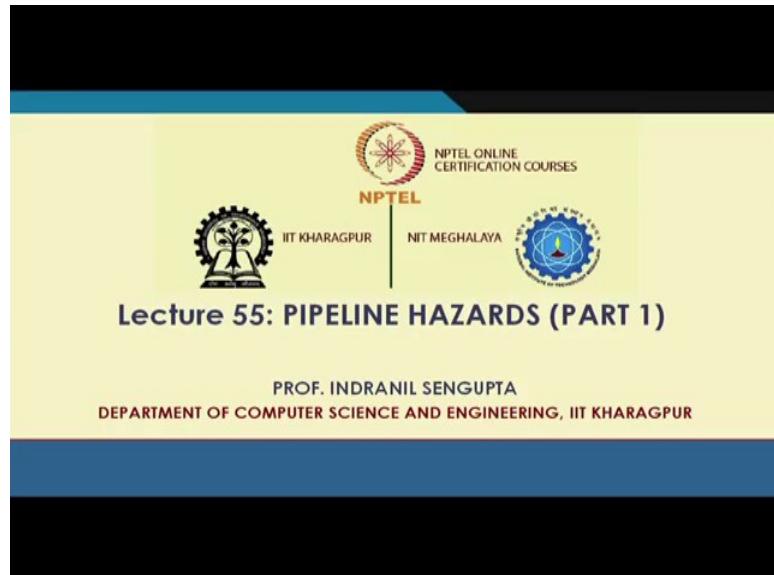
Thank You.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 55
Pipeline Hazards (Part 1)

In the last lecture we have seen the pipeline structure for the MIPS32 processor. You may say that it is an ideal pipeline as we have not considered many of the real problems or conflicts that can arise and cause the so-called hazards during instruction execution in the pipeline.

(Refer Slide Time: 00:57)



From this lecture onwards we shall be looking at these conflicts or hazards, that can impact or degrade the performance on a pipeline to a great extent, and what are the approaches we can use or follow to avoid such degradation to the extent possible. We start our discussion on pipeline hazards in this lecture.

(Refer Slide Time: 01:23)

Introduction

- Pipeline Hazards:
 - Ideally, an instruction pipeline should complete the execution of an instruction every clock cycle.
 - Hazards refer to situations that prevent a pipeline from operating at its maximum possible clock speed.
 - Prevents some instructions from executing during its designated clock cycle.
- Three types of hazards:
 - a) **Structural hazard:** Arise due to resource conflicts.
 - b) **Data hazard:** Arise due to data dependencies between instructions.
 - c) **Control hazard:** Arise due to branch and other instructions that change the PC.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us try to first understand what is a pipeline hazard. We have seen an ideal pipeline. An instruction pipeline in the ideal sense should complete the execution of one instruction every clock cycle, but when we say there is a hazard, it means some kind of a scenario or some occurrence of some events because of which we cannot operate the pipeline at its maximum possible speed. Because you see there are several instructions that have entered the pipe already, several instructions are in various stages of execution, there can be some dependencies between the instructions. They may be trying to access some common resources because of which there can be a conflict, because of branch instructions I have said earlier we may have to wait till we know the outcome of the branch and the target address. There are several such instances that may prevent a pipeline to work with its maximum possible capability or speed.

Loosely speaking such situations are called hazards, they prevent a pipeline from operating at its maximum possible clock speed, which means that we cannot feed an instruction every clock cycle. Such hazards can prevent some instructions from executing during its designated clock cycle, means suppose an instruction was suppose to be entering the pipeline now, but maybe I will have to make it wait for one clock cycle and make it enter the pipeline in the next cycle. Broadly speaking hazards can be classified into 3 types; structural, data and control.

Structural hazard these arise due to resource conflicts, like one example you have already seen in the memory access -- instruction fetch IF and MEM, two instructions may be trying to access instructions and data, that is kind of a structural hazard. If there was a single memory module then one of the instructions have to wait, but because we have used separate instruction and data caches they can proceed together.

Because of data dependencies between instructions, say one instruction is producing a result may be the next instruction is using that, such dependencies are called data hazards. And control hazards arise due to branch and other instructions like interrupts that changes the program counter.

(Refer Slide Time: 04:54)

What happens when hazards appear?

- We can use special hardware and control circuits to avoid the conflict that arise due to hazard.
- As an alternative, we can insert *stall cycles* in the pipeline.
 - When an instruction is stalled, all instructions that *follow* also get stalled.
 - Number of stall cycles depends on the criticality of the hazard.
 - Instructions *before* the stalled instruction can continue, but no new instructions are fetched during the duration of the stall.
- In general, hazards result in performance degradation.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY MADRAS

So, what happens when such hazards show up? Well there are several alternate strategies, some of them use some special hardware to detect the hazard. This is one alternative; some hardware will be there, which will be trying to detect such situations that can result in hazard, and try to overcome them automatically. But as an alternative we can simply insert stall cycles without using special hardware, this is a cheaper alternative. Like suppose I find that there is a hazard. If I allow the next instruction to enter the pipe there will be an clash somewhere. I hold the instruction back, maybe I feed it in the next clock not in the present clock. If I do that I say that I am inserting a stall cycle, for one cycle I am not feeding anything in the pipeline, I am stalling the pipeline for one cycle and then again I am feeding the next instruction after that. Then the hazard

will not show up. This is one simple study, but if I follow this principle -- suppose I stall one instruction then all the instructions that follow this particular instruction will also get stalled because unless this instruction enters the pipe those instructions will not be able to enter the pipe. Depending on the criticality of the hazard we shall be seeing the various situations where number of stall cycles can vary.

And the important thing to notice is that whenever we are inserting stall cycles like this, the understanding is that the instructions which have already entered the pipeline run in the process of execution, they can proceed normally there is no problem, but I am holding back the instructions that are following, I am inserting one or more stall cycles. They will be delayed, and after the stall cycles they will be fed into the pipeline. This is what I just now said, instructions before the stall instruction can continue, but no new instructions can be fetched during the duration of the stall. In this strategy, we may have to insert such stall cycles and this kind of hazards will result in performance degradation, because we are not allowing instructions to be fed or entering the pipe in every clock cycle that is why we are not able to get the full pipeline performance that is possible in the ideal case.

(Refer Slide Time: 08:38)

Calculation of Performance Degradation

$$\text{Pipeline speedup} = \frac{XT_{\text{nopipe}}}{XT_{\text{pipe}}} = \frac{CPI_{\text{nopipe}} \times C_{\text{nopipe}}}{CPI_{\text{pipe}} \times C_{\text{pipe}}} = \frac{C_{\text{nopipe}}}{C_{\text{pipe}}} \times \frac{CPI_{\text{nopipe}}}{CPI_{\text{pipe}}}$$

- We can treat pipelining as a way to reduce the CPI, or reduce C.
 - Let us consider that we are decreasing the CPI.
- The ideal CPI of an instruction pipeline can be written as:

$$\text{Ideal CPI} = \frac{CPI_{\text{nopipe}}}{\text{Pipeline Depth}} \rightarrow \text{Speedup} = \frac{C_{\text{nopipe}}}{C_{\text{pipe}}} \times \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{CPI_{\text{pipe}}}$$

 IIT KHARAGPUR |
 NPTEL ONLINE CERTIFICATION COURSES |
 NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us make a simple calculation. With respect to a non-pipelined version suppose you are trying to estimate the speedup in a pipeline. A simple measure of speedup is to divide the execution time of the non-pipelined version with the execution time of the pipeline

version. How I can calculate this execution times? One simple way will be to multiply the CPI of the non-pipelined version, with the CCT, assuming that the number of instructions are the same in the both the cases. Similarly for the pipeline version CPI for the pipeline version multiplied by the CCT of the pipeline version.

Now when we talk about pipelining we can either say that we are trying to reduce the CPI or we can argue that we are trying to reduce C, but actually reducing C is not true because we are not really reducing the clock cycle time, but you can argue in some way that in a non-pipelined version you had the whole computation as a block, as if your clock was running five times slower.

But now by making a pipeline with five stages, you are making the clock run five times faster that may be one argument, but a better argument that we will be following here is that we are not saying that we are making clock faster, but we are saying we are making CPI smaller.

Another thing is the ideal CPI. In the absence of any hazards it will be equal to the CPI of the non-pipelined version divided by pipeline depth. Pipeline depth means number of stages in the pipeline, now in our MIPS example pipeline depth is 5. In the ideal case we are getting a speedup of 5, and CPI is reducing by 5 times. The expressions follow.

(Refer Slide Time: 12:06).

- If we restrict ourselves to stall cycles only, we can write:

$$CPI_{\text{pipe}} = \text{Ideal CPI} + (\text{Pipeline Stall Cycles per Instruction})$$
- We can thus write:

$$\text{Speedup} = \frac{C_{\text{nopipe}}}{C_{\text{pipe}}} \times \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles / instruction}}$$
- If we ignore the increase in clock cycle time due to pipelining (i.e. $C_{\text{nopipe}} = C_{\text{pipe}}$)

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles / instruction}}$$



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



NATIONAL
INSTITUTE
OF
TECHNOLOGY



Now suppose we are using the method of inserting stall cycles for removing hazards. The actual CPI of the pipeline will obviously be greater than the ideal CPI. It will be ideal CPI + average pipeline stall cycles per instruction; this will be the impact of the hazard on the average case. So, CPI_pipe you can write like this.

This is an expression that you can use to estimate this speedup for a pipeline in presence of stall cycles. We shall be using this later in some examples.

(Refer Slide Time: 13:55)

(a) Structural Hazards

- They arise due to resource conflicts when the hardware cannot support overlapped execution under all possible scenarios.
- Examples:
 - Single memory (cache) to store instructions and data :: while an instruction is being fetched some other instruction is trying to read or write data.
 - An instruction is trying to read data from the register bank (in ID stage), while some other instruction is trying to write into a register (in WB stage).
 - Some functional unit (like floating-point ADD or MULTIPLY) is not fully pipelined.
 - A sequence of instructions that try to use the same functional unit will result in stalls.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL TECHNOLOGY

Let us look at the hazards now one by one. Structural hazards to start with we said that they arise due to resource conflicts. If I have a single copy of a resource and two instructions or two stages are trying to use the same resource, we have structural hazard. We have seen two examples earlier, one is for memory and the other is for register banks, the ID stage is trying to read from register, WB stage is trying to write into a register. If the hardware does not support concurrent or overlapped execution, then structural hazards will show up. These two examples already we have seen. If we have a single L1 cache not separate instruction and data, then when an instruction is being fetched and some other instruction is trying to read or write, the next instruction will have to wait. So, you have to insert a stall cycle.

Similarly an instruction is trying to read data from the register bank while some instruction is trying to write a register. Again if you do not follow that principle I mentioned earlier, that you do write in the first half of the clock and read in the second

half of the clock, then the instruction that is trying to read data will have to wait -- you will have to insert stall cycles in that case.

If you do not take precautions or put in some additional hardware to detect and avoid this kind of hazard, your stall cycles will get inserted; otherwise your instruction execution process will not be correct. And another example I am giving, let us say some of the functional units like floating point add or multiply, they are not fully pipelined. And suppose there are two instructions one by one consecutive they are trying to use floating point add or multiply. Because they are instructions that consume more than one cycles they will result in stall cycles, because when say the first instruction is doing multiply they will be using multiple clock cycle during the EX stage, because multiply cannot finish in one cycle. So, it is consuming the EX stage for more than one cycle. If the next instruction is also requiring floating point multiplication, it will have to wait.

(Refer Slide Time: 17:21)

Instruction	1	2	3	4	5	6	7	8
LW R1,10(R2)	IF	ID	EX	MEM	WB			
ADD R3,R4,R5		IF	ID	EX	MEM	WB		
SUB R10,R2,R9			IF	ID	EX	MEM	WB	
AND R5,R7,R7				IF	ID	EX	MEM	WB
ADD R2,R1,R5					IF	ID	EX	MEM

Let us take some examples. Let us say we have a single port memory system; that means, I do not have separate instruction cache and data cache and I have a example sequence of instructions, let us say I have a load followed by a sequence of arithmetic and logic instructions. In the ideal case instruction execution will proceed like this.

But as I said this load instruction will be trying to access memory here, do a load, and this instruction will try to do the instruction fetch here in IF. This will be a clash and be a structural hazard. Suppose we have not replicated the memory. What is the solution? I

can insert a stall cycle here. The control circuit will automatically detect that there is a conflict, this instruction is trying to access memory this was a load instruction. So, you cannot do an instruction fetch here. It will automatically insert a stall cycle and it will resume the instruction fetch in the next cycle. The instructions that follow will also incur a one cycle delay, that is what I said that you stall an instruction, and all instructions that follow will also be stalled by that equal number of cycles.

Here we show that how we can insert a stall cycle to eliminate this structural hazard.

(Refer Slide Time: 19:30)

Example 1

- Consider an instruction pipeline with the following features:
 - Data references constitute 35% of the instructions.
 - Ideal CPI ignoring structural hazard is 1.3.

How much faster is the ideal machine without the memory structural hazard, versus the machine with the hazard?

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles / instruction}}$$

$$\text{Speedup}_{\text{ideal}} = \frac{1.3 \times \text{Pipeline Depth}}{1.3 + 0}$$

$$\text{Speedup}_{\text{real}} = \frac{1.3 \times \text{Pipeline Depth}}{1.3 + 0.35 \times 1}$$

$$\frac{\text{Speedup}_{\text{ideal}}}{\text{Speedup}_{\text{real}}} = \frac{1.65}{1.3} = 1.27$$



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



NATIONAL
TECHNO



Now, let us work out a simple example. We consider an instruction pipeline, where data references constitute 35% of the instructions, which means load and stores. So, out of all instructions 35% are load and stores, and ideal CPI well ignoring structural hazard is 1.3.

Now the question is how much faster will be the ideal machine without the memory structural hazard. That means, we have already replicated I-cache and D-cache. We use this expression that we worked out earlier. For the ideal machine the ideal CPI is 1.3, and for the ideal machine there is no structural hazard. So, pipeline stall cycles per instruction will be 0. For the real machine what will happen? For all such data reference instructions there will be one cycle delay because of this stall. So, ideal CPI is 1.3 and pipeline stall cycle per instruction will happen 35% of the time and for each such case there will be 1 stall cycle inserted.

If you just calculate the speedup this becomes $1.65 / 1.3 = 1.27$. So, the ideal machine is 1.27 times faster than the real machine. This means that in reality whenever you have this kind of hazards, your performance degrades. There will be about 27% degradation in the performance. Now the question is why cannot we remove structural hazards by inserting additional hardware since this is so important. The question is how much is the cost to replicate the hardware, there are a few things like for example, I-cache and D-cache as I said in the example, this you can possibly do without much increase in cost, but there are a few things which you possibly cannot replicate. To reduce cost of implementation sometime structural hazards remain in certain cases. For instance pipelining all the functional units may be too costly, for instance the divider. The addition and subtraction and multiplication are relatively easier to implement in a pipeline, but division is more difficult.

(Refer Slide Time: 23:14)

Structural Hazards

- Why structural hazards appear in real machines?
 - To reduce the cost of implementation.
 - Pipelining all the functional units may be too costly.
 - If structural hazards are not frequent, it may not be worth the effort and cost to avoid it.
- Memory access structural hazard is quite frequent.
 - Makes use of separate instruction and data caches in the first level.

```

graph LR
    CPU[CPU] --- L1iCache[L1 iCache]
    CPU --- L1dCache[L1 dCache]
    L1iCache --- L2Cache[L2 Cache]
    L1dCache --- L3Cache[L3 Cache]
    L2Cache --- MainMemory[Main Memory]
    L3Cache --- MainMemory
  
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

And third thing is that you can bank on Amadahl's law again, you see that how frequently such structural hazard scenarios occur. If you see that such structural hazards are not that frequent then you may avoid the effort and cost, you may see that the floating-point division in a program appears very rarely. So, let us not invest anything to improve that to remove that hazard, let that hazard remain. Whenever there is a floating point division followed by another division let stall cycles be inserted, but for most of the scenarios such occurrences will not happen. We are trying to make the common case first, but because memory access structural hazard is very frequent, we replicate the

hardware because this is something we cannot compromise, because if we use a single integrated cache for every memory access we may have to use one stall cycle that is too expensive.

(Refer Slide Time: 24:30)

(b) Data Hazards

- Data hazards occur due to data dependencies between instructions that are in various stages of execution in the pipeline.
- Example:

Instruction	1	2	3	4	5	6
ADD R2,R5,R8	IF	ID	EX	MEM	WB	
SUB R9,R2,R6		IF	ID	EX	MEM	WB

R2 written here

R2 read here

Unless proper precaution is taken, the SUB instruction can fetch the wrong value of R2.


IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES


NATIONAL TECHNOLOGY FOUNDATION



Now, let us come to data hazards. Data hazard means there is some dependency among the instructions, let us say I have an add instruction which produces a result in R2, there is a following instruction which uses the value of R2. So, unless you take proper precaution you see normally what happens? Normally this add instruction will be writing the value of R2 in WB stage and subtract instruction will be prefetching all register operands in ID. So, it is trying to prefetch before the value has been written.

If you do not take proper precaution this subtract instruction can fetch the old value of R2, and not the latest one because R2 is written here and you are trying to read R2 in the ID phase earlier in time, this is what data hazard means. So, you possibly have to insert two or more stall cycles, because unless WB completes you cannot use ID, maybe this ID you have to shift here, see register read & write you can do in parallel, but at least you have to shift it here, but you will see that we can do better.

(Refer Slide Time: 26:06)

- Naïve solution by inserting stall cycles:
 - After the SUB instruction is decoded and the control unit determines that there is a data dependency, it can insert stall cycles and re-execute the ID stage again later.
 - 3 clock cycles are wasted.

A naïve solution is by inserting stall cycles like that after the subtraction is decoded, we see that here also we are using R2, same operand as the previous one. We will be waiting till WB completes, by inserting stall cycles. In the naïve implementation 3 stall cycles should be used.

(Refer Slide Time: 26:29)

- Naïve solution by inserting stall cycles:
 - After the SUB instruction is decoded and the control unit determines that there is a data dependency, it can insert stall cycles and re-execute the ID stage again later.
 - 3 clock cycles are wasted.

Instruction	1	2	3	4	5	6	7	8
ADD R2,R5,R8	IF	ID	EX	MEM	WB			
SUB R9,R2,R6		IF	ID	STALL	STALL	ID	EX	MEM
Instr. (i+2)			IF	STALL	STALL	IF	ID	EX
Instr. (i+3)				STALL	STALL		IF	ID
Instr. (i+4)				STALL	STALL			IF

While decoding you find out that R2 is here and already the earlier instruction has R2 as target. So, now you will have to use 3 stalls this will be converted into a stall, again two

more stalls, you will have to repeat ID again here. All subsequent instructions will be facing those stalls. So, 3 clock cycle are wasted.

(Refer Slide Time: 27:13)

- How to reduce the number of stall cycles?
 - We shall explore two methods that can be used together.
 - a) **Data forwarding / bypassing:** By using additional hardware consisting of multiplexers, the data required can be forwarded as soon as they are computed, instead of waiting for the result to be written into the register.
 - b) **Concurrent register access:** By splitting a clock cycle into two halves, register read and write can be carried out in the two halves of a clock cycle (register write in first half, and register read in second half).

The slide footer features three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Technology.

But fortunately because of the simplicity of the instructions set we can reduce the number of stall cycles. How you can do that? We shall explore two methods; one is called data forwarding or bypassing in a second one we have already seen concurrent register access this we have already seen. We are splitting a clock cycle in two halves, register writing is done in the first half and reading in the second half. The second one is data forwarding or bypassing; the idea is very simple. The first instruction that was generating the result, R2 was writing the result in WB, but if you look at the instruction execution process, the value of the result was already calculated at the end of EX because suppose it is an add instruction the actual addition is carried out during EX.

But the value is written to the register in WB. So, if we take the output of the ALU directly that is already calculated and by using some additional MUX you forward that value to the next instruction whenever you it needs it. This way you will not have to wait for the WB stage, you may get the value earlier -- this is called data forwarding or bypassing, by using additional hardware consisting of MUX. The data required can be forwarded as soon as they are computed, instead of waiting for the result to be written into the register.

(Refer Slide Time: 29:08)

Reducing Data Hazards using Bypassing

- Basic idea:
 - The result computed by the previous instruction(s) is stored in some register within the data path (e.g. ALUOut).
 - Take the value directly from the register and forward it to the instruction requiring the result.
- What is required?
 - Additional data transfer paths are to be added in the data path.
 - Requires multiplexers to select these additional paths.
 - The control unit identifies data dependencies and selects the multiplexers in a suitable way.



As I have said that the result computed by the previous instruction is stored in some register, it is just the output of the ALU -- there is a register ALUOut you recall. So, the value is there. You take the value directly from there, do not wait till WB, and forward to the instruction that require the result. To do this you need some additional connections or data transfer paths and some additional multiplexers. Your control circuit also becomes a little complex; it will have to analyze the source and the destination registers of the consecutive instructions, and it will automatically activate or deactivate this kind of forwarding paths whenever required.

(Refer Slide Time: 30:10)

Instruction	1	2	3	4	5	6	7	8	9
ADD R4, R5, R6	IF	ID	EX	MEM	WB				
SUB R3, R4, R8		IF	ID	EX	MEM	WB			
ADD R7, R2, R4			IF	ID	EX	MEM	WB		
AND R9, R4, R10				IF	ID	EX	MEM	WB	
OR R11, R4, R5					IF	ID	EX	MEM	WB

- The first instruction computes R4, which is required by all the subsequent four instructions.
- The dependencies are depicted by red arrows (result written in WB, operands read in ID).
- The last instruction (OR) is not affected by the data dependency.



Let us take an example here that shows lot of data dependency, like this ADD instruction generates R4, this R4 is used in all consecutive four instructions. So, there is a data dependency in all. In the normal case result will written in WB, but this instructions are trying to read them here, here, and here. The first instruction computes R4 that is required by all subsequent three instructions -- the dependencies are shown here, but the last instruction of course, it is using ID after WB, it is not affected. So, it is only the three consecutive instructions that need to be looked at.

You see this we have already solved by splitting the register access, we are saying writing is done in the first half of the clock cycle and reading is done in the second half of the clock cycle. So, this conflict is already resolved. Actually we are left with these two conflicts, but one thing you see it is possible to remove these two conflicts also. Why? Because the first instruction is calculating R4 at in the EX stage the result is available here.

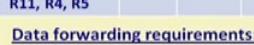
You need not have to wait till WB to be written into R4, you can directly take from here and forward it to the EX stage here where this SUB will be requiring. If this ADD will be requiring it, from here you can forward it.

(Refer Slide Time: 32:11)

Instruction	1	2	3	4	5	6	7	8	9
ADD R4, R5, R6	IF	ID	EX	MEM	WB				
SUB R3, R4, R8		IF	ID	EX	MEM	WB			
ADD R7, R2, R4			IF	ID	EX	MEM	WB		
AND R9, R4, R10				IF	ID	EX	MEM	WB	
OR R11, R4, R5					IF	ID	EX	MEM	WB

Data forwarding requirements:

- The first instruction (ADD) finishes computing the result at the end of EX (shown in RED), but is supposed to write into R4 only in WB.
- We need to forward the result directly from the output of the ALU (in EX stage) to the appropriate ALU input registers of the following instructions.
 - To be used in the respective EX stages, shown by the arrow.






Data forwarding means the value of R4 is actually getting calculated here, and you are forwarding the value like this. But as I have said you need to forward only to the two consecutive instructions, you do you really need beyond that. We need to forward the

result directly from the output of the ALU in the EX stage, to the appropriate ALU register of the following instruction. When the next instruction enters the EX stage, this instruction will already be in the MEM stage. From there the result has to be fed back to the input. How you can do that? By using some multiplexers and additional connection paths.

(Refer Slide Time: 33:06).

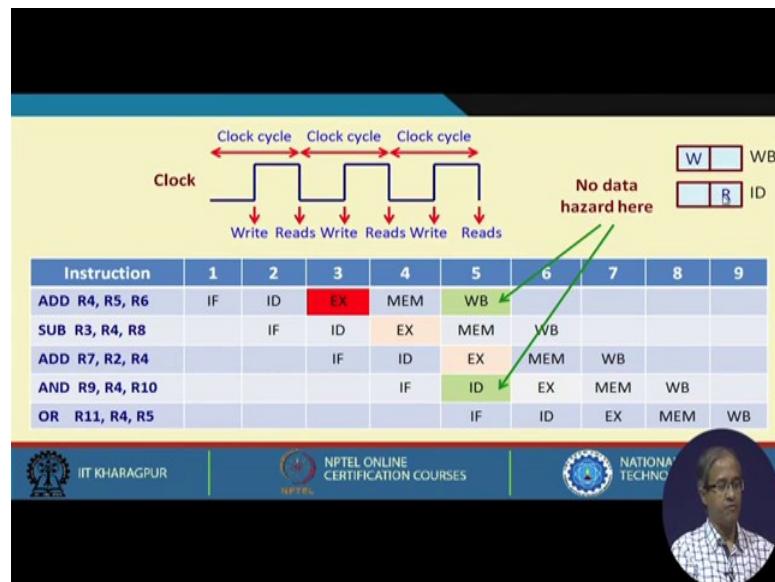
Reducing Data Hazard by Splitting Register Read/Write

- The data forwarding concept as discussed can solve the data hazards between ALU instructions.
- It is desirable to reduce the number of instructions to be forwarded, since each level requires special hardware.
 - In the forwarding scheme as discussed, we need to forward 3 instructions.
 - We can reduce this number to 2 by avoiding the conflict between the first and the fourth (AND) instruction, where WB and ID are accessed in the same cycle.
 - We perform register write (in WB) during the first half of the clock cycle, and the register reads (in ID) during the second half of the clock cycle.
 - The data hazard between the first and fourth instructions get eliminated.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL TECHNOLOGY

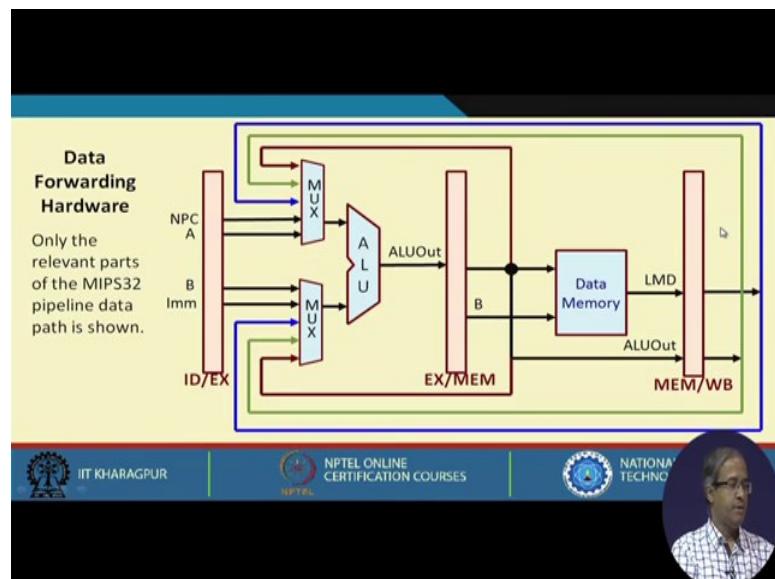
And this already we have seen earlier. By splitting register read & write, we have already avoided or reduced one dependency. In the naive forwarding there was the requirement of forwarding 3 instructions, but if you split register read write we reduce it to 2. This we have already discussed earlier in detail.

(Refer Slide Time: 33:37)



Because of this register splitting, this conflict has been avoided.

(Refer Slide Time: 34:05)



But for the others this is the forwarding hardware we require, this is the EX stage this is the ALU where the actual calculation is done. Normally the data input of the ALU is coming from the earlier stage, but because of the forwarding we use some additional connections from this ALUOut from here, because you do not know that which operand will be requiring this may be the first operand or the second operand. Similarly if there is a dependency of the second most instruction, the instruction will already be moved here.

So, this ALUOut is copied here, this value also is getting fed back here or here. Well here I am showing another feedback path also, this will have required for LOAD instruction.

But in the example you have not seen this yet, but this is the complete forwarding hardware. For LOAD instruction data will be read into the LMD, this LMD value is also forwarded. You see basically you need some additional inputs to the MUX, earlier it was at 2-to-1 MUX, now it has become a 5-to-1 MUX, this is the forwarding hardware that I am talking about. The first instruction that generates the result maybe it has come here now that partial result is here, the next instruction has come here and is trying to use that result, you forward it from here or the instruction generating result has come here, the next to next instruction has reached here. You forward it from here. All possible paths are shown. So, either you forward it from ALUOut here, ALUOut here, or for LOAD instruction from LMD here.

With this we come to the end of this lecture, what we have seen here is the importance of handling and tackling hazards because they can cause a very significant degradation in performance. We had looked at structural hazards and mentioned how they can be avoided by replicating resources wherever possible. Then you looked at data hazards. Data hazards are important and apparently without anything you are requiring 3 stall cycles for at least the ALU instructions with data hazards, but what we saw is that using forwarding hardware and by splitting the register access we are able to totally eliminate data hazard related stalls for ALU instructions.

In the next lecture we shall see that what will happen if there is a data hazard for a load followed by an ALU instruction, which uses the loaded value. There we shall see that it is not possible to totally eliminate stall cycle, but we can try to reduce it again. This we shall discuss in the next lecture.

Thank you.

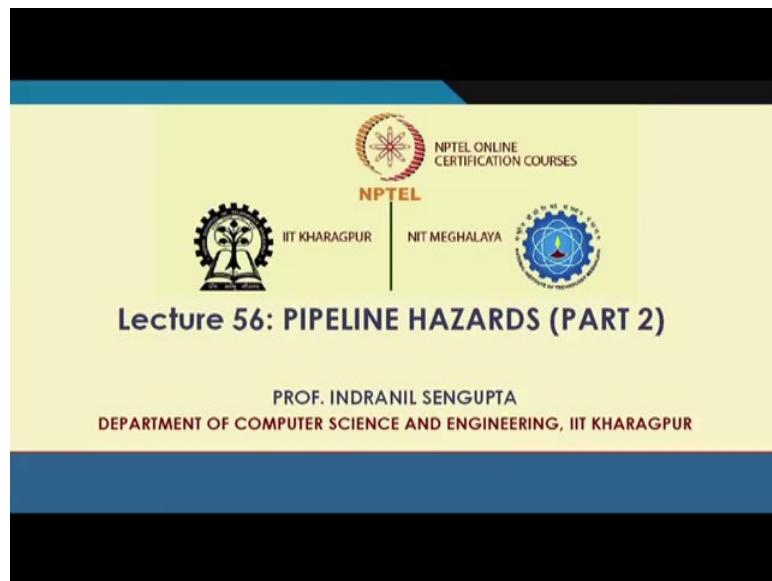
Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 56
Pipeline Hazards (Part 2)

In the last lecture we started our discussion on hazards in a pipeline. If you recall we talked about three kinds of hazards: structural hazard, data hazard and control hazard. We were discussing about the data hazards, and the examples that we took were for a case where there were data dependencies between ALU instructions. One of the ALU instruction was generating a result in a register, while the subsequent instructions were using those register values as inputs. We saw that apparently there was a problem that requires stalls, but we also came up with the solution. We proposed two things first we said that we can use some kind of a data forwarding hardware.

The earlier instruction, which can compute the result already in the EX stage, so that data value can be taken directly from the output of the EX stage, and through some MUX can forward it to the input of the following instructions that will be using their values. And the second thing that we talked about is that, we where we are permitting split access to the register bank. The clock cycle was divided into two halves, during the first half we are allowing writes to happen, following the second half we are allowing reads to happen.

(Refer Slide Time: 02:11)



We continue with the discussion here. In this lecture we shall again be talking about data hazard to start with.

(Refer Slide Time: 02:18)

Data Hazard while Accessing Memory

- In MIPS32 pipeline, memory references are always kept in order, and so data hazards between memory references never occur.
 - Cache misses can result in pipeline stalls.

Instruction	1	2	3	4	5	6
SW R2, 100(R5)	IF	ID	EX	MEM	WB	
LW R10, 100(R5)		IF	ID	EX	MEM	WB

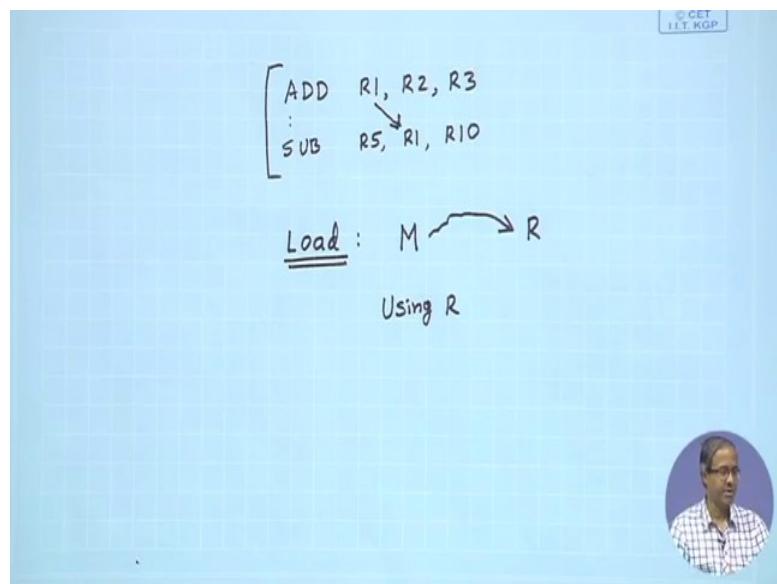
Accessed in order; no hazard

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NIT MEGHALAYA

But now we will be considering the situation when the hazard is occurring while accessing memory.

In the earlier cases you were saying that there was a hazard between ALU instructions.

(Refer Slide Time: 02:42)



It was something like this, there was one instruction let say ADD that was generating a result in register R1, and possibly a subsequent instruction was using the value of R1. Here there was a data dependency, now what we are saying that these kind of data dependencies can also happen because for memory access, like you can have a LOAD instruction. LOAD instruction means from the memory you are loading some value into some register, and then some subsequent instruction is using this register. This is what we mean by data hazard during memory access.

There are two situations we shall be looking at. The first situation you can see in this example here is that, there is a STORE instruction followed by a LOAD instruction. Here the apparent hazard or data dependency with respect to the memory location. The first instruction is writing some value, the content of the register R2 into a memory location whose address is $R5 + 100$, while the second instruction is trying to load that same value from that same memory location, and this value suppose to be loaded into some other register R10.

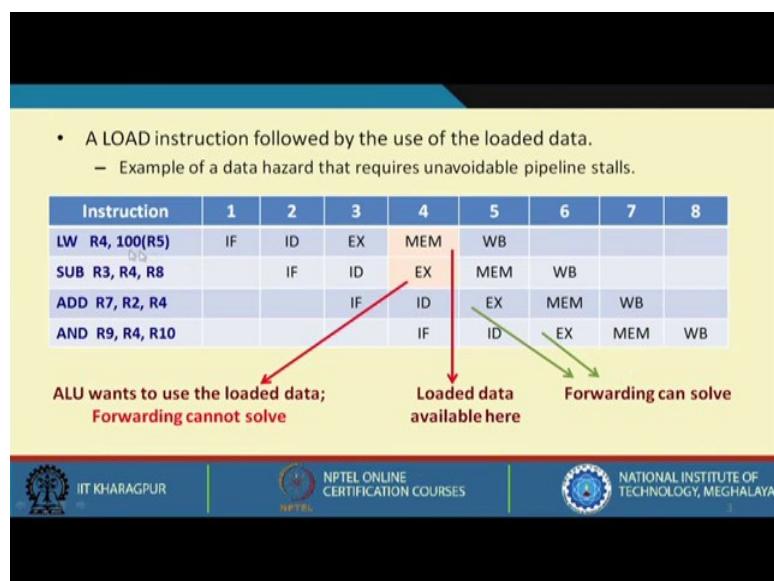
Now, whenever there is a data dependency like this, if you look at the MIPS32 pipeline structure you will see that all memory accesses take place only during the MEM stage. Memory accesses do not take place in the other stages in the pipeline, they take place only during the MEM stage. So, the previous instruction will reach the MEM stage earlier, the next instruction will reach the MEM stage one cycle later. That dependency is

automatically maintained. It never happens that the next instruction is requiring a data which is not yet written into the memory.

In this example the first instruction was using MEM here, second instruction would be using MEM here. This store will occur here load will occur here. So, there is no conflict, but this one thing which I had been missing of course, if there is a cache miss we are ignoring here. If there is a cache miss then the store instruction can take longer, then possibly this kind of hazard may occur, but again this cache miss is a relatively infrequent event. We always try to design a memory hierarchy that can give you a hit ratio, which will be about 98-99%.

So, the occurrence of the cache misses is very rare. We ignore that occurrence for the time being. As I said the data are accessed in order, there is no hazard taking place here.

(Refer Slide Time: 06:44)



Let us look at another situation where there is a LOAD instruction in the beginning. The LOAD instruction is loading the value of a memory location R5 + 100 into a register R4, and these subsequent instructions are using R4. This is a load instruction followed by the use of the loaded data. Here you can observe one thing, the LOAD instruction will be getting the data from the memory only at the end of the MEM stage.

But the next instruction SUB is requiring this R4, and the subtraction is supposed to take place during the EX stage. So, the value of R4 is required at the beginning of the EX

only. This is not possible because the memory value is loaded here and will be available only at the end, while the second instruction is trying to use that same value at the beginning of the EX.

This is a scenario where you have a data hazard that is unavoidable. If you implement forwarding hardware you cannot forward it, because you are trying to go back in time. The first instruction is loading the result only at the end of cycle 4, and the next instruction is requiring that value at the beginning of cycle 4. As I said the loaded data will be available here at the end of cycle 4 while the ALU for the second instruction wants to use the loaded data here at the beginning of cycle 4. So, data forwarding will not be able to solve this problem. You recall when there was a dependency between ALU instruction, it was possible to solve this problem using data forwarding because for the ALU instructions the result is computed in the EX stage only, not in the MEM stage.

But here we are talking about one time step ahead, only at the end of the MEM stage will our data be made available to us. This is one scenario where this stall cycle will be unavoidable, but for the following instructions let say this ADD and the AND they are also trying to use R4, but here you can use forwarding, because the loaded value is already available at the end of R4, from there you can simply forward it to this EX and also you can forward it to this EX because it is ahead in time. By using that same forwarding hardware we talked about earlier we can solve the problem for the subsequent instructions. The only problem is for a load, followed by its immediate use. There has to be one stall cycle in between.

(Refer Slide Time: 10:14)

- What is the solution?
 - As we have seen the hazard cannot be eliminated by forwarding alone.
 - Common solution is to use a hardware addition called *pipeline interlock*.
 - The hardware detects the hazard and stalls the pipeline until the hazard is cleared.
 - The pipeline with stall is shown.

Instruction	1	2	3	4	5	6	7	8	9
LW R4, 100(R5)	IF	ID	EX	MEM	WB				
SUB R3, R4, R8		IF	ID	STALL	EX	MEM	WB		
ADD R7, R2, R4			IF	STALL	ID	EX	MEM	WB	
AND R9, R4, R10				STALL	IF	ID	EX	MEM	WB

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

I said since we are not able to solve this problem by using forwarding, what we do; we have to insert some kind of a stall cycle. What we do is normally is that, there will be a special hardware that will be added to your data path to your decoding logic that will be called the *pipeline interlock*. Here only you will be able to know that your operands are R4 and R8, and already previous instruction is having R4 as the destination. The hardware keeps track of that that, which register was the destination in the previous instruction and which are the registers that have been used in the current instruction.

If it finds that there is a conflict like that, then a stall will be inserted in the pipeline -- in this case a one cycle stall. The second instruction will be decoded and found that it uses R4. So, it cannot start its EX here, it has to defer by one cycle and it can start the EX here. When one instruction gets deferred all subsequent instructions will also get deferred.

(Refer Slide Time: 12:15)

- A terminology:
 - **Instruction Issue:** This is the process of allowing an instruction to move from the ID stage to the EX stage.
 - In the MIPS32 pipeline, all possible data hazards can be checked in the ID stage itself.
 - If a data hazard (LOAD followed by use) exists, the instruction is *stalled* before it is *issued*.

The footer of the slide features logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Here we introduce a terminology called instruction issue, we say we are issuing an instruction.

You see the instruction that is operating on some data will typically be some ALU instruction: ADD, SUB, MUL like that. This operation is being carried out in the EX stage, before that in the ID stage you are decoding the instruction and trying to find out what the instruction type is, what are the input operands, and so on. When you are moving from ID to EX, it means that you are actually starting to execute the operation. It is during that phase you say that we are issuing the instruction.

When you say instruction issue this is actually the process, when an instruction is moving from the ID stage to the EX stage. Now one good thing about the MIPS32 pipeline is that because of the simplicity of the instructions, decoding is also very simple and the kinds of data hazards that are possible all can be detected at the end of the ID stage itself, because when you are decoding an instruction you already know that well this is an ADD instruction, these R4 and R8 are by input operands and so on. You can check whether any previous instruction is writing the value of R4 or R8; if so, there will be a hazard that will be detected. If the control unit finds out that the previous instruction was a load and the destination register is being used by the present instruction, it will know that at least one stall cycle is required.

(Refer Slide Time: 14:36)

The slide contains the following text and tables:

- A common example:
 $A = B + C$
- Pipelined execution of the corresponding MIPS32 code is shown.

Instruction	1	2	3	4	5	6	7	8	9
LW R1, B	IF	ID	EX	MEM	WB				
LW R2, C		IF	ID	STALL	EX	MEM	WB		
ADD R5, R1, R2			IF	STALL	ID	EX	MEM	WB	
SW R5, A				STALL	IF	ID	EX	MEM	WB

Instruction Scheduling or Pipeline Scheduling:

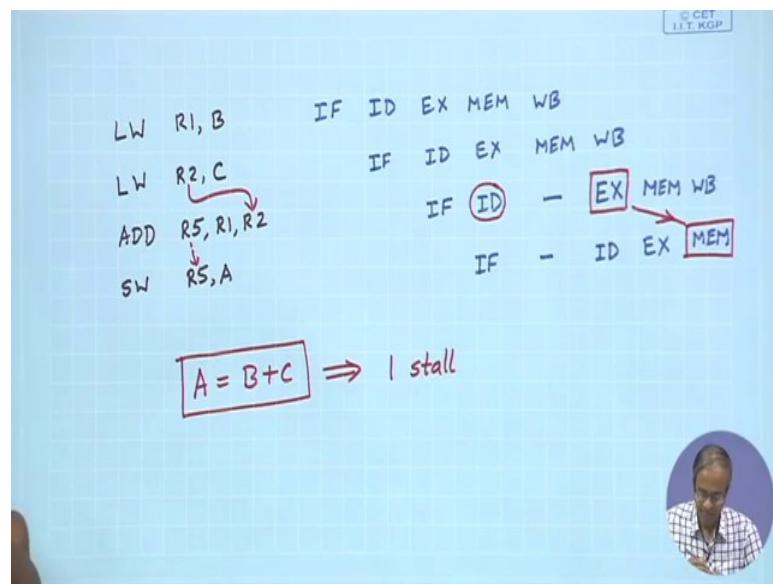
- Compiler tries to avoid generating code with a LOAD followed by an immediate use.

Logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya are displayed at the bottom.

Let us look at a very common example that appears in typical programs. An instruction like $A = B + C$ is just a representation of this kind of operation. The MIPS32 instructions for this is written here; for simplicity we are simply writing A B C, but actually it will be something like some number within bracket some register number where the address of these loaded. A straight forward implementation of this operation will be to load the value of B into some register R1, load the value of C into some register R2, add R1 and R2, and put the result in R5 and finally, store R5 into A.

But here the second load and then ADD causes a data dependency, and there will be a hazard and there will be a stall. This stall is unavoidable.

(Refer Slide Time: 16:09)



You see here the first load can proceed without any problem, it is doing a fetch, it is doing a decode, execute, mem and write back. Come to the second instruction, well again there is no dependency here. The second instruction also need no stalls.

But here there is a dependency between this loaded value of R2 and the use. When this ADD is fetched here and is decoded here you are coming to know that this hazard is there. So, what we do? You insert a stall here and you issue the instruction; that means, you move it to the EX stage, after this stall then MEM then WB. There will be instruction fetch here. For store instruction during the EX stage the address is calculated and store occurs here in MEM. Now see here also there is a data dependency, this R5 to R5, but the ADD instruction already has computed R5 at the end of this EX stage and the store instruction is requiring it here. So, there is no problem -- there is no hazard here, no stalls are required here.

So, only a one stall will be required in this case.

(Refer Slide Time: 19:12)

The slide is titled "Example 1". It shows a C code segment:

```
A C code segment:  
x = a - b;  
y = c + d;
```

Below it, the original MIPS32 code is shown:

```
MIPS32 code:  
LW R1, a  
LW R2, b  
SUB R8, R1, R2  
SW R8, x  
LW R1, c  
LW R2, d  
ADD R9, R1, R2  
SW R9, y
```

Annotations indicate "Two load interlocks" between the first two loads and between the last two loads.

To the right, the "Scheduled MIPS32 code" is shown:

```
Scheduled MIPS32 code:  
LW R1, a  
LW R2, b  
LW R3, c  
SUB R8, R1, R2  
LW R4, d  
SW R8, x  
ADD R9, R3, R4  
SW R9, y
```

Annotations indicate "Both load interlocks are eliminated".

The slide footer includes logos for IIT Kharagpur, NPTEL, and National Technology Transfer Center, along with a photo of a man.

Let us check another example because this will illustrate a few other things, here there is not one, but two operations in sequence. Just for illustration I am assuming that the values are a b c d and the result is stored in x and y. When you translate this into MIPS code, a straightforward translation will be something like this. First you come to $x = a - b$; load a into R1, load b into R2, subtract R1 and R2, store in R8, store R8 in x. First one is done then come to the second one, well here we are reusing the registers R1, R2 because R1 or R2 are not required any more. So, we load again c in R1, load d in R2, add R1 and R2, and store in R9, and storing R9 in y.

This is a straightforward implementation, but you see when do this kind of implementation where are the problem cases? Here we find there is a load followed by a use R2 is used here, here there is another load R2 followed by use. So, there will be a one stall cycle required here and one stall cycle required here. There will be 2 load interlocks. If you run this code just like this, the control unit will be inserting two stall cycles. The total time that we required to execute this code will be 2 clock cycles extra because of the data hazard that cannot be resolved.

Now, let us do something. Here we have assumed that we are using the same registers. Now let say suppose we have lot of registers with us we can spare more registers for this code, let us try this out. What we do is something like this -- we call this **instruction scheduling** and this modified code is called scheduled MIPS code. So, what is the

modification? Well the first modification is that, we are not using this same registers. The first instruction is using R1 and R2, but for this second one we are using R3 and R4. Not only that, the compiler will be moving some instructions around like normally it will like this load sub store, load add store, but here load sub and store for the second sequence the load was the one of the load is moved here.

Now, what is the purpose of moving it here? Purpose of moving it here is that you see that problematic data hazard has been eliminated. The problem case was a load followed by its immediate use. Here we have inserted another load instruction in between which is loading R3 that is not used here. So, R2 is here there is one cycle gap already, and R2 is used here. This is the only difference we made and the rest we have not changed, and there is another modification we did, that this store instruction for the first one, this also we pushed down and moved it after this load, we move this store instruction after this load to avoid this second interlock.

So, by doing this, this loading R4 and this using R4 there is one gap in between. The compiler can do this kind of code analysis and can move instructions around, these are very interesting problems for the compiler to solve and if we do this then both the load interlocks are eliminated. This modified code will be able to run without any stall cycles; you can save two clock cycles in the execution of this code right. So, this is the advantage of scheduling of the code.

(Refer Slide Time: 24:05)

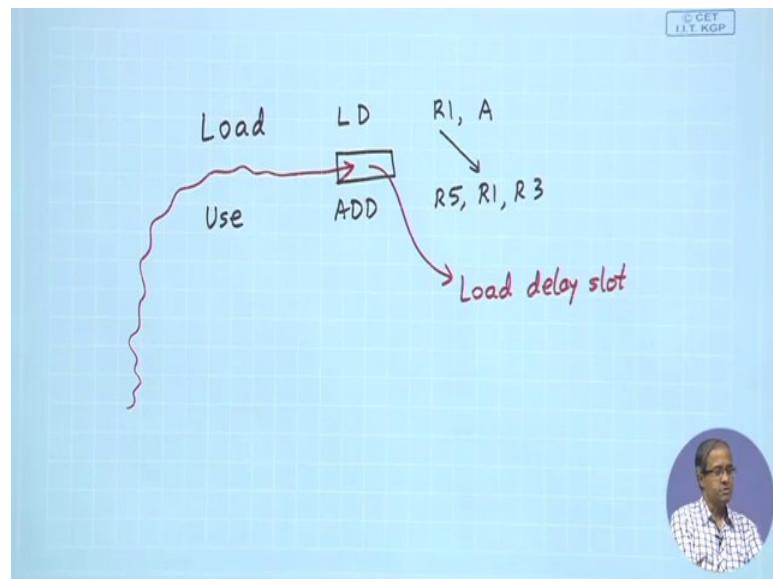
• Some observations:

- Pipeline scheduling can increase the number of registers required, but results in performance improvement in general.
- A LOAD instruction requiring that the following instruction do not use the loaded value is called a *delayed load*.
- A pipeline slot after a LOAD is called the *load delay slot*.
- If the compiler cannot move some instruction to fill up the delay slot (scheduling), it can insert a NOP (No Operation) instruction.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

What we saw that scheduling can improve performance, but in general it increases the number of registers required. In the previous case we had to use two additional registers R3 and R4.

(Refer Slide Time: 24:34)



What we are saying here is that whenever there is a LOAD followed by its immediate use, as if there is a gap or *slot* coming into here between the two instructions -- this gap is called the ***load delay slot***. If we do not do anything then the control unit has to insert one stall cycle here.

But now the compiler knows that there is a load followed by its immediate use. It will try to move some other instruction from somewhere else into this delay slot. If it can do that then it can avoid that wastage of one cycle. Coming back to this the load instruction that requires that the following instruction do not use the loaded value, we call it as ***delayed load*** and that slot is called delay slot or in this case ***load delay slot***. As I had said the compiler will try to move instructions around and try to fill up this delay slot. If it cannot find an instruction to move, then it can insert a special no operation (NOP) instruction, which does nothing.

You may wonder why at all there is an instruction like NOP, which does nothing. Here is one application of NOP you can think of. If the compiler cannot find any instruction to move, it can generate a new NOP instruction put it there in the slot. The size of the code will increase, but there will be no hazard -- the control unit will have to worry less. The

control unit becomes simpler; it does not have to check for any hazard if the compiler takes the responsibility.

(Refer Slide Time: 27:39)

The slide has a yellow header bar with the title "Types of Data Hazards". Below the title is a list of bullet points:

- Broadly three classes of data hazards are possible:
 - a) **Read After Write (RAW)** – Most common scenario.
 - b) **Write After Read (WAR)** – This kind of hazard cannot happen in MIPS32 as all reads are early (in ID) and all writes are late (in WB).
 - c) **Write After Write (WAW)** – This kind of hazard is possible in pipelines where write can happen in more than one pipeline stages.
- For MIPS32 integer pipeline, only RAW hazard is possible, and only one type (*LOAD followed by immediate use*) can result in performance loss.
 - Compiler tries to eliminate the performance loss using instruction scheduling.

At the bottom of the slide, there are logos for IIT Kharagpur, NPTEL, and NIT Meghalaya.

Generally speaking, data hazards can be of three types. The first one is the one that you have seen already, read after write. Some instruction is writing a data somewhere, register or memory, some other instruction is reading from there.

There can be some other kind of data hazard also, write after read. Reading takes place first writing takes place later. Well in MIPS32 this will never happen because in the normal instructions reading of the registers take place always in the ID stage and all writes take place in WB. So, it is never the case that you do the write before you do the read. Because of the characteristic of the MIPS instructions this will normally not happen.

Similarly, write after write. Two instructions both are writing, let us say the first instruction has written first, second instruction next. The first instruction is writing later the second instruction is writing earlier will never happen for the pipeline that we have seen, because all instructions take 5 cycles to complete.

Later on we shall see how we can extend the pipeline to floating point, and other multi-cycle operations. There we will see that this kind of write after write hazard can also occur. In the MIPS32 integer pipeline only the first kind of hazard is possible and the

only problem case that results in stall cycles is a load followed by immediate use. All others can be eliminated by forwarding, and we put lot of responsibility on the compiler. The compiler tries to move instructions around through instruction scheduling and try to fill up the delay slots.

(Refer Slide Time: 29:56)

(c) Control Hazard

- Control hazards arise because of branch instructions being executed in a pipeline.
 - Can cause greater performance loss than data hazards.
- What happens when a branch is executed?
 - The value of PC may or may not change to something other than $PC_{old} + 4$.
 - If the branch is *taken*, the PC is normally not updated until the end of MEM.
 - The next instruction can be fetched only after that (*3 stall cycles*).
 - Actually, by the time we know that an instruction is a branch, the next instruction has already been fetched.
 - We have to redo the fetch if it is a branch.

Now coming to control hazard, control hazards arise because there are branch instructions in a pipeline. Branch instruction means the next instruction to be executed can either be the sequentially next instruction or instruction from somewhere else if the branch is actually taken or it is a jump.

The next instruction to be executed will be from the target, and not the next sequential instruction. This is the problem in a pipeline because in the pipeline the instructions are coming one by one, they will all be entering in the pipeline. Suddenly we find out that early instruction was a branch it has to be taken. So, the following instructions all have to be ignored, they will have to be discarded and the new instruction is fetched from the target. When a branch is not executed the value of the PC is normally changed to the old $PC + 4$.

But when a branch is taken it can be something else, the PC can be replaced by a target branch address, but that target branch address is normally not known till the end of the MEM cycle, which means that you may require 3 stall cycles to take the decision that whether the branch is taken or not. By the time you take decision already 3 instructions

have entered the pipe. Thus, there is a lot of loss in performance; this is what meant by control hazard.

(Refer Slide Time: 31:45)

Instruction	1	2	3	4	5	6	7	8	9	10	11
BEQ_Label	IF	ID	EX	MEM	WB						
Instr. (i+1)		IF	Stall	Stall	IF	ID	EX	MEM	WB		
Instr. (i+2)			Stall	Stall	Stall	IF	ID	EX	MEM	WB	
Instr. (i+3)				Stall	Stall	Stall	IF	ID	EX	MEM	WB
Instr. (i+4)					Stall	Stall	Stall	IF	ID	EX	MEM
Instr. (i+5)						Stall	Stall	Stall	IF	ID	EX
Instr. (i+6)							Stall	Stall	Stall	IF	ID

Example:
We have seen that 3 cycles are wasted for every branch.
Assume – 30% branch frequency, $CPI_{ideal} = 1$.

Actual CPI = $0.7 \times 1 + 0.3 \times 4$
 $= 1.9$
Speed becomes almost half.



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



NATIONAL INSTITUTE OF
TECHNOLOGY, MEGHALAYA

This is what I mean. Suppose the branch instruction is executing, the next instruction is fetched, but because it is a branch instruction it has been decoded here. The branch outcome will be known only at the end of MEM. You cannot fetch the next instruction before that. So, there will be 3 wasted cycles, and you will have to again fetch here because this fetch can be from the new address. As you can see you will require 3 stalls cycles, this is a very bad scenario; obviously, let us say if our branch frequency is 30% in a program and the ideal CPI is 1, then the actual CPI will be in the seventy percent of the case CPI is 1, but for the remaining 30% it will be 1 + 3 stall cycles; that means, 4. So, it is 1.9; that means, the speed almost becomes half.

(Refer Slide Time: 33:07)

How to Reduce Branch Stall Penalty?

- The penalty can be reduced if both the following are achieved:
 - a) Determine whether the branch is taken earlier in the pipeline.
 - b) Compute the branch target address earlier in the pipeline.
- How to achieve (a) in MIPS32?
 - In MIPS32, the branches require testing a register for zero, or comparing the values of two registers.
 - Possible to complete this decision by the end of the ID cycle where the registers are pre-fetched, by adding special comparison logic.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

There are many ways to reduce the branch penalty. To reduce the branch penalty we have to tackle two sub-problems: (a) determine that whether the branch is taken or not, and (b) what is the branch target address? Both of these we have to calculate early enough so that our stall cycles can be reduced. Now in MIPS32 because of the simplicity of the instructions, this is easier. Computing whether a branch is taken or not it easy because the branch are like “branch if zero”, “branch if non zero”, just we have to check whether the register is 0 or non-zero. The registers are already fetched in ID, and you can add a simple zero comparator that hardly takes any time; whenever you are fetching that register you check whether it is 0 or non-zero.

So, the decision is already known at the end of ID itself. In MIPS32 the branches either require testing for 0 or comparing two registers because you are fetching all the registers in ID. You can come to know about the decision of the branch at the end of the ID itself. You have to add some special comparison logic.

(Refer Slide Time: 34:33)

- How to achieve (b) in MIPS32?
 - Both the possible PC values can be pre-computed in the IF stage by using a separate adder.
- By the end of the ID cycle, we know whether the branch is taken and also know the branch target address.
 - Requires a single cycle stall during branches.

Navigation icons: back, forward, search, etc.

Logos and text at the bottom:

- IIT KHARAGPUR
- NPTEL ONLINE CERTIFICATION COURSES
- NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

And the second one is to compute the branch target address. This you can also calculate earlier by using a separate adder. At the end of the ID cycle you know whether the branch is taken or not taken, and also you know the branch target address. You actually do not need to wait for 3 cycles, you need to wait for a single cycle like this.

(Refer Slide Time: 35:05)

Instruction	1	2	3	4	5	6	7	8	9	10	11
BEQ Label	IF	ID	EX	MEM	WB						
Instr. (i+1)		IF	ID	EX	MEM	WB					
Instr. (i+2)			Stall	IF	ID	EX	MEM	WB			
Instr. (i+3)				Stall	IF	ID	EX	MEM	WB		
Instr. (i+4)					Stall	IF	ID	EX	MEM	WB	
Instr. (i+5)						Stall	IF	ID	EX	MEM	WB

Navigation icons: back, forward, search, etc.

Logos and text at the bottom:

- IIT KHARAGPUR
- NPTEL ONLINE CERTIFICATION COURSES
- NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, earlier we have been saying that you have to wait till MEM, but now we are saying that because of the simplicity of MIPS, at the end of ID we will come to know that whether the branch is taken and also what is the address. We can start fetching after a

maximum of 1 stall cycle. We shall see later how this branch penalty can be further reduced in general.

We have come to the end of this lecture. In the next lecture we shall continue with this discussion on control hazard because there are many ways to try and reduce that the penalty due to this branch instructions, the control hazards to the extent possible. We shall discuss these things in our next lecture.

Thank you.

Computer Architecture and Organization

Prof. Indranil Sengupta

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Lecture – 57

Pipeline Hazards (Part 3)

In the last lecture we were looking into the control hazard associated problems, we had seen that for MIPS32 pipeline implementation, we can reduce the branch penalty to 1 cycle. Because we mentioned that at the end of the ID stage we can know both the outcome of a branch, whether it is taken or not taken branch, and also the branch target address. In the worst case we have to incur a penalty of 1 cycle. You should say that well one cycle is fine for load followed by use, we have one cycle here also. Let us make a simple calculation.

(Refer Slide Time: 01:09)

The image shows handwritten calculations on a light blue grid background. At the top right is a small logo for IIT KGP. The text is as follows:

$$\begin{aligned} \text{CPI}_{\text{ideal}} &= 1 \\ \text{30% branch} \\ \text{Branch penalty} &= 1 \text{ cycle} \\ \therefore \text{Actual CPI} &= 0.70 \times 1 + 0.30 \times 2 \\ &= 1.30 \\ \Rightarrow & 30\% \text{ degradation} \end{aligned}$$

We saw in the example we took in the last lecture that the ideal CPI was 1, 30% of branch instructions, but here we are also saying that branch penalty is 1 cycle. So, what will be the actual CPI in that case? The actual CPI will be 70% of the case when there is no branch CPI will be 1, and for the 30% of the cases there will be 1 cycle penalty. The actual CPI will be 1.3. You see you have a 30% degradation that is quite substantial.

Let us see how we can reduce this further. This is the topic of our discussion in this lecture. Our specific target here will be to reduce the pipeline branch penalties.

(Refer Slide Time: 02:14)

The slide has a blue header bar. Below it, the title 'Reducing Pipeline Branch Penalties' is centered in a yellow section. The main content is a bulleted list of four techniques:

- Four techniques based on compile-time guesses are discussed.
 - a) The simplest scheme is to freeze the pipeline and insert (one) stall cycle.
 - Main advantage is simplicity.
 - b) We predict that the branch is *not taken*, and allow the hardware to continue as if the branch was not executed.
 - Must not change the machine state until the branch outcome is finally known.
 - If the prediction is wrong (i.e. branch is taken), we stop the pipeline and restart the fetch from the target address.
 - Illustration on next slide.

At the bottom, there are logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Rourkela. To the right of the NPTEL logo is a circular portrait of a man.

We start by discussing four broad techniques.

First is a very naive approach -- we have seen that a branch will incur 1 cycle penalty. Whenever there is a branch we freeze the pipeline, insert 1 stall cycle. This is the simplest approach, main advantage is simplicity, but as I shown in the example just now that the overhead can be significantly high. There can be 30% performance overhead.

The other approaches are based on some kind of prediction. The second approach says we predict that the branch is not taken, and you allow the hardware to continue as if the branch is not executed at all. That means, the next instruction is fetched, it is executed and we continue as if nothing has happened, until the branch outcome is actually known to us. When the branch outcome is actually known to us then we can know whether our prediction was right or wrong. If we see that our prediction was right, we do not do anything. The next instruction has already entered into the pipe and was already executing, let it continue and we do not incur any stall cycle for that. But only if we find that our decision was wrong, it was actually a taken branch and we are assuming not taken, then we will have to stop that, insert a stall and fetch the new instruction from the target.

(Refer Slide Time: 04:15)

c) We predict that the branch is *taken*.

- As soon as the branch outcome is decoded and the target address computed, fetching of instructions can commence from the computed target.
- Since in MIPS32 pipeline, we come to know about the branch outcome and target address together (at the end of ID), there is *no advantage* in this approach.
- May be used for complex instruction machines where the branch outcome is known later than the branch target address.

	Instruction	1	2	3	4	5	6	7	8	9	10
Taken or Not	BEQ Label	IF	ID	EX	MEM	WB					
Taken Branch (1 cycle penalty)	Instr. (i+1)		IF	ID	EX	MEM	WB				
	Instr. (i+2)			Stall	ID	EX	MEM	WB			
	Instr. (i+3)				Stall	IF	ID	EX	MEM	WB	
	Instr. (i+4)					Stall	IF	ID	EX	MEM	WB

Let us take an example. If it is a not taken branch then there is no penalty; that means, it is a branch instruction, but the branch is not taken. The sequentially following instruction will be executing after that. The hardware will be assuming that the branch is not taken, and it will fetch the consecutive instructions without any stall. For such cases there will be no penalty, but if the branch is actually taken then at the end of the ID stage you will come to know that your prediction was wrong, because it is here you come to know that your branch is taken or not taken by decoding the registers. If it is wrong then there will be 1 stall cycle, insert it and for this case you will be incurring a one cycle penalty. For some of the cases there will be no penalty for some cases there will be one cycle penalty.

The third approach is we predict that the branch is taken, just the reverse. We assume that the branch is always taken, but unfortunately for MIPS32 you will see that this prediction does not help because if we predict that the branch is taken; that means, you will always be fetching the next instruction from the target address and the target address is known only at the end of ID. So, only after ID you can start the fetch. Anyway this 1 cycle will get lost. Irrespective of whether it is taken or not taken branch, for MIPS32 this one cycle penalty will always be there. Here the idea is that whenever the branch outcome is decoded and the address is known; that means, at the end of ID, fetching of instructions can continue from the target; that means, here.

As I had said for MIPS32 we know the branch outcome and the target address both together. So, there is no advantage in this approach because in both cases there is 1 cycle penalty; for more complex instruction machines may be this will help.

(Refer Slide Time: 06:52)

d) We can use a technique called *delayed branch*, which makes the hardware simple but puts more responsibility on the compiler.

- If a branch instruction incurs a penalty of n stall cycles, the execution cycle of a branch instruction is defined as follows:

Branch instruction
 Successor₁
 Successor₂
 ...
 Successor _{n}
 Branch target if taken

Branch delay slots For MIPS32, $n=1$

- Just like in load-delay slots, the task of the compiler is to try and fill up the branch-delay slots with meaningful instructions.
- Instructions in the branch-delay slot are *always executed* irrespective of the outcome of the branch.

In such machines, possibly the branch address is known earlier, but whether it is a taken or non taken branch is known much later in the instruction execution cycle. For those kind of complex instructions this strategy will work better, but not for MIPS.

And the last approach is called something called *delayed branch*. It says that there is a branch instruction, let that branch execute it can be a taken branch or not taken branch. Now, the assumption is that the slots that are following that branch instruction where you are normally inserting these stall cycles, for MIPS there was 1 stall cycle I said for branch. We call it as a *branch delay slot* -- what we are saying is that the instruction that is there in the delay slot; that means, one instruction we fetched after the branch will always be executed and the compiler knows that irrespective of the branch is taken or not taken. So, the compiler will try to put some instruction in the delay slot, which is supposed to be executed every time the branch is executed, irrespective of it is a taken or non taken; this is the basic idea.

This is called delayed branch. Here we are making the hardware simple, but we are putting all the responsibility on the compiler. In general a branch instruction can have n stall penalties, but for MIPS $n = 1$, after this n penalty the next target address will be

known. The next instruction can be fetched only after that. These n successor instructions are called branch delay slots, for MIPS it is 1. The compiler will try to move instructions around and try to fill them up, and as I said the interpretation is that wherever instructions are put in the branch delay slots in this strategy, these instructions are always executed irrespective of the outcome of the branch.

(Refer Slide Time: 09:13)

Some Examples

<pre> ADD R3,R4,R5 BEQZ R2,L DELAY SLOT ... L: </pre>	<pre> L: ADD R1,R2,R8 SUB R3,R4,R5 BEQZ R3,L DELAY SLOT </pre>	<pre> ADD R1,R2,R8 BEQZ R3,L DELAY SLOT SUB R3,R4,R5 </pre>
\downarrow	\downarrow	\downarrow
<pre> BEQZ R2,L ADD R3,R4,R5 ... L: </pre>	<pre> L: SUB R3,R4,R5 BEQZ R3,L ADD R1,R2,R8 </pre>	<pre> ADD R1,R2,R8 BEQZ R3,L SUB R3,R4,R5 </pre>

The slide also features logos for IIT Kharagpur, NPTEL, and National Institute of Technology, Anna University, along with a portrait of a speaker.

Let us take some examples. Suppose I have an ADD instruction followed by a BEQZ, and there is a delay slot after that. You see the ADD instruction is executed before branch always. So, there is no harm if you move this ADD to the delay slot because whenever there is a branch the delay slot will also be executed. We are not wasting the delay slot, rather we are moving a useful instruction. Whenever the branch is taken or not taken, ADD will always be executed.

The other example is a little complex. Again, we can move this ADD instruction to the delay slot. This is just a responsibility of the compiler; compiler will analyze the code and find out whether it can move some instructions like this.

Similarly for the third example, this SUB can move to the delay slot.

(Refer Slide Time: 12:02)

- Additional overhead for delayed branches:
 - Multiple PC's (one plus the length of the delay slot, i.e. $n + 1$) are needed to correctly restore the state when an interrupt occurs.
 - Consider a taken branch instruction followed by instruction(s) in the delay slot(s).
 - The PC of branch target and PC's of delay slots are not sequential.

For delayed branches there are some other difficulties, like you will be having some multiple PCs, in fact, it will be $n + 1$. This means other than the target address, all the instructions in the delay slot also need to be restored if there is an interrupt in between. So, there are multiple values of the PC that needs to be saved because the PC of the branch target and the PC of the delay slots are not sequential.

(Refer Slide Time: 12:47)

Example 1

- Consider the MIPS32 pipeline with ideal CPI of 1. Assume that 20% of the instructions executed are branch instructions, out of which 75% are taken branches. Evaluate the pipeline speedup for the four techniques discussed to reduce branch penalties.
- Solution:
$$\begin{aligned} \text{Speedup} &= \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \frac{\text{Pipeline stall cycles / instruction}}{\text{Pipeline Depth}}} \\ &= \frac{1}{1 + (\text{Branch frequency} \times \text{Branch penalty})} \end{aligned}$$

Let us take an example. Consider pipeline with a ideally CPI of 1, let us say 20% of the instructions are branch and out of them 70% are taken, and the remaining 25% are not taken.

Using the four strategies we discussed let us evaluate the speedup. The speedup will be calculating using this formula, this ideal CPI multiplied by pipeline depth divide by ideal CPI plus stall cycles per instruction.

(Refer Slide Time: 13:40)

(a) Stall pipeline Branch penalty = 1 Speedup = $5 / (1 + 0.20 \times 1) = 4.17$	(b) Predict not taken Branch penalty = 1 Speedup = $5 / (1 + (0.20 \times 0.75)) = 4.35$
(c) Predict taken Branch penalty = 1 Speedup = $5 / (1 + 0.20 \times 1) = 4.17$	(d) Delayed branch Branch penalty = 0.5 Speedup = $5 / (1 + (0.20 \times 0.5)) = 4.55$

IIT Kharagpur | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MANGALORE**

This were derived earlier already because ideal CPI is 1, you put 1 and pipeline stall cycle per instruction in this case will be branch frequency multiplied by branch penalty.

Let us look at the four possibilities. For the stall pipeline strategy always the branch penalty will be 1. For all the branches you are stalling. For 0.2 branch penalty will be 1. So, 0.2 multiply by one if you calculate speedup becomes 4.17.

The second strategy is predict not taken. Here you will be incurring the overhead only for the 75% of the cases where the branch is actually taken. The speedup is higher, 4.35.

But predict taken here incurs the penalty all the time because there is no advantage, which is same as stall pipeline.

And for the last strategy with delayed branch, here for the branch you are trying to move an instruction in the delay slot. I am assuming that there is 50% probability that the compiler will be able to fill up the delay slot. So, I am multiplying this by 0.5. If you calculate you will see that you get a higher figure, 4.55.

Interrupts pose a more difficult problem in a pipeline. Let us see in the MIPS 5-stage integer pipeline whenever interrupt comes what are the issues.

(Refer Slide Time: 15:13)

Dealing with Interrupts in MIPS32

- Interrupts complicate the design of an instruction pipeline.
 - Overlapping of instruction executions make it more difficult to decide whether an instruction can safely change the state of the machine.
 - Some interrupts can force the machine to abort the instruction before it is completed (e.g. page fault).
 - The most difficult interrupts to handle in a pipeline have the properties:
 - a) They occur within instruction
 - b) They have to restarted

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Interrupts can complicate the design of the pipeline. Overlapping of the instruction execution makes it difficult to decide whether an instruction can safely modify something. When I say safely change the state, it means either changes the value of some register or something. Suppose an instruction changes the value of a register, and later it is found that there is an interrupt, and that instruction has to be withdrawn, but already it has modified a register.

So, the instruction should not change the state of the machine before it is known that the interrupt has occurred or not. Some interrupts can force the machine to stop the instruction before it is completed like page fault interrupt. Whenever you are trying to fetch an instruction from the memory, if it is not there in memory, it has to be fetched from disk, which is the example of a page fault. Under those cases the instruction has to be restarted after the requested word is brought into memory, will again execute that instruction. So, such interrupts are more difficult. The most difficult interrupts have the properties that they occur in between an instruction, and they have to be restarted.

(Refer Slide Time: 17:09)

The slide contains the following text:

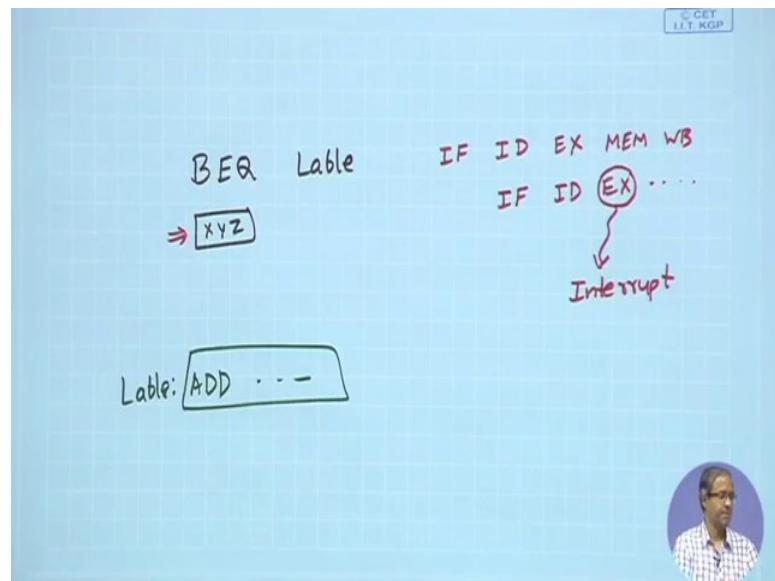
- When an interrupt occurs, the pipeline state can be saved safely as follows:
 - A TRAP instruction is forced into the pipeline at the next IF cycle.
 - Until the TRAP is taken, disable all writes for the *faulting instruction* and all others that follow. → **prevents any state change**
 - After the interrupt handler takes control, it saves the PC of the faulting instruction.
- For delayed branches, suppose that an instruction in the branch-delay slot causes the interrupt.
 - If the branch is taken, then the instructions to restart are those in the slot plus the instruction at the branch target.
 - A number of PC values need to be saved and restored.

At the bottom of the slide, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Rourkela.

One strategy for interrupt handling can be like this. Whenever an interrupt occurs -- this interrupt can occur in any stage -- it can occur in IF, ID, EX, etc. A special trap instruction or a some kinds of a flag is forced into the pipeline at the next instruction fetch cycle, which will indicate that an interrupt has occurred and the control unit will not allow any writes to occur; it will disable all writes not only for the instruction that generated the exception, but for all the instructions which follow it and this will continue until the trap reaches the WB stage. Until the trap comes out of the pipe we will be disabling all writes, and when the trap comes out then only the interrupt handler or the interrupts will be activated, it will save the PC of the instruction and it will process the interrupt.

Therefore delayed branches it is a difficult condition when the instruction in the branch delay slot might have caused the interrupt.

(Refer Slide Time: 18:56)



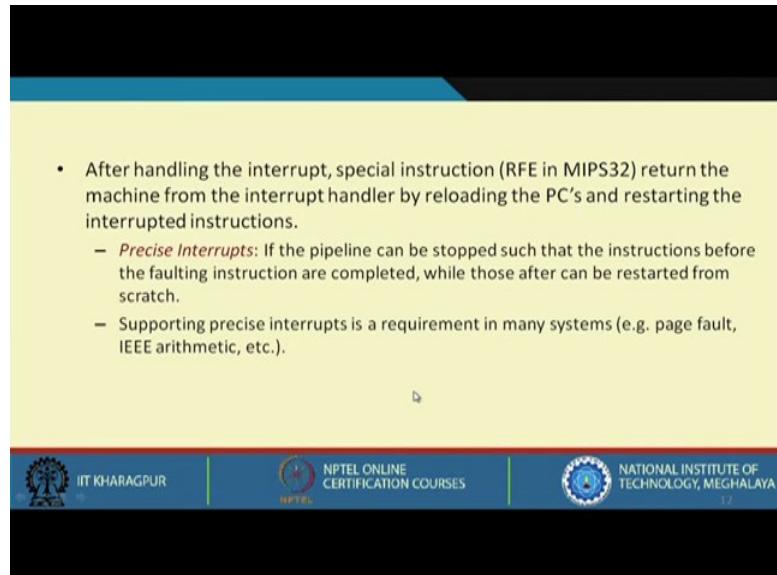
What I mean to say is that there is a branch, let us say BEQ Lable, and there was one delay slot. There are some instructions here, let us say xyz, and this instruction has generated an interrupt. In the pipeline this branch instruction was already there, it was fetched, it was decoded, it was executed, it was suppose to do MEM and WB and the next instruction is supposed to also execute along with it. So, xyz should also be supposed to be fetched here, decoded here, executed here, and so on. What I am saying is that suppose this instruction during the EX stage generates an interrupt. We will stop everything not only this, but also this branch instruction. And when you come back you will have to not only restart this xyz, but also the instruction that is here in this target Lable.

So, multiple PC values have to be saved, the PC of this xyz, and also the PC of this. These PC values will also have to be restored when the interrupt is handled, after that there is a special instruction like return from exception (RFE). This is the instruction that needs to be given at the end or the last instruction of the interrupt handler. For this kind of delayed branch kind of machines, the RFE instruction has to do a lot -- it will have to reload multiple PCs.

There is a terminology called ***precise interrupts***; precise interrupts says that let us say an interrupt is occurred. If it is possible for the control unit to stop the pipeline such that all instructions before the instruction that generated the interrupt can complete, and all

instructions that follow will wait, they will be restarted later, then we say it is a precise interrupt.

(Refer Slide Time: 22:00)



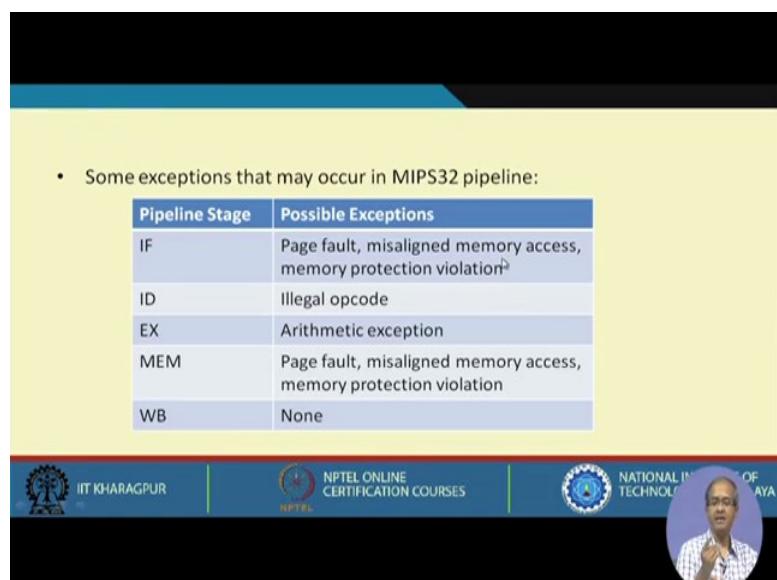
The slide contains a bulleted list under a heading 'Precise Interrupts' (partially visible). The list includes:

- After handling the interrupt, special instruction (RFE in MIPS32) return the machine from the interrupt handler by reloading the PC's and restarting the interrupted instructions.
 - Precise Interrupts*: If the pipeline can be stopped such that the instructions before the faulting instruction are completed, while those after can be restarted from scratch.
 - Supporting precise interrupts is a requirement in many systems (e.g. page fault, IEEE arithmetic, etc.).

At the bottom, there are logos for IIT Kharagpur, NPTEL, and National Institute of Technology Meghalaya, along with a navigation arrow.

If you see the definition, if the pipeline can be stopped such that the instruction before the faulting instruction are completed, while those after can be restarted from scratch then we say it is a precise interrupt. Well there are cases like page faults and also for IEEE arithmetic, this precise interrupt is a necessity.

(Refer Slide Time: 22:33).



The slide lists some exceptions that may occur in MIPS32 pipeline:

Pipeline Stage	Possible Exceptions
IF	Page fault, misaligned memory access, memory protection violation
ID	Illegal opcode
EX	Arithmetic exception
MEM	Page fault, misaligned memory access, memory protection violation
WB	None

At the bottom, there are logos for IIT Kharagpur, NPTEL, and National Institute of Technology Meghalaya, along with a circular video player showing a person speaking.

The designers of these pipelines spent efforts to ensure that interrupt handling is precise. So, what are the kinds of interrupts that can be generated in the five stages? In the IF stage, there can be page fault while fetching the instruction, there can be misaligned memory access because we said that all instruction fetches have to be from a multiple of four, memory address should be a multiple of four, but by mistake if you have given a misaligned address that will also generate an exception, and memory protection violation where you are trying to access from a memory location while you do not have the access permissions.

(Refer Slide Time: 23:39)

- Multiple interrupts may also occur in the same clock cycle.
- An example:

Instruction	1	2	3	4	5	6
LW R1,100(R2)	IF	ID	EX	MEM	WB	
ADD R4,R5,R6		IF	ID	EX	MEM	WB

- Can cause a data page fault (in MEM) and arithmetic exception (in EX) at the same time (clock cycle 4).
- As a possible solution, we can deal only with the data page fault and then restart the execution. The second interrupt will occur again, and will be handled later.

For ID you can get illegal opcode. For EX you can get a divide by 0 arithmetic exception. MEM can generate page fault, misaligned access, memory protection violation. And WB there is nothing.

Multiple interrupts may occur in the same cycle this is one danger. Like let us say a load instruction followed by an add here. The load instruction can generate a page fault during MEM, while this add can generate an arithmetic exception during EX. So, two interrupts are being generated in the same clock cycle.

One solution is that if this kind of a thing happens, you ignore the second interrupt and only handle the first one and again restart. So, if the second instruction generated an interrupt it will generate it again. We deal only with the page fault and restart the execution, the second interrupt will occur again and will be handled at that time.

(Refer Slide Time: 24:26)

- Difficulty: Interrupts may appear out of order.
 - LW causes a data page fault (cycle 4), and ADD causes an instruction page fault (cycle 2).
- A possible solution to this problem is discussed next.

Instruction	1	2	3	4	5	6
LW R1,100(R2)	IF	ID	EX	MEM	WB	
ADD R4,R5,R6		IF	ID	EX	MEM	WB

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

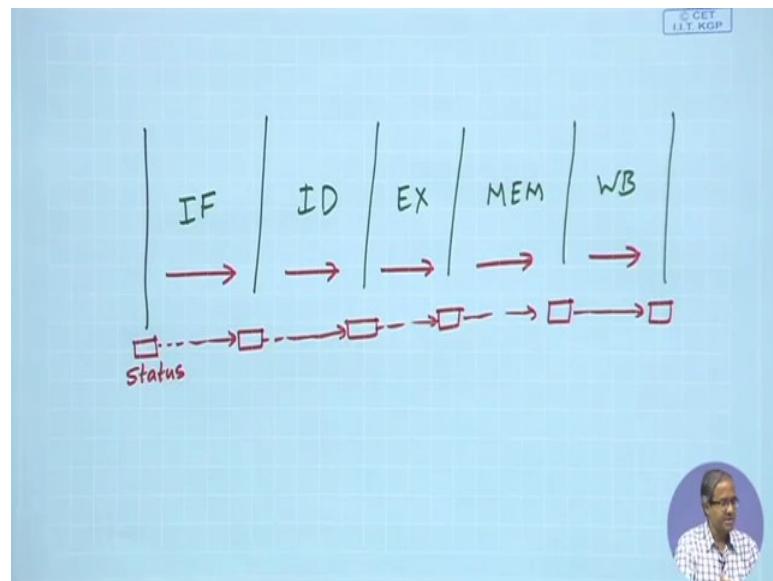
This is one solution you can think of. The second difficulty is that interrupts may appear out of order. Let us take another example. If first load instruction is having a page fault here in MEM, and this add instruction can have page fault in IF. So, it is out of order. For out of order interrupts, the later instruction is generating interrupt earlier. A possible solution to this will be just discussing briefly. The solution is like this -- the hardware will post each interrupt in a status vector like you see you have the pipeline stages.

(Refer Slide Time: 25:08)

- Possible Solution:
 - The hardware posts each interrupt in a status vector, which is carried along with each instruction as it moves through the pipeline.
 - When an instruction reaches WB, the interrupt status vector is checked and handled if present.
 - Guarantees that all interrupt handling is carried out in *precise* order.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

(Refer Slide Time: 25:23)



You have the IF, you have the ID, you have EX, you have MEM, you have WB. What we are saying is that when the instructions move from one stage to the next, there is also a special status register that is part of the inter-stage latches, this will also move from one stage to the next.

This is the status this will move like this. What we are saying is that this status vector is carried along with instruction as it moves in the pipe, and in that status vector you set a bit indicating that there is an interrupt and also the type of the interrupt. You do not do anything here, you let it move and when it reaches WB only then you process the interrupt. You allow it to move till WB when it reaches here, you see what is the type of the interrupt and then you handle that interrupt.

When the instruction reaches WB, the interrupt status vector is checked and handled, but then preciseness of the interrupt is guaranteed because the first instruction will be reaching WB earlier, the following instruction will be reaching WB later. So, maybe the interrupts are generated out of order, but the first instruction will always reach WB earlier than the next instruction. Thus, the interrupt for the first instruction will reach WB earlier. This is what we mentioned that interrupt handling will be carried out in precise order and with this we come to the end of this lecture and in the next lecture we shall be discussing some more methods to improve control hazard handling.

Thank you.

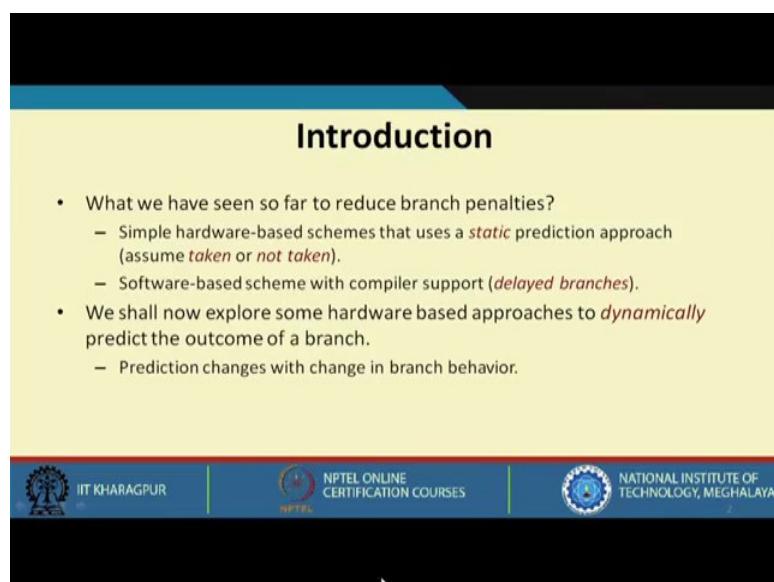
Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 58
Pipeline Hazards (Part 4)

We continue with the discussion on control hazards. If you recall what we are discussing in our last lecture, we looked at various schemes with which we were trying to reduce the branch penalty. But what were our approaches? We were making some predictions -- static predictions, we were assuming beforehand branches taken or not taken, or we were relying on the compiler to move codes around and give the pipeline a better code to execute, which will generate less number of stalls. But both these approaches are static in some respect, which means whatever we are deciding or predicting or assuming is statically done once.

Now, we look at some approaches where we try to exploit the dynamic behavior of a branch instruction to improve or enhance the performance with respect to the control hazards.

(Refer Slide Time: 01:43)



Introduction

- What we have seen so far to reduce branch penalties?
 - Simple hardware-based schemes that uses a *static* prediction approach (*assume taken or not taken*).
 - Software-based scheme with compiler support (*delayed branches*).
- We shall now explore some hardware based approaches to *dynamically* predict the outcome of a branch.
 - Prediction changes with change in branch behavior.

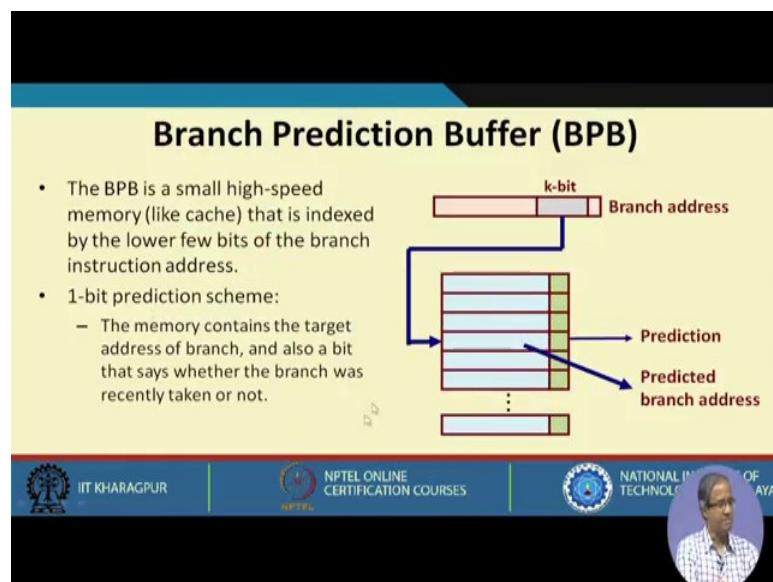
IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

What I have just now mentioned this is stated here. In our earlier lectures we have seen that in order to reduce the branch penalties, we were using some kind of static approach; either we are doing some prediction, as I had said we are assuming that the branches

taken or the branches not taken, or we are relying on the compiler to try and fill up the branch delay slots with some useful instructions. But now we shall discuss purely hardware based approaches to dynamically predict the outcome of a branch.

It is dynamic, so the prediction can change with time. Like you see in a program whenever there is a branch depending on the program some of the branches can be taken most of the time, some other branches can be not taken most of the time. So, you really cannot say statically beforehand that all branches are mostly taken or mostly not taken.

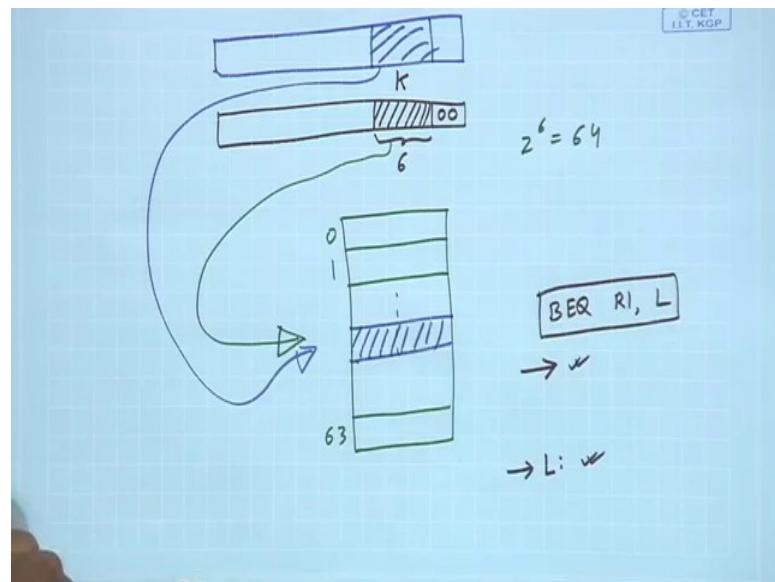
(Refer Slide Time: 03:17)



The first approach we talk about uses a hardware data structure called a branch prediction buffer.

Branch prediction buffer is depicted here. So, what this BPB is? It is a small high-speed memory, and this is the address of the branch that is shown in pink. Now you recall in MIPS32, every instruction starts with an address that is a multiple of 4, which means that branch address will also be a multiple of 4.

(Refer Slide Time: 04:06)



Thus whenever we have a 32-bit address the last 2 bits will always be 00, this means multiple of 4, then you have remaining 30 bits. Here you are selecting a few bits from the lower side, say K number of bits. K is a design parameter, this can be 4 5 6 whatever, let us say K = 6.

Our approach will be something like this. These K bits of the branch address are used to access this table BPB. BPB is like a small high-speed memory that is indexed by the lower few bits of the instruction. Let us come back to this again; if I take K = 6, 2^6 means 64. So, I will be having a small memory with 64 entries, 0, 1 up to 63. These few bits will be used as the address to access this memory. What is stored in this memory? Here the predicted branch address is stored. The predicted branch address can be either the address of the next instruction or the address of the target, like suppose we have an instruction BEQ R1,L.

When this instruction is executed, if it is not taken branch, next instruction executed will be the next one, if it is a taken branch next instruction executed will be the one from the label L. So, this is one possible branch address, this is the other possible branch address. The predicted branch address is stored here. The idea is that you check the last few K bits, you look up this table and see what is stored here. And whatever is stored here that is your predicted branch address from where you start fetching the next instruction.

You have some additional bits stored in the one bit prediction scheme. We have single bits with every entry, this will tell whether the last prediction was a taken branch or a not taken branch. Let us say 0 means taken and 1 means not taken.

(Refer Slide Time: 07:24)

- 1-bit prediction scheme (contd.):
 - The prediction may be incorrect → it may correspond to some other branch instruction that has the same lower-order k address bits.
 - In this scheme, instruction fetching begins from the predicted address.
 - If the prediction is later found to be wrong, the prediction bit is inverted.
- Drawback of the 1-bit prediction scheme:
 - Consider a branch that is taken most of the time. When it is not taken, there will be two incorrect predictions, rather than one.
 - Same is true for the reverse case (i.e. a branch not taken most of the time).

In this one bit prediction scheme you see just one thing. Whenever you encounter a branch instruction you check these K bits, go to the index and also check the prediction because you will also come to know later whether the branch is actually taken or not taken. You can compare with the prediction whether this is matching whatever was there, whether it is the same. If it is matching then you know that you are correct. But there are some problems -- this one bit prediction scheme means the prediction may be incorrect because the value you're getting from the BPB may correspond to some other branch instruction also that has same lower order K address bits.

Like here you are using a particular branch address, you are looking at the last K bits. There can be some other branch address for which accidentally these K bits are identical. So, they will both point to the same location. You will be trying to get the same entry here in the BPB and say that this will be your target branch address; obviously, maybe this entry was for this one. Now by mistake for the other one also you map here. So, that will be a wrong prediction. Thus, sometimes a prediction may be incorrect and here the instruction fetching will begin from the predicted address, and during execution later on

-- well for MIPS32 I said at the end of ID you will come to know whether your prediction was right or wrong.

At the end of ID anyway you will know whatever you have taken from BPB and you are going ahead was right prediction or a wrong prediction. If you later on see that your prediction is wrong, you can appropriately change the prediction bit in the BPB because as I just said there is a prediction bit in the BPB because it tells whether the last prediction was taken or not taken, and instruction fetching will begin from the predicted address. Now the drawback here is that suppose I have a branch instruction that is taken most of the time. So, when it is not taken there will be 2 incorrect predictions rather than 1. Let us look at a scenario; suppose I have a loop executing 100 times. So, for 99 of the times it will be a taken branch, but for the last time it will be coming out of the loop that will be a wrong prediction.

Out of this 100 the last time we will be getting one wrong prediction. In BPB that entry was showing as taken next time when you again go back and enter that loop, maybe this is a nested loop, next time when you enter that loop again the first iteration of that loop there will again be a miss prediction because last time you had said the prediction bit to not taken because it has come out of the loop. So, there will be 2 miss predictions for every execution of the loop.

(Refer Slide Time: 11:41)

Example 1

- Consider the following nested loop where the inner loop iterates for 20 times before exiting.

```
for (loop=0; loop<1000; loop++)
{
    ...
    for (i=0; i<20; i++)
        A[i] = A[i] + 5;
}
```

- For every execution of the inner loop, there will be two mispredictions:
 - Last iteration of the current loop.
 - First iteration of the next loop.
- Though the branch is taken 95% of the time (19 out of 20), correct prediction occurs only 18 times.
 - 90% accuracy.



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



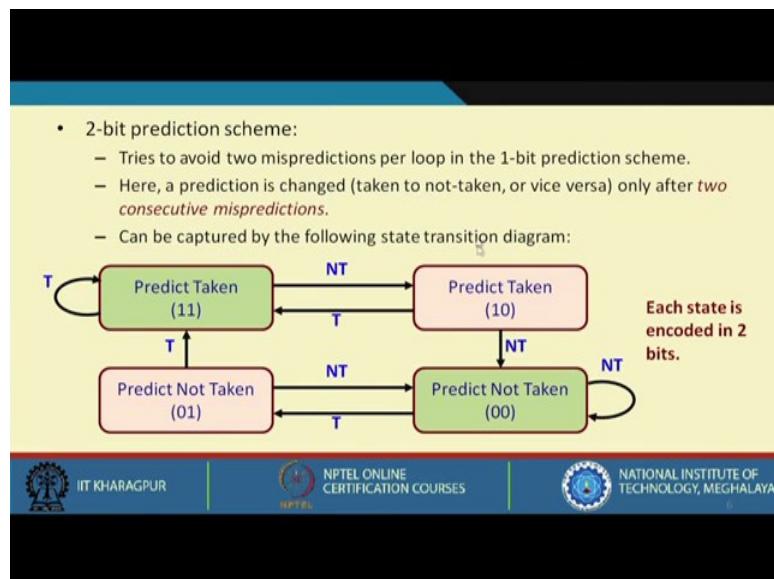
NATIONAL INSTITUTE OF
TECHNOLOGY TRIVANDRUM



Let us take an example. Here we have an inner loop that executes 20 times and this is inside and outer loop. I am looking at the inner loop. The inner loop is executing 20 times. We are saying that for this inner loop there will be 2 mispredictions, last iteration of the current loop. When $i = 19$ last time it will be coming out of the loop, but for i equal to 0 to 18, it will be taken, but for $i = 19$ it will be not taken, but when it comes back and enters the loop again for the next iteration of the outer loop, again it will start with a $i = 0$ and last time it was a not taken prediction.

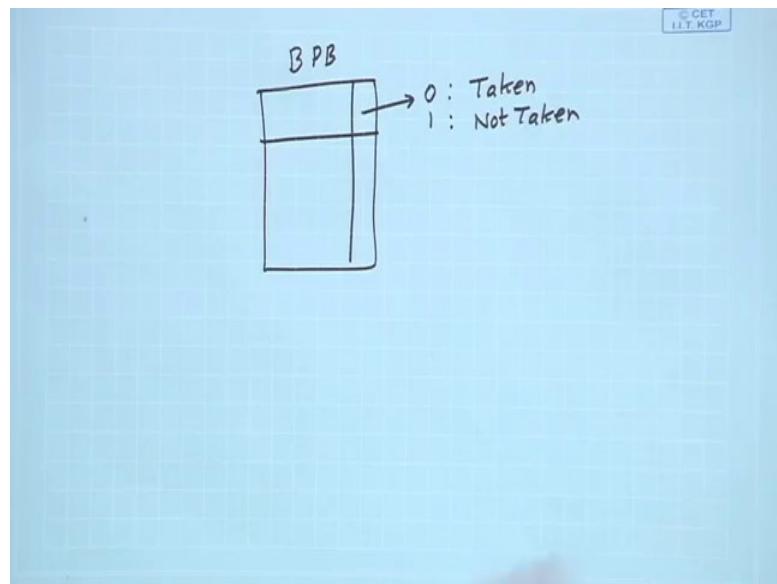
So, with not taken for $i = 0$ it will try to take the branch. It will be a misprediction again. The first iteration of the next loop we will also have the misprediction. Actually though this branch is taken 95% of the time in reality, 19 times it is taken 1 time it is not taken, but this one bit prediction scheme provides correct prediction 18 times and twice it is mispredicting. Thus, it is having 90% accuracy.

(Refer Slide Time: 13:19)



This is one drawback of this one bit prediction scheme. To avoid it you can have a 2 bit prediction scheme because the trouble with the earlier scheme was that every time there was a misprediction, we were flipping that bit like in that branch prediction buffer BPB there was this prediction bit this was 0 or 1.

(Refer Slide Time: 13:33)



We were flipping this bit as soon as there is a misprediction, we are not taking this fact into account just like the earlier example, that most of the practical scenarios will be like this and if you follow this immediately flipping policy there will be 2 miss predictions in every loop. Our modified strategy will go like this; we will not change the prediction every time there is a misprediction, we will be waiting 2 times. If there are 2 consecutive mispredictions, then only we will be changing the prediction information. If we do it then if you think of the earlier case then 19 of the times we can have correct prediction. Here we try to avoid the 2 mispredictions for loop in the example that I have shown for the one bit prediction scheme.

For the 2 bits scheme I am showing a state transition diagram. These are the 4 states are shown, the green states are the stable states -- 00 indicates predict not taken 11 indicates predict taken. While you are here if your loop is taken you remain here, and while you are here if it is not taken you remain here. But if it is predict taken if you get a not taken misprediction, you do not straight away go here, but rather you come to an intermediate state, this is still predict taken, but maybe. Here if you see that next time again there is a misprediction then only you move here. Which means that your behavior of the loop has possibly changed earlier it was mostly taken, now somehow it has become mostly not taken.

But if you see that the next time it is again taken we again come back here. Similar is the case here. If it is not taken and if you find it is a taken, a misprediction, you temporarily come here -- this is again predict not taken, but not green may be case. If it is again a taken; that means, 2 consecutive mispredictions then you permanently move here and if it is again a not taken you again come back here. Since each state is encoded in 2 bits we call it a 2-bit prediction scheme. Here your prediction is changing only after 2 consecutive mispredictions. In general this will give you better behavior as compared to the one bit prediction scheme.

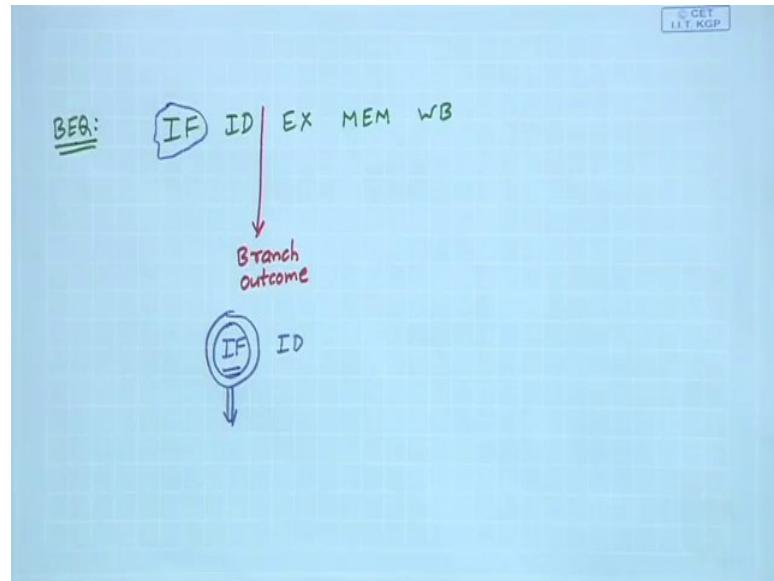
(Refer Slide Time: 17:09)

- An observation:
 - It may be noted that the 2-bit prediction scheme as discussed cannot speed up the MIPS32 pipeline.
 - In MIPS32 pipeline, both the branch outcome and branch target address gets known together (viz. at the end of ID).
 - To improve MIPS32 pipeline performance, we need to know from what address to fetch by the end of IF.
 - Whether the instruction is a branch (not decoded yet).
 - What is the predicted next value of PC?

Now, the thing is that for a MIPS32 pipeline, really you do not gain much with this because in MIPS32 pipeline as I had said because of the simplicity of the instruction set and instruction encoding, we will come to know everything at the end of ID anyway. We will know whether the branch will be taken or not taken, we will also know the address of the target at the end of ID itself.

So, whether we are using 1-bit or 2-bit prediction scheme really does not help much for MIPS32 because we have to wait till the end of ID, to come to know that whether our prediction was correct or not correct. So, for MIPS32 we need to do something more.

(Refer Slide Time: 18:18)



If you look at the pipeline stages again, it was IF, ID, EX, MEM and WB. Suppose this is a branch instruction that was executing, what I am saying is that it is only at the end of ID you come to know about the branch outcome precisely. So, all your decision will be here.

What we are trying to do here is that can we do something in the IF stage itself, so that fetching can start in the immediately next cycle with some predictive accuracy. Because this is the branch instruction we cannot know until ID is over anyway, but here we are saying that well we do not know this is the branch but you see there are 2 things. Firstly, I am saying that the instruction unless you decode it during the ID stage, you will not know that it is a branch instruction and if you do not know it is a branch instruction then all these things become meaningless -- it can mean ADD instruction also, but what you can do better is you apply a little intelligence. Suppose I maintain the memory address of the instructions I know – like the instruction at memory address 1000 is a branch.

Whenever I am fetching an instruction I am comparing whether the value of the PC from where I am fetching is the address of a known branch instruction that I had seen earlier. There will be another table, I compare to see if 1000 is there. If 1000 is there in the table, I will know that this is a branch instruction. I can start my manipulation during IF itself, I will not have to wait till ID. This is the philosophy we will be following now. This last statement actually talks about that to improve the MIPS32 performance we need to know

from what address to fetch by the end of IF. We want so many things earlier before the instruction is decoded.

(Refer Slide Time: 21:42)

Branch Target Buffer (BTB)

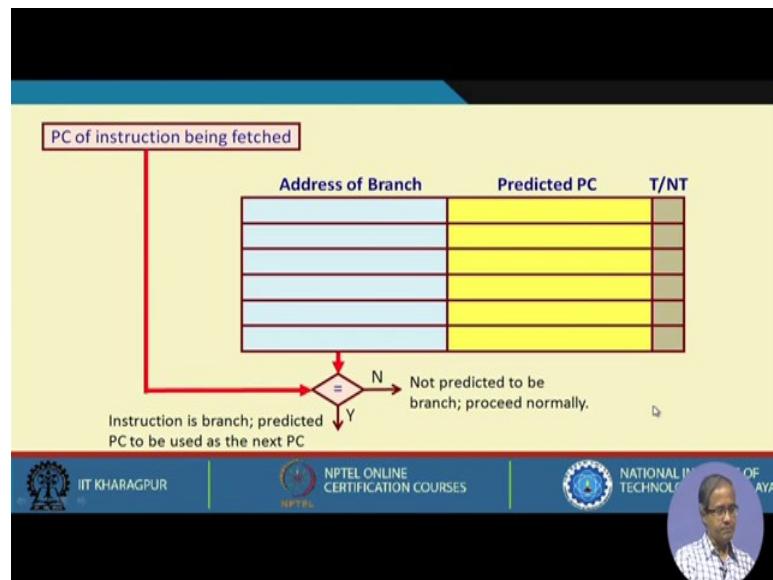
- The BTB is a high-speed memory that stores the predicted address for the next instruction after a branch.
 - Since we are predicting the next instruction address from where fetching will start *before decoding* the instruction, it is required to know whether the fetched instruction is predicted as a taken branch.
 - The PC of the instruction being fetched is matched against a set of instruction addresses stored in the BTB → represent addresses of *known branches*.
 - BTB also stores whether the branch was predicted T or NT, and helps keep the misprediction penalty small.

IT Kharagpur | NPTEL Online Certification Courses | National Institute of Technology, Meghalaya

We use something called branch target buffer. BTB is the high-speed memory just like the earlier data structure we saw like BPP, that stores the predicted address for the next instruction after branch. Whenever there is a branch instruction from where you will be fetching the next instruction will be stored there; either the next instruction following or the target instruction of branch. Since we are predicting the next instruction from where we are fetching before the decoding, because we will also have to know whether fetch instruction is predicted as a taken branch or not. We will see this data structure how it is there.

Just as I said the PC value from where you are fetching the instruction also needs to be stored in that memory. Because that will tell me whether that is the known address of a branch instruction.

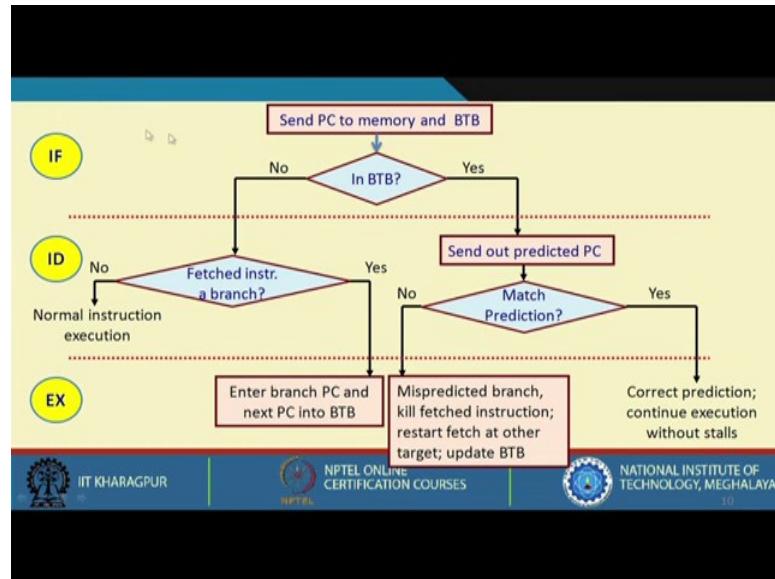
(Refer Slide Time: 23:16)



Let me show a diagram. First you see how the table looks like. There are three parts; the first part is the address of the branch instruction -- this is the PC value, then the predicted next address. While you are fetching an instruction this is actual an associative memory we will have to search this in parallel. When an instruction is being fetched, you parallelly check whether this PC value is already present here or not. If we see that it is already present, it means that the instruction is a branch instruction because only the addresses of the branch instructions should be stored in this table.

If the PC is matching here you will definitely know that this is a branch instruction and if it is a branch instruction, then the corresponding predicted PC value, that you will be using as your next PC from where to fetch the instruction. But if you see that it is not matching, then it is not predicted as a branch you proceed normally. And also there is a third field taken or not taken. You also keep this so that if there is a prediction mismatch you can update this. The PC of the instruction being fetched is matched against the set of instruction addresses stored in the table. As it said this table contains addresses of branches that were encountered earlier, and it also stores whether the branches were predicted as taken or not taken.

(Refer Slide Time: 25:28)



This is the data structure and this is how you are comparing. This is the flowchart, which tells you what happens during the IF stage what happens during ID, what happens during EX. Now during the IF stage as you can see your sending the PC to memory for fetching the instruction, in parallel you are also send it to BTB for searching. Suppose you find a match, which means it is a branch instruction, you send out the predicted PC. At the end of the ID the instruction that was fetched will also have completed the decoding process. Then you also know whether your prediction and the decoding of the instruction is matching or not. If you see that the predictions match, which means whatever predicted PC value you are using is the correct prediction, you continue execution without any stalls.

But if you see that prediction is wrong because your table says that it is taken, but after decoding you see that it is not taken; that means, it is a mispredicted branch. So, if it is a mispredicted branch then you will have to kill the instruction we have fetched. You need a stall here, and you will have to restart fetch at the other target, and once you do this you will also have to update BTB because now your prediction has changed.

You will have to modify this T and NT and also update the predicted PC, and here if it is not in the BTB it can mean two things, that it is either not a branch instruction or it is a branch instruction, but you are seeing it for the first time. Whether or not it is a branch instruction, that again you will come to know during ID, if you see that it is a branch

instruction then you will have to enter this information into BTB. Because you are seeing the branch instruction for the first time, but if it is not a branch instruction you proceed normally to the EX cycle.

(Refer Slide Time: 28:10)

The slide has a blue header bar with the title 'Branch Prediction'. Below the header is a yellow content area containing a bulleted list and a table. At the bottom is a red footer bar with logos for IIT Kharagpur, NPTEL, and NIT Meghalaya.

- There will not be any branch penalty if an entry is found in the BTB and is correct.
 - Otherwise, there will be penalty of at least one clock cycle.
 - Since the BTB may also need to be updated, the penalty can be two clock cycles.

Found in BTB	Prediction	Actual Branch	Penalty Cycles
Y	T	T	0
Y	T	NT	2
Y	NT	NT	0
Y	NT	T	2
N	-	T	2
N	-	NT	1

NPTEL ONLINE CERTIFICATION COURSES

This table shows the various penalties, you see if an entry is found in the BTB and the prediction is correct -- prediction is T and the actual branch is also T, then there is no penalty; or it is NT and branch is NT then also there is no penalty.

But if the prediction is wrong the penalty cycles can be 2. Why 2? Earlier we had seen that penalty is 1 for branch, but here since you also have to update the BTB, we assume that for updating BTB we need another cycle. That is why the penalty is 2. The other table entries can be explained similarly.

(Refer Slide Time: 29:43)

Example 2

- Consider the following parameters of a MIPS pipeline with BTB:
 - 90% of the branches are found in BTB.
 - 8% of the predictions are incorrect.
 - 75% of the branches are taken.

Compute the branch penalty.

Delayed branch requires penalty of 0.5 clock cycles on the average

$$\begin{aligned}\text{Branch penalty} &= (\% \text{ Branches found in BTB} \times \% \text{ Mispredictions} \times 2) \\ &\quad + (\% \text{ Branches not found in BTB} \times \% \text{ Taken branches} \times 2) \\ &\quad + (\% \text{ Branches not found in BTB} \times \% \text{ Not-taken branches} \times 1) \\ &= 0.90 \times 0.08 \times 2 + 0.10 \times 0.75 \times 2 + 0.10 \times 0.25 \times 1 = 0.32 \text{ clock cycles}\end{aligned}$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MANGALORE

This is a small example that you can work out based on these values. There is a BTB, and 90% of the branches in a program are actually found in BTB, and 8% of the predictions are incorrect and the remaining 92% are correct. And 75% of the branches are actually taken. What will be the branch penalty? The branch penalty can be calculated as a weighted sum using values from the table. The final value is coming to 0.32 clock cycles.

(Refer Slide Time: 31:33)

Conditional Instructions

- Conditional instructions can help eliminate some branch instructions and hence also the corresponding branch penalties.
 - MIPS32 has a number of conditional instructions (e.g. conditional move).

An example code:
`if (X == 0) A = B;`

Assume:

- R1 holds X
- R2 holds A
- R3 holds B

Using branch
`BNEZ R1, L
ADDU R2, R3, R0`

L:

Conditional move
`CMOVZ R2, R3, R1`

Essentially, we converted the control dependence into data dependence.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MANGALORE

And lastly we will look at some conditional instructions that also help in reducing branch penalties. In fact, MIPS has a number of such conditional instructions. What are conditional instructions? Let us look at a justification why need this, consider a C code like this “if ($X == 0$) $A = B;$ ” ; this a conditional move.

Let us say register R1 holds the address of X, R2 holds the address of A, R3 holds the address of B. Using conventional branch instruction our program will be like this. But there is a conditional move (CMOVE) instruction in MIPS; it says if R1 is 0 then move R3 to R2. Thus, you can avoid the branch instruction altogether. Whatever control hazards were there you are avoiding this. Basically we are converting the control dependency into data dependency. MIPS has a whole set of conditional instructions just for this purpose, the compiler can use it to advantage and eliminate many branch instructions.

With this we come to the end of this lecture. Over the last few lectures we discussed the various hazard scenarios in the MIPS pipeline and looked at many of the modern techniques that are used to detect and mitigate the effects of hazards. You shall see some other advanced issues later in some lectures.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 59
Multicycle Operations in MIPS32

In our earlier discussions on the MIPS32 pipeline, we basically concentrated on the integer instructions. We had seen how the pipeline works, we had seen the effects of the hazards, the interrupts and various ways to handle these problems and to improve the performance. But now let us come to the real scenario, in a real computer, we do not only have the integer instructions, we need also to process scientific data that are in the form of floating point numbers. So, we need floating point arithmetic; floating point processing is to be done in the computer system itself. So far we have assumed our EX stage will be able to finish a computation in a single cycle, but that is a little too optimistic realistically speaking. Can you finish a multiplication or a division operation in one cycle?

Well you make the clock sufficiently slow, but that is not a good idea. If you do that everything else will also become slow. Moreover, we have seen that for floating point operations, we need multiple cycles to carry out the operation itself, which means essentially the EX cycle itself may require multiple clocks. We are talking about something called multi-cycle operations. The topic of our lecture today is multicycle operation in MIPS32.

(Refer Slide Time: 02:24)

The slide has a dark blue header bar. Below it, a yellow section contains the title 'Introduction' in bold black font. Underneath the title is a bulleted list of points. At the bottom of the slide is a dark blue footer bar with three logos: IIT Kharagpur, NPTEL, and NIT Meghalaya.

Introduction

- Real implementation of MIPS32 will consist of both integer and floating-point units.
- Floating-point operations are more complex than integer operations.
 - Will require more than one cycles in the EX stage.
 - Makes the pipeline scheduling and control more complex.
 - New types of data hazards may appear that are otherwise not possible in the MIPS32 integer pipeline.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

As I said that the real implementation of MIPS32, where you can run all kind of applications, will contain both integer and floating point units. Floating point operations are more complex than integer operations, not only that even within in the integer operations as I said multiply and divide can be more complex than add and subtract.

These kind of operations are slower in general and may require more than one cycles to finish during the EX stage. This obviously makes the pipeline scheduling and controlling much more complex, and we shall see more different interesting kinds of hazards can appear because of this. Because you see earlier what we assumed is that that all instructions are of the same size, they take same time 5 units; but now we say instructions can vary in time. May be the earlier instruction will be finishing later, next instruction will be finishing earlier, lots of such problems can occur.

(Refer Slide Time: 03:55)

(a) Solution 1

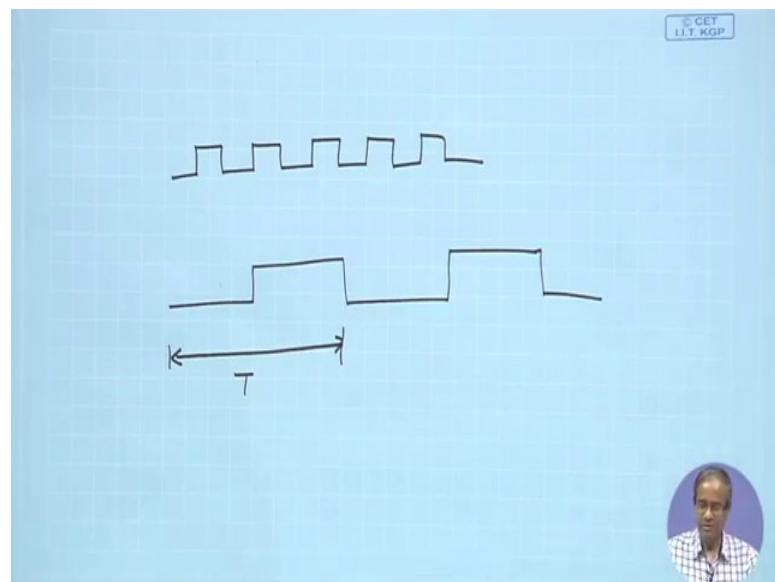
- Do not make any change in the pipeline control.
- Use a slow clock such that the ALU operations for floating-point instructions can finish in one clock cycle (in EX stage).
- Drawback:
 - Other operations are also slowed down, causing severe degradation in performance.
 - Not acceptable in practice.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES |  NATIONAL INSTITUTE OF TECHNOLOGY KALYANI



The first and naive solution is not to make any change in the control, just slow down the clock, you give sufficient time within a clock period to complete it. May be earlier your clock signal was like this.

(Refer Slide Time: 04:22)



But now you make the clock like this. Obviously, this is not a good idea because you are making the clock period slow, means all the stages not only EX; IF, ID, EX, MEM, WB, will all be running at this speed T . If you slow down the clock everything will slow

down. Clearly this is the drawback as I said other operations are also slowed down and this will cause a severe degradation in performance.

(Refer Slide Time: 05:11)

(b) Solution 2

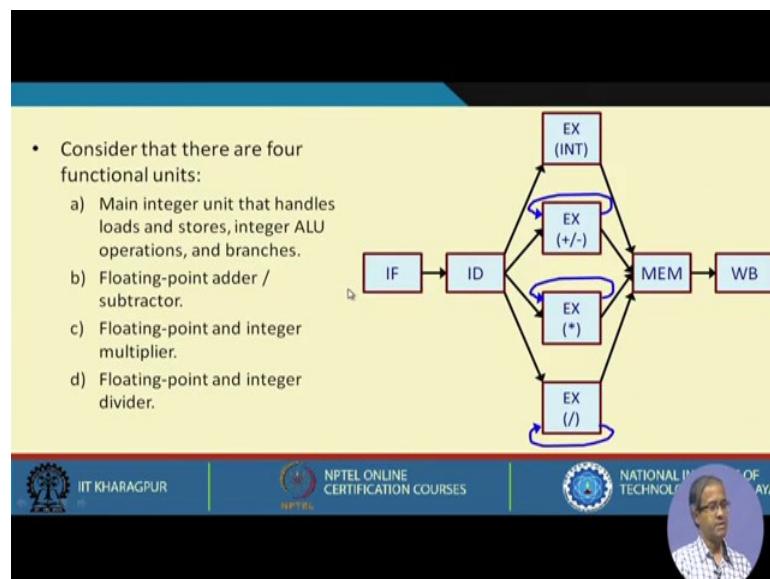
- We allow the floating-point arithmetic pipeline to have a longer latency.
 - EX cycle is considered to be repeated several times.
 - The number of repetitions can vary depending on the operation.
- The EX stage will have multiple floating-point functional units.
 - For example, one for addition/subtraction (pipelined), one for multiplication (pipelined), and one for division (non-pipelined).
 - A stall will occur in the pipeline if the instruction to be issued will either cause a structural hazard for the functional unit, or a data hazard.
 - Pipelining the functional units can avoid the structural hazard.

A more practical solution is that we do not change or modify the IF, ID, MEM and WB because they do not change, they remain as it is. The only concern is the EX where the arithmetic operations get carried out, which we are saying can vary in complexity or. Some are integer, some are floating point, even within integer, they can be of various complexities this is our concern. Let us assume integer and floating point for the time being. We allow floating point arithmetic pipeline to have a longer latency. Logically speaking what we have saying is that in the instruction execution cycle, the EX cycle is repeated more than one clock cycles.

So, we are repeating the EX cycle several times and how many times we are repeating depends on the complexity of the operation. Here as you shall see that in a real computing platform, EX stage will actual be having multiple functional units; not just one. May be there will be one functional unit that will be handling addition and subtraction. There will be one functional unit for multiplication and one for division. Now there can be stalls due to structural hazards as well, since there is a possibility of a stall if you are trying to issue an instruction means your instruction moves from ID to EX, it may so happen that the earlier instruction which was there in the pipeline, it was using the same functional unit and it was still somewhere in the EX.

So, the next instruction cannot enter EX; unless the EX or the arithmetic unit is itself pipelined. You recall, we discussed the pipeline implementation of some of the floating point operations, this is why we need that. If they are, not pipelined then while a previous instruction is doing multiplication the next instruction has to wait for the multiplication until the earlier one has completed. But if the multiply operation itself is pipelined, when the first instruction is in the second stage of multiplication, the next instruction can enter the first stage. They can proceed in overlapped fashion.

(Refer Slide Time: 08:38)



If you have pipelining in the functional unit, then you can avoid structural hazard; otherwise, there will be structural hazard in the pipeline in the EX stage. This is a logical picture where we are assuming that the EX stage has 4 functional units. The first one is our conventional integer unit that we have seen earlier, for that the sequence will be IF, ID, EX, MEM, WB. The main integer unit will be handling all load and store operations from memory, the integer ALU operations and also branches. There will be a floating point adder, subtractor, another functional unit, this blue arrow indicates that you may have to iterate it several time, this is a multicycle EX stage, there can be a multiplier floating point and also integer multiplier, I said integer multiplication can also take more than one cycles. So, this stage can do both floating point or integer multiply, and this is a division floating point or integer division unit.

As you can see; other than the integer EX unit, the other EX units are multi-cycle, they will need multiple cycles of EX.

(Refer Slide Time: 10:01)

MIPS32 Floating-Point Extension

- In the floating-point extension of MIPS32, there are 32 floating-point registers F0 to F31, each of size 32 bits.
- For double-precision operations, register pairs can be used to store the data:
 - Register pair <F0, F1> → referred to as F0
 - Register pair <F2, F3> → referred to as F2
 - Register pair <F30, F31> → referred to as F30
- Some examples of double-precision floating-point instructions are shown in the next slide.

IIT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MANGALORE

Earlier we had briefly looked at the MIPS32 floating point registers and some of the instructions therein. Just a quick recap; in the MIPS32, there are 32 floating point registers which are named F0 to F31, they are all of 32 bits in size. When we work with double precision numbers; that means, 64 bits, then we can take register pairs <F0, F1>, <F2, F3> and so on, till <F30, F31>, and are named as F0, F2, ..., F30 respectively. Like this, you can use register pairs for double precision operations. Now I am giving here some examples for such operations.

(Refer Slide Time: 11:06)

The slide has a yellow header bar with the title "Floating-Point Instruction Examples". Below the header, there are three numbered items:

- Load into a floating-point register pair:
L.D F6, 200(R2) // F6 = Mem[R2+200]; F7 = Mem[R2+204];
- Store from a floating-point register pair:
S.D F4, 40(R5) // Mem[R5+40] = F4; Mem[R5+44] = F5;
- Arithmetic operations on floating-point register pairs:
ADD.D F0, F4, F6
SUB.D F12, F8, F20
MUL.D F4, F6, F8
DIV.D F8, F8, F10

At the bottom of the slide, there are logos for IIT Kharagpur, NPTEL, and National Institute of Technology, Rourkela, along with a circular video player showing a person speaking.

Let us say the first instruction L.D means load double precision, you see this is loading from memory. Whenever you are specifying the memory address, we are specifying the integer register 200(R2), which means memory $200 + R2$, and we will be loading into F6 and F7. This is $R2 + 200$, this will be $R2 + 204$ because memory is byte oriented and every word is 32 bits. Similarly store; S.D means store double.

Similarly, you can have the other operations; all are double precision; ".D" means double.

(Refer Slide Time: 12:26)

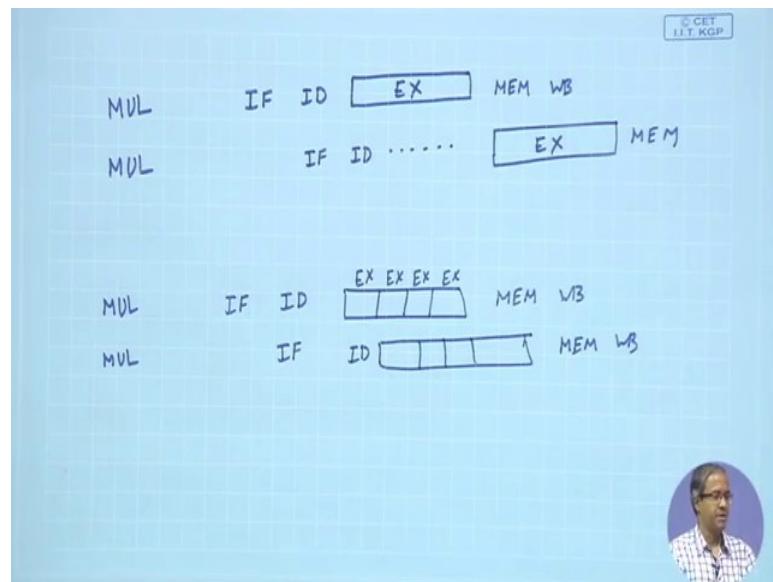
The slide has a yellow header bar with the title "Latency and Initiation Interval". Below the header, there is a bulleted list:

- The multi-cycle arithmetic units are often pipelined to allow overlapped operation and hence improved performance.
- Definitions:
 - Latency:** The number of cycles between an instruction producing a result and another instruction using it.
 - Initiation Interval:** The number of cycles that must elapse between issuing two operations of the same type.

At the bottom of the slide, there are logos for IIT Kharagpur, NPTEL, and National Institute of Technology, Rourkela, along with a circular video player showing a person speaking.

We talk about latency and initiation intervals. In this kind of multi-cycle pipelines whenever we talk about latency, what does latency means. You see just as I mentioned that the arithmetic units are often pipelined wherever possible, because unless you do a pipeline, you cannot overlap the operations. I am giving a small example; suppose there is one multiplication instruction followed by another multiplication instruction.

(Refer Slide Time: 13:06)



Let us say I have an instruction; IF, ID, then EX will take much longer; I am showing it like this much extended execute cycle, then MEM then WB. But for this one, you can do IF here, you can do ID here, but you cannot start EX until this early instruction has finished it. You can start the EX cycle only here, after this has finished, then MEM, this is if it is not pipelined. But if it is pipelined then your situation will be like this – IF, ID, let us say in the EX it takes four stages. So, there will be 4 cycles of EX, then MEM and WB. For the second one; you can do IF here, you can do ID here, and it can start the EX from here itself.

Now latency means -- now we are talking about the data dependencies, there is one instruction that is producing a result and another instruction is using it. How many minimum number of clock cycle will be required between them? Like we have seen earlier that a load instruction followed by its use, in an integer case, requires one stall cycle, which means latency was 1. But for floating point operations latency can be more. And initiation interval means how frequently we can issue instructions of the same type.

If it is fully pipelined, we can issue it every cycle, like in the previous example of multiplication I have shown. These are the typical values that we will assume. These values are very realistic; for integer operations that we was earlier latency is 0 and initiation interval 1.

(Refer Slide Time: 15:22)

Typical Values Assumed

Functional Unit	Latency	Initiation Interval
Integer ALU	0	1
Data Memory (integer / FLP load)	1	1
FP add / subtract	3	1
FP multiply	6	1
FP divide	24	25

Assumptions on number of EX stages:

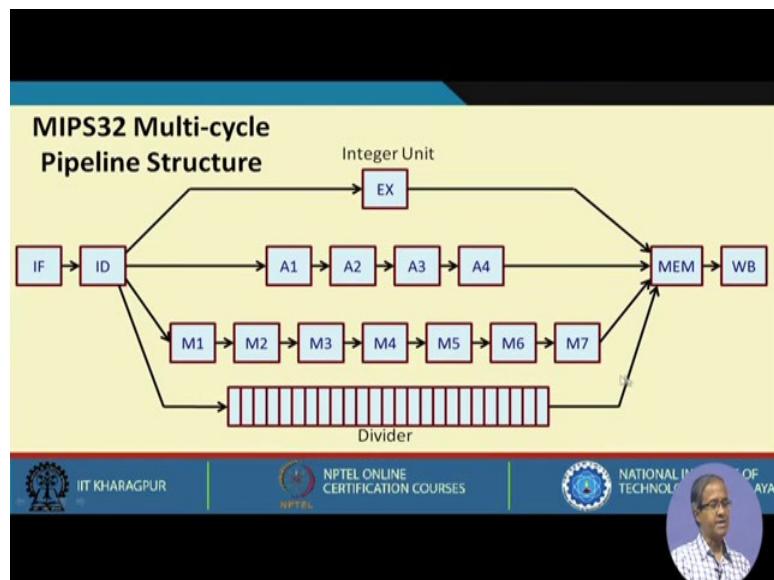
- FP add/subtract: 4
- FP multiply: 7
- FP divide: 1 (not pipelined)

It is possible to have up to:

- 4 outstanding FP add/subtract
- 7 outstanding FP multiply
- 1 FP divide.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY TRIVANDRUM

(Refer Slide Time: 17:35)



Our architecture looks like this. These are the pipeline stages IF, ID, MEM and WB, and here these are the EX stages. The integer unit has a single EX, for floating point addition there are 4 stages, for floating point multiplication there are 7 stages, and division is a

single stage, but with delay of 25. It is non-pipelined, this is how a typical you see division. It is not pipelined because the cost of pipelining a divider is much more, only for very sophisticated number crunching computing platforms; you need very fast division operations there you can do it, but otherwise the frequency of division is much less as compared to other operation like addition subtraction multiplication.

(Refer Slide Time: 18:51)

Some Scenarios												
MUL.D	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB	
ADD.D	IF	ID	A1	A2	A3	A4	MEM	WB				
L.D		IF	ID	EX	MEM	WB						
S.D			IF	ID	EX	MEM	WB					

Out of order completion of instructions

L.D F8,0(R5)	IF	ID	EX	MEM	WB							
MUL.D F4,F8,F10	IF	ID	-	M1	M2	M3	M4	M5	M6	M7	MEM	WB
ADD.D F6,F4,F12	IF	-	ID	-	-	-	-	-	A1	A2	A3	A4
S.D F6,0(R5)			IF	-	-	-	-	-	ID	EX	-	-

Stalls arising due to RAW hazards



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



NATIONAL INSTITUTE OF
TECHNOLOGY, MEGHALAYA

There are some RISC architectures where division is not pipelined. Some interesting scenarios you can see here. I have shown one multiply followed by add; these are all double load and store. The multiply will need 7 cycles in the EX stage, M1 to M7; add will require 4 cycles; load and store will take 5 cycles in total. Now you see, out of order completion of instructions occur. The third instruction finishes first, fourth instruction finishes next, then the second instruction, then the first instruction. There will be out of order completion of instructions. The second instruction illustrates some interesting read after write hazards; you see here the example of we will take as a load which loads a double precision value in F8, then multiply which uses F8 stores result in F4, and add which uses F4 result in F6, and the store which stores in F6.

The load will take 5 cycles like this, loaded value is available at the end of MEM. With data forwarding, multiply operation can start only after the data is loaded after MEM. So, there will be one stall cycle required. Here multiplication can start. Now add requires F4, here again even using data forwarding, F4 will be known only after multiplication is over

here at the end of M7. So, add we can issue, that means from ID to EX, only after that; there will be 6 stall cycles here. Similarly add will finish here at the end of A4, but after A4, there is a MEM stage, and store will get its MEM stage only after that and store will happen here. So, there will be 3 additional stalls here.

We can see that there are so many stalls because of RAW hazards in floating point operations. You see the number of stall cycles is significantly higher in this case for multi-cycle operations. The compiler needs to take care of these kind of things and there are very sophisticated techniques which compilers use; some these we will discuss, to prevent and try to fill up these delay slots.

(Refer Slide Time: 21:53)

The slide shows a pipeline diagram with 12 stages: IF, ID, M1, M2, M3, M4, M5, M6, M7, MEM, and WB. The first instruction (MUL.D) has three write-backs (F4, F8, F10) occurring in stages M1 through M7. The second instruction (SUB.D) has four write-backs (F6, F8, F10) occurring in stages M1 through M5. The third instruction (L.D) has one write-back (F6,0(R5)) occurring in stage M1. A note below the diagram states:

- Three instructions are trying to write into the FP register bank simultaneously.
- WAW hazard for the last two conflicting instructions (both writing F6).
- No conflict in MEM as only the last instruction accesses memory.

Logos for IIT Kharagpur, NPTEL, and NIT Trichy are visible at the bottom, along with a portrait of a speaker.

Now another interesting example is shown. This result in a write after write (WAW) hazard that was never possible in an integer pipeline. See there are 3 instructions. I am showing how MUL, a SUB and a L (load) with 2 instructions in between, I am not interested what instructions these are -- they are the independent instructions, but the point is that they are writing into some registers. MUL will require 7 EX stages, SUB will require 4 EX stages, L will require 1, but interestingly we see the all reach the WB stages together. There can be resource conflict structural hazard in the WB stage, first one is trying to write into F4, this one is trying to write into F6, this is also trying to write into F6. There is a conflict, these needs to be handled, these instructions should not be allowed to reach WB stage at the same time. Some stall cycles in WB also needs to be

inserted here, but for MEM in this example, there is no conflict because only L is accessing memory, but here means although these reach MEM together, but here there is nothing to be done in MEM for MUL and SUB, only this L requires MEM.

The conflict in WB is important, here you need to insert stalls. With this, we come to the end of this lecture where we have tried to appraise about the problems that arise in multi-cycle operations; what are the kind of issues and hazards that can show up and of course, in a real machine there has to be a very sophisticated mechanism to handle all these kinds of problems.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 60
Exploiting Instruction Level Parallelism

In this lecture, we shall be discussing about exploiting instruction level parallelism. If you recall, we talked about the multi-cycle operations; they occupy more number of clock cycles in the EX stage and if there are data dependencies then even with data forwarding significant number of stall cycles are required.

For the integer operations, earlier we saw that in the worst-case only one stall cycle is required, and here there can be several. We can again look towards the compiler and give the compiler much more responsibility to try and fill up the delay slots that are there.

So, now the compiler has to work much harder because now we have so many delay slots. There are many interesting techniques that the compilers use. There is something called instruction level parallelism, compiler tries to expose more parallelism and then utilize that to reduce number of stalls. We shall try to illustrate this with some examples.

(Refer Slide Time: 02:11)

Introduction

- To keep the pipeline full, we try to exploit parallelism among instructions.
 - Sequence of unrelated instructions that can be overlapped without causing hazard.
 - Related instructions must be separated by appropriate number of clock cycles equal to the pipeline latency between the pair of instructions.

Instruction producing result	Destination instruction	Latency (clock cycles)
FP ALU operation	FP ALU operation	3
FP ALU operation	Store double	2
Load double	FP ALU operations	1
Load double	Store double	0

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Our objective is to keep the pipeline full to reduce the number of stall cycles as much as possible. For that purpose, we will have to exploit parallelism among instructions. What

do really mean by parallelism? You see, if there is no parallelism, it means instructions are executed in sequence. In a particular order, second instruction depends on the first instruction, third instruction depends on the second instructions, fourth depends on the third instruction -- there is no parallelism. But if the instructions are independent, then we can run them in parallel; which means even we can exchange the order of instructions without any problem.

Whenever I can expose more parallelism, I can do two things. One is I can move instruction around much more freely. Secondly, if I have multiple functional units available, like I have a multiply, an adder, then I can try to execute an addition instruction and multiply instruction together such that there is some kind of parallelism.

As it is said, when we talk about parallelism, it means sequence of unrelated instructions that can be overlapped and if there are unrelated, obviously there will not be any data hazards.

But if the instructions are related; means the output of one instruction is used as the input of another instruction, then they have to be separated by appropriate number of clock cycles equal to the latency. Latency depends on the type of the operation. Now this table summarizes the typical latency figures between the broad kinds of arithmetic operations, particularly floating point we are focusing on. So, when there two floating point ALU operations the latency is 3.

These are the figures we shall be using to work out an example. In addition, we have seen earlier branches will have one clock cycle delay, we are not talking about predictive methods, the hardware prediction other things -- simple branch using branch delay slot.

(Refer Slide Time: 05:34)

- In addition, branches have one clock cycle delay.
- The functional units are fully pipelined (except division), such that an operation can be issued on every clock cycle.
 - As an alternative, the functional units can also be replicated.
- We now look at a simple compiler technique that can create additional parallelism between instructions.
 - Helps in reducing pipeline penalty.

We will incur one cycle delay and the other assumption that we make is: we discussed in the last lecture that the functional units like adder, multiplier are fully pipelined except the division unit, and because of the pipeline, you can initiate an operation like addition, subtraction, multiplication in every clock cycle.

The alternate philosophy could have been to have multiple functional units instead of pipelining, let us say the adder unit that consists of 4 stages, we could have used 4 adders, but that unnecessarily would increase the cost 4 times. So, pipelining is a much more elegant method where cost is not increasing that much, but effectively you are getting 4 times throughput.

We look at a compiler technique where some additional parallelism can be created. Suppose I have written a program, just we see some parallelism can be identified. The compiler can say, let me try to generate some more parallelism. That we will see. If it is done then the pipeline stall cycles can be reduced quite significantly.

(Refer Slide Time: 07:27)

Example 1

```
for (i=1000; i>0; i--)  
    x[i] = x[i] + s;
```

Add a scalar s to a vector x

Assume:

- R1: points to x[1000]
- F2: contains the scalar s
- R2: initialized such that 8(R2) is the address of x[0]

MIPS32 code

```
Loop: L.D F0,0(R1)  
       ADD.D F4,F0,F2  
       S.D F4,0(R1)  
       ADDI R1,R1,#-8  
       BNE R1,R2,Loop
```

Loop: L.D F0,0(R1)
 stall
 ADD.D F4,F0,F2
 stall
 S.D F4,0(R1)
 ADDI R1,R1,#-8
 BNE R1,R2,Loop
 stall

9 clock cycles per iteration (with 4 stalls)

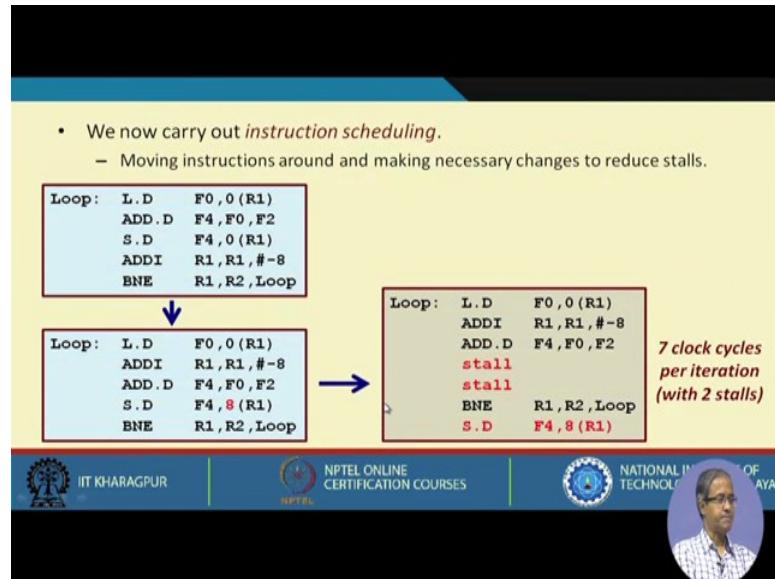
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY MANGALORE

Let us take an example like this. We assume that there is a vector; that means, an array of size 1000, s is a scalar number, and we are adding s to all the elements of the array. This is the C code and we are assuming that R1 points to the last element of the array, F2 register contains the scalar s, and R2 is pointing to the element which is just before the first element of the array; that means, if I add 8 to R2 it will be pointing to x[0]. So, actually R2 is pointing to one element before x[0]. We will be initializing R1, F2, R2 such that my code can become like this. We are loading 0(R1); R1 is pointing to x[1000]; that means, the first number is loaded into F0, F0 is added with F2 and store the result in F4, finally we are storing F4 back into 0(R1).

Every element is 64 bits; that means 8 bytes. So, after one addition, we are decreasing R1 by 8. We can either do ADDI minus 8, or SUBI 8 from R1. This way the loop goes on. When it crosses x[0], you stop or otherwise you go on.

Let us analyze this code first. You see there is a load followed by an add. According to our earlier table, a load followed by its use will incur 1 stall, and add followed by store will incur 2 stalls. For a branch, there will 1 stall cycle as usual. So, you see that for this loop that consists of five instructions, for every loop it will actually required 9 clock cycles per iteration, with 4 stalls.

(Refer Slide Time: 11:01).



Let us now try to do instruction scheduling that we learnt earlier. Let us keep the same code, let us try to move some instructions here and there and try to reduce these stalls. The first thing is that this add immediate instruction; this you are moving earlier, we are moving it here such that between load and add the stall disappears. We have moved this, add immediate here, and because we decremented this R1 earlier, now this store becomes 8(R1) because already we have decremented by 8. So, this 0 is change to 8, then we have the add followed by store and BNE, this was the modified thing. This store is moved to after BNE to fill up the branch delay slot, this also you can do because the store and branch are independent and whenever these branch is done this store will also happen along with that.

If you do that then you see for this loop, there will 7 cycles per iteration, there are 2 stalls. Our program had 1000 iterations in the first version, there were 9 clock cycles per iterations which means there were total of 9000 clock cycles required, but now in this version where having 7 clock cycles per iteration, we have brought it down to 7000. But you see with this code, however hard the compiler tries, it cannot improve any further. So, what is the way out? The way out is to do something called loop unrolling, let us try to explain what it is. You see this was our original loop.

(Refer Slide Time: 13:21)

The slide illustrates the process of loop unrolling. On the left, the original loop is shown:

```
Loop: L.D F0,0(R1)
      ADD.D F4,F0,F2
      S.D F4,0(R1)
      ADDI R1,R1,#-8
      BNE R1,R2,Loop
```

An arrow labeled "Unroll loop 3 times" points to the expanded loop code on the right:

```
Loop: L.D F0,0(R1)
      ADD.D F4,F0,F2
      S.D F4,0(R1)
      L.D F6,-8(R1)
      ADD.D F8,F6,F2
      S.D F8,-8(R1)
      L.D F10,-16(R1)
      ADD.D F12,F10,F2
      S.D F12,-16(R1)
      L.D F14,-24(R1)
      ADD.D F16,F14,F2
      S.D F16,-24(R1)

      ADDI R1,R1,#-32
      BNE R1,R2,Loop
```

A note below the expanded code states: $Cycles per iteration = 27 / 4 = 6.8$.

Logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Rourkela are visible at the bottom.

The original loop was looping 1000 times. Now what I am doing, I have written a small loop for adding 2 numbers and I am repeating it 1000 times. Now in the modified version; what I do? I unroll the loop, this is called unrolling. Unrolling means see earlier I was only adding $x[i]$ with s.

(Refer Slide Time: 14:17)

A handwritten note on a grid background shows a for loop unrolled 250 times. The loop adds four array elements sequentially in the same loop:

```
for (i=1000; i>0 ; i=i-4)
{
    x[i] = x[i] + s;
    x[i-1] = x[i-1] + s;
    x[i-2] = x[i-2] + s;
    x[i-3] = x[i-3] + s;
}
```

A bracket groups the four assignments, and the text "250 times" is written next to it.

Now what I do I add four array elements one by one in the same loop as shown. I unrolled the loop 4 times. In the for loop, earlier it was starting with $i = 1000$; same thing, but in the previous case, I was going up to 0. Here I will be doing $i = i - 4$,

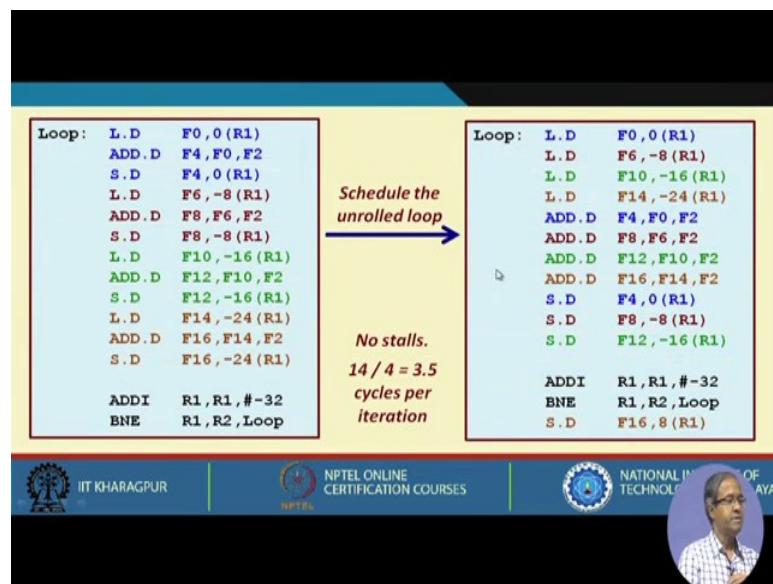
decreasing it by 4 times in every iteration, and will be repeating till i greater than equal to 0.

So, the idea is the same; what I do is that instead of the original loop that was repeating 1000 times, now i will loop 250 times, but in the body of the loop I have 4 copies of addition. There is no data dependency between these four additions; this is called loop unrolling.

The unrolled version of the code is shown. Once you have done this for 4 elements, instead of subtracting 8 from R1; now you are subtracting 32 from R1; and the branch condition remains the same at the end. Now you see even without doing an instruction scheduling, we just calculate the stall cycles. In total there are 14 instructions, you count there are 14 instruction plus 13 stalls, so total number of cycles is 27. In this version of the program cycles per iteration will be $27 / 4 = 6.8$.

Now you see we have exposed so much parallelism. We can move instructions around much more freely.

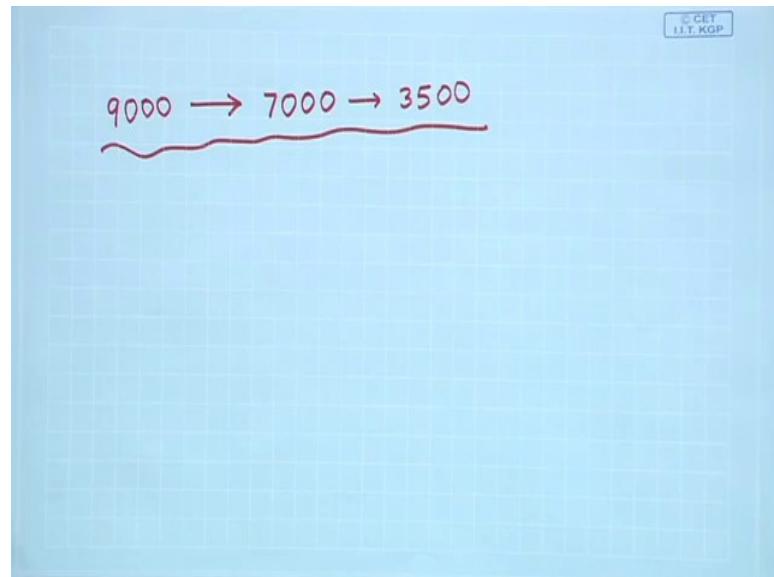
(Refer Slide Time: 18:07)



Let us move instructions around. What I have done is, all the load instructions I have pushed together in the beginning. All the 4 adds I have pushed together, there are 4 stores, I have put in the branch delay slot and I have adjusted the numbers according 0, -8, -24. The index of the last store is also adjusted accordingly.

You see in this version, there are no stalls because there are no dependencies; they are all independent. There are 14 instructions, and 14 clock cycles. So, $14 / 4 = 3.5$ cycles per iteration. If you just compare in the earlier version for computing 1000 numbers,

(Refer Slide Time: 19:58)



you were doing effectively 9 cycles per iteration; from there, after instruction scheduling you went to 7 cycles per instruction; but here we have brought it down to 3.5.

You see there is a quite drastic reduction in the number of clock cycles.

(Refer Slide Time: 20:44)

Loop Unrolling :: Summary

- Loop unrolling can expose more parallelism in instructions that can be scheduled.
 - Effective way of improving pipeline performance.
- Can be used to lower the CPI in architectures where more than one instructions can be issued per cycle.
 - a) Superscalar architecture
 - b) Very Long Instruction Word (VLIW) architecture

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

If the compiler does this instruction scheduling, it can bring down the number of clock cycle to a great extent. To summarize, loop unrolling can expose more parallelism in instructions.

So far the pipeline he have seen, our ideal CPI was 1, but now we are talking about machines where this CPI value can be less than 1; using some kind of parallelism.

Let us say two instructions are executing together. In every cycle, we are executing two instructions. The CPI will be 0.5. So, broadly there are two approaches -- we will be briefly talking about, (a) superscalar architecture, and (b) very long instruction word (VLIW) architecture.

(Refer Slide Time: 22:04)

A Superscalar Version of MIPS32

- Superscalar Machines:
 - Machines that can issue multiple independent instructions per clock cycle when they are properly scheduled by the compiler.
 - Can result in a CPI of less than 1.
- How does it work?
 - The hardware can issue a small number (say, 2 to 4) of independent instructions in every clock cycle.
 - The hardware checks for conflicts between instructions.
 - If the instructions are dependent, then only the first instruction in the sequence will be issued.

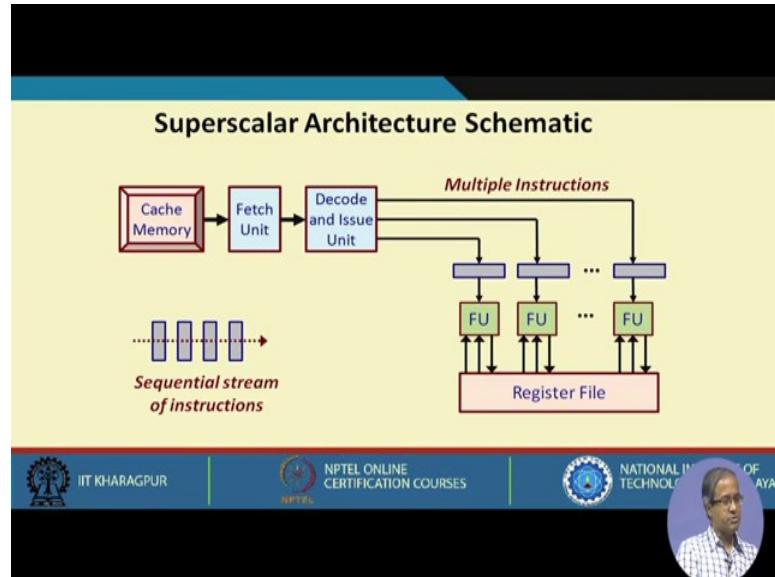
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY WARANGAL

First we look at a superscalar version of MIPS32. A superscalar machine is a computer system, which can issue multiple instructions in every clock cycle. We have seen a pipeline for MIPS32. Now you imagine a superscalar version of MIPS where there are two pipelines. In every clock cycle we will be fetching 2 instructions, and will be feeding the 2 instructions in the 2 pipelines. In every clock we will be feeding 2 instructions to the pipeline. In general the number of such pipelines can be more, it can be 4 or even higher, this is what is meant by superscalar.

Machines can issue multiple independent instructions. If there is a dependency between instructions you cannot start them together. We can fix the number of instructions to 2 to

4, the hardware can check for conflicts whether they can start together; if there is a conflict then only one of the instruction can be issued and the other has to wait.

(Refer Slide Time: 23:41)



The superscalar architecture schematically looks like this. The fetch unit will be more sophisticated, from cache memory it will have to fetch more than one instruction every cycle. Decode and Issue unit will also be decoding several instructions together and there will be multiple pipelines, I am calling them functional units, you can imagine that as if they are independent pipelines, they are all accessing a common register file.

Depending on the capability of the machine suppose 4 instructions will be fetched together, they will issue 4 instructions concurrently to the 4 pipelines. So conceptually, it is like this -- the instructions are stored sequentially, they are fetched much faster, and are fed to the 4 pipelines. This is the concept of superscalar architecture.

(Refer Slide Time: 24:54)

Example

- Suppose two instructions can be issued every clock cycle.
 - One can be a load, store, branch or integer ALU operation.
 - The other can be any floating-point operation.

Integer instr.	IF	ID	EX	MEM	WB		
FP instr.	IF	ID	EX	MEM	WB		
Integer instr.		IF	ID	EX	MEM	WB	
FP instr.		IF	ID	EX	MEM	WB	
Integer instr.			IF	ID	EX	MEM	WB
FP instr.			IF	ID	EX	MEM	WB

- Used only for illustration.
- We have not shown how FP operations extend the EX cycle.



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



NATIONAL INSTITUTE OF
TECHNOLOGY TRIVANDRUM



An example for a superscalar architecture with 2 functional units is shown.

One of them can handle load, store, branch or integer operations, and the other functional unit can handle floating operations. In every clock cycle, you can start 2 instructions, one integer and one floating point. Here we are not showing multicycle operations, because in general for floating point there will be multi-cycle, just for the sake of illustration we are showing like this.

(Refer Slide Time: 25:42)

- How to check dependency between instructions in a stream?
 - Can be checked dynamically by the hardware.
 - Compiler can take the complete responsibility of creating a package of instructions that can be simultaneously issued.
 - Hardware does not dynamically take any decision about multiple issue.
 - Also referred to as *VLIW architecture*.



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



NATIONAL INSTITUTE OF
TECHNOLOGY TRIVANDRUM



Dependency between instructions will be checked dynamically in the hardware, this is important. As an alternative, we can give some responsibility to the compiler, but that is not for superscalar architecture, for the other kind of architecture VLIW. What the compiler can do? See for superscalar; instructions are fetched, the hardware dynamically checks whether there are conflicts; if no conflicts they are sent to the pipeline together.

But now I am saying there is another kind of architecture called VLIW, where the compiler is trying to create packets of instructions like each packet will consist of 4 instructions and the compiler will ensure that the 4 instructions are such that there is no conflict between them. So, hardware need not check for anything, it simply takes a packet and issues to the 4 functional units just like that.

(Refer Slide Time: 26:52)

• Some issues:

- If we issue an integer and a FP operation in parallel, the need for additional hardware is minimized.
 - Different register sets and functional units are used.
- Only conflict is when the integer instruction is a FP load, store or move.
 - This creates contention for the FP register ports and can be treated as a structural hazard.
- In the original MIPS32 pipeline, load instructions have a latency of 1.
 - In the superscalar version, the next 3 instructions cannot use the result of load without stalling.
 - Branch delay also becomes 3 cycles.

Now there are some issues. Like for example, if we issue an integer and floating point operation in parallel because they use different register sets, different functional units. So, additional hardware required is less because they do not normally share register sets, the only conflict is when the instruction that is handling integer unit is a floating point load, where the loaded value has to be stored into a floating point register. That can lead to a hazard.

Another issue is that for the original MIPS pipeline, whenever there is a load instruction latency was 1, but for the superscalar version, it is not 1, it will be 3 because not only the next instructions, the next 2 instructions also have to wait. Thus, latency will be 3; as 3

instructions have to wait rather than 1. Similarly branch delay will also become 3 cycles and not 1 as in MIPS.

(Refer Slide Time: 28:11)

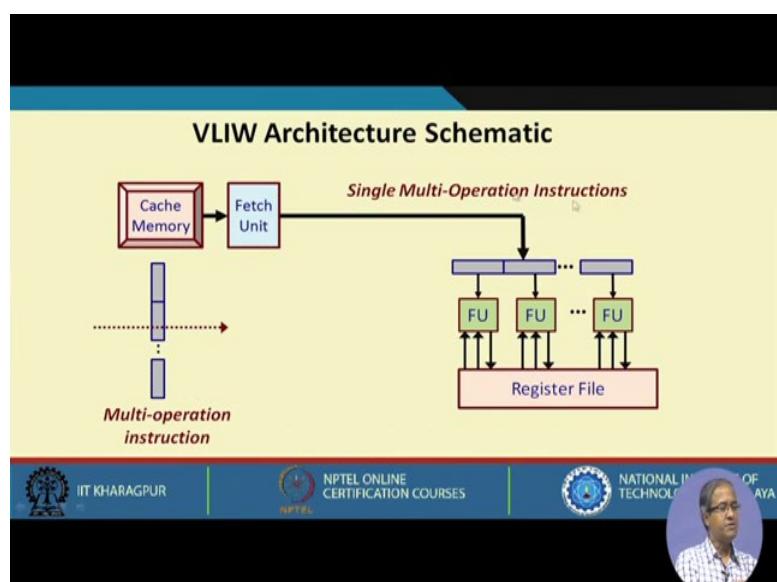
VLIW Architecture

- In a Very Long Instruction Word (VLIW) machine, an instruction word is typically hundreds of bits in length.
 - Specifies a number of basic operations / instructions, each using different functional unit.
 - Multiple functional units are used concurrently when a VLIW “*macro-instruction*” is being executed.
 - All the functional units share a common register file.
- Similar to superscalar architecture in concept, but responsibility of identifying set of instructions that can run concurrently lies with the compiler.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY ROURKELA

The instruction is made much wider, which means an instruction word can store several instructions together; several instructions together is referred to as a macro instruction may be 2 or 4 like that. There are similarly several functional units, the compiler will be generating these macro instructions, by grouping instructions that can run concurrently.

(Refer Slide Time: 28:50)



You see architecture wise here it looks similar, but it takes a packet of instruction. The instructions are coming as packet, not one by one. An entire packet comes where the instructions in a packet are guaranteed to be independent -- the compiler has ensured that. So, fetch will directly feed here. Let us take an example of the unrolled program that we saw earlier.

(Refer Slide Time: 29:20)

The slide contains the following text:

```

Loop: L.D    F0,0(R1)
      ADD.D  F4,F0,F2
      S.D    F4,0(R1)
      L.D    F6,-8(R1)
      ADD.D  F8,F6,F2
      S.D    F8,-8(R1)
      L.D    F10,-16(R1)
      ADD.D  F12,F10,F2
      S.D    F12,-16(R1)
      L.D    F14,-24(R1)
      ADD.D  F16,F14,F2
      S.D    F16,-24(R1)

      ADDI   R1,R1,#-32
      BNE   R1,R2,Loop
  
```

We try to schedule this unrolled program code on a VLIW processor, assuming that there are 4 functional units:

- Two memory reference units (to handle LOAD and STORE).
- One floating-point arithmetic unit.
- One integer operation and branch unit.

At the bottom, there are logos for IIT Kharagpur, NPTEL, and National Institute of Technology, Rourkela, along with a portrait of a man.

Let us see how we can run it on a MIPS processor and schedule it where we assume that MIPS processor as 4 functional units, where 2 of the functional units are memory reference units that can handle load and store. There is one floating point and one integer operation that can also handle branch.

We see for load and store, there are several instructions for floating point; there are add instructions, and for integer operation there is only this branch and this add immediate, , let us see how they can be scheduled.

(Refer Slide Time: 30:13)

Scheduling on a VLIW Processor			
Load / Store 1	Load / Store 2	FP ALU	Integer
L.D F0, 0 (R1)	L.D F6, -8 (R1)		
L.D F10, -16 (R1)	L.D F14, -24 (R1)		
		ADD.D F4, F0, F2	
		ADD.D F8, F6, F2	
S.D F4, 0 (R1)		ADD.D F12, F10, F2	
S.D F8, -8 (R1)		ADD.D F16, F14, F2	ADDI R1, R1, #-32
S.D F12, -16 (R1)			
S.D F16, -24 (R1)			BNE R1, R1, Loop

Clock cycles / iteration = 8 / 4 = 2.0



Here I am showing one possible scheduling; these are the 2 load/store functional units. This is the floating-point functional unit, this is the integer functional unit. The loads can be placed together in the first 2 cycles. After the loading is done we can put the adds here in FP unit. Then the stores, there will be delay up to 2 cycles -- you can schedule them like this. There are 8 cycles for 4 instructions. The cycles per instruction become 2.0, because of the parallelism supported by 4 parallel hardware units. For processing 1000 numbers, number of clocks will be 2000.

With this we come to the end of this lecture. Here we have looked at some of the ways of parallelizing or speeding up the basic MIPS32 processor that contains not only integer units, but also floating point units.

Nowadays many of the processors that we see around us are actually based on superscalar architectures. One thing I have not mentioned here; this is actually beyond the scope of this class. You see when you have multiple issues, so many instructions are coming; you are dynamically decoding them and detecting the presence of hazards. You need something called **scoreboard** that is a hardware based data structure maintained by the control unit dynamically, and it will immediately tell you whether there is a scope for any hazards or not. These we are not discussing in this lecture.

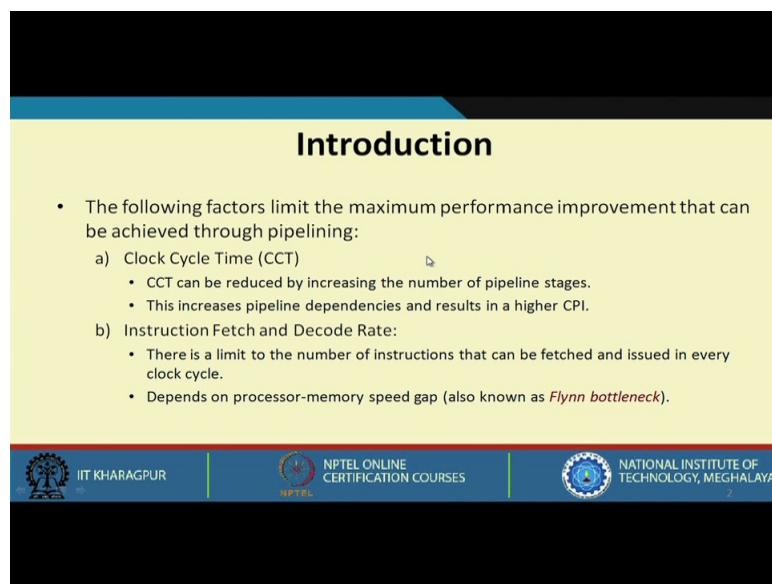
Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 61
Vector Processors

In our last lecture if we recall, we looked at various ways of increasing the instruction level parallelism in a program. We looked at the very well known technique called loop unrolling, which is used by compilers to expose more parallelism in a program, and also it enables the compiler to do instruction scheduling -- move instructions around to reduce the number of stall cycles. We also saw the so-called superscalar and VLIW architectures, which can enhance the performance of a pipelining by allowing more than one instruction to be issued in every clock cycle. In this lecture let us look one step further, let us say how we can further parallelize computations, but we are looking at specific kinds of computations. Computations that are carried out on something called vectors.

(Refer Slide Time: 01:26)



The slide has a dark blue header bar. Below it, a yellow section contains the title 'Introduction'. Underneath the title is a bulleted list:

- The following factors limit the maximum performance improvement that can be achieved through pipelining:
 - a) Clock Cycle Time (CCT)
 - CCT can be reduced by increasing the number of pipeline stages.
 - This increases pipeline dependencies and results in a higher CPI.
 - b) Instruction Fetch and Decode Rate:
 - There is a limit to the number of instructions that can be fetched and issued in every clock cycle.
 - Depends on processor-memory speed gap (also known as *Flynn bottleneck*).

At the bottom of the slide, there are three logos with their respective names: IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA.

So, the topic of today's lecture is vector processors. Let us try to see first that how we can look at the constraints in a pipeline and what are the factors that limit its performance. Broadly there are two factors: the first factor relates to the clock cycle time; like you see in a pipeline in if you can make the clock cycle time faster then

obviously, your instruction execution will become faster. So, your instructions will execute faster you can issue instruction at faster rate. And this clock cycle time can be reduced by making the pipeline stages simpler, which means increasing the number of stages. But if you increase the number of stages, there will be one difficulty; you see here earlier in the MIPS32 pipeline even though there were 5 stages only; you saw that there were lot of dependencies and hazard crating scenarios that were coming. By using forwarding and other techniques we could reduce most of them, but still for certain dependencies we had to use some stall cycles.

If you increase the number of stages even further, this constraint will become even more pronounced, there will be more kinds of dependencies that will be showing up. This clock cycle time reduction by increasing the pipeline stages can increase dependencies in general, which can result in a higher cycle for instruction CPI.

The second constraint is the instruction fetch and decode rate. Here what we have saying is that the instructions have to be ultimately fetched from memory. Earlier we were saying that one instruction has to be fetched per cycle, but now when we are moving into superscalar and other kind of parallel computation, here we are demanding that in every clock cycle we need to fetch more than one instructions. Well of course, this has something to do with the way we are organizing our memory. You have studied earlier that if we use memory interleaving, we can have some kind of parallel access. For instance for a 4-way memory interleaving we can access or read 4 words in every clock cycle. Organization of the memory also produces a constraint to the maximum memory CPU bandwidth.

(Refer Slide Time: 04:41)

- Vector Processor:
 - Provides high-level instructions that operate on entire arrays of numbers (called vectors).
 - A single vector instruction is equivalent to an entire loop.
 - No loop overheads are required.
- Example:
 - A, B and C are three vectors containing 64 numbers each.
 - The three vectors are mapped to vector registers V1, V2, V3 (say).
 - The following vector instruction computes $C_i = A_i + B_i$

ADDV V1, V2, V3

Now, this processor memory speed gap is sometimes also referred to as Flynn bottleneck. In general as I said there is a limit to the number of instructions that can be fetched in every clock cycle. Now, we are moving on to vector processors, let us see what a vector processor is. Here we are operating on entire arrays of numbers called vectors, like let us say I have an array of numbers not one, but several. Let us say there are total of 64 numbers. So, what I am saying is that this all the 64 numbers taken together, we are referring this as a vector.

(Refer Slide Time: 05:15)

© IIT KGP

Vector

$A, B \rightarrow \text{vectors}$

$C \rightarrow \text{vector}$

$\underline{C = A + B}$

What we are saying is that let us say A is a vector, B is also a vector, let us say C is also a vector, where you want to store the result. I can simply write an instruction $C = A + B$. This is an example of vector instruction, where although we are using a single instruction, but actually 64 additions are taking place. The first element of A is added to the first element of B, second element A is added to the second element of B, and so on, and the results are stored in corresponding elements of the vector C. So, in a conventional processor where these kinds of vector operations are not there, we had to use a loop. In every loop we have to add one pair of numbers, then decrement a loop counter, check for 0 or some condition then again branch back.

So, there are some loop overheads, but here it is a single instruction where there are no loop overheads. Expectedly they should run faster. As I had said single vector instruction is equivalent to an entire loop, and I had said no loop overheads are required. Just the example that I have given that we are adding two vectors, let us say the instruction will look like this “ADDV V1, V2, V3”. So, V2 and V3 are added element by element, and the result is stored in V1. Each of the elements can be, for example, double precision.

(Refer Slide Time: 07:40)

Basic Vector Processor Architecture

- A vector processor typically consists of an ordinary pipelined scalar unit plus a vector unit.
- All functional units within the vector unit are deeply pipelined, resulting in a shorter clock cycle time C.
 - Deep pipelining on vectors do not result in hazards, since every computation is independent of the others.
- We shall illustrate some concepts based on a hypothetical vector processor that is based on the MIPS32 architecture.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY RAIPUR

If you can map the vectors to the so-called vector registers, we can use a single instruction to add them. Let us now look at the basic vector processor architecture how does it look like. In a vector processor the basic hardware is nothing but a heavily

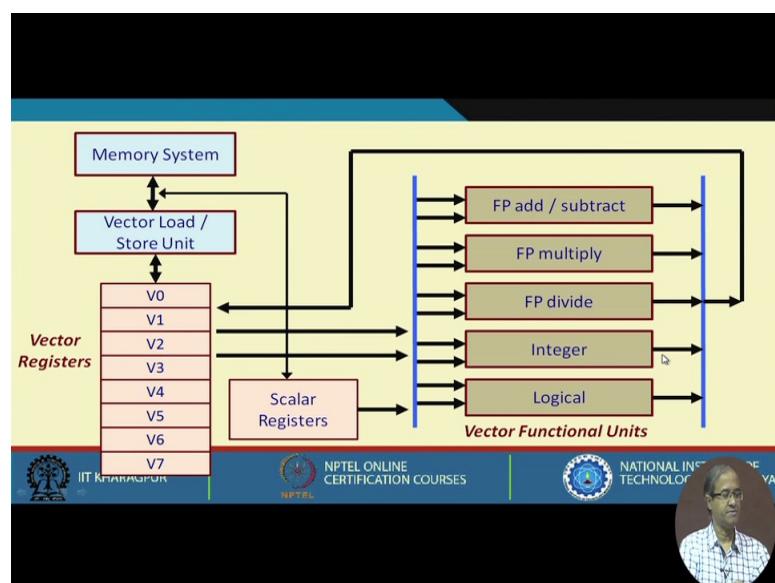
pipelined scalar unit. It is an extension of a pipelined processing unit, where in the execution unit we earlier talked about multi-cycle operations, they are also pipelined.

Here the arithmetic units are separately pipelined, they can be separately fed with data, and the results can be picked up separately. So, all the functional units will be deeply pipelined to the extent possible so that you can feed data at a faster rate. There will be two things: there will be an ordinary pipeline scalar unit, and in addition there will be a vector unit. Vector unit will consist of the vector registers, the arithmetic units and so on.

Inside the vector unit we have all the functional units like adder multiplier and divider etc., there all deeply pipelined so that for execution in the functional units the pipelined clock cycle time will be as less as possible. Now another thing you see here, in a normal instruction set pipelining when we are feeding a number of instructions sequentially in the pipe, there were the situations of hazards, but for vector operation suppose I am adding two vectors a and b storing the results in c.

There are 64 operations which are going on in parallel in overlapped fashion, but the operations are all independent -- as a[1] and b[1] are added result is stored in c[1], then a[2] and b[2] are added result in stored in c[2], and so on. There is no dependency across successive computation. So, there are no question of hazards coming in, this is another very good thing for vector processing.

(Refer Slide Time: 10:29)



These kinds of deep pipelining do not result in hazards, because the computations are independent. Let us look at a hypothetical vector processor that is an extension of the MIPS32 architecture. This is a very high-level schematic I am showing you; let us say there are 8 vector registers V0 to V7. Let us assume each of this vector registers can hold 64 double precision numbers.

So, each of the registers is actually a set of 64 64-bit registers, there are 8 such. And these vector registers, two of them can be read and you can write into one of them every cycle. And in addition there are the standard scalar registers as you saw in MIPS32, the integer and the floating point registers, there is a memory system, the memory will be interfaced to the scalar register just like as you saw earlier, but in addition there will also be a vector load store unit, through which you can load an entire vector from memory into one of this registers or you can store one of the vector registers into memory, and on this side you have the vector functional units.

So, you see there are so many vector functional units, each of them are having two inputs and one output, and they can be fed concurrently. So, the bandwidth of this bus should be high enough so that while some vector addition is going on in parallel, a vector multiply should also be going on.

(Refer Slide Time: 12:38)

- About the Vector Processor ISA:
 - Vector Registers
 - There are 8 vector registers V0, V1, ..., V7.
 - Each vector register can hold 64 double-words.
 - Each vector register has 2 read ports and 1 write port, to allow overlapped operations.
 - Vector Functional Units
 - Each functional unit is fully pipelined and can start a new operation every clock cycle.
 - A hardware control unit detects hazards (conflicts for functions units and also for register accesses), and inserts stalls as required.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY ROURKELA

Now, as I said in that architecture there are 8 vector registers. So, each vector can hold 64 numbers, each are double precision or double words. Each vector register has two

read ports and one write port as this diagram shows. It is not just for the whole bank, but for each of the vectors and for the vector functional units they are all fully pipelined; they can start an operation every clock cycle, and if there is any dependency means across this functional units like I am carrying out a vector addition, the next instruction is a vector multiplication which uses one of the vectors that is produce as the result. So, there will be latency or a stall you have to wait. This is kind of dependency is between instructions, but within an instruction there are no dependencies.

So, all the 64 operations can go on in an overlapped way without any stalls, without any loop overheads. If there any dependencies across instructions, only then stall cycles can be inserted.

(Refer Slide Time: 14:07)

The slide has a yellow background with a black header bar at the top. The content is organized into two main sections:

- Vector Load/Store Unit**
 - The load/store unit is also fully pipelined and allows fast loading and storing of vectors.
 - Memory system is also deeply interleaved to allow parallel access.
 - After an initial latency, one word can be accessed per clock cycle.
- Scalar Registers**
 - These are the normal scalar and floating-point registers of MIPS32.
 - Can be used to provide data as input to the vector functional units, as well as to compute memory addresses for vector load/store.

At the bottom of the slide, there are three logos and their respective names: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Rourkela. To the right of the NPTEL logo is a video feed of a person speaking.

Regarding the vector load stored unit as I said that is also fully pipelined and it allows fast loading and storing of the vectors, and to allow this the memory system has to be very deeply interleaved. So, 2-way or 4-way interleaving will not do may be it will be 16-way or 32-way or 64-way interleaved. So, memory organization has to be much more complex.

So, you need to modify the memory system as well. After the initial latency that can indicate the access time of the memory, you can access the words one per cycle. And in addition to that this scalar registers are also there like MIPS32, these are the normal integer and floating point registers. Now these registers can be used to carry out integer

operations of course, but they can also be used to provide data as input to the vector functional units. There are some operations where you may want to add say a scalar number to all the elements of a vector, the scalar number can come from one of the scalar registers. Moreover you can also use these scalar registers to compute the memory address, which will be used by the vector load store unit, from which memory address you want to load or store.

(Refer Slide Time: 15:48)

Example 1

- Consider the SAXPY or DAXPY vector operation: $Y = a * X + Y$ where X and Y are vectors (of size 64), and a is a scalar.
 - Rx contains starting address of X
 - Ry contains starting address of Y
 - R1 contains the address of the scalar 'a'.

MIPS32 Code	Vector Processor Code
<pre>L.D F0, 0(R1) ADDI R4, Rx, #512 L: L.D F2, 0(Rx) MULT.D F2, F0, F2 L.D F4, 0(Ry) ADD.D F4, F2, F4 S.D F4, 0(Ry) ADDI Ry, Rx, #8 ADDI Ry, Ry, #8 BNE R4, Rx, L</pre>	<pre>L.D F0, 0(R1) LV V1, 0(Rx) MULTSV V2, F0, V1 LV V3, 0(Ry) ADDVV V4,V2,V3 SV V4, 0(Ry)</pre>

IIT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MANGALORE

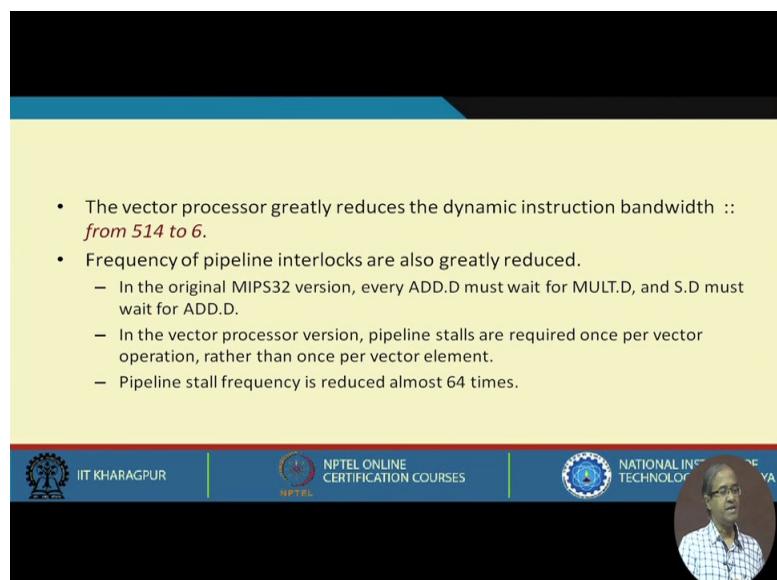
So, scalar and vector registers can be used together. Let us take an example. We look at a computation like this: $Y = a * X + Y$, where X and Y are vectors, 'a' is a scalar, this is sometimes called single precision SaXpY or double precision DaXpY loop. SaXpY stands for $a * X + Y$; if it is double precision we call it as DaXpY. Here we are assuming that X and Y are both vectors of size 64 and 'a' is a scalar. Let us assume Rx is a register that contain this starting address of the vector X in memory, Ry is a register containing the address of Y, and the scalar register R1 contains the address of the scalar number 'a'. The conventional MIPS32 code to add these numbers will look like this. So, what we do? We load the scalar number into F0, this Rx contains the starting address of X there total 64 numbers double precision.

So, there will be 512 bytes because each number will be 8 bytes long. So, we are adding 512 to Rx and storing it in R4. So, R4 will be storing the memory location which is immediately after this vector X. Vector X is there, Rx points to the first element and R4

will point to the element just after the last element. So, will be incrementing Rx and we will be comparing with R4, and when becomes equal we stop.

We have saying multiply the scalar F0 with all the elements of the vector V1, and store the result in V2. Then we are loading the vector Y into another register V3 the whole vector, then we are adding V2 and V3 storing the result in V4 and we are storing the vector finally back into Ry. Now you see in this code there were no loops; here these three instructions with no loop overheads. So, vector processor has one advantage that it reduces the dynamic instruction bandwidth.

(Refer Slide Time: 20:01)



Dynamic instruction bandwidth is equal to the number of instructions that are actually getting executed.

Let us look at the MIPS32 code version, there were 8 instructions in the loop, and the loop was going 64 times. So, $8 * 64 = 512$, and outside the loop there are two more instructions – total of 514. So, earlier 514 instructions where fetched and executed now only 6 instructions will be fetched and executed. This is a great reduction in the instruction bandwidth. Similarly the stalls the pipeline interlocks are also greatly reduced because in the original MIPS32 version you will see there will be lot of dependencies, but here within the vector instructions there are no dependencies, but across there can be. Like here you will loading V1, next instruction V1 is used like that. So, the vector processor instruction the stalls are required once per vector operation; that means,

instruction rather than once per vector element. In the original version stalls were occurring in every iteration; that means, once per vector element here within a vector instruction there are no stalls, but between vector instructions that can be stalls.

In general the pipeline stall frequency is also reduced by almost 64 times, these are the main advantages. Now they something called vector start up and initiation rate.

(Refer Slide Time: 22:59)

Vector Start-up and Initiation Rate

- The running time of each vector operation in the vector processor has two components:
 - a) **Start-up Time:** Arises due to the pipeline latency of the vector operation.
 - Mainly determined by the depth of the pipeline.
 - A latency of 8 clock cycles means that the operation takes 8 clock cycles, and also there are 8 stages in the pipeline.
 - b) **Initiation Rate:** Time per result once the vector instruction is running.
 - Usually 1 per clock cycle for individual operations.
- The total time to complete a vector operation of length n ($n \leq 64$) is:
$$\text{Start-up Time} + (n \times \text{Initiation Rate})$$

At the bottom of the slide, there are logos for IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Rourkela, along with a portrait of a man.

You see the running time for the vector operations has two components; one is called the start up time. It means after how much time the first result will be available, this depends on the depth of the pipeline. Because of the pipeline latency this is determined by the depth of the pipeline. So, if I say that my start up time is 8 this will mean that my pipeline depth is 8, and it will take 8 clock cycles for the first result to come out. And initiation rate is that once the vectors are being operated on what is the delay with which I can feed the successive elements of the vectors.

This is usually one per clock cycle because the execution units are deeply pipelined. In general if you have a vector operation that is operating on n elements, typically n is less than or equal to 64, then it will be equal to the start up time plus n TSo, will be start up time plus n roughly, this will be the total time to compute the operation.

(Refer Slide Time: 24:00)

Example 2

- Suppose the start-up time of vector multiply operation is 12 clock cycles. After start-up, the initiation rate is one per clock cycle. What will be the number of clock cycles required per result for a 64-element vector?

Solution:

- Clock cycles per result = Total Time / Vector Length
= $(12 + 64 * 1) / 64 = 1.19$

The slide footer includes logos for IIT Kharagpur, NPTEL, and NIT Warangal, along with a video player showing a person speaking.

Let us take a simple example let us say this start up time of a multiply operations 12 clock cycles, which means it is a 12 stage pipeline. So, after starting up initiation rate is one per clock cycle. So, we had trying to estimate what will be the number of clock cycles required per result for a 64-element vector.

You see for all the 64 elements you can use the previous formula to compute the total time, start up time plus n multiplied by initiation rate. Now I want time per element per result, it is 1.19. So, in 1.19 cycles we are getting one result for the vector operation.

(Refer Slide Time: 25:00)

Factors Affecting Start-up and Initiation Rates

- For register-register operations,
 - The start-up time (in clock cycles) will be equal to the depth of the functional unit pipeline.
 - The initiation rate is determined by how often a new set of operands can be fed to the functional unit (usually, 1).
- Typical depths of the functional units:
 - FP addition / subtraction: 6 stages.
 - FP multiply: 7 stages.
- If a vector computation depends on an uncompleted computation, stall cycles need to be inserted → **extra 4 cycles start-up penalty**.

The slide footer includes logos for IIT Kharagpur, NPTEL, and NIT Warangal, along with a video player showing a person speaking.

But there are certain factors which affect this start time and initiation rates, like for register to register operations the start up time this I have mentioned already will be equal to the number of stages in the pipeline. And initiation rate is typically one; this is determined by how often we can feed a new set of data to the pipeline.

But if there is a dependency between vector operations; that means, if a computation depends on an uncompleted computation due to a previous instruction, then some stall cycles may need to be inserted.

(Refer Slide Time: 26:10)

- Independent vector operations using different functional units can proceed without any penalty or delay.
MULTV V1, V2, V3
ADDV V5, V2, V4
- For the vector processor, we define the *sustained rate* as the time per element for a collection of related vector operations.
 - Will be typically greater than 1, due to start-up costs.

The slide footer features logos for IIT Kharagpur, NPTEL, and the National Institute of Technology, Tiruchirappalli, along with a portrait of a man.

So, let us take an example like this where the operands are all independent, this operation V2, V3 stores in V1 this V2, V4 stores in V5. So, this no dependency, for such cases you can proceed without any penalty or delay, but if there is a dependency then in the previous slide I mentioned some extra cycle startup penalty is there, that penalty have to be given for the second instruction. Even with forwarding we have to wait for this operation to be complete then you can forward it here.

So, sustained rate is defined as across a number of operations what is the time per element for a collection of operations; let us take an example.

(Refer Slide Time: 27:00)

Example 3

- For vector operands of length 64, consider the following vector instructions:
MULTV V1, V2, V3
ADDV V7, V4, V5
 - For the MULT instruction,
 - Starting time = 0
 - Completion time = $7 + 64 = 71$
 - For the ADDV instruction,
 - Starting time = 1
 - Completion time = $1 + 6 + 64 = 71$
 - Sustained rate :: 128 FLOPS in 71 cycles = 1.8 FLOPS/cycle

IIT Kharagpur | NPTEL Online Certification Courses | National Institute of Technology, Rourkela

Let us say there are two instructions like this multiply followed by add when the operands of length 64. For the multiply instruction the starting time is 0 because in the clock cycle 0 it is starting. So, completion time will be $7 + 64 * 1 = 71$, and add will start after one clock cycle in a pipeline. So, its starting time will be 1. So, completion time will be $1 + 6 + 64 * 1 = 71$. So, in total 71 clock cycles we are completing $64 + 64 = 128$ floating point operations.

(Refer Slide Time: 27:54)

Overheads for Load/Store Unit

- This is significantly more complicated for vector processors.
- LOAD operation:
 - Start-up time is the time to get the first word from memory into a register.
 - If the rest of the vector can be transferred without stalling, the vector initiation rate will be equal to the rate at which new words are fetched or stored.
 - High-order memory interleaving is used.

The diagram illustrates the architecture of a vector processor. It starts with a **CPU** block, which has bidirectional arrows connecting to a **MUX / DEMUX** block. From the **MUX / DEMUX** block, arrows point to a series of **Registers** (represented by four small boxes). Finally, arrows point from the registers to a stack of **Interleaved memory modules** (represented by three large boxes). A bracket on the right side of the registers and memory modules is labeled "Interleaved memory modules".

IIT Kharagpur | NPTEL Online Certification Courses | National Institute of Technology, Rourkela

So, 128 floating point operation seventy one cycle means 1.8 flops per cycle this is the sustained rate. Talking about the overheads of load store unit we just mentioned it earlier.

Well we have the CPU we have the memory here. Here the memory has to be very heavily interleaved. You can access these banks in parallel, you can store them temporarily in registers, and through a multiplexer and demultiplexer network these registers can be read or written at a very fast rate to the CPU from the CPU. So, for load operation this start up time will be the time to get the first word from memory into register; that means, it will be equal to the access time of the memory. How much time is required to get the data here and the multiplexer to forward the first data, and since the remaining data are already in register, you can feed them one by one at successive clock. So, vector initiation rate will be equal to the rate at which new words can be fetched. This memory organization has to be done very carefully such that very high data transfer rate can be sustained.

(Refer Slide Time: 29:04)

Operation	Start-up Penalty
Vector add / subtract	6
Vector multiply	7
Vector divide	20
Vector load	12

For store operation here start up time is not that important, because store does not produce results in some vector register; it is storing into memory. But if there is a load instruction that follows this store and it uses the same data, then the load may have to wait. These are the typical start up penalties: for add and multiply they are 6 and 7, divide it is typically 20, and for load and store it is typically 12.

(Refer Slide Time: 29:39)

Other Vector Processing Concepts

- Vector Length Register
 - Specifies the length of any vector operation.
- Loading and storing vectors with strides
 - Vector elements are stored in memory with uniform spacing between elements.
 - Adjacent elements of a vector are not sequential in memory.
- Strip Mining
 - How to split loops if the original loop handles vectors that are larger than that supported by the hardware?

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

ANother vector processing concept I am very briefly mentioning, well sometimes you may have a situation where you do not have to operate on the whole 64 element vector, may be we have a vector of size 30, then there is register called vector length register. You can load it with 30, only the first 30 elements will be operated on.

Second one is loading and storing vectors with strides. This is something like this, see normally when you load a vector from memory you can say that I will be loading them from consecutive memory locations, other thing is that you can specify something called a stride or a gap let us say the gap is 10. So, the first vector element is loaded from say memory location 0, the second location second is loaded from 10 third is loaded from 20, 30, 40; that means at gaps of 10. This is sometimes used for example, if you have a two dimensional array, they are typically stored in memory in row major or column major order. Suppose you want to load a row or a column of the vector in a register.

If we used stride you can do both. For column if it is stored in row major order, if we have a gap you can access the elements in a column by specifying the stride you can also load the column in a vector register.

And the third is called strip mining, which says that well your vector register is of length 64, but suppose my original program that I am running it means it is running for let us say 200 cycles. So, what do we do? So, 64, 64, 64 I can do it for three times. So, it will be 192. So, 8 vector elements will be left. This is called strip mining. I will be factoring

it out $64 + 64 + 64$ and whatever is left out that I will be using as a separate 4th vector operation.

These are some concepts that are there along with vector processing, which helps a programmer to write vector programs. With this we come to the end of this lecture. I tried to give you a brief overview about what vector processing is and how vector computers work. If you look at a real vector processor that is available commercially, you will see there are lot of other details involved there, but here we are not going into those details.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 62
Multi – Core Processors

We have seen so far, how we can make our processor faster by using concepts of pipelining, superscalar, VLIW, these kind of concepts. Now these are become very standard concepts nowadays. Whenever you buy a computer system with an embedded processor, you will see invariably these technologies are built into that as a standard feature. Now the question comes if you look at the modern processors that goes inside our desktops and laptops and also our mobile phones today, you have heard about multi core processors.

So, what it is? Multi-core essentially means -- we have the capability of fabricating large systems on a VLSI chip, we are fabricating more than one processors inside the chip, that is very roughly the concept of multi-core. So, the question arises why we are actually going for multi-core, let us try to address these issues in this lecture.

(Refer Slide Time: 01:45)

Introduction

- Multi-Core Processor:
 - A processing system composed of two or more independent cores or CPUs.
 - The cores are typically integrated onto a single integrated circuit die, or they may be integrated on multiple dies in a single-chip package.
- Cores share memory:
 - In modern multi-core systems, typically the L1 and L2 cache are private to each core, while the L3 cache is shared among the cores.
- In symmetric multi-core systems, all the cores are identical.
 - Example: multi-core processors used in computer systems.
- In asymmetric multi-core systems, the cores may have different functionalities.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

As I said a multi-core processor is nothing but a processor system, which consists of two or more CPUs. CPUs are typically independent -- they are not strongly connected or they are not dependent on each other. They can independently execute programs.

So, for a multi-core processing system we have two or more independent cores or CPUs. Now these cores come inside a single package, you can buy a core I3, I5, I7 processor from the market. So, inside the same package there will be multiple cores: 2 or 4 or 6 whatever. Now these cores or processors are all fabricated inside the same die within the chip. There are two alternatives. You can either have a single die, die means a silicon where the circuits are fabricated, or in modern-day VLSI packages, you can have multiple dies that are connected or bonded together inside the same package.

Now if you look at the modern processors, the L1 and L2 caches are private to each of the cores, however, the L3 cache is common and is shared by all the cores. These are symmetric systems where all the cores are identical -- like the core I3, I5, I7 multi-core processors are called symmetric, they are all identical. But in asymmetric multi-core systems the cores may have different functionalities. Well you think of the multi-core system that goes inside your mobile phones.

Now, you have heard of quad core, octa core processors, there are 8 cores, but there is no reason to believe that all those 8 cores are identical, they have some specific purposes. Some of them may be specific DSP cores, some of them may be specific processor cores like ARM, some of them may mean doing some specific functionalities.

(Refer Slide Time: 04:57)

Why Multi-core?

- It is difficult to sustain Moore's law and at the same time meet performance demands of various applications.
 - Difficult to increase clock frequency, mainly due to power consumption issues.
- Possible solution:
 - Replicate hardware and run them at a lower clock rate to reduce power consumption.
 - 1 core running at 3 GHz has the same performance as 2 cores running at 1.5 GHz, with lower power consumption.

IIT KHARAGPUR | **NPTEL ONLINE CERTIFICATION COURSES** | **NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA**

They may be running at different clock frequencies, different instruction set and different capabilities that is called asymmetric.

So, why we go for multi-core? Now you see you have earlier heard about Moore's law. Moores law says that the number of transistors or gates that you can put inside a single chip will grow exponential with time -- will double approximately every 18 months or so. Now Moore's law somehow has been sustained till now, but regarding the performance demands there have been some constraints like although you can potentially increase the frequency of the clock, but you are unable to do so, why? Because your chip power consumption directly depends on the clock frequency and if you increase it your power consumption can become unmanageably high and your chip might be burnt out.

So, you will have to limit your clock frequency within a manageable level, simply to tackle the power consumption issue not due to anything else. There are two possible solutions you can think of, the first solution can be to replicate hardware. You use two computer systems that are running side by side and run them at a lower clock; like for example, just compare a core that is running a 3 GHz clock, and two cores running at 1.5 GHz clock, both of this will be having the same performance, but this second done will be having much lower power consumption. Because power increases in a super linear fashion with increase in the clock frequency. That is why you will see that in the modern generation of processors although the number of transistors is increasing number of cores are increasing, over the last ten years or so, the clock frequency has not increased much.

(Refer Slide Time: 07:22)

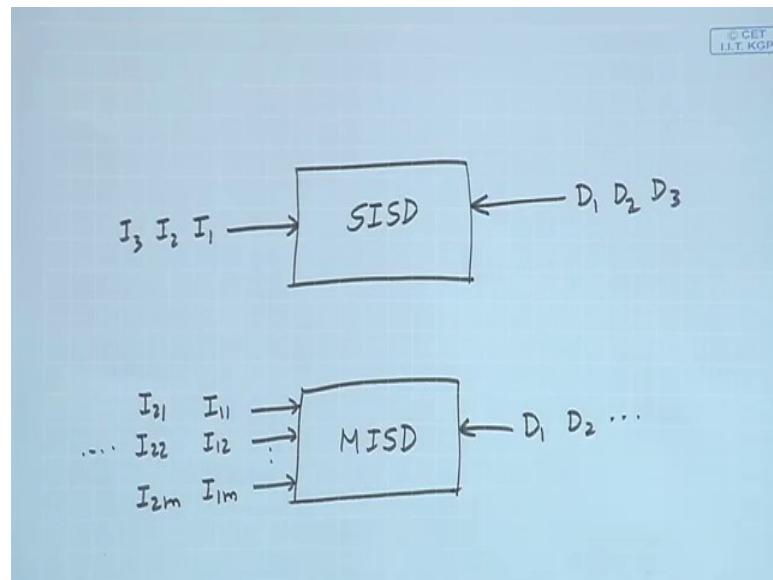
The slide has a dark blue header and footer. The main content area has a light yellow background. The title 'Taxonomy of Parallel Architectures' is centered in bold black font. Below the title is a bulleted list of parallel processing models:

- Single instruction-stream single data-stream (SISD)
 - Traditional uniprocessor systems.
- Multiple instruction-stream single data-stream (MISD)
 - No commercial implementation exists.
 - Pipelining can be argued as a type of MISD processing.
- Single instruction-stream multiple data-stream (SIMD)
 - Array and vector processors.
- Multiple instruction-stream multiple data-stream (MIMD)
 - Multiprocessor systems (various architectures exist).

At the bottom, there are three logos with their respective names: IIT Kharagpur, NPTEL, and National Institute of Technology Meghalaya.

Talking about the parallel architectures, a very rough classification is sometimes called Flynn's classification, it goes like this. We categorize depending on the instruction streams and data streams. First is representative of the traditional single processor or uniprocessor systems.

(Refer Slide Time: 07:59)



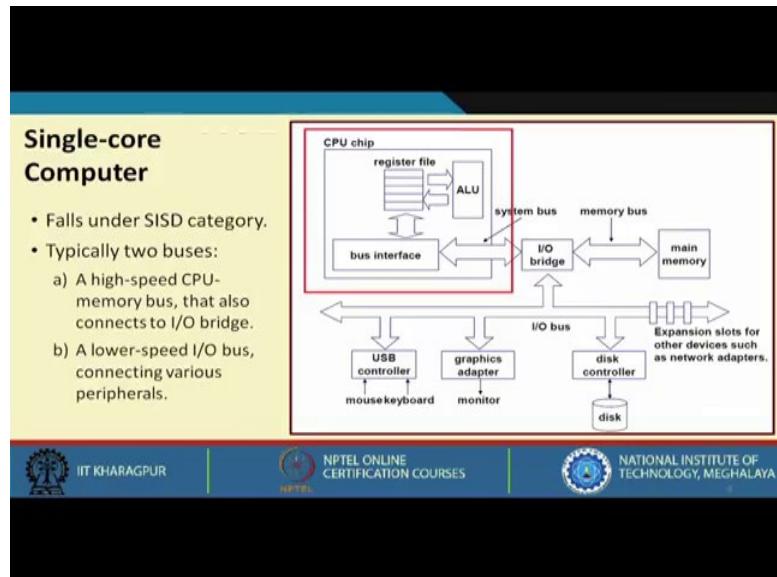
This is called single instruction stream single data stream (SISD). This means I have a computing system where single instruction stream or instructions are coming one by one I_1, I_2, I_3 like this. Data stream mean these instructions are operating on data, data are also coming sequentially one by one D_1, D_2, D_3 . Suppose I_1 is working on D_1 , I_2 is operating on D_2 , I_3 is operating on D_3 like that, this is SISD.

The second classification says multiple instruction stream single data stream (MISD). It says that I am feeding several instructions let us say I_{11}, I_{12}, I_{1m} . So, I feed m instructions together, then I feed I_{21}, I_{22}, I_{2m} and so on. But it says that they are working on single data streams like these m instructions are working on single data. Conceptually speaking it is a little difficult to visualize this kind of computation, where we have a single data, but multiple instructions are working on it. Strictly speaking no real systems are there that can fall under this MISD category.

For single instruction stream multiple data stream (SIMD), is something which you can correlate vector processors that we discussed in our last lecture. Their a single vector

instruction operates on all the elements of a vector, single instruction multiple data this is so called SIMD.

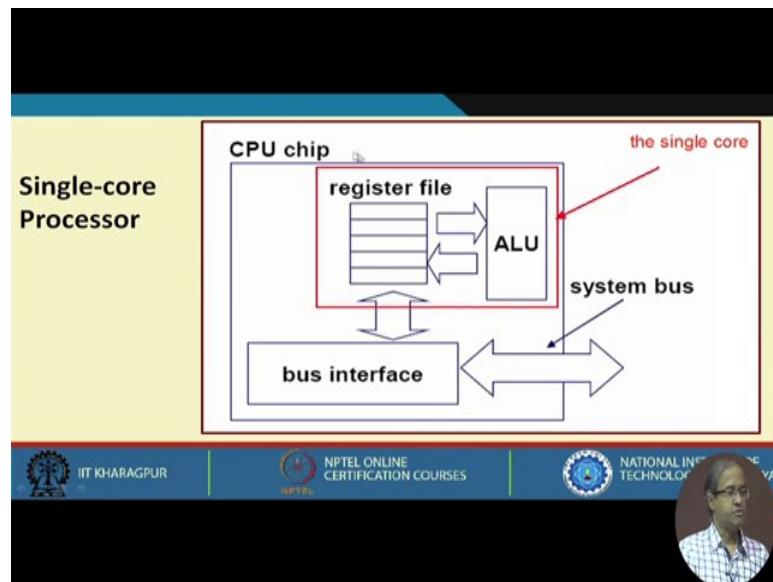
(Refer Slide Time: 12:06)



Let us look at single-core computer, the one that falls on the SISD category. Well the example that we take is similar to what we see inside our desktops, but not the modern desktops but those of earlier years. This is a typical architecture that you used to see, this was the CPU chip that consists of the ALU, registers some other circuits, bus interfaces and there was something called IO Bridge. This IO bridge had two purposes, it interfaces the CPU with the memory system -- this was a very high speed bus, and there was a relatively lower speed IO bus; this IO bridge also interface this IO bus with this so that the IO devices could transfer data to main memory and maybe the CPU.

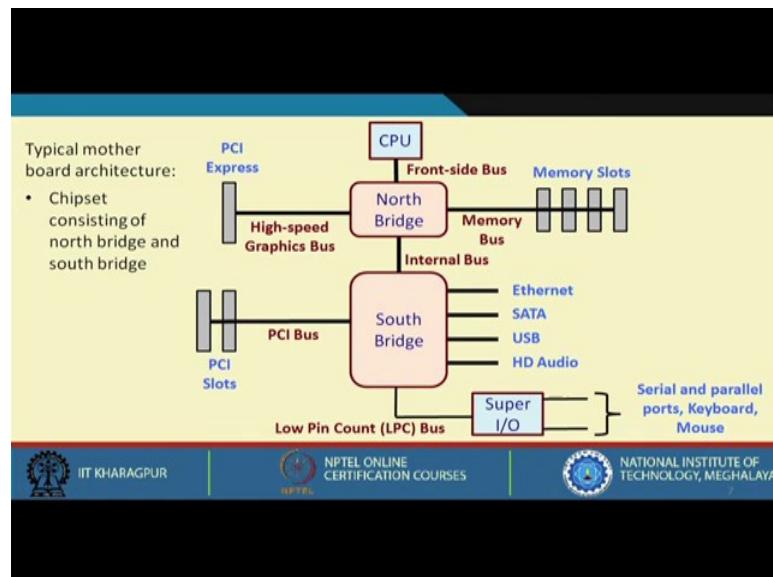
The IO bus had connections to some USB controllers which provider some USB ports through which we can connect a variety of devices like mouse, keyboard many others, graphic adaptor you could connect your display monitor various disk controllers, you can connect to a disk sub systems and there were expansion slots.

(Refer Slide Time: 13:48)



When you talk off a core, every core essentially consist of a register file, ALU and of course some control unit.

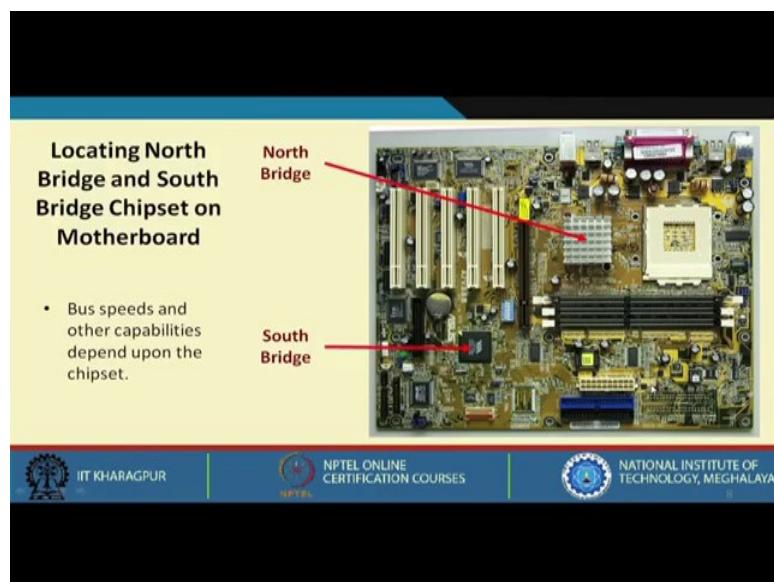
(Refer Slide Time: 14:20)



This is a typical motherboard architecture shown here for a desktop PC. Well the processor is typically connected to a chip that is called the north bridge, this is a very fast switch, what does it interfaces the CPU with the memory bus -- in the memory bus there can be memory slots. In the slots you can plug in memory modules.

And on the other side you have a PCI express bus which is typically used to connect high speed graphics for your monitors and other things, which need very high speed data transfer. And on the other side it is connected to another switch this called south bridge, this is of a relatively much slower speed. South bridge interfaces with so-called PCI bus, which consist of several PCI slots through which you can connect network adaptors and different kind of circuit boards and there are also direct connections available in modern motherboards, which are connections to network interface like Ethernet disk interfaces like SATA, USB, audio interface, and there is another bus which is meant for low speed devices this is called LPC or low pin count bus. This is typically a serial bus working at a much lower speed like serial parallel ports, keyboard, mouse, these kind of devices are connected here.

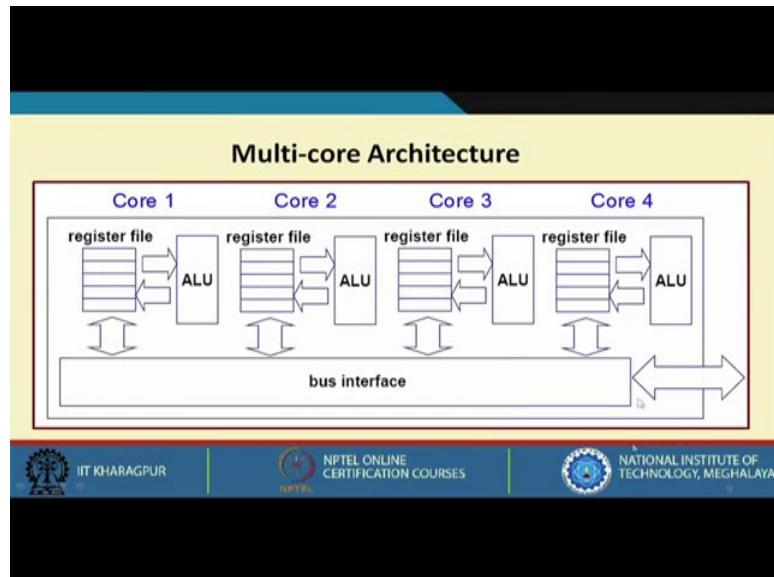
(Refer Slide Time: 16:29)



This is the rough picture of the motherboard of a desktop. I am showing you pictorially one of the motherboards here, you can see this is your processor out here and this part you see this is your north bridge and the fins that you can see this is actually heat sink, this is a very high speed chip, this dissipate slot of power. So, you need a heat sink to dissipate the heat and this small chip out here this is your south bridge, these are the PCI slots these are the memory slots and the other components. You see this is a motherboard which used to be several years back, but if you see the motherboard of one of the modern day PCs, you will see that many of the circuits have been collapsed into a single chip,

there are very few chips and a very small motherboard you will see in the desktops that are available to.

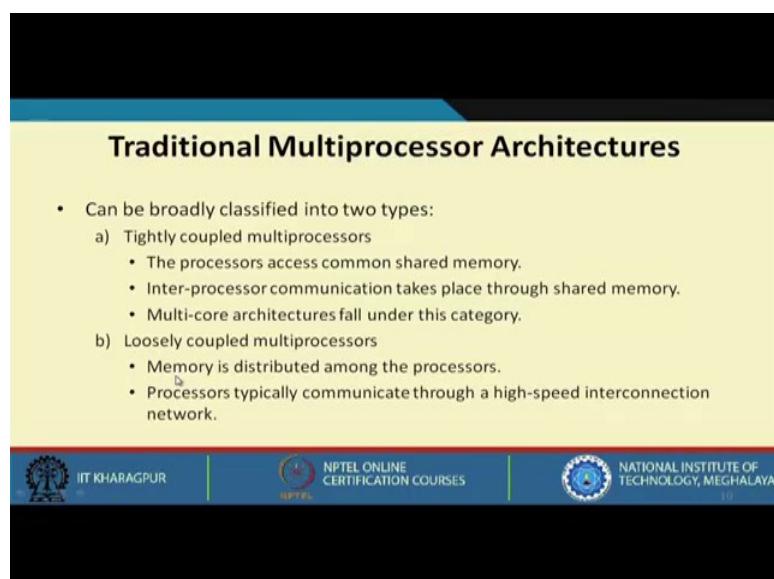
(Refer Slide Time: 17:49)



We looked at one core consisting of a register file, ALU, plus control. Now for multi core you have several such and you have some kind of a fast bus interface using which they can communicate among themselves, and there will be an interface with north bridge with the outside world.

This is a very generic picture, let us look into some more specifics.

(Refer Slide Time: 18:34)

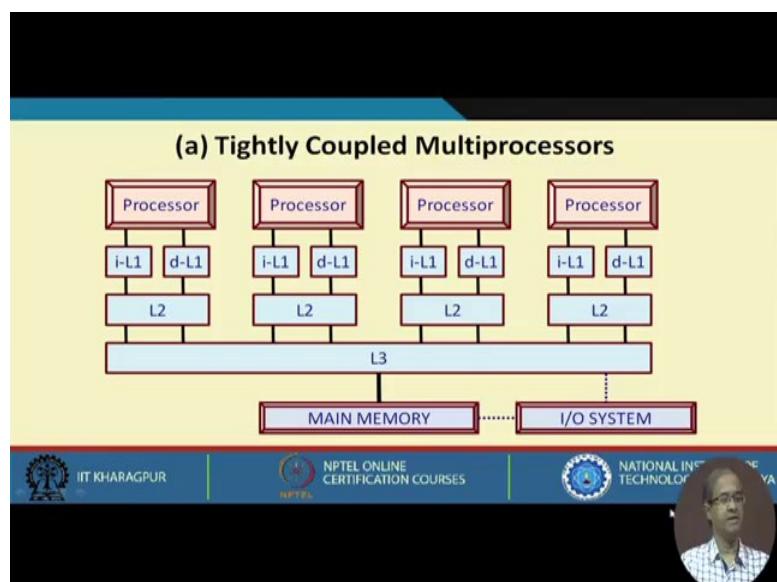


Broadly speaking this kind of multiprocessors can be classified as either tightly coupled or loosely coupled. Tightly coupled means the processors or the cores have access to common shared memory, in other words this processors are all inside the same box. So, inside the same box there are the processors there is also the common shared memory. When you buy a computer system, you buy the system along with all the processors and also memory.

Inter processor communication can take place through shared memory like one of the processor can write some data into the memory, the other processor can read the data from that memory. The standard multi core architectures usually fall under this category, but another kind of multiprocessor architecture is also available, this is called loosely coupled multiprocessors, also called clusters or cluster computing. Here there is no shared memory. Memory is distributed among the processor, each processor has its own memory and communication is not via shared memory, but through some high speed interconnection network.

There are processors, there will be a very high-speed interconnecting switch, the processors will be connecting via that switch. They will be sending and receiving messages from other processors, this is how loosely coupled processors work.

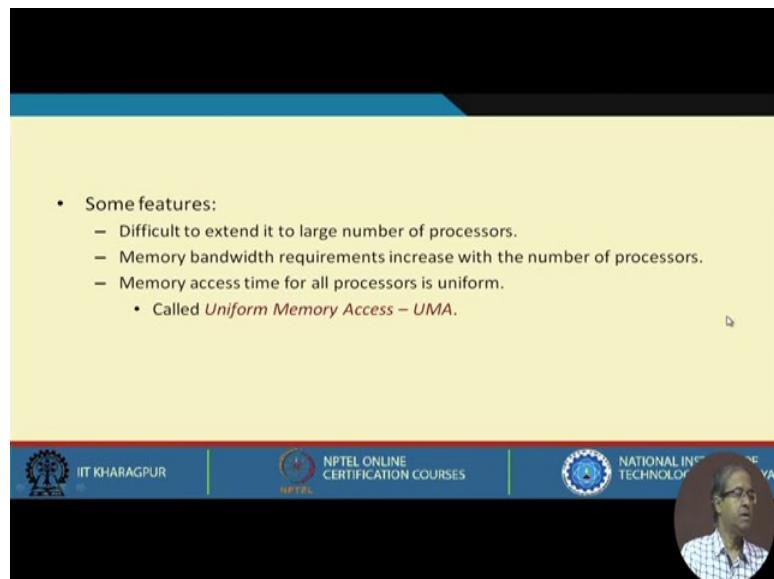
(Refer Slide Time: 20:33)



Now, pictorially a tightly coupled multiprocessor looks like this, this is a very typical diagram. You can see the different cores, now each of the core is having its own

instruction in data cache in level 1 and also level 2. These are all private to the processors, but typically the L3 cache is common across all the processors and of course, the main memory is also common.

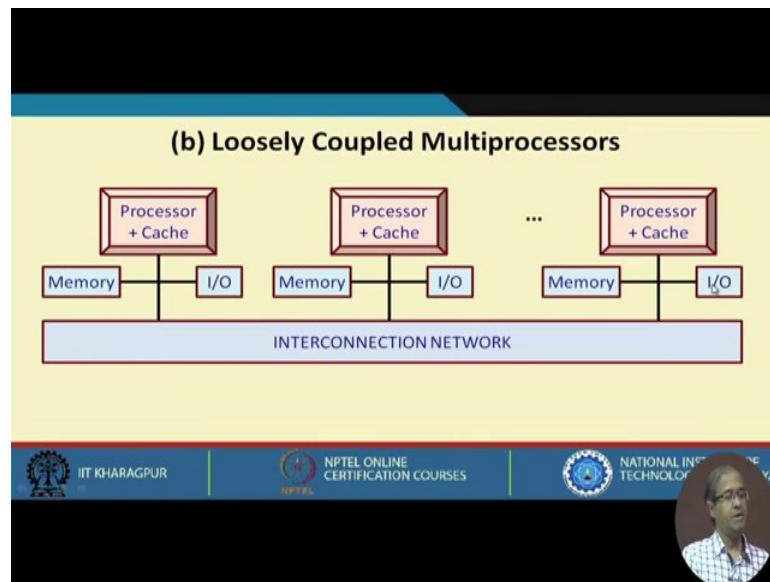
(Refer Slide Time: 21:18)



Because of the constraint here you see the constraint of L3 and main memory so many processors are trying to access them. So, if you increase it to say 8, 16, 32 they will put lot of load on L3 cache and main memory. There is a scalability issue, you cannot very easily extend it to larger number of cores or processors because the memory bandwidth requirement will also increase that are very difficult to handle. Now in this kind of architecture the memory access time for all processors is uniform this is sometimes called uniform memory access or (UMA).

Why it is uniform? Because irrespective of which processor you are, the path to access memory is this same.

(Refer Slide Time: 22:18)



So, the delay is the same -- that is why it is called uniform memory access.

But it contrast loosely coupled multiprocessor look like this, there will be multiple processors with their own private caches. So, all L1, L2, L3 can be inside, they will be having the local memory, they will be connected via an interconnection network. A processor can access its local memory, and also via the interconnection network it can access the memory of the other processors.

You can see the access times will not be uniform, well accessing local memory will be faster, but accessing the other memories will be slower. This is sometimes called non uniform memory access or NUMA.

(Refer Slide Time: 23:11)

- Some features:
 - Cost-effective way to scale memory bandwidth.
 - Communicating data between processors is complex and has higher latency.
 - Memory access time depends on the location of data.
 - Called *Non Uniform Memory Access – NUMA*.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY WARANGAL

This is also a cost effective way to scale memory bandwidth, because you see most of the time when you run a program on a core or processor, it will be accessing its local memory only. Very rarely it will be trying to access the memory of the other processors. So, in this kind of architecture, even if I increase the number of processors, scalability will not be that much of a challenge, the overhead will not go up in a proportional way.

The only flip side is that the communication between the processors is complex using that switching network and has higher latency. When you are mapping some applications to run on them, you try to map in such a way that most of the time

(Refer Slide Time: 24:08)

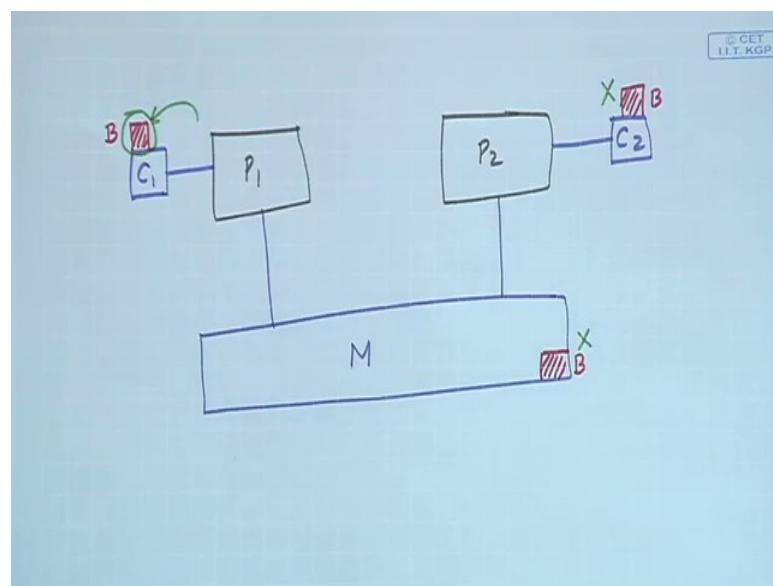
Cache Coherency Problem in Multiprocessors

- Maintaining coherence between data loaded in processor caches is an issue in multiprocessor systems.
 - Same memory block is loaded into two processor caches.
 - One of the processors updates the data in its local cache.
 - Data in the other processor cache and also memory becomes inconsistent.
- Broadly two classes of techniques are used to solve this problem:
 - Snoopy protocols
 - Directory-based protocols

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

they will be accessing the local memory and very rarely there will be accessing the other processors. In multiprocessors there is one important problem, we are not going into the detail of this problem in this lecture because it is a little beyond the scope.

(Refer Slide Time: 24:26)



This cache coherence is a very important problem; I am trying to illustrate this. Suppose I have two processors let us say P₁ and P₂, just assumed they have their own cache memories C₁ let us say C₂.

Let us assume they have access to a common memory M. So, what might happen now you see there can be one block in this memory, the processor might be using this block. So, this block will be loaded in the cache. It is possible that the other processor is also trying to access this, and this block will also be loaded in the cache. Let us say B this is the original block. A copy is here and a copy is here. Now what might happen the processor P1 might write something into this block B that will make this block dirty, but as soon as it writes into B what will happen you can imagine, these two blocks will become invalid. So, whatever is loaded in P2's cache will be stale because this block was recently modified and the copying memory will also be stale.

This is called the cache coherency problem, where there are multiple copies of the block which may be located in several places. One of them may be they good while others may be incoherent. There are many methods and techniques that have been proposed to solve this problem. One is call the snoopy protocol where in the memory bus itself you continuously check whether someone is doing a write into its local cache. If it does you immediately send the information to the other memory controls also so that they can check whether there also having a copy of that same block in their cache. If so, they will immediately invalidate. And directory based protocol is that where you store the information about the blocks in some central place where the memory blocks are stored.

With this I come to the end of this lecture. In this lecture if you recall we tried to give you a broad classification of computer systems with some examples, what kind of processing or computing systems fall under which category and some of the problems and issues. If you are interested to know more detail on this, there are advanced level courses and computer organization architecture available you will get to know about these in those courses.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 63
Some Case Studies

In this lecture, we shall be looking at some of the case studies of some modern day processors that we see around us. So, we shall be basically looking at two such case studies, one is that of a so called graphics processing unit or GPU and the other is will be looking at very briefly the evolution of the Intel class of processors.

(Refer Slide Time: 00:48)



(Refer Slide Time: 00:50)

Introduction

- Graphics Processing Unit (GPU) is a processor optimized for 2D/3D graphics computation, video rendering and visual computing.
- It is a highly parallel, highly multi-threaded multiprocessor optimized for visual computing.
- It provides real-time visual interaction with computed objects via graphics objects like image and video.
- It serves both as a programmable graphics processor and a scalable parallel computing engine.
 - Heterogeneous computing systems combine a GPU with a CPU.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

We start with GPU.

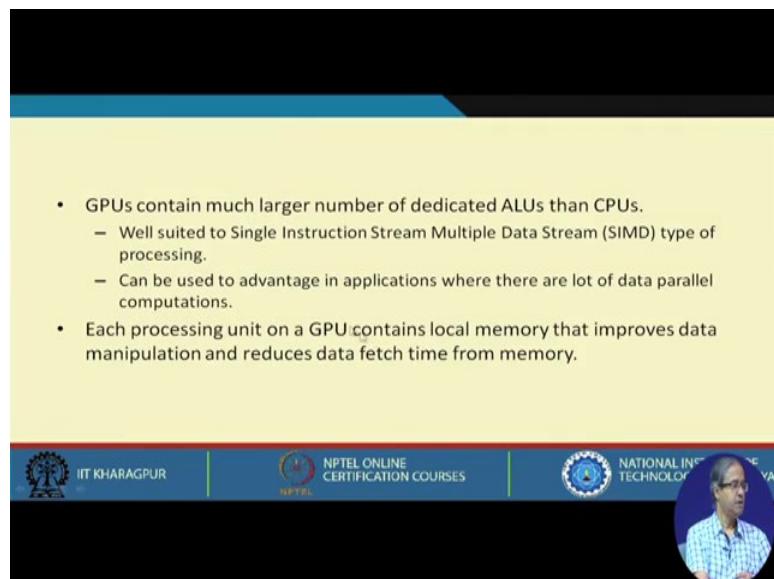
Let us try to explain what a graphics processing unit is. Well traditionally GPU were used as a graphics accelerator to a processor. Over the years, the requirements and the demand of graphical processing video multimedia has increased. So, the processor with the help of its own instructions, finds it very difficult to handle all the graphics processing tasks. The trend is that a separate chip or a processor is used which is dedicated for all the graphics multimedia animation kind of applications that has come to be known as GPU or graphics processing unit.

So, GPU is a processor that is optimized for graphics computations. These processors have specific features that help in video processing, visual computing, etc. Essentially a GPU is a highly parallel multi-threaded processor. GPU is like a very highly parallel computer architecture which resembles the SIMD kind of processing where there are number of simple processors which are under control of a single control unit. With the help of very specific instructions, it can provide real time rendering of videos and similar applications.

Now, the interesting thing is that because of its inherent architecture, it is a highly parallel SIMD machine. So, why use it only for graphics, we can also use it as a scalable parallel computing engine to execute some applications; very high-level parallelism is there. In general, we can have a heterogeneous computing system where the parts of the

computation that can be parallelized can run on GPU while the other part can run on the conventional CPU. So, you will see such computer systems available today where CPU and GPU are residing inside the same box where the GPU is not looking after the graphics, but it acts as a parallel co-processor to the main CPU.

(Refer Slide Time: 03:53)



Some of the characteristics of GPU is based on the SIMD mode of computation for processing. There are large numbers of arithmetic logic units, in the order of 1000s. So, in a normal CPU, we talk about 2 core, 4 core or 6 core, but here, we are talking about 1000s. Such kind of architecture can be used in application where you are processing on large arrays or vectors of data and there is highly parallelism or parallel mode of computations available in the application. Each processing unit also contains some small local memory so that some local data can be loaded and processing can be carried out. Of course, there has to be a more sophisticated load store unit. From the memory, you may have to load a vector or array of data, similarly you may have to store a vector or array of data.

(Refer Slide Time: 05:16)

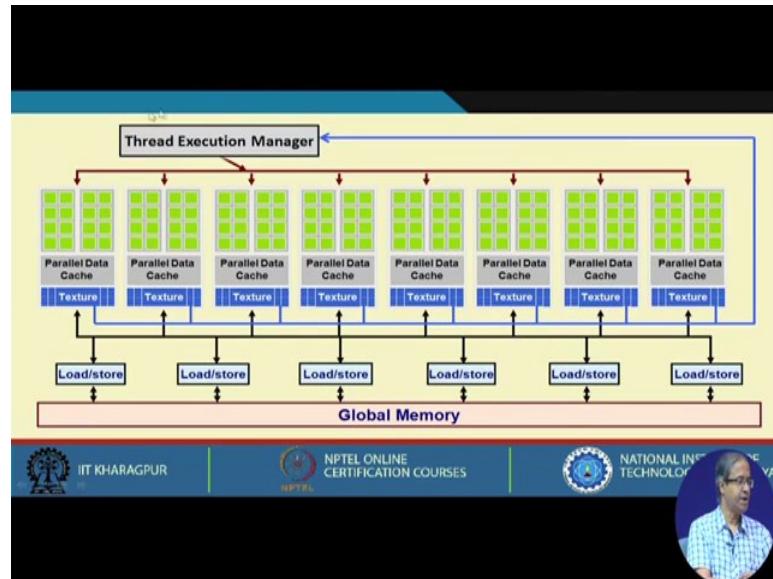
- Hybrid systems combine CPU and GPU.
 - For programs that have one or very few threads, CPUs achieve better performance than GPUs as they have lower operation latencies.
 - For programs having a large number of threads, GPUs with higher execution throughput can achieve much higher performance than CPUs.
 - Many applications use both, executing the sequential parts on the CPU, and numerically intensive parts on the GPU.
- Modern GPUs are massively parallel with more than 100 cores, supporting 1000s of threads.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, BOMBAY

For applications where the number of threads are very less that does not use multithreading CPUs work better because CPUs with a few threads are much more efficient, their instruction sets are much more powerful, their operation latency is also less.

But if you think of programs that have thousands of threads; running them on a CPU may not be that efficient. Here GPUs come into the picture, and can provide very high throughput for multithreading kind of application where the number of threads are very large. There are applications that use both kinds of computing: the sequential parts can be running on the conventional CPU, and the numerically intensive parts that are heavily multithreaded can run on GPUs.

(Refer Slide Time: 06:51)



I am showing you a typical architecture of a GPU. The processors are divided into clusters; they are not placed together, they are placed in clusters. In this example, you can see each cluster is having $8 + 8 = 16$ cores, and there are 8 such. There are a total of 128 cores. Each of these clusters is having data cache from where data can be loaded and stored, and there is a bus that connects these clusters to load store units. Now to increase the bandwidth between the memory and these cores, there can be multiple load store units that can run in parallel. Here I am showing 6 load store units.

There is a thread execution manager -- this is more into the software part I am going. The threads can be scheduled on these clusters and inside the cluster, there will be a simple operating system kind of a support where depending on the available cores, a thread will be running on one of the cores. Also, as a result of computation, the thread manager can be informed and thread manager can act accordingly. We have a highly parallel computing system where these so-called cores can be very simple as compared to a normal processor core, but suppose there are 128 cores, I can potentially carryout 128 multiplications at the same time, I can carry out 128 additions at the same time.

(Refer Slide Time: 09:00).

The slide has a dark blue header and footer. The main content area is yellow. The title 'Does x86 chips use microprogramming?' is in bold black font. Below it is a bulleted list:

- The dilemma:
 - RISC architecture supposedly execute instructions faster than CISC.
 - RISC architecture can be efficiently implemented using hardwired control.
 - CISC architecture may prefer microprogramming because of complex instructions.
- So what is actually done in a CISC processor like x86?
 - A combination of microprogramming and hardwired control.

At the bottom, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology Meghalaya.

Let us digress a little bit. Consider the x86 family of chips that are based on the complex instruction set or CISC architecture; do they use microprogramming because there is a dilemma that I talked about earlier. Towards the beginning of the class, you have seen that the RISC architectures provide you with an implementation that on the average runs faster than an equivalent CISC architecture platform. A program will run faster on a RISC architecture as compared to a CISC architecture.

Now, the question is because of legacy reasons and backward compatibility the Intel series of processors that are the most widely used in the world today, they have to be stuck with this CISC kind of architecture. So, are they compromising on poorer performance this is the question. The dilemma that I am talking about is that RISC architectures execute instruction faster than CISC, but RISC architecture can be efficiently implemented using hardwired control, but in a CISC architecture because of the fact that some of the instructions are very complex, if you want to build a pipeline directly from there, the pipeline may become quite complex. So, there can be a compromise you can have make.

You can use microprogramming to break the complex instructions into simpler instructions, and at the next level you can have a RISC kind of an architecture that will be executing the simpler instructions using a combination of micro programming and hardwired control.

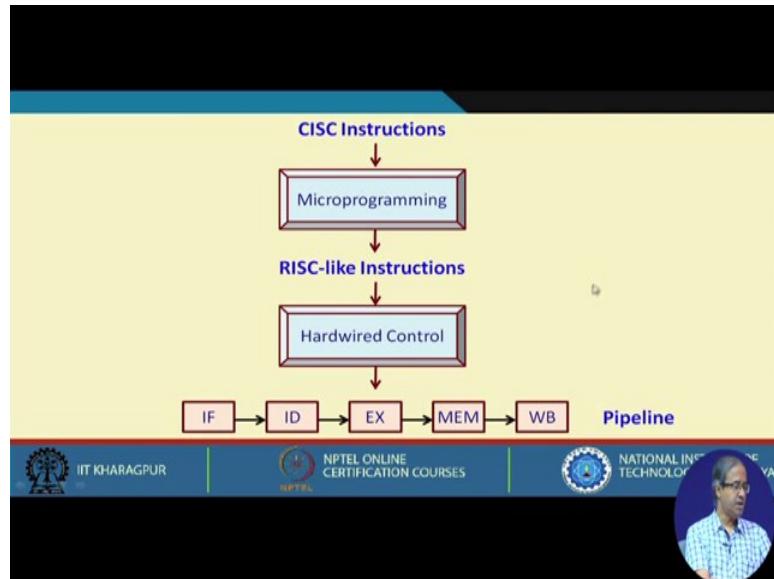
(Refer Slide Time: 11:22)

- Intel chips are CISC based, which use microprogramming to break the complex instructions into simpler sub-operations.
- The sub-operations are very similar to RISC instructions.
- So, at the instruction level the processor can be considered to be using microprogramming.
- At the lower (hardware) level, it may be considered to be using hardwired control to execute the RISC-like instructions in a pipeline.

The Intel processor chips there based on CISC instructions, they use microprogramming to break the complex instructions into simpler sub-operations, but here we note that all instructions need not be broken, there may be some instructions already like RISC, they need not have to go through this.

These sub-operations are very similar to RISC instructions. At the high level, you can regard that we are using microprogramming, and break up into this RISC kind of instructions. At the lower level, you may consider that we are using hardwired control to execute those instructions in a very efficient and compact pipeline.

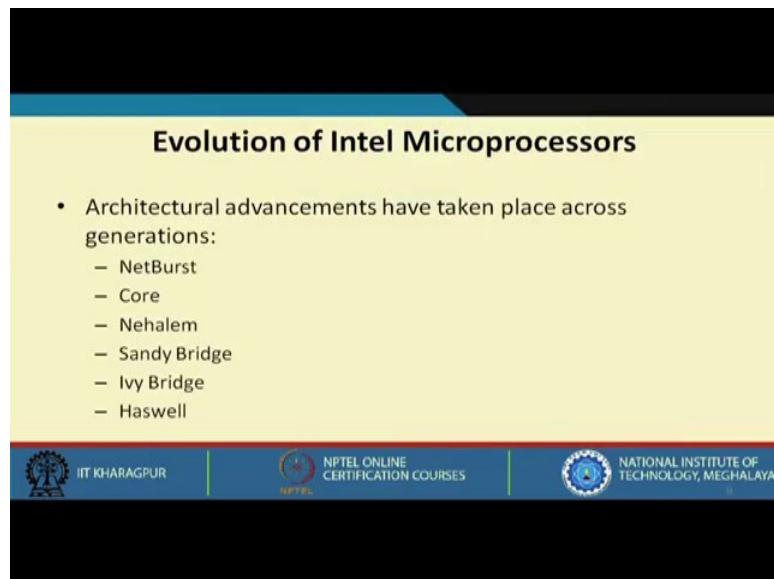
(Refer Slide Time: 12:28)



Pictorially it can look like this. You have this CISC instructions at the top level, then you have microprogramming which will be translating them into simple operations that are RISC like instructions, which in turn will be executed on a pipeline, here I am showing the pipeline of MIPS32.

This is how instructions are typically executed in a modern day Intel processor.

(Refer Slide Time: 13:10)



Now, I shall be looking at the evolution of the Intel processors. Of course, we shall not be going into the details because it may involve a lot of advanced topics that we have not

covered in the class. I shall be trying to give an overview of how this evolution has taken place and what are these different families. You have been hearing that Intel processor families have gone through so many generations Nehalem, Ivy bridge, Sandy bridge.

The point is that over the years, there have been architectural advancements that have taken place in the Intel processor families. Some of the earlier architectures were called Netburst that was not so popularly known, then core; you have seen the core2duo kind of processors, you must have heard of it, then Nehalem, Sandy bridge, Ivy bridge, Haswell -- these are some names you may have heard. Let us look briefly what are the features.

(Refer Slide Time: 14:29)

(a) Netburst Architecture

- Hyperthreading:
 - Single processor appears to be two logical processors.
 - Each logical processor has its own set of registers.
 - Increases resource utilization and improve performance.
- Rapid Execution Engine:
 - ALUs run at twice the processor frequency.
 - Basic integer operations execute in $\frac{1}{2}$ processor clock tick.
 - Provides higher throughput and reduced latency of execution.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY BOMBAY

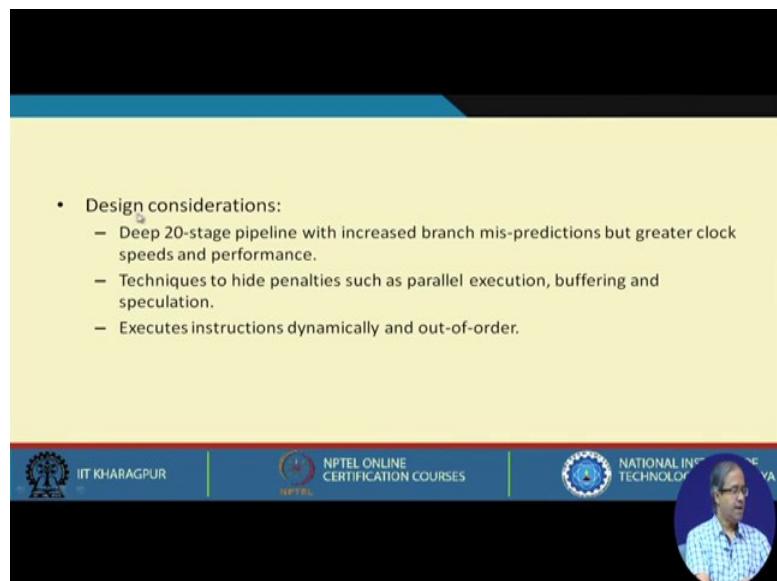
The first of these architectures is the Netburst architecture. The first thing is that it used hyperthreading. In hyperthreading, users are provided with some kind of virtualization where a single processor appears to be 2 or more logical processors. As if 2 threads are running on the same processor. To the threads the processor seems to be a separate dedicated virtual processor. Because you have 2 threads you can say that there are 2 virtual processors that run those 2 threads. Each of these logical processors had their own sets of registers, which means we had 2 different register sets one for thread 1 and one for thread 2.

Now, by doing this resource utilization, performance can be improved to a great extent because you see, but whenever you use multithreading with a single set of registers, when you switch from one thread to the other; you may have to save some registers and

restore the new registers, but here because there are separate set of registers; thread switching is very fast. You simply switch to the other register set. There was another concept here that was called the rapid execution engine, where ALUs were running faster than the processor. The ALU clock was running at twice the frequency as compared to the processor clock.

The basic integer operations are executing in half the processor clock tick. If you correlate with the MIPS32 architecture, for multi-cycle operations that are there in the EX stage, you are trying to reduce that time to half. This will result in higher throughput and of course, reduce latency of execution.

(Refer Slide Time: 16:53)



There was not much consideration on reducing the number stages in the pipeline to convert it into a RISC kind of architecture. So, it had a very complex pipeline comprising of 20 stages where the complex instructions were directly mapped and executed.

For branch instructions because of the deep pipeline branch mis-predictions were pretty high, but because of the simpler stages, the clock frequency could be made very high and performance could be made reasonably on the higher side and various techniques to hide the various kinds of penalties in the pipeline were also implemented.

And by special hardware, control instructions were executed dynamically and also out of order.

(Refer Slide Time: 18:50)

(b) Core Architecture

- Multiple cores and hardware virtualization.
- 14-stage pipeline (less deeper than Netburst).
- Dual-core design with linked L1 cache and shared L2 cache.
- Macrofusion: Two program instructions can be executed as one micro-operation.
- Intel Intelligent Power Capability: Manages run-time power consumption of the execution cores.
- Includes advanced power gating: turns on individual processor logic subsystems only if they are needed.
- Prefetching unit is extended to handle separately hardware prefetching by each core.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY ROURKELA

Next came the so-called core architecture. The multi-core architecture came from this family. Here you had multiple cores and again hardware visualization by using multithreading. Here the pipelines are made simpler; instead of 20 stage in Netburst it became 14 stages and dual core design. The first version used 2 cores, and they had dedicated L1 cache and shared L2 cache.

L3 cache was not there in this family. There was a concept of macro fusion where 2 program instructions logically can be executed as one micro-operation, one in the first core, other in the second core. It also started to use some intelligent power management capability, where runtime power consumption of the execution core was measured and accordingly, it was controlled.

Then advanced power gating was used for low power. Some individual processor logic subsystems were turned on and off depending on whether they are required or not, and there was a separate pre-fetching unit that was extended to support instruction fetching by 2 cores in parallel.

(Refer Slide Time: 20:31)

(c) Nehalem Architecture

- Family of processors introduced:
 - Core i7 processors for business and high-end consumer markets.
 - Core i5 processors for mainstream consumer markets.
 - Core i3 processor for the entry-level consumer market.
- Features of Nehalem:
 - Integrated memory controller.
 - Advanced configuration and power states.
 - Improvements to the pipeline (L2 Branch Predictor, L2 TLB, etc.).
 - Three-level cache.
 - Hyper-threading support.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY WARANGAL

Then came the Nehalem architecture which is the beginning of the kind of cores that we see today, i3 i5 i7; these have been very commonly used terminologies towards today. So, core i7 was primarily meant for business and high consumer markets. Today, however, we want these i7 processors to be used inside our laptops also for daily use.

The cost has gone down, power consumption has also gone down. i5 was meant for mainstream consumer market, and i3 for entry level market and some of the features of Nehalem which started to appear was that the memory controller was integrated within the chip and the power management was made more sophisticated. There are multiple power states like high power, medium power, low power. The operating system can initialize the power state of the processor and accordingly run programs on that power state. So, depending on that, you can conserve battery or you can run your application faster whatever you want. And there are several improvements made to the pipeline using branch predictors, sophisticated translation look aside buffers, etc. and here the third level of cache L3 was also introduced, and of course hyper threading support was there.

(Refer Slide Time: 22:11)

- Design considerations:
 - Hyper-threading is reintroduced to cater to increasing number of thread based applications.
 - Cores are placed on a single die to reduce latencies.
 - L1 and L2 caches are private to each core, with a large shared L3 cache.

Some of the design considerations here was that hyperthreading was again introduced in this family because there were several applications which demanded larger number of threads that started to appear because of that; this hyperthreading concept was again introduced here and all the cores they were placed on the same integrated circuit die, it was a single chip and the first 2 levels of the caches were private to the core while the L3 cache was shared among the core.

Here the number of cores were either 2 or 4.

(Refer Slide Time: 22:55)

- Some features:
 - Intel Advanced Vector Extensions (AVX)
 - Integrated graphics unit on the same die
 - Next generation Intel Turbo Boost technology
 - High bandwidth and low latency modular on-die Ring Interconnect
 - Integrated memory controller

Then let us have a brief look at the Sandy bridge architecture that came after that. Here some vector extensions were made to the processor, some features that were earlier present in vector or array processors, because of application demands some of these were integrated in the processor itself there was a separate engine which was called AVX advanced vector extension.

Secondly the graphics unit the GPU was also integrated on the same die. There was a feature called turbo boost technology that also appeared. Here the system can automatically check the temperature of the chip, and if it finds that the temperature is not too high, then at least for a few seconds it can go in a turbo execution mode where the frequency will be increased, of course, heat generated will also be increased and you can run your programs much faster.

So, for a few seconds you can move to a turbo boost mode and you can execute some applications very fast, but you cannot do it in a sustained manner because the temperature of the chip will be going up quite significantly and inside the die there was some ring kind of interconnects which was proposed that connected the cores and also the memory management unit and the memory controller was also on chip.

(Refer Slide Time: 24:38)

(e) Haswell Architecture

- Next generation branch prediction
- Improved front-end
 - Initialize TLB and cache misses speculatively
 - Handle cache misses in parallel to hide latency
 - Improved branch prediction
- Deeper buffers for more instruction parallelism
- More execution units, shorter latencies
- More load/store bandwidth for better prefetching

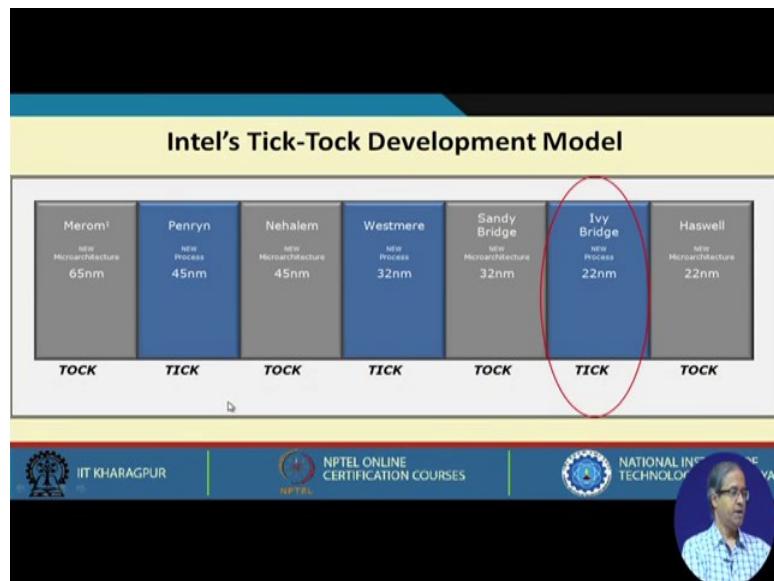
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY ROURKELA

Then the Haswell architecture that was one of the latest came up. Here the branch prediction was made much more sophisticated using lot of hardware support and there was some improvement in the front end like the TLB and the cache misses there were

some speculative processing here, cache misses were handled in parallel, multiple cache misses were handled in parallel to hide the individual latencies. Branch prediction was improved to a great extent, for the load store unit there were much deeper buffers you could fetch larger number of instructions keep it in a buffer and from the buffer you can issue them to the pipeline as and when required.

Suppose you have 16 instructions in a buffer you can decide out of those 16 which one instruction to execute next. You can have out of order execution also. Here you have larger number of execution units, it is actually highly superscalar.

(Refer Slide Time: 25:59)



This picture key illustrates the Intel philosophy of developing the newer generation of the processor, this is called the tick tock development model.

Now, the idea of tick tock development model is like this. Intel comes up with a new architecture based on the present fabrication technology -- this is called tock architecture, then it moves to tick means the same architecture is re-implemented using a better fabrication technology. Then in the next generation you again move from tick to tock; that means, a new architecture family with the same technology then again tick a better technology comes you move the same architecture with a better technology.

With this will come to the end of this lecture, where we very briefly looked at a couple of case studies one was that of a graphics processing unit and the other was the evolution of the Intel family of processors.

Thank you.

Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 64
Summarization of the Course

Hello. We have at last come to the end of this course on computer architecture organization. Over the last 12 weeks we have discussed the various features concepts that are pertaining to computer organization and architecture. We have looked at some of the basic concepts. We have also looked at some of the advanced concepts that are there in many of the modern day processors. Before we say good bye to you, we would like to summarize the coverage of this course as we have carried out over the different weeks. I will be requesting Dr. Datta to talk about the first 6 weeks of the course that she had handled.

(Refer Slide Time: 01:13)

Coverage of the Course

- WEEK 1:
 - Evolution of computer systems
 - Basic operation of a computer
 - Memory addressing and systems software
 - Software and architecture types
 - Instruction set architecture

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So we have covered 12 weeks in total. Initially we started with the evolution of computer system where we have looked into how computers are evolved over the years and of course, where we stand today. In fact, the last lecture that you have heard that would have given you a picture of where we are now. We have also discussed about the basic operation of a computer, how we can how a particular instruction is executed. What are the operations a computer performs, various memory addressing and system softwares,

like when we say that we are executing an instruction, we have shown that we store the instructions or the programs you can say in memory. And one by one we bring those instruction through some registers to our processor and then we execute it. In this process we require 2 important registers that is memory address register and memory buffer register or memory data register.

Then we talked about various software and architectural types, like von Neumann architecture and Harvard architecture. If you recall our discussion of von Neumann architecture there we discussed about that how both program and data are stored along site and each time processor needs to access a particular data or instruction it fetches from memory bring it to the processor and then it executes. But both if you want to access either the instruction or the data you have to hit to the same memory because we are storing both the data and the program in a same memory. This kind of architecture is called von Neumann architecture. And we also discussed about Harvard architecture. And we have seen that in Harvard architecture we stored the data and instruction in 2 separate memory. And this is widely used when we have discussed cache memory.

We have seen the advantages of using both Harvard architecture as well as von Neumann architecture. And if you think of that going from the first lecture to the last lecture some way or the other way you can relate these 2 methodologies or theses 2 basic types of architecture that we have discussed. We then discussed about instructions at architecture basically ISA is the programmer view of the computer. How the programmer actually looks the sees the computer I mean, what is happening inside the computer. Like when we say a programming modeled. Through that programming model we can actually bring out how we can execute an instructions, step by step what are steps required we can actually have it in more detail if we looked into the programming model. ISA is the programmer view of the computer.

(Refer Slide Time: 04:53)

The slide has a black header and footer bar. The main content area is yellow. It contains a bulleted list under the heading 'WEEK 2:'.

- WEEK 2:
 - Number representation
 - Instruction format and addressing
 - CISC and RISC architecture
 - MIPS32 instruction set and programming
 - SPIM: A MIPS32 simulator

At the bottom, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

Then we moved on with week 2, where we have seen we have briefly discussed about number representation and then we moved on with number representation with later part of the later weeks of computer arithmetic.

We discussed here a very important aspect of instruction format and addressing nodes. An instruction consists of 2 parts. I have discussed it in very much detail. An instruction is divided into basically 2 parts, opcode and operand. And we have seen specifically for MIPS architecture we have discussed that how the instruction format is. Then you think of that how the computer will understand that this particular data that is in at the operand will be fetch from either memory or from register. So, various addressing modes that are used today and that are existing we also discussed about that. And we closely discussed about CISC and RISC architecture and as a key study we have taken MIPS 32 instruction set and we have discussed a various programming.

Then we also discussed about SPIM that is a MIPS32 simulator. So, we have shown small codes in a (Refer Time: 06:30) which where we have written some assembly language code for some very easy program adding 2 numbers or factorial of a number or a palindrome of a number. Those examples we have shown, and through those examples what we have seen is that, that how we can write an assembly language code. And how an assembly language code using SPIM can be executed.

(Refer Slide Time: 06:58)

The slide has a black header and footer bar. The main content area is light yellow. At the top left, there is a small circular logo. Below it, the text 'WEEK 3:' is followed by a bulleted list:

- WEEK 3:
 - Measuring CPU performance
 - Choice of benchmarks
 - Summarizing performance results
 - Amadahl's law and applications

At the bottom, there is a dark blue footer bar containing three logos and their respective names: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

Then we moved on with week 3, where we actually discussed about the quantitative approaches in evaluating the design alternatives.

By this what we mean is that these are very important concept in modern day technology if you think of it. And here we actually shown that how we can say that a computer A is better than computer B, on what ground you can say that. What are the how quantitatively you have to say that we just cannot say that this computer is better than that. So, what are the design principles that are followed.

(Refer Slide Time: 07:43)

The slide has a black header and footer bar. The main content area is light yellow. At the top left, there is a small circular logo. Below it, the text 'WEEK 4:' is followed by a bulleted list:

- WEEK 4:
 - Design of control unit
 - Hardwired and microprogrammed control
 - Non-pipelined implementation of MIPS32 ISA

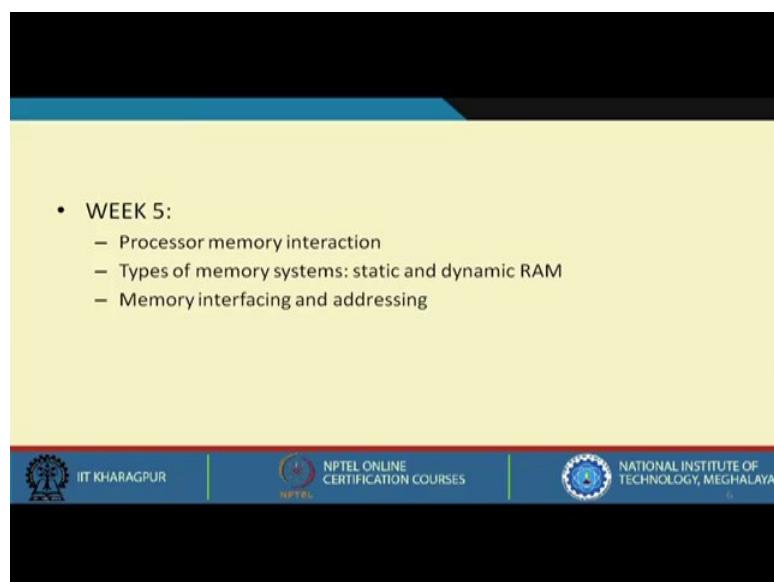
At the bottom, there is a dark blue footer bar containing three logos and their respective names: IIT Kharagpur, NPTEL Online Certification Courses, and National Institute of Technology, Meghalaya.

And then we have taken many examples to evaluate it, and the famous Amdahl's law and its applications we have discussed. Then we moved on to week 4 where we discussed about the design of control unit. How the instructions get executed inside a processor. The concept of micro-instructions and micro programs.

And then we also looked into 2 different ways namely hardwired and micro programmed control unit design. So, in hardware hardwired control unit design we have seen that it is much faster and how the instructions that need to be executed must be simpler in nature. And micro programmed control unit design where complex instruction can be executed in some way or other. We have a processor inside a processor. So, we have a similar kind of mechanism like processor does it fetches some data from memory it executes it. In the similar way micro programmed control unit it fetches some codes some micro instructions from the small memory and then it executes it one by one by one.

And here we have also discussed the non-pipelined implementation of MIPS32 instruction set architecture.

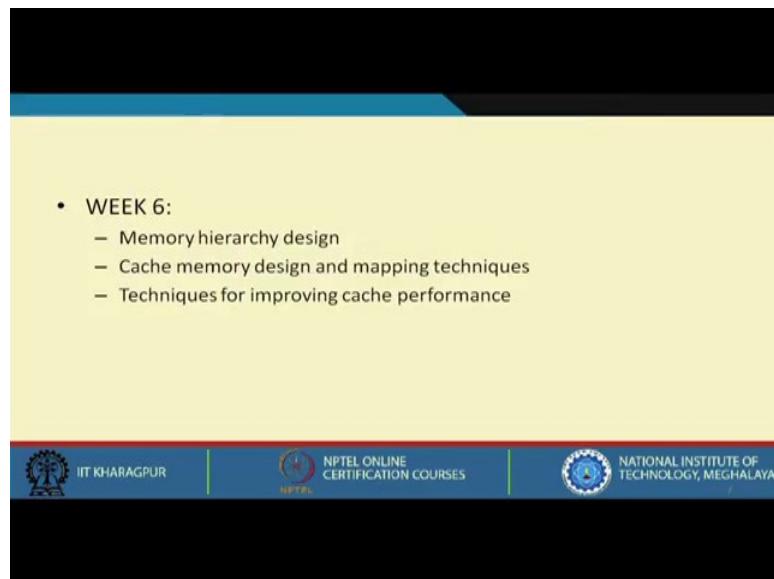
(Refer Slide Time: 09:05)



Then we moved on with week 5 where we discussed about we have been discussing about processor memory interaction from the beginning of the lecture if you remember, but we actually moved into what is memory in week 5. Where we discussed about the various memory technologies that are used today. And what are the types of memory that are there, we discuss thoroughly about the memory interfacing. We cannot use a very

large memory, how a large a smaller memories are combined together to form a large memory, memory modules are combined together to form a large memory we discussed about all these aspects memory interleaving.

(Refer Slide Time: 09:57)



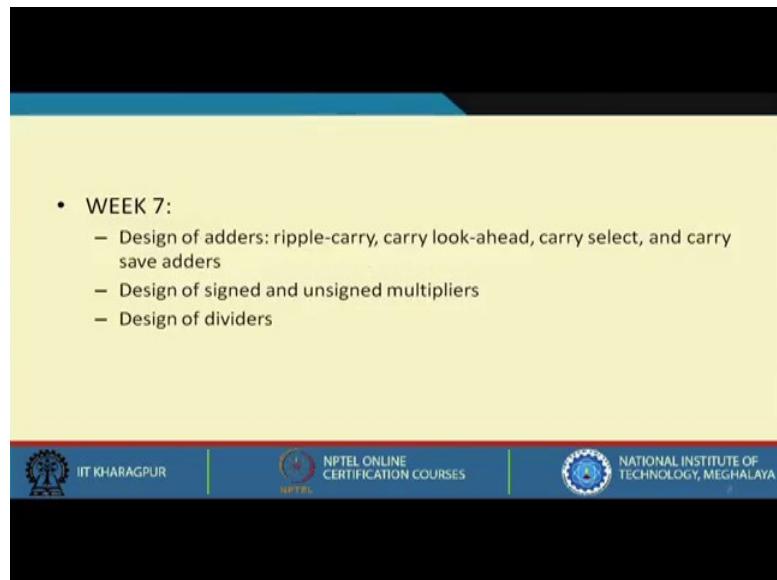
All these things we discussed in week 5. Then we moved on with week 6 where we discussed about memory hierarchy design. So, when we call a memory hierarchy design; that means, how we are moving from one level of the memory to the next level to the next level and so on.

If you think of a processor inside processor you have some registers. Register is ultimately some kind of memory only it is storing some kind of data, whatever is storing some kind of information data it is a storage unit. Then you move down to the next level where we have seen cache memory. That is the next level of memory where you have cache. So, cache is a kind of memory which sits between your processor and the next level of memory that is main memory. And what it does frequently accessed instructions or data are brought from main memory to cache memory and then it is sent to the processor. The next level after main memory is other disk and other memory systems that are there.

This is how the memory hierarchy is existing. And one important thing if you look into it is that the memory hierarchy when we move from registers then to cache memory then to the next level of main memory and then to disk the size is always

increasing. And size is increasing to a great extent, but the speed is equally decreasing. So, the registers are the fastest then the cache memory then the main memory and then the next secondary memory. So, here also we discussed about various techniques for improving cache performance, we discussed about various mapping techniques because if a memory is large we are bringing some data from a large memory to a small memory, we need to understand that which data we are bringing it where.

(Refer Slide Time: 12:06)

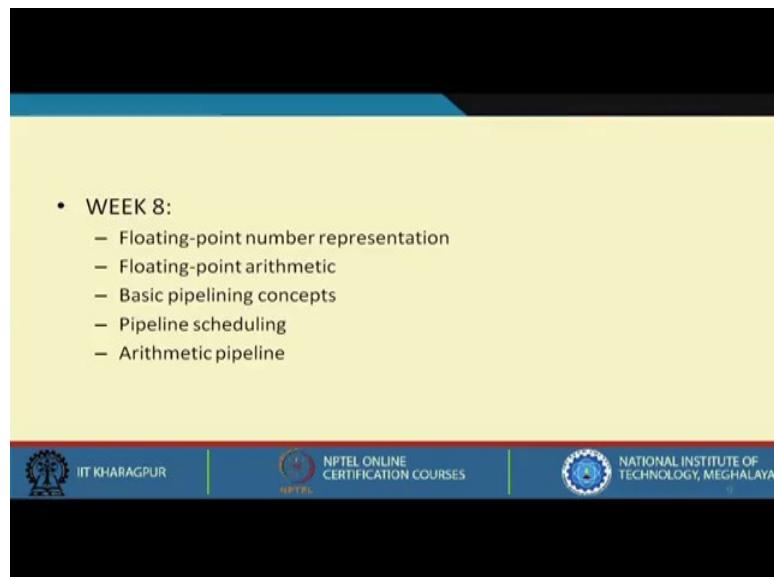


The mapping techniques are very much important and we discussed in detail about all those techniques. And finally, we discussed about the various improvement strategies.

In week 7 we started our discussion on arithmetic circuits because as you know the computers are suppose to carry out some calculations, and in that sense the arithmetic logic unit can be considered as the brain of the system or the heart of the system. So, the actual calculations are carried out there. In week 7 we started by discussing the various kinds of adders. So, their characteristics their harder complexities their speeds, and how we can design them? We discuss the designs of ripple carry adder, carry locate adder, carry select adder and also carry save adders. Then we moved on to the design of signed and unsigned multipliers, we looked at the shift and add kind of multiplication, we looked at Booth's multiplier then also we looked at some kind of faster multiplier using combinational blocks for example, carry save multiplier, carry save adder. That is a combination multiplier which runs very fast.

And lastly we looked at the design of various kinds of dividers. Specifically we discussed 2 algorithms restoring and non-restoring division. And also we discussed what kind of hardware is required to implement these kind of division algorithms.

(Refer Slide Time: 13:50)



Then we continue to week 8 where we moved onto the floating point numbers. Well since we did not discussed floating point number representation earlier we started by discussing the floating point number format. Now in particular the IEEE standard for representing floating point numbers. Then we talked about how we can carry out floating point addition subtraction and multiplication division and what kind of hardware requirements are there what kind of additional processing steps are there. We talked about those things. Our objective later on was to see your explore how this arithmetic circuits can be implemented in an arithmetic pipeline So as to run them faster with a higher latency and high lower latency and higher throughput.

We looked at the basic pipelining concepts first. What are the different kinds of pipelines how we can schedule the pipeline So that so called collisions do not occur, and then we explored how we can map some of the arithmetic algorithms like floating point addition, floating point multiplication these 2 examples you worked out into a pipeline. One thing we also mentioned that getting a pipeline implementation for divide division operation though not impossible is very difficult.

(Refer Slide Time: 15:47)

- WEEK 9:
 - Hard disk and solid-state disk
 - Input-output organization
 - Data transfer techniques
 - Interrupt handling

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

The pipeline implementation of dividers requires lot of hardware. There are many processors where division is often not pipelined. Then in week 9 we moved on to the input output part of the processing.

We started with the most important kinds of secondary memory devices that we see or that we use namely the hard disk and of course, the computing technology called solid-state disk which is very fast replacing hard disks. Nowadays if you buy laptops you often come with an option that you can get SSDs or solid-state disk instead of hard disks. Because in SSDs there are no moving parts, while computing even if you shake your laptop nothing will happen, which was a danger for hard disc because there was a mechanical moving part. So, if there are any jerks there is a chance of the hard disk getting damaged.

Then we moved on to the various input output organizational issues. How devices can be connected to the processor the concept of input output port, the concept of memory mapped I/O interfacing, then the concept of I/O mapped I/O interfacing. We walked out a lot of examples to illustrate the basic ideas on these concepts. Then we started our discussion on the various data transfer techniques that are possible. Broadly we talked about there are 2 methods, programmed and direct memory access. In this week we discussed mostly the programmed I/O concepts, mainly the synchronous I/O transfer

then the asynchronous I/O transfer or hand shaking and thirdly the most important interrupt driven I/O transfer.

In particular we discussed the interrupt processing in some detail where we talked about how multiple device interrupts can be handled, how interrupt priorities can be handled, the concept of interrupt nesting, the concept of identifying the interrupt the interrupting device using the concept of interrupt vector or some other thing vector interrupt.

(Refer Slide Time: 18:16)

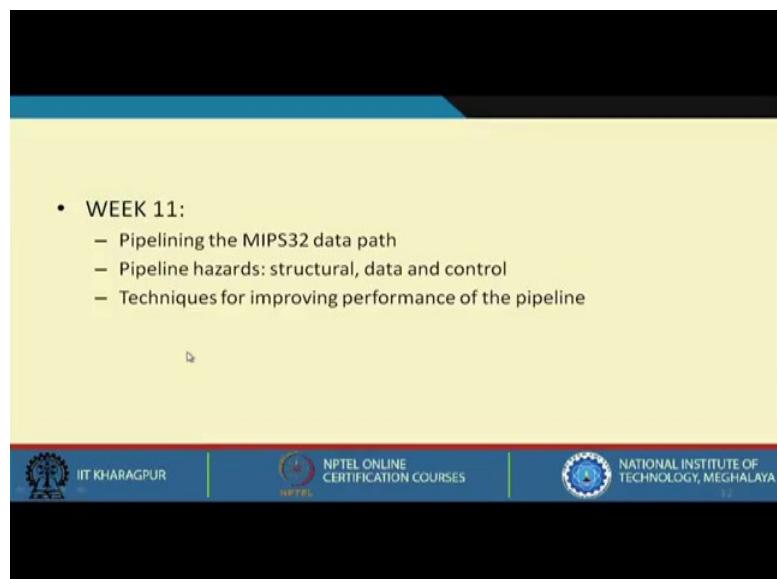
- WEEK 10:
 - Direct memory access (DMA) transfer
 - Interfacing of keyboard and printer
 - Bus standards inside a computer system
 - The USB bus standard

These are the things we had discussed during this week. Now in week 10 we first discuss the direct memory access transfer that is supposed to be very fast. So, instead of the processor executing an instruction to transfer data between an I/O device and memory here there is an external hardware controller called the DMA controller that takes hold of the memory bus and directly transfers data between I/O and memory without CPU intervention. Naturally all data transfer can be carried out at hardware speed without any instructions being executed.

Then we looked at some example interfacing, we looked at the examples of how we can interface a keyboard and a printer and we discussed how various methods like a synchronous interrupt driven these methods can be used along with these kind of devices. For more complex devices very similar things can be used, but for the sake of illustrations we took 2 examples that are sufficiently simple and easy to understand.

Then we looked at the various bus standards that are there inside a computer system. So, within a computer system if you look at an Intel motherboard there are so many kinds of buses that exist inside the system. These buses are often segregated using some bridges. There is a concept of a north bridge south bridge at the various internal buses are connected to one of these bridges. Then you have the external bus standard that today we almost universally use a standard called USB or universal serial bus. Earlier we had many different standards competing, but nowadays out of a conscious effort by the computer manufacturers we have come down to a standard the USB which can cater to the requirements of various different categories of I/O devices. From very slow ones like keyboard and mouse to very fast ones like hard disk. So, we also discussed this.

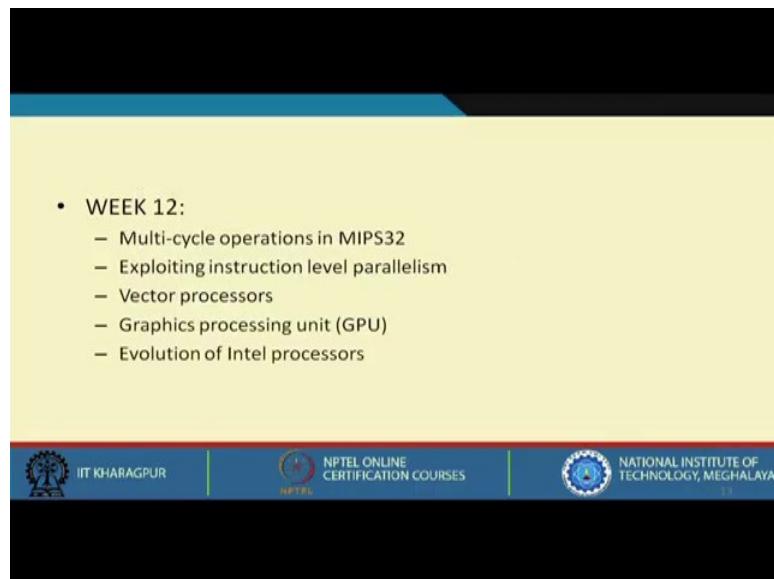
(Refer Slide Time: 20:35)



Now, during week 11 we explored the MIPS32 data path again. Earlier Dr. Datta discussed the hardware implementation of the MIPS32 instruction set, but here we just extended the discussion and discussed how we can implement MIPS32 instruction execution in a pipeline, specifically we talked about a 5 stage pipeline implementation. Because of the simplicity of the instruction set the pipeline implementation is also seen to be very simple. Then we looked at various different kinds of so-called pipeline hazards, that limit is the maximum speed up that can be achieved in a pipeline. We talked about structural hazards, data hazards, control hazards what kind of penalties it they incurred and we also suggested various techniques by which these penalties can be reduced well in some cases they can be eliminated also.

Various techniques various predictive branch performance improvements we had also proposed to improve the performance of a pipeline in general. Many of these techniques are actually used in modern computer systems today. Though we can call them as advanced topics, but any state of the hard processor we see anywhere in our mobile phone in our laptops wherever they use all these techniques in a very big way.

(Refer Slide Time: 22:19)



And finally, in the last week, we talked about first the multi cycle operation in MIPS32. How we can extend the MIPS32 pipeline to handle operations which may require more than 5 cycles? Particularly the ex stage for the floating point operations, the ex stage itself can require more than one cycles to run. Similarly multiplication division they may require multiple cycles to run in the ex stage. These are something called multi cycle operations. So, we specifically discussed how multi cycle operations can be handled, how data hazards are tackled in such a scenario, how interrupts and exception processing can be done.

Then we looked at how we can exploit instruction level processing instruction level parallelism by using techniques like loop unrolling which is a compiler technique so that our target architecture that today is highly parallel can exploit the parallelism there. Given the program the compiler will try hard to expose more and more parallelism by a concept which is called unrolling the loops repeating copies of the loops multiple times. So, that more parallelism can be exploited. We discussed vector processors. How or in

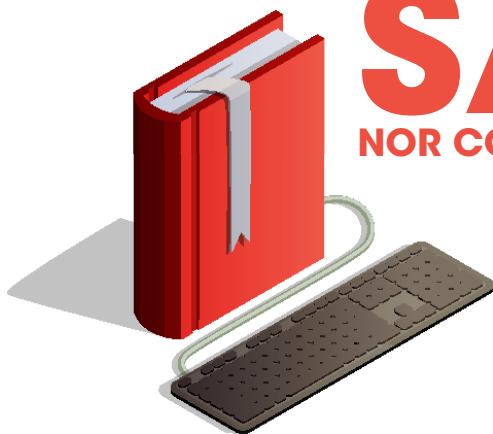
what way we can extend the basic MIPS32 pipeline to also incorporate processing of vectors of data, using now using a single instruction we can operate on 2 vectors?

Basically we are avoiding a loop translating it into a single instruction that is of course, much more efficient. Then we looked at some of the case studies. We looked at in particular the graphics processing unit or GPU and then we looked at very briefly how Intel processors have evolved over the years. This was roughly the total course coverage that we have gone through over the last 12 weeks just one point for you well in the examinations that will follow the kind of questions that you can expect will be quite similar to the assignments that were given to you. If you have been serious in solving the assignments understanding the concepts, we do not think it will be very difficult for you to tackle the examination. So, let me wish you all the best.

As we told in the introductory video that we will be starting with very basics and then we move on with some higher concepts and higher things. I think we have done that So far in these 12 weeks. And I must be sure that you have enjoyed this course. At any point of the time you have any difficulties in any section you can directly contact us through email. And I hope that the examination process that you will be having will be really fruitful and you will be actually gaining something out of it. At what said by Professor Sengupta that you will be getting some question definitely from assignments that is true, but you should also try to understand and go in to the subject such that any kind of question if comes, you should be able you should be in a position to answer it.

Thank you, thank you.

**THIS BOOK
IS NOT FOR
SALE
NOR COMMERCIAL USE**



(044) 2257 5905/08



nptel.ac.in



swayam.gov.in