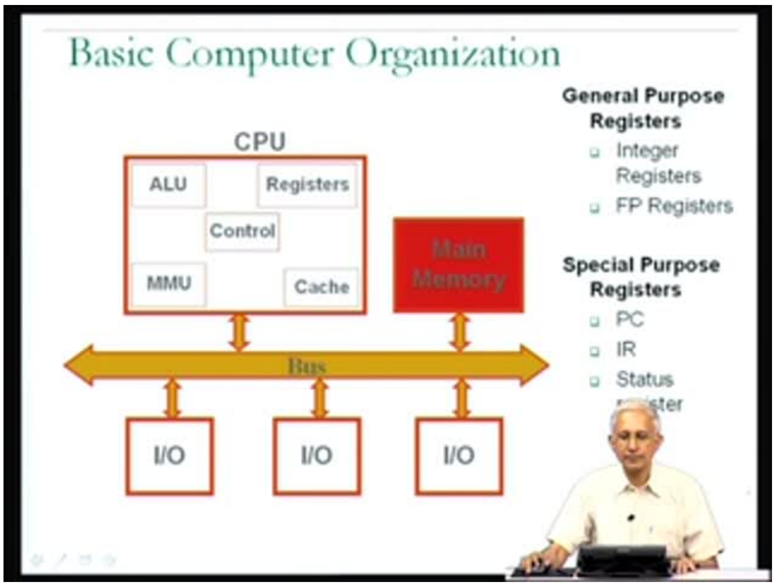
High Performance Computing

Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 02
Lecture No. # 04

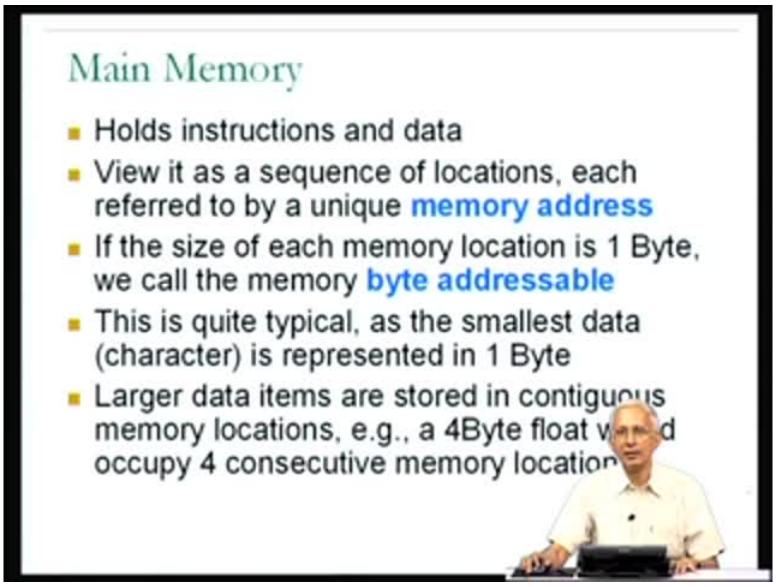
Welcome to lecture number four of the course on High Performance Computing. Let me remind you quickly, what happened in the previous few lectures, we have been looking at the basic organization of a computer, you remember that, there are three main parts to a computer - the CPU, the main memory and the IO devices.

(Refer Slide Time: 00:27)



We looked in the previous lecture a little bit at the CPU, understanding that there are many registers in which, data can be stored temporarily, while it is in use by the processor. We also looked at the main memory a little bit; we understand that main memory is capable of storing the entire program in execution, including both its instructions and its data. And in looking at main memory, we realize that, it is slower than the processor and therefore, that is why the general purpose registers are important, it is possible for the greatest the program to cause data to be copied from main memory into a register and therefore, accessible by the processor at much higher speeds, than if would have been in main memory.

(Refer Slide Time: 01:14)

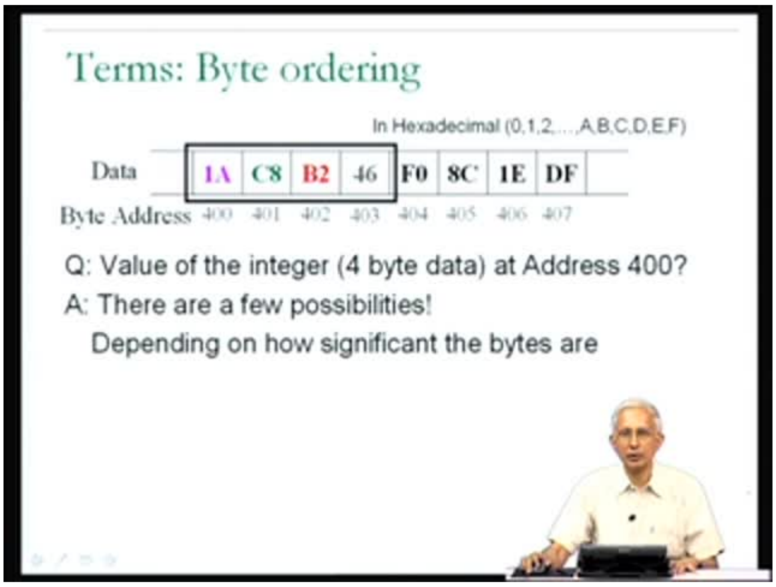


Main Memory

- Holds instructions and data
- View it as a sequence of locations, each referred to by a unique **memory address**
- If the size of each memory location is 1 Byte, we call the memory **byte addressable**
- This is quite typical, as the smallest data (character) is represented in 1 Byte
- Larger data items are stored in contiguous memory locations, e.g., a 4Byte float would occupy 4 consecutive memory locations

And as far as main memory is concerned, the key points we made last time, that it was both instructions and data and that individual elements in memory are refer to using addresses, which are unique integers associated with a different entities in memory.

(Refer Slide Time: 01:31)



Terms: Byte ordering

In Hexadecimal (0,1,2,...,A,B,C,D,E,F)

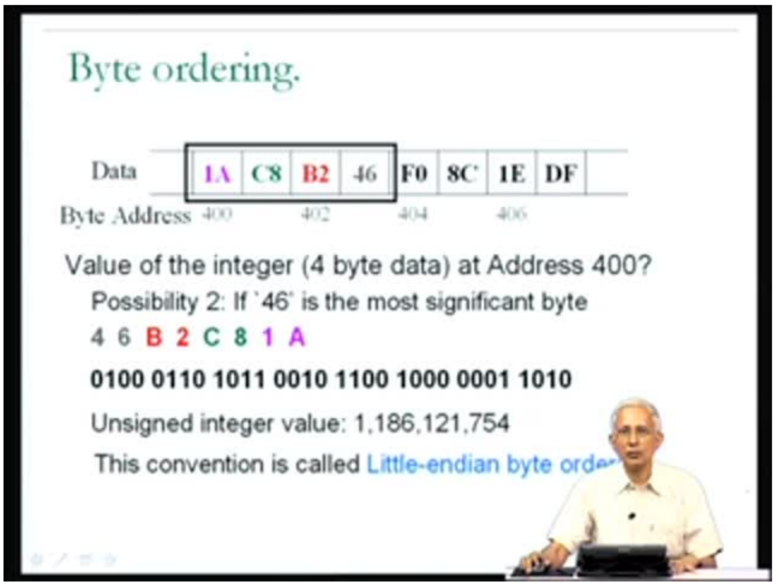
Data	1A	C8	B2	46	F0	8C	1E	DF
Byte Address	400	401	402	403	404	405	406	407

Q: Value of the integer (4 byte data) at Address 400?

A: There are a few possibilities!

Depending on how significant the bytes are

(Refer Slide Time: 01:38)


Byte ordering.

Data	1A	C8	B2	46	F0	8C	1E	DF
Byte Address	400		402		404		406	

Value of the integer (4 byte data) at Address 400?

Possibility 2: If '46' is the most significant byte

4 6 B 2 C 8 1 A

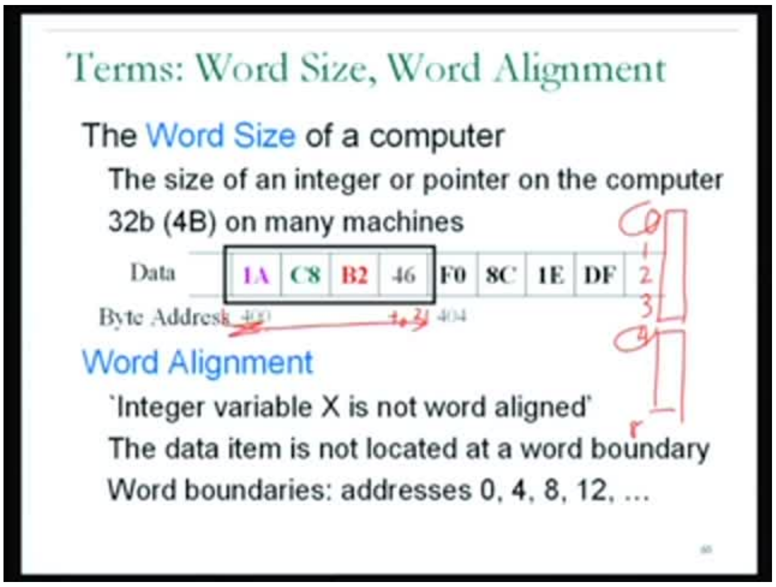
0100 0110 1011 0010 1100 1000 0001 1010

Unsigned integer value: 1,186,121,754

This convention is called Little-endian byte orde

An important concept in connection with addressing is what is called byte ordering? We understood that, there are typically two alternatives there are use - little-endian and big-endian byte ordering and this is of course, an important consideration.

(Refer Slide Time: 01:53)


Terms: Word Size, Word Alignment

The Word Size of a computer

The size of an integer or pointer on the computer

32b (4B) on many machines

Data	1A	C8	B2	46	F0	8C	1E	DF
Byte Address	400				404			

Word Alignment

'Integer variable X is not word aligned'

The data item is not located at a word boundary

Word boundaries: addresses 0, 4, 8, 12, ...

Today, will continue to look at main memory with one final consideration and that is, what is known as other than byte ordering, the next important consideration is what is known as a word size or word alignment?

Now, in hearing about computers, we often hear people talk about 32 bit computer or a 64 bit computer. And we understand that, both 32 bits and 64 bits are referring to the size of something and most specifically now, we will understand that, it actually referring to something known as the word size. So, the word size of a computer, gives us an idea about both the typical size of an integer or let say a pointer would be on that computer; and therefore, if you are told that you have a 32 bit computer, you would achieve we would understand, this means that the typical integers on the computer are stored in a 32 bit unit.

And similarly, the size of a pointer, like a pointer to structure or pointer to an integer, but also be of the same size. So, today we hear about 32 bit which means 4 byte machines we also hear, but 64 bit machines. And the thing to note here is that, as we learned in the case of byte ordering, if there is an element in memory which is more than 1 byte in size such as, in this diagram that we have on the screen the 4 byte entity 1A C8 B2 46, whatever it may be, then there is the possibility that it is a word, in other words in my C program, it might represent an integer variable let us say.

And the concept of word alignment then becomes important; know the concept of word alignment is quite simple. Sometimes, when you are compiling a program, you may get a warning, which says the integer variable X is not word aligned and may have wanted what this means. You notice I mention that, this is a warning that you may get, it may not be an error meaning, that is not a mistake in your program, but it say something that, you may want to change to get your program to run more effectively.

Now, the idea of an integer variable not being word aligned is that, it is address it is starting address may not be, what we call a word boundary. Now, if we look at the entity which has been darkened in the diagram above, if you will notice that, it is 4 bytes in size which is 32 bits in size and it seems to start at the address 400, going on until the address 403.

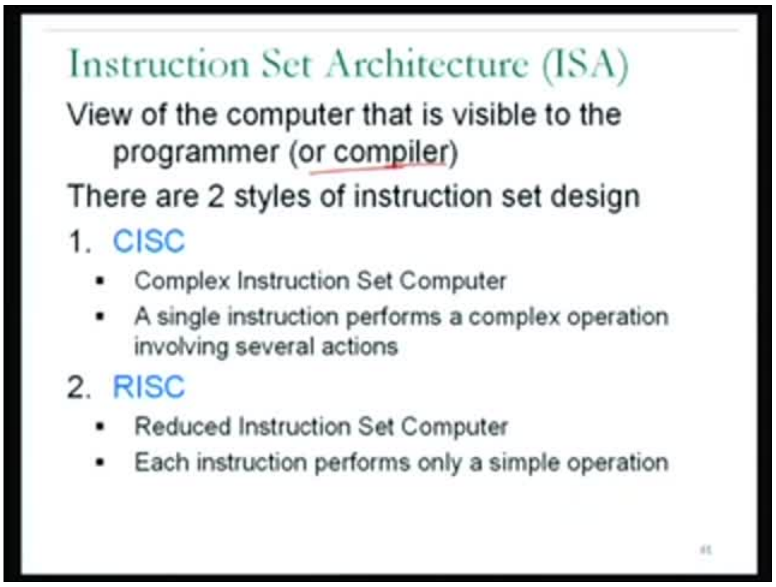
The key thing to note is that, it starts at the address 400 and therefore, the next entity in memory would start at the address 404. If you actually look back to the beginning of the first bytes in memory byte 0 1 2 3 etcetera, the concept to have in mind here is, that if I had put word size entities into each of the units of memory starting from byte 0 onwards, then the first word size entity would have started at address 0, the second word size entity would have started at address 4 and so on.

And I would then say that, these elements in memory are word aligned in that, the starting address of that 4 byte entity or the word size entity is itself a multiple of 4, from the perspective of having started from the beginning of memory and put only word size entities into memory.

So, in that sense the word boundaries among memory addresses would be 0 4 8 12 etcetera, all the multiples of 4. And that is why, the word alignment concept comes in, you say that in address is word aligned, on a machine which has a word size of 4 bytes; if the byte address is a multiple of 4. And this could have some performance connotations, you do not have any impact on the correctness of your program, it is not necessary that all your variables be word aligned for the correct execution of your program, but it may be important for the efficient fast execution of your program. So, I just want to make sure that we understood the concept of word alignment.

Now, with this we have starting idea both the various components of the computer system - the processor, the memory and we can now go ahead and so with this, we have the good understanding of the essential properties of the processor or CPU and the main memory.

(Refer Slide Time: 05:51)



Instruction Set Architecture (ISA)

View of the computer that is visible to the programmer (or compiler)

There are 2 styles of instruction set design

1. CISC
   - Complex Instruction Set Computer
   - A single instruction performs a complex operation involving several actions
2. RISC
   - Reduced Instruction Set Computer
   - Each instruction performs only a simple operation

And will now, take one step back and try to understand, the computer system from the perspective of the instructions, that computer system is capable of executing. And technically this is refers to as the instruction set architecture of the computer often a deviated as ISA.

So, in short the instructions set architecture of a computer is the view of the computer from the perspective of a programmer or to be more precise a compiler, why do I say it to be more precise, if you think about a most programmers, we use computer do not actually know the instructions, the low level instructions of the computer.

Since, the program in a high level language such as, Java or C or C plus plus, whatever it may be. And therefore, to say that the instruction set is the view of the computer to the programmer would be missing leading, most programmers do not view the computer in this way.

But when I write the C program, the C program gets complied; it gets converted into instructions of the kind that are understandable to the hardware. And therefore, the compiler itself must have a view of the computer which understands instructions. Hence, the second description which says that, the ISA is the view of the computer from perspective of the compiler, but again there are programmers to do program at this level and therefore, for those programmers this would be there perspective of the computer.

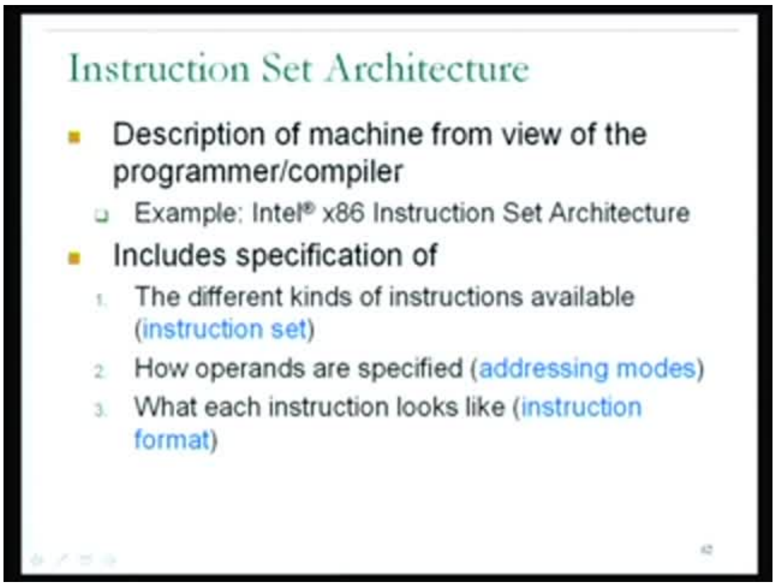
Now, typically for any computer processor or computer specifically for the processor, one will find the instruction set architecture completely described in a manual, that is available from either the manufacture of the processor of in the website, other websites. And this book typically we call the ISA manual or the instruction set architecture manual and typically it is a very large book because, it contains information about each of the instructions in various ways and all the information which might be needed by a person who's going to write a compiler or by a programmer, who want to use those instructions.

So, in general there are two kinds of computers from the perspective of the styles in which instruction sets could be designed and they are refer to as the CISC and the RISC styles of design.

Now, we are going to look at particular example, which is of the RISC style, but for completeness, I mean just tell you that, the CISC style, which is the Complex Instruction Set Computer style, as suppose to the RISC style, which is the Reduced Instruction Set Computer style. In the CISC style, you could have individual instructions which are somewhat complicated, hence the name complex instruction. So, in the individual instructions may actually do multiple operations whereas, in the reduced instruction set style of designing computers, instruction set would be made up of individual instructions each of its does only one somewhat simple operation.

And you might imagine that RISC computers being simple and having simple instructions may result in hardware they can do the simple instructions very fast whereas, the CISC computers having more complicated instructions the hardware's; obviously, going to be a little bit more complicated, as far as the individual instructions are concerned. And also from the perspective of the compiler designer, it may be easier to generate simple instructions to translate a C program into simple instructions. For these are another reasons the large number of the current processors use the RISC style, which is why we are going to concentrate on the RISC style.

(Refer Slide Time: 09:10)



Instruction Set Architecture

- Description of machine from view of the programmer/compiler
  - Example: Intel® x86 Instruction Set Architecture
- Includes specification of
  1. The different kinds of instructions available (instruction set)
  2. How operands are specified (addressing modes)
  3. What each instruction looks like (instruction format)

Now, to get a specific idea about the instruction set of a RISC style computer, we do need to look at a instruction set. So, let me first of all tell you, what you should expect to see, if you open an instruction set architecture manual, I am not suggesting that anyone do this, this as I said it is very large and very detail book.

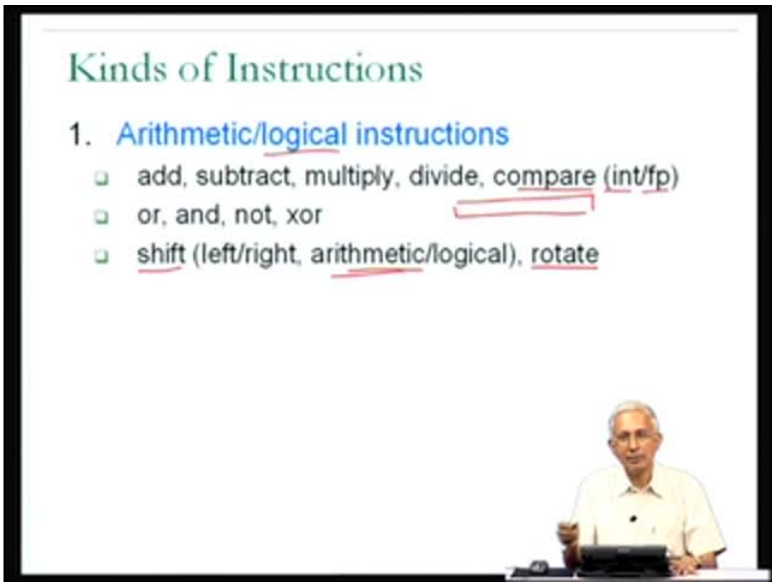
But, we do, it would be useful first to understand instructions at this level. So, I mention that instruction set architecture describes the machine from the perspective of the programmer or the compiler. And you may for example, have about of the Intel x86 architecture, many of the computers, laptops, desktop which you use actually implement the Intel x86 instruction set.

So, every instruction set has a name and one can get the manual is knowing the name of the instruction set of the manual will contain is a complete specification including three key components. The first key component is a description of what the different kinds of instructions provided by the instruction set are. And typically the instruction set architecture manual will contain one page for each of the instructions. So, if there are 200 different kind of instructions, they were be one page for instruction meaning 200 pages, just for that purpose. So, the first important component of the ISA manual would be the description of the individual instructions.

The second component is a specific description of how the operands to the instructions are to be specified. And the term for how operands are specified in technical term that is use for this is to refer to them as the addressing modes, So, we will try to understand a little bit about, what the different kinds of commonly used addressing modes are in this lecture.

Finally, the third important component of the ISA manual is going to be a specification of what each instruction looks like, in the binary form. In other words, it is what bit pattern is with correspond to each of the instructions and this is referred to as the instruction format. And once again, this would be clearly of great importance to the compiler writer, who has to write a program which can translate a C program which, you or I could read into a binary bit level machine language program which can be executed by a computer and obviously, the format of the invisible instructions in bits could be necessary to do this.

(Refer Slide Time: 11:20)


Kinds of Instructions

1. Arithmetic/logical instructions
   - add, subtract, multiply, divide, compare (int/fp)
   - or, and, not, xor
   - shift (left/right, arithmetic/logical), rotate

First, let us start of by trying to understand a little bit more about, what the different kinds of instructions might be included in an instruction set architecture.

Now, very clearly, the designer of an instruction set architecture would have to include all of the different instruction which, might be necessary for the difficult typical kinds of programs, that are written by users of the computer system. And therefore, depending on what application the computer system is likely to be used for the instruction set architecture could differ.

We will make some comments in this lecture about, the instruction sets of general purpose processors, which are meant for generic kinds of uses, not for specific uses. The specific use might be something like, processor which is design to be use inside a cell phone, that might be little bit different in a processor which is used to be inside a personal computer. The personal computer must have general capabilities. The processor inside a cell phone could have more specific capabilities and therefore might have a more finally, tuned instruction set.

But as far as, we have concern for the kinds of computers which we work on. We will expect that, there must be a good representation of arithmetic and logical instructions. In other words, instructions which can do arithmetic operations such as, add, subtract, multiply, divide, both on integers and on real values.

In addition, we might expect that, they should be instructions which can do logical operations. We will see a little bit about what logical operations are. So, the arithmetic instructions, we are quite comfortable with add, subtract, multiply, and divide but, I will also include one more in this and that is an instruction which can do comparison, to see if one integer value is less than, greater than are equal to another integer value, for example.

You might argue that, this could be done using the subtract instruction but, the danger of using the subtract instruction is that, when I subtract one integer from another, there is a possibility of an over flow. In other words, let us suppose that I have very large negative value and I subtract from it a very large positive value, then the result is going to be an even larger negative value, which might not fit in the bits available as per the number of bits available for representing negative to defined integers.

Therefore, rather than using subtract to compare two integers values, it is better to have a special instruction, which might be able to do the comparison through which one can do the comparison without doing a subtraction since, a subtraction might result in an error condition.

I would like to again point out is, there as for as arithmetic constructions are concerned, they may be a need to provide separate instructions for the different kinds of arithmetic have to be done. And integer arithmetic is different from real arithmetic, which is represented in computers typically by a floating point arithmetic, they would be separate hardware to do these two and therefore, they must be separate instructions for the purpose.

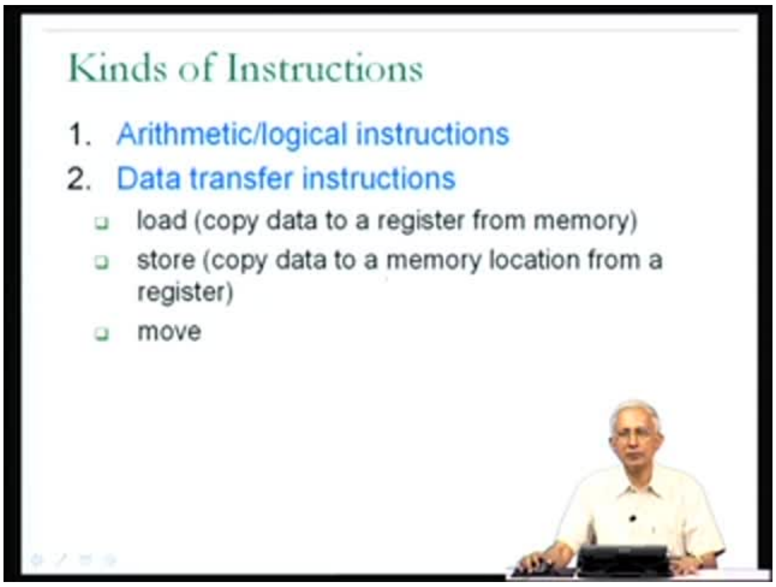
So, they could be two sets of arithmetic instructions for example, they could be add instruction for integers and separate add instruction for floating point values. Next, as I said they would be a need for logical instructions examples of logical operations or, and, not, an exclusive or, and those of you are familiar with these will realize that, they are operations at the bit level. So, the or operation between two bits and so on. So, almost all instructions sets provide these logical operations. So, with this you understand that most of the expressions inside the programs that, you write could get evaluated using the instructions provided in the arithmetic and logical instruction set.

In addition is often happens that is some other instructions are useful and that most instructions sets will contain instructions called shifts and rotates. Shift instruction is an instruction, which will take a value which might be stored in a register and cause the contents of the register to get shifted, either to the left or to the right, in other words the big pattern in the register, the least significant bits is the bit in the register will move to the most significant part of the register, thereby changing the value in the register.

So, the different kinds of shifts is might be implemented, some of which might have meaning as arithmetic operations and with some thought you will realize that, shifting the binary value integer value for example, to the left by one bit may be equivalent to multiplying it by 2 and therefore, they may be is notion of using shifts to implement some kind of arithmetic multiplication, division and so on. And also instructions which are called rotate in which, the bits after being shifted which, would otherwise fall of the most significant end could be rotated back to the least significant end.

So, instructions of this kind, which might actually be used to do arithmetic but, which may otherwise be used for the simple purposes of shifting data to the left or to the right or rotating data within a register.

(Refer Slide Time: 16:10)



Kinds of Instructions

1. Arithmetic/logical instructions
2. Data transfer instructions
   - load (copy data to a register from memory)
   - store (copy data to a memory location from a register)
   - move

So, this is a typical compliment of arithmetic and logical instructions, now another class of instructions which, is clearly necessary for the kinds of processors that, we are talking about and instructions, which can be used to transfer data from one place to another.

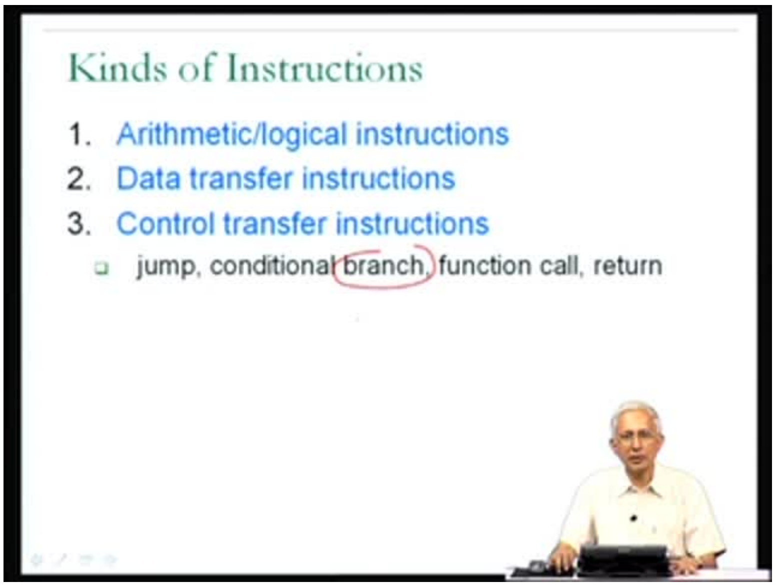
Now, we know that the data, which the program deals with is going to start off, when your program starts executing with a data all being present in main memory, but there are vocationally in order to improve the execution time of your program, you may want to transfer some of the data for example, the variable x the value associated with the variable x, you may want to transfer it from main memory into a register, to make the access to that value even faster. So, must be instructions for this kind of a data transfer to take place.

So, we will talk about 2 or 3 different kinds of data transfer. The first is, what is typically known as a load and instruction, which can be used to copy data from main memory into a register. So, one can talk about loading the variable x from memory into register R1. They must be also the converse data transfer, where one could copy a data value from a register into main memory and that is typically refer to as a store. So, they could be stored instructions in an instruction set to copy data from a register into a main into a main memory location.

And finally, they could be instructions which are generating known as move, which could be used for example, to copy data from one register to another register, of one memory location to another memory location and some instructions sets may contain move instructions. So, move is a more generic term, load will typically refer to an instruction, they can be used to copy data from main memory into a register and similarly, store will typically refer to instruction they can be used to copy data from a register in the CPU into main memory.

So, in any processor which has registers general purpose registers, it would be necessary to have instructions of these kinds and these instructions are refer to as data transfer instructions. So, that is two classes of instructions, we expect to find in our general purpose instruction sets.

(Refer Slide Time: 18:14)


Kinds of Instructions

1. Arithmetic/logical instructions
2. Data transfer instructions
3. Control transfer instructions
   - jump, conditional branch, function call, return

Now, a third if you think about a little bit would definitely have to be control transfer instructions, there will be situations in a program where, I am not happy with a sequential execution of first executing the first instruction, then executing the second instruction, then executing the third instruction and so on. And this, I have some kind of instructions which can be used, for example to create a loop or to create an if then else, my programs will be quite uninteresting and it would not even be possible to easily compile C programs, which can contain constructs like loops, repeat loops, while loops or if than else into machine instructions.

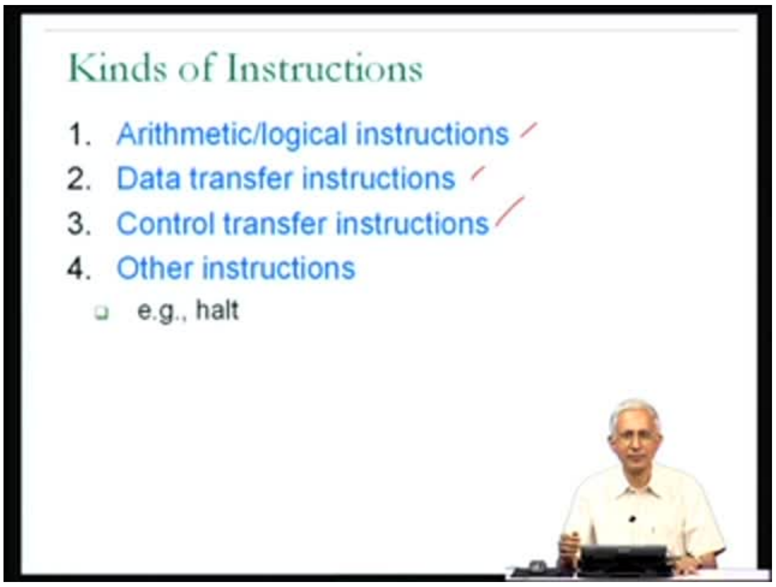
So, the control transfer instructions are primarily, there control to change from the default pattern of executing instructions in an order, first, then the second, then the third within the program to more interesting kinds of execution pattern based on a high level constructs, that I just refer to if then else is repeat while etcetera.

So, what kind of control transfer instructions could one find in instruction sets? Answer is there are several kinds that one finds and they typically go by name such as, jump, conditional branch. Function call, if you think about the event of a function call you realize that two is the control transfer and return in other words, the return from a function call to the point in the program where it was called.

So, most often when one refers to a jump, one is referring to control transfer instruction, which is not conditional, which is why the term conditional branch or just branch. So, the conditional branch is often refer to just as branch, is meant for uses where condition will be specified within the instruction, if the condition is true, then the control transfer will take place whereas, if the condition is falls the control transfer will not take place and instead of transferring control to that specified instruction, the next instruction in the program will simply be executed.

So, your typical instruction set will have a variety of different kinds of instructions for different purposes (()). If the jump might be very useful to implement something like a goto, the conditional branch would be very useful for, if then else is, for repeat loops, for while loops etcetera, considerably. The function call would be essential for the implementation of function calls and the return from a function call to the point of call.

(Refer Slide Time: 20:38)



Kinds of Instructions

1. Arithmetic/logical instructions
2. Data transfer instructions
3. Control transfer instructions
4. Other instructions
   - e.g., halt

Now, there are other instructions set that, one can expect to find in an instruction set and I will sort of generately classify them as other instructions, there are functionality that, we are used to using in programs actually typically false, how can be satisfied by the first three classes of instructions. But, there are other things which must be included in instruction set, for reasons their might not be; obviously, this point. But let me, just give you one example, there may a need for an instruction, they can be executed for the graceful shutting down of a computer.

Here, I am not referring to an instruction that is going to be used for the end of the execution of a program that is a must simpler event. Here, I am specifically referring to what an instruction that may have to be executed for the powering down of the computer, that instruction might be call the halt instruction and that is very clearly or very special kind of an instruction, which are typical user is not going to use in a program. If I am writing a program, it is very unlikely that, I will have to use the halt instructions since unlikely that, I will have to shut down the computer as part of the execution of my program.

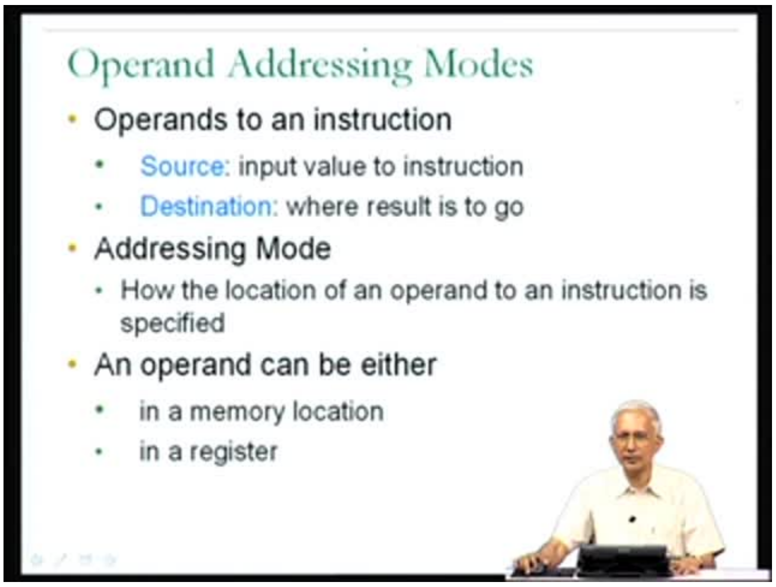
Whereas some more important programs, which handle the management of the computer as a whole may have to use instructions of this kind and they could be several other instructions of this kind, which I am not meant for the usual programmer, but I meant for the privilege program, the very special programs they control the computer system on as a whole.

So, there are many different kinds of instructions and they are typically classified into these four classes, please do bear in mind, that in the arithmetic and logical instructions they could be sub classes, such as the arithmetic instructions which are of integer type the arithmetic instructions which are of 14 point type.

Now, with this we have an understanding of the first important part of an instruction set architecture description; let us move to the second. I will indicated that the second important part, they are three components one finds an instruction set architecture manual or description.

The first is the complete specification of all the instructions; the second is the addressing modes, so let us look at addressing modes next.

(Refer Slide Time: 22:42)



Operand Addressing Modes

- Operands to an instruction
  - Source: input value to instruction
  - Destination: where result is to go
- Addressing Mode
  - How the location of an operand to an instruction is specified
- An operand can be either
  - in a memory location
  - in a register

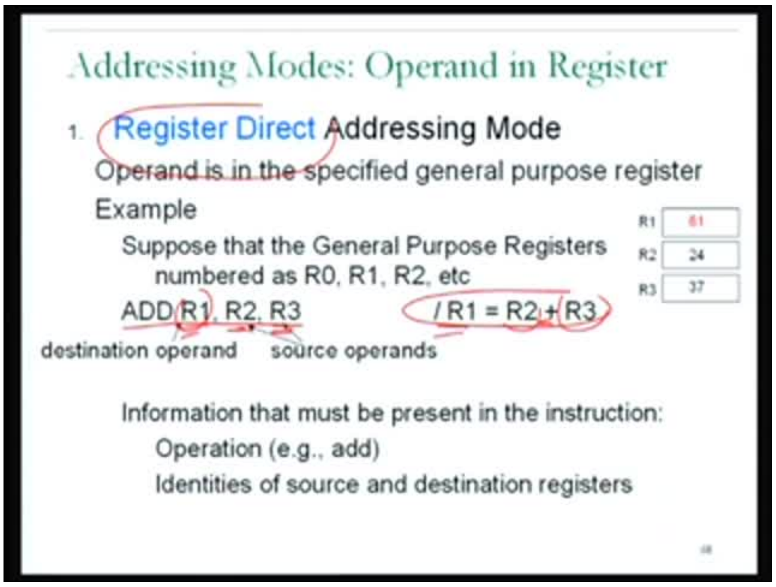
So, here in the slide we talk about the operand addressing modes and the question that, one should first ask is, where can the different operands to an instruction come from, so I have an add instruction.

What are the different possibilities as far as, where it is instruction operands could come from. Little bit of thought, you realize that, any add instruction is going to have two kinds of operands is going to have its source operands, which are input values to the instruction. The word for the input values to the instruction, the technical term is to refer to them as the source operands and then this going to be an another operand, which is the place or the destination of the result of the add and that would be refer to as destination operand.

So, your typical add instruction may have two source operands and one destination operand. And the question of how the operands are specified is what is taking into account or indicated by the instruction set architectures addressing modes. Now, where could the operands, where could the source operands be? If you think about it a little bit you realize that, the source operands to an instruction, could either be in memory or they could be in registers therefore, we are going to have different addressing modes for operands, which are in memory in a separate addressing modes for operands, which are in registers.

So, there going be different kinds of addressing modes, which is why we going to treat the topic of addressing modes, as much more detail than we look at the topic of instructions. Because, it is more important really critical first to understand, but the different kinds of addressing modes are, it is a much more certain point as far as the different kinds of instructions were concerned for us people, who have program before into fairly obvious that, they had be arithmetic logical etcetera the different kinds of instructions, where the whole concept of addressing modes may appear to be quite new to many of us.

(Refer Slide Time: 24:24)



Addressing Modes: Operand in Register

1. Register Direct Addressing Mode

Operand is in the specified general purpose register

Example

Suppose that the General Purpose Registers numbered as R0, R1, R2, etc

ADD R1, R2, R3 / R1 = R2 + R3

destination operand source operands

Information that must be present in the instruction:

Operation (e.g., add)

Identities of source and destination registers

So, let us start by trying to understand some of the addressing modes, which are used for operands to instructions, where the operands have within a register. Now, the simplest possible case is where the operand value is containing within one of the general purpose registers of the CPU. And in this case the addressing mode is refer to as the register direct addressing mode in suggesting that, the operand value is directly available within one of the general purpose registers of the CPU and therefore, within the instruction one will expect to find the identity of that register and nothing else is needed. So, ensure the in the case of the register direct addressing mode, the operand value is within the specified general purpose register.

Let us look at an example, of instruction which uses the register direct addressing mode. Now, for this and the subsequent examples that am going to use in today's lecture and in fact, for lectures to come, I am going to assume that the general purpose registers of the computer that, we are talking about or refer to a numbered as R0 R1 R2 etcetera.

Now, within the binary form of the instruction, they may not be represented by the R just the 0 the 1 and the 2 will suffice, but since, we are human beings and we would like to be more comfortable with notation that we use at least in a lecture, I will use R 0 to represent the general purpose register R 0 etcetera.

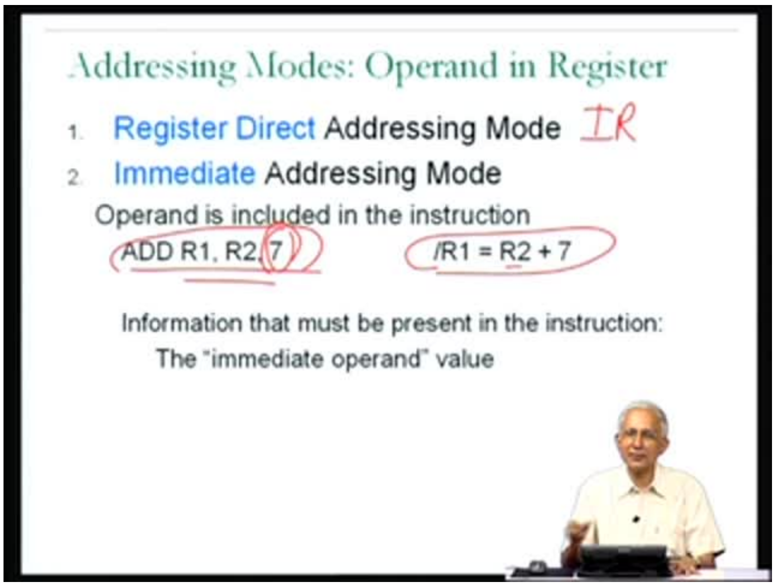
So, let us look at this instruction, over here I have shown you the instruction on the left and on the right is a comment. So, here we have an add instruction add R1 R2 R3 and the way you should interpret this instruction; and this is an instruction which does an addition, most specifically it is source operands are R2 and R3 and its destination operand is R1. So, that is convention that, we are going to use in the lectures to follow, I will put the destination operand first and the source operand subsequent to that.

So, what is this instruction actual telling me? It is telling me that, there are two source operands to the add instruction. The first source operand value is contained within general purpose register R2, the second source operand value is contain within general purpose register R3 of the CPU, so these two values will have to be taken, they will have to be added by the ALU and after this result should go into general purpose register R1 of the CPU. And in this case all the three operands are actually specified in this register direct addressing mode. So, just make sure we understand this, two source operands are R2 and R3 the destination operand is R1.

Let us suppose I had a situation, where before this instruction executed, if I had look into the register, I found that register R1 contain the value 17, register R2 contain the value 24 and register R3 contain the value 37, then what this instruction is asking is that, the value with in R2, in other words 24 is added to value within R3, in other words 37, the sum of 24 and 37 is 61, to after the addition has been done, this instruction is requesting that the value 61 go into register R1; and therefore, the effect of this instruction would be that R1 will have a new value of 61. And that is all there is to understand about the register direct addressing mode, it is the way that an instruction can specify an operand, which is currently available within one of the general purpose registers, very clearly the program what I have to load the value from its main memory location into that register by some previous instruction.

So, register direct is the simplest addressing mode. So, what information must be present within the instruction for an operand, which is specified in register direct addressing mode, very clearly the identity, I mean other than the operation, which the instruction is meant to do, such as the fact that this particular instruction is meant to do the add operation, the identities of which source and destination registers.

(Refer Slide Time: 28:23)



Addressing Modes: Operand in Register

1. Register Direct Addressing Mode IR
2. Immediate Addressing Mode

Operand is included in the instruction

ADD R1, R2, 7 /R1 = R2 + 7

Information that must be present in the instruction:
The "immediate operand" value

Therefore, in this case the two source registers are R2 and R3, the destination registers is R1 therefore, the identities of R2 R3 and R1 would have to be present within the instruction in the case of the register direct addressing mode. Now, that was nice and simple. Let us look at a slightly more complicated addressing mode, which deals with operands, which are in registers. Now in fact, I can think of only one more, which is generally likely to be available and that is the immediate addressing mode.

Now, the ideas of immediate addressing mode as a name sort of suggest is that, the operand value is included within the instruction itself. The operand value is not present in one of the general purpose registers of the processor, but rather it is present within the instruction itself and I will come back to that, why I have included the immediate addressing mode in this particular slide a little bit later, let us look at an example.

So, here I have an instruction add R1 R2 followed by the number 7 and the interpretation, the meaning, which I provided in this comment, is that the value within R2 is to be added to the value 7 and result of the addition is to go into register R1.

So, one of the operands to the add instruction in this case is the value 7 and if you look at the instruction, you see the value of the operand present within the instruction in what is known as the immediate addressing mode. And just to run through this example, very clearly if R2 contain the value 24 prior to this instruction, after 24 and 7 have been added the result is 31, 31 would be the new value of R1.

Now, what information must be present in the instruction very clearly, the immediate value seven must be present within the instruction, that is the meaning of an instruction of this kind or you may be wondering, why did he include immediate addressing mode on a slide and title addressing modes operand in register, the operand 7 is not in a general purpose register.

So, you may be wondering, why I included this addressing mode on the slide. If you think about it little bit, you realize that, when this instruction is being executed, the instruction as a whole is present within the CPU and in fact, it is present in one of the special purpose registers of the CPU known as the instruction register.

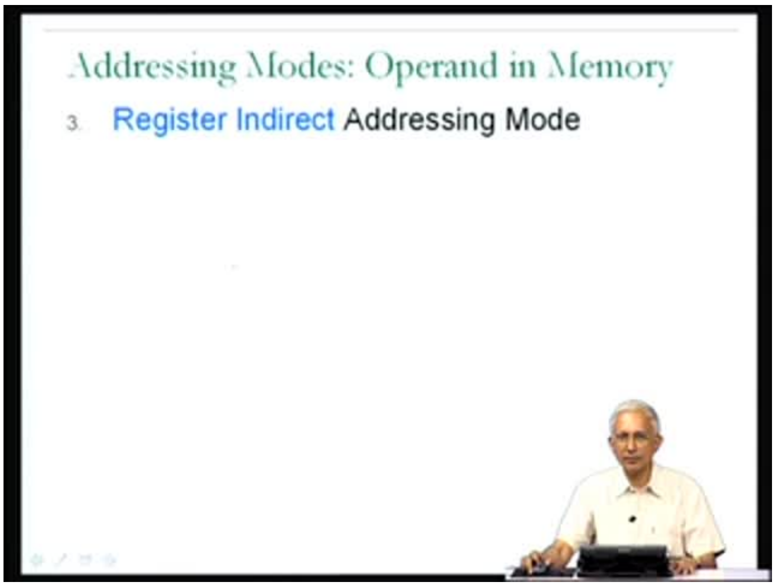
It is only, if an instruction is present within the CPU specifically inside the instruction register, that it can be executed instruction had to be fetch has to be fetch from main memory into instruction register in order to be executed.

Therefore, when this instruction is executed it is present inside the instruction register and therefore, the value 7 is also present inside the instruction register within the CPU, which is why I have included it in the slide. It is the situation that, the operand value is present in a register, it is happens to be the instruction register.

So, these are typically the two kinds of addressing modes relating to operands being present within registers. The first case that the case whether, operand is present in a named general purpose register. The second the immediate addressing mode by the operand is present within the instruction register having been specified within the instruction.

Now, the bulk of the addressing modes, which we are going to study from this point on relate to operands, which are not present within the CPU cannot present in any register, but rather the operands are present in main memory. And there are many different mechanisms, in other words many different addressing modes, which could be used to specify operands the location of an operand in main memory.

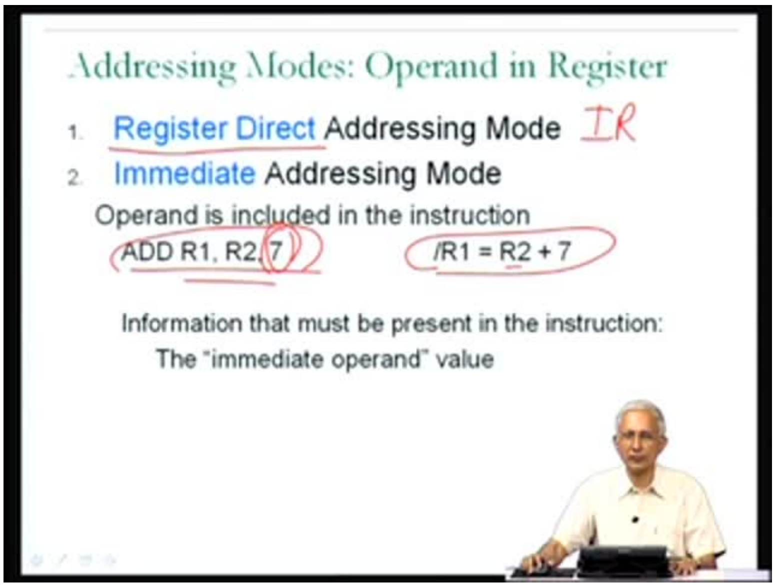
(Refer Slide Time: 31:48)



Addressing Modes: Operand in Memory

3. Register Indirect Addressing Mode

(Refer Slide Time: 31:56)



Addressing Modes: Operand in Register

1. Register Direct Addressing Mode IR
2. Immediate Addressing Mode

Operand is included in the instruction

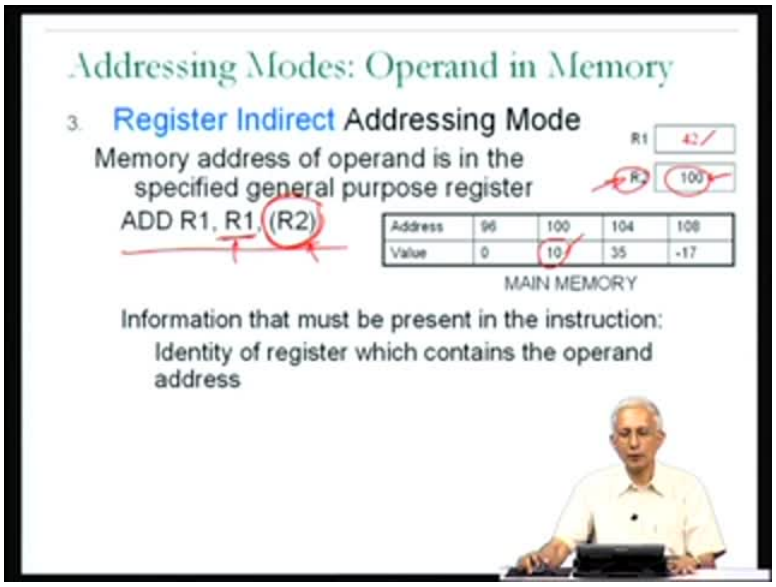
ADD R1, R2, 7 /R1 = R2 + 7

Information that must be present in the instruction:

The "immediate operand" value

Now, the first, which I will talk about is something known as the register indirect addressing mode. I will go back to the previous slide, the addressing mode, which we had there was called the register direct addressing mode.

(Refer Slide Time: 32:02)



Addressing Modes: Operand in Memory

3. Register Indirect Addressing Mode

Memory address of operand is in the specified general purpose register

ADD R1, R1, (R2)

R1 42

R2 100

Address	96	100	104	108
Value	0	10	35	-17

MAIN MEMORY

Information that must be present in the instruction:

Identity of register which contains the operand address

So, this is a different name, the register indirect addressing mode. The ideas are register indirect addressing mode is that, the operand is present somewhere in memory and specifically the memory address of the operand is specified within a general purpose register that is why it is known as register indirect.

One of the general purpose registers will indirectly tell you the operand value, because that general purpose register will contain the address in memory of that operand value. Let us look at some examples of this one example should suffice. So, here again I have use an add instruction this add instruction is adding 2 or 1 and operand from memory, which is specified in register indirect addressing mode.

So, the register indirect addressing operand in this case is, shown with R2 with in brackets, that is in option used convention in instructions set notation in many assembly languages, when they show general purpose register and put parenthesis around it, that is an indication that, they are talking about an operand, which is specified in register indirect addressing mode whereas, operands, which are specified in register direct addressing mode may just be represented by the identity of the register. So, this particular instruction actually has one destination operand and two source operands, one of the source operands is specified in register direct addressing mode, the other source operand is specified in register indirect addressing mode. So, the first operand is coming out of register R1, the value is present in register R1, the second source operand value is coming out of memory, where in memory that is what the register indirect addressing mode tells the processor.

So, to get an example of this we need a more thorough picture, we have to know the current contains of register R1, the current contains of register R2, but we will also have to know the current contains of various memory locations, which is why if at this particular example, I show you some memory locations in terms of the different addresses and some of the values relevant for this particular example.

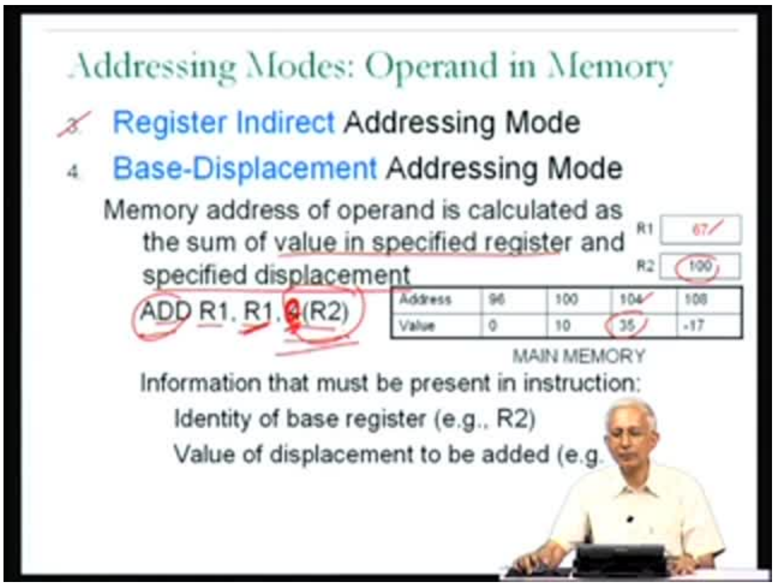
So, let suppose, that I have this instruction and I have a situation, where R1 currently contains a value 32, R2 currently contains a value 100.

So, what done is the meaning of this, the first operand value is 32, the second operand is containing within the memory location specified by register R2, in other words it is contain within memory location 100, which means that the second operand value is 10.

So, that the result of the addition is going to be to add 32 to 10 yielding 42, which should be the new value within register R1. So, the register indirect addressing mode is a simple example of an addressing mode use to specify the location in memory of an operand, in this case the operand value was 10 and it was specified in register indirect addressing mode by indicating that, register R2 contains the address of the operand term.

So, what information must be present within the instruction, if an operand is specified in register indirect addressing mode. Simply, the identity of the register which contains the operand address, in this case R2 and that is why we saw in the example, the identity of R2 have to be specified within the instruction for an operand specified in register indirect addressing mode in this way.

(Refer Slide Time: 35:20)



Addressing Modes: Operand in Memory

3. Register Indirect Addressing Mode
4. Base-Displacement Addressing Mode

Memory address of operand is calculated as the sum of value in specified register and specified displacement

ADD R1, R1, 4(R2)

R1	67
R2	100

Address	96	100	104	108
Value	0	10	35	-17

MAIN MEMORY

Information that must be present in instruction:

Identity of base register (e.g., R2)

Value of displacement to be added (e.g.

Moving right along, we could have more complicated addressing modes. Let me show, you an example minor extension to register indirect, which adds to its functionality substantially. Now, this addressing mode, which I am going to talk about next is often refer to as the base displacement addressing mode, as I said it is a minor extension on the register indirect addressing mode.

Now, what happens in the case of base displacement addressing mode is that, the memory address of the operand has to be calculated and it is calculated as the sum of the value in the specified register, along with a specified displacement. Recall that in the case of register indirect addressing mode, the memory address of the operand was the value in the specified register, we did not have the additional computation of a sum with a specified displacement that is why, I said this is an extension of the register indirect addressing mode.

Moving right along let us look at an example, the again an add instruction add R1, R1, R1 is the destination it specified in register direct, R1 is the first was it is also specified in register direct addressing mode. Here, we have a new notation and this is a notation, which is again is often used to specify an operand in base displacement addressing mode.

So, here what is being indicated is that, the value which is contained within register R2 is to be added to the specified displacement or the value 4 and what the result of doing, that addition will give you the memory address of the operand, to our add instruction, this one is complicated, but as we going to see in later lectures, this is a very useful kind of addressing mode to have. Let us took an example, again we will use the same example from the previous register and memory picture.

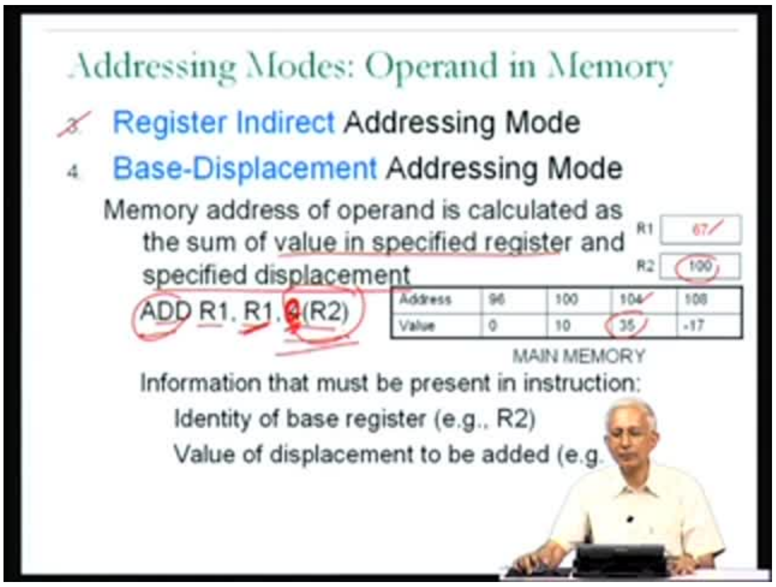
So, once again R1 contains a 32, R2 contains a 100 and memory is as shown. So, when this instruction comes up execution, in order to find out, what the second operand is, what the processor has to do is, it has to look at the contents of R2, where it finds 100, then it has to add to that value the value 4, yielding 104. It then knows that the operand to the add instruction is contained at memory address 104 and that the value of the operand is in fact, 35.

The first operand was previously identified as being 32 therefore, this add instruction is going to add 32 to 35 yielding 67, which would be the new result, resulting value in register R1.

So, the sounds like lot of work to do just to specify where an operand is in memory, but I will just make two observations. So, first is that as I said this is a simple extension to the register indirect addressing mode. And in fact, whenever you see a base displacement operand, you should bear in mind that, if a displacement of 0 was used then, when the address of operand is calculated by the processor, it will amount to being exactly the same as the register indirect addressing mode, which is why a processor which gives you the base displacement addressing mode might not actually give you the register indirect addressing mode, it is not necessary, you can achieve the register indirect addressing mode yourself by using a displacement of 0.

So, it is not the case that every processor, that you see will have all the addressing modes that we are going to talk about is very clearly not necessary to have, the register indirect addressing mode at all if the base displacement addressing mode is present, which might explain by the base displacement addressing mode is important.

(Refer Slide Time: 35:20)



Addressing Modes: Operand in Memory

3. Register Indirect Addressing Mode
4. Base-Displacement Addressing Mode

Memory address of operand is calculated as the sum of value in specified register and specified displacement

ADD R1, R1, 4(R2)

R1	67
R2	100

Address	96	100	104	108
Value	0	10	35	-17

MAIN MEMORY

Information that must be present in instruction:

Identity of base register (e.g., R2)

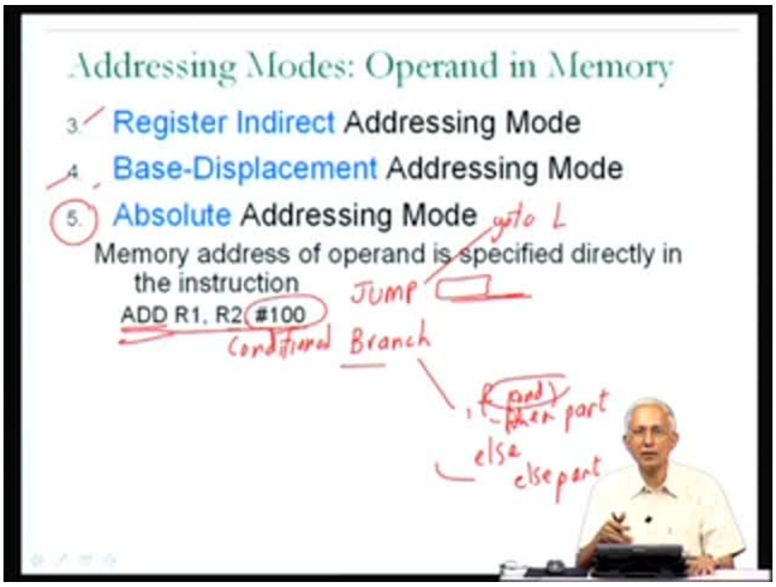
Value of displacement to be added (e.g.

It is a generalization of the register indirect addressing mode though, which more functionality is available and allows the register indirect addressing mode to also be achieved. And we look at some examples in coming lectures, you realize that it is very useful for the kinds of situations, that come up when the kinds of programs that, we write C programs or Java programs execute.

So, what information must be present in the instruction as far as an operand specified in base displacement addressing mode is concerned very clearly two pieces of information. The first is the identity of the base register, in other words the fact, that the base register of this operand is R2 and secondly, the value of the displacement, which in this case was 4.

The value which is to be added to the contents of the base register, in order to compute the address of the operand in memory and again I will repeat that, this is one of the key addressing modes, that we are likely to encounter.

(Refer Slide Time: 39:43)



Addressing Modes: Operand in Memory

3. Register Indirect Addressing Mode
4. Base-Displacement Addressing Mode
5. Absolute Addressing Mode
   Memory address of operand is specified directly in the instruction
   ADD R1, R2, #100

goto L
JUMP
Conditional Branch
if (cond) then part
else
else part

Now, moving right ahead, I like to talk a little bit about the next addressing mode, which is call the absolute addressing mode. Now, the absolute addressing mode is a simple addressing mode in that, the memory address of the operand is specified directly within the instruction.

And therefore, to look at an example of this would be simple, we would see something like again I am just using the add instruction generically and there is an operand here in a new format, this is not the kind of format, that we used for any of the previous addressing modes, I am just using it for the absolute addressing mode. This is not a very standard notation for the absolute addressing mode, every instruction said every assembly language will tend to use its own notation so in our examples, I will use this count sign in front of a value to indicate there is the absolute addressing mode.

So, as from the description, we understand that this means that, the operand is available at memory address 100 and therefore, the address 100 in memory is where the operand value would be available, that will be added to the contents of register R2 to provide the value that is to be put into the destination register R1.

Now, let me just back of a little bit and point out that, there are other instructions other than arithmetic logical instructions within instruction set for example, there are data transfer instructions and there are control transfer instructions and it might be the case that for something like a control transfer instruction for example, for a jump it might make a lot of sense to use this absolute addressing mode, again the way to think about this might be, I ask to you to relate the jumping instruction to let say a goto in your C program.

So, what form to the goto take in your C programs, how you specified the destination the target of the goto within the C statement, you actually had a label. So, you are indicating that the current instruction should cause control to transfer to the instruction, which is labeled L. And it is slightly that the jump instruction, if it is going to relative the goto statement of C programs is going to have the capability of specifying an arbitrary instruction in the program, for which reason the absolute addressing mode might be the most useful addressing mode to use for the jump instruction.

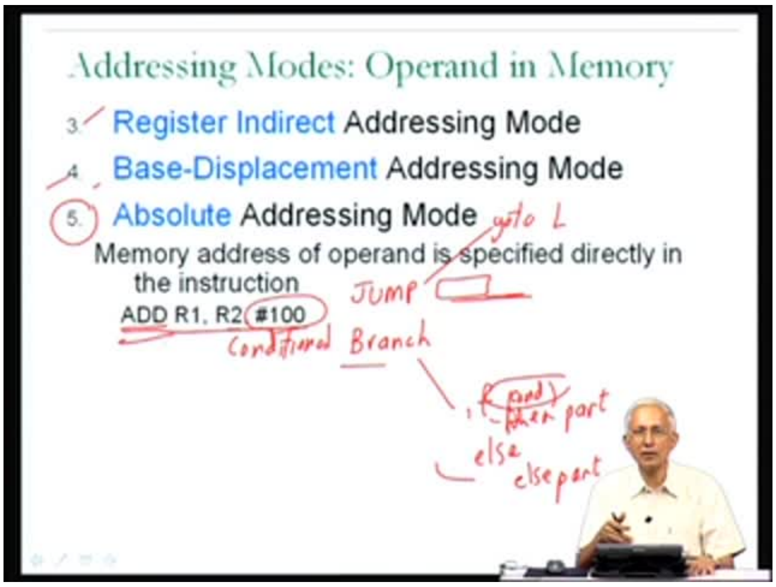
Whereas, it might not actually make a whole lot of sense to use the absolute addressing mode, in the way that we have shown in this example, the only reason, that I have used the absolute addressing mode using the add instruction examples, because I have use the add instruction, example for all of the previous instruction addressing mode descriptions.

So, do bear in mind that, there are other instructions other than the add instructions, there are various instructions for which different addressing modes might under being useful. Just you carry this example or little bit forward, let me point out that one of the other control transfer instructions, that we had was the conditional or branch instruction. And I indicate that the conditional branch instruction might relate to, if program the if then else you called that, in the if then else construct in a C program, you have a condition and if the condition is true, then the then part of the if then else is executed, but it is a condition is false, then the else part of the if then else is executed and you may have C programs in which you have if then else is.

So, clearly this going to be an instruction, which could be used to check whether condition is true and it is conditions is true, transfer control to the then, part and if the condition is false transfer control to the else part.

Now, the property of the if then else is going to be the typically that, then part may be very close to the instruction, which evaluates the condition the conditional branch instruction and therefore, it may be unnecessary to have an arbitrary memory address associated with the conditional branch instruction but, rather we may want to have some kind of an abbreviated version of the memory address, present inside the memory conditional branch instruction, for which reason we might find other addressing modes, which make sense for the conditional branch instruction. And I will refer to this in the next lecture when we continuous our discussion of the different addressing modes, but in brief we talked about several addressing mode up to this point in time.

(Refer Slide Time: 39:43)



Addressing Modes: Operand in Memory

3. Register Indirect Addressing Mode
4. Base-Displacement Addressing Mode
5. Absolute Addressing Mode
   Memory address of operand is specified directly in the instruction
   ADD R1, R2, #100

Some of the addressing modes make sense for any instruction set for example; I have suggested that the register direct addressing mode from the previous slide is likely to be present in any instruction set for processor, which has general purpose registers.

The mediate addressing mode is also likely to represent in any such processor the same is likely to be true for the base displacement addressing mode, but not necessarily for the register indirect addressing mode. Since, they would be situations, where you do not expect the processer to provide something that can be implemented using one of the other addressing modes. The absolute addressing mode is likely to be present for some instructions, it may not be present for add instructions since, it might be difficult to imagine situations, where a program would prefer having an operand in memory to having the same operand in a register, having the operand in register will allow the instruction to execute so much faster.

Therefore, one would not expect to see all combinations of all addressing modes with all instruction types in an instruction set. The person designing the instruction set will judiciously have for each different kind of instruction based on its type or is take logical control transfer data transfer, may be one or the other of the different addressing modes, rather than having all the addressing modes available for all the instructions, by doing this the complexity of the instruction set can be reduced. And the therefore, the complexity of the hardware, which is used to implement the instruction set can also be reduced.

So, with this in this lecture we have seen that the instruction set architecture of a computer, involves three different components. Complete description of the instructions the different kinds of instructions, which comprise the instruction set a complete description of all the addressing modes, which are used for specifying operands within the different instructions of the instruction set as well as the format of the instructions.

In the next lecture, we will continue our discussion of the different addressing modes and then, we will proceed to the little bit at the formats, before proceeding to look at several examples of, we look at a specific example a complete instruction set in some detail and then look at different code examples, which are implemented using that instruction set. Ultimately, the best way to understand the instruction set is not by reading the instruction set architecture manual, but by using the instruction set. And we will therefore look at some specific examples of core fragments wewill look at them in the instruction form not in the instruction form not in the binary form buy in the assembly language form. So, with this we will stop lecture 4 and continue with in addressing modes in lecture 5. Thank you.