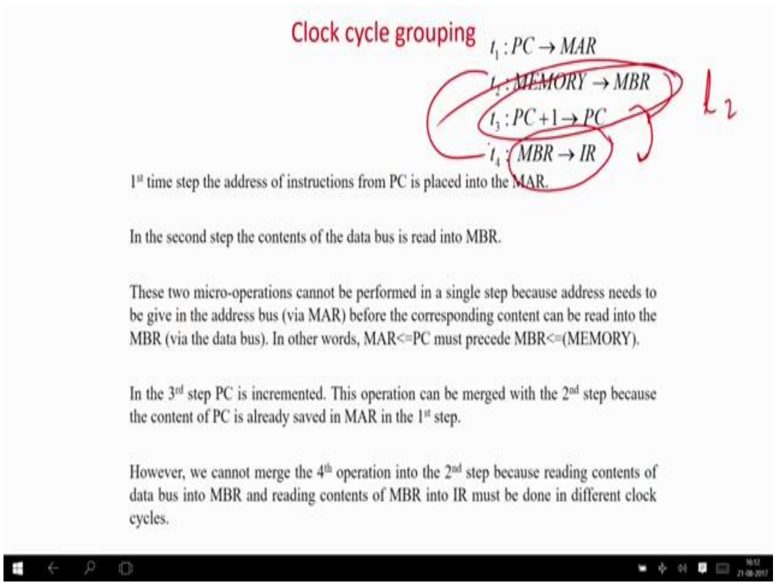
(Refer Slide Time: 32:45)



Clock cycle grouping

$t_1 : PC \rightarrow MAR$

$t_2 : MEMORY \rightarrow MBR$

$t_3 : PC+1 \rightarrow PC$

$t_4 : MBR \rightarrow IR$

1st time step the address of instructions from PC is placed into the MAR.

In the second step the contents of the data bus is read into MBR.

These two micro-operations cannot be performed in a single step because address needs to be give in the address bus (via MAR) before the corresponding content can be read into the MBR (via the data bus). In other words, MAR<=PC must precede MBR<=(MEMORY).

In the 3rd step PC is incremented. This operation can be merged with the 2nd step because the content of PC is already saved in MAR in the 1st step.

However, we cannot merge the 4th operation into the 2nd step because reading contents of data bus into MBR and reading contents of MBR into IR must be done in different clock cycles.

So, in this fetch stage so, PC is equal to you are keeping the value of the memory address register can I actually merge this 2 in a single time step not possible, because in the first unit the value of the program counter will go to the memory address register, you give some time give one unit of time for that, then the memory address register will be read, now the memory will know that I have to supply the data which is actually the instruction from the address, which is given in the memory address register.

So, if I merge this 2 together there will be a lot of hotspots, because I you tell me that I have to deliver from this address and deliver it now. How can I do that I require some time to read the address, go to the relevant location, bring the data and give it to you. So, I cannot merge t1 and t2 at a single point of time, but if you look at it I am giving some because this job is over, you have told me what is the address.

Now I go and bring the data from that memory location already you have given me from where I have to fetch the data. Then the job of the program counter is over, because program counter is telling me the address from where I have to give from the get the data from.

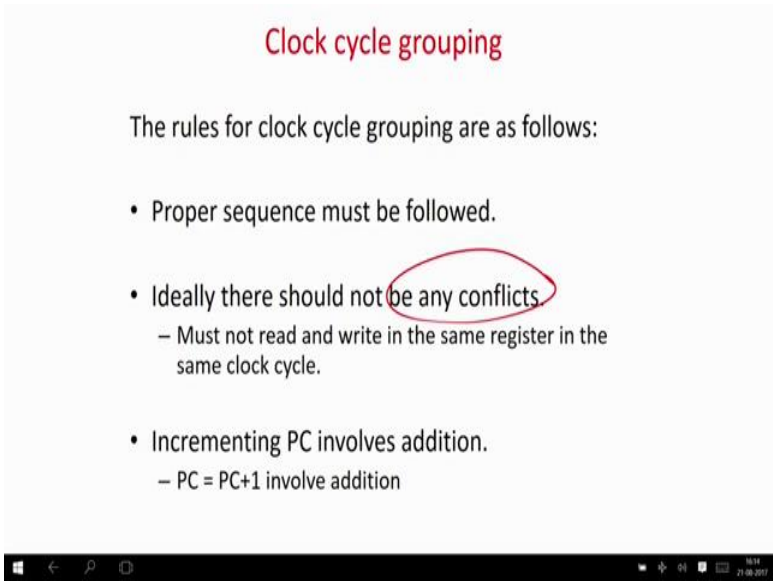
So, you have told me the address and now I am going to get the data from the memory. Now the program counter is free. So, better you can increment the program counter. So, it is very obvious I can merge step 2 and 3 together, because after reading the value of the program counter to the memory address register my job is done I am free at hand.

So, if you are free at hand then actually I can reuse the PC that indicates I am incrementing it by one. So, memory, taking the data from the memory to the memory buffer register and program counter increment these 2 micro instructions you can do it in time steps 2, because they are 2 non-dependent micro instructions. Of course, this then the last is memory buffer register we will be writing it to the instruction register of course, PC you can be merged with t2 PC increment can also be merged with t4. So, any way you can merge.

Of course again I cannot merge step 2 and step 4, because step 2 actually tells that I am bring the data from the location. Before the data is brought to the memory buffer register you cannot directly transfer to the IR. So, these 2 are memory interdependent instructions and 2 and 4 are interdependent instructions. So, there cannot be merged, but this one is an independent instruction even this 2 are dependent. So, you can either merge it with this or you can merge it with this.

So, basically 3 time steps or micro instructions are required or 4 micro instructions are required, but in 3 time steps you can complete the fetch stage, that is what is saving is called a clock grouping that I am grouping. So, either I am grouping basically these 2 guys together or I can merge this 2 guys together. So, 4 micro instructions, but I can do it in 3 time steps.

(Refer Slide Time: 35:27)



Clock cycle grouping

The rules for clock cycle grouping are as follows:

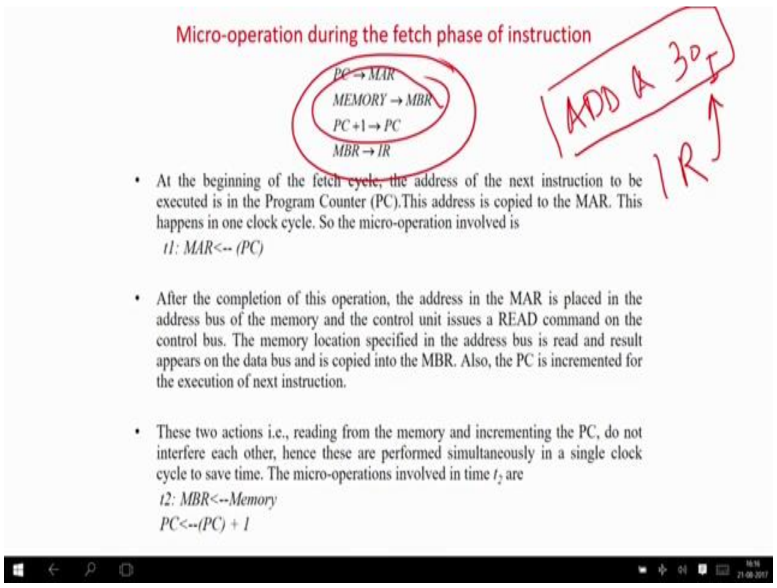
- Proper sequence must be followed.
- Ideally there should not be any conflicts.
  - Must not read and write in the same register in the same clock cycle.
- Incrementing PC involves addition.
  - PC = PC+1 involve addition

So what is clock grouping proper sequence should be maintained you cannot alter the sequence that is I am getting the data; before I put the registers in address that cannot be done. Proper sequence should be follows; ideally there should not be any conflicts that is there cannot be any biasness I mean there can't be any what we call race condition kind or a actually conflict of interest kind of a thing that is I want to read from that register at the same time you cannot write to the register. Those instructions are called conflicting instruction, because they are using a common resource either for read and write.

So, you cannot do at the same time and program counter is always involved in an increment. So, you try to fit the program counter at some place after the data has been read or after the instruction has been read, then the program counter is free you can try to increment the program counter and try to fit it in some place where there is no conflict.

That is how basically the ideas of clock grouping is very simple, sequence you cannot change you cannot have any conflict, like if I want to read from register R1, you cannot put a micro instruction at that same level in which there is an updating. Similarly we have given an example and, but program counter is bit free because whenever the program counter has given the data to the memory address register to fetch the instruction it is free you can use it and put it in parallel.

(Refer Slide Time: 36:42)



Micro-operation during the fetch phase of instruction

$PC \rightarrow MAR$

$MEMORY \rightarrow MBR$

$PC+1 \rightarrow PC$

$MBR \rightarrow IR$

- At the beginning of the fetch cycle, the address of the next instruction to be executed is in the Program Counter (PC).This address is copied to the MAR. This happens in one clock cycle. So the micro-operation involved is
  t1: MAR<-- (PC)

- After the completion of this operation, the address in the MAR is placed in the address bus of the memory and the control unit issues a READ command on the control bus. The memory location specified in the address bus is read and result appears on the data bus and is copied into the MBR. Also, the PC is incremented for the execution of next instruction.

- These two actions i.e., reading from the memory and incrementing the PC, do not interfere each other, hence these are performed simultaneously in a single clock cycle to save time. The micro-operations involved in time t_2 are
  t2: MBR<--Memory
  PC<--(PC) + 1

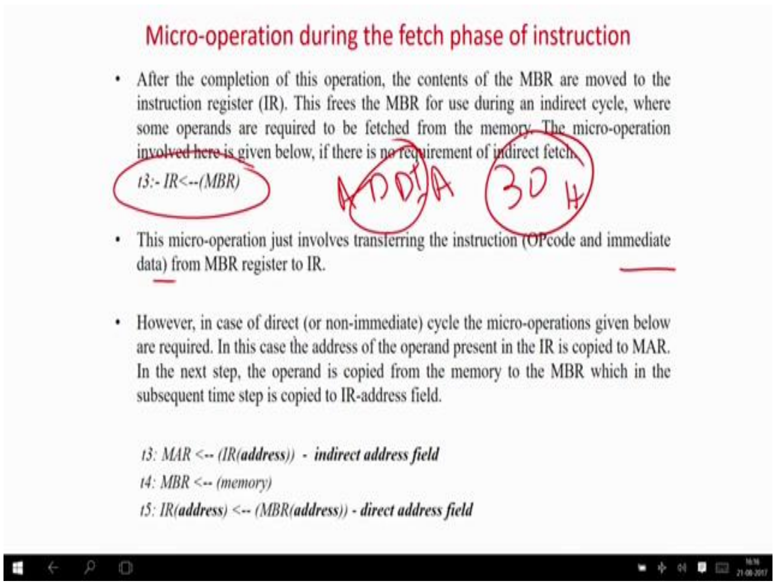
Again we will now see this stage we have already discuss discussing for the fetch.

So, first stage is program counter value to the memory address register, memory buffer register will take the value from the memory, you can merge this 2 and the memory buffer register will write to the instruction. This is very simple that is for example, if you have an instruction say

ADD accumulator 30 immediate. For such an instruction this is very simple, first you will fetch it then actually may be from this program counter will give the value in the memory address register. So, in that memory address let us say that this instruction is there. So, PC will be incremented and then the instruction register will have this part.

Now, you want to actually execute this. Then exactly what happens basically depending on whether 30 is an immediate or 30 is a memory location the number of instructions in the decode phase will change, that is we are going to look at it.

(Refer Slide Time: 37:40)



Micro-operation during the fetch phase of instruction

- After the completion of this operation, the contents of the MBR are moved to the instruction register (IR). This frees the MBR for use during an indirect cycle, where some operands are required to be fetched from the memory. The micro-operation involved here is given below, if there is no requirement of indirect fetch.

t3:- IR<--(MBR)

- This micro-operation just involves transferring the instruction (OPcode and immediate data) from MBR register to IR.

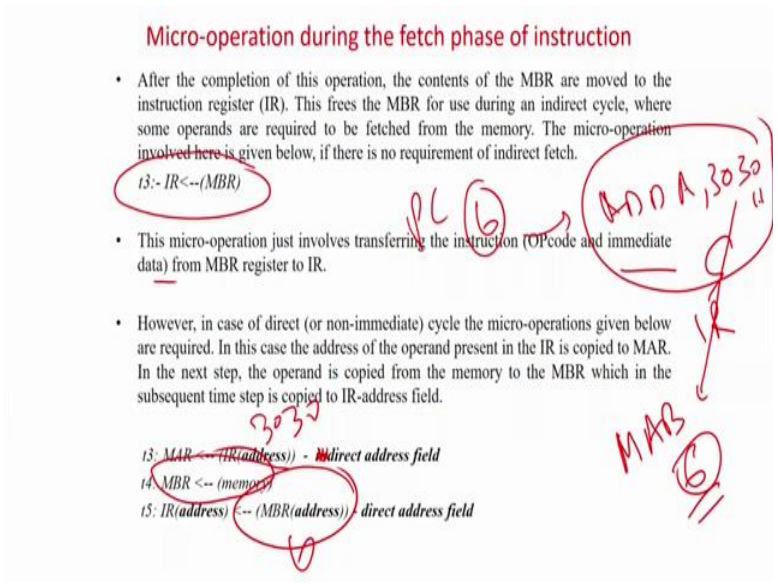
- However, in case of direct (or non-immediate) cycle the micro-operations given below are required. In this case the address of the operand present in the IR is copied to MAR. In the next step, the operand is copied from the memory to the MBR which in the subsequent time step is copied to IR-address field.

*t3: MAR <-- (IR(**address**)) - **indirect address field***
t4: MBR <-- (memory)
*t5: IR(**address**) <-- (MBR(**address**)) - **direct address field***

So, in the third stage it says that the memory buffer register will write to the instruction register. So, if it an immediate data more or less our job is done, because now your instruction register basically has the memory buffer register which is having the data called ADD accumulator 30 H which is a may be immediate.

In case of immediate you did not do anything, because when the memory buffer register has dumped the value in an instruction register, you have already fetched that may be this ADD immediate opcode corresponds to immediate. So, you can directly take and there is no further micro instructions required, but assume that in this case it is a indirect or it can be a direct, but it is a non-immediate mode of addressing, then in this case what happens?

(Refer Slide Time: 38:28)



Micro-operation during the fetch phase of instruction

- After the completion of this operation, the contents of the MBR are moved to the instruction register (IR). This frees the MBR for use during an indirect cycle, where some operands are required to be fetched from the memory. The micro-operation involved here is given below, if there is no requirement of indirect fetch.

  t3:- IR<--(MBR)

- This micro-operation just involves transferring the instruction (OPcode and immediate data) from MBR register to IR.

- However, in case of direct (or non-immediate) cycle the micro-operations given below are required. In this case the address of the operand present in the IR is copied to MAR. In the next step, the operand is copied from the memory to the MBR which in the subsequent time step is copied to IR-address field.

  t3: MAR <-- (IR(address)) - direct address field
  t4: MBR <-- (memory)
  t5: IR(address) <-- (MBR(address)) - direct address field

Say it is let us take ADD accumulator 3030 Hex. So, it is a direct addressing mode. So, we are doing that take the value from the memory location 3030 Hex, you go to the memory location called 3030 whatever data is there you ADD it to the accumulator and store it.

So, in this case what happens this one will come to the accumulator. So, this whole thing is now. So, this instruction or the value of 3030 is now in the instruction register, but now what happened we have to now again give this value of 3030 to again to the memory address register, because this is some location called say 6. Say program counter value was 6. So, program memory location 6 had this instruction. So, it is now in the instruction register but now again you have to now load the memory address register with this 3030.

Then again we have to read the value of memory location 3030 to the memory buffer register and only that can that will again brought into the instruction register and then again you can add it, that means a 2 stage process. Immediate means you can directly whatever is in the memory buffer register like we have seen ADD accumulator 30 you can directly take in the buffer register, go to the instruction register ADD it and you are done. There is no further more steps required, but if it an indirect mode or a direct mode and non-immediate. That is data is not available in the instruction itself then you have to go for multiple stages. Like instruction register address you have to again feed it to the memory address register that is this 3030 again I have to feed it to the memory address register, next memory buffer register memory will right to that.

For example in 3030 we have the value called 6; that means, we are going to ADD the value 6 to A and store it back to the accumulator, that is memory location 3030 let us assume that has the value 6. So, in step 3 basically you are going to read the value of ADD A, 3030 hex in the instruction register.

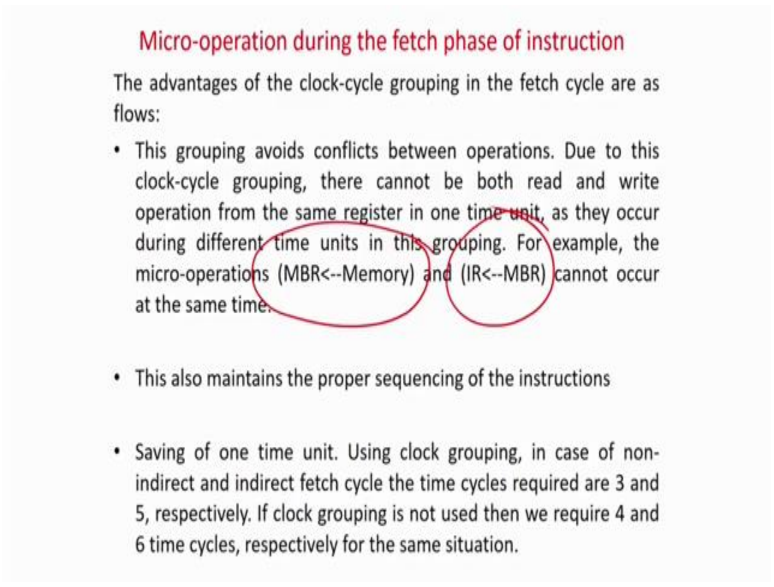
Now instruction register will know that again I have to go for it's an indirect. So, it's a basically direct addressing, but a non-immediate addressing. So, it will again load the value of 3030 in the memory buffer register. So, memory buffer register ah memory address register will have the value now 3030 and now the memory buffer sorry the memory address register will take the value of 3030 from the instruction register, memory ad buffer register will take the value from the memory location 3030 and put it to the memory buffer register.

And then this memory buffer register, which is having the value of 6 now will be loaded to the instruction register. So, now, the instruction register will have ADD A, 6. So, it will load it to the accumulator after adding. So, in fact, now some more instructions are required like in this case all this will be required which will solve the problem, but again if you see none of the instructions can be put in one time step. Because in this case you are reading in time t3 you are reading the instruction register address that is 3030 in the memory buffer register. You have to give some time then only the memory address register will be read and you are going to give the value in the memory buffer register.

After sometime the memory buffer register address that is the data from the address that is 6 will be loaded to the instruction register. So, in fact, you cannot parallelize because this one is dependent on this and this one is dependent on this. So, you cannot save any time step in that. So, only time step which we save is that in time step 2 we have merged the memory buffer register writing from the memory and we add program counter.

So, what we have seen. So, if it is a simple immediate mode of addressing then we require 3 steps 1 2 and 3, that is in step 1, this can be considered step 2 and this can be considered step 3 and if it is a non-immediate mode; that means, may be a direct mode for indirect mode may be several other steps will be required, because more complicated your instruction is more number of micro instructions will be required. So, is that in case of a direct we need another 3. So, total number of stages time steps will be 5.

(Refer Slide Time: 42:25)



Micro-operation during the fetch phase of instruction

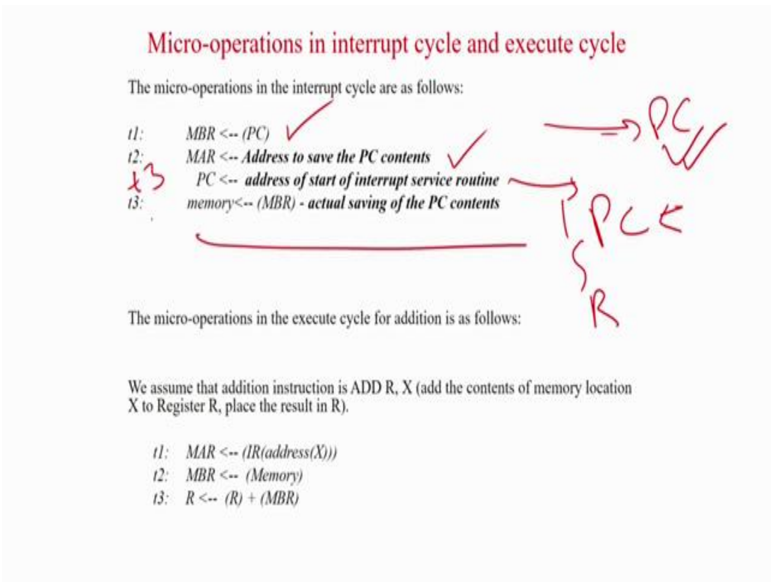
The advantages of the clock-cycle grouping in the fetch cycle are as flows:

- This grouping avoids conflicts between operations. Due to this clock-cycle grouping, there cannot be both read and write operation from the same register in one time unit, as they occur during different time units in this grouping. For example, the micro-operations (MBR<--Memory) and (IR<--MBR) cannot occur at the same time.
- This also maintains the proper sequencing of the instructions
- Saving of one time unit. Using clock grouping, in case of non-indirect and indirect fetch cycle the time cycles required are 3 and 5, respectively. If clock grouping is not used then we require 4 and 6 time cycles, respectively for the same situation.

So, what I was telling you is that we are writing in the slide that what are the dependent dependency like for example memory, memory, memory to memory buffer register and instruction register to the memory. We cannot put these 2 micro instructions together, because first the memory will memory value will go to the memory buffer register after sometime from memory buffer register it will go to instruction register, you cannot merge these 2. So, you have to have a proper sequence.

So, what we have seen if you are not going for any clock grouping then the number of time units required to go for a fetch immediate instruction will be 4. And if you optimize it one will be saved that is the incrementing of the PC will be saved then you require 3. Similarly if this is a non-immediate mode like a direct mode of instruction. So, this is either 5 or 6, because we are optimizing the program counter increment which can be fit with any other stage.

(Refer Slide Time: 43:17)



Micro-operations in interrupt cycle and execute cycle

The micro-operations in the interrupt cycle are as follows:

```
t1:     MBR <-- (PC)
t2:     MAR <-- Address to save the PC contents
t3        PC <--  address of start of interrupt service routine
t3:     memory<-- (MBR) - actual saving of the PC contents
```

The micro-operations in the execute cycle for addition is as follows:

We assume that addition instruction is ADD R, X (add the contents of memory location X to Register R, place the result in R).

```
t1:   MAR <-- (IR(address(X)))
t2:   MBR <--  (Memory)
t3:   R <--  (R) + (MBR)
```

Before we end some other micro operations for different other stages of instruction I am showing over here. Say for example, this is for the interrupt cycle. So, if there is a interrupt instruction is there what are the micro instructions. So, in the beginning we all know that the micro instruction when it's interrupt you have to save the value of program counter and whenever the interrupt service routine have been done, you have to again come back and pop up the value of program counter and restart from where we have left.

So; obviously, we have to store the value of program counter value somewhere. So, you write the value of program counter in the memory buffer register, please note that we are not writing the program counter in the memory address register. Program counter writing in the memory address register means you are fetching some instruction. Here we writing the value of program counter to the memory buffer register, because we are saving the value of PC. So, saving the value of PC means the value of PC I am writing to memory buffer register, from memory buffer register it will basically go to a place where it will be saved.

Now, where I have to save it in the memory that will be actually the address to save the PC contain, that is actually a stack address where we save the value of program counter, sometime temporary registers, before the interrupt service routine is started. So, you actually in the; what you do? So, you can see that first stage I write the value of the program counter in the memory buffer register, second stage I write in the memory address register to save the value of the program counter.

So, now basically already in the first stage we have saved the value of your program counter in the memory buffer register. Now I give the address where we have to save the value of the program counter in the memory, but at the first stage PC is now free we have already saved the value of PC in memory buffer register and after that you are going give the address in the memory address register where the value of MBR has to be saved indirectly the PC will be saved, but in the first stage only the PC has saved.

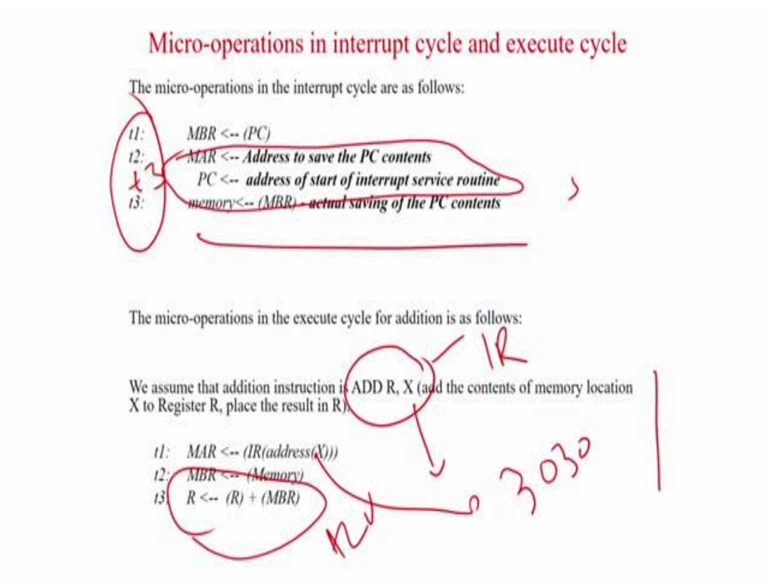
So, in time 2 you can put PC to the starting address of the interrupt service routine you do not require a t3 over here t1 you save the PC in memory buffer register, step 2 you write the address in the memory buffer register sorry memory address register where you want to store the value of PC, at the same time you need you increment the value of PC, because PC is free.

So, in this case you jump to the interrupt service routine and then in the time step 3 the memory buffer register will write the value to the memory, whose address was given in the memory address register. So, in step 3 actually you are physically saving the value of the program counter. So, instead of 3; 4 you can save it in 3 time units.

Basically we have merged because of clock grouping which could merge the memory address register writing and the program counter increment, because already we have saved the value of program counter. Another simple this was all about basically load, store, fetch and some kind of interrupt type of part of the macro routine, but now let us take a very simple instruction like ADD R, X basically that is you want to ADD the value of R to some memory location and memory location is X content of X you want to ADD with R and save it in register R.

So, basically what are the micro instructions involved for this? Fetch already we have seen may be it has been fetched etcetera then what is the case? So, basically this is already in the instruction register, because we assume that it has been fetched.

(Refer Slide Time: 46:30)



Micro-operations in interrupt cycle and execute cycle

The micro-operations in the interrupt cycle are as follows:

```
t1:   MBR <-- (PC)
t2:   MAR <-- Address to save the PC contents
        PC <--  address of start of interrupt service routine
t3:   memory<-- (MBR) - actual saving of the PC contents
```

The micro-operations in the execute cycle for addition is as follows:

We assume that addition instruction is ADD R, X (add the contents of memory location X to Register R, place the result in R).

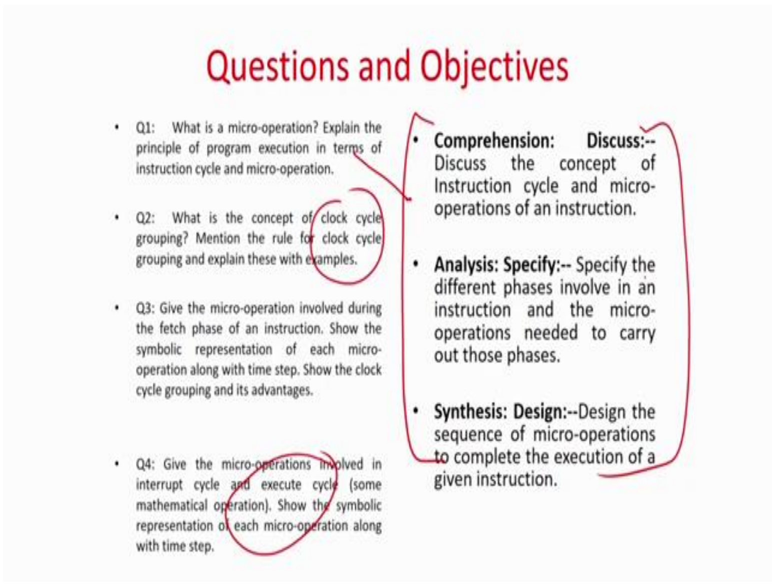
```
t1:   MAR <-- (IR(address(X)))
t2:   MBR <-- (Memory)
t3    R <--  (R) + (MBR)
```

So, in the from the instruction register you have to get the value of X. Address X means the value of X that is 3030 in the example we are looking at. So, memory address register will be fed with the value of address of X.

Now, the memory buffer register after some amount of time we load the value that is of 3030 assuming that is X the value in the memory location X, which we are calling as 3030 in the example that value in that location will be loaded to the memory buffer register in time 2 and in time 3 basically, which is the logic operation the accumulator sorry the arithmetic and logic unit will be involved it will ADD the value of memory buffer register with the register R and it will be saved in R. So, this ADD R, X will be done in 3 micro instructions.

And of course, you can find out that there is lot of interdependence, because neither you can parallelize. Means you cannot put memory this first instruction with second and second with third. So, so they are all interdependent instructions. So, three instructions are required for that.

(Refer Slide Time: 47:27)



Questions and Objectives

- Q1: What is a micro-operation? Explain the principle of program execution in terms of instruction cycle and micro-operation.
- Q2: What is the concept of clock cycle grouping? Mention the rule for clock cycle grouping and explain these with examples.
- Q3: Give the micro-operation involved during the fetch phase of an instruction. Show the symbolic representation of each micro-operation along with time step. Show the clock cycle grouping and its advantages.
- Q4: Give the micro-operations involved in interrupt cycle and execute cycle (some mathematical operation). Show the symbolic representation of each micro-operation along with time step.

- **Comprehension: Discuss:--** Discuss the concept of Instruction cycle and micro-operations of an instruction.
- **Analysis: Specify:--** Specify the different phases involve in an instruction and the micro-operations needed to carry out those phases.
- **Synthesis: Design:--**Design the sequence of micro-operations to complete the execution of a given instruction.

So, basically in this unit what we have seen in this unit basically we have got an idea that there are macro instructions and each macro instructions must be implemented in terms of some lower level instructions, which are we calling as micro instructions. And how micro instructions basically together are responsible for implementing a macro instruction and each micro instructions are so finite or so atomic we need not break it down into further level.

Basically each microinstruction involves kind of some kind of control signals which will actually make them operate, that we are going to see in the future units to be coming, but macro instructions micro instructions at the module at the unit level or the atomic level together they make the macro instructions and a code executes. For each micro instructions we will see that there are some kind of signals, which are generated, we will see in the future units how to generate the signals and how basically they involve in real hardware to execute, like if I say ADD, how the corresponding micro instruction? What are the signals it generates to exactly make it add the 2 operands by the CPU ok?

So, before we end basically let us look at the some of the questions, like first question is what is micro instruction? Explain the principle of program instruction in terms of micro instructions and micro operations. And second instruction if you look at it this these are the objectives, like discuss the concept of instruction cycle and micro instruction operation. So, directly if you are able to answer this question this instruction this objective is satisfied. What is the concept of clock grouping like specify the different phases that are involved in the micro instruction to carry out those phases? Design the sequence of micro instructions to complete instruction execution?

So, if you are able to specify and design of course, you will be able to optimize it in terms of clock grouping. Give the micro instructions involved in fetch instructions, give the micro instructions involved in the interrupt cycle and execute cycle. So, these questions basically we have this questions for assignment, when you will be solving this questions basically you are easily going to satisfy these objectives. Like the first one is basic concept of micro instructions, third and fourth questions basically tells you about different phases of the instructions and basically what are the micro instructions involved in it.

So, basically these questions actually satisfy these objectives. So, after solving I mean after discussing this unit you should be able to solve this questions and hence satisfy the objective. And whenever you are going to try to optimize this sequence then the concept of clock grouping is coming.

So, basically with this we come to end of the first unit, from the second unit onwards basically we will be more specifically looking into the control signals, which are generated by each of the micro instructions, which exactly or accurately result in the flow of the code.

Thank you.