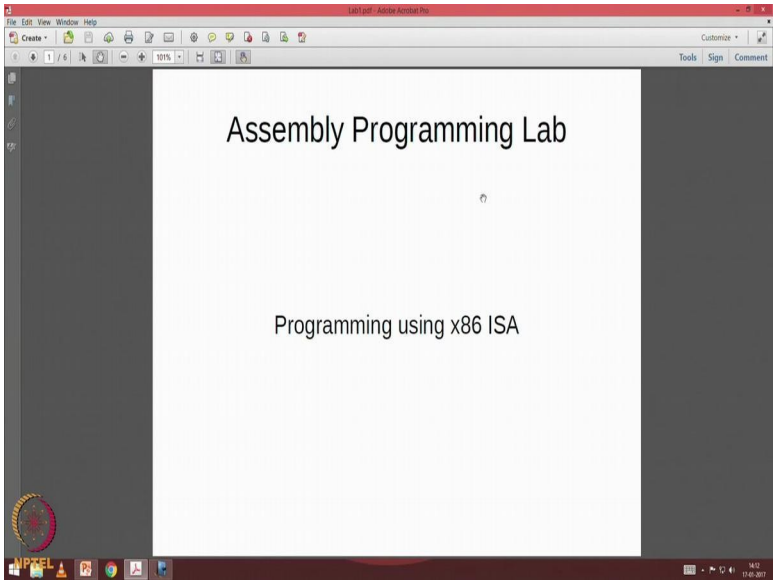
Computer Organization and Architecture
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 07
(Part – I)
Programming using X86 ISA – Addressing Modes

(Refer Slide Time: 00:28)



So, today I will just introduce you to the x axis assembly, it is like you know understanding assemble language is like reading the dictionary actually you do not view the entire dictionary before starting speaking the first sentence in English right correct, if you read the entire dictionary before you want to start a drink the first sentence whole life will be reading only dictionary. So, what we will do is that I will give you some very quick snapshot and from there you start working and we were we will give a link to the Intel manual is the around thousand pages again whenever you need you can refer that manual we will teach you how to refer that manual and their you start and we go forward on this.

Please note that what we are trying to do in the lab is really the crucial part of this course this is what we going to you will be practical using unless you do you go to industry where you manufacture chips or actually start designing architecture its probably I will be happy even if you know one percent of the total population I have educated in last 16

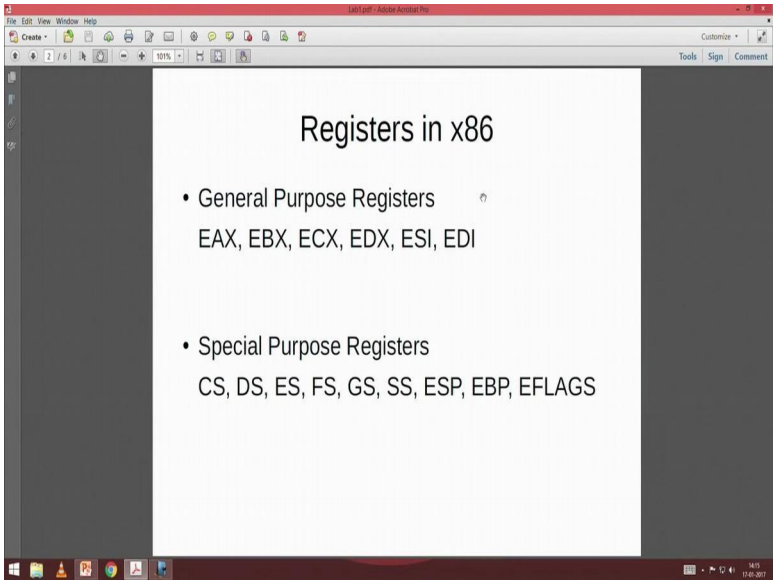
years go and do that I will be happy, but many many of you would land up in jobs which are going to use some of these concepts in the lab not from the theory class, but theory has to be taught. So, that you appreciate the lab, but unless you do this lab this whole course is a junk waste scrap whatever you want to call it all synonyms of that right.

So, we will concentrate much on this lab and at the end of this we will write a small decent in the sense complex enough mini kernel of an operating system in assembly language and that is something very very important and very interesting and as you know the x 86 which is followed by Intel and or one of the most complex instruction set computer ever in the history of computer science ever done by mankind. So, the most toughest of the architecture and instructions said. So, once you understand this understanding any other architecture I can write and give you in blood would be the most easiest thing in your life it will be child's play.

So, I am going to put you into that grill. So, that everything else in this world becomes very easy in the area of computer organization. So, let us let us do the best on this lab I have been doing it for last 10 years now and there are students have finished within the first month they have finished all the 4 exercises there are cases. So, is you actually spend some time understand all the videos are now available the videos are shared you actually spend some time in understanding those videos and doing those exercises you will be much at a you will be in certainly in a better advantage and if you win we will give you all the 4 exercises it will not be exactly the same there will be some difference and if we can complete those exercises easily then the meet February, our first week of march you can wind up this lab and so that will be good for you good for us good for everyone.

So, with all these nice things let me start this education of x 86, right.

(Refer Slide Time: 04:09)



The first thing is that there is something called a memory hierarchy, right, what do you mean by a memory? What is a hierarchy that some there are some entities above that there are some more entities about that there are some more entities and like that. So, you build hierarchy means layers in which one layer sits in top of the another layer and then one on top of it and one on top of that. So, one of the interesting hierarchy that we will see in this course is what we term as memory hierarchy memory hierarchy means what. So, we have varieties of memories the fastest of the memory is the register which you use like a scratch pad right which is inside your CPU. So, you just do some small computation it is a temporary result you just keep it in the register and then reuse it then there is something called cache memory which is nothing, but a replica of the main memory, but smaller in size then there is a main memory and then after that there is a hard disc and all this everyone were using everyone of you who use desktop will know about all these things, right.

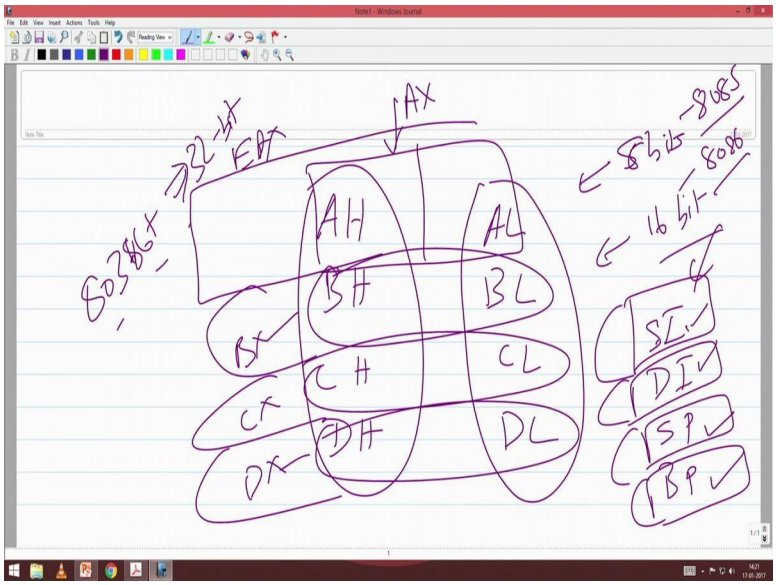
Now, the fastest of the memory is basically call registers and let us see how. So, in the x 86 we have actually 8 general purpose registers in which 2 of them are used for certain special purpose also. So, you see a EAX, EBX, ECX, EDX, ESO, EDI, these are 6 general purpose registers which could be used for storing some temporary results and then as you see down in the special purpose register your ESP and EBP; the ESP is basically called a stack pointer and the EBP is a base pointer the usage of these will be also we will discuss as we

proceed in this lecture now this ESP and EBP can also be used

as general purpose registers for storage purposes right, but normally they are not used they are used for certain specific purpose, but nothing stops you the Intel x 86 does not stop you from using ESP and EBP as general purpose registers.

Now, why this is EAX right now there are also very interesting for us to find out some history here the first processor that came in existence was an 8 bit processor, right.

(Refer Slide Time: 06:52)



Which had some general purpose registers namely we call it as AH, AL, BH, BL, CH, CL, DH, DL, there were right general purpose registers each of right bits because this was an 8 bit architecture what do you mean by k bit architecture a k bit architecture is one which can handle K bits at a time for example, it can handle it and add 2 k bit numbers it can multiply 2 K bit numbers right; right. So, that is what we mean by a k bit architecture.

So, initially Intel hand in an 8 bit architecture these are all the 8 general purpose registers each having 8 bits right then came as 16 bit architecture, but Intel did not want to keep changing lot of things. So, they wanted 16 bit registers. So, then they just merge these 2 and call it as AX merge these 2 and call it as BX merge these 2 and call it as CX and DX. So, in the Intel mission still I can use ax; that means, 16 bits I can still say AH or AL which means the first 8 bit and the second 8 bit of that 16 bit. So, I can access AX as 16 bit in total

if it is a 16 bit operand, still I can use AH and AL if there are 8 bit operands

can you give me example of a 8 bit operand one 8 bit data type characters right characters are still 8 bit.

So, if I am going to work with ASCII, I can still used these smaller version of the system, Intel does not deserve separate AX, AX is nothing, but the merging of AH and AL. So, if I go and change AL the value of AX will also change correct then it introduces it introduced 4 more registers SI, DI, SP, BP, right, this SI and DI are basically index registers we will we will talk about it later SP and BP are the stack pointer and the base pointer again usage of that we will start.

So, these are all 16 bit registers then this 16 bit was the 8068 bit was the 8085 then came 16 bit which is 8086, then came 8686 plus other machines for quite long time this was 32 bit architecture. So, what Intel did was it used this ax as the part and it extended this AX by another 16 bit and we call it as EAX, similarly EBX, ECX, EDX, ESI, EDI, ESP, EBP. So, today when we look at EAX ax is a 32 bit register in which the first 16 bits can be called as AX and you can access it individually in that AX the first 8 bit is called AL the second 8 bit is called AH still I can go and access AL and AH individually, right. So, this is basically how the general purpose registers are used and these register are used for storing some intermediate computation will give you some examples as we proceed in this lecture in addition to this general purpose registers there are certain special purpose registers and some of this special purpose registers used or called segment registers we will tell you what are segment registers later now there is something called code segment data segment extended segment another the extended FS data segment, GS another extend segment SS is the stack segment E flag which are flag registers.

So, there are many set of special purpose registers we will understand the usage of the special purpose registers as we proceed in this course we will also understand the usage of the general purpose registers. So, we proceed in this course so, but what I what is the take away from this slide the take away from this slide is there is something called a general purpose register which is very fast storage which are limited in number we have 6 or 8 of them and so and the some you know seven of special purpose registers. So, these registers are inside the CPU and they can be accessed very fast. So, that the program the compiler when it complies the program uses this general purpose registers for quick storage and retrieval right it need not go to the memory, memory is at least 2

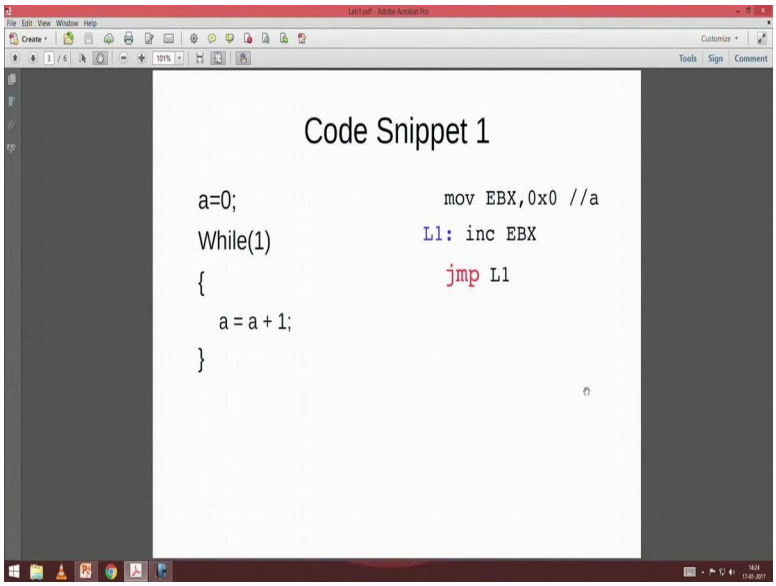
orders of magnitude lower than the CPU. So, I will not go and write everything into the memory rather I can have a scratch pad or is also called scratchpad register.

So, that I just keep the temporary answer and take it back. So, this is about registers in x 86 in contrast what we did in the third semester where we had in the architecture we are only how many registers.

Student: 2.

Two registers right now we have 8 now let us take this very simple code snippet I want to convert the; so one of the important thing is that the first as I told you whatever we try to learn in this course will have direct implication on operating systems and direct implication on compilers, right.

(Refer Slide Time: 13:27)



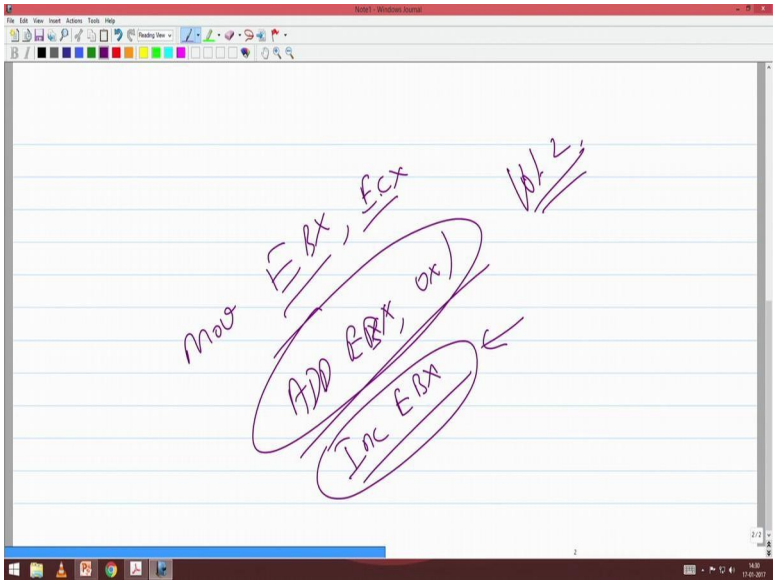
So, whatever we learn in computer organization will be from the compiler point of view and from the operating system point of view right. So, now, we will look at from the compiler point of view. So, this is a c code what you see on your left hand side a equal to 0 while one a equal to a plus one very simple code. So, how do we convert it into x 86 assembly slowly one by one we will read the instructions and many of these instructions see the out of ninety percent of the instructions are not used only 10 percent of the instructions are extensively used.

So, learning even 10 percent is around seventy not even five percent or even 2 percent of the instructions are widely used right some so; that means, we have to learn some 20-25 instructions at the most and we can do a fair decent conversion of your c programming into assembly program for example, let us look at this let us assume that a is going to be mapped onto a register EBX, right. So, I say move EBX comma 0 the syntax is like this move in says instructions which will move something from the right operand to the left operand right operand is 0, the left operand is EPX, EBX is a register this move instruction will move 0 into EBX, now if I say EBX is representing a of in my program then a actually gets a value 0, right.

Now, I increment EBX because a is mapped onto EBX increment EBX I and C means increment; increment by 1 and what I do I to jump L one jump is an unconditional jump surely you should jump jumped L 1; L 1 is label. So, this will keep on incrementing. So, there is essentially an infinite loop while one is an infinite loop and that while one gets actually mapped onto this. So, this is a very simple translation of what you see on the left to a coded assembly. So, what you learn from this code is a very trivial code, but what you are learning is that for example, we learn and instruction call move in which the destination operand is the first one and the source operand is the second one. So, whatever is from the second; second is moved to the first right and there are 2 operands to move one is the source from which you take and put it into the destination.

Now, the source operand in this case is as immediate operand it is a constant that is why we call it as an immediate operand immediate means the value of what you want to move is immediately available with the instruction current. So, that 0 x 0 what do you mean by 0 x; 0 x stands for hexadecimal. So, 0 x essentially means whatever is going to follow 0 x is going to be a hexadecimal value and that is 0. So, in this move EBX comma 0 x 0 2 addressing modes used one is called the immediate addressing mode another is called the register addressing mode in a register addressing mode I specify which register it is in the immediate addressing mode I just give the value that I want move in contrast even if we had if we are done something like if we have done something like move EBX, ECX both are register addressing mode.

(Refer Slide Time: 16:27)



Now, the content of ECX should be moved to into EBX content of now in contrast to this what we see here is that this is a that the value itself is available there 0 is available there I am not using a register to specify that value. So, that is why we call it as an immediate addressing mode and the next one specifies the register to which I need to store. So, that is called a register addressing mode.

So, what we have we have learn the move instruction we have learned 2 addressing modes already the immediate and the register addressing mode the next is increment EBX. So, if we want the increment by one I need not say add EBX comma one is equivalent to inc EBX right I could have written add EBX comma one here right I could have written add EBX comma sorry 0 x one instead of increment EBX. So, if you go into volume 2 of Intel you will find that this will consume more bytes than this; this will consume tremendously more bytes than increment and that is why the compiler would prefer whenever I want to increment a register by one or memory location by one I will prefer in inc instruction rather than an add instruction are you able to get this because the add instruction will consume more bytes while an inc construction will consume very less number of bytes and a add instruction will take more time to execute inc can take very less time to execute. So, this is a very simple example.

Now, let us go to the next one any doubts in this.

Student: No.

No, yeah Kavya?

Student: (Refer Time: 18:37) hexadecimal OS.

OS is because I wanted to teach hexadecimal here. So, if you just put 0 then it becomes decimal right and you can also put binary if you put 0 b here right I do not know I am not clear about tri 0 b and put binary, but if you just do not put anything it is treated as decimal if you put 0 x it is treated as hexadecimal and I hope all of you know hexadecimal representation of data right; ya, doubt?

Student: We can move any constant like this or by last time only 0 and 1 we could assign directly.

No any constant you can put.

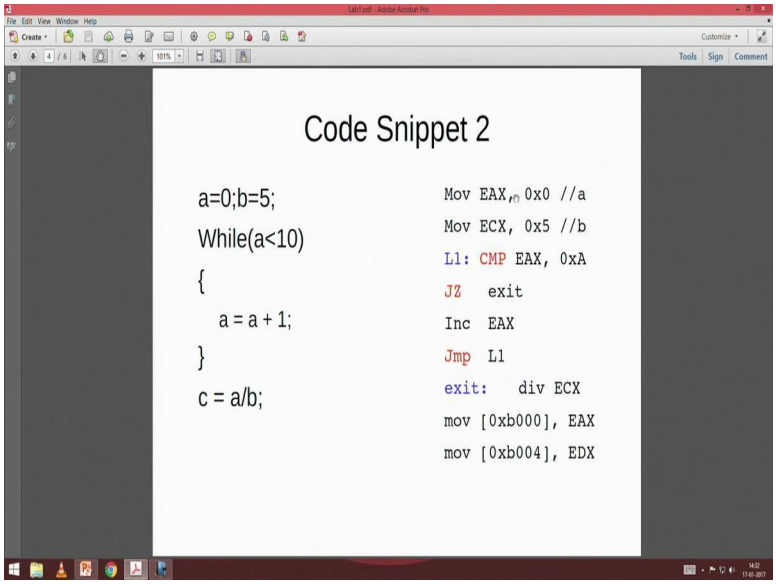
Student: Sir we can.

Any thirty 2 bit constant you can put any 30 bit constant you can move into this EBX.

Student: No, no like directly like this.

Directly move EBX comma whatever what do you like sub number you can put no problem yes. So, any constant can go here normally you do initialization with constants, but you can initialize with arbitrary constant also right can you tell me one example where initialization should be done with the arbitrary constant. So, random number right the c defer random suppose I am using S rand, right the CD for random should be a arbitrary value.

(Refer Slide Time: 20:14)



Let us take this a equal to immediately I am answering question a equal to 0 b equal to five while a is less than 10 a equal to a plus one x equal to a by b so much. So, now, so lot of things that we want to give through this course method first and foremost let us say a is assign to AX and b is assign to ECX. So, we just say move EAX comma 0 move ECX comma five that this. So, a equal to 0 b equal to five is mapped on to move EAX comma 0 and ECX comma 5 correct.

Then I am compiling while a is less than 10. So, the condition needs to be checked at the beginning at the loop. So, so this is a loop starts as the moment I see while I put one label here because I need to come back to that again and I compare EAX with 0 x a 0 x a means 10 decimal 10 the moment I compare EAX with a right then please note that we had something called a e flag registered right. So, the moment I compare the result of that comparison will be recorded in the flag register. So, in the flag register you will have something called 0 flag less than flag greater than flag, right.

So, if I am comparing left with right if left is less than right then j less than flag will be set less than flag will be set their if left is equal to right than equal to flag will be set if left is greater than right then the greater than flag would be set right, so if flag may have different conversation, but let us understand it like this, right. So, how do you compare 2 numbers subtract it. So, when will 2 numbers be equal by subtraction the answer is 0

why? So, when the when the left hand side is greater than right hand side greater than

flag would be said when the left hand side is equal to the right hand side 0 flag will be set because the answer is 0. So, that is something called 0 flag if an arithmetic computation gives me an answer immediately that 0 flag will be set right and if the left hand side is greater than the right hand slide some greater than flag would be set.

But for our understanding today if the left hand side is equal to right hand side then the 0 flag would be set. So, what I do is the moment the moment I say compare EAX comma 0 x a at the end of this instruction one of these flag should be set now j is z exit please understand what is j z means jump on 0. So, this jz is called a conditional jump in contrast to the previous one which is an unconditional jump you are not checking any condition you keep on jumping till one the previous thing that you saw jmp is an unconditional jump, but what you see here is a conditional jump and if what are you checking I am checking 0 why I am checking 0 because if my previous arithmetic operation did not give me a 0 result what is a previous arithmetic operation here compare if may compare gives me a 0 result then I jump to exit is a label here.

Otherwise I go I increment EAX because a is mapped down t o ex and then I go back do L 1. So, this is an unconditional jump jz is a conditional jump while jmp is a unconditional jump got it. So, till yes any doubts.

Student: Sir jz; jz could have been below increment EAX.

Student: (Refer Time: 24:40) would not have check this EAX is 0 or not.

No, no, no, if I put increment EAX; if I say increment EAX if I put j z after increment EAX.

Student: jz is jump x 0 what is we saying to said (Refer Time: 24:57) 0 or not.

No, no, no, we have previous operation did it give your 0 answer that what I am that the question that we are trying to. So, if I put jz is exit after increment EAX then this loop will never terminate or it will terminate after 2 power 32 or 2 power 32 minus.

Student: (Refer Time: 25:19).

2 power 32 iterations, correct, it will it will tell me after 32; 2 power 32 iteration.

Student: Sir, here we come on the while loop if EAX becomes equal to 10.

Student: But we have checking for it should be less than 10 at initially suppose a was 12 we initialize it then it was not where infinite loop.

Which one? Yeah, yeah, it will go into infinite loop.

Student: Because then as per the c called it should (Refer Time: 25:47).

Yeah. So, I should put j l, but this program will correctly execute, but this is not.

Student: Yeah.

This is not a faithful representation of this good. So, could you find some enhancement to this program I want you to start thinking.

Student: (Refer Time: 26:08).

Yeah, come in, did you understand this when I say j z I will; I as a processor will go and see the 0 flag, if the 0 flag is set to 1, I will go and do that jump otherwise I will not do the jump right now when will the 0 flag set be set if the answer of your previous arithmetic computation previous arithmetic computation if it is 0 then the 0 flag will become one in this case when I compare EAX with 0 x a correct when EAX reaches a value 10 this compare operation will make the 0 flag one because the way you compare is if I want to compare a and b I do a minus b if that a minus b is less than 0 than I say a is less than b if a minus b is equal to 0 then I say a equal to b if a minus b is greater than 0 then I say a is greater than b.

So, this compare EAX 0 x a essentially subtract the content of EAX with 10 are you following till now right now when EAX itself is within EAX if we have 10 ten minus 10 essentially become 0 then this j z flag if the 0 flag will be set to one by whom this compare operation then when this jz is executing it will go and see whether the 0 flag is set to one it find yes it is set then it will go and go to exit it will not execute these 2 instructions.

So, first I start with EAX as 0 I compared EAX with 0 with 10 it is not if I compared with 10 will the 0 flag will be set after this yes or no.

Student: No.

No. So, I will j z is take this branch will it go to exit yes or no.

Student: No.

No now I will increment EAX; EAX becomes one now I do an unconditional jump back to L one I go here again now I will compare one with 10 will the 0 flag be set.

Student: No.

No will j z execute no. So, the meaning jz will not take the branch. So, again I will come EAX will become too again I go here I compared 2 with 10 then I compare 3 with 10, 4 with 10, 5 with 10, 6, 7, 8, 9, 10, compare 10 with 10 will the 0 flag now be set, yes, now this when I jz, now it will take the branch it will come here. So, this loop which is comprises increment EAX that is the only thing inside this right this will be executed till EAX becomes 10 the moment EAX becomes 10 then this particular instruction will not be executed; that means, this is not equivalent this will not get executed and essential you will come back correct are you able to follow.

Student: (Refer Time: 29:56).

Ok.

Student: If you get a jz exit and then say you get jl exit like a. So, it is it equivalent to checking for greater than or equal to.

Jz exit.

Student: Then jump if more exit can you can you do that.

Jl exit you are you are actually jle axit.

Student: Yeah I know, but (Refer Time: 30:18).

You can give yeah.

Student: Just getting multiple; multiple; multiple; lines of.

Yeah, yeah, you can do, but what he says is j z exit will not go and change the 0 flag or anything, it is only a jump instruction only arithmetic instruction will touch those flags.

Student: (Refer Time: 30:38).

Yeah.

Student: (Refer Time: 30:40). So, we said.

Compare now there is nothing called compare flag thus 0 flag negative flag positive flag.

Student: (Refer Time: 30:46). So, when we.

When you do that j z exit the 0 flag will never; will not be touched, the jump instruction will not go and manipulate the flags. So, what we have done here please understand is that I am now establishing a communication between the compare instruction and the conditional jump instruction right there is a communication that is happening through this 0 flag right between the compare construction and the jump instruction. So, there is a dependency of the j z instruction on the compare construction right rather than looking at those 2 other. So, I am talking about this why I am talking about dependency will take later, but please understand if you take the first 2 instructions if I swap nothing will happen right there is no dependency between that move and the next move right, move ax and this, but there is a great dependency between this compare and j z are you able to follow this, right.