

(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.

Handwritten notes: "LOAD" and "3030" are circled in red. "LOAD" is also written in red ink to the right of the slide.

Now, basically what are the instruction types? So, basically even 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 loops. So, basically and some standard printf and scanf statement. 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 say 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 do add subtract multiply etcetera and control like you have loops. If, then, for, while etcetera that they fall under the category of control instruction. So, instructions are basically only of this three 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 so memory location can be a register, another memory location, a register to another memory location, a register to register etcetera. So, any memory to any memory transfer is a data transfer operation like for example, if I say `LOAD R1, 3030`.

It means it will take the value whatever is available in memory location 3030 and it will put in register number one, this is a two address instruction and what is it. Even you can have a single instruction like we can say `LOAD 3030 h`. So, what it will mean in this case I have not specified any register means its de facto standard is the accumulator. So, we stored the value whatever is available in memory location 3030 into the accumulator.

(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.

Arithmetic and logic instructions as I told you they are the basic mathematics we do like ADD R1 3030, that is add the value of 3030 memory location to register one and store in register two this is a 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 in register R1.

(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.)

Handwritten notes: JU 3030, JZ 3030, SUB 3030

So, generally this is a logical instruction and many most of the logical instructions, basically if you see will have a single one address instruction. So, it's not a very standard rule but generally.

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 a logical operation, but in that case they will have two addresses, but what I what I mean to say that single operand instruction or single address instructions are mainly type of logical instructions.

But, there can there are many logical 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 happens you go to this if this happens you go back etcetera.

So, very important means you change the flow, that is it never happens that you execute step 1, step 2, step 3 and done. Basically at many steps we 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 a code. The code takes instructions based on something either you will execute this or execute that.

So, that is why actually there are control instructions at the heart of any programming. So, generally here in this case also main memory we 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 is in memory location 3030. So, generally what happens if I say add 3030 hex?

So, what is that mean it will mean that whatever value is available at 3030 add with *R1* and store back in *R1* sorry accumulator, because it's 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 this is a Von Neumann architecture. So, any place can have a data any place can have an instruction like, now there can be some conditional instruction it is saying that jump on 0, 3030; that means, but whenever there is a conditional instruction before that some other instructions has been executed based on which it has been done like for example, you can say that SUB 3030.

So, what does it mean it will mean you will take the memory location data 3030, whatever is available in the accumulator subtract it and 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 set there is a zero flag nonzero flag. So, whenever we will come to that we will read about it and also Professor Dekka might have also has discussed something of some elaboration on the flags. So, whenever some mathematical operations are done or logical operations are done some flags are set, like 0 is a well known flag carry is a well known flag.

So, some flags are set or reset. So, if you subtract the value of whatever was present in the accumulator with whatever value was present in 3030, if the answer is 0. So, zero flag will be set otherwise 0 flag will not be set. So, that means say we want to say that if the memory location value of 3030 and the value of the accumulator are equal, then I want to go to sorry I should not call it 3030. There is a confusion let me call it 3000.

(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.)

Handwritten red annotations: 'ADD 3030', 'SUB', 'Acc'.

Ok so, 30 memory location 3000 has a variable 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 lying 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 to 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 SUB 3030 hex. So, if these two numbers are equal, then zero flag will be set; then when I am executing the instruction jump on 0 to 3030 it will check whether the zero flag is set, if the zero flag is set it will go to the memory location 3030 execute the instruction there or it will continue from where the previous instruction was there; that means, it will not jump to 3030 rather increment the program counter and go on. So, therefore, these are actually basically control instructions.

Very very important there are two types jump conditional 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 execute it, conditional means basically it will depend on certain conditions, how the conditions are set. Based on some operation some 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 in 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 three address instruction format. So, R1 30 hex so as I told you in this case additions are all different type.

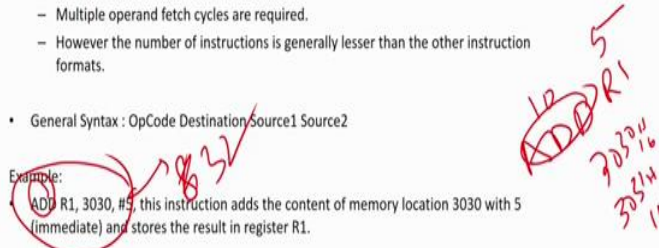
(Refer Slide Time: 29:33)

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 *R1*. So, this is a special type of an instruction means similar addition compared to this add instruction *R1* 3030 hex and say 3031 hex. So, the first one we will tell that whatever available over 30 whatever available at this.

These two instructions has to be a variables has to be added and put the result in *R1*. 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 *R1* 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 2^8 my; it's 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 way it functions, but it can also vary in length like if I say that add *R1* and the two memory locations if length is $10 + 5 + 16 + 16$ bits, but here the immediate I can restrict not to 16 bits, because in this case it's a memory location memory address size here it's the range of number I want to give, I can even give it 32 bits making it longer, because I can get a very large precision number. So, I can keep it to be 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 it may also vary. So, therefore, this opcode and this opcode will 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, we require such number of this one.

So, now again coming back to the story 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 you 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 words in the memory as I told you, multiple operand fetches that is 3 operands means 3 times we have to talk to the memory to get the value and for a single instruction you have to read different locations in the memory, join them get the instruction totally long instruction you have to you know the instruction to be split in two memory locations you have bring them and join them and so forth; however, the number of instruction less required to execute is less because in one instruction we are able to do much more operation. 2 instruction format is the most widely accepted.

(Refer Slide Time: 32:33)

3-address, 2-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

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 source; that means, what happen is that sometimes actually like as I showed you ADD R1, R2 that means it says that; whatever is the value of R1 value of R2 you have to add to R1 and store 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 and result they already have here is the source as well as a destination and this is generally the source. So, so it can be the store it can have the memory location it can also be immediate, but for all these cases the opcode will change and there variants of ADD. So, this is one example where it says that $R1, 3030$; that means, the value of memory location 3030 is to be added to $R1$ and is stored back to $R1$. So, in this case this is both a 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 to $R1$. But for many cases sometimes this was also used as a source, but that is more real. In two instruction format generally this is a source as well as the destination, but in this three case in this three address case generally, this was a destination itself. People does not add this plus this plus the value of this and store back there, but for many cases many instruction formats or many machine architecture this was a source as well as destination.

But; that was less popular this format is was more popular. What was the more format popular format that is these two are the sources and this one is the distinction, but in two address the source as well as the 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 add 3030, if nothing is mentioned, that is a register which is the accumulator so it's easier to write this instruction size is small and the effect is also similar to

a two address format, because in that case also you have to explicitly mention the register, but in this case you may not be able it is not required to explicitly maintain mention 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 format

- 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 and more you make the instruction sizes smaller more number of instruction will be required to execute a simple code or a single or a given code and more number of operands or more number of addresses you put it less number of instructions will require will be required to solve this same problem or the same code there is obvious basically, but the theory is 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 instruction format is 0 address format, in zero address format basically only the operation is specified, but a de facto 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 and add the result and write it back. So, basically you have the you have to have the 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 a procedure or whenever you want to jump from one memory location to other or you have to execute what do I say a interrupt.

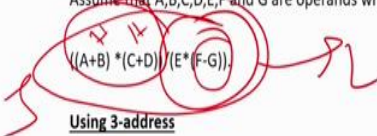
So, in that case what happens? Basically, you have to store back store the old programs status word old value pc old value of register. So, that whenever after execution the interrupt service routine or the procedure you have to come back we have to get the value. So, where these values are stored; so, 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 de facto stack always available in the CPU. So, generally you can always use the same stack 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 $A + B + C + D$ 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.



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.

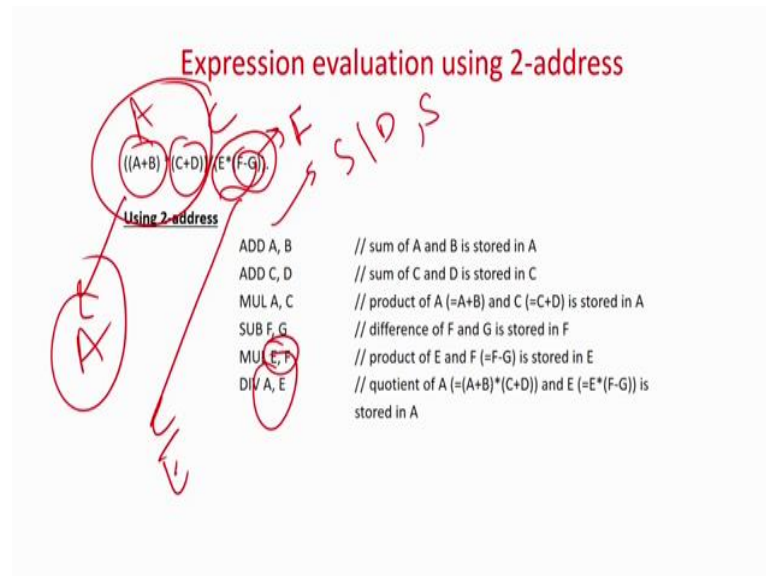


So, we are taking the architecture three address instructions in this case and in this case we are taking say that say add this is the destination and these two are the sources. We are not considering the cases that is the source or destination together we have taken the case that this is a destination 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 subtract F, G the value will be stored 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 store it in L. So, this part is basically now L and then you have to divide J and L. So, J and L you have to divide and you have to store back the value in M. So, finally, it is 2; so, how many instructions 1, 2, 3, 4, 5, 6. So, six instructions actually solves the whole problem for you 1, 2, 3, 4, 5, 6 done.

(Refer Slide Time: 38:03)



Now, next two address. So, in this case as I told you this is both the source as well as destination and this is the source. So, now, what we are 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 store it in C. So, now, this one is A this one is C.

Now, you say that F - 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 - G. So, this one will now become F, then you multiply E and F you multiply these two then this whole thing will become your E and then finally, you add A and E and the store the value.

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