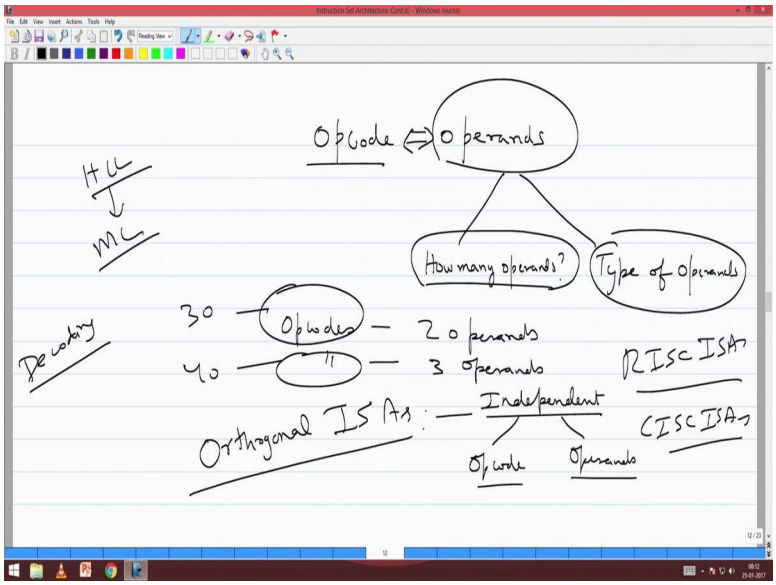
Computer Organization and Architecture
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture -12
Orthogonal ISA, C Constructs Mapping, Addressing Modes

And equivalently this was done by Intel and equivalently there was that 3D now by MD.

Now we will continue on this discussion on the evolution of instruction set architectures.

(Refer Slide Time: 00:33)



So, next step would be as we know that an instruction basically has an opcode and operands, on the operand side there are many decisions that need to be taken for example, how many operands right. So, what is the rational behind having these many amount of operands. So, this one very important decision that we need to take and the next is type of operands and another important thing is there a relation

between opcode and operands. So, can I have say some instruction there are some in opcodes for which there are 2 operands, some opcodes for which there are 3 operands and so on. If that is the case and then this these sets are quite large like there are some say 30 different unrelated opcodes for which there will be 2 operands, and some 40 opcodes for which there are 3 operands.

Then what would happen here is as I told you earlier there are 2 levels of translation, first is you are high level language gets compiled into a machine language then this machine language is interpreted by your hardware. So, every instruction has to be understood by your hardware, and that is what we call as decoding and instruction right. So, there are ones and zeros that are coming into your hardware and the hardware should understand yes I have to do I had on this subtract on this.

So, that is there is an implicit understanding there is happening within the hardware and when that is happening the hardware should essentially realise what is the opcode and what are the operands right that is also a part of you know your understanding procedure understanding process. Now if I can go and interpret my opcode and the operands concurrently, then my understanding process becomes faster right if I can go and interpret the operands independent of the opcode right any let any let there be any opcode, my decoding or understanding of what the operands are independent of is independent of the opcode if suppose such a scenario exist then your understanding process becomes easy.

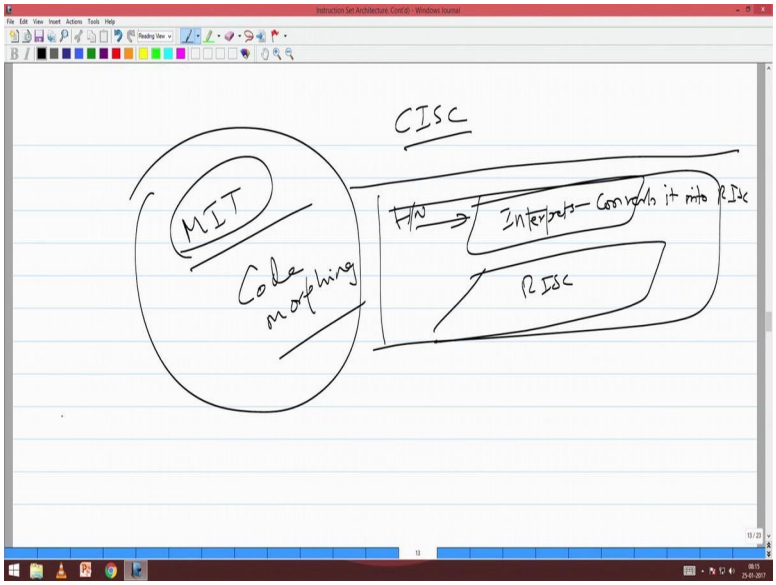
I will be understanding what do you mean by understanding we will when we see the actual hardware we will understand what you mean by understanding why what do you what do you mean by hardware understanding the you know instructions, but I can go and understand what the operands concurrently with while I am trying to interpret what the opcode is. So, instruction set architectures where the opcode is independent of the operands and vice versa are called orthogonal ISA's. Now we will try and be as orthogonal as possible, it is not possible to be completely orthogonal and we will try and be as orthogonal as possible. So, if you look at the hack ISA that we did last semester it is almost orthogonal right. Now what is one necessary condition for an orthogonal ISA, what is one condition that will make orthogonal ISA possible?

Student: Fixed length.

Fixed length. So, I need to have fixed length if I do not have fixed length automatically it implies that depending on some opcode some operand will be somewhere etcetera. So, mostly the RISC ISA's or more orthogonal than the CISC ISA's; because RISC ISA's or

more or fixed length when compared to CISC ISA's. So, there are 3 points that we want to cover today an instruction comprises opcode and operands, now we have to discuss more on the operands before that for the implicit translation process interpretation process there is a there are opcode there are operands and there should be an deciphering of the opcode and operands by the hardware, and that is going to be fast if you are going to have orthogonal orthogonality established between them and I will try to be as orthogonal as possible I cannot have that 100 percent orthogonality, and RISC ISA's are much more orthogonal then the CISC ISA's right that is why RISC process do work faster is not that CISC does not work.

(Refer Slide Time: 06:09)



But actually what happens is today if I have a CISC processor, the way the CISC processor is constructed this is what we infer nobody tells you how it is done is that all this CISC instructions there is a layer which interprets it, and converts it in to RISC. Every CISC instruction gets converted into a underlying RISC instruction and then below this is a RISC processor which executes.

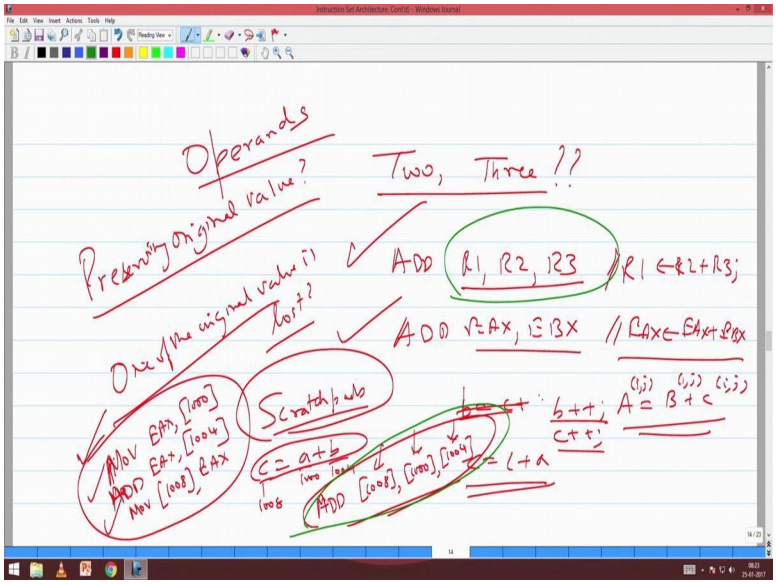
So, from the operating system or from the outside view you only see a SISC processor, but the CISC instructions are not directly executed on a CISC processor by say CISC

hardware, the underlying hardware is a CISC hardware and all your RISC instruction

CISC instructions are interpreted or converted into a set one or more RISC instructions and it is executed on CISC processor.

This is how. So, building CISC processors are becoming difficult and this is how the complexities basically taken care of. So, there is one hardware layer intermediately which will interpret this CISC and convert to RISC. So, this particular concept is called code morphing. So, there is been an extensive research in MIT. So, just google this on code morphing is a very old concept, but it is a very interesting concept.

(Refer Slide Time: 07:51)



Now we will go on the next discussion would be on the operands. So, how many operands should I have 2, should I have 3 right these are all questions right and. So, why should I have 3 why should I have two? So, what are the decisions? So, can you give me some reasons for why should I have 3, 3 operands we like say add R 1 R 2 R 3. So, I these are 3 operands instruction R 1 is equal to R 2 plus R 3, but I could I have again add EAX comma EBX, where EAX is EAX plus EBX; why should I have this why should I have this what are the pros and cons of having a 3 operand versus a 2 operands.

Student: If you want to preserve the original value then you do not want to put it back to the result value.

So one thing is preserving original values right, this allows you to preserve original values while this your origin one of the original value is lost, but normally registers are basically used as scratch pads they only store temporary results. So, there is no reason there is. So, for example, normally I will have things like b equal to c plus or I will have b plus plus where I have to update that value or c plus plus, c is equal to c plus a right.

So, normally when you look at many of the, you know programming when you see the actual you know code you will see that a variable modifies itself as a part of your assignment statement right the normal thing that you see. So, again if you ask our analytical proof I cannot give, but this is how code is written very rarely you will say for example, somewhere like matrix multiplication you will say a equal to b plus c a i j is very rarely you will see such type of constructs. So, many point of time we will have a variable that is getting incremented or get that is getting added to some other variable. So, that is why we are happy with this scratch pads.

Where we are happy with you know 2 registers here see the point is when I even I say c equal to a plus b both a and b are in memory both a and b are in memory. So, I have to fetch it from memory right I have to go and fetch it from memory, and then add it. So, my easiest way is let a b stored at 1000, b b stored at 1004 I will go and bring into the thing is I will if you let if you use this I will just say move in to EAX 1000, and then I can say add a x with 2 1004 and say let this be 1008, so I will move 2 1008 a x this is the easiest code I can write here right. So, even if you have instructions like this these instructions are basically stored in memory and I can go and access from there and store right are you able to get this, but you could ask me right I could can I have something like even add 1008 1000 1004 this is simplest instruction, but I will have 3 memory addresses here. So, if I have a 3 address 3 operand instruction can I do something like this first and foremost yes. If I have a operand instruction and each operand can be a memory then I can do the, but is it practically possible or not what are what is the; what are the difficulties in this.

Student: Writing in to the memory.

I will write in to the memory read 1004 and 1000 from memory, and then write it write

the answer add it and write back the answer in to 1000 8 what will be the practical difficulty here.

Student: Temporary register we need temporary some scratch by the store memory.

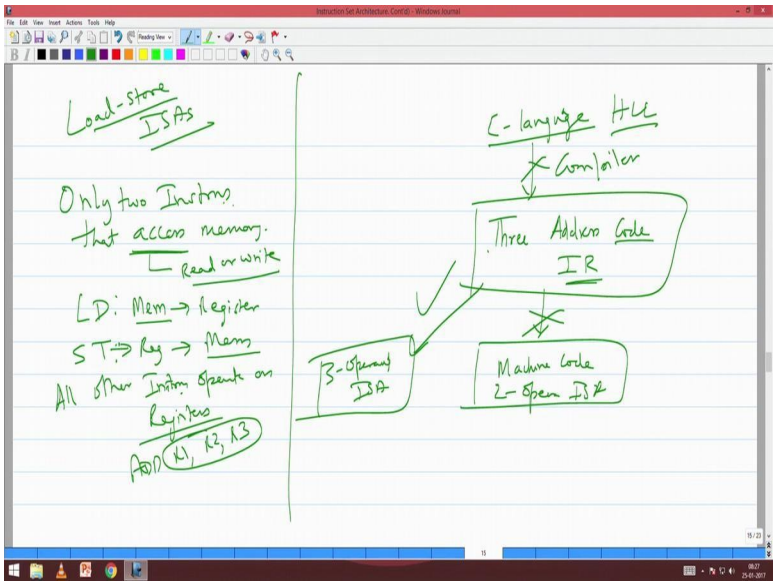
No I can directly see if you use the Intel architecture you can directly access from memory as a part of your instruction, I could have something like this for sure right these 3 are valid instructions in Intel move a x add a x and this. So, what stops you from having such type of such type of things we may need more or registers, but more imp practically what is the difficulties not only takes time. Now I am taking 3 instruction time here it is impossible to implement this because I for me to access memory I have a memory management unit and it can at a point of time take only one memory request from one CPU right that is the that is the practical difficulty, I could not have 2 memory request then the whole memory management unit construction of that memory management unit becomes extremely complex are you able to get this right. So, that is very very important.

So, that is a there is a difficulty in even creating this type of a memory management where I can go and read 2 simultaneously I can read I can give 2 address read 2 address and also write one right. So, and this should be viewed in the context of pipe lining which we will be talking next. So, there is a practical difficulty in having 2 memory operands in the same instruction leave alone 3 even 2. So, if you carefully look at the entire Intel ISA that 1000 or pages 1000 plus pages, you will never find an instruction that has explicit 2 memory accesses.

Right it will not have 2 explicit 2 memory accesses there are some set of instructions we will see where both the destination and source are memory, but there are very very specific instructions right, but you will not find 2 fellows which are explicit memory access. So, that is something that we need to basically keep in mind. So, there are certain practical limitations which stop you from having a now we are not even talking about 3 operands, we are basically saying that 2 operands itself I cannot have both the operands as memory right now, but why do. So, some of the RISC architectures do have 3 address right. So, what is the difficulty in having 3 other difficulty, in having 3 addresses and

what are something say. So, what are some characteristics of RISC ISA's which enable you to have these 3 addresses one thing is.

(Refer Slide Time: 15:56)



So, let us take some dig in to compilers has basically you take a C-language construct now this basically gets converter into what we call as 3 address code or what we call as 3 address intermediate representation yeah right. So, what is, so basically then this 3 address code is basically converted in to a machine, machine code right.

In our in the hack jack to hack conversion your jack got converted in to a one address code correct because it was push pop on to the stack, but push in to something pop into something push from something right. So, there was one address in many cases so, but in general your C-language or any high level language the compiler will convert it in to a 3 address code; that means, an instruction with 3 operands right. So, now, if I have a 3 address 3 operand ISA this translation actually becomes simpler. So, if I have a 3 operand ISA was that say this is a 2 operand ISA; certainly this is much easier than this. When you do the compiler course you will also understand that generating a 3 address code is much simpler there right. So, that is one motivation for lot of you know lot of ISA is to have 3 address operands, but when I look at 3 address operands those type of

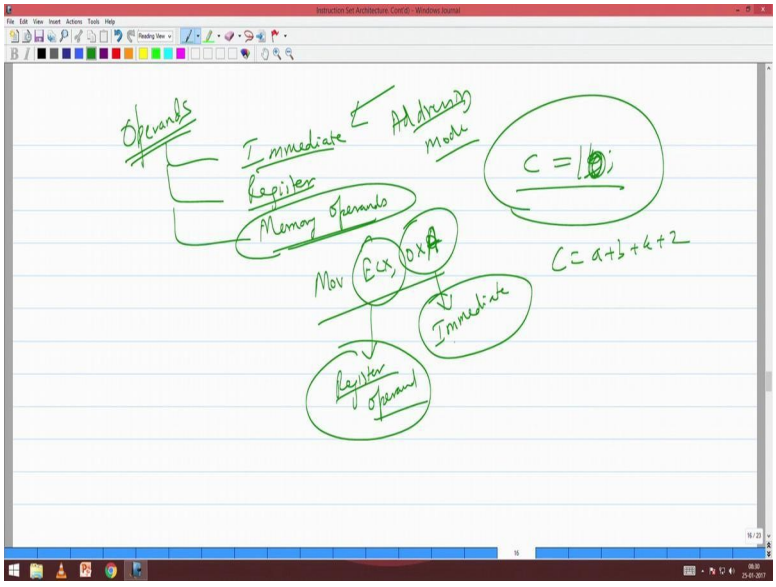
ISA’s fall in to one category called load store ISA’s what do you mean by a load store

ISA.

There are only 2 instructions that touch memory that access memory. So, in the entire computer science analyse of computer science accesses essentially means read or write that is one single word for read or write only 2 instructions in the RISC ISA in the in the load store ISA access memory one is called load another is called store. Load will move from memory to a register and store will move from register to memory all other instructions operate on registers like add R 1, R 2, R 3 these are all registers. So, I will not add a memory operand inside the add operation.

So, I will not have memory operands any instruction except load and store. So, many of the ISA's today the ISA inside you are favourite arm processor, which is in your mobile phone are all 3 address ISA's there are RISC ISA's they are all of fixed length and they are all load store ISA's because then the entire process becomes much more simplified. So, if you also carefully look at the hack ISA it was also a load store ISA, there was a load and store instruction there is a instruction. So, if you go carefully and look back into a hack ISA which you did last semester it is also a low storage ok.

(Refer Slide Time: 20:07)



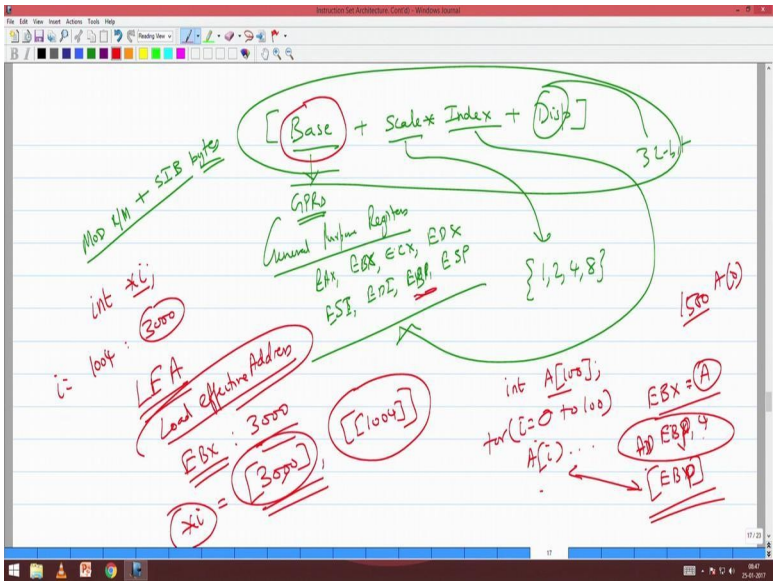
Now, next, so what are the type of operands that we could have? We have studied we

have been discussing on the number of operands, now what are the type of operands that we could. We could have immediate operands, we could have register operands and we could have memory operands. Immediate operand means in the instruction itself the value of the operand would be therefore, for example, suppose I have something like c equal to 0 this can translate to move ECX comma 0 x 0 this is a hexadecimal representation, or c equal to 5 c equal to 10 then I can say move ECX comma 0 x A this is a hexadecimal value now this is basically an immediate operand and you will see lot of places where you are doing this initialisation.

So, this is an immediate operand while this is called a register operand or we call it as a register addressing mode. So, these 3 are also called as addressing modes, the mode by which I address the operands this is a register operand or register addressing mode this is an immediate the next one. So, there is. So, they have also seen some implicit the addressing modes like your dev that we talked last time right.

There are some registers that are hidden that on explicitly stated in the instruction, but they get modified when you do a dev instruction. So, we will talk about explicit. So, we have already talked about implicit operands. So, we are now talking about explicit operands here. So, here ECX is a register operand and this is an immediate. So, whenever I see a variable and that variable is temporarily used right. So, or I am, so I will start using register operations whenever I see a long expression a plus b plus k plus 2 then I will start using register operands here, whenever I see a constant I will use an immediate operand. So, these are all very simple things that we see the more complex things come in memory. So, Intel has come out with excellent thought processing in terms of memory let us go and see the different memory operand.

(Refer Slide Time: 22:40)



So, in Intel could have base plus care into index plus displacement this can go up to 32 bit displacement. Base you can have all the general purpose registers we call it as GPS general purpose registers. I could have stack I could have EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP all these things I could have then the scale is 1, 2, 4, 8 index again can be any of these d s p and displacement can be any 32 bit displacement, and I could have a combinations of this right. I could have just base alone I can neglect everything else I could have base alone I could have displacement alone I could have scale index alone I could have just index, I could have base plus scale into index, I could have scale in to index plus displacement I could have base plus displacement on all possible combinations I could have right.

And if we actually go in to if the Intel manual you will see that just to handle this complexity there is something called mod R slash M plus scale index by sib bytes; and it is a big matrix that you see in 2 pages very big matrix. So, the way this is translated in to machine language, that is your base this addressing mode is translated into machine language that should be a representation right. So, when the machine sees it should know that this is a base this I should know it is address displacement it should know this is base plus scale into index it should all these combination the hardware should

understand. So, you should translate it in a way that the hardware understands.

So, for break making the hardware understand what sort of memory addressing you are trying to do there is something called mod r m plus sib bytes 2 bytes there mod r m byte and sib bytes. So, if you look at the Intel manual 2 manual 2 first or manual one last, you will see a very big page with this mod r m and sib bytes. So, there will be something close to 512 entries something like that before going in to why we need this I should also tell you that Intel provides this 2 512 bytes right of extensively co extremely complex byte you know representation, it is the compiler actually sets it out to create that representation then the hardware again sets it out consumes lot of your battery power to find out what you are representing.

So, all this are put in to your hardware please note that this interpretation of your in instruction is it is done by your hardware. So, there is an hardware which is going to interpret your mod r m and sib byte right so, but when we use traditional compilers right some of the compilers which say I have a 32 bit version of compiler all these fellow shouts if you use those compilers if you go and disassemble the code and see out of say all these 512 possible combination, 10 combinations are only used there are some combinations which are never used, you take all the codes disassemble them and see in a particular domain like you can take all your you know network domain or your web transaction domain or your you know financial transaction domain or no normal computing normal c code engineering domain, very rarely you will see all these addressing modes completely used.

You will see only 10 percent or 20 percent of this 256 rarely 10 or 15 of your 25 512 bytes getting used there. So, all the remaining thing or inside your you know hardware they are part of your decoding right decoding, they consume power they heat it up for some fan to cool it further and again the fan will get heated up and this will cooling. So, some adiabatic thermodynamics cycle is happening there is that nothing is getting used.

So, the point here is that when a when we when that is why many of these software comp many of these hardware companies processor companies do have their own software division they will have their own compiler division, and the work of that compiler division is to see how to use this hardware. So, I always say hardware is like building a dam you have to fill it with water otherwise there is not use for that dam. Who will fill it

with water? A good compiler will fill it water you understanding this right i.

So, when I give an interface there should be a layer which will try and use that interface to the best possible thing. So, that is why many of these companies today if we go and look at Intel AMD IBM all there are very very strong software division which will write compilers and other things to basically see that their hardware is utilized very well. So, that is another very important thing that we need to keep in mind. Now coming up why at all these so many complexities that we have look at right, now all this who is driving all these things there should be some programming language construct that is going to drive this right. Now for example, if I have int star i this is a pointer right correct how do I manage this i store. So, i will be say stored in some 1004, i will be stored in 1004 which has some 3000 value right now when I say star i it is the content of 3000 that I am trying to access.

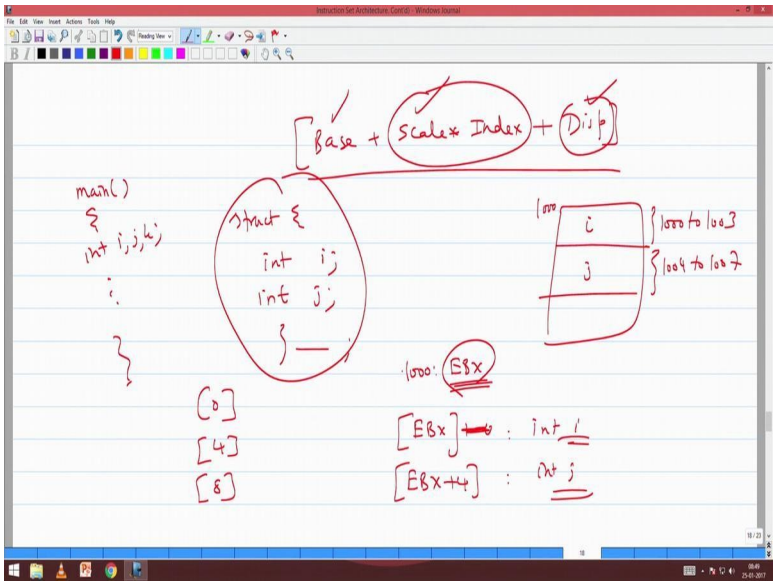
So, this value 3000 will be stored somewhere in 1004 and I am going to access this 3000. So, there is one indirection that is happening here right. So, this i will not store the address, but we will not store the value, but it will store the address where it is stored. Now, how do it go and manage? You manage it with basically these you know the in the instructions the register addressing modes right. So, there are something called LEA what you call as load effective address. So, there are certain instructions I just leave it to you to go and find out what this LEA. Now these are instructions and then that LEA will basically you know try and load in to your register again. So, we need something like 3 1000 in registers. So, some register EBX has 3000, I need this 3000 there then I can go access this 3000. So, star i would be the content of 3000 which is nothing, but the content of content of 1004 correct are you able to follow this right star a is the content of i right star a is the content of 3000 which is this.

So, there are instructions basically used to handle these type of activities. So, LEA is one instruction which can be used, but nevertheless suppose I have. So, let us just use where we are using one register for addressing. So, suppose I have int I int A of 100 right and then I say I equal to for i equal to 1 to 100 I just do some processing on A. So, what I can do is first EBX will be storing A; let us say A starts at 1500 A of 0 is stored at 1500 this is.

So, EBX will have A right now and what I do every time when I start doing here I will increment EBX by 4 and I will access EBX, are you getting this? I will increment EBX by four and I will start accessing EBX. So, what is as my loop goes on from 1 to 100, A 0 would be one 1500 when I come to A 1 sorry when I come to A 1 then what happens by the time I increment EBX by 4. So, it is 1504 and the content of 1504 this. So, I can basically have you know EBX one of the. So, EBX is basically we will use EBP also because it is a base pointer sometimes you can use a b p right. So, I can keep incrementing that register I can use any of these registers, but I can keep incrementing the registers to keep accessing the next 10 next this elements.

So, this is one use of you know register addressing mode register based register direct memory addressing mode right. If I just use this register this is register direct right, but then we need to go and talk about the other parts like right.

(Refer Slide Time: 33:09)



So, I said base plus scale in to index plus displacement. So, how do you do this, when we will I use base plus displacement suppose I have a struct. So, I have int i int j and something. So, the structure will be stored structure will start at 1000, I will be stored from 1000 to 1003 while j will stored from 1004 to 1007. Now where will. So, I can use

say 1000 as EBX then I can say EBX plus when I compile. So, I assume that the

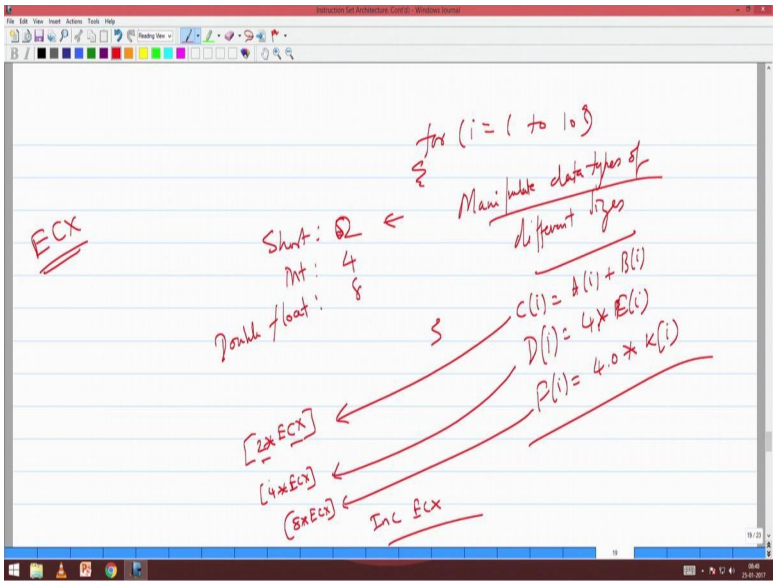
structure base of the structure will be load at EBX.

So, I can now say EBX plus 0 just EBX would be i and EBX plus 4 will be j. So, this gives us base plus displacement. So, other thing is if I do not have something like struct this is quite easy. So, I have main and in which I have int i j k as I told you and yesterday's lab class the compiler assume it starts from 0 whenever it wants to access i it will say 0, j it will say 4, k it will say 8 and so on. So, just displacement also we have seen just displacement just base, base plus displacement now let us talk about scale index etcetera, where do I use base plus scale into index plus displacement can you give some interesting things.

Student: Array of structure or structure a structure in which there is an array both of this are.

Now more than that why do I want this scale and scale into index? So, let us let me go into some very nice thing right.

(Refer Slide Time: 35:31)



So, suppose I have a loop, this loop is trying to manipulate data types of different sizes this normally you will see right it will manipulate say a short integers which is of say 2

bytes, it will manipulate an int which is of say byte 4, 4 bytes in size it will it will go and manipulate you have double precision floating point which is 8 bytes, all this can happened here right. So, there could be some C i is equal to A i plus B i there could be some a b c d D i is equal to 4 into a b c d E i then I could have F i is equal to 4.0 into K i all these things can be part of your right then what could happen is that this i is a variable I would like to keep it in ECX c actually stands for a counter.

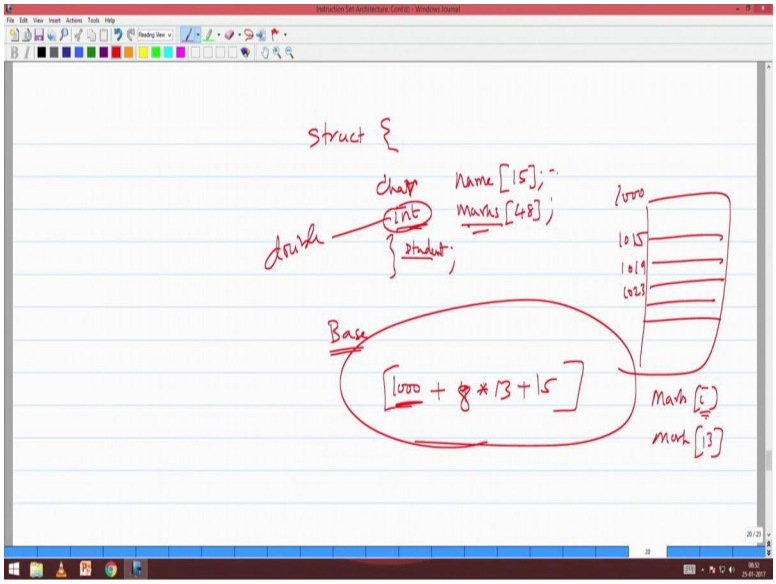
So, normally when the Intel when the compiler translates, the c always goes for a counter there is a there also a stands for an accumulator or an. So, it is. So, there were in initial architectures which are only the a registers just an accumulator based architecture right, c stands for counter right. So, like that you know S i and D i they stand for index registers we will see some of them in the next s p stands for stack pointer b p stands for base pointer etcetera. So, normally ECX will go and like what we did in one of the earlier thing let me try and go there I do not want to keep modifying that e b p right here what did I do for I just asked the I got incremented I kept adding 4 to this, but I do not want to do it because I want I the ECX to represent I if I keep modifying ECX as I increases if I keep modifying ECX by four adding ECX adding 4 to ECX then for this I should add 2 p ECX for this I need to add you know 8 p ECX right I have to mod you know modify ECX in 3 different waste handle these 3 different arrays which is going to be extremely complex for me I do not want to do it.

So, what I do is I when I am when I am doing accessing this I will do it as 2 into ECX; when I do this I will do it as 4 into ECX, when I do this I will do it as 8 in to ECX, and after doing one iteration I will say increment ECX, and I will make the base of no there are so many bases here. So, we will we will try there is lot of work that needs to done here, but this is how I will use this scale in to index right the scale is used because I would like to maintain one counter variable and if that loop is going to handle multiple arrays of different sizes then I need to basically go and modify this interface right. Now ECX will be I and your I will keep incrementing ECX as I proceed in your iteration, but then accessing individual entities here we will go with this scale.

So, that is an use of a scale. So, the notion of scale came into your; the notion of scale came into this you know addressing mode because we wanted multiple data type. So,

when an array acts when a loop acts on array of multiple data types then scale becomes very useful. Now can we tell what base plus scale into index plus displacement can you give me one example right.

(Refer Slide Time: 40:02)



So, I can take a struct in which I could have a string right. So, how do you do this or int clue some 48 courses right. So, I will put ok.

So, how do you, where you will where you can use this. So, there could be a base this struct would be say student record. So, there could be a base. So, let their this record is stored at 1000 the name starts or your marks actually start more than name marks would be interesting we start that say some 15, 1000, 15 and it is 4 4 bytes and so on. So, and if I want to access marks I right and this, this let us say. So, I say 1000 plus the displacement would be this 15 1000 is the base I am accessing some field which is at least 15 bytes away.

So, this will be 15 plus I say marks of I right. So, I is this index that will be this index. So, I want marks of 4. So, 4 and int, int is the size of this. So, four if I want marks of 8 marks of 8 let us say marks of some 13, your thirteenth course then essentially I going say

suppose say suppose I start giving you real real marks. So, double then this would

become 8 in to right. So, we can create one example. So, this is base plus scale into index plus you can construct several such example now this also explains why this particular combinations very rarely used ok.

So, what we have done today we have understood about you know the relationship between opcode and operands, what are the benefits of you know a risk architecture what is orthogonal ISA and then we went and discussed about different types of operands and we also showed some mapping between c constructs and the way we address these operands.

Thank you.