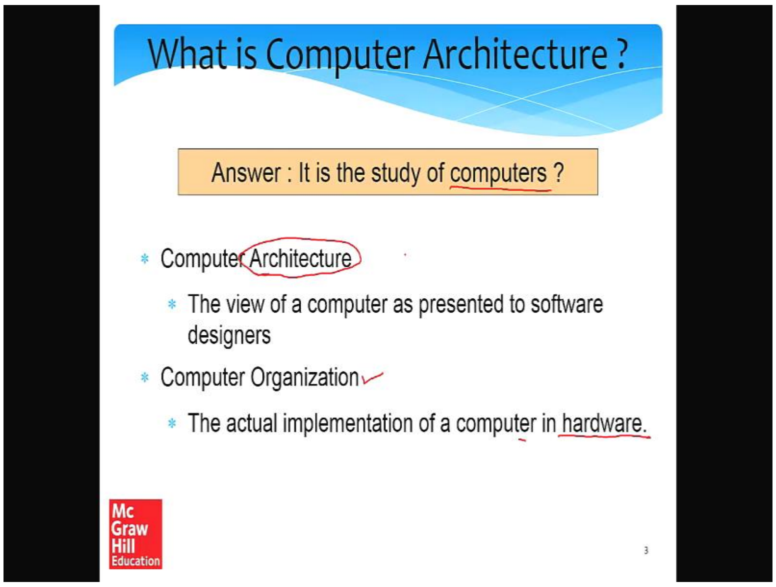
Computer Architecture

Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 01
Introduction to Computer Architecture

Hello everybody, welcome to the first lecture, which is a lecture on Introduction to Computer Architecture. The aim of this particular lecture is to first give you an overview of the field which is computer architecture, and then also to give you an overview of the book; that this particular chapter is based on, is the first chapter of the book Computer Organization and Architecture, published by McGraw hill in 2015. So, towards the end of this chapter I will show you the outline of the way concepts are presented in the book, and how the rest of the lectures will proceed.

(Refer Slide Time: 01:41)



What is Computer Architecture ?

Answer : It is the study of computers ?

- Computer Architecture
  - The view of a computer as presented to software designers
- Computer Organization
  - The actual implementation of a computer in hardware.

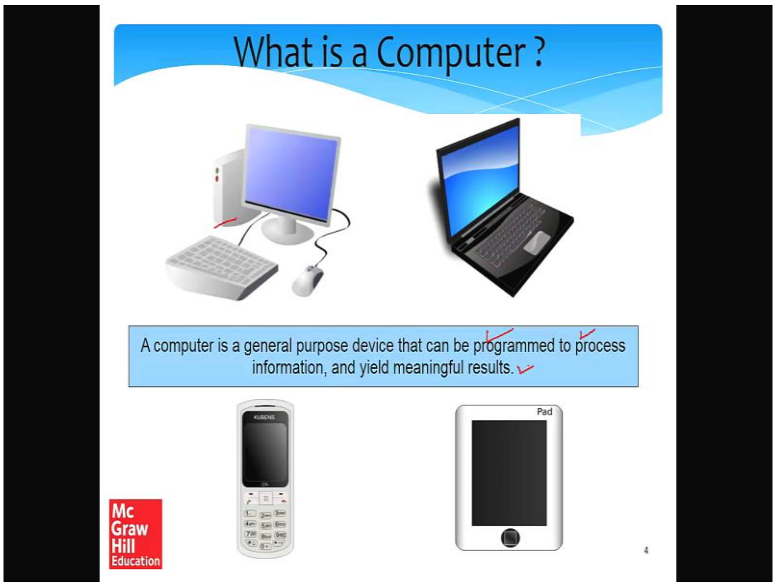
Mc Graw Hill Education

3

Let us begin by describing what exactly is computer architecture. So, the answer is very simple, it is a study of computers. So, computers as you know are there everywhere. The computer that I am using at the moment to record this video. Computers are used in cell phones, and cameras, and then nowadays computers are there in watches, they are almost there everywhere. So, let us distinguish between two kinds of terms that are used for the study of computers; the first is computer architecture, and the second is computer organization. So, computer architecture is a view of a computer; that is presented to software designers, which essentially means, that it is an interface or a specification that the software designers see, it is a view that they see, of this is how the computer works and this is how they should write, software for it, whereas computer organization, is the actual implementation of a computer in hardware.

Oftentimes, the terms computer architecture and computer organization, are actually confused or computer architecture is used for both. So, that is common, but we should keep in mind that there are two separate terms; one of them is computer architecture, and the other is computer organization.

(Refer Slide Time: 03:21)



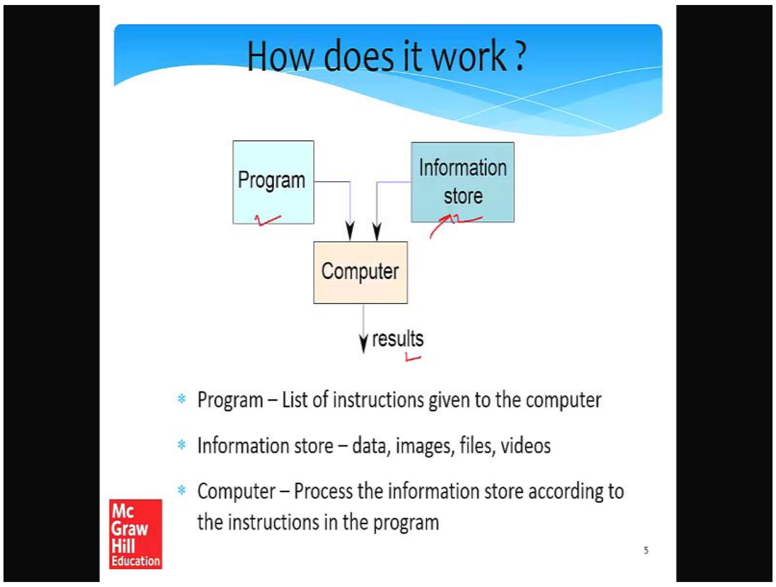
What is a Computer ?

A computer is a general purpose device that can be programmed to process information, and yield meaningful results.

So, again is a computer, we have computers everywhere, we have a computer on the desktop over here, we have a computer in a laptop or phone, an iPad. So, we can define a computer, as a general purpose device, that can be programmed, to process information, and yield meaningful results. So, mind you this definition has several facets to it; the first is, that a computer you should be able to program it. So, circuit that does a specific action is actually not a computer.

So, for example, let us say that you have small thermometers on top of the room, which is showing what is the current temperature: that is not a computer even though you know is showing it is temperature on a nice screen. The reason is that this device cannot be programmed. Second, it needs to be able to process some information that is given from outside; like you enter some keyboards some information via a keyboard or a mouse. It is processing that information; it needs to yield meaningful results. So, all these three facets are important for defining what a computer is.

(Refer Slide Time: 04:45)



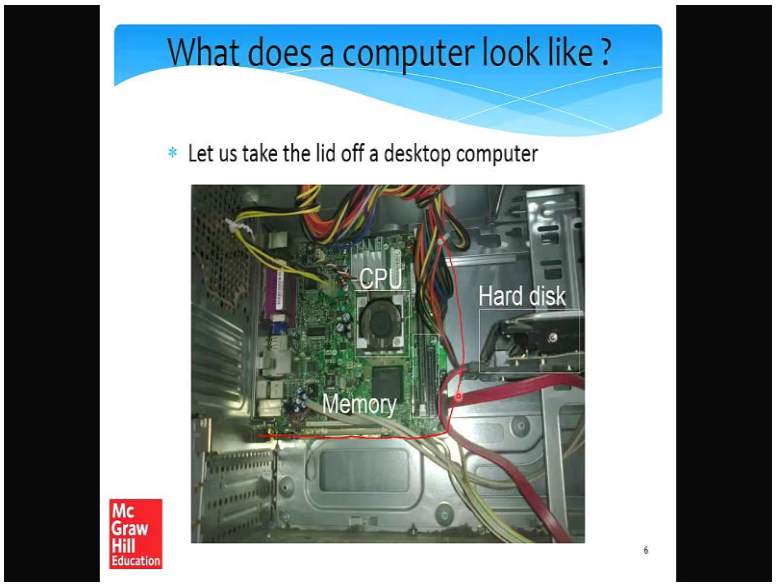
How does it work ?

- Program – List of instructions given to the computer
- Information store – data, images, files, videos
- Computer – Process the information store according to the instructions in the program

So, how does the computer work? A computer has a program which tells the computer what needs to be done; that is because, the way that we have defined a computer, is that it should be possible to instruct it to do something. So, having a program is point number one. Second there needs to be some information that the program will work on. For example, let us say you are trying to you clicked the photograph, and the photograph has some red eyes. So, we are trying to remove the red eye effect in photographs. So, in this case the photograph will be the information stored, and the program will be the piece of code; that is working on the information, photographs in this example, and then the finished good looking photographs is the result.

So, what is the program again, it is a list of instructions given to the computer. The information store is all the data image images files videos that a computer might process, and the computer once again is an intelligent device that can be instructed to do something, on the basis of the instructions, it processes some known information, to generate new and better and meaningful results.

(Refer Slide Time: 06:08)


What does a computer look like ?

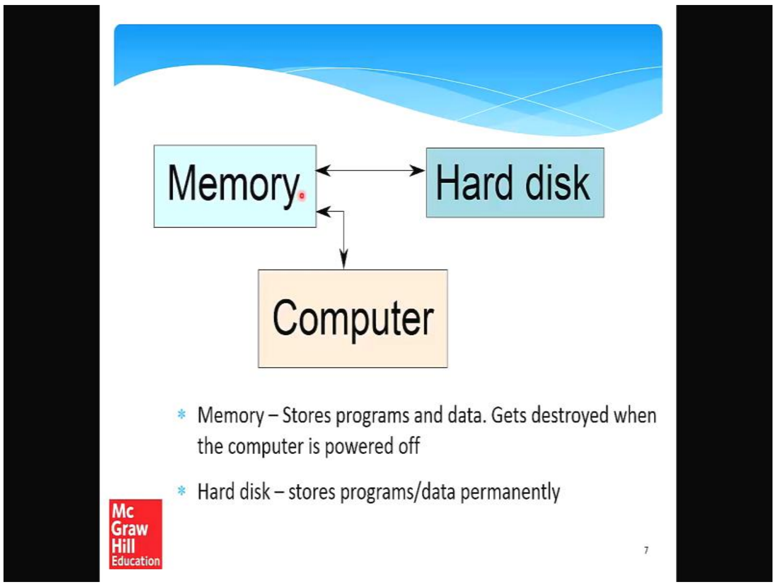
- Let us take the lid off a desktop computer

So, let us take the lid off a desktop computer and see what is over there. So, if you take, if you open a desktop computer, the first thing that you see over here, is a green board, and this is called the motherboard. So, this is a circuit board for the rest of the circuit is r. The two most important circuits is that at least we are interested in at the moment, is the CPU the central processing unit, which is the main brain of the computer. This is the CPU. You would also see a small fan on top of it the job of the fan is to remove heat, and it is the other rectangle over here which is called the memory of the main memory. So, this temporarily stores the information that the CPU the processor is going to use, and the computer processor reads data from main memory processes it and writes it back.

We also another very important unit over here which is the hard disk, this also saves information. What are the key differences between the memory in the hard disk; number one, the memory storage capacity is small. Maybe in today’s day and age it might be like 32 gigabytes, whereas, the hard disk storage capacity might be 10 times more or 20 times more. So, so we will get into the definition of how much what is a kilobyte or megabyte or gigabyte in chapter two, but essentially it is a unit of storage. So, the hard disk can typically store 10 times more data, but it is also significantly slower.

The other advantage of a hard disk is that if I turn the power off, all the data in the memory will get erased, whereas the data in the hard disk will remain. So, those are the differences. So, in this course, we will primarily be interested in these three unit is which are the CPU the memory and the hard disk, but mind you there are many other smaller processors all over the motherboard. So, will not have a chance to talk to them, but we will have a chance to at least discuss some of them, in the last chapter, in chapter 12 not at the moment though.

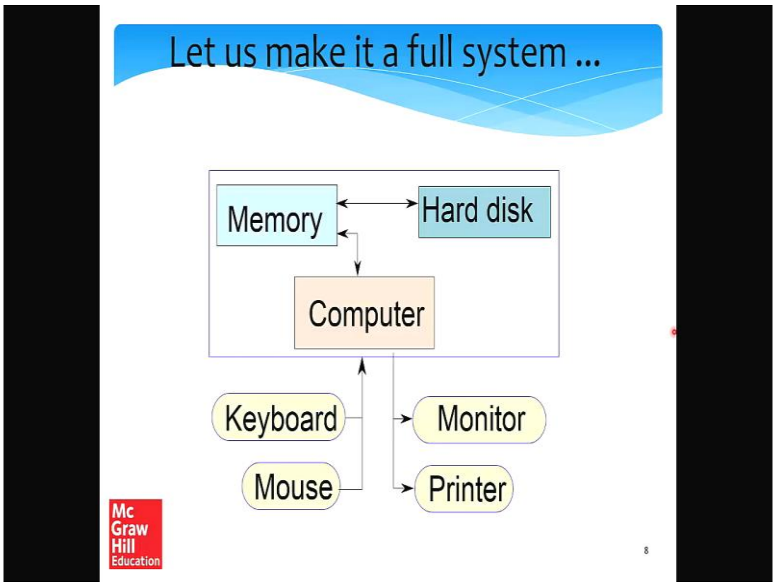
(Refer Slide Time: 08:36)



- Memory – Stores programs and data. Gets destroyed when the computer is powered off
- Hard disk – stores programs/data permanently

So, what does a simple computer look like? Simple computer, like the one that we looked at right now, has this computer a CPU, which does the processing; it has the memory and a hard disk. So, the hard disk maintains all the information’s, when a computer is powered off. When it is powered on, some of the information comes to the memory, and then the computer reads information from the memory, works on it, and again writes the results back.

(Refer Slide Time: 09:09)



Let us make it a full system ...

What more do we need to add to make this a full functioning system. We need to add I O devices, input output devices. This can include a keyboard a mouse for entering information. And for displaying information, it can be a monitor or a printer, to display the kind of information that a computer has computed. It can be other media also; like the network or a USB port, but we will gradually see what these are, at the moment let us confine ourselves to a very simple system.

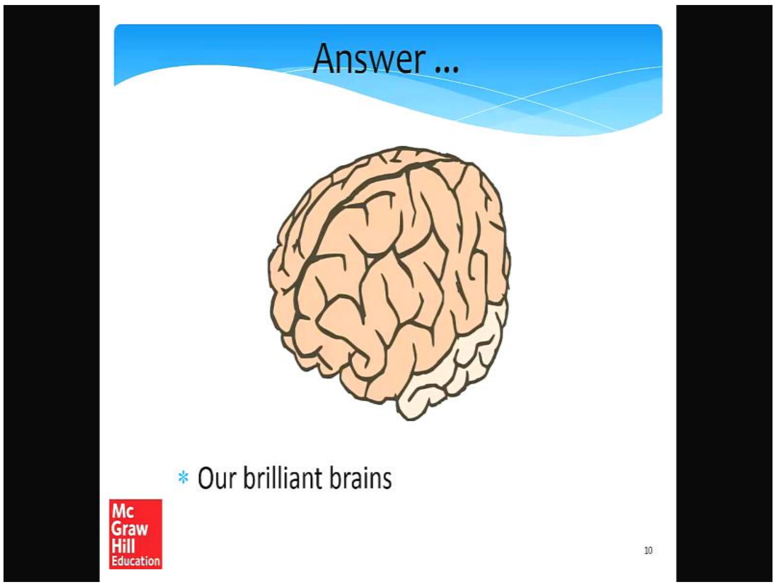
(Refer Slide Time: 09:54)

Food for Thought...

- What is the most intelligent computer ?

So, some food for thought, what do you think is the most intelligent computer. So, we will have many of these burger icons throughout the presentations. So, this will stimulate the student to think a little bit more.

(Refer Slide Time: 10:10)

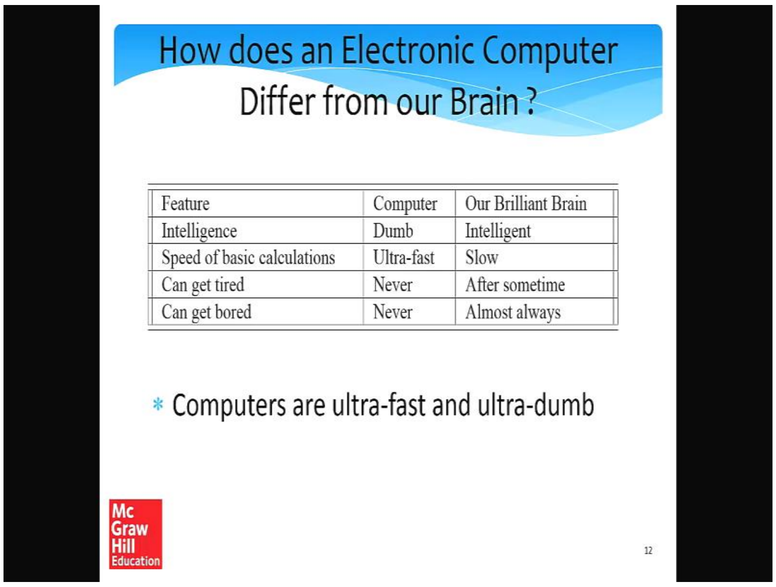


Answer ...

* Our brilliant brains

10

(Refer Slide Time: 10:20)



How does an Electronic Computer Differ from our Brain ?

Feature	Computer	Our Brilliant Brain
Intelligence	Dumb	Intelligent
Speed of basic calculations	Ultra-fast	Slow
Can get tired	Never	After sometime
Can get bored	Never	Almost always

* Computers are ultra-fast and ultra-dumb

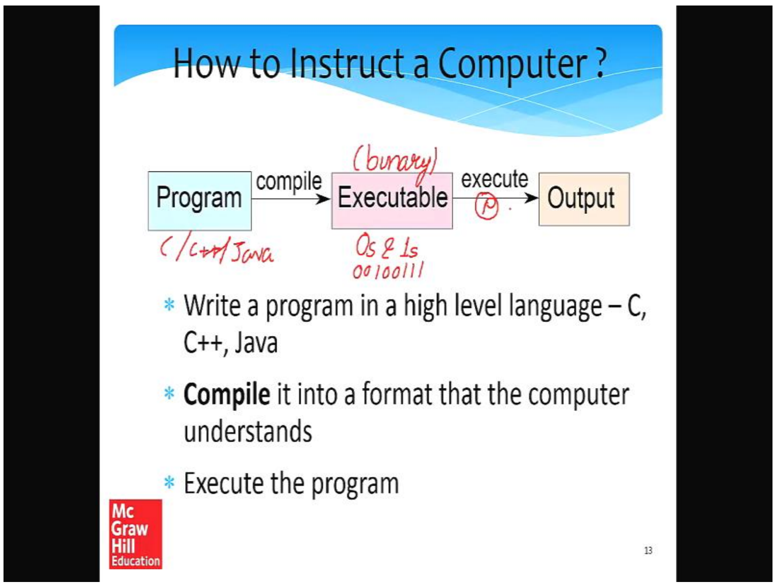
12

So, the answer is our brilliant brains. They are clearly the best computers by far and. So, how does a computer differs from our brilliant bring. Well our brilliant brains are extremely intelligent, they can do very complicated things, and they can think about the meaning of life, they are endowed with abstract thought. So, basically they can think about the difference between, thought thinking memory consciousness. Computers have none of that they are very dumb machines. All that they can do is they can add, they can multiply, they can subtract, but the advantage here is and that is mind you a very big advantage, that the speed of basic calculations in a computer is very high.

So, it might be doing dumb things; like addition and subtraction, but it is doing them a billion times a second. As a result it becomes ultra-powerful. Whereas, even the fastest human beings will not be able to do even a thousand or a hundred or even ten additions a second. So, this essentially says that we might be capable of very abstract and profound thought, but we are not capable of doing a lot of simple things very quickly; that is where computers excel.

Well some more advantageous. Computers never get tired, they never get bored, and they never get disinterested. Whereas, we human beings after some time we get tired, almost always we get bored, and disinterested. So, in a nutshell computers are ultra-fast and yet ultra-dumb; that is fine.

(Refer Slide Time: 12:03)



How to Instruct a Computer ?

$$\begin{tikzcd} \boxed{\text{Program}} \arrow[r, "\text{compile}"] & \boxed{\text{Executable}} \arrow[r, "\text{execute}"] & \boxed{\text{Output}} \end{tikzcd}$$

(binary)
C/C++/Java
0s & 1s
00100111
P

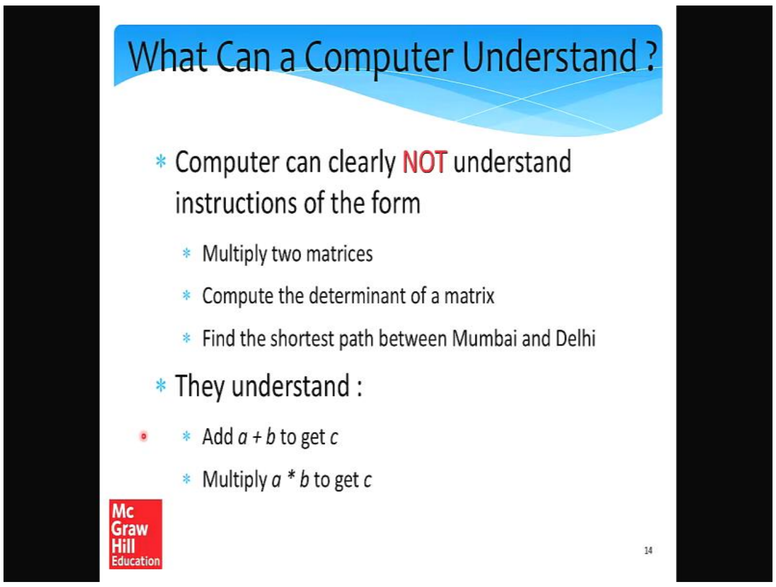
* Write a program in a high level language – C, C++, Java
* **Compile** it into a format that the computer understands
* Execute the program

13

So, how exactly do we tell a computer what needs to be done. So, what we typically do, is that we write a program, and the program is written in any other languages that you people know; it can be C, it can be C++, it can be java, it can pretty much be any language that students know, it does not matter. So, after that this is compiled into another program called an executable. So, the issue is that computers do not understand complicated languages; such as C++ or java. We only understand a language comprising of 0’s and 1’s. So, the language only has a 0 and a 1. So, any kind of a computer program would just be a sequence of 0’s and 1’s, like the sequence that I am writing right now.

After we have created an executable; so by the way an executable is also called a binary, it is also referred to as a binary. Once you have built the executable, the executable can be sent to the processor. So, I need to have a processor over here, and the processor will execute the program, and get the desired output. So, what exactly is the job of a compiler, the job of a compiler is to compile, compile what. Compiled programs written in a programming language such as c or C++, and convert it to a sequence of 0’s and 1’s. What is the job of a processor? The job of a processor is to take the sequence of 0’s and 1’s, understand what they are telling it to do, do it and generate some meaningful output.

(Refer Slide Time: 13:57)

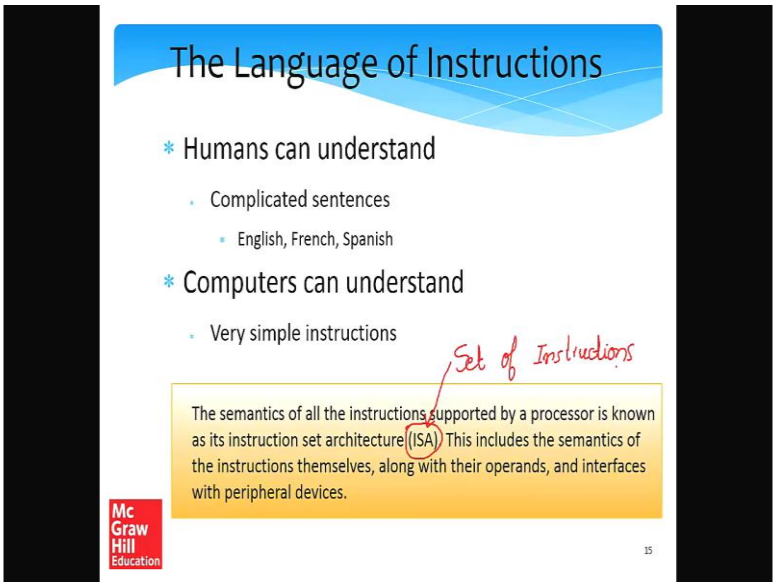


What Can a Computer Understand ?

- Computer can clearly **NOT** understand instructions of the form
  - Multiply two matrices
  - Compute the determinant of a matrix
  - Find the shortest path between Mumbai and Delhi
- They understand :
  - Add *a* + *b* to get *c*
  - Multiply *a* * *b* to get *c*

So, what can a computer understand? So, computers are not smart enough to understand instructions of the form multiply two matrices or compute the determinant of a matrix, find the shortest path between Mumbai and Delhi. So, they are simply not that smart, that they can do such problems with such high level of difficulty. They are not even smart enough to answer a question what is the time right now. So, what do they understand? They understand very simple statements, add a plus b to get c, multiply a and b a times b to get c.

(Refer Slide Time: 14:36)



The Language of Instructions

- Humans can understand
  - Complicated sentences
    - English, French, Spanish
- Computers can understand
  - Very simple instructions

Set of Instructions

The semantics of all the instructions supported by a processor is known as its instruction set architecture (ISA). This includes the semantics of the instructions themselves, along with their operands, and interfaces with peripheral devices.

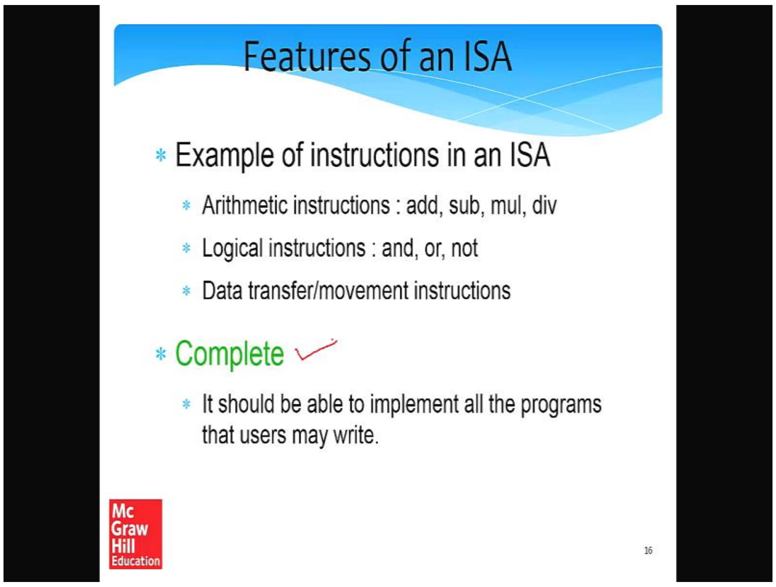
Mc Graw Hill Education

15

So, now the question is that how does this differ from humans. Humans can understand very complicated sentences in very complicated languages; English, French, Spanish, in Hindi, Chinese, Japanese, Italian. Computers in comparison can understand very simple instructions, extremely simple instructions; the semantics of all of the instructions. So, what is the meaning of semantics, it is a meaning, it is a way that instructions are used as what they stand for. So, the semantics of all the instructions supported by a processor is known as it is instruction set architecture or ISA. So, this includes the semantics of the instructions themselves, how the instructions are written, along with their operands, and interfaces with peripheral devices.

So, just to summarize what is an instruction set architecture or an ISA. It is pretty much the set of instructions, that architecture provides or alternatively a processor understands right. It is basically the set of instructions, that are given processor understands, and these are very simple set of instructions add, subtract, multiply types, but different processors might understand different kinds of instructions, or the instructions might be written in different ways; that is the reason you will have different ISAs, but pretty much the idea is the same.

(Refer Slide Time: 16:17)



Features of an ISA

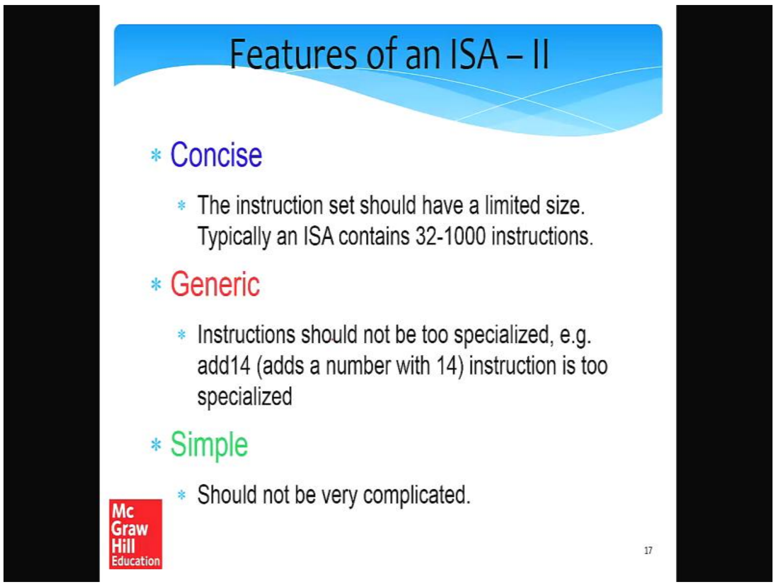
- Example of instructions in an ISA
  - Arithmetic instructions : add, sub, mul, div
  - Logical instructions : and, or, not
  - Data transfer/movement instructions
- Complete
  - It should be able to implement all the programs that users may write.

16

So, what would be examples of instructions in ISA arithmetic instructions, add, subtract, multiply, divide. Logical instructions and or a not, and data transfer instructions to move data between the memory and the processor slash CPU. So, what are the desirable features that you want of an ISA, and what are the absolutely critical features that you want. So, the most critical feature that you wanted it needs to be completed. What I mean by complete, is that it should be possible, it should be able, to implement all the prog programs that users may want to write.

So, you one user might want to write a program to read todays temperature and send it over the internet. One user might want to write a program. So, take a photograph or the blackboard, and recognize all the characters. So, irrespective of whatever is the program; the ISA should be able to, in a sense realize all the programs that users may want to write. So, we will discuss this later, but this is the notion of completeness, that whatever I want to write, it should be possible on a given processor.

(Refer Slide Time: 17:41)



Features of an ISA – II

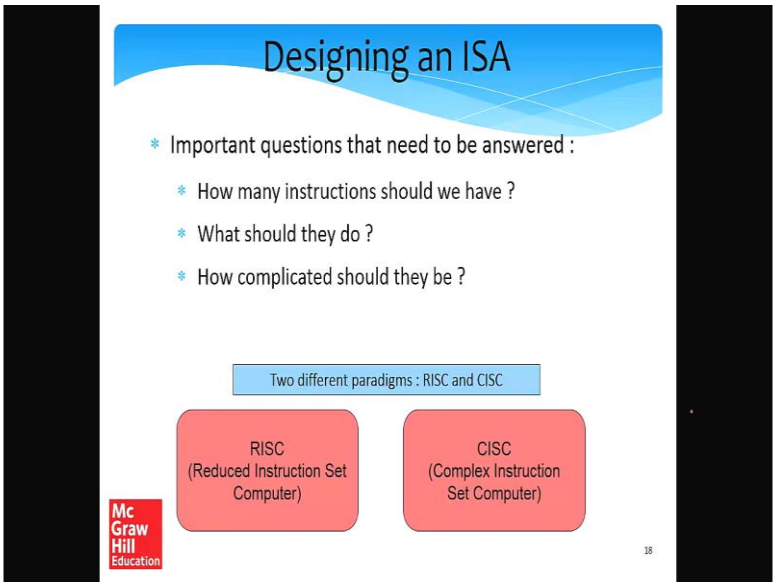
- Concise
  - The instruction set should have a limited size. Typically an ISA contains 32-1000 instructions.
- Generic
  - Instructions should not be too specialized, e.g. add14 (adds a number with 14) instruction is too specialized
- Simple
  - Should not be very complicated.

17

Next the Russell needs to be, the ISA needs to be concise, in the sense we do not want too many instructions, and we do not want 10000 instructions. We want a relatively smaller set of instructions. What is small is a debatable point, but it as if now ISA contained somewhere between 32 to a 1000 instructions, and different ISAs contain a different number of instructions. And instruction should also be generic, in the sense it should not be too specialized.

We should not have an instruction of the form, let us say add 14 that adds a number with just 14, because this instruction is too specialized, it is very unlikely that this instruction will be used by all the programmers, or even a large number of programmers. So, this is probably a bad idea, and we also want the instructions to be simple, what again is simple is a subjective thing, but the instruction should not be extremely complicated.

(Refer Slide Time: 18:47)


Designing an ISA

- Important questions that need to be answered :
  - How many instructions should we have ?
  - What should they do ?
  - How complicated should they be ?

Two different paradigms : RISC and CISC

RISC (Reduced Instruction Set Computer)

CISC (Complex Instruction Set Computer)

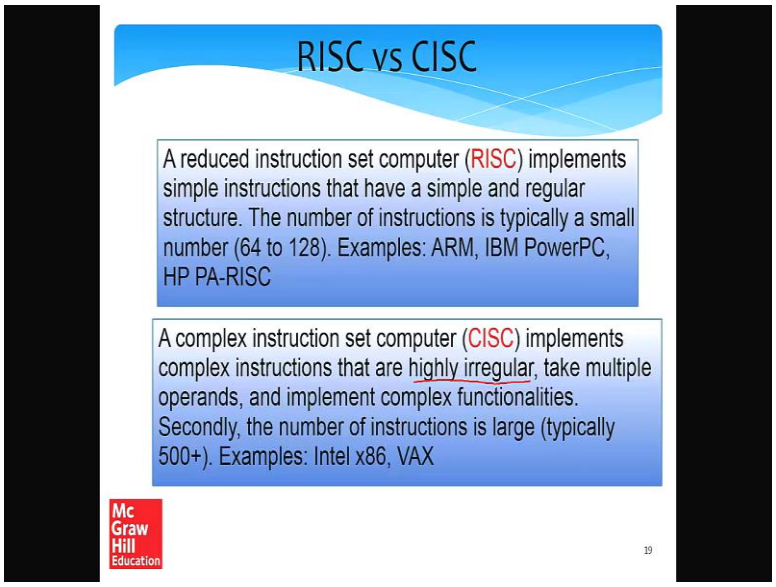
18

Now, let us come to designing an ISA. So, we already know that an ISA should be complete, concise, generic and simple. Completeness is a necessary criteria, being generic and concise and simple they are very subjective criteria. So, we can that, there is a massive leeway there. So, there are very important questions that need to be answered. Like how many instructions should we have, what exactly should they do, how complicated should they be.

So, there are different paradigms with which we can design instruction set architectures. So, we can have what are called RISC architectures, and we can have CISC architectures. So, RISC architecture is called a reduced instruction set computer or computing architecture. Here the idea is that we will have a fewer set of instructions. The instructions themselves will be really simple, such that it is very easy to understand them inside a processor.

So, ARM processors which are typically used in all the cell phones are examples of RISC processors. Similarly we have CISC processors, which have a very different paradigm. So, instead of having fewer instructions, CISC processors will have a lot of instructions.

(Refer Slide Time: 20:15)



RISC vs CISC

A reduced instruction set computer (RISC) implements simple instructions that have a simple and regular structure. The number of instructions is typically a small number (64 to 128). Examples: ARM, IBM PowerPC, HP PA-RISC

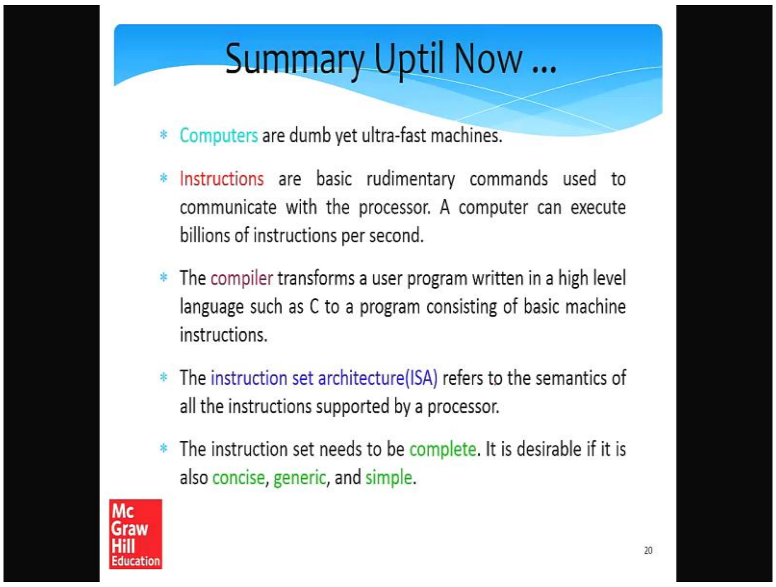
A complex instruction set computer (CISC) implements complex instructions that are highly irregular, take multiple operands, and implement complex functionalities. Secondly, the number of instructions is large (typically 500+). Examples: Intel x86, VAX

19

And they will also be, as we see on this slide highly regular, meaning that it is possible one instruction might take one argument, another instruction might take three arguments. They might take multiple arguments operands and implement very complex functionalities. So, this would be an example of a CISC construction set, and the number of examples of such instruction sets are Intel at x 86. Intel processors power most of the desktops and laptop computers today, to summarize whatever been saying. We have two kinds of instruction sets RISC and CISC. RISC means reduced instruction set, where you will have few instructions; they are simple and have a very regular structure. The biggest example as of today is the ARM instruction set, and ARM processes are used in almost all cell phones, and tablet us and smaller computing devices.

CISC instruction sets are complicated, they have a lot of instructions, and some of them are complex, and Intel uses them, Intel and A M D processors use them. So, they have their positives and their negatives, and we will get a lot of chance to discuss them as we go, as we move through this lecture series and, but we are not in a position right now to evaluate the tradeoffs.

(Refer Slide Time: 21:46)



Summary Uptil Now ...

- Computers are dumb yet ultra-fast machines.
- Instructions are basic rudimentary commands used to communicate with the processor. A computer can execute billions of instructions per second.
- The compiler transforms a user program written in a high level language such as C to a program consisting of basic machine instructions.
- The instruction set architecture(ISA) refers to the semantics of all the instructions supported by a processor.
- The instruction set needs to be complete. It is desirable if it is also concise, generic, and simple.

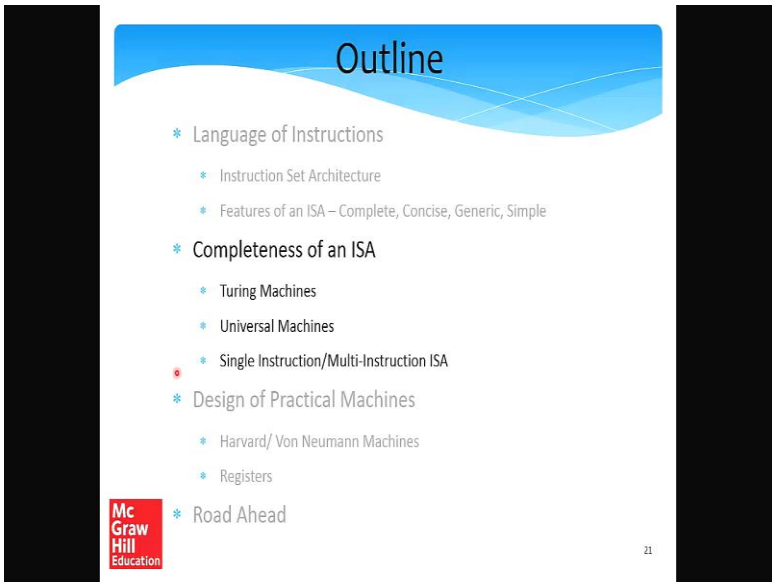
Mc Graw Hill Education

20

So, what is the summary up till now; computers are dumb yet ultra-fast machines, instructions are basic rudimentary commands used to communicate with the processor, very basic simple commands. The compilers job is to transform a user program written in a language such as c, to the language of instructions which are encoded in terms of 0's and 1's; such that the processor can understand it. The instruction set architecture refers to all the instructions that a processor supports, including how exactly an instruction works, what are it is semantics. This means what is the meaning what does an instruction do, how many arguments does it take, and what are the other features of an instruction. So, we want an instruction set to be complete, in a sense it should be possible to write any kind of program with it.

It is additionally desirable if it is also concise generic and simple, but then again we are getting into the RISC verses CISC debate, which has been a very consuming debate in the computer architecture community. So, that is the reason I said that you would want a concise generic and simple ISA, but there are other tradeoffs as well. We are just not in a position to discuss the tradeoffs right now.

(Refer Slide Time: 23:16)



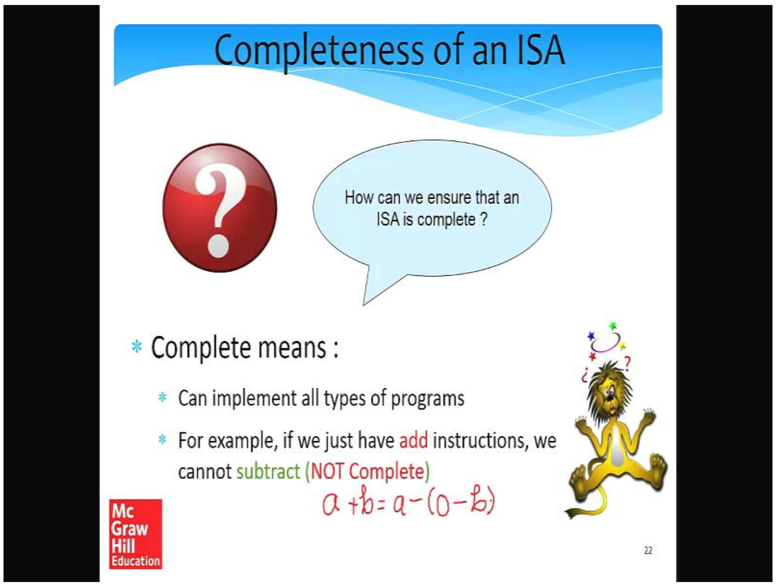
Outline

- Language of Instructions
  - Instruction Set Architecture
  - Features of an ISA – Complete, Concise, Generic, Simple
- Completeness of an ISA
  - Turing Machines
  - Universal Machines
  - Single Instruction/Multi-Instruction ISA
- Design of Practical Machines
  - Harvard/ Von Neumann Machines
  - Registers
- Road Ahead

21

So, coming to the outline, what we have studied up till now is pretty much the language of instructions, and what exactly is an instruction, what is an ISA; this part of how to decide if an ISA is complete or not is optional; and students who are not really interested in theoretical computer science can skip this part and directly move to the next part.

(Refer Slide Time: 23:48)



Completeness of an ISA

How can we ensure that an ISA is complete ?

- Complete means :
  - Can implement all types of programs
  - For example, if we just have add instructions, we cannot subtract (NOT Complete)

$a + b = a - (0 - b)$

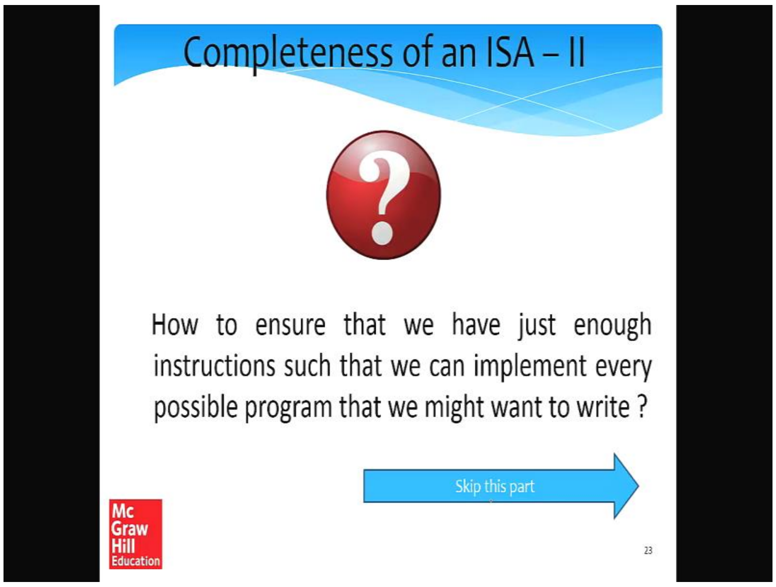
22

So, let us start this part a little bit, and then I will have an arrow, which the students will be able to click when they download the slides, and they would be able to skip the part. So, this particular subsection of the book in this particular part of the lecture, deals with the following aspect. How do we ensure that an ISA is complete, which means it can implement all kinds of programs, let me give an example.

Let us assume that we just have add instructions, if you just have add instructions can we subtract, can we compute 5 minus 3, the answer is no, but if you just have subtract instructions, can we. We can definitely subtract, but can we also add. The answer is yes. The reason answer is yes is as follows that a plus p is equal to. So, as you see what I have done is, that using subtracts instructions you can implement addition, but using add instructions you will not be able to implement subtraction. So, clearly subtract is a much more powerful instruction than addition. So, there are some other instructions that are more powerful than others.

So, what we need to see is that we need to have a set of instructions, such that no other instruction is more powerful than the entire set, and all programs that we want to write can be implemented in that set.

(Refer Slide Time: 25:33)



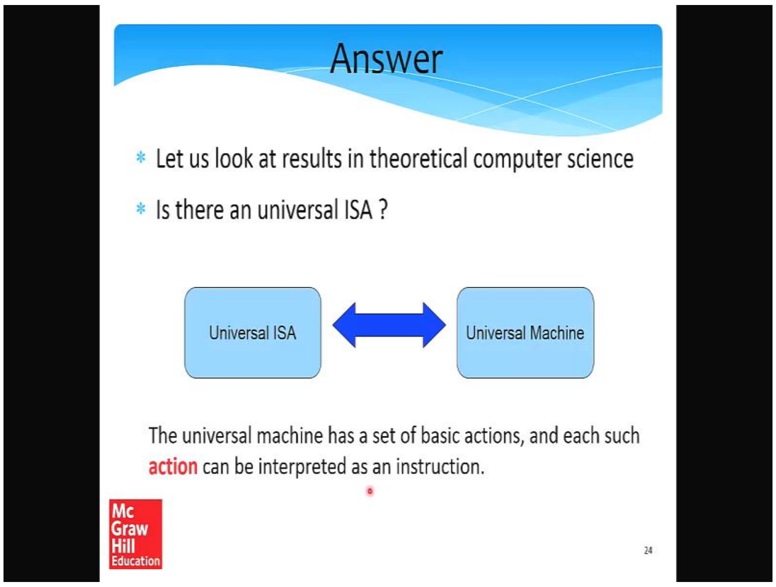
Completeness of an ISA – II

How to ensure that we have just enough instructions such that we can implement every possible program that we might want to write ?

Skip this part
Mc Graw Hill Education
23

So, how do we ensure that we have just enough instructions, such that we can implement every possible program that we might want to write? Well to answer this we need to take recourse to theoretical computer science. Some of this material can be slightly complicated. So, students who are interested can skip this part and directly go to slide 35. And the power point is in front of you all that you need to do, is essentially click the arrow over here, essentially click this arrow it will take you to slide 35.

(Refer Slide Time: 26:20)



Answer

- Let us look at results in theoretical computer science
- Is there an universal ISA ?

Universal ISA
Universal Machine

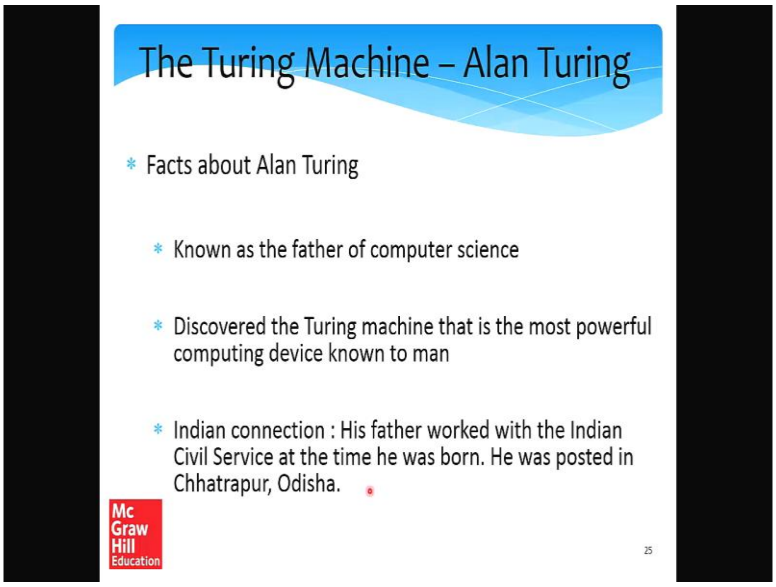
The universal machine has a set of basic actions, and each such **action** can be interpreted as an instruction.

24

But in the lecture, let us just get an overview of what is this part. So, what we want is a universal ISA. A universal ISA is an ISA which can implement all programs, known to mankind. This is the same as implementing a universal machine, for a machine and a processor are being used synonymously. The universal machine has a set of basic actions, and each such basic action can be interpreted as an instruction. So, they are more or less the same thing, because after the instruction is an action, when you are asking the processor to add two numbers, we are asking it to do an action.

So, the fact that I am asking the processor to do something that is an instruction, and the fact that the processor is doing it is an action. So, every instruction is correlated with every action. So, in that sense a universal ISA and universal machine pretty much mean the same thing.

(Refer Slide Time: 27:20)



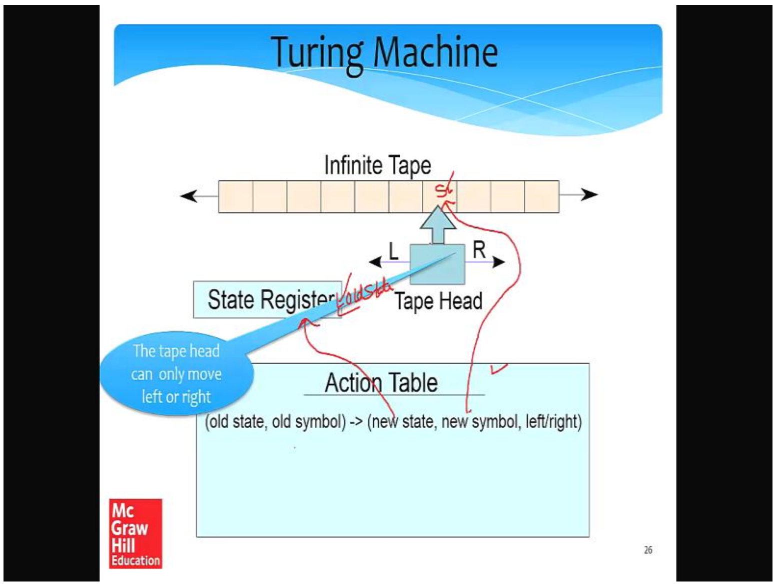
The Turing Machine – Alan Turing

- Facts about Alan Turing
  - Known as the father of computer science
  - Discovered the Turing machine that is the most powerful computing device known to man
  - Indian connection : His father worked with the Indian Civil Service at the time he was born. He was posted in Chhatrapur, Odisha.

25

So, how do we build one such universal machine that can compute the results of all kinds of programs? Well the answer to this is attributed to a gentleman for Alan Turing, who is the father of computer science, and he discovered the Turing machine that is the most powerful computing machine known to man; so his father. There is an Indian connection to Alan Turing. His father worked with Indian Civil Service at the time that he was born and, but again this is disconnected with the lecture per say. So, what is the Turing machine, it is a theoretical device, it is not a practical device, but at least theoretically it can compute the results of almost all the programs, that we are interested in writing.

(Refer Slide Time: 28:06)


Turing Machine
Infinite Tape
St
L
R
State Register
old St
Tape Head
The tape head can only move left or right
Action Table
(old state, old symbol) -> (new state, new symbol, left/right)
26

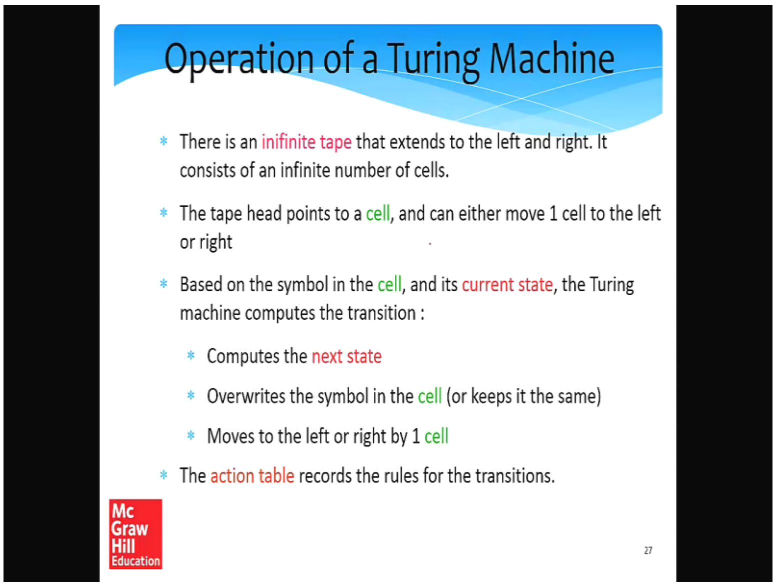
What is it? It is a simple device, very simple device. So, let us consider an infinite tape. A tape is, as shown in the diagram, it consists of basic cells, and each cell contains a symbol. A symbol can be a letter a number you can define anything. A symbol is basically an element of a certain set, where the set can be the set of letter, set of numbers, set of words, it does not really matter. So, every cell contains a symbol, and the tape is semi-infinite in the sense it. I am sorry the tape is infinite. So, the tape extends in both directions infinitely.

There is a tape head, that points to a current symbol on the tape, it can either move to the left or to the right, in every, show in every round, it can either move to the left or to the right. There is an additional structure called a state register, which maintains what is the current state, and the current state is one among a set of finite possible states. So, the way that a Turing machine works is as possible, and the way it works is encoded in the action table. So, we look at the old state. I am sorry we look at the old state over here, what is the old state, or the current state over here and what is the symbol under the tape head, using the old state and the old symbol, we find out the new state.

So, the new state is written into the state register. We find the new symbol, so the current symbol is overwritten with a new symbol, and we move the tape head one step to either the left or to the right.

So, this is an incredible machine in the sense that it might not be obvious to you, but this can implement any program that you and I are interested in writing. So, how will such a simple tape head that can only move left or right to it?

(Refer Slide Time: 30:22)



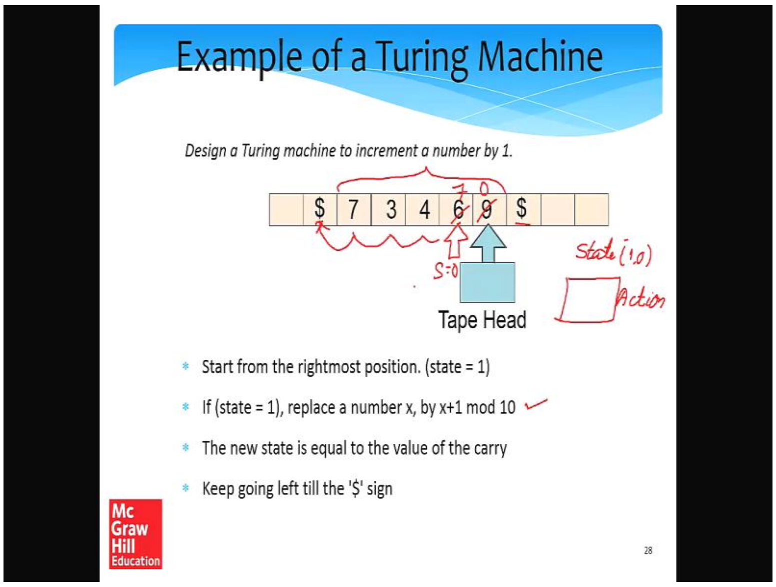
Operation of a Turing Machine

- There is an inifinite tape that extends to the left and right. It consists of an infinite number of cells.
- The tape head points to a cell, and can either move 1 cell to the left or right
- Based on the symbol in the cell, and its current state, the Turing machine computes the transition :
  - Computes the next state
  - Overwrites the symbol in the cell (or keeps it the same)
  - Moves to the left or right by 1 cell
- The action table records the rules for the transitions.

27

Well, let us take a look at an example then only we will find out how it works. So, I am skipping this particular slide, because this summarizes what I just described, about the operation of a Turing machine.

(Refer Slide Time: 30:37)



Example of a Turing Machine

Design a Turing machine to increment a number by 1.

- Start from the rightmost position. (state = 1)
- If (state = 1), replace a number x, by x+1 mod 10
- The new state is equal to the value of the carry
- Keep going left till the '$' sign

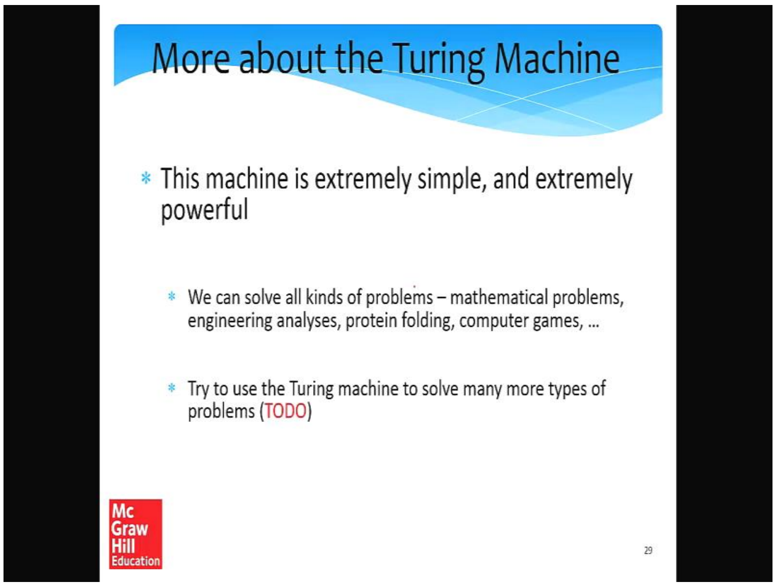
28

So, let us take a look at a simple example, there are a few more examples in the book with far more details. I would advise you to go to the book and read that part, but let us at least take a look at a very simple example over here. Let us assume that we have a number that is written on the tape of the Turing machine, where each cell contains a single digit, and it is demarcated on both sides for the dollar signs. The tape head starts from the unit are place, and what we want to do is increment the number by 1, add 1 to it. So, what we need to do is we need to figure out two things.

One is that what would be the states that we keep in a state register, and how do we construct the action table. Well the states that we want to keep. Let us have two kinds of states 1 and 0, the state 1 basically means that we need to add 1 to the current digit; that is under the tape head. So, you can think of it as a carry right, that 1 needs to be added, and 0 means that nothing needs to be added. So, let us have only these two states 1 and 0. So, here is how we would start out to construct the action table. We will start from the rightmost position with a state equal to 1, which means that 1 needs to be added. If the state is equal to 1, we replace whatever number x that is there over here with x plus 1 mod 10, which means that we replace 9 with 0 in the first round.

So, the new state will be equal to the value of the carry. So, since I added 1 to 9, we still have a carry of 1. So, the next state will be equal to the value of the carry which is still 1, and the tape head will now move over here. So, the tape head will then see that, it has 6 under the tape head. So, it will come to this line over here. Replace 6 by 7, but since there is no carry. Here the state will be equal to 0. Subsequently the tape head will move to each of these locations, and since the state will be 0, nothing needs to be changed, and finally, the computation will complete, when it hit is the dollar sign over here. So, what we have seen over here is an example, of how the Turing machine can be used, and how it can be used to effect a very simple computation, which is to add 1 to a number.

(Refer Slide Time: 33:18)



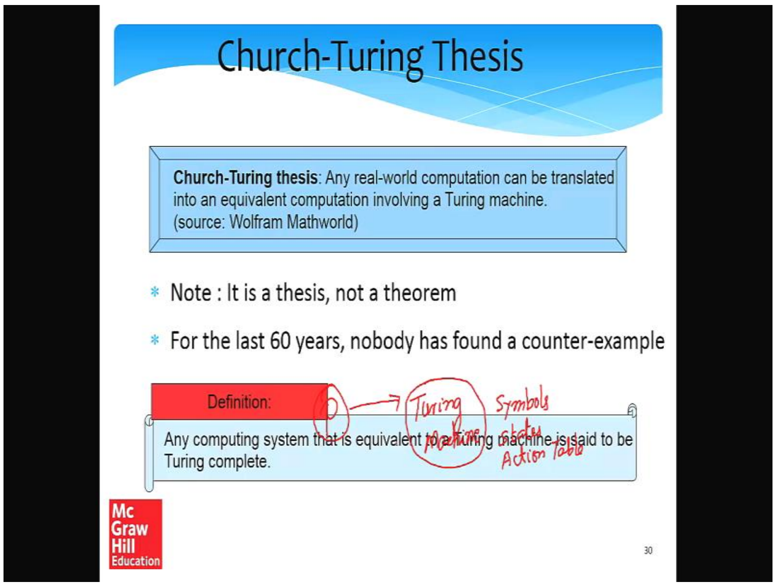
More about the Turing Machine

- This machine is extremely simple, and extremely powerful
  - We can solve all kinds of problems – mathematical problems, engineering analyses, protein folding, computer games, ...
  - Try to use the Turing machine to solve many more types of problems (TODO)

29

So, this machine looks simple, it actually is simple, yet it is extremely powerful. We can solve all kinds of problems, starting from mathematical problems, engineering problems, protein folding, games, you name it you can do it, the question is how well start by reading the book, and then read a book on theoretical computer science. Any languages in formal language, or any book on formal languages or automata theory, will give you the answer.

(Refer Slide Time: 33:57)



Church-Turing Thesis

Church-Turing thesis: Any real-world computation can be translated into an equivalent computation involving a Turing machine.
(source: Wolfram Mathworld)

- Note : It is a thesis, not a theorem
- For the last 60 years, nobody has found a counter-example

Definition:

Any computing system that is equivalent to a Turing machine is said to be Turing complete.

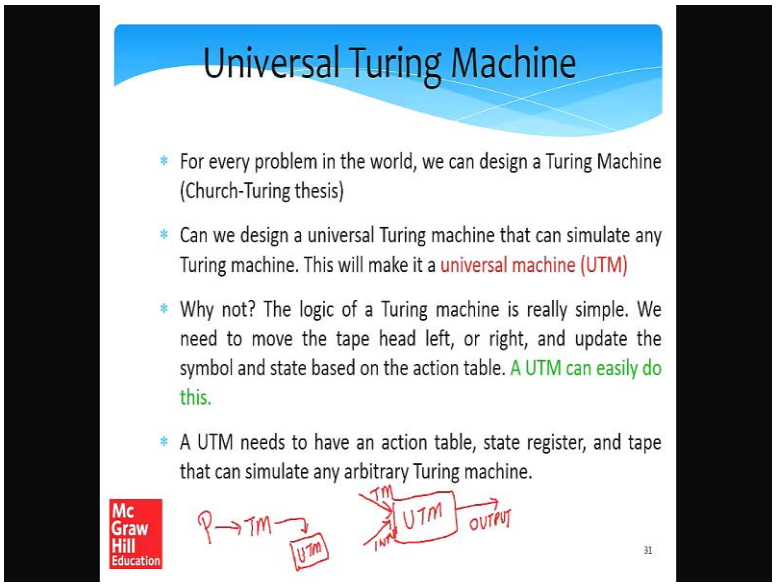
30

So, you can try using the Turing machine to solve many more kinds of problems. So, now, I should discuss; what is the Church Turing Thesis. So, during the time that Turing was doing his work, there was another mathematician could Alonzo Church. So, both of them Alonzo Church also came up with another formalism, which can be used to write programs for anything that we possibly want to write for. So, together the thesis is attributed to them. What is the Thesis? It is not a theorem, it is essentially a statement which is said without proof, and it is up to others to find a counter example, but in the last 60 years or so we have not found a counter example; so this Church Turing Thesis.

So, you can think of a thesis as a hypothesis. The Church Turing thesis says, that any real world computation can be translated, to an equivalent computation involving a Turing machine. So, what I can do is that if I have a program which is a real world computation. I can translate it, to exactly the same and equivalent competition, which does exactly the same on a Turing machine.

So, what do I need to do, is in the Turing machine all that I need to do, is create a set of symbols that need to be written on the tape, create a set of states, and then populate the action table; that is all that I need to do, to basically create a Turing machine for any possible program. So, mind you there is no proof of this, is just that we have not found a problem that cannot be solved on a Turing machine in the last 60 years. So, any computing system that is equivalent to a Turing machine, is said to be Turing complete.

(Refer Slide Time: 36:00)



Universal Turing Machine

- For every problem in the world, we can design a Turing Machine (Church-Turing thesis)
- Can we design a universal Turing machine that can simulate any Turing machine. This will make it a universal machine (UTM)
- Why not? The logic of a Turing machine is really simple. We need to move the tape head left, or right, and update the symbol and state based on the action table. A UTM can easily do this.
- A UTM needs to have an action table, state register, and tape that can simulate any arbitrary Turing machine.

$$\begin{tikzcd} P \arrow[r] & TM \arrow[r] & UTM \end{tikzcd}$$

$$\begin{tikzcd} TM \arrow[r] & UTM \arrow[r, "OUTPUT"] & {} \\ INP \arrow[ur] & & \end{tikzcd}$$

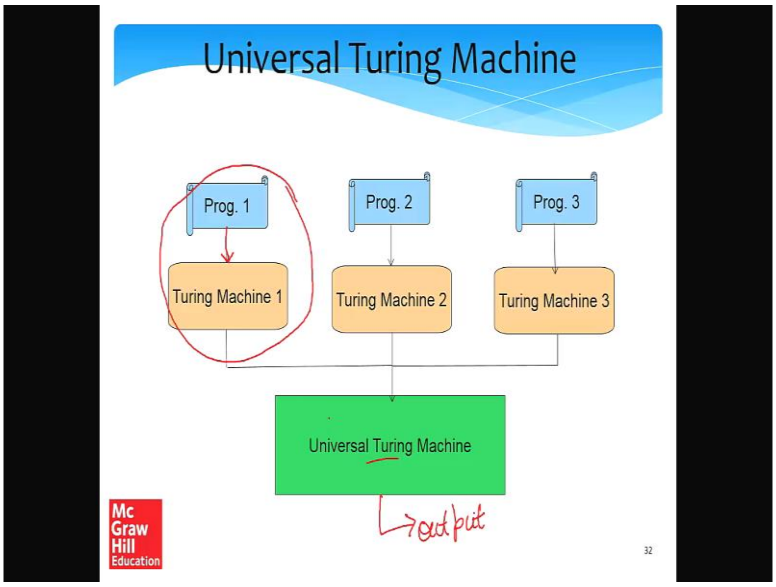
31

And. So, we will be using this term frequently at least the next few slides. So, now, for every problem in the world we can design a Turing machine, this Church Turing thesis tells us. Now can we design a universal Turing machine that can simulate any other Turing machine? So, what is a universal Turing machine? So, let us call it an UTM. So, what a UTM does is as follows that if a Turing machine is given to it, and the inputs are given to it. So, whatever is written on the tape is given to it. Can it produce an output?

The output that the original Turing machine would have produced, can it in a sense simulate the Turing machine, and simulate it is action table, all of it is actions, and produce the output to the original Turing machine would have produced. If we can design such a machine we will call it the universal Turing machine, or the UTM, and the UTM it unit is a truly universal machine, because every program can be mapped to a Turing machine, and if the Turing machine can run on a universal Turing machine.

We basically have a universal machine that can produce the results for any program. So, the question is why not. The logic of a Turing machine is really simple, what do we do. We read the current state and the symbol that is pointed to by the tape head. Update take it then consult action table, change the current state, write a new value in the tape, write overwrite the earlier symbol if required, and then move the tape head left or right. If this is all that needs to be done, a UTM or universal Turing machine can easily do this. It can take a Turing machine as input, and then easily simulate it. A UTM needs to have it is own action table state registered, and a tape to simulate any arbitrary Turing machine, but that is not very hard to do, and definitely some of you who are listening to this video, can take a crack at doing this, it is not that very hard at all.

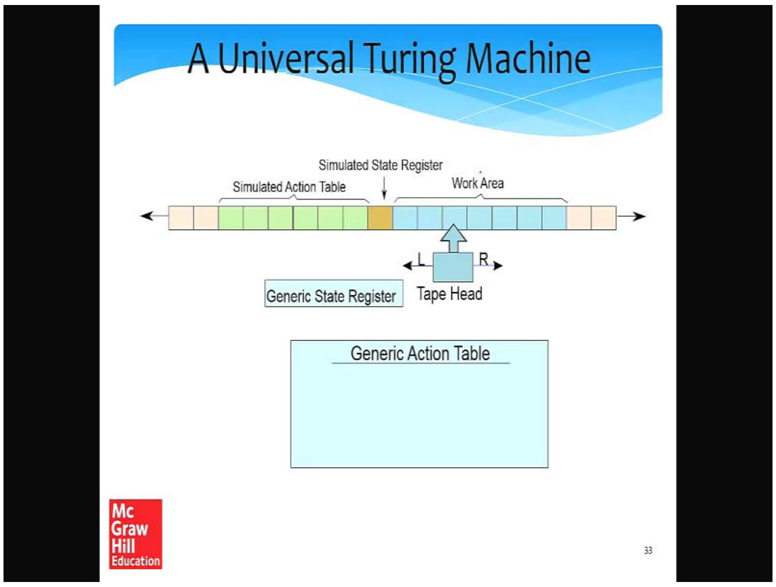
(Refer Slide Time: 38:27)



So, where do we stand right now? We stand at this point, where we know from the Church Turing thesis that given every program, given any program we can make it Turing machine, to implement the logic of the program and given inputs, a Turing machine and provide the output. So, basically given any program or any problem that we want to solve, we can always create a Turing machine for it. And we have a universal Turing machine, which given the Turing machine and the input, we will be able to produce the meaningful output.

So, the universal Turing machine by itself is a computer, which can be programmed. How is it being told what to do? Essentially the Turing machine is that is being given as input, is telling that Turing UTM what to do, and the tape of this Turing machine has the information, that needs to be processed, and it is the job of the UTM to simulate the Turing machine that is being given to it, and generate meaningful output.

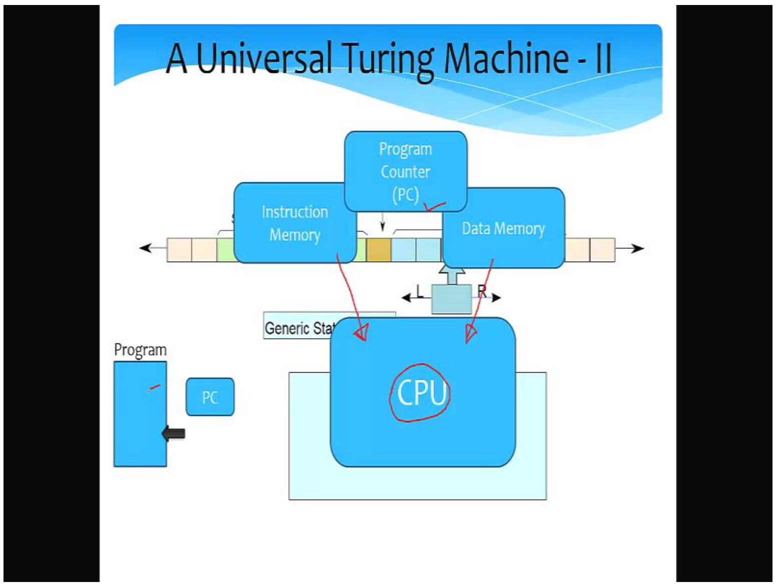
(Refer Slide Time: 39:49)



So, how does a universal Turing machine work? Well it is very simple. So, it has a generic action table, which is the action table of the UTM. It has a generic state register, and it has it is tape head. On the tape, it has the action table of the Turing machine; that it is simulating called the simulated action table. It simulates the state register of the Turing machine that it is simulating.

So, it has a simulated state register. It also simulates the tape of the Turing machine that it is simulating, plus it has it is own temporary symbols that it uses. So, we can refer to the tape and a temporary symbol area the work area. So, this is pretty much how, you would design a universal Turing machine, and it is fairly simple to do so.

(Refer Slide Time: 40:42)



A Universal Turing Machine - II

So, how exactly is this related to a modern process? The way it is related is actually very simple and very obvious the generic action table that you have. So, what exactly is it doing? It is finding out that what does simulate a Turing machine want to get done and it is doing it. So, we can think of the generic action table as the CPU, as the processor, which is a general purpose computing device you give it instructions, the processor will give you meaningful results. The work area is pretty much the tape of the simulated Turing machine plus an area where you can keep some temporary data and symbols, can be referred to the data memory, but this is where contains all the information.

The simulated action table which tells the UTM what needs to be done, essentially contains all the instructions. So, that we can think of it as the instruction memory, and the simulated state register, is basically telling you that at what point are we in the simulated Turing machine, and what needs to be done next. So, we can think of this as a program counter. So, these are essentially the four basic elements of any computing system.

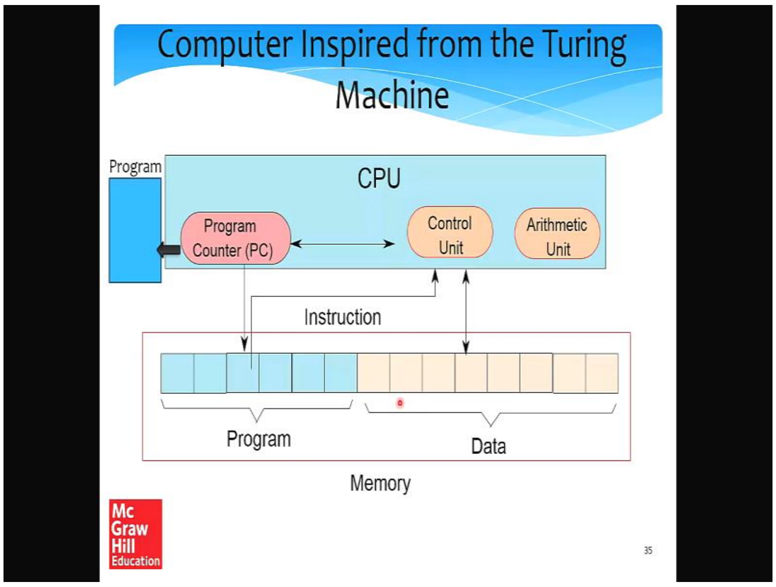
Let me go over there once again. The CPU is pretty much the brain of the system, it is a generic computing device, and where given an instruction it does whatever it is told to do. The instructions pretty much come from the instruction memory, which holds all the instructions, and the instructions work on the data. So, they come from the data memory and we also need to know that given a certain instruction of, what do we need to do next right in a given the current instruction that we are doing. What exactly do we need to do next, and what should be the behavior of the next few instructions, and this can only be done by having some kind of a storage area, to at least save the fact that where exactly are we there inside the program, and the pretty much like the current state, and this in a modern processor is called a program counter and this concepts.

So, mind you this, you can take a look at the book at the relevant chapter and you can see a better mapping between a universal Turing machine and these concepts. So, I am just presenting a very high level view out here. The job of a program counter is basically to indicate to the hardware, as well as to the software, what exactly is the point in the program, at which the processor is currently there. So, think of a program, and the program. Program counter is typically abbreviated as PC.

So, it holds for which point in the program are we currently there at, and it continuously keeps changing, as the program keeps executing. So, it keeps on changing it is position. So, the simulated state register has sort of become the program counter in a practical implementation. Even though you have not understood possibly most of the theoretical aspects of this; here is the important takeaway point which will also do summarized in the next slide, that from a theoretical device, we have created a practical device, with pretty much cutting down the theoretical device into multiple chunks, and assigning a practical device to each chunk.

So, the generic action table may merit the CPU. The simulated action table we merit the instruction memory which has all the instructions. The state register we merit. The simulated state register we made that the program counter or the PC. So, what this essentially tells us, is that where exactly are we they are inside a program, and the data memory, pretty much tells us that, what is the data that we need to work on.

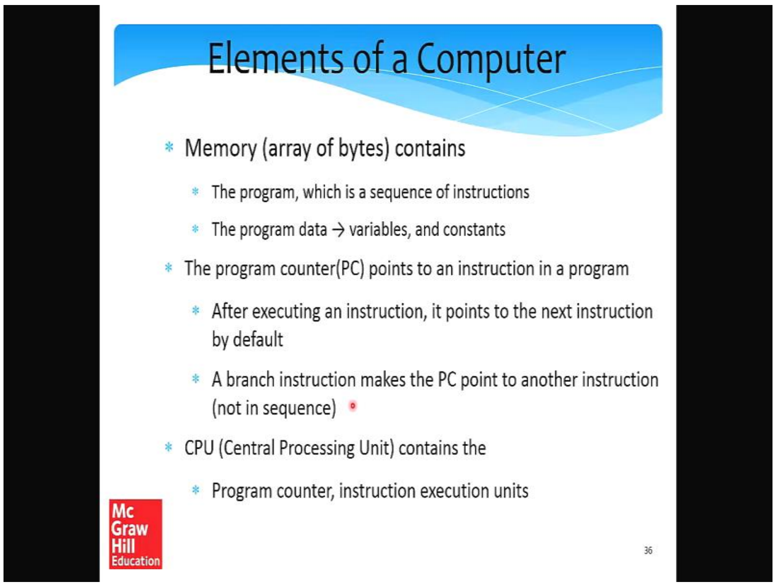
(Refer Slide Time: 45:21)


Computer Inspired from the Turing Machine

For all of those who skip the theoretical part welcome back. So, the computer that we have designed with, you know after getting inspired from a theoretical device called a Turing machine. So, for those who skip the discussion on theoretical devices, a Turing machine is a theoretical device, which inspired a practical device, the practical device being the modern computer. So, the modern computer has certain components. So, the components are the CPU, or the central processing unit, whose job is to read instructions interpret them and execute them. So, it has an arithmetic unit.

So, the arithmetic unit the job of that is to add, subtract, multiply, divide. It has a control unit to pretty much run programs with if statements and for loops. It has a program counter, which says which line of the program we are currently executing. So, a small animation over here, to visually show what a program counter is like. So, it points to a certain instruction that we are currently executing. We have a program, which is the physical set of instructions, in the instruction memory, and we have the data in the data memory.

(Refer Slide Time: 46:45)



Elements of a Computer

- Memory (array of bytes) contains
  - The program, which is a sequence of instructions
  - The program data → variables, and constants
- The program counter(PC) points to an instruction in a program
  - After executing an instruction, it points to the next instruction by default
  - A branch instruction makes the PC point to another instruction (not in sequence)
- CPU (Central Processing Unit) contains the
  - Program counter, instruction execution units

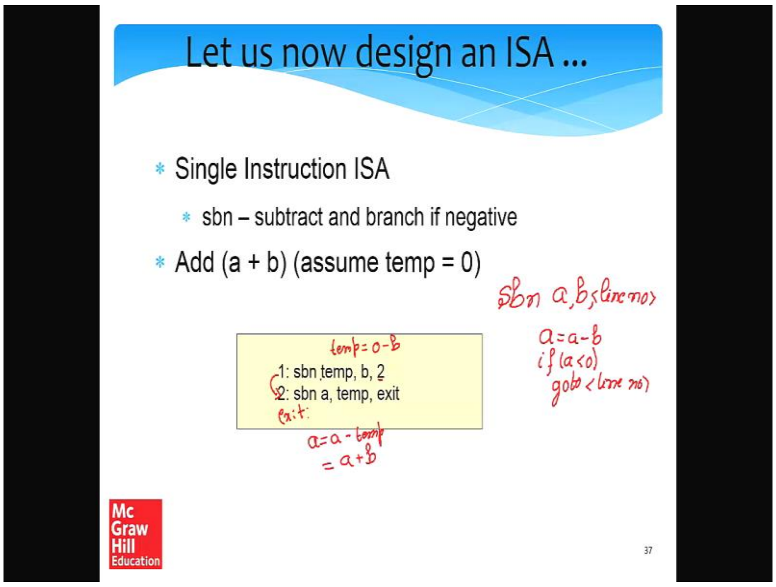
Mc Graw Hill Education

36

So, what again are the elements of a computer, let us just go through. So, the memory is an array of bytes. We will discuss what exactly the byte is in the next chapter, but we can think of it as a unit of information. So, the memory is a large array of bytes, in that it contains the program, which is a sequence of instructions, we can maybe divide the memory into two types, instruction memory and data memory. The program data which is the variables that you use, the constants that you use, the files that you work with, are all there in the program data memory.

So, this is the memory part. The program counter points to a given instruction in a program, after executing an instruction you move to the next instruction, and just in case you have if like in C E A of s statement in some languages have a go to statement. So, at that point the processor jumps to a new point in the program, and the program counter reflects that. Finally, you have the CPU or the central processing unit, which contains all the execution unit is and the program counter.

(Refer Slide Time: 48:01)


Let us now design an ISA ...

- Single Instruction ISA
  - sbn – subtract and branch if negative
- Add (a + b) (assume temp = 0)

```
1: sbn temp, b, 2
2: sbn a, temp, exit
exit:
```

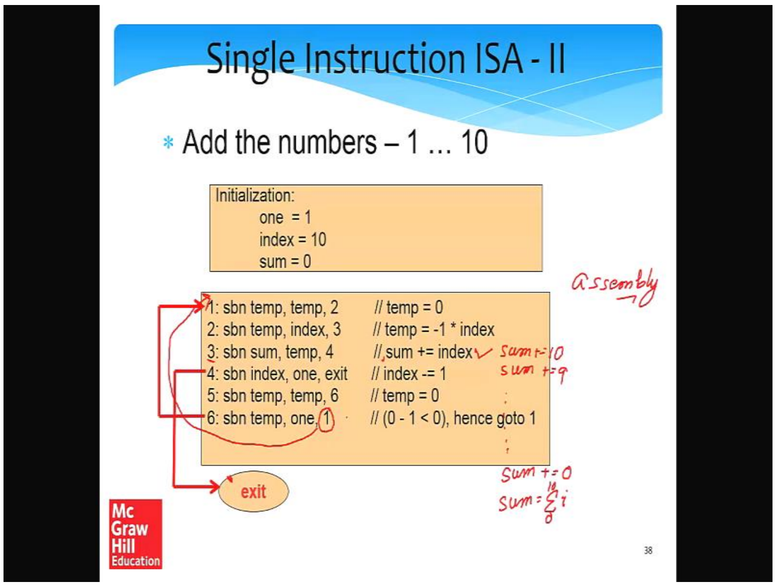
temp = 0 - b
a = a - temp
= a + b
sbn a, b, <line no>
a = a - b
if (a < 0)
goto <line no>
37

So, let us now design an ISA, which is Turing complete. For those who missed the theoretical part what this means is, that the ISA is equivalent to a Turing machine or it can be. In other words you can say that it can be used to implement all kinds of programs. So, surprisingly a single instruction just one instruction is Turing complete in a sense I can write a program with it. And in fact, I can write all kinds of programs with it. So, this instruction is sbn, subtract and branch is negative. So, let me maybe slightly digress and tell the readers what exactly an sbn is. So, consider that you want to add two numbers a plus b, and assume the variable temp initially has 0. So, the v that an sbn instruction works is as follows.

So, what an sbn does, is that it essentially subtract b from a. So, it compute a is equal to a minus b, and if the result is negative now, if the result is negative, it will go to the line number; that is mentioned over here; otherwise the program will go to the next instruction. So, in this case what we do is if we assume temp is 0. So, what we are essentially computing is temp is equal to 0 minus b. So, irrespective of the outcome we from instruction one we come to instruction two, and here again we compute a is equal to a minus temp, which is pretty much equal to a plus b.

So, in this case we are adding a plus b with the sbn instruction, and we can assume that the next instruction is exit. So, in both cases irrespected the result being positive or negative we just leave the program. So, the very surprising thing, which also can be proven, is that this simple instruction can be used to implement all kinds of programs, why would you want to or not want to do it let us discuss later.

(Refer Slide Time: 50:51)



Single Instruction ISA - II

* Add the numbers – 1 … 10

```
Initialization:
        one  = 1
        index = 10
        sum = 0
```

```
1: sbn temp, temp, 2        // temp = 0
2: sbn temp, index, 3       // temp = -1 * index
3: sbn sum, temp, 4         // sum += index      sum += 10
                                                 sum += 9
4: sbn index, one, exit     // index -= 1
5: sbn temp, temp, 6        // temp = 0
6: sbn temp, one, 1         // (0 - 1 < 0), hence goto 1
```

exit

assembly

sum += 0

$sum = \sum_{0}^{10} i$

38

So, let us discuss one more example which is to add the numbers from 1 to 10; a simple arithmetic se progression series is being added. So, let us assume that the following variables have the following values to begin with; one as the name suggests contains 1, index contains 10 and the sum is 0. So, that we would essentially write a program to add numbers from 1 to 10, is actually very easy in such kind of an instruction set. It is all that it requires is only 6 lines, and here is how it works.

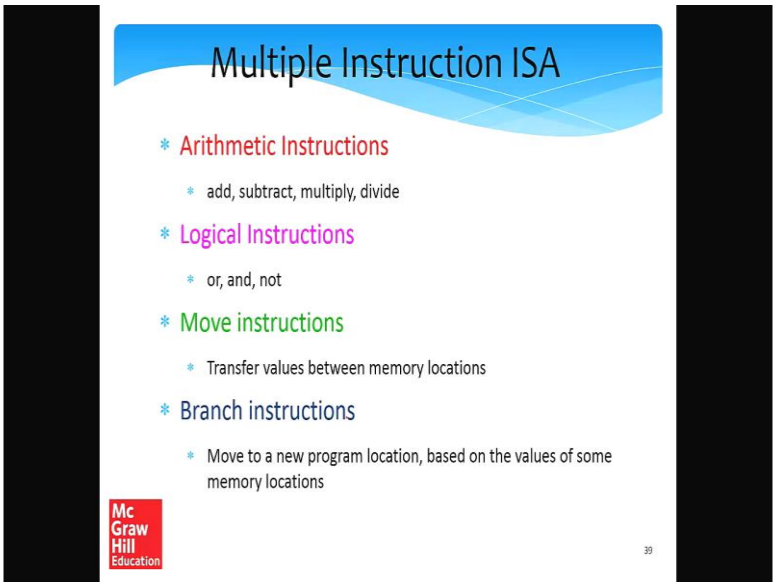
So, we take a variable temp and subtract it with itself. So, temp is equal to temp minus temp, the temp is equal to 0. Irrespective of the outcome we go to instruction two. There we compute temp is equal to 0 minus index or temp is minus 1 times index, then we come to the third line where we add some plus equal to the index. So, this is same as the addition trick that we showed in the last slide, we are essentially computing sum is equal to sum plus index, using these three lines, and mind you here irrespective of the outcome from 3 we go to 4.

Then what we do, is that we subtract 1 from the index, we are essentially adding some plus equal to 10, then some plus equal to 9, some plus equal to 8 and in that fashion. So, we subtract 1 from index, and if we see that it has become negative, means it is time to get out of the loop. So, we just jump and we come here to the exit point. Otherwise we do the same trick we set temp to 0, and we subtract 1 from temp. So, since the outcome is known it is minus 1, this essentially acts like a go to statement, where we go to statement number 1 and we jump from here to here. So, we have also implemented a for loop, where we loop for, we loop till the exit condition is satisfied.

Once the exit condition is satisfied we come out. So, what is the final outcome of this piece of code is that we are doing an addition in this line. See initially sum is 0. So, first we do sum plus equal to 10 then we decrement index. So, in the next iteration we do sum plus equal to 9. Similarly we just keep going, till we have sum plus equal to 0. So, ultimately sum contains the sum of all the numbers from 0 till 10, and this is exactly what we wanted to compute. So, we have written our first simple program, using a very simple instruction, using only one instruction, is being subtractive negative, and within 6 lines you have achieved quite a bit.

So, such kind of method or a style is also called low level programming, where instead of using any high level primitive, you have just used very simple machine instructions. So, this is also the first assembly program that we have written. So, what is an assembly program? It is well it is the kind of low level program that we just wrote, and as you can see it is very easy to actually simulate this, run this on a computer. So, we are all set when it comes to writing such simple instruction based programs.

(Refer Slide Time: 54:48)



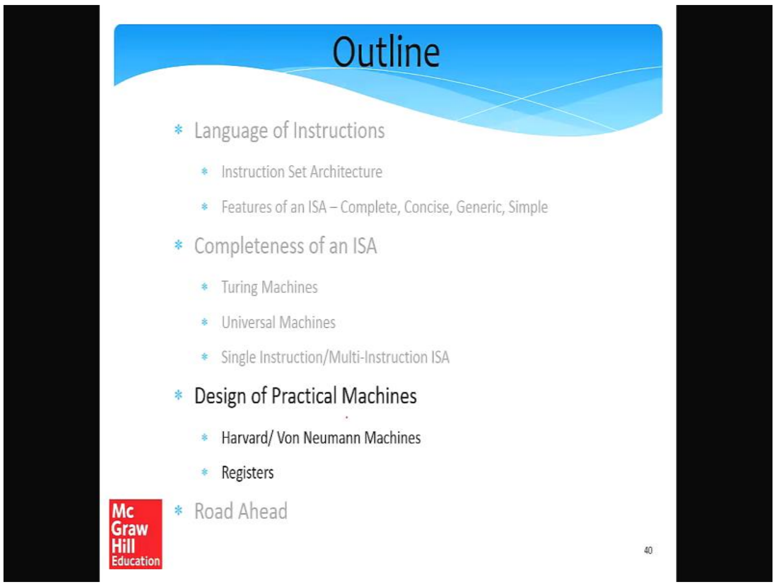
Multiple Instruction ISA

- Arithmetic Instructions
  - add, subtract, multiply, divide
- Logical Instructions
  - or, and, not
- Move instructions
  - Transfer values between memory locations
- Branch instructions
  - Move to a new program location, based on the values of some memory locations

39

Now, the question is that this exercise was partly theoretical also, in modern ISAs you will have multiple instructions. You will have separate instructions for add, subtract, multiply, divide, in our logical instructions for and or and not. You will have move instructions to transfer values between memory locations, and you will have branch instructions to move to new program locations; like implemented go to, as we did in the last slide, based on the values of memory locations. You will have all kinds of branch instructions to implement if statements, for loops, while loops since all.

(Refer Slide Time: 55:32)



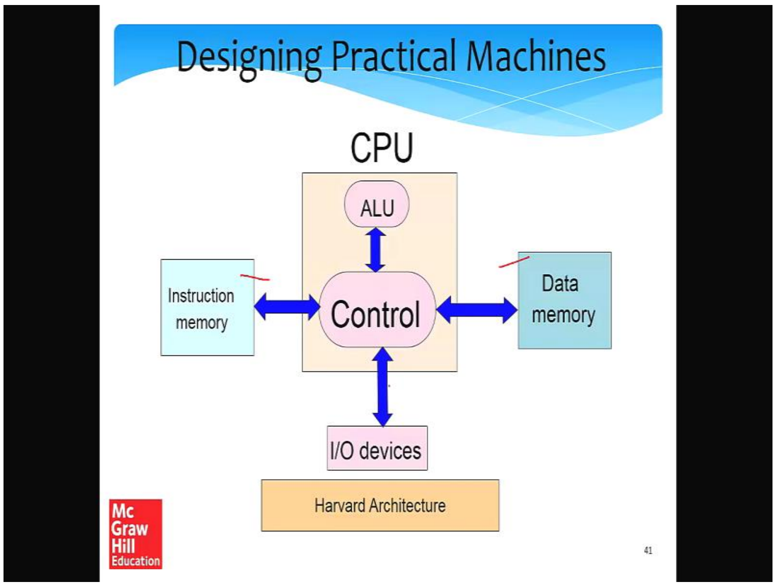
Outline

* Language of Instructions
  * Instruction Set Architecture
  * Features of an ISA – Complete, Concise, Generic, Simple
* Completeness of an ISA
  * Turing Machines
  * Universal Machines
  * Single Instruction/Multi-Instruction ISA
* Design of Practical Machines
  * Harvard/ Von Neumann Machines
  * Registers
* Road Ahead

40

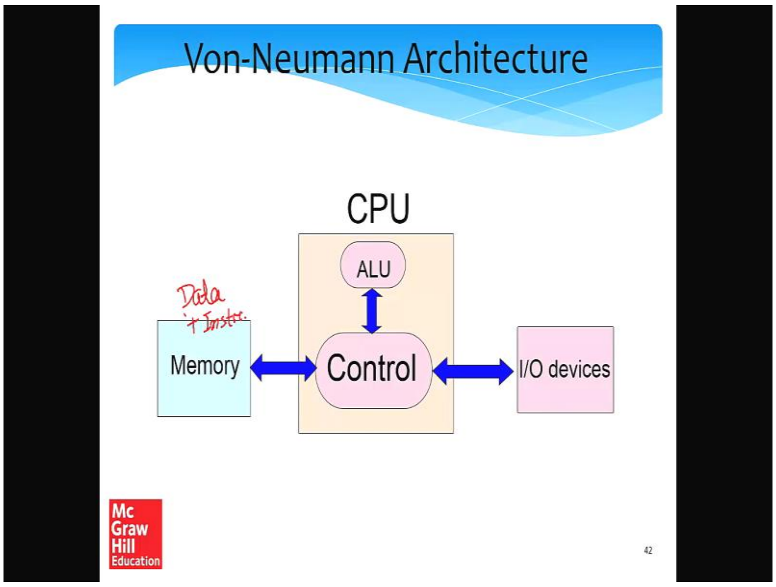
Finally we look at the design of practical machines. So, how exactly would we extend our theoretical results to practical results?

(Refer Slide Time: 55:43)



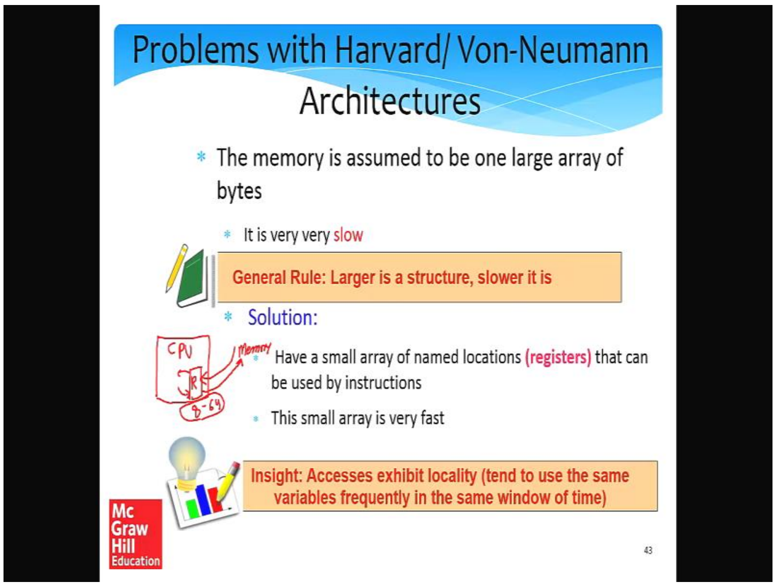
So, there are two kinds of machine categories. So, one practical machine is known as the Harvard architecture. So in this we have the CPU, we have a separate data memory and a separate instruction memory and once the results are computed they are communicated to input output I O devices such as a monitor or a printer, and we can see the results.

(Refer Slide Time: 56:06)



The other is a one Neumann architecture, where the data if it is, you know almost the same, it is just the data and instruction memory is fused into one that is all; that is the only difference.

(Refer Slide Time: 56:19)



Problems with Harvard/ Von-Neumann Architectures

- The memory is assumed to be one large array of bytes
  - It is very very slow

General Rule: Larger is a structure, slower it is

- Solution:
  - Have a small array of named locations (registers) that can be used by instructions
  - This small array is very fast

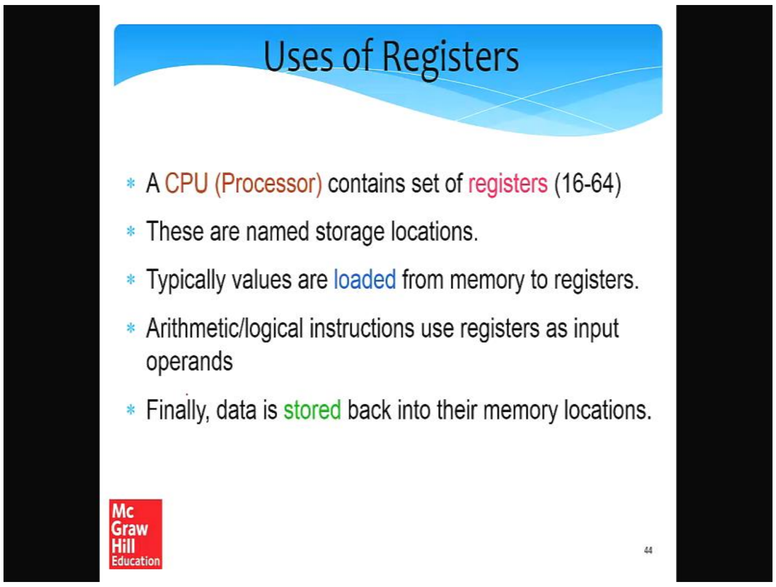
Insight: Accesses exhibit locality (tend to use the same variables frequently in the same window of time)

So, whatever we assumed up till now that the memory is assumed to be 1 large array of bytes. A byte is a unit of information, we will discuss it in detail in the next chapter, but if you assume such a large structure, the main problem that comes, is that larger is a structure slower it is. So, that is the reason I mean we need to introduce a small concept here, but we will have a lot of opportunities to discuss this in detail, in great detail. So, we want to introduce a small array of named locations, called registers, which are there inside the CPU itself.

So, it is very fast. So, our model of computing would be like this, that if you have the CPU, will be small array of registers inside it. So, most of the data will be coming from the array of registers, mainly because we 10d to use the same data again and again over the same window in time. And whenever a data is not there in registers we can read it from memory, put it in registers, work on it from register to register the very fast, and once we do not have any more space we can write it back to memory, and then again read something else from memory. So, memory is large and slow the set of registers are small and fast.

So, typically we will have somewhere between 8 to 64 registers, and this makes accessing the set of registers extremely fast and. The reason that this paradigm works, is because accesses exhibit locality, which means that you sort of 10 to use the same kind variables, same variables actually, frequently in the same window of time. You keep them in registers, you can very quickly access them, and you do not have to access a large and slow structure like memory.

(Refer Slide Time: 58:21)

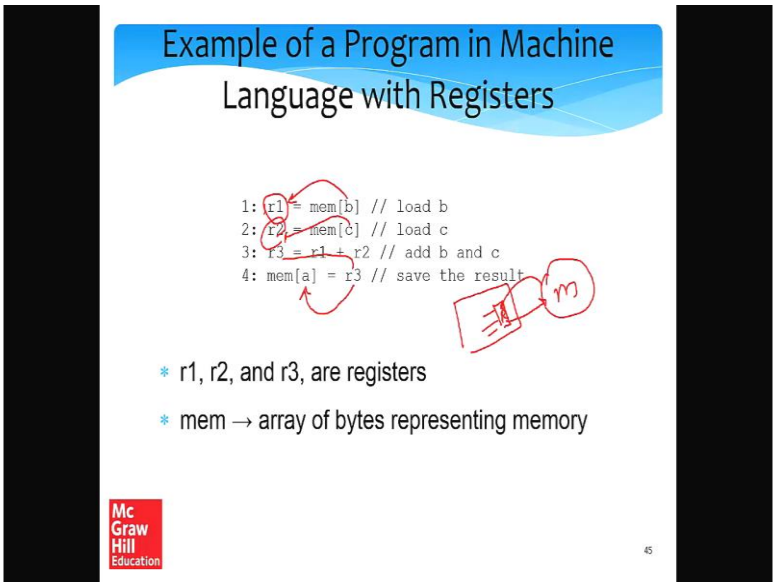


Uses of Registers

- A CPU (Processor) contains set of registers (16-64)
- These are named storage locations.
- Typically values are loaded from memory to registers.
- Arithmetic/logical instructions use registers as input operands
- Finally, data is stored back into their memory locations.

So, the users of registers of the name storage locations are as follows, that we read them from memory to the set of registers, using load instructions. We do whatever we want to do with them, with our arithmetic and logical operation set of processor supports.

(Refer Slide Time: 58:46)



Example of a Program in Machine Language with Registers

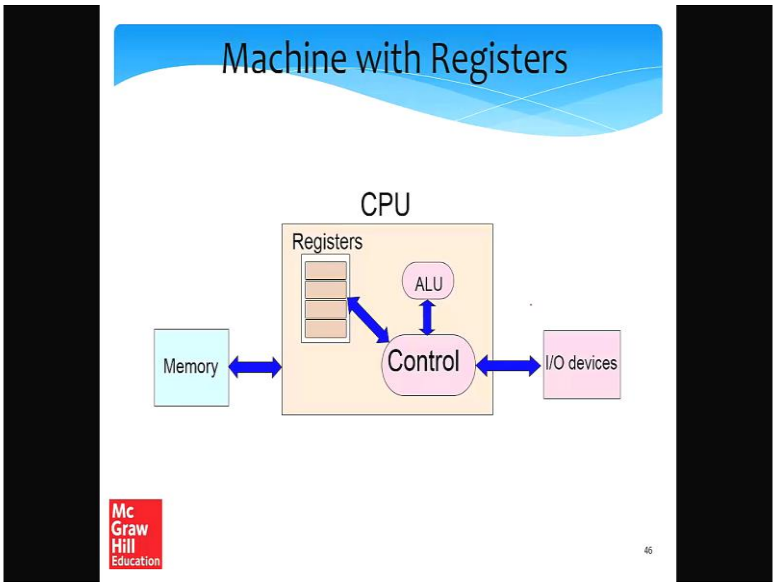
```
1: r1 = mem[b] // load b
2: r2 = mem[c] // load c
3: r3 = r1 + r2 // add b and c
4: mem[a] = r3 // save the result
```

- r1, r2, and r3, are registers
- mem → array of bytes representing memory

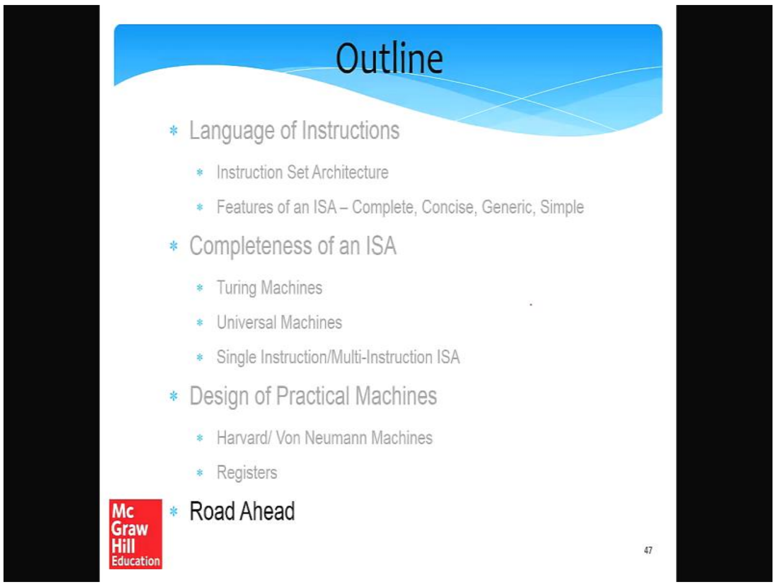
Finally data is stored back into their memory locations. So, it is a simple example over here. So, it is a tradition to write registers as r 1 r 2 r 3 and so on. So, here what you see is a register r 1 we are reading something from every location b into register r 1. We are reading something from memory location c, into register r 2. Then we are adding the contents of the two registers, saving it in r 3. And again we are saving the contents of r 3in memory.

So, of course, we can write a much bigger program where we read in more locations into more registers, and then do a lot of arithmetic operations on registers. So, this will ensure that programs run really fast, and just to recapitulate, if we have the CPU over here and the set of registers. The set of registers are very fast. So, whatever we want to read comes from memory to the registers, and we operate on the register that is typically the case, and then again when you want to get in something else from memory, we write back the value back to memory.

(Refer Slide Time: 59:58)



(Refer Slide Time: 60:10)



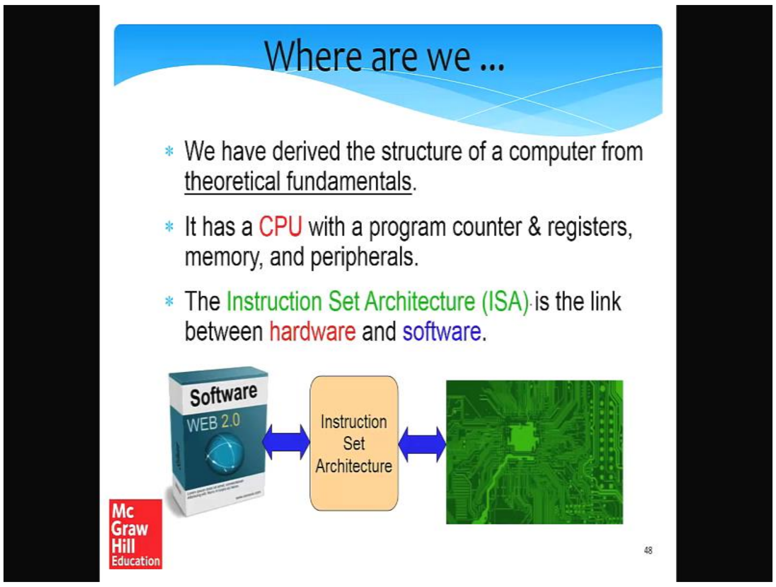
Outline

* Language of Instructions
  * Instruction Set Architecture
  * Features of an ISA – Complete, Concise, Generic, Simple
* Completeness of an ISA
  * Turing Machines
  * Universal Machines
  * Single Instruction/Multi-Instruction ISA
* Design of Practical Machines
  * Harvard/ Von Neumann Machines
  * Registers
* Road Ahead

47

So, what does a new machine look like right now? Well is the same one Neumann architecture only with registers inside the CPU.

(Refer Slide Time: 60:13)



Where are we ...

* We have derived the structure of a computer from theoretical fundamentals.
* It has a CPU with a program counter & registers, memory, and peripherals.
* The Instruction Set Architecture (ISA) is the link between hardware and software.

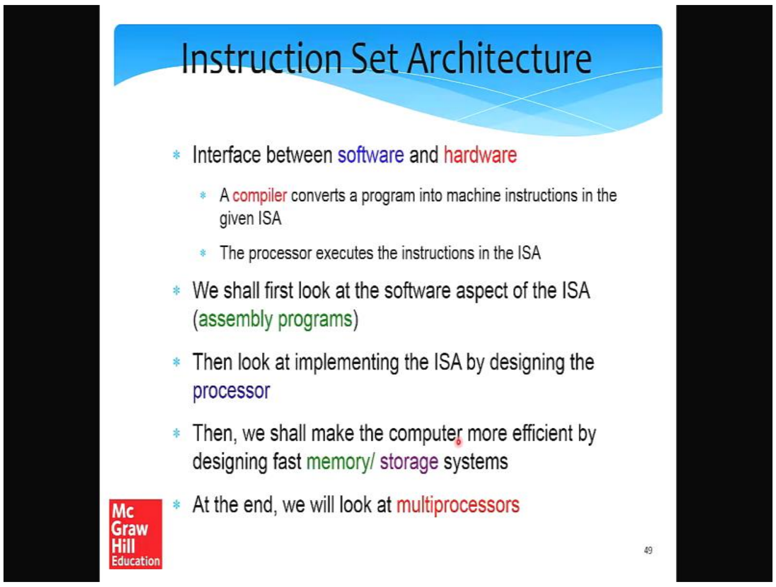
Software
WEB 2.0
Instruction Set Architecture

48

So, what is left the row ahead? So, what we have done in this small lecture is that we have derived this structure of a computer processing unit, from theoretical fundamentals, from extremely theoretical arguments we have sort of derived what a computer should look like. So, one is it needs to have a CPU which is the brain of the computer, which pretty much runs all the instructions. We need to tell, have a program counter to tell us where we are in the current program, which instruction we are executing. We need to have storage locations such as registers and memory. And finally, we need input output devices like mikes or keyboards or monitors printers.

So, we learned a lot about the instruction set architectures, which is basically the link between hardware and software, in the sense that software is written in c or C++ or java or some such language. The compiler converts it to the language that the processor can understand which is a set of instructions. Then the set of instructions are sent to the processor which is implemented in hardware; such that the results can be generated, and we display to the user via I O devices

(Refer Slide Time: 61:35)



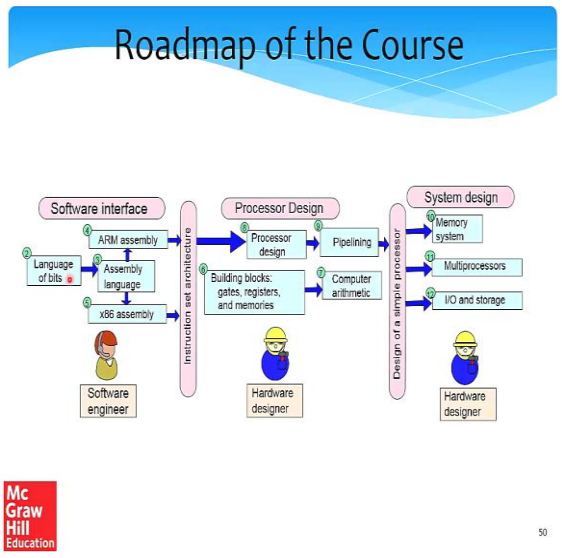
Instruction Set Architecture

- Interface between software and hardware
  - A compiler converts a program into machine instructions in the given ISA
  - The processor executes the instructions in the ISA
- We shall first look at the software aspect of the ISA (assembly programs)
- Then look at implementing the ISA by designing the processor
- Then, we shall make the computer more efficient by designing fast memory/ storage systems
- At the end, we will look at multiprocessors

49

So, once again the ISA is an interface between hardware and software. So, what we shall do is, we shall first look at the software aspect of the ISA, and how to write low level programs using processor instructions, called assembly programs. Then we shall look at how to implement the ISA by actually designing a processor from scratch. Then we look at making the computer more efficient, by designing fast memory and storage systems, and in chapter 11. It is not exactly the end, but one chapter before the end we will also look at this interesting field of multi process, where multiple processes are used to achieve a common objective or a common goal. So, we will look at all of that, during the course of this book.

(Refer Slide Time: 62:26)



So, what is the road map of the course? The road map of the course is simple, that we subsequently take a look at the language of bit is, which is pretty much what we can do, with bit is and bytes and how to represent information in 0’s and 1’s. We will study three kinds of low level assembly languages to program processers; one is ARM assembly, mainly for ARM processors used in phones. We look at a generic assembly language that we will design from scratch.

Then we shall look at x 86 assembly, the assembly language use in Intel and a m d processors. The middle half of the course we will look at processor design, where processor design looks at, well many things, we look start from the building blocks, gates registers and memories to look at how basic logic gates, and how registers are made how memories are made, we look at all of that from the circuit point of view. And then we will have slides on how to add subtract and multiply numbers computer arithmetic.

And then design processors in chapter 8 and 9. So, we will start with simpler processor designs and move into more complicated once. After designing a simple processor we look at designing a larger system, a more complicated memory system. A system with many processes multi-processing systems, and also integrate I O devices and storage devices. So, this is pretty much the overview of the book, which is divided into 12 chapters. We just covered chapter number 1. So, we have 11 more to go.

Thank you.