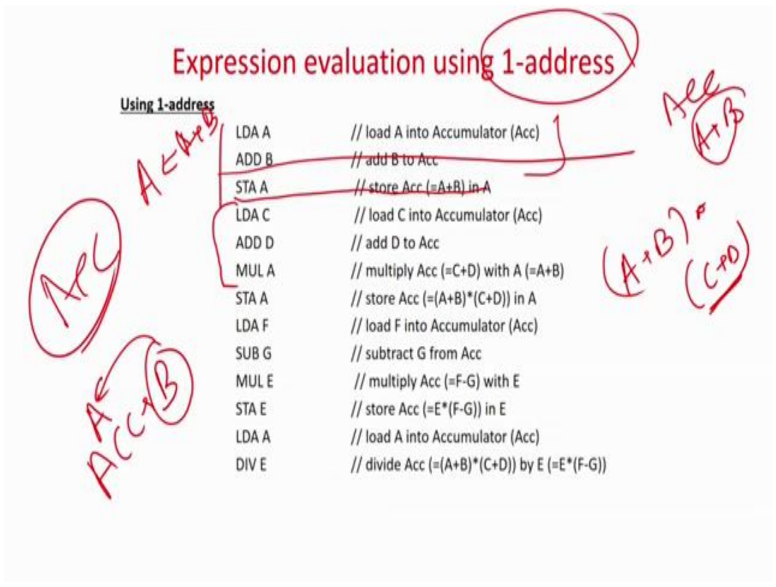
(Refer Slide Time: 38:51)



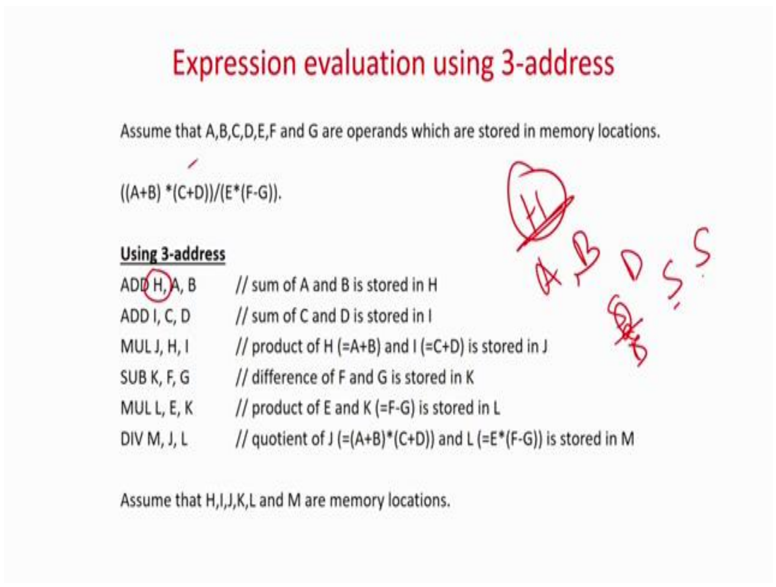
Expression evaluation using 1-address

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
```

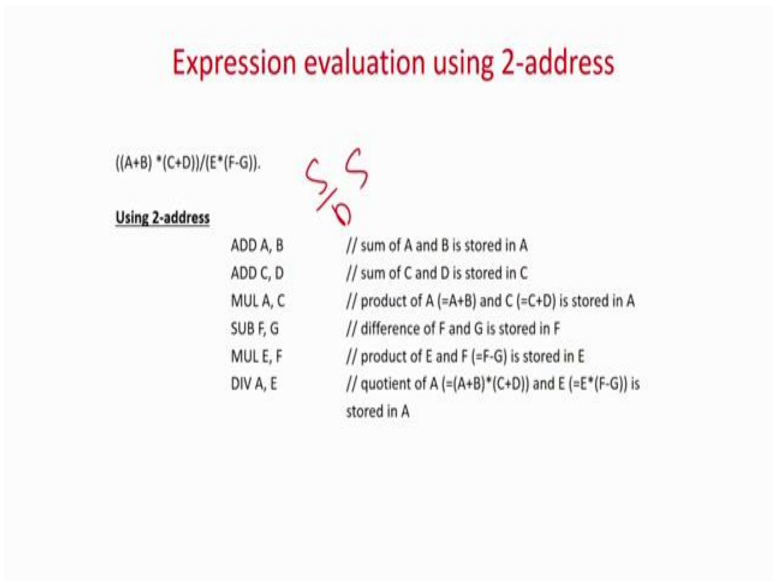
Assume that H,I,J,K,L and M are memory locations.

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 we have kept the format like, A and B and the value is stored in H and using this H as a only destination, but as a homework 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 for this code, but you can easily find out codes where the length can be a made 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 effectively these two formats have become similar.

(Refer Slide Time: 39:51)


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
```

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 the real class comes. Now, I am taking a single address instruction, now de facto is accumulator everywhere accumulator is de facto. Now, big problem that is in this case; what we have done? we have said that ADD A, B and the store the value of A, C, D value is in C, but if you do not write anything in a single address instruction, the we go with the de facto is accumulator.

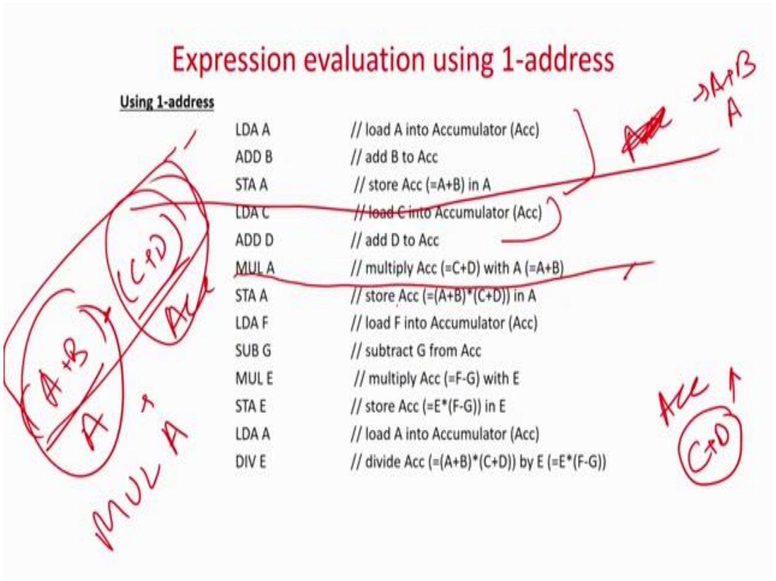
But, accumulator is only one. So, what you have to do? You have to use the accumulator and at the same time you have to again free it, before it can be used for some other like; for example, what was are answer (A + B) * (C + B). So, what is the first thing? (A + B) * (C + D) this will be correct.

Now, what I said ADD A so, what it will do it will add the value of don't say it's 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 there in the accumulator? The value of B. The value of A was in the accumulator you load the value of B to A and store back 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 is now after these two steps you have the accumulator which is having the value of A + B; now may be you will be using the accumulator to do C + D, because accumulator is only one.

So, I have free it 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 + B. Now I say that store A; that means what? That is A + 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 + B. So, in nutshell what happens I load the value of A in accumulator, I add the value of A + B and store it in the accumulator and I 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))
```

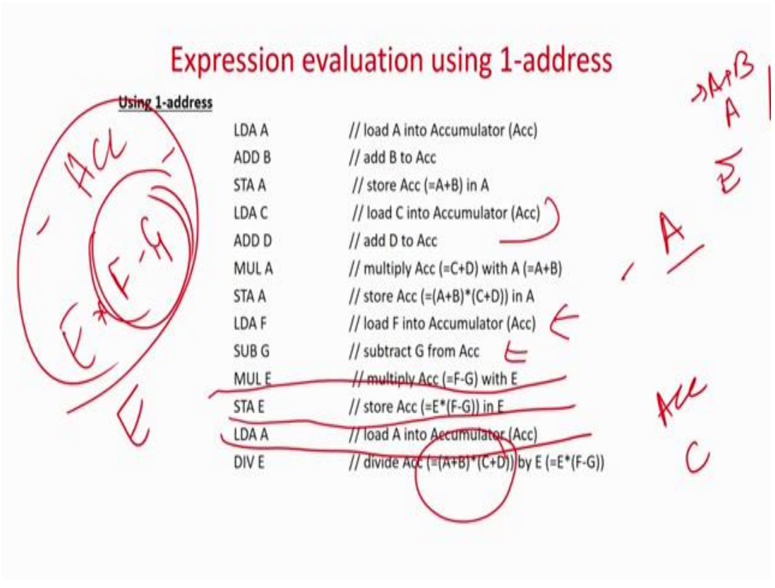
Similarly, these two things will do the same thing. So, what I am doing load C. Now accumulator will let me just erase it. So, up to this accumulator has the value of A + B. Now accumulator is free basically accumulator is A, is having the value of A + B and accumulator is free now. So, in this case 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 + D in the accumulator. Now, what I had to do? My basic operations will be performed in (A + B) * (C + D). So, this one is now stored in A and C + D is in the accumulator.

So, you can write an instruction MUL A; that means what C + D? Which is already in the accumulator; will be multiplied with A which now has the initial value A + B and accumulator will have the value of this whole thing and that is the first part of the expression A + 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 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 = F - 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))
```

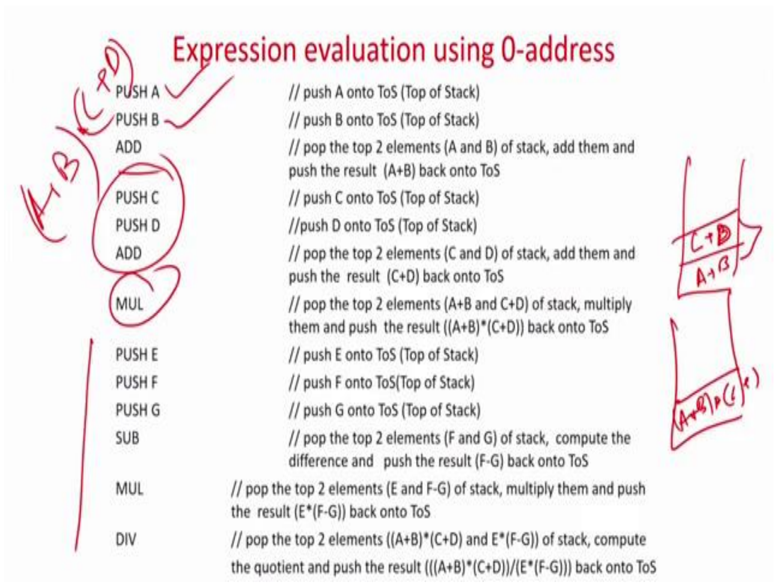
The next expression E * (F – G), E * (F – G) if you do this. So, now, you can see, what I have done? I have loaded the value of F in the accumulator I have subtracted it with G. So, now, the accumulator will have F this accumulator will have F - G now you multiply with E. So, this is done. So, now, the whole accumulator will have this.

Now again you store E; that means, this whole value will be now stored in E, because the accumulator has to be free. So, I am freeing the accumulator multiply this, then I am storing the value of accumulator to E the accumulator is freed. Now, I again load the accumulator with A, because A was having this whole value and then you save it to E. 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 = A + B * C, that part of the equation is loaded over there and then you do this operation and you are done. So, you can if you look at this program carefully you will easily absorb it now, you will be 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 is a single accumulator and if you 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 bring new value from the memory location to the accumulator do it. That is why if you are using 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 a source and destination and take an expression which we have three variables at a time like A + B + C B, C + A + B + C * C + D + 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
```

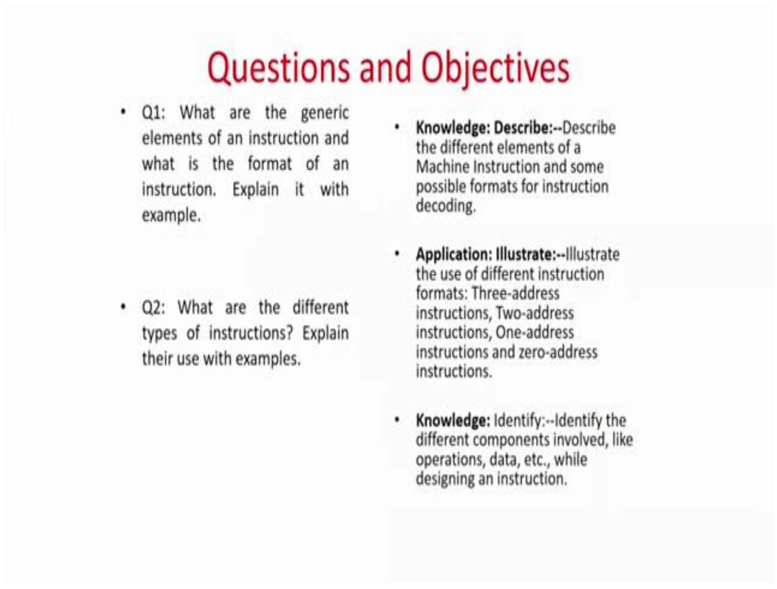
So, generally the de facto standard is less number of lesser number of addresses shorter will be the instructions, but more number of instructions for a code and the other way around. So, now, the before we complete let us take a very simple that same example with the zero address we can go through the whole code, but I will just tell what happens. So, for example, first you have to put A + B it will have you have to do (A + B) * (C + 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 + 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 + B over here. Now, you have to do C + D then you have to say push C push D. So, C and D 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 + B; sorry C + 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 + B) * (C + D) 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 your answer done.

So, in a so generally the number of stack based operation will be more in number compared to even a single address instruction or zero address that is stack based if the more number of instructions compared to a one address instruction. So, this is very 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 if 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

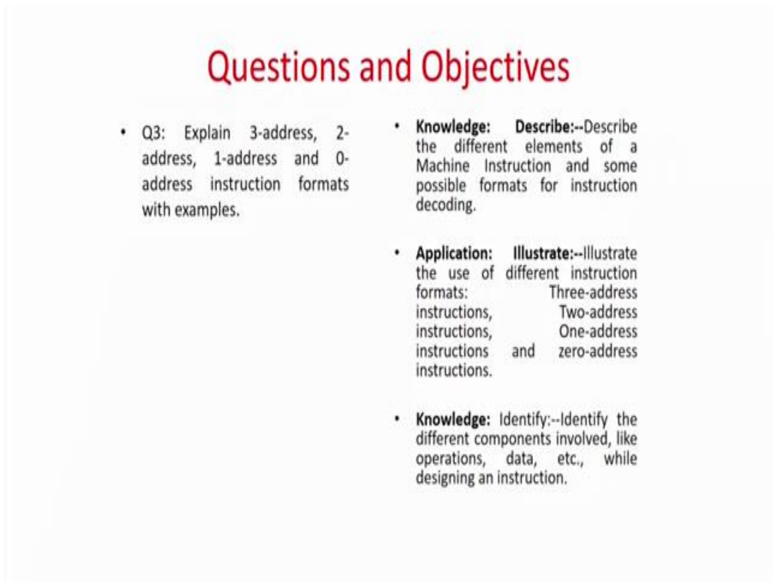
- Q1: What are the generic elements of an instruction and what is the format of an instruction. Explain it with example.
- Q2: What are the different types of instructions? Explain their use with examples.

- **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.

Ok so, that is what basically we have in this unit where we have showed basically; what is the basically 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 with 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 explain 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 will take a data business. Now, explain three address, two address, one address 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 work like as I told you add can be a very different type that is 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 we will go to the instruction set design ok.

Thank you.