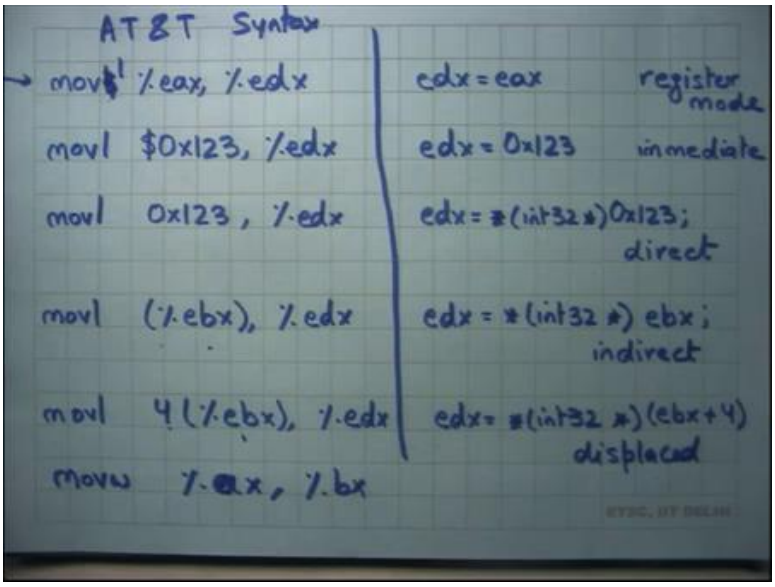
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 05
X86 Instruction Set, GCC Calling Conventions

Welcome to Operating Systems Lecture 5. Yesterday we were we started looking at the hardware interface for an operating system and we started with the x86 architecture. And, we were looking at the instructions set of the x86 architecture just to get ourselves familiar with all the instructions.

(Refer Slide Time: 00:25)



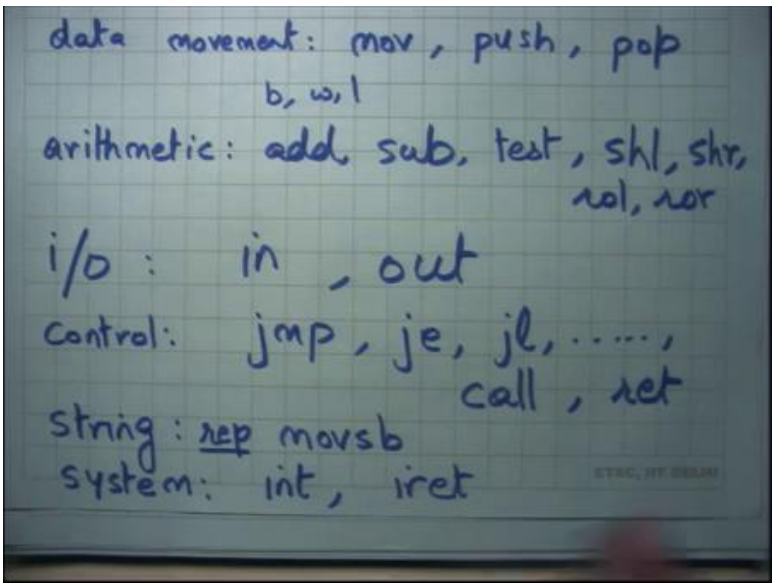
And you know we said that the so, we here the syntax of the x86 instructions on this side and this is the AT&T syntax where the destination operand is the last operand and the source operand is the first operand or you know if the three operands in the first and second operands are the source operands.

And, you know there are these different addressing modes which you may have already studied in your computer architecture class. So, you can move register to register. This is a register mode. So, every instruction can operate in either these modes. We know we could very well replace move with add or subtract; that means, the same thing you know it will it means a corresponding thing mov l dollar 0x123, this is the immediate mode. Basically, means that assign immediate value one two three to this register.

Move without a dollar basically means treat this address as a memory address and so, this is the direct memory addressing mode right; you may have seen it in your computer architecture class. Then you could say you know if you bracket the register name like this, then this basically means it is an indirect addressing mode basically means use the address in EBX to dereference memory and get the contents into the destination operand right.

So, in the C equivalent it basically means EDX = *EBX where EBX is treated as a 32-bit as a pointed to a 32-bit value. And, then there is also the displaced mode. So, you can specify a displacement along with the indirect mode. So, which basically means that add 4 to EBX and then dereference it and these displacements or immediate values or memory addresses can be 32-bit byte also in 32-bit mode on in x86 right. So, you could have a full 32-bit displacement with a 32-bit register value add them and then dereference it alright.

(Refer Slide Time: 02:37)



So, the different types of instructions on x86. There are let us say data movement instructions like move, push, pop right. We have already seen move push and pop are instructions that operate, specifically on the stack. I am going to discuss these very soon. Each of these opcodes can be suffixed with a character which indicates what the size of the operand.

For example, move l basically means the 32-bit operand and move b mean means an 8-bit operand and move w means a 16-bit operand. So, b, w, l is the I mean just syntax of AT8T to specify 8, 16 and 32 then there are arithmetic instructions like add subtract.

Student: Sir what are b, w, l can you explain?

What are b, w, l?

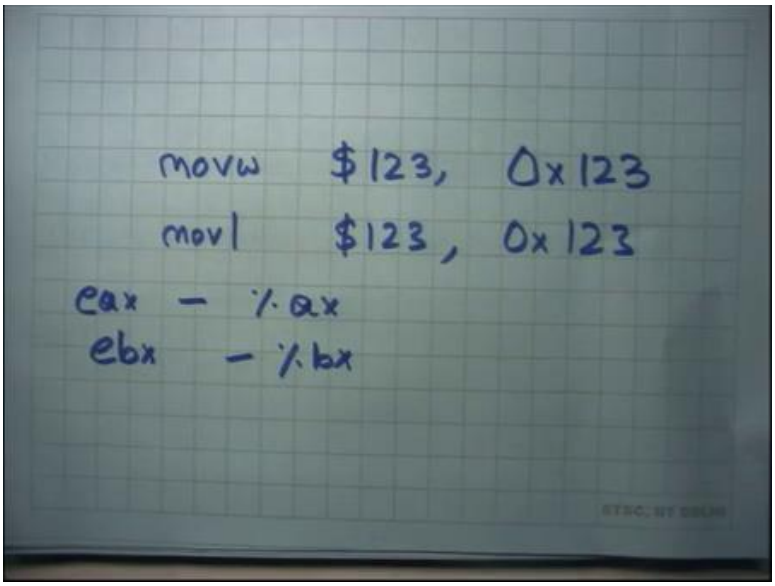
Student: b, w, l.

b, w, l are just suffixes to op codes which specify the size of the operand right. So, for example, in this here when I say mov I actually suffixing it with l, it basically means treat these values as 32-bit values right. If I add instead used b here, it would have mean meant I am I am working on 8-bit values right.

Actually, if I say b here this is an invalid instruction, this is an illegal instruction because you know I am specifying 32-bit registers and I am using b here. So, it is you know there are some instruction that are not possible. So, l is the only valid legal value here. But if I want to say for example, mov w I would want to say mov w ax bx right. Often the suffix is inferable from the operands.

So, for example, if I did not say w and I just said move ax bx the assembler would be able to figure out that it is a 16-bit operand. So, it you know it will it itself for example, suffix it with w, but for some cases it may not be possible to infer the operand size. For example, if I have something like let us say if I have something like mov $123 to memory address 123 right. So, it is basically saying move this immediate value to this memory address right and there is ambiguity here because we have not specified the operand size and it is not possible to infer the operand size from the operands right.

(Refer Slide Time: 05:11)



So, mov w will mean treat this as a 16-bit value and store only 2 bytes to this address and mov l would mean treat this as a 32-bit value and move it to. So, in such instructions you are required to specify a suffix otherwise assembler will say there is some ambiguity in your code right.

Student: So, how do we address the first 16-bits of a 32-bit register?

How do you address the first 16-bits of a 32-bit register? Well x86 gives you this ability to say you know; so, the first 16-bit is of EX and AX right and the EBX are BX and so on.

Student: What about the last 16-bit?

You cannot act as a top 16-bit is of a register directly not allowed alright. So, add subtract you know then and we also said that these instructions also set up the e flags register appropriately. So, example if it is a negative value etcetera. So, they also write the flags and there are instructions like jump equal, jump not equal, jump less which are which read the flags to make control flow decisions.

Then there also instructions like test which actually do not do anything they just set the flags. So, test is going to check if a value is equal to 0 or not or things like that or you know you can also have things like shift left or shift right or rotate left or rotate right and

so on you know. So, there is a huge list of operands that can happen just to give you a flavor of what.

Then there are IO instructions like in and out right. So, so far what we have seen are basically instructions that are do arithmetic or data movement that operate on either registers or memory, but what if I wanted to access a device? There are special instructions which are called IO instructions and the instructions are IN and OUT and what they mean and what their syntax is we are going to talk in a few minutes.

Then there are control flow instructions like jump you know jump equal, jump less than, jump greater than and so on. So, lot lots of different things and then there are special instructions for function calls. So, there is an instruction called call and there is an instruction called return which are basically equivalent to saying function call and return and what their semantics are we also going to discuss in a few minutes alright ok.

And then there are some instructions which are called string instructions right. So, string instructions are of this type mov sb where sb stands for move a byte of a string right and the default and this instruction has default operands the memory address of the. So, it is a string instruction which says move 1 byte of data from the memory pointed to by register ESI to the memory pointed to by EDI alright.

So, this is a special instruction that can actually do two memory accesses in one instruction. It will read a byte from the address of ESI and write a byte to the address of EDI right. And the reason this is supported in the x86 architecture is because there are many string operations that are usually done and you can actually prefix this instruction with what is called a repeat prefix that basically executes this instruction repeatedly each time incrementing ESI and EDI.

So, ESI is considered the source address or the source pointer, EDI is considered a destination pointer, mov sb means move a byte from the source pointed to the destination pointer prefixing it with repeat basically means keep repeating this. And, each time keep incrementing ESI and EDI till a certain condition has reached and you can also specify what under what condition you want to stop.

So, you can imagine you know the hardware designers what they are had in mind was let us say things like string copy right. So, if you want to implement string copy one way to

implement, it is to you know implement in using many instructions like a for loop which basically you know increments a counter in software. But because it is such a common operation you know the hardware designer said let us have a special instruction for it. And, there is just one instruction which will for example, do mem copy or string copy or things like that ok. Just to know because I mean just in case you see this kind of code in your programming assignments you should know what it means alright.

And then there are you know special instructions which we can call system instructions. These instructions are very, are instructions that are required by an operating system. So, so far, the instructions we are really talked about are instructions that user program may require accessing memory, accessing registers; accessing IO may or may not be allowed for a user for an application.

So, even in and out can be considered system construction in that sense, but then there are special instructions for system which you know allow you to for example, raise an interrupt right. What does means is it is going to it is called a software interrupt instruction which basically emulates that I have received an interrupt.

how that is used what it means etcetera we are going to look at in a moment, but just to give get give you a flavor of what exists and then there is another instruction called iret which says return from an interrupt right. So, you can say you can simulate an interrupt you can get into the interrupt handler and then you can have a return from interrupt instruction ok. Never mind, we will be going to look at it in more detail in a few minutes.

Student: Sir in and out instructions are like do they take input from keyboard and tend to console or?

So, what do they you know in and out instructions mean? Just hold on we are going to discuss this very soon right.

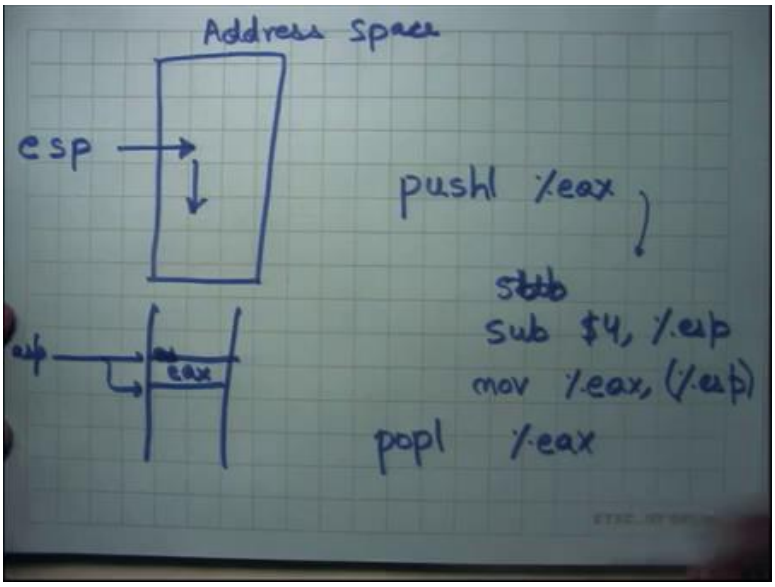
Student: What is test?

Test is an arithmetic instruction that just tests let us say your operands to see if they are equal to 0 or not or they at ands them and sees what. So, I mean it is an example of an instruction that does not necessarily modify it is operands, but set some flags based on the values of the operands. So, often you know if you want to execute control flow, you

can execute test and then you can execute jump conditional to or based on the value of the test alright ok.

So, we are going to discuss in a you know in the next couple of lectures how system calls work how, how operating system instructions work etcetera, but before that let us just talk about how function calls work right. So, how do you when you call a function what happens what kind of code is generated by the compiler and what kind of you know. So, how does it all work inside? It just in the context of function calls before we start understanding how things work for a system call and how does OS do thing different things etcetera alright.

(Refer Slide Time: 12:23)



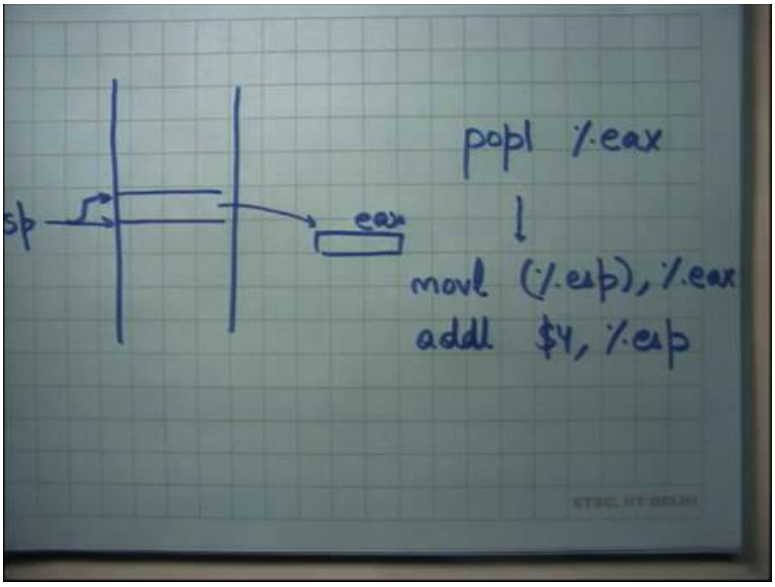
So, every process has an address space alright and in the address space, there is one pointer which is the stack pointer, or you know the 32-bit x86 architecture it calls with the esp extended stack pointer right. And this pointer basically the semantics of this pointer are that it should it points to the top of the stack and. And so, and the other convention on x86 is that that stack grows downwards right.

So, the stack is initialized at some value and then you know if you for example, call a function, then the stack is going to grow downwards; if you allocate a local variable the stack is going to grow downwards, assume we have seen the use of stack in other courses like programming languages or architecture for doing function calls right yes alright.

So, there are some instructions that actually access the stacks. So, for example, push I can push a register onto the stack, or I can push an immediate value, or I can even push a memory operand onto the register right. So, this can be replaced by a register immediate or memory whatever you like. Basically, means the semantics of this are subtract 4. So, in this case it is a push l because I am pushing a 32-bit value; 32-bit means 4 bytes. So, it basically means subtract 4 from ESP grow the stack downwards and mov EAX to the memory location pointed to by ESP right that is what push means.

So, here is an ESP you call push, it is going to decrement stack and it is going to put EAX in that. So, in other words what is going to happen is if this was ESP, let us say this was ESP and it is pointing here then when you call push what will happen is ESP is going to now point here and this memory location will now contain the contents of EAX right ok. Similarly, pop EAX basically has the opposite semantics.

(Refer Slide Time: 15:09)

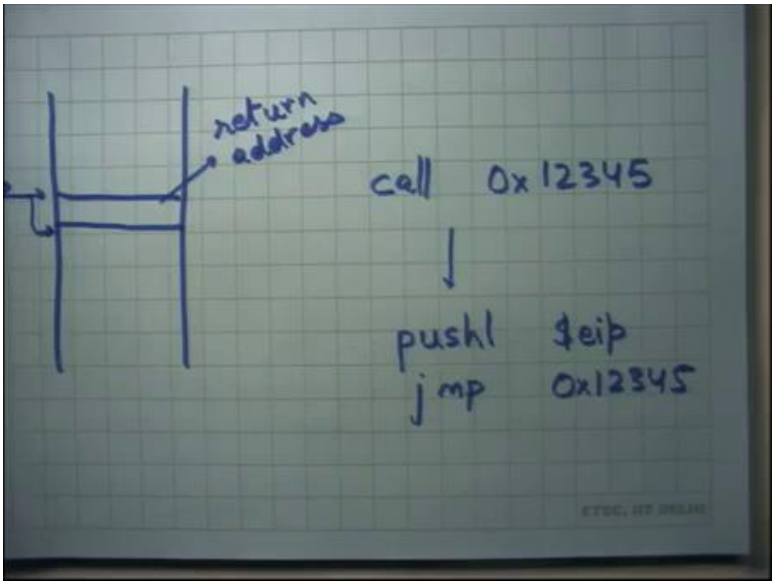


So, let us say this is ESP, I call pop EAX what it is going to do is it is going to move ESP contents of ESP into EAX and add 4 to ESP alright. So, in other words let us say ESP was pointing here, you executed pop what is going to happen is whatever was contained here is now going to go to the EAX register and stack pointer is now going to point here right. So, push grows the stack pop shrinks the stack alright.

Then there is function call right. So, we looked at we saw that there is an instruction called call. So, I can actually say call you know some address 1 2 3 4 5 which basically

simulates a function call. And so, in this case what happens is let us say here is the stack and before this call instruction ESP was pointing here.

(Refer Slide Time: 16:28)



So, call instruction is going to do what? It is going to push the current EIP right. Notice that no such instruction exists; I am just using it to explain it to you what the operations of this are right.

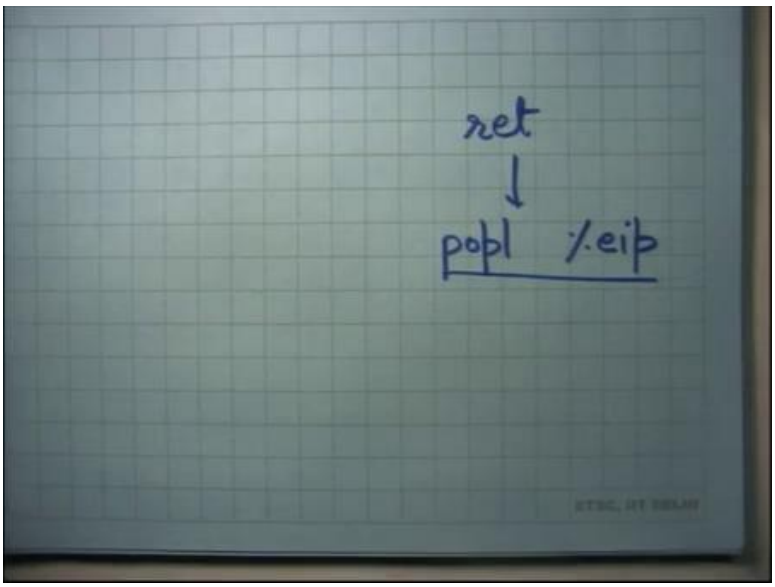
So, it is going to push the current EIP after it has been incremented. So, you know the as soon as instruction is fetched, the EIP gets incremented and the incremented EIP is what you push onto the stack right and then you jump to alright. So, basically you just push the address of the next instruction on to the stack and you jump to the destination right. So, in other words what happens is ESP gets decremented stack grows and this contains the address of the.

Student: Call.

Instruction following the call site; the instruction following the call site right; this is also called the return address right. You can imagine why you need the return address on stack because when the function calls return that is where he is going to jump back right. So, that is what it means right when you make a function call it is going to execute that and then when it calls return you are going to come back to the next instruction. So, that

is where the way it is done is basically using the stack and a return address stored on the stack alright. And then there is the return instruction.

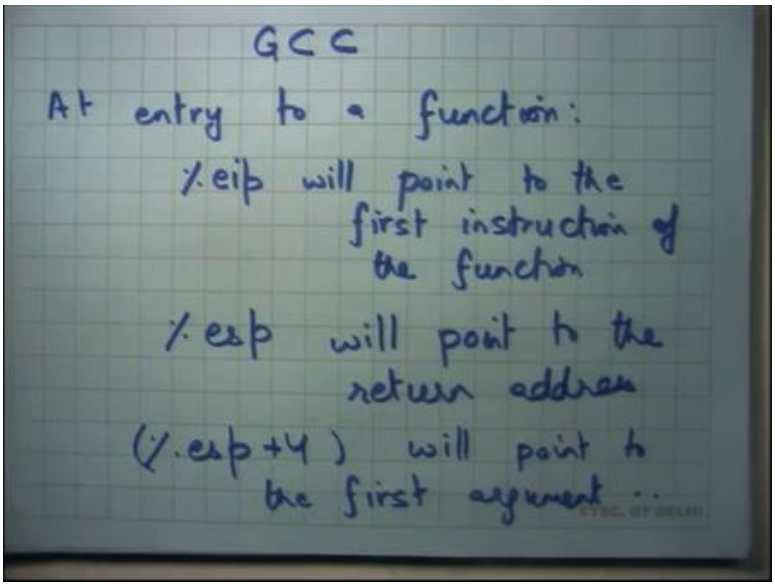
(Refer Slide Time: 18:19)



Basically, what it does is it just pops from the stack and puts it in EIP right. Once again this is not a real instruction. I am just, I am just using it to explain what return does right. So, it will increment the stack pointer which means it will shrink the stack and whatever the value was there it is going to put it in the program counter or the EIP and so, that is where you go next right. So, it is an indirect branch in some sense alright.

So, a compiler usually dictates how the stack is going to be used right. So, a compiler will have some conventions on how the stack will be used in case of a function call right. So, typically you would have different functions in different files and each function will be compiled separately and the compiler will have some conventions on how this function body should be organized such that it can fit in well with the caller and its callees right. So, let us look at what kind of what kind of conventions does a compiler have.

(Refer Slide Time: 19:28)



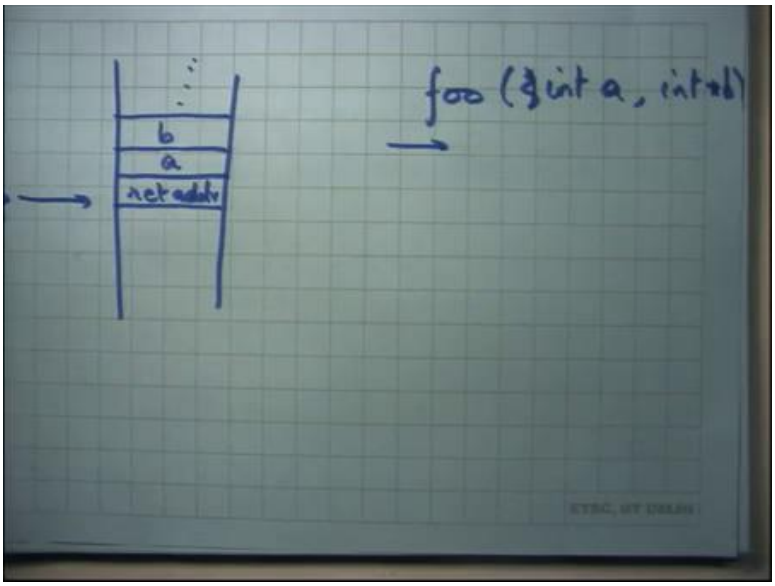
And let us look at one compiler which is which is a popular compiler that we all use is GCC alright. So, GCC says that at entry to a function EIP will point to the first instruction or the function right. Obviously, when I call say call some function name, the next EIP the next program counter should be the first instruction of that function right, ESP will point to the return address right. So, we saw when you say call the return address automatically gets pushed and so, if you execute call you know your GCC is basically just using the same thing as what the call instruction does right.

So, if you are inside a function call as soon as you enter the function call the program counter is the first instruction of the function and the stack pointer should be pointing to the return address right. So, if the first if the first instruction of the function is let us say return, then you just go back to the return address alright. And, ESP + 4 alright so, this is as you can see it is a very 32-bit architecture specific thing 4 bytes for the return address. Let us assuming that return address is 4 bytes and ESP + 4 should point to what.

Student: Unit is.

Yeah, the first argument of the function alright.

(Refer Slide Time: 21:42)



So, in other words if there is a function foo and I make a call to foo at this point; my stack layout must and let us say the foo takes you know two arguments int a int star b then if at entry to the function ESP should point somewhere in the address space wherever it points this should be the return address right and this should be a right. And, this should be b and so on right. If there are multiple arguments, they just stack on top of one another right.

So, that is the convention of a compiler. So, that is you know that it uses and that is what most compilers will use that a function when it starts. You can assume the following things about the stack layout, it can assume that the return address that ESP and the first argument is that ESP + 4 and all the flags and arguments are above it right. So, it is a responsibility of the caller to set up the stack in such a way before it calls the function.

Student: Sir.

Yes.

Student: Why is it sir when the function after the function call it will use them in the opposite way right? We can use a and b and then the return address. So, would it be would it be easy if we study opposite way that the return address is above?

Interesting question. So, why is return address? So, this is in some sense in the top of the stack right. So, why is the return address at the top of the stack and why are the arguments below it, why cannot it be the opposite way?

Student: Because push in the end.

Yeah so, firstly, the number of arguments is variable. So, you know it is hard for it is much easier to say that return address is right at the top because otherwise you have to worry about how many arguments there are and based on that you have to compute whether return address is going to live. So, as opposed to that because return address is the most common thing that is going to happen and so, let us have it at the top.

Secondly when you are calling the function you know that is when you want to push the return address. And so, you are going to push something and then you are going to call the function and that is when you want to push the push the return address because after that is the next instruction that needs to be executed. Otherwise you know I would have to worry about where I have, where I should return, and I have to push that explicitly ok. So, the instruction that pushes that call that actually makes the call and pushes the return address should be the last instruction in the caller.

Student: So, when we want to actually the function use a and b that we have to go up you can keep return address in your register using b, then bring it back.

So, if I have to use a b, then I have to actually go up is that right.

Student: As a just like a RAM.

It just it is just RAM right. So, it is I do not have to go up I just specify an address I just say 4 + ESP. So, it is it is actually no less efficient than say ESP right even I can say a 100 plus ESP. It is equal in efficiency; it is just an address computation alright

Student: Sir.

Yes

Student: Even x86, there is no function of link register that is l r which is in arm assembly language has l r which is link register.

Student: So, does it have.

So, so, the question is really about arm and x86 arm has a link register and x86 does not have a link register that is true why and why not let us just let us defer that discussion. But x86 does not have a special register where it stores the return address x86 in state stores the return address on stack right. Other architectures like arm have a special register where they will store the return address and now it is the responsibility of the software to actually move that a register to the stack in case of nested function calls right. So, that that overhead or that extra work from the software is avoided in x86.

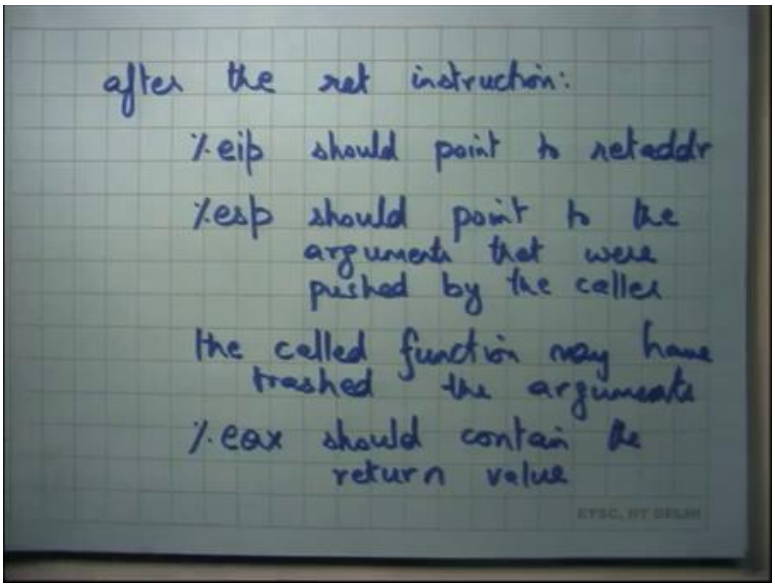
Student: Sir, we use branch and link and automatically copy that to.

Sure, if you use branch and link what it does is it sets the return address into l r that link register right. And now if you make another function call, nested function call then you before that you have to save the l r onto the stack.

Student: Yeah.

So, x86 avoids that in some sense alright and. So, that is the entry at the entry of a function. And after the return instruction after the return instruction or after the function return, the stacks should be organized as follows.

(Refer Slide Time: 26:22)



EIP should point to the return address alright, ESP should point to the arguments that were pushed by the caller alright.

Student: Should not it be the last argument that going to or first argument?

All the arguments right; so, add function return the stack should be exactly as the caller left it before calling the function right. So, I am basically specifying a contract between the caller and the callee right. There is a caller which is calling the function and the callee which is been called right and there is a contract between the caller and the callee and that is what I am specifying here.

And esp should point to the arguments that are pushed by the caller. The function may have trashed the called function may have trashed the arguments alright. So, for example, if I call foo int a int b int star b, foo is free to change the value of a alright. And, when it returns the caller should not assumed that the value of a is exactly what he had sent it as it had sent it. It should assume that the stack is laid out in the same way, but it should not make assumption on the value of the arguments right, just a convention of the compiler alright. The other thing is EAX should contain the return value right.

Student: Excuse me sir why b pushed before a?

Why is b pushed before a just a matter of convention ok? There is no reason one reason potentially could be that there are some functions that actually allow multi variable number of arguments. For example, printf allows you a variable number of arguments and the number of arguments depends on the value of the first argument which is the format string right. So, to be able to support this kind of thing you know the first argument should be right close to the ESP and all the other arguments can be a variable.

Student: Sir anyways the caller has to remove all the argument and the return address from the stack right after there is return.

Sure.

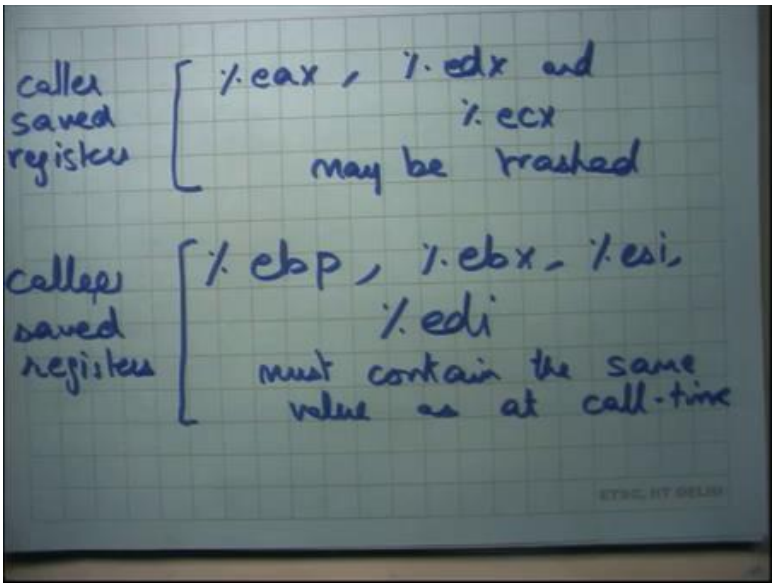
Student: So, while the callee or automatically pops all of them out?

So, question is why does not the callie just remove the arguments before returning, why does the caller need to do it alright.

Student: Sir.

See it is, it does not matter whether the caller does it or the callee does it; there is some the amount of work does not change right. The total amount of work does not change it is just easier to organize it in this way because the return address sets at the bottom. So, the callee just returns and then the caller can clean up the stack above it right. So, the clean the cleaning of the stack is done in the same order in which the stack was set up basically alright. EAX should contain the return value and then there is another convention which says you know certain registers can be overwritten.

(Refer Slide Time: 30:18)



So, for example, EAX, EDX and ECX maybe trashed by the callee and certain registers EBP, EBX, ESI and EDI must contain the same value as at call time right. So, here is another convention which basically say certain registers are allowed to be overwritten by the callee and certain registers must have exactly the same value as the callee as a as at call time right ok.

So, the terminology is these are called caller save registers and these are called callee save registers right. Basically, means the callee is free to trash these values registers. So, the caller must save them before making the call if it actually cares about those values right.

On the other hand, these registers the caller has the contract with the compiler that you know these are not going to be modified. So, it does not need to save them. If the callee does need these registers, then it is the responsibility of the callee to save them and then restore them before returning. So, if the caller and the callee follow this contract apart from that a function a called function is allowed to do anything else advance right. So, the called function can do anything it likes, but it should ensure that on return EIP will point to the return address that was pushed on the stack as it as the stack was setup on function entry.

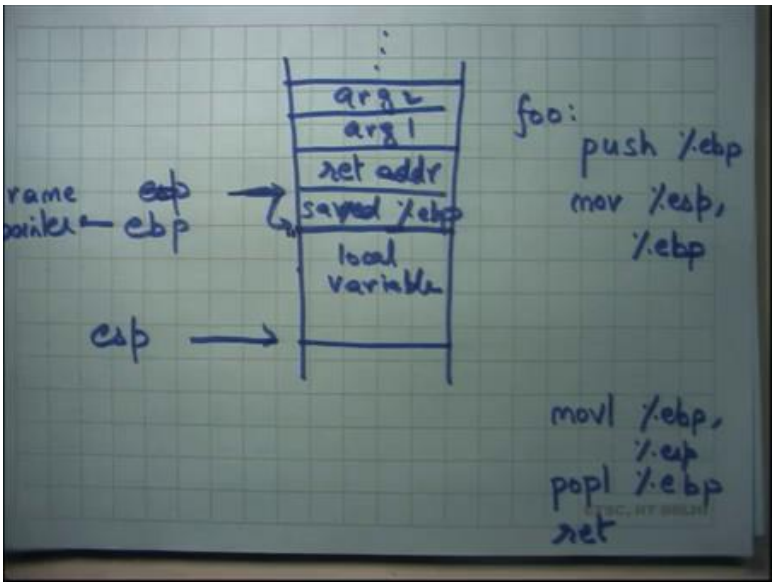
ESP will point to the arguments as I had set them up to be. It may have trashed the arguments I do not care EAX will contain the return value. These registers I may have overwritten, but these registers must contain the same value as I have as at call time fine.

If this contract is followed you know, I can call different functions and I do not and I can I can I can compile these functions separately and call them from one another and they will all work they can they can be forked together easily right. They need to follow this contract apart from that they can do anything else alright.

In your homework, we will ask you a question on why do you need separate caller and callee save registers, why could not, why does it makes to say sense to say that some part of the registers should be saved by the caller. And, some part of the registers should be saved by the callee, why cannot we just say all registers should be preserved right in which case all registers become callee save registers or conversely we could have said all registers should be can be overwritten maybe trashed in which case all registers become caller save registers.

Both these are options, but, but compilers usually take the middle path; half the registers are caller saved and half the registers are callee saved something to think about alright. We going to discuss that later.

(Refer Slide Time: 34:10)



Finally, on a function call so, at the entry of a function call the stack looks something like this, this is the return address this is argument 1, this is argument 2 and so on right and the function now gets to execute. Typically, what GCC does is it will also. So, the first instruction in the function would basically also save the current EBP value. So, called at the saved EBP alright.

And so, and it is so, it is going to push EBP. So, the first instruction of our function will typically be push EBP right. Let us say the first instruction of the function foo. The first instruction is going to push the current value of EBP, and it is going to set the current value of stack pointer to EBP or move ESP to EBP right. What is it doing? It is basically saying that alright. So, let us see what happens. I am going to push EBP. So, ESP becomes this and then I basically say move ESP to EBP.

So, what happens is EBP becomes this then the function may have some local variables and so, it is going to allocate some local variables. And, the way it allocates local variables are also in stack. So, one way to allocate local variables is keep pushing the values if they are initialized or just subtract a value from the stack. So, you know the size of the local variables; if there are like ten variables, if the ten integers in the local variables, then you will just subtract ESP by 40 right.

And so, ESP will point somewhere here, and you can de reference these local variables by using the appropriate offset in esp. And EBP is now pointing to the ESP value at the

start of the function right. So, in other words EBP is acting as a frame pointer right and you can use EBP to actually dereference the arguments. So, if you want to get to the argument number 10, then you just say EBP plus 4 plus 10 into 4 right that is argument number 10. If all arguments are integers let us say right. So, it acts as a frame pointer and the saved EBP is the frame pointer of the.

Student: Sir caller.

Of the caller right the saved, EBP is the frame pointer of the caller right. So, I made a call at the entry of the function the EBP value would still be the frame point of the caller. I first save the frame point of the caller, then I modify the frame pointer to my own frame pointer I modify EBP to my own frame pointer and now I start allocating local variables right.

So, that is at entry and before I return, what do I need to do? I need to do the opposite which is what let us say mov EBP to esp. So, I want to reset the stack exactly as so, I may have grown the stack a lot I may have allocated lots of variables I may have called lots of functions. And, now you know one single instruction way of resetting the stack is just to say move EBP to ESP because EBP is containing my frame pointer. So, if I just say mov EBP to ESP I have reset the stack back up and then I just say POPL EBP that is going to restore the saved EBP to my EBP. So, I get the callers EBP back into the EBP register and then I say return right. So, I go back to the caller right.

So, in this way I save the EBP value, I clobber the EBP value in my code; I write over it, I clobber it. And then I before I turn, I just pop it back. So, recall that EBP was a callee save register and so, the contract is that the EBP should be exactly the same as I gave it to you right. And so, this mechanism is actually doing that for you alright. So, this is a typical GCC behavior that you will stay save the frame point of the previous caller and you will initialize the new frame pointer and before returning you are going to restore the frame pointer of your caller.

What this allows you to do things like you know looking debugging things like you can look at the back trace of the of the current call chain. So, for example, if I want to so, say how many of you have used the backtrace command on GDP or something right.

Student: For the first homework.

For the first homework you have used it right. So, back trace tells you what the call chain is right. So, who called whom who called whom who called whom and how am I here and that is basically done using this frame pointer and saved EBP thing right? So, what it does is it looks at the current EBP. So, firstly, it looks at the current EBP and from that it knows where in which function I am, then it looks at the current EBP and from that it just displaces it by 4 to get it is return address and from the return address it knows it is callee. It is caller right and from that it also knows the saved EBP of the caller and from that he recourses right.

So, he can just do that over and over. So, he gets he has the capability of getting the frame pointer of all it is callers based on this recursion right. I just look at my EBP de reference, it I get the EBP of my caller I look at that EBP de reference I get the EBP of it is caller and so on right. And plus, if I dereference plus 4 of it, I also get the return address. So, with each frame pointer, I also had the return address and that is that is how I basically have the call chain of the entire. So, that is how the back-trace command for example, is implemented on gdb right.

Student: So, sir how does this call chain ends?

How does this call chain end? Well you know, you could just have a terminating EBP value whether let us say it is a zero at some saved EBP is 0 for example, at main. So, you know you could just have some convention that this EBP is going to terminate and also notice that EBP was not strictly needed you know apart from debugging purposes, what am I using EBP for what I am using frame pointer for?

I am using it to reset the stack number 1, but resetting the stack I could have done anyways right because I know how many arguments I have pushed, I just need to pop that many arguments you know or how. If I incremented it by 40 or decremented by 40, I just need to increment it back by 40. So, I did not really need EBP to reset the stack, the second thing I am using EBP for is to get to the arguments right.

So, I say EBP plus 4 is argument number 1 EBP plus 8 is argument number 2 etcetera, but even that is not necessarily needed because the compiler knows that at this point esp is you know 70 bytes of away from the frame pointer and so, because the compiler has this static information all it needs to do is add seventy to esp and then add you know 4 or

10 or whatever to get to the argument. So, EBP is not strictly needed, but it is a convenient way of doing things its edge debugging right.

Student: Sir.

Yes.

Student: Sir what if the called function somehow access the same EBP by also accessing local variables?

So, what if the called function the callee.

Student: Yes.

Overwrites the return address or overrides the saved EBP? So, that then he is violating the contract ok.

Student: But their called function might not know that that location it (Refer Time: 42:20) for it is accessing local variables, but somehow it is coded in such way that while accessing local variables it accesses saved EBP.

So, that is a bug in their function right. If you are supposed to be accessing your local variables and that and your program could potentially access the saved EBP that is a bug in your program in your function unless it was intended that way right. And in fact, such bugs have existed historically in a lot of our programs and such bugs have been used to.

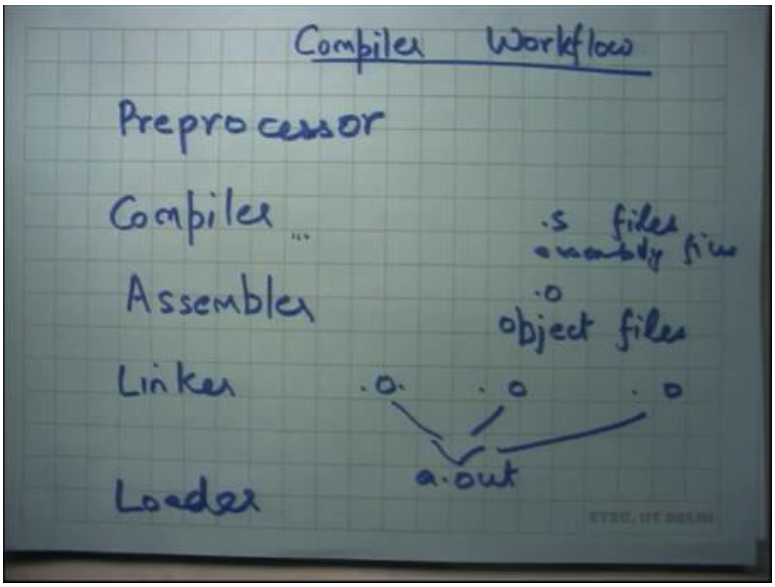
So, the question is if I am accessing a local variable, let us say the local variable is an array and then I offset into an array and the offset happens to be bigger than the size of the allocation and so, now, I can actually clobber my EBP saved EBP or return address and things like that and based on that I can actually change the execution control flow. Well let us say I clobber the return address can actually jump somewhere else.

And this is a very common attack called the buffer over flow attack and such attacks have existed and basically this means that there was a bug in your code what you should have done was before dereferencing should have check the size right.

So, the code should really check the size before dereferencing always right that is the that is the safe guard that that all code must take and such bugs have existed, but people

have are much more aware today and such bugs are much rare alright. Finally, let us talk about the GCC let us come talk about a compiler workflow just to complete this discussion.

(Refer Slide Time: 44:09)



A compiler has a preprocessor which takes your source program which has this hash include or hash define directives and preprocesses them.

So, it is really a source to source transformation it takes C code which has this hash defined directives and just produces another C code. So, it actually does not look at the syntax at all it has does macro expansion it just replaces that hash define variable with it is value or hashing it expands hash include with the contents of that file and so on right. So, that is a preprocessor. Then there is a compiler that takes a source file, parses it, and generates an assembly code right.

Assembly code is something that we have seen so far like move all. These all these human readable assembly code like movl, addl and so on you know with their arguments. So, compiler basically parses C syntax and generates assembly code alright. Then there is an assembler that takes the assembly code which is the string which a human can read and makes binary code that a machine can read right that is an assembler.

The binary code is stored in dot o files like which are you know called object files compiler makes dot s files which are assembly files right. So, it gets assembled into binary code. These are called object files is machine readable code and then there is a linker which takes multiple object files combines them to make one executable. Let us say a dot out right. So, you could have multiple file multiple source files each source file gets compiled to an object file. These functions, these files could have independent functions like foo bar etcetera as long as they form follow the function calling contract it is ok. Foo can call bar in different files and the linker is going to ultimately link them all and make them and attach these all these references between foo and bar alright.

And then there is a loader takes a dot out and starts a turning alright. So, where have we seen the loader before? In the exit system call right. So, when you call exit system when you call the exit system call, you basically give the name of the executable file and inside the OS, there is a loader that will load that file.

The file should be in a certain format and that file is going to now get pasted into the memory space of that process and it will you will be started running by transferring control to the first instruction of that process right. So, that is the job of the loader so, that is how you know that is the workflow of a compiler and linker and a loader alright.

So, let us stop here and we will continue next time.