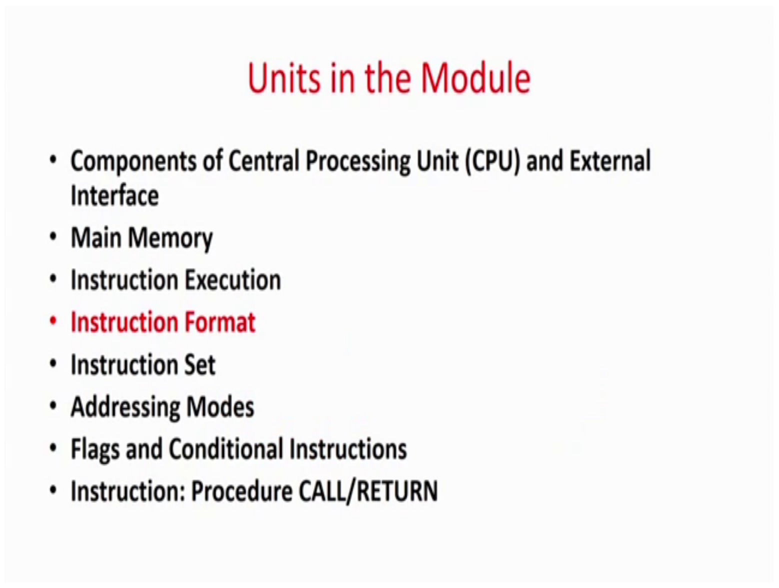
Computer Organization and Architecture: A Pedagogical Aspect

Prof. Jatindra Kr. Deka
Dr. Santosh Biswas
Dr. Arnab Sarkar
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Lecture - 10
Instruction Format

Welcome to the 4th lecture on that in the 4th unit of the module on addressing mode, instruction set and instruction execution flow.

(Refer Slide Time: 00:35)



Units in the Module

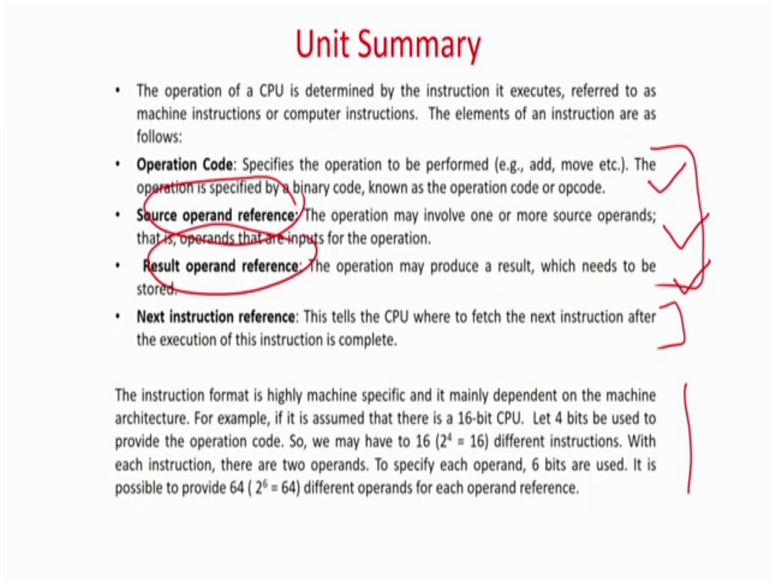
- Components of Central Processing Unit (CPU) and External Interface
- Main Memory
- Instruction Execution
- Instruction Format
- Instruction Set
- Addressing Modes
- Flags and Conditional Instructions
- Instruction: Procedure CALL/RETURN

So, in the last three units we have basically discussed what are the what is the basically a CPU? What it consists of; and how it is interfaced with the main memory and then we have gone into the details of basically, what is the basic motive of this module is to understand how a basically I instruction executes in a CPU. So, in that in that direction, first we had seen in the last unit that how our instruction is basically executed.

That in a one human architecture we know that it is already the data and the code is in the main memory and then slowly one after another the instructions are phased decoded and executed. So, that is basically the instruction execution. So, up to that we have seen in the last unit.

So, now in today's unit we are going to focus on a instruction format, because as I told you this model is mainly deliver or you would able to understand, basically how to design an instruction given a set of requirements for a set of specification that is the main goal actually. So, for that we first now look in a more generic fashion and what is an instruction and what is the basic format? So, this is the unit four of this module.

(Refer Slide Time: 01:39)



Unit Summary

- The operation of a CPU is determined by the instruction it executes, referred to as machine instructions or computer instructions. The elements of an instruction are as follows:
- **Operation Code**: Specifies the operation to be performed (e.g., add, move etc.). The operation is specified by a binary code, known as the operation code or opcode.
- **Source operand reference**: The operation may involve one or more source operands; that is, operands that are inputs for the operation.
- **Result operand reference**: The operation may produce a result, which needs to be stored.
- **Next instruction reference**: This tells the CPU where to fetch the next instruction after the execution of this instruction is complete.

The instruction format is highly machine specific and it mainly dependent on the machine architecture. For example, if it is assumed that there is a 16-bit CPU. Let 4 bits be used to provide the operation code. So, we may have to 16 ($2^4 = 16$) different instructions. With each instruction, there are two operands. To specify each operand, 6 bits are used. It is possible to provide 64 ($2^6 = 64$) different operands for each operand reference.

And then as we told that the course is being delivered in a pedagogical perspective. So, first let us see what is the unit summary. So, the in the in this unit basically you will be studying the general format of an instruction. So, as an instruction as we have discussed in the last unit then basically instruction executes or does the general operations in a computer. So, if there is an operation to be done. So, we require basically two things one is, what operation I have to do? And, basically on what operands you have to do the operation.

So, basically OpCode and source and result operands these are the two very important. So, the most fundamental thing even if is a non computer perspective. So, even if I ask you that you have to operate add two numbers. So, this is one of the very basic instruction we do in the at the are the low level school days.

So, this is basically an instruction. So, what is the format of an instruction? So, if I order you something. So, I have to tell you, what to do and also I have to tell you on what objects you have to do the operation and where you have to store the result. So, that is actually call the OpCode; that is, what operation you have to do?.

Like for example, you have to move an operand, you have to add two numbers etcetera, etcetera. And then you have to tell that what are the operands? That is the in case of a computer the operands are actually some immediate operations some immediate values which are in the instruction or in the more broad transfer the time being you can thing that we all have values of the operands, that is the variables and the values of the variables are stored in some memory is a one human architecture.

So, a source operand reference; that is, where the value of operand is stored in the memory, that is; the second part which is basically has to be there in the instruction. Then of course, I do some operation, now what I do with the result in that has to be stored in somewhere stored in memory whether it may be a register it can be a memory location etcetera.

So, that is the result operand reference that is; where you have to store the result. So, these things are basically of our layman language as for as well as our computer prospective these three stuff should be there in a instruction.

But, when you are thing over computer prospective or a code prospective, then after one instruction you have to execute another instruction. So, of course, you have to also tell in that instruction that which is the next instruction to be fetched. So, the reference of the next instruction that whether, it is the next immediate instruction or whether it is a jump instruction whether it will go to some other forward referencing or it may move back.

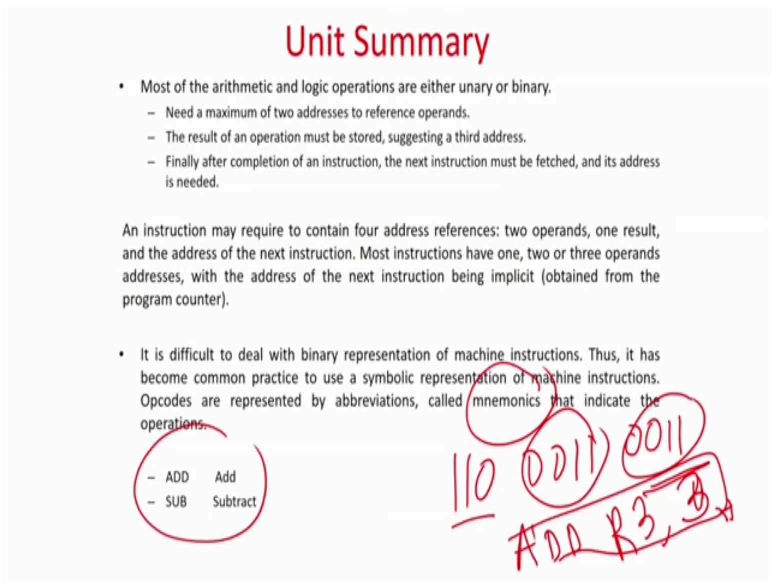
So, that reference also should be there so. In fact, if you talk of an instruction you have basically, what to do? On, what to do where you have to store the result and what to do next whether this is last instruction or whether the next immediate instruction has to be taken or then to some condition you have to some other places to jump in the memory.

So, that is the basic format of an instruction again as I told you one very important as I, now we give little bit like what how many? How do you decide the length of an instruction? So, of course, you have OpCode. So, it is represented in binary. So, if I said that the OpCode is 3 bits. So, how many operations are possible 2 to the power 3, 8 operations are possible.

So, if your specification says that 8 operations are fine like you can have stored add subtract multiply, then more or less I am very happy with 3 bit OpCode, but if I have lot most to do like I want to add immediate I want to add two numbers from a register, then, if I can have multiply I can have subtract I can have divide and it is what not.

So, in that case the numbers of instructions are much more. So, in that case the number of bits in the OpCode will be larger. So, as I given in the example in the summary that depends on the bit size of the OpCode of the reference you can decide how many instructions or different type of operations are supported.

(Refer Slide Time: 05:17)



Unit Summary

- Most of the arithmetic and logic operations are either unary or binary.
  - Need a maximum of two addresses to reference operands.
  - The result of an operation must be stored, suggesting a third address.
  - Finally after completion of an instruction, the next instruction must be fetched, and its address is needed.

An instruction may require to contain four address references: two operands, one result, and the address of the next instruction. Most instructions have one, two or three operands addresses, with the address of the next instruction being implicit (obtained from the program counter).

- It is difficult to deal with binary representation of machine instructions. Thus, it has become common practice to use a symbolic representation of machine instructions. Opcodes are represented by abbreviations, called mnemonics that indicate the operations.
  - ADD Add
  - SUB Subtract

Then, basically as I told you many times in the last 2, 3 units that basically instruction is divided basically into three types like: mathematical, arithmetic, operation. Then you can have some load store operation and there is re drive and there is some logical operation that is jump on 0 jump not on 0 etcetera; and one more thing.

So, there are basically that therefore, actually the next part means of basically; if these things are more or less of basic purgative of a instruction that these are the basic stuff required like OpCode source destination and what next instruction and basically three categories of instruction like arithmetic logic etcetera.

So, if you take a logical memory operation. So, sorry in arithmetic operation, we generally have two operands it can be add multiply subtract. And generally we take two sometimes unary operations unary operands also can be there like for example, this is the number you want to nugget it.

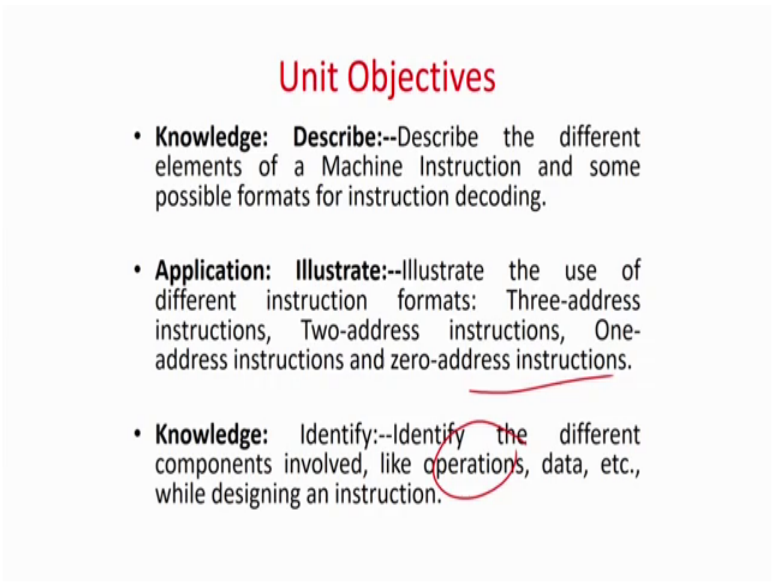
So, one operand is also possible, but there cannot be any 0 operand instruction, that is; very obvious and again before you go to the main stuff. Actually, as I told you that all these instructions basically are represented in binary like for add there should be an OpCode and there may be the representation of can be 101, sub the OpCode representation may be 111, but if you write a instruction like say 110.

Then may be 0011 and then 0 0 11 something like that. Then, what it means? It will mean add and this may correspond to the third register and this may correspond to 3 memory location number 3. So, it will say add whatever value of the variable stored in third memory location to be register number three very difficult to understand.

So, we generate like in a of mnemonic fashion a add R 3 to 0 3 sorry 3 hex. So, it was it add 33 hex. So, what does it mean you take the value of memory location 3 in hex and then added through number 3 and give to it will be addition as to read back.

So, these way instructions are represented will which we can read very nicely and easily. So, therefore, with all these mnemonics these are actually this abbreviations are called basically mnemonic way of representation like instead of add instead of 101, we will write add sub to note by the binary version. So, from now onwards throughout this course whenever we will talk about the instructions we have to understand eagerly that they are all represented in binary in a physical computer, but in this case for use of representation we do it in a abbreviation for which are embolics. So, that is what is the summery of this chapter of this unit.

(Refer Slide Time: 07:52)



Unit Objectives

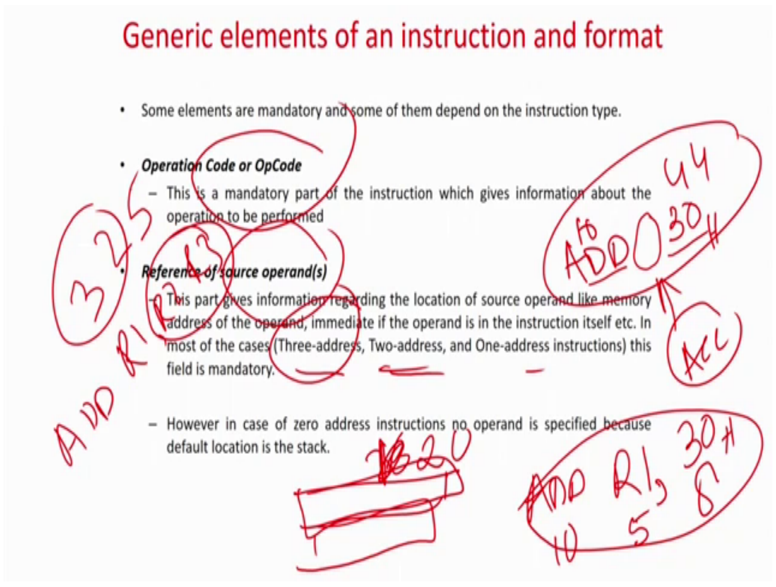
- **Knowledge: Describe:**--Describe the different elements of a Machine Instruction and some possible formats for instruction decoding.
- **Application: Illustrate:**--Illustrate the use of different instruction formats: Three-address instructions, Two-address instructions, One-address instructions and zero-address instructions.
- **Knowledge:** Identify:--Identify the different components involved, like operations, data, etc., while designing an instruction.

So, what are the objectives? That is going to we are going to fulfil after doing this unit basically we will be available to describe you will develop knowledge. So, recall type of an section in objective which will say the describe the different element of a machine instruction and some possible formats; that is, how basically an instruction loops, then you will be aware to illustrate very important instruction formats which were developed.

In the pedagogy of computer science or in the history of computers; that is three address instruction two address one address and even 0 address instruction very interesting that we will have many operand are there, but how it is operate. So, 0, 1, 2, 3 basically these are the formats OpCode is single.

Because, OpCode is will tell you what to do and the operands can be 0, 1, 2, 3 the 0 is very interesting and it will not have 7, 8, 9 instruction I means 9 operands, why? Because, otherwise the instruction will be very long it will be very difficult to store in the memory and then as a knowledge you will be tell the different you will able to identify the different type of component involved like given an instruction with which section represents the operation which section represents the data where the data is located etcetera.

(Refer Slide Time: 08:51)



Generic elements of an instruction and format

- Some elements are mandatory and some of them depend on the instruction type.
- *Operation Code or OpCode*
  - This is a mandatory part of the instruction which gives information about the operation to be performed
- *Reference of source operand(s)*
  - This part gives information regarding the location of source operand like memory address of the operand, immediate if the operand is in the instruction itself etc. In most of the cases (Three-address, Two-address, and One-address instructions) this field is mandatory.
  - However in case of zero address instructions no operand is specified because default location is the stack.

So, let us go to in details of the unit. So, basically the generic elements of an instruction and its format so, there are some of the very mandatory features or the mandatory part of an instruction; that is, the OpCode as we already discussed unless you tell what to do nobody can set up this an instruction.

So, average instruction will minimum have an OpCode; that is minimum and this is generally represented binary. So, if you have 1000 instructions. So, if you have 1000 different stack to do to. So, you should have 10 bits for the OpCode. So, that is very simple. So, based on the number of instructions required you pick a log and that will be size of the OpCode, then very important is the source of operands like where.

So, it can be 3, 2, 1 ; that means, if I say add say 30 hex ; that means, whatever is the value of the instruction is says that add, then the memory location 30 hex then the value, but add where. So, in this case if nothing is mentioned is a de facto standard then it is a accumulator. So, whatever value.

In the accumulator, as I already this as accumulator is a special type of register the value of the whatever value will be having in memory location 30 hex will be added to the value which is already stored in accumulator and it will be added back to this. So, if I do not write anything in this place. So, it will de facto means that it is accumulator.

So, this is a single word sorry single one address or a single operand type of an instruction. Now, why this is very good? Because, the size of the instructions are small because you can say add 30 and that maybe if you have some 1000 instructions. So, this one will take 10 bits and this is 4 plus 4, 16 bit so, 16 bits. So, your memory word length will be 16 bits, but say for example, two address.

So, in this case I can have something like add may be say a register one and then can be 30 hex now you see as we find. So, I have 10 bits for that, because you require 1000 plus instruction types, then this time may be if you have 32 registers. So, all registers will have different binary values.

So, it may be taken 5 bits 32 registers should be the power 5 bits 32 5 bits will be taken and should be again 32 bits. Now we can see 18 plus 5 so, it is 23. Now, 23 bits are require. So, if you assume that your memory word length is say 16 bit. So, in this case what will happen.

This instruction if the there is a two address instruction will not fit into a single memory word. So, it will become a double word memory the double word instruction, that is; one part will be here and then will be other part will be here. So, you will format it in such a way. So, that it become 32 bits that we can in alignment can be taken care of, but in main point is to say that now it will become a double word instruction.

So, first now how to even we fetch the instruction you have to take two memory locations at a time at a time you cannot do. So, you first fetch a part of the memory, that is; first location then you fetch the second part join them in the instruction register may be there can be can be two instruction resisters the width of the instruction registers has to be increased to instruction register you put in parallel jointly and then you decode it and.

So, you we will understand that it is more cumbersome to do it even, then if you have add and I said that de facto standard is accumulator in this case that is 8 plus 10, 18. So, in this case also it is with difficult. So, it may go ahead, but let us assume that for the time being if I have 20 size memory, then you can easily see that add 30 hex you easily fit in one word. So, these are taking all are hypothetical examples, but to just illustrate the concept that if you have a large long instruction.

Then basically the problem is that you will require multiple words to store in the memory, and when you fetch and decode there all more number of steps involved and the hardware is also complex, because if I say that the width of the memory is 20 or the word size is the 20 bits; then this instruction will fit 10 plus 4 plus 4, 18.

So, just a single instruction will be fetched in single instruction will be fetched decode and execute it, but if you take a double two address instruction. So, in case is 23. So, it will be one memory location plus another. So, generally these instructions are formatting in such a way that either it takes one word or it takes two word not generally one and half this just to give an example.

That, in now it will take more than one word. So, instruction and decoding and all those things will be much more cumbersome similarly with the three address of course, three address means you can have say something like that I can say add R 1 R 2 R 3. So, in this case the value of R 1 will be added to 2 and will be stored at R 1 so. In fact, what happens that I can add three numbers together?

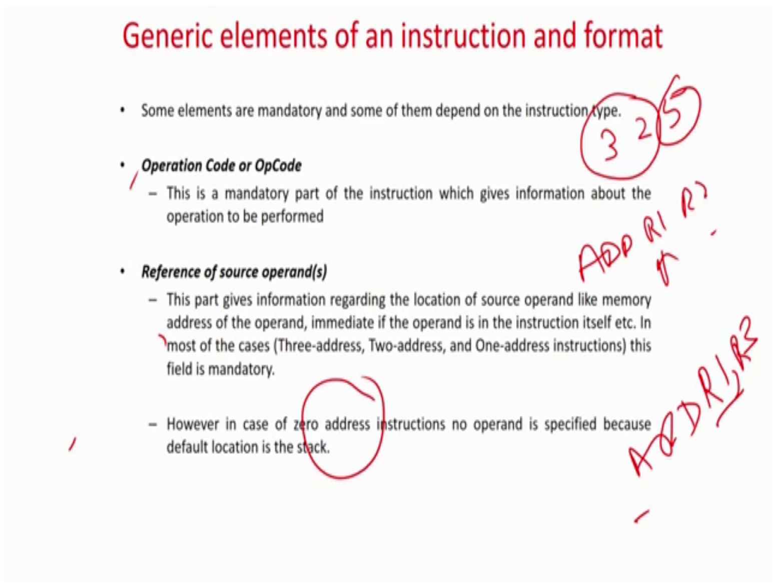
So, it may have the value of 3 it may have the value of 2 it may have the value of value 5. So, I require 3 plus 2 plus 5, I want to do you can write add R 1 R 3; that is, the value of R 3 will be added we will be added 2 R 2 will be added to R 1 and everything will be stored back in R 1, basically at this is the instruction format or.

How the instruction happens in this case? If it is the single if it is two address, then basically you can take only R 1, R 2, that is; first we will be adding 3 plus 2 2, then we will be stored in some temporary register like R 1 and then again the new value you have to add to the existence.

So, the by multiple numbers of steps so, let us look at it in a more nice fashion. So, that the; so, disadvantages already we have seen that if you take very long long instructions, then the memory words will be more and then the problem arise will be you have to have two memory location has to be fetched for a instruction, then decoder it will be slower and more hardware complex, but what the advantage is larger the instruction.

You can do same operation with less number of instructions like three address say that I have to add 3, 2 and 5. So, they are in three different registers for the time being.

(Refer Slide Time: 14:39)



Generic elements of an instruction and format

- Some elements are mandatory and some of them depend on the instruction type.
- *Operation Code or OpCode*
  - This is a mandatory part of the instruction which gives information about the operation to be performed
- *Reference of source operand(s)*
  - This part gives information regarding the location of source operand like memory address of the operand, immediate if the operand is in the instruction itself etc. In most of the cases (Three-address, Two-address, and One-address instructions) this field is mandatory.
  - However in case of zero address instructions no operand is specified because default location is the stack.

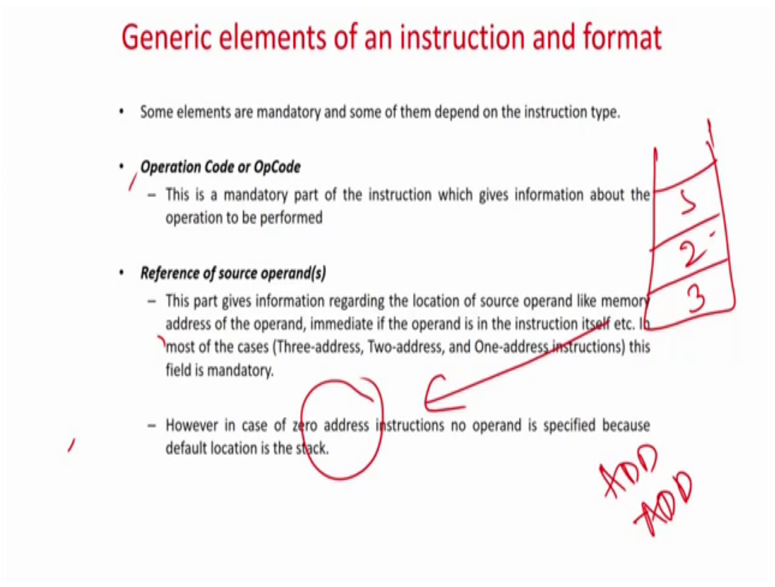
Let us assume. So, we can write add R 1 R 2 R 3. So, very simple value of R 3 will be added to R 3 will be added to R 1 and everything will be stored back to R. So, one instruction you do the purpose, but if it is a two word instruction; then you are gone. So, in fact, what will happen it will it will not be able to do it in 2.

So, this is the case. So, R 1 and R 2 so, first 3 plus 2 will be added and it will be stored in R 1. So, next instructions you have to write add R 2; sorry R 1, R 3. So, first stage what will happen 3 plus 2 will be added, because in R 1 there is 3 R 2 there is 2 they will be added and the value will be stored in R 1. Next instruction will be add R 1, R 3, R 3 it is 5. So, now, it will 5, 5, 10 and it is stored.

So, you require two instructions to solve the problem. So, your code will become larger. So, therefore, basically this is a trade off, but if you look at the current rate people have all gone for shorter instruction, then because our computers are nowadays become more and more faster than the number of executing one after another instruction is quite faster.

So, people have gone in this direction that then the instruction smaller less width execute more number of instructions per cycle, because your CPU processor are much faster let them making these instructions very complex and taking multiple words in the memory that is what is the trend and very interestingly we will said about; what is the 0? And this instruction there is no operand specify like.

(Refer Slide Time: 16:17)



Generic elements of an instruction and format

- Some elements are mandatory and some of them depend on the instruction type.
- *Operation Code or OpCode*
  - This is a mandatory part of the instruction which gives information about the operation to be performed
- *Reference of source operand(s)*
  - This part gives information regarding the location of source operand like memory address of the operand, immediate if the operand is in the instruction itself etc. In most of the cases (Three-address, Two-address, and One-address instructions) this field is mandatory.
  - However in case of zero address instructions no operand is specified because default location is the stack.

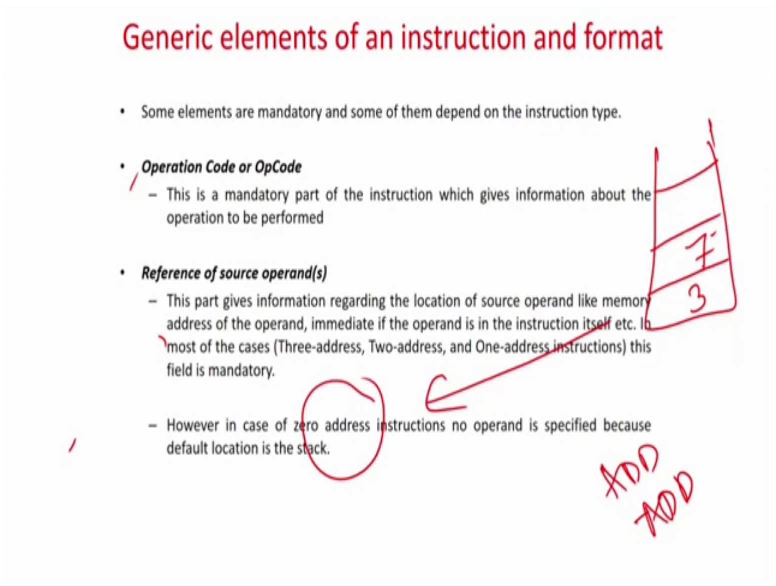
If I say add then where are the operands. So, whenever you say that I am doing with a zero word instruction or zero address instruction so. In fact, there is a stack involved with it. So, in that stack there will be different elements like for example, there are three there are these two. So, and then you (Refer Time: 16:24) the 5.

So, in case what will happen you have to write add and add. So, what is going to happen in this case. So, whenever you say add or any instruction you give it will take the first two elements depending if is the two address or three address.

Basically, but in fact, in case of zero address instruction basically, why it happens is that we default there is a stack attached to it and whatever operation you attach like add if you say add it will take the first two elements on the top the stack. We will we will pop that added and the value will be pushed to this stack may be if you say nugget is a single bit instruction is single operand instruction.

So, of the first value of the stack will be popped up made 5 and push it back. So, if I write add. So, what will happen 5 and 2 will be popped up and then what is it will be added up. So, it will be first it will be popped up.

(Refer Slide Time: 17:14)



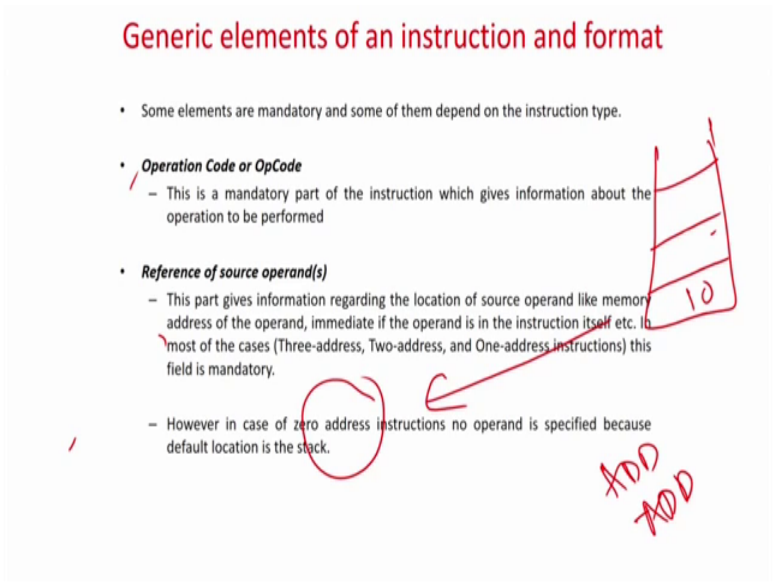
Generic elements of an instruction and format

- Some elements are mandatory and some of them depend on the instruction type.
- ***Operation Code or OpCode***
  - This is a mandatory part of the instruction which gives information about the operation to be performed
- ***Reference of source operand(s)***
  - This part gives information regarding the location of source operand like memory address of the operand, immediate if the operand is in the instruction itself etc. In most of the cases (Three-address, Two-address, and One-address instructions) this field is mandatory.
  - However in case of zero address instructions no operand is specified because default location is the stack.

So, 5 and 2 will be popped up 7, the value of 7 will be written over here.

Next if I say another add the first two will be taken and they will be popped up and the value of 10 will push back.

(Refer Slide Time: 17:20)



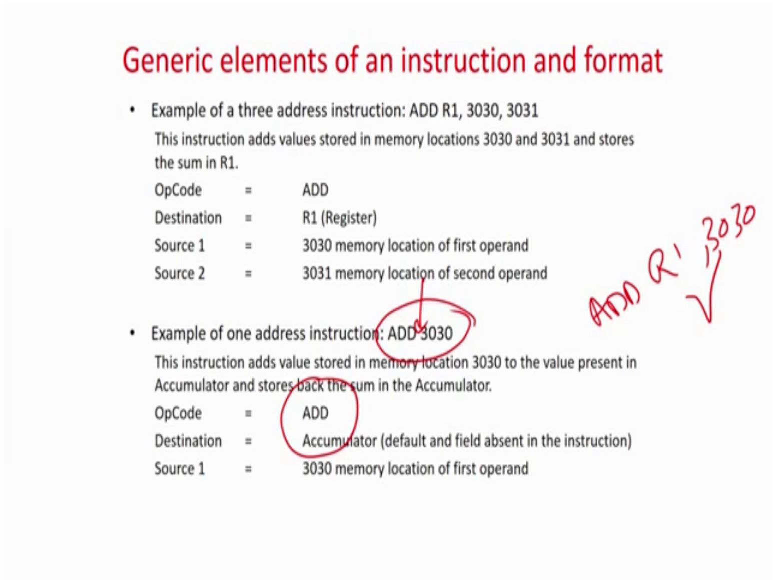
Generic elements of an instruction and format

- Some elements are mandatory and some of them depend on the instruction type.
- ***Operation Code or OpCode***
  - This is a mandatory part of the instruction which gives information about the operation to be performed
- ***Reference of source operand(s)***
  - This part gives information regarding the location of source operand like memory address of the operand, immediate if the operand is in the instruction itself etc. In most of the cases (Three-address, Two-address, and One-address instructions) this field is mandatory.
  - However in case of zero address instructions no operand is specified because default location is the stack.

So, whenever I told that zero address means is nothing to be surprised; that means, in the instruction itself you are not saying where are the operands, but the there is a different stack attached to it and it will start operating on the elements of this stack my first pushing means popping them up and doing the operation and push back.

So, if is the unary operation like nugget all those single; will be popped add means two will be popped. Generally, we have never have that three values are popped and the operations has been done that architecture was they were popular ok.

(Refer Slide Time: 17:48)



Generic elements of an instruction and format

- Example of a three address instruction: ADD R1, 3030, 3031

  This instruction adds values stored in memory locations 3030 and 3031 and stores the sum in R1.

OpCode	=	ADD
Destination	=	R1 (Register)
Source 1	=	3030 memory location of first operand
Source 2	=	3031 memory location of second operand

- Example of one address instruction: ADD 3030

  This instruction adds value stored in memory location 3030 to the value present in Accumulator and stores back the sum in the Accumulator.

OpCode	=	ADD
Destination	=	Accumulator (default and field absent in the instruction)
Source 1	=	3030 memory location of first operand

So, now we have told so, many theory. So, let us try to give some proper examples. So, is an instruction of a three instruction; three address instruction add R 1 3030 hex and 3031 hex. So, what did we say the OpCode is ADD. So, as I told you will be always using mnemonics will be never writing the binary value destination is the register R 1 if they are 32 registers all the registers they itself they will also have some binary representation if there will be all 00001, because if there are 32 number of registers.

So, the number of which reflect will be 5 and or this R 1 is it will be 00001, but for each of the representation will always use a numonic format. So, three words three address. So, this is the location of the first.

First operand this is the location of the second operand and as I told you slowly we will see that when we will be going more advanced into the instruction types. So, sometimes add it may tell that I have to add what is the location value here what is the value here I have stored it R 1.

Sometimes it may also mean that whatever value is present in 31, 30 you add them along with the whatever the values in R 1 and the result is stored in R 1. So, that will tell on the time of instruction as I told you that many times you have 1000 plus instruction. So, how many instructions can be possible right very difficult to think 1000 instruction add multiply subtract load jump etcetera, but you will be think that.

The number is not impractical, because add can be of similar types in this case I can say that it is add 2. So, I can say that it is add 2, what are they doing? It means I will take the variables or the operands from the last two or the last two operands and will add to R 1 and I can also say there can be another format R 3 add 3 what it will do?.

It will take the value present in 31 30 as well as an R 1 and together the value will be stored in R 1 just like if I say that add single address instruction add 3030 hex. So, what does it mean it means whatever the value is present in 30 add to the accumulator and write it back so; that means, you can have the formats or the possible in which case.

Also the register in the involved in the operation is that then just load and store. So, which one you make a restart and no effective and short. So, what I wanted to say that add can be of several types even something can be write add immediate, likewise; I can say that add immediate what does it mean add immediate 30 h. So, what does it mean? Add immediate means this 30 is no not a memory location this is basically the immediate value of 30 in h.

So, whatever in the accumulator will be added to 30, and it will be written back to this written back to the accumulator immediate means the value of the operand is specified in the instruction itself. So, in other words add can be of you know different types 20 types multiplication can be of. So, much variation store can be of. So, much variation jumps can also have; so, many variations.

That is, why the number of instruction if it is the basic operands is like add multiply subtract store can be very few, but the variations are huge in number. So, therefore, 1000 if an instructions or 500 in an instructions or 2000, different instruction is impractical number they are very much practical. So, anyway that is not the concentration of the or consideration or the not the method of I mean not the point of focus here you just trying to see what are the different instruction format and basically how they are represented.

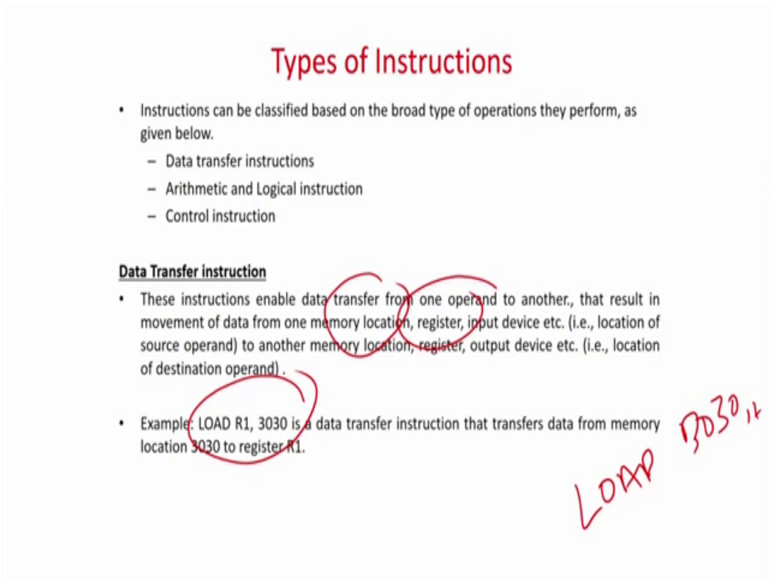
So, in this case we tell you that it is a 3 address instruction. So, these are the 3 address and this is the OpCode means where it is stored. Now, the values etcetera are been will be coming later when we will be going into more depth of instruction, how to design an instruction? What are the different types of instructions etcetera?

Now, the instructions say design etcetera will be taking up three units, then all those things will come that what is the add what is add immediate etcetera, then this is actually example of a single address instruction as I told you. So, one address instruction means basically there is nothing mentioned over here this is basically the accumulator when this is already the accumulator is the factor given over here.

So, whatever the value will be represent in 30 memory location will be added to the value already stored in the accumulator and you have to stored back in this one. So, these are the basic instruction I have shown you as an example ok.

In fact, if you if you consider two or instruction. So, you can have add R 1 3030. So, in that case it is a 2 address instruction. So, one will be the memory location what will be kind of R 1 and you can store it back in the R 1.

(Refer Slide Time: 22:14)



Types of Instructions

- Instructions can be classified based on the broad type of operations they perform, as given below.
  - Data transfer instructions
  - Arithmetic and Logical instruction
  - Control instruction

Data Transfer instruction

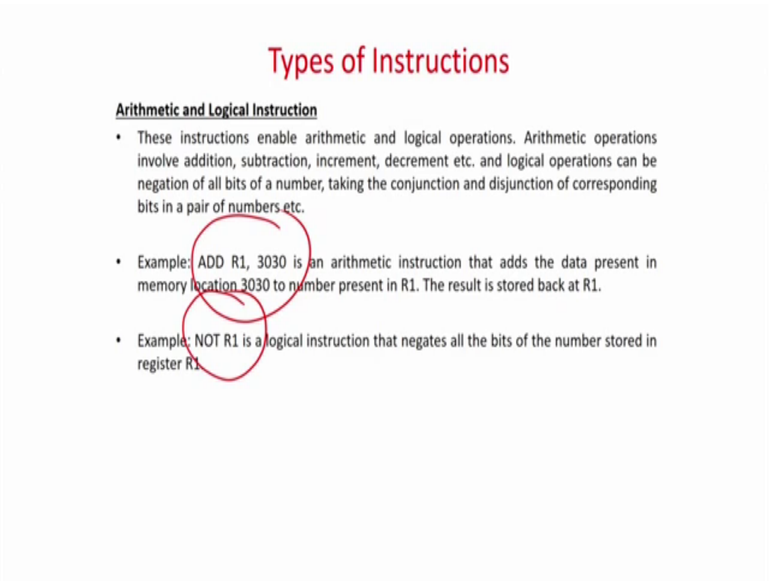
- These instructions enable data transfer from one operand to another., that result in movement of data from one memory location, register, input device etc. (i.e., location of source operand) to another memory location, register, output device etc. (i.e., location of destination operand) .
- Example: LOAD R1, 3030 is a data transfer instruction that transfers data from memory location 3030 to register R1.

Now, basically what are the instruction types. So, basically if you have look at the C program what do you have you declare some variables, then you do some addition multiplication subtraction and you have groups. So, basically and some stretched and standard printf and scanf treatment so, basically no code can have anything other than this that is data transfer instructions arithmetic and logical instructions and basically control instructions. So, whenever you see scanf storef and storing some variables basically they are nothing, but data transfer operation you get the value of the data from the memory then.

Arithmetic and logical instruction; that is the most important one like you go add subtract multiply etcetera and control like you have loops. If they in for while etcetera that they fall under the category of control instruction. So, instructions are basically only of this drivel and we can play around with it having different formats or different variations of them like for example, what is the data transfer instruction in case of a architecture basically you transfer data from one memory location to other one memory location can be a register another memory location a register to another memory location a register to register etcetera. So, any memory can in memory transfer is a in the transfer operation like for example, if I say load R 1 3030.

It means it will take the value whatever is available in this that is in 3030 and it will put a register number one this is the two address instruction and what (Refer Time: 23:38 you can have a single instruction like we can say load 3030 h. So, what it will mean in this case I am not specified any register means is a de facto standard in the accumulator. So, we stored the value whatever is available in memory location 3030 into the accumulator arithmetical logic instructions.

(Refer Slide Time: 23:55)



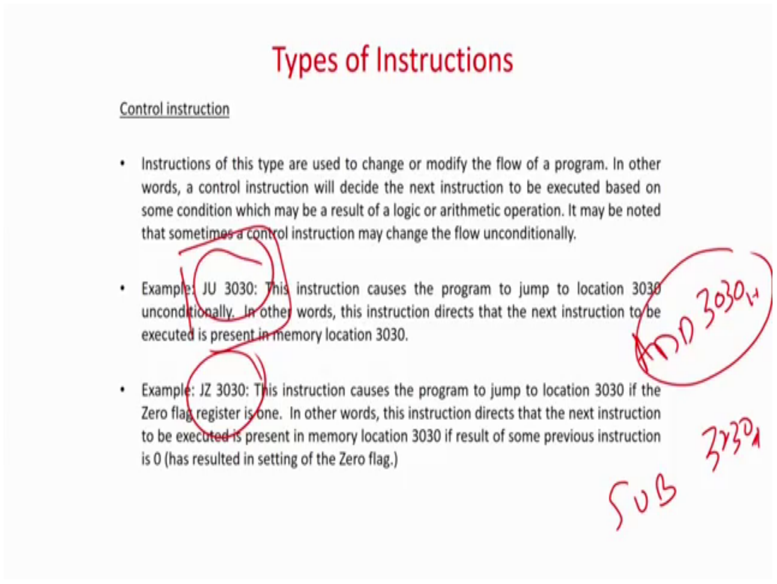
Types of Instructions

Arithmetic and Logical Instruction

- These instructions enable arithmetic and logical operations. Arithmetic operations involve addition, subtraction, increment, decrement etc. and logical operations can be negation of all bits of a number, taking the conjunction and disjunction of corresponding bits in a pair of numbers etc.
- Example: ADD R1, 3030 is an arithmetic instruction that adds the data present in memory location 3030 to number present in R1. The result is stored back at R1.
- Example: NOT R1 is a logical instruction that negates all the bits of the number stored in register R1.

As I told you they are the basic mathematics we do like add R 1 3030, that is add the value of 3030 memory location to register one and store in register two this is the two address instruction this is again see not one. So, this is basically a logical instruction that will negate the bits of the number stored.

(Refer Slide Time: 24:19)



Types of Instructions

Control instruction

- Instructions of this type are used to change or modify the flow of a program. In other words, a control instruction will decide the next instruction to be executed based on some condition which may be a result of a logic or arithmetic operation. It may be noted that sometimes a control instruction may change the flow unconditionally.
- Example: JU 3030: This instruction causes the program to jump to location 3030 unconditionally. In other words, this instruction directs that the next instruction to be executed is present in memory location 3030.
- Example: JZ 3030: This instruction causes the program to jump to location 3030 if the Zero flag register is one. In other words, this instruction directs that the next instruction to be executed is present in memory location 3030 if result of some previous instruction is 0 (has resulted in setting of the Zero flag.)

In register R 1. So, generally this is the; logical instruction and many most of the logical instructions, basically if you see will have a single one address. So, it is not a very standard rule, but generally not then you can say not negate all those things basically then shift which is a left shift right shift.

So, generally they have a single operand. So, single address basically then, but not all basically sometimes we can have bitwise and bitwise or ok. So, in that case this is also logical operation, but in that case they will have two addresses, but what I; what I mean this seen see that single operand or single address instruction is mainly type of logical instructions.

But, there can there are many legal instructions which have two operands like and of two numbers bitwise ok. Then next is very important instruction, because most of the code will have lot of logic logics; that means lot of logical or control that is if this happen you go to this if this happens you go back etcetera.

So, very important are the is I means which will be flow, but it never happens that you execute step 1, step 2, step 3 and done basically as many steps will check if this has been the condition I want to do this else I want to do that that is; why that is the idea of OpCode? The code instructions based on something either you will execute this or execute that.

So, that is why actually are; control instructions and the heart of any programming. So, generally here in this case also main memory will find that they are single address instructions like jump 3030. So, what it tells that unconditionally whatever happens you jump to the instruction which a memory location 3030. So, generally what happens if I say add 3030 h?

So, what is that mean it will mean that whatever value is available at 3030 add with R 1 and stored back in R 1 sorry accumulator, because it is a single address instruction, but when I say jump 3030. In that case what happen it is telling that the instruction available in 3030 has to be executed. So, if you take this scenario. So, in this case 3030 is having a instruction to be executed; and if you take this scenario.

So, in case the memory location is 3030 has a data. So, as I told you is the one human architecture. So, any place can have a data any place can have an instruction like, now there can be some condition instruction if it is saying the jump on 0, 3030; that means,, but whenever there is the condition instruction before that some other instructions has been executed based on which it has been done like for example, you can say that sub write it thirty.

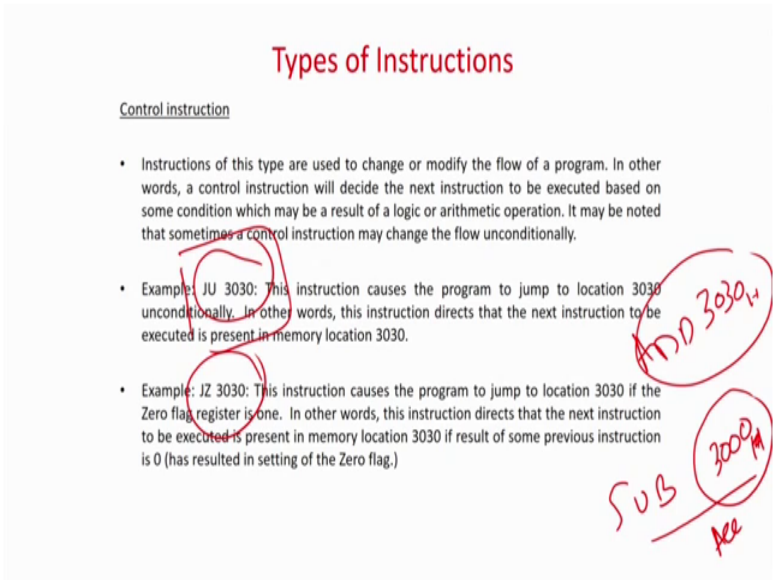
So, what does it mean it will mean you will take the memory location data 3030, whatever available in the accumulator subtract it as store the value in the accumulator, but whenever such operations are done there are some flags there is the flag register.

So, that will be safe there is a zero flag nonzero flag. So, whenever we will come to that we will read about it and also Professor Deka might have also has discussed something of some elaboration on the flats. So, whenever some mathematical operations are done or logical operations are done some flags are said like 0, is available in one flag carry is a well one flag.

So, some flags all set or reset. So, if you subtract the value of whenever as presented the accumulator with whatever value vas present in 3030, if the answer is 0. So, zero flag will be set otherwise 0 flag will not be set. So, then we say you want to say that if the memory location value of 3030 and the value of the accumulator as equal, then I want to go to sorry I should call it 3030.

There is confusion let me call it 3000 confusion.

(Refer Slide Time: 27:48)



Types of Instructions

Control instruction

- Instructions of this type are used to change or modify the flow of a program. In other words, a control instruction will decide the next instruction to be executed based on some condition which may be a result of a logic or arithmetic operation. It may be noted that sometimes a control instruction may change the flow unconditionally.
- Example: JU 3030: This instruction causes the program to jump to location 3030 unconditionally. In other words, this instruction directs that the next instruction to be executed is present in memory location 3030.
- Example: JZ 3030: This instruction causes the program to jump to location 3030 if the Zero flag register is one. In other words, this instruction directs that the next instruction to be executed is present in memory location 3030 if result of some previous instruction is 0 (has resulted in setting of the Zero flag.)

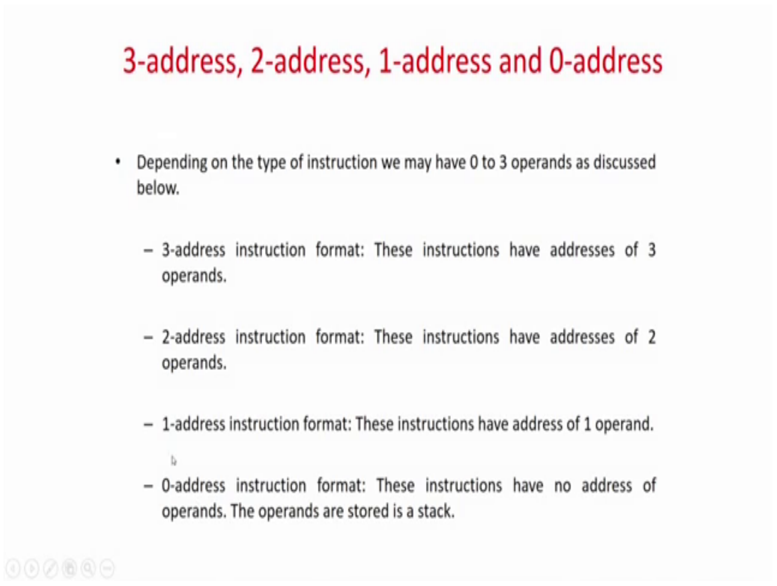
So, the so, 30 memory location 3000 has a variable in the location of a variable which has some value. So, I want to check whether this value is equal to the value available in the accumulator if those two values are equal, then I will jump to the instruction which is there in the memory location 3030.

So, in this scenario 3030 is having a instruction and memory location 3000 3000 hex is basically having a data. So, I compare this data with the accumulator; if they are equal then what I am going to do is that; I am going execute the instruction which is available in 3030. So, this is the instruction jump on 0.

So, that is the control instruction. So, what it does, but before that generally I should have done an instruction which is set my zero flag. So, suppose; I have done some equal to 3030 hex. So, if this two numbers are equal, then zero flag will be set; then when I am executing the instruction jump on 0 to 3030. You will check whether the zero flag is set, if the zero flag is set if you go to the memory location 3030 execute the instruction there have to continue from where the previous instruction was there ; that means, it will not jump with 3030 whether increment the program counter and going. So, therefore, these are actually basically control instructions.

Very important with they are two types jump condition and unconditional control unconditional means whatever be the case you go to that; that is, I mean what do I say that memory take the instruction there and executing conditional means basically it will depend on certain conditions, how the conditions are set. Based on some operations flag values are set and based on those flag values it will take. So, I have given you an example.

(Refer Slide Time: 29:27)

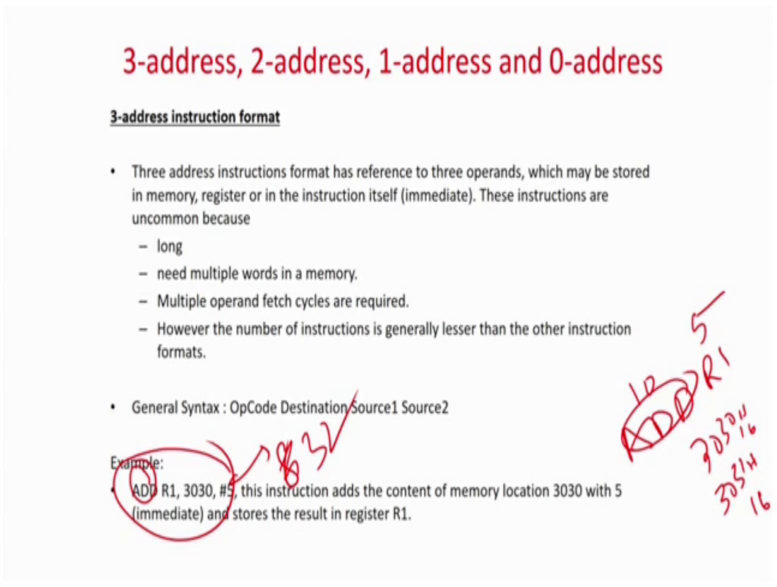


3-address, 2-address, 1-address and 0-address

- Depending on the type of instruction we may have 0 to 3 operands as discussed below.
  - 3-address instruction format: These instructions have addresses of 3 operands.
  - 2-address instruction format: These instructions have addresses of 2 operands.
  - 1-address instruction format: These instructions have address of 1 operand.
  - 0-address instruction format: These instructions have no address of operands. The operands are stored is a stack.

Now, again as I told you three address two address one address and zero address that is how many operands are there? So, this is the there address section format. So, R 1 30 hex so, as I told you in this case additions are have different type.

(Refer Slide Time: 29:33)



3-address, 2-address, 1-address and 0-address

3-address instruction format

- Three address instructions format has reference to three operands, which may be stored in memory, register or in the instruction itself (immediate). These instructions are uncommon because
  - long
  - need multiple words in a memory.
  - Multiple operand fetch cycles are required.
  - However the number of instructions is generally lesser than the other instruction formats.
- General Syntax : OpCode Destination Source1 Source2

Example:

- ADD R1, 3030, #5, this instruction adds the content of memory location 3030 with 5 (immediate) and stores the result in register R1.

So, you it is saying that whatever the value of memory location available in 3030; add with the immediate value 5 and store the result in R 1. So, this is a special type of an instruction means similar addition compared to this add instruction R 1 3030 x and say 3031 hex. So, the first one we will take that whatever available over 30 whatever available over this.

These two instructions has to be a major instructions to be added and put the whether in R 1. In fact, if you observe this instruction size may be quite large maybe 10 bits here 5 bits here this is 8, 16 16. So, you can understand that if you have an instruction which is the add R 1 and two memory location which is quite larger may be compared to this, because in this case 5 is an integer and the integer may be.

Around 16 bits or something I can I can I may not keep it as a 16 bit is size of this immediate range. I can think that the number or range of the numbers which are put in immediate values. So, I can restrict it to 8 bits that is 2 to the power 8 my; it is my decision or my format of design. So, I keep it.

So, what I want to say that? It not only the adds add instructions can be vary in the wave function, but it can also vary in length like if say the add R 1 and the two memory locations if length is 10 plus 5 plus 16 plus 16 bits, but here the immediate I can restrict not to bits, because in this case is a memory location memory address size here which is the range of number I want to get like, even if we get 32 bits.

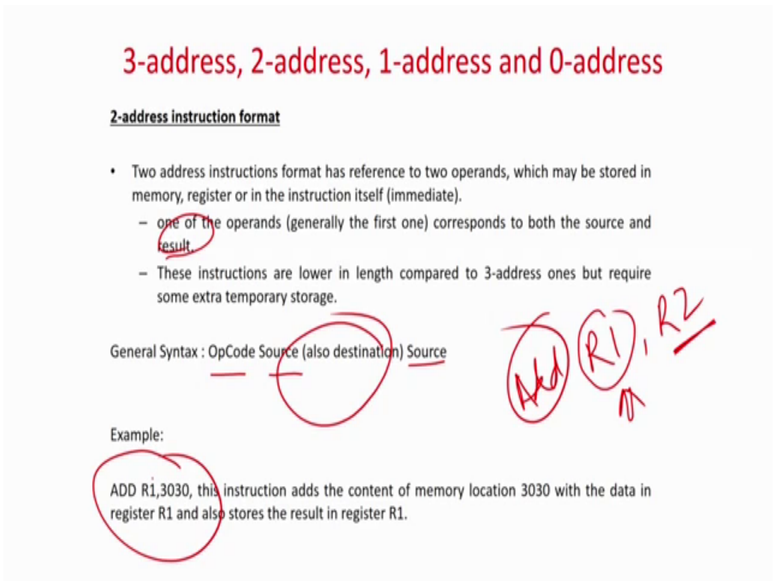
Make it longer, because I can get a very large precision number. So, I can keep it in the 32. So, in other words what I mean to say is that a same instruction same address instruction like three address, two address given any address instruction format the length may also vary and the way the same operation like add I mean also vary then.

So, therefore, this OpCode and this OpCode is vary. So, therefore, the number of types of adds the number substrates type of subtraction also varies in nature and also the OpCodes will be different. So, therefore, you reveal such number of this one. So, now again coming back to the storage the basic format of three address instruction is that there will be OpCode destination source and source sometimes this can be source as well as the destination.

That means to again take this source 1, source 2, this can be source 3 and you write back the value of the destination problem is quite long read multiple what is in the memory as I told you multiple operand features that this 3 operands means 3 times you have to talk to the memory to get the value and for a single instruction you have to read locations in the memory join them get the instruction totally long instruction you have to you know the instruction may be split into memory locations opera bring down join them and so, and so, and so, on; however, the number of instruction less required to execute is less because in one instruction you have around do much more operation 212 instruction.

Format is the most widely accepted.

(Refer Slide Time: 32:33)



3-address, 2-address, 1-address and 0-address

2-address instruction format

- Two address instructions format has reference to two operands, which may be stored in memory, register or in the instruction itself (immediate).
  - one of the operands (generally the first one) corresponds to both the source and result.
  - These instructions are lower in length compared to 3-address ones but require some extra temporary storage.

General Syntax : OpCode Source (also destination) Source

Add R1, R2

Example:

ADD R1,3030, this instruction adds the content of memory location 3030 with the data in register R1 and also stores the result in register R1.

So, it is it says that OpCode source so; that means, what happen is that sometimes actually like as I showed you add R 1 R 2 that mans its say that; whatever is the value of R 1 value of R 2 you have to add to R 1 and stored back. So, this one is both are source as well as the destination. So, that is what has been stored over there.

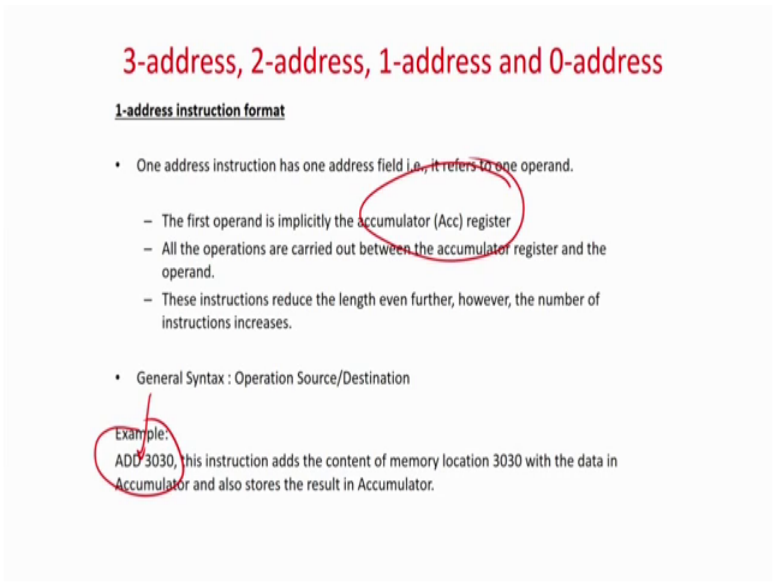
So, generally the first one, generally one of the operand generally the first one corresponds to both source end result they already have here is the source as well as a destination and this is generally the source. So, so it can vary the store it can have the memory location it can also be immediate, but for all these cases we OpCode will change and their variants of a. So, this is.

One example where it says that are 1, 3030; that means, the value of probability 3030 is to be added to R 1 and is stored at to R 1. So, in this case this is both are source and a destination, but in this case as I told you. So, generally speaking is a destination generally, they will add these two numbers and give the value of 1, 1 and for many cases.

Sometimes this one you get the source, but that is more real in two instruction format generally this is a source as well as the destination, but in these three case in these two address case. Generally, this was a destination itself to be does not add this plus this plus the value of this and stored back there, but for many cases many instruction formats or many machine architecture these are all sources of destination.

But; that was less popular this one maybe this was more popular what was the more format popular format that is these sources and this is on to the distinction, but in two address the source as well as before destination.

(Refer Slide Time: 34:15)



3-address, 2-address, 1-address and 0-address

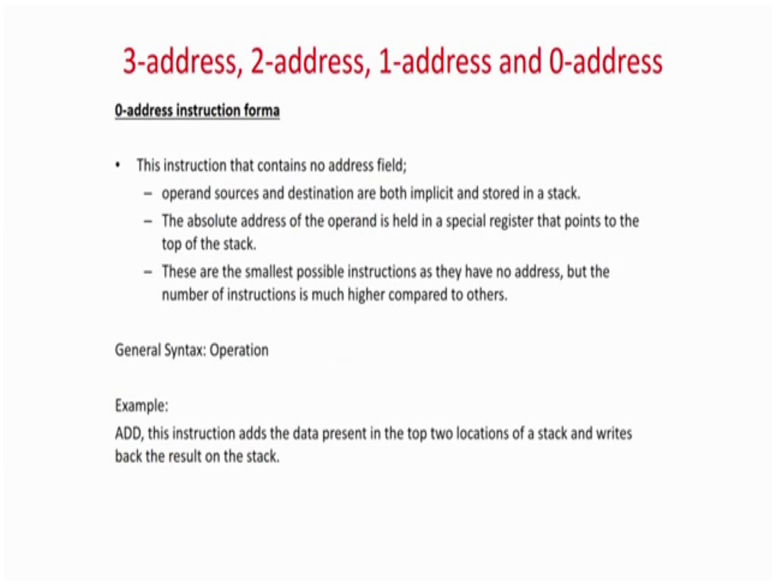
1-address instruction format

- One address instruction has one address field i.e., it refers to one operand.
  - The first operand is implicitly the accumulator (Acc) register
  - All the operations are carried out between the accumulator register and the operand.
  - These instructions reduce the length even further, however, the number of instructions increases.
- General Syntax : Operation Source/Destination

Example:
ADD 3030, this instruction adds the content of memory location 3030 with the data in Accumulator and also stores the result in Accumulator.

Then one address in this case as I told you one is a de facto standard is the accumulator; that means, whenever I say at 3030, if nothing is mentioned, that is a register which is the accumulator to its easier to write this instruction size is small and the affect is also similar to a two address format, because in that case also you have to make it specifically means any registered, but in this case you may not be able it is not equal to explicitly maintain mentioned the register name. So, instruction sizes are less that is the case.

(Refer Slide Time: 34:44)



3-address, 2-address, 1-address and 0-address

0-address instruction forma

- This instruction that contains no address field;
  - operand sources and destination are both implicit and stored in a stack.
  - The absolute address of the operand is held in a special register that points to the top of the stack.
  - These are the smallest possible instructions as they have no address, but the number of instructions is much higher compared to others.

General Syntax: Operation

Example:
ADD, this instruction adds the data present in the top two locations of a stack and writes back the result on the stack.

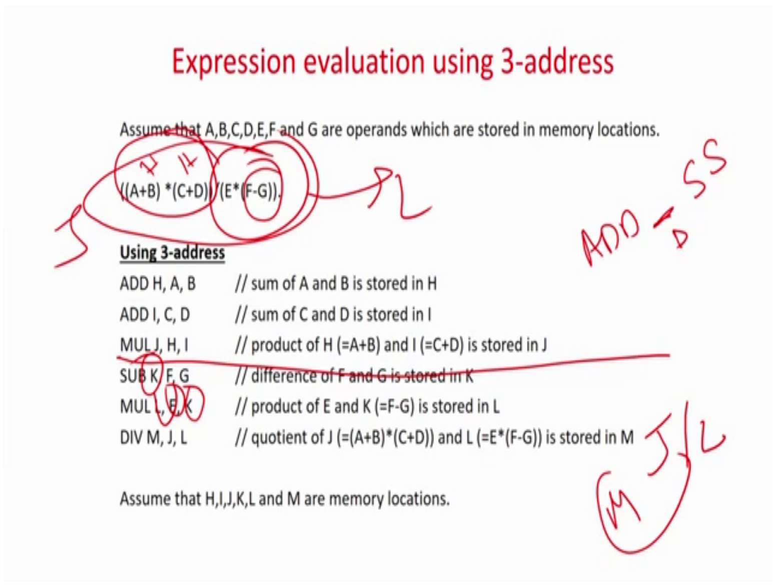
But one thing you have to understand that as more than more you make the instruction size as smaller more number of instruction will be report to execute the simple code or a single or given code and more number of operands ore more number of addresses you put it less number of instructions will require will be require to solve this a same problem or the same code there is obvious basically, but the theory that more longer you make the instruction hardware is more complex decoding fetching is more complex and therefore, the modern trend is towards simple instructions and execute them faster and as I told you the last stray format is 0 address format in zero address format basically only the operation is specified, but a defective standard is that you have a stack with it.

So, if I say add. So, what it will do? It will pop up the two locations at the result and write it back. So, basically you have the you have to been extra headache of a stack, but. In fact, there are many other ways this is zero address format is a stack comes that is the problem, but again the instruction sizes are small, but this zero address instruction has lot of rules in the system or handling the internals of a CPU execution like, the program counter. Whenever you execute procedure or whenever you have to jump from one memory location to other or you have to execute what do I say or in the

So, in that case what happens? Basically, you have store back store the old programs status word old value pc old value of register. So, that whenever after execution the interrupt service routine or the procedure have to come back we have to get the value. So, where these values are stored; So, they are basically stored in the stack and depending on the return and come back if we pop up the values and use them and before going to service the interrupt you have to store that in the stack.

So, there is a defect to stack is always double in the CPU. So, there are you can always use the same stack for a or a part of the stack or the same architecture for zero address instruction. Now, before we close down let us see a very practical example. So, this is a code I am not written the code. So, let us say that I want to add A plus B plus C plus D and subtract by this one.

(Refer Slide Time: 36:38)



Expression evaluation using 3-address

Assume that A,B,C,D,E,F and G are operands which are stored in memory locations.

((A+B) *(C+D))/(E*(F-G))

Using 3-address

ADD H, A, B // sum of A and B is stored in H
ADD I, C, D // sum of C and D is stored in I
MUL J, H, I // product of H (=A+B) and I (=C+D) is stored in J
SUB K, F, G // ~~difference of F and G is stored in K~~
MUL L, E, K // product of E and K (=F-G) is stored in L
DIV M, J, L // quotient of J (=(A+B)*(C+D)) and L (=E*(F-G)) is stored in M

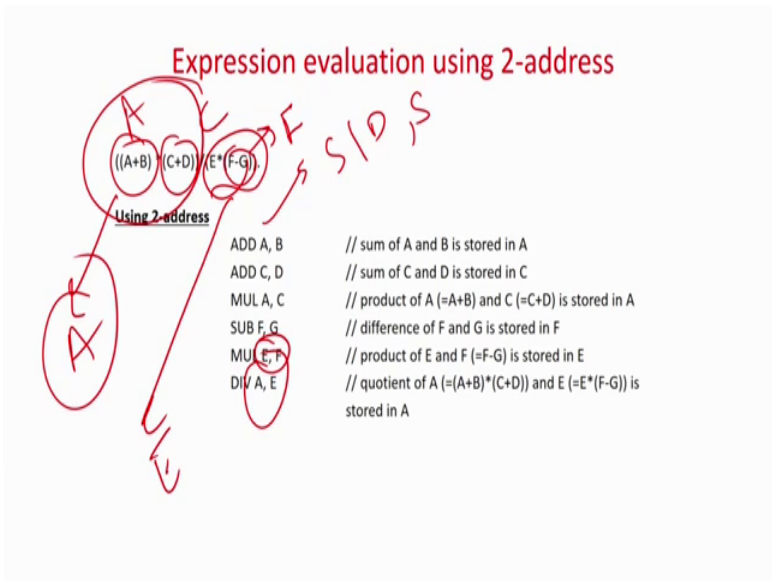
Assume that H,I,J,K,L and M are memory locations.

So, we are taking the architecture three address instructions in this case and in this case we are taking say that say add this is the destination and these two are the sources. We are not considering the cases that is the source or destination together we have taken the case that this is a destination only right. So, in this case so, some of so, first instruction is add h a B.

So, value of A and B are added and stored in C next instruction is add I, C, D. So, value of C, D is added and stored in I then you say that multiply H I. So, this one is actually now H and this is I. So, this one is done. So, whole thing this is computed as G now you take this. So, in this case you are subtracting F and G. So, if substrate F G the value will be stored at in K.

Now, you say MUL multiplication of K this is your K. So, this is where K is stored. So, you take K and multiply E and stored in the K. So, this part is basically now LN and then you have to divide J and l. So, J and L you have to divide and you have to stored that the value of L. So, finally, it is 2; so, how many instructions 1, 2, 3, 4, 5, 6. So, since the instruction actually shows the whole problem for you 1, 2, 3, 4, 5, 6 done.

(Refer Slide Time: 38:03)



Expression evaluation using 2-address

((A+B)*(C+D))/(E*(F-G))

Using 2-address

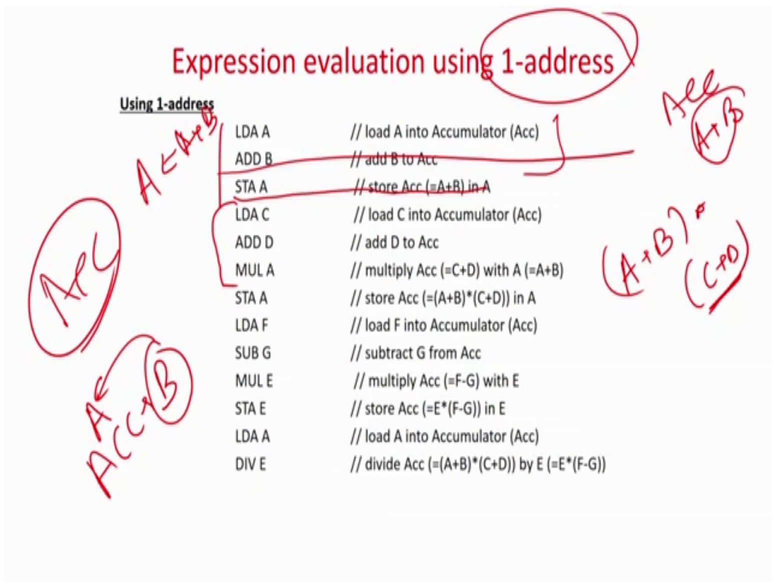
ADD A, B	// sum of A and B is stored in A
ADD C, D	// sum of C and D is stored in C
MUL A, C	// product of A (=A+B) and C (=C+D) is stored in A
SUB F, G	// difference of F and G is stored in F
MUL E, F	// product of E and F (=F-G) is stored in E
DIV A, E	// quotient of A (=(A+B)*(C+D)) and E (=E*(F-G)) is stored in A

Now, next two address. So, in this case as I told you this is both the sorry; if this source as well as destination and this is the source. So, now, what you have doing ADD A, B. So, the value of A and B are added and stored in a then C, D ADD C, D value of C, D and stored in C. So, now, this one is A this one is C.

Now, you say that F minus G some multi you can do the multiplication A and C you multiply these two; now this whole thing becomes a simple the subtract F minus G. So, this one we will have become F, then you multiply ELF you multiply these two when these whole thing will become your E and then finally, you add A and E and the store E F added.

So, A and E were added and the value will be stored in a. So, now, this is a final answer very interestingly how many instructions are equal if you look at this very easy to understand.

(Refer Slide Time: 38:51)



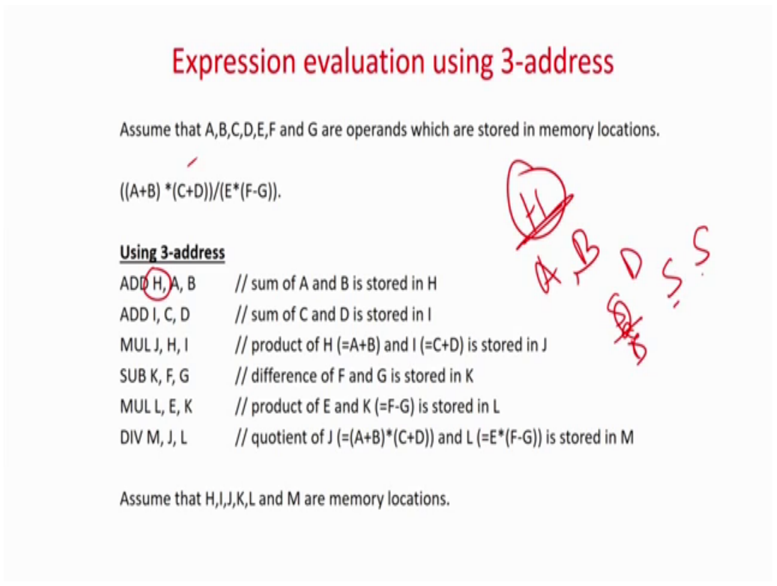
Expression evaluation using 1-address

Using 1-address

```
LDA A        // load A into Accumulator (Acc)
ADD B        // add B to Acc
STA A        // store Acc (=A+B) in A
LDA C        // load C into Accumulator (Acc)
ADD D        // add D to Acc
MUL A        // multiply Acc (=C+D) with A (=A+B)
STA A        // store Acc (=(A+B)*(C+D)) in A
LDA F        // load F into Accumulator (Acc)
SUB G        // subtract G from Acc
MUL E        // multiply Acc (=F-G) with E
STA E        // store Acc (=E*(F-G)) in E
LDA A        // load A into Accumulator (Acc)
DIV E        // divide Acc (=(A+B)*(C+D)) by E (=E*(F-G))
```

How many 1, 2, 3, 4, 5, 6 then there is no sorry then there is no advantage.

(Refer Slide Time: 39:01)



Expression evaluation using 3-address

Assume that A,B,C,D,E,F and G are operands which are stored in memory locations.

((A+B) *(C+D))/(E*(F-G)).

Using 3-address

```
ADD H, A, B    // sum of A and B is stored in H
ADD I, C, D    // sum of C and D is stored in I
MUL J, H, I    // product of H (=A+B) and I (=C+D) is stored in J
SUB K, F, G    // difference of F and G is stored in K
MUL L, E, K    // product of E and K (=F-G) is stored in L
DIV M, J, L    // quotient of J (=(A+B)*(C+D)) and L (=E*(F-G)) is stored in M
```

Assume that H,I,J,K,L and M are memory locations.

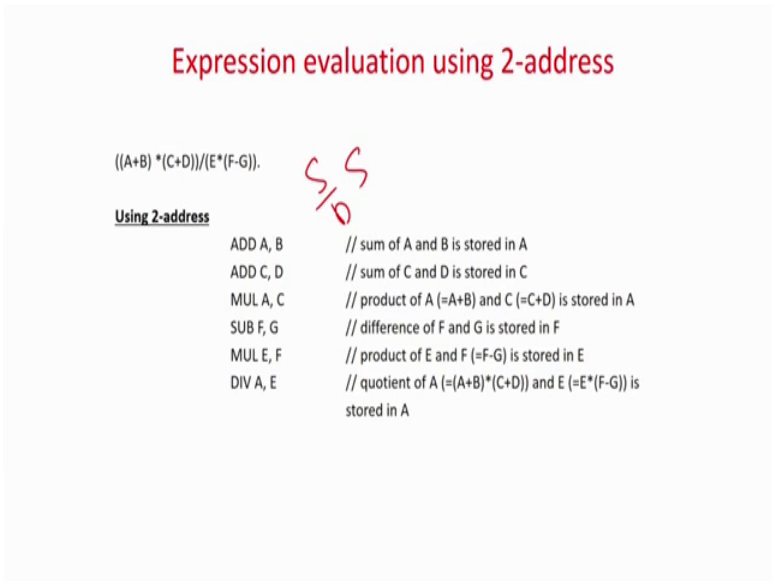
Now, why this is no advantage still always?

I will tell that; I will take the case of the two word this instruction, because this is also taken 6 and this is also taken as 6, but as I told you here you have kept the format like, A and B and the value is stored in each and using this edge as a only destination, but as a home work. You can always try and you will find that the number of instructions will reduce.

If you also allow this one to be a source as well as a destination it may not happen from this code, but you can easily find out codes where the length can be a much smaller. If you consider this as a source destination as well as these two are the sources. So, you will find out that the number of instructions will be much lesser in some examples, but here as I have considered this as the only source only destination and these two are the sources.

So, basically this one is actually source destination source. So, now, in this case affectively these two formats have become similar.

(Refer Slide Time: 39:51)



Expression evaluation using 2-address

((A+B) *(C+D))/(E*(F-G)).

S/D S

Using 2-address

```
ADD A, B        // sum of A and B is stored in A
ADD C, D        // sum of C and D is stored in C
MUL A, C        // product of A (=A+B) and C (=C+D) is stored in A
SUB F, G        // difference of F and G is stored in F
MUL E, F        // product of E and F (=F-G) is stored in E
DIV A, E        // quotient of A (=(A+B)*(C+D)) and E (=E*(F-G)) is
                stored in A
```

So, the same type of instructions same number of steps has been taken, but if you take a more generic way of doing it number of instruction sizes will be less number of instruction will be less now there will class come. Now, I am taking a single address instruction, now defect two is accumulator everywhere accumulator is defective. Now, big problem that is in this CAE; what you have done? You have say that an A comma B and the store the value of a C comma D value is in C, but if you do not writhing anything in a single address instruction, there is enough with the defective accumulator.

But, accumulator is only one. So, what you have to do you gave to use the accumulator and at the same time you have to again free, before it can be used for some other like; for example, what was are answer A plus B star C plus B. So, what is the first thing? A plus B star C plus D this will be n correct.

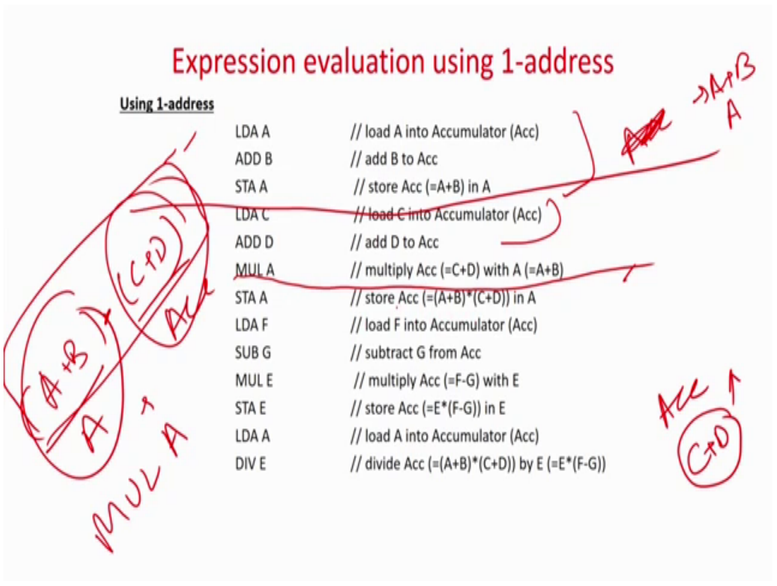
Now, what I said A ADD t so, what it will be it will add the value of do not saw as a load, because I am loading the value in the accumulator. Now, I say ADD B. So, what it does it will ADD accumulator plus B. So, what is that the accumulator in the value of B? The value of A in the accumulator you note the value of B to A and stored at the result. So, if B will be added to A and in the accumulator.

Because, A has the value of, because accumulator has the value of A, and it will be stored in the accumulator after these two what happens? Now you are having in accumulator us; now after these two steps you have the accumulator which is having the value of A plus B; now may be you will be using the accumulator to this C plus, because accumulator is only one.

So, if the free how can I free it I have to store the value of accumulator to memory location A. Now a accumulator has the value of A plus B. Now I say that store A; that means what? That is A plus B is stored in A.

So, therefore, after load A add B and store A, what is going to happen? A is going to have the value of A plus B. So, in action what happens I wrote the value of A in accumulator, I add the value of A plus B are stored in the accumulator and I am store the value of accumulator to A and free the accumulator I have to free the accumulator, because there is accumulator is one. So, now, I freed it now A is having the A value of B.

(Refer Slide Time: 42:20)



Expression evaluation using 1-address

Using 1-address

LDA A // load A into Accumulator (Acc)
ADD B // add B to Acc
STA A // store Acc (=A+B) in A
~~LDA C // load C into Accumulator (Acc)~~
ADD D // add D to Acc
MUL A // multiply Acc (=C+D) with A (=A+B)
STA A // store Acc (=(A+B)*(C+D)) in A
LDA F // load F into Accumulator (Acc)
SUB G // subtract G from Acc
MUL E // multiply Acc (=F-G) with E
STA E // store Acc (=E*(F-G)) in E
LDA A // load A into Accumulator (Acc)
DIV E // divide Acc (=(A+B)*(C+D)) by E (=E*(F-G))

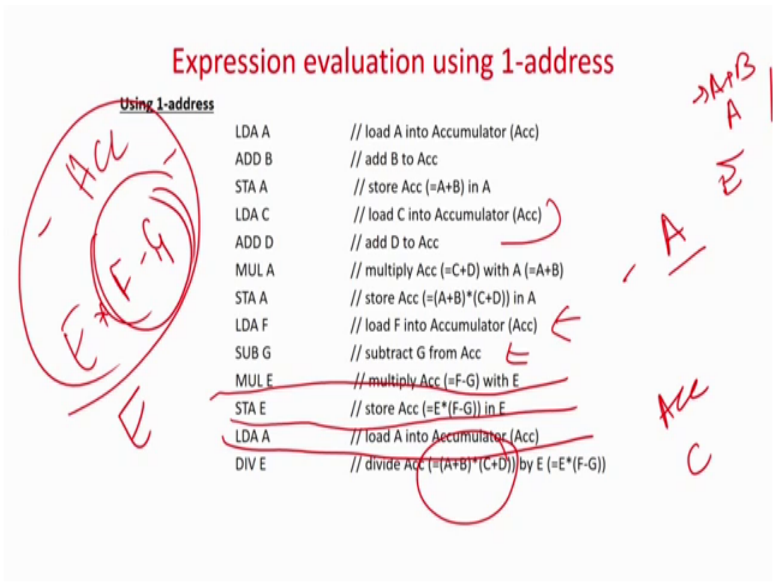
Similarly, these two things will do the same thing. So, what I am doing load C. Now accumulator will let me just erase it. So, up to this accumulator has the value of A plus B. Now accumulator is free basically accumulator is A, is having the value of A plus B and accumulator is free now. So, in this case now I load the value of accumulator C; then I ADD B.

Now, accumulator is going to have the value of after these two statements I have the value of C plus D will be accumulator. Now, what I had to do? My basic operations we performed in A plus B star C plus D. So, this one is now stored in A and C plus D is in the accumulator.

So, you can write an instruction MUL A; that means what C plus D? Which is already in the accumulator; when we multiplied with A which now has the initial value A plus B and accumulator will have the value of this whole thing and the first part of the expression A plus B starts it. So, up to this first part is done; now again I have to free the accumulator, because I have to also do this part of the operation. So, again whatever the value of this was in the accumulator.

So, again I store it in the this whole thing I store it in memory location A and the whole thing is, now in the memory location A. Again, I have to free the accumulator. So, now, it is very much in the similar procedure you can see. So, in the next step is basically store is done, now load F in the accumulator subtract G. So, F equal to F minus G. So, the next expression was to check.

(Refer Slide Time: 43:44)



Expression evaluation using 1-address

Using 1-address

```
LDA A        // load A into Accumulator (Acc)
ADD B        // add B to Acc
STA A        // store Acc (=A+B) in A
LDA C        // load C into Accumulator (Acc)
ADD D        // add D to Acc
MUL A        // multiply Acc (=C+D) with A (=A+B)
STA A        // store Acc (=(A+B)*(C+D)) in A
LDA F        // load F into Accumulator (Acc)
SUB G        // subtract G from Acc
MUL E        // multiply Acc (=F-G) with E
STA E        // store Acc (=E*(F-G)) in E
LDA A        // load A into Accumulator (Acc)
DIV E        // divide Acc (=(A+B)*(C+D)) by E (=E*(F-G))
```

The next expansion E star F minus G E star F minus G if you do this. So, now, you can see, what I have done? I have noted the value of F in the accumulator have subtracted it G. So, now, the accumulator will have F this accumulator will have F minus G now you multiply with E. So, this is done. So, now, the whole accumulator will have this.

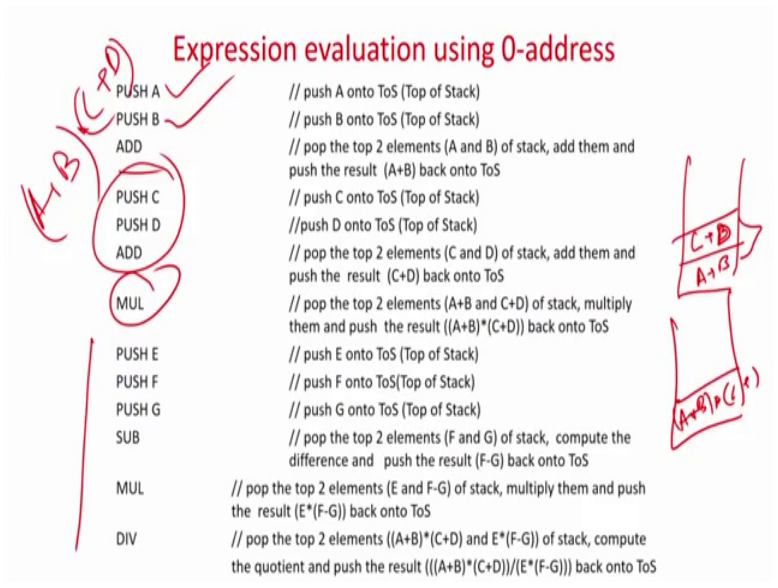
Now again you store E; that means, this whole value will be stored in E, because the accumulator has to be free. So, I am freeing the accumulator multiply this, when I am storing the value of accumulator to E the accumulator is freed. Now, I gain load the accumulator with A, because A was having this whole value and then you say dp. So, basically A was the first part of the equation, E is the second part of the equation before that is see the accumulator.

Again load the accumulator with A that is A equal to A plus B star B in to C, that part of the equation is loaded over there and then you do this operation and even done. So, you can if you do this program carefully it will be easily and joint with down will be as it will easily able to figure out, but what I mean to say is that now the number of instruction is 1, 2, 3, 4, 5, 6 quite long; why? Because, if there is a single accumulator there are single accumulator and if are using one word address.

So, every time you do some operation you have to free the accumulator, because before doing the another operation you may have to doing new valued from the memory location to the accumulator do it that is; why: If you are doing the single address instruction it will take the more number of instructions. In the most general case in this case I told you please take an example where the third operand can be a or the first operand.

Can be both are source or destination and take an expression which we have three variables at a time like A plus B plus C B, C plus A plus B plus C star C plus d plus k three variables you take and then you take this format and then this format we will always have less number of instructions over here that you can try.

(Refer Slide Time: 45:32)



Expression evaluation using 0-address

```
PUSH A    // push A onto ToS (Top of Stack)
PUSH B    // push B onto ToS (Top of Stack)
ADD       // pop the top 2 elements (A and B) of stack, add them and
          push the result (A+B) back onto ToS
PUSH C    // push C onto ToS (Top of Stack)
PUSH D    //push D onto ToS (Top of Stack)
ADD       // pop the top 2 elements (C and D) of stack, add them and
          push the result (C+D) back onto ToS
MUL       // pop the top 2 elements (A+B and C+D) of stack, multiply
          them and push the result ((A+B)*(C+D)) back onto ToS
PUSH E    // push E onto ToS (Top of Stack)
PUSH F    // push F onto ToS(Top of Stack)
PUSH G    // push G onto ToS (Top of Stack)
SUB       // pop the top 2 elements (F and G) of stack, compute the
          difference and push the result (F-G) back onto ToS
MUL       // pop the top 2 elements (E and F-G) of stack, multiply them and push
          the result (E*(F-G)) back onto ToS
DIV       // pop the top 2 elements ((A+B)*(C+D) and E*(F-G)) of stack, compute
          the quotient and push the result (((A+B)*(C+D))/(E*(F-G))) back onto ToS
```

So, generally the defective standard is less number of lesser number of addresses shorter will be the instructions, but more number of instructions for a code and the other we around know. So, now, the before we complete let us take a very simple that same example with the zero address we can go through the whole code, but I will just tell what happens. So, for example, first you have to put A plus B it will have you have to do A plus B star C plus D. So, this is the only thing you have to do other things are.

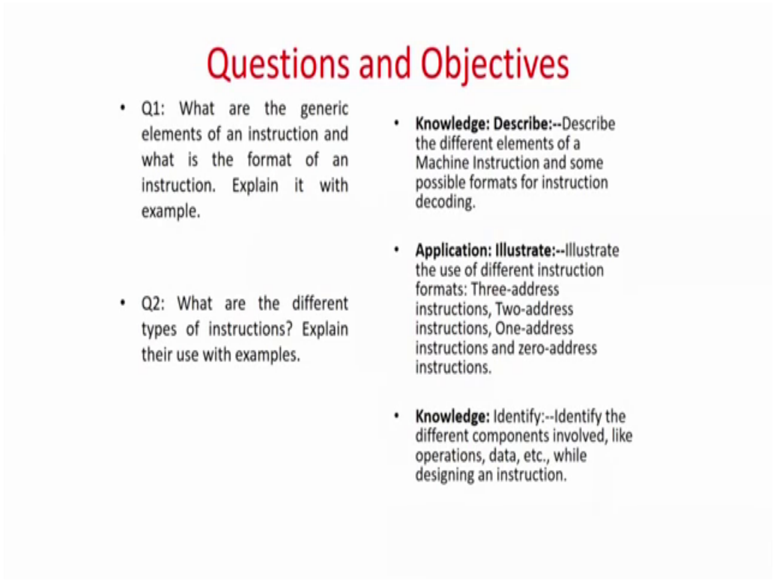
Similar so, we can easily explain. So, first you have 2, because the first two is whenever a single zero address instruction means is a stack. So, A plus B; how you do you have to first push a, then push B and then you have to say. So, push A push B are got then you say ADD. Now, what happens when add is done de facto standard it will pop this up and you will have the value of A plus B over here. Now, you have to be C plus D then you have to say push C push D. So, C and B will be there then you say ADD. So, it will be automatically you got and it will be same got and which will be one it will be equal to C plus D; sorry C plus D, then you say MUL, then what will happen these two values will be popped and multiplied.

So, now the stack will have the first value that is A plus B star C plus question will be done. Similarly you can very easily interpret; how it will happen? So, again now EFG you have to do. So, you have to push and pop and you have to sorry you will get it answer done.

So, in so, generally the number of stack based operation will be more in number compared to even a single address instruction or where is a stack based if the more number of instructions compared to a one (Refer Time: 47:20) instruction. So, there will logical basically. So, if you have very long instructions or a very complex instructions and more number of operations can be done together the number of instructions will be less.

And it is a very simple instruction the number of instructions required will be more to solve the same problem.

(Refer Slide Time: 47:34)



Questions and Objectives

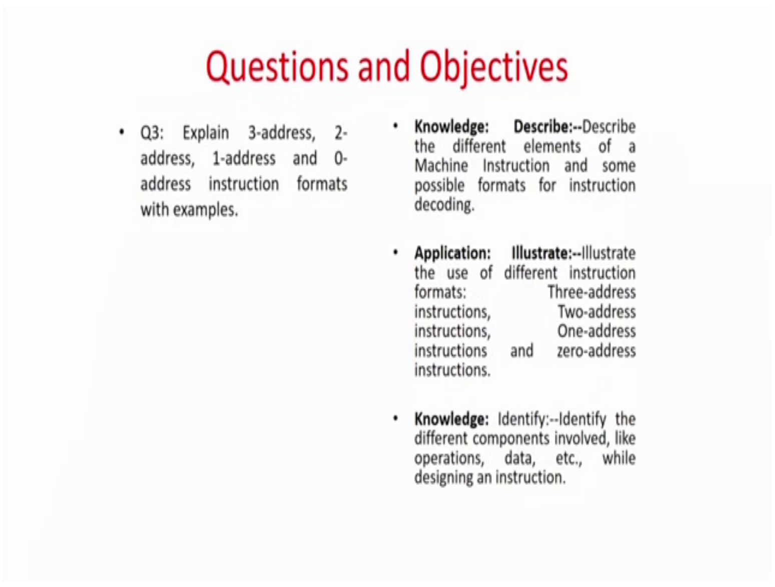
- Q1: What are the generic elements of an instruction and what is the format of an instruction. Explain it with example.
- Q2: What are the different types of instructions? Explain their use with examples.

- **Knowledge: Describe:**--Describe the different elements of a Machine Instruction and some possible formats for instruction decoding.
- **Application: Illustrate:**--Illustrate the use of different instruction formats: Three-address instructions, Two-address instructions, One-address instructions and zero-address instructions.
- **Knowledge:** Identify:--Identify the different components involved, like operations, data, etc., while designing an instruction.

So, that is what basically we have in this unit which where we where we have showed basically; what is the basically a instruction format? What it has? And basically depending on a format what are the good things and what are the bad things. So, some questions like and we see how it meets the objectives, what are the generic elements of an instruction explain is an example.

So, you can see that it can easily handle we describe the different elements of a machine instruction and some possible format for instruction decoding. So, easily if you solve this problem you can very easily meet this objective what are the different type of different type of instruction explained with the examples.

(Refer Slide Time: 48:16)



Questions and Objectives

- Q3: Explain 3-address, 2-address, 1-address and 0-address instruction formats with examples.

- **Knowledge:** **Describe:**--Describe the different elements of a Machine Instruction and some possible formats for instruction decoding.
- **Application:** **Illustrate:**--Illustrate the use of different instruction formats: Three-address instructions, Two-address instructions, One-address instructions and zero-address instructions.
- **Knowledge:** Identify:--Identify the different components involved, like operations, data, etc., while designing an instruction.

This one and these two are the objectives which is satisfied even for this one because different type of instruction means; if I take a data business. Now, explain three address, two address, one address as formats with example already we have discussed and it actually satisfied; that if you are able to explain different type of instructions with different addressing formats you will be able to solve you have able to meet all this recall based objectives.

So, with this we conclude this unit and the next from the next unit in the next unit. What we are going to do? Basically, we are going to see more specific idea of how basically the instructions will one like as I told you add can be your very different types that is these two addressing modes at these may be two words, three words, one word is one operand is available in the memory 1 in the register and you have the add them two.

So, how different type of same operations like, but how different ways the instructions can be instructions work what are the different variations what are the different type of instructions how can we make them as a set of instructions. So, more in depth will go to the instruction set design ok.

Thank you.