Computer Architecture
Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 11
x86 Assembly Language Part-II

(Refer Slide Time: 00:25)

Let us start this part by describing the mov instruction. So, the mov instructions arguably the most versatile instruction in the x 86 instructions set, say it can actually do a lot. So, in terms of RISC instructions, it combines, it is a combination of three things.

(Refer Slide Time: 00:46)

So, in if you recall in the RISC instruction set, we used to have a mov instruction which was nowhere as powerful as this. So, the mov instruction in the RISC instruction set was actually moving the value of a register or a constant to a register. So, the CISC mov instruction does that. So, we can think of it as a RISC moves plus it is also a load. So, you can load a value from memory and put it in a register, and it is also stored. So, it can store, something from a register to a memory address. So, the mov instruction in x 86 is equivalent to a RISC move a load and store all combined into one.

So, what is the semantics, the semantics is very simple, the semantics is that the mov instruction takes two operands. The first operand is a source as well as a destination right. So, it can either be a register or a memory, the second operand is clearly a source. So, it can be a registered a memory or an immediate. So, as discussed in some other earlier slides, that most x 86 instructions are actually in a two address format, they take two operands, and the first operand is both a source as well as a destination.

So, we should see this is standard format for almost all instructions. So, here the idea is like this. So, what is the mov, for the case of the mov instruction, we you know the duality of a source and a destination is not very important, so we will not over stress on this aspect here.

So, say let us consider an example, let us say we write move m o v eax, ebx. So, in this case this is very simple, that the first instruction is the destination the second, sorry the

first operand is the destination, and the second operand is a source. So, we move a value the source value, to the destination; the value of the source to the destination. So, this is extremely the mov instructions extremely versatile, instead of this I could have always written numb, something of this type.

So, in this case, this is actually a memory address right. So, what we are doing is that we are reading from memory. So, the first thing that we are doing is we are computing the value of this expression, by accessing the registers esp and eax. Then we are accessing memory we are reading 4 bytes, and then we are saving those 4 bytes in the register ebx. So, this is a load. I could have similarly, just interchanged both and maybe just written.

So, this actually would have been a store, because we would have read the value of ebx and put it in a memory address. So, the former is an example of a load, and the latter is an example of a store. So, the mov instruction can move values between registers. It can move a value from a register to memory, which is the same as a store. It can move a value from memory to a register, which is the same as a load. So, as you can see we have pretty much combined the functionality of three RISC instructions; a normal register or constant mov instruction, a load and a store instruction all into one, which makes this particular instruction very interesting, because if we just know this we will be able to write a lot of assembly programs, and mind you the standard rules that we discussed in the last few slides still hold, which means that both the operands cannot be, both the source operands cannot be memory locations.

Which means that we cannot use this instruction, to move values between two memory locations, but other than that we can easily achieve a load and store, and every memory operand will have the following form, it will start with a left square bracket, end with a left square bracket, and the value of the memory address will be specified within the address, where we can have a base register scaled index and n offset, and in x 86 parlance an offset, is known as a displacement.

Similarly, we have the movesx and movezx instructions. So, the regular mov instruction assumes to the source and the destination to the same size, but the movsx and the movzx instructions, actually assume that the source and destination have different sizes. So, let us consider this. So, the genetic semantics is, that let us say we are trying to. So, this sort

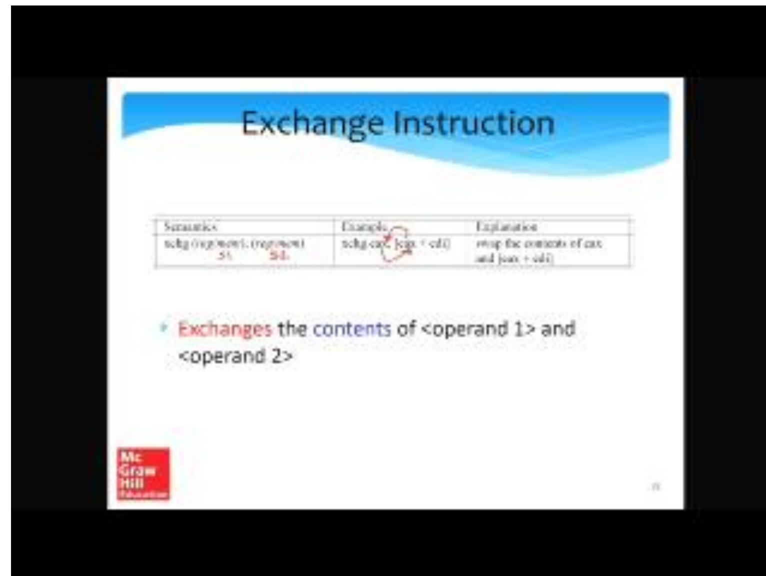
of slightly restricted semantics in the sense, the first operand can only be a register, and the second one has to be registered or memory.

So, once this value is being moved over here, we can for example, write movsx ax bx. So, mind a bx is a 16 bit registers and eax is a 32 bit register. So, what we do is that we read the value of bx, and s stands for sign. So, the s here stands for sign, sign extends. So, the sign extend, extend the sign of bx, and then we save it in a x. So, second operand in this case can be either 8 or 16 bits; that is fine and we extend the sign of bx and save it in eax.

Similarly, where the movezx instruction, where we do not extend the sign, but treat it as an unsigned number. So, with zero extend it, which means that we add the, we the m s b bits instead of them being replaced by the sign bit, they are replaced by the number zero, and this number is stored in the registered ax and so in this case also the second operand is can be either 8 or 16 bits, but instead of extending the sign, we treat the number in the smaller registered bx as unsigned, and replace the m s d bits. Sorry do not replace the m s b bits, but set the m s b bits to be 0.

That is pretty much it. So, it is a special kind of mov instruction, where we are moving from a source with which has a less number of bits, to a register that can accommodate more bits. So, the question is that those extra bits what should they be set to. these extra bits which are in m s b position in the case of s x are set to the sign bit, So, which means that the value is maintained in the case of z x they are set to zeros which is an unsigned extension.

(Refer Slide Time: 08:33)



So, we have seen the mov instruction movesx and moveszx. So, let us now take a look at the exchange instruction x c h g. So, the exchange instruction is also very important, but we will not be able to fully understand the meaning of an exchange instruction and why it is that important, till we actually read chapter eleven, and even in chapter 11 we will only discuss a little bit of it. So, exchange instruction is very useful while writing parallel, alos we will discuss maybe a little bit of exchange over there.

So, the main idea is that exchange takes two arguments two sources which are also destinations by the way. So, e source 1 can be a resistor or memory, and source two can be register memory, I can maybe write s 1 and s 2.

So, for example, this can be, but mind you both cannot be memory locations. So, the same constraint still holds. So, we can write exchange eax and a memory location eax plus edi in square brackets. So, we will essentially swap or exchange the contents. So, the contents of eax will come here, and the contents of this memory location will come in to eax right. So, it will exchange the contents of eax and eax plus edi will just swap the contents, of operand 1 and 2. So, what are the instructions you have seen up till now. We have seen the move series of instructions and exchange.

(Refer Slide Time: 10:08)

So, given the fact that we have seen this, there are some more instructions that we need to see, to get a better idea of the memory and the data movement instructions. So, two of those instructions are push and pop. So, why do we have this? The reason we have this is, because an x 86 processor, is explicitly aware of the stack, unlike simple RISC where that was one of the case.

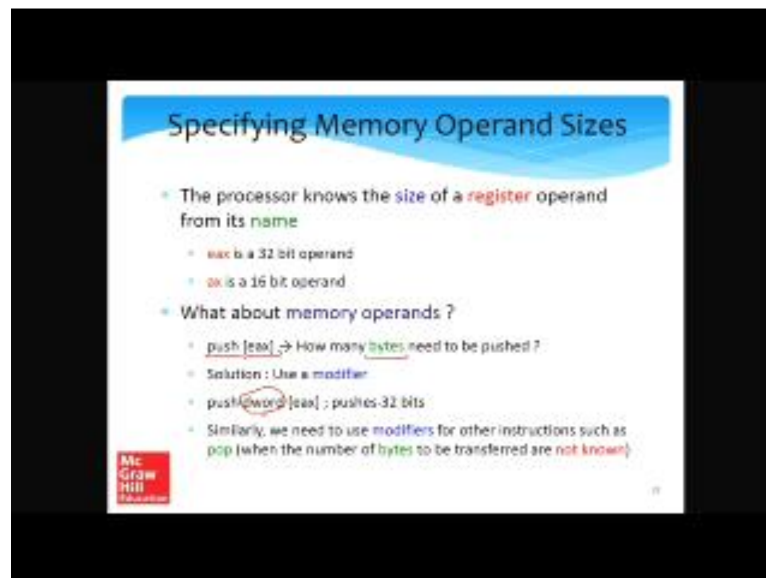
So, an x 86 processor is aware the stack pointer is stored in the register esp and so that is the reason it defines two instructions called push and pop, which take a single operand. So, let us consider the push instruction. It takes a single operand. The operand can be a registers, can be a memory, or an immediate value a constant. So, for example, we can see you push ecx on the stack. In this case the value of. So, this is similar to this small code snippet over here, this is exactly what the processor would do. So, given the fact that this is a cisc instruction set, a single instruction can actually do a lot. A single instruction can do the same as 4 or 5 equivalent RISC instructions, maybe sometimes even more.

So, what we would do is first we will read the value of ecx put it in a temp value, then we will decrement the stack pointer by four right, because we are pushing it in assuming a downward growing stack, and then the value of, then what we will do, is that we will access the new stack location, put in the value of ecx.

Now, let us consider the pop instruction. So, the pop instruction does the reverse. So, that is the reason. So, we will see the explanations of the same sequence of steps are just written in reverse, to make the understanding easier. So, in the pop case, we can pop the value from the top of the stack, and put it in a register or memory .For example, it is a pop ecx. So, just you know these statements are in reverse. So, first we read the value under the stack pointer into a temporary variable. So, this is sort of an equivalent programmatic explanation of pop would do, then we increment the stack pointer, and then what we had read from the stack, we assign that to ecx.

And it is the definitely possible to write this program without the temp variable, but it has just been and to show the order of actions of, what exactly the processor would internally do and, but definitely we can make the program simpler, if some of you are not happy with three sub instructions we can replace it with two, but the basic idea would still remain the same, that we the push the value onto the stack, and retain the stack pointer in. While pushing we decrement it, and while popping we inc increment the stack pointer.

(Refer Slide Time: 13:29)



So, specifying memory operand sizes. So, let us take a look at it. So, the processor knows the size of the register operand from the name of the register for example, eax is a 32 bit operand we know that, ax is a 16 bit operand we know that, an r ax is a 64 bit operand we also know that, but what about memory operands. So, let us take a look at, some

instruction or the form push a x. So, in this case, we read the value of ax we get a 32 bit memory address, we access memory, we read some bytes and push them onto the stack, but the question is how many bytes need to be pushed. Is it 2 bytes 4 bytes 8 bytes how many bytes.

So, that is the reason it is necessary to provide a modifier to the push instruction, such that especially when we are trying to push something into a memory address, we can specify how many bytes need to be pushed. This is not necessary for a register, but for a memory operand this is necessary. So, in this case we will actually push 32 bits, because we are using a d modifier, and d means 4 bytes or 32 bits.

Similarly, we need to use modifiers for other instructions such as pop, when the numbers of bytes that need to be transferred are not known. In a case of a register since we know it is size, we are assuming that we will fill up the entire registers, but in a case where we are not really sure, because we are transferring something from the stack to another memory location, we do not know for sure. So, that is the reason it is necessary to use this particular modifier over here; such as d word to say that we want to push 32 bits.

(Refer Slide Time: 15:25)

Let me now discuss some of the common modifier, depending upon the size, depending upon the number of bits that need to be transferred from memory, we can use different kinds of modifiers, with instructions that access memory. The first modifier is byte. So, it specifies at 8 bits or 1 byte needs to be transferred. The second modifier is word. So, it

means that 16 bits need to be transferred. So, I need to make a point here in simple RISC and in ARM, we assume that a word is 32 bits of 4 bytes. So, this is not correct in x 86, because of historical reasons. So, in x 86 a word is assumed to be 16 bits or 2 bytes. So, this is what a word means, in the context of x 86.

So, we use the modified d word or double word in the last slide, slide number 27. In this case we assume that we are transferring 4 bytes or 32 bits, to support till the newest and latest 64 bit instruction set, we have the modified q word which is squared word or 8 bytes or 64 bits. So, after we have seen this, let us take a look at this example over here which I have deliberately chosen, as a slightly complicated example. So, we will take a look at 4 x 86 assembly statements, and at the end we need to answer this question, what is the value of ebx after these four instructions. So, let us not take a look at answer just right now. So, the first; let me maybe give you 10 seconds for you to take a look at these four instructions and work it out, and after that I will work out each and every instruction and show you the steps.

So, let us see. So, the first instruction sets that a register eax to 10. So, this is nice and straightforward the second instruction moves eax to the location of the stack pointer points to. So, let us say the stack pointer points to this location, and since it is a downward growing stack which grows downwards towards lower memory addresses. What we do here, is that we save the value of eax which is 10 to the location pointed to by the stack pointers. Subsequently what we do is that we take the value that is contained in the location of the stack pointer points 2, and push it again on the stack. So, this particular instruction is complicated. So, I would like the readers to pay special attention. So, here what we do is, the first thing in any push instruction, is we see what is the value of the argument, you know what is it that needs to be pushed.

So, what needs to be pushed is a d word or a double word or 4 bytes. So, we need to read 4 bytes from the current stack pointer. So, the current stack pointer contains 10. So, essentially we are pushing 10. So, what we do now. So, if you take a look at the pseudo code of push; that was explained a couple of slides ago, what we now need to do is, that we need to decrement the stack pointer by 4 in the new location, which is the next location on the stack, we need to push the value of 10, which is what we just read. Subsequently the next instruction; so now, what is the current status, the current status is of the stack pointer, points to this location which is the next location. So, let us see

another memory address, the previous memory address of the stack pointer was 1000. So, the current memory address of the stack pointer is 1004.

So, the next instruction, the last instruction reads the contents of the memory address the stack pointer points 2 which in this case is 10, and the value of 10 is transferred to ebx. So, what is the value of ebx at the end of this code snippet, it is 10. So, the important point to remember, is that in any push instruction we first read the value, read the value of the operand that we are trying to push, then we decrement the stack pointer and essentially in the new stack top, at the new stack top we push in the value. So, this instruction I deliberately made it complex. What was the complexity the complexity was, that the argument of the push instruction was a memory address that was determined by the current value of the stack pointer. Just to ensure that the understanding is clear. So, let me just go back to the point where we define the push instruction. You see here we first read the value of the push argument, whatever is being pushed into a temporary variable, then we subtract the stack pointer, and at the new location we push in this.

So, the advantage of this, is a just in case this argument contains the stack pointer it will get taken care of, as we saw in this particular example.

(Refer Slide Time: 21:22)

So, now that we have taken a look at the memory instructions. Instructions are basically moved and it is variants, exchange pushes and pops right. So, let me just write it let me just go back to the previous slide, and write all the instructions that we have seen up till

now. So, we have seen moves, and we have additionally seen two of it is variants s x and z x.

We have also seen the exchange instruction that exchanges, the values at two locations. We have seen the push and pop instructions; that are explicitly for dealing with the stack. Interesting thing is that in x 86, the hardware is explicitly aware of the stack, and it uses the stack pointer. Not the case in other architectures, not in simple RISC and ARM, but in this case the push instruction, actually reads the stack pointer decrements it, which is good.

So, now let us take a look at ALU instructions. ALU instructions follow exactly the same format, as we had discussed earlier when you get the generate format of to address instructions. So, they take two operands; the first operand, is the first source as well as the destination. Second operand is the second source. So, the first operand can be either a register or a memory. Second operand can either be a register or a memory or an immediate or a constant. Example, if you have add eax ebx, in this case explanation is that we are setting eax equal to eax plus ebx. So, mind you the same rules hold. Rules are that both these operands, both the source operands s 1 and s two, cannot be memory operands, only one of them can be a memory operand; that is the first rule that we talked about. So, and also the second rule is that, the first operand is a source as well as the destination.

So, the subtract instruction is similar, it is add and sub, so the names are also similar. And here also we set ax equal to eax minus ebx to ARM. So, in ARM recall that we had two instructions, where we were able to add with a carry or subtract with borrow. So, the x 86 is very similar, it also has the adc instruction; adc instruction adc eax ebx, will essentially set first add eax plus ebx, then add the value of the carry bit. So, this is just to ensure that we can add very large numbers, even if they do not fit within a register.

So, we can add two pair of numbers. We can add two numbers or a single pair of numbers, then the carry bit can be saved, and it can be used in a future addition. So, that is the reason this particular instruction is eax plus ebx plus the value of the carry bit; sbb the same thing, adc is add with carry, and sbb is subtract with borrow. So, let us consider an example is sbb eax ebx. So, we are setting eax as eax minus ebx, and this is minus the carry bits. In this case the carry bit actually works as a borrow bit. So, unlike ARM and x

86, the carry bit in the borrow bits are the same and have the same connotation. So, basically what we do we first do a regular subtraction, and then we subtract the borrow bit. Call it the carry bit, but it can be interpreted the borrow bit as well. So, we do the subtraction in that manner.

So, since this part we have seen earlier also in chapter 4, and for those readers have not read chapter four, let me just explain once again. So, the main advantage of adc and sbb, is to ensure that we can add numbers that are much larger, add or subtract numbers that are much larger than 4 bytes, in the case of a 32 bit o s. would we do that we will essentially save the numbers in a set of registers add one pair, and then save the carry, use a carry to add the second pair and so on.

(Refer Slide Time: 26:05)

So, let us now take a look at some simple one address single operand instructions in the one address format. So, these are very handy and useful actually. So, the instructions are inc for increment, dec for decrement, and neg for negatives. So, they take a register or a memory as an argument. So, in one case we increment the value of the register or the memory location by one; in the case of dec we subtract, and in the case of neg what do we, do we multiply the number with minus 1.

So, let us consider an example nothing is complete without it us write an x 86 assembly code snippet to compute eax is equal to minus 1 times eax plus 1. So, in this case what do we do, since the value of eax is changing what we do, is that we first increment eax

which means setting eax to eax plus 1 and then we do call neg eax which means subtract eax plus 1 with minus 1.

(Refer Slide Time: 27:24)

Now, let us take a look at the compare instruction. Always have a compare instruction we have to have rather a compare instruction, we cannot live without it. So, the compare instructions job is exactly the same as was the case, in simple RISC and ARM. So, here we compare two operands. So, in this case source one and source two. So, the first operand can either be a registered or memory. Second operand can either be a register or memory or an immediate. So, we need to compare both of them and save the re results in the flags register, which as we saw comes with different names flags for sixteen bit e flags for 32 bits, and r flags for 64 bits. So, in this case if we are comparing eax and ebx plus four. So, ebx plus four is actually a memory address, we are comparing the contents at this memory address.

So, we compare both the values in eax and ebx plus four, the the memory operand corresponding to be expressible, and we set the flags. We can compare with an immediate also. So, we compare the contents of a ecx with 10, and we set the flags.

(Refer Slide Time: 28:49)

Multiplication and Division Instructions

- The imul instruction has three variants
  - 1 operand form → Saves the 64 bit result in edx:eax
  - eax contains the lower 32 bits, and edx contains the upper 32 bits

Now, let us take a look at multiplication and division instructions these instructions are always complicated, but they are not all that difficult, they are similar to ARM, but there are certain complexities involved. So, let us take a look at the imul instruction in the single address format. So, in a single operand form in the one operand form, which is the first one what do we have. So, what we do is that if we call imul ecx. So, by default it multiplies the value in eax with that of ecx. So, that is the default behavior that whatever is there in eax it will multiply it with ecx.

So, now in the case of a 32 bit instruction set, the value of eax is 32 bits in the value of ecx is also being put into 32 bits. So, you can see that the product can in theory be a large number right. So, how large can can the product be, let us assume it is twos complement. So, basically in a twos complement system, if you multiply minus 2 to the power 31, with minus 2 to the power 21, it is 2 to the power 62.

So, pretty much the product has to be represented in. in this case 63 bits 63 the bad numbers. So, let us take 64 bits a power of 2. So, basically we will unless we allocate 64 bits is a chance that we will have an overflow, if we multiply two numbers which is a problem. So, in a single operand form what we will do is, we will multiply whatever is the value given, with the value stored in eax the final product will be 64 bits, and we will save the upper 32 bits in edx, and we will save the lower 32 bits in eax. So, what are we doing, we are multiplying in this example eax with ecx.

So, assume eax contains a value seventeen, and the ecx contains the value thousand will represent the product as a 64 bit number all right. So, the 64 bits is 8 bytes. So, the upper 4 bytes will have an upper 4 bytes in a lower 4 bytes right; significant 4 bytes and least significant 4 bytes. So, the upper 4 bytes will get saved in edx and the lower 4 bytes get saved in eax.

(Refer Slide Time: 31:44)

Let us now consider the two operand form. So, the two operand form given in the next slide, basically multiplies takes one register at the first source, and then we multiply the register with either the contents of a register or a memory. So, we multiply ecx with eax plus four; this is the memory address the contents of it. So, in this case, what we do is that we do not save a 64 bit product, rather what we do is we treat it as a regular instruction, and absolutely regular instructions similar to you know add and subtract and so on.

So, in this case we will just set ecx as ecx times eax plus four if there is an overflow the overflow flag will be set, but will not save a 64 bit product will simply take the first register, treated as. First registered in this case is a source as well as a destination as has been the case other ALU arithmetic logical instructions. So, what we do is that we take the first and second sources multiply them, and save them at the location of the first operand

(Refer Slide Time: 33:02)

imul Instruction - III

- 3 operand form
  - First operand (destination) → register
  - First source operand (register or memory)
  - Second source operand (immediate)

Now, let us consider a three operand form of the imul integer mul instruction. So, in this case we have a register the first operand is register, the second operand is registered or a memory location, the third operand is an immediate. So, in this case since the cisc instruction set we can take the liberty, of having a lot of instructions, a lot of different instruction formats, pretty much you know whatever suits us. So, in this case what we do is that, the first operand is the destination. Then we will have source one: the first source and the second source. So, what we do here is that this is the first source operand you multiply this with five, because the last one is immediate. So, we multiply the first source with 5, which is an immediate, and then we store the result in the first operand which is a register.

So, these are pretty much the three variants of the imul instruction. So, as you see, you know these three variants were chosen with a lot of care, and you know thoughtful consideration. First variant was chosen, because in some cases we might want to store a 64 bit product, and so that additional facility was given, and this facility is there in the ARM instruction set as well. The second case we want to treat in, implement and imul instruction the same way as we have been implementing other instructions like add and subtract. In the third case designers realized, that maybe it is very common, that you multiply the contents of a register or memory with an immediate, and save it in other register. So, we should have an instruction for it. So, that is the reason they have an instruction for it.

(Refer Slide Time: 35:03)

Let us take a look at the idiv instruction, which is definitely complicated. So, let us go back as you see idiv instruction takes a single argument, and this is exactly what we will expand in the next slide. So, the idiv instruction, the integer divide instruction, which is complicated, we shall see why. This is also one reason why the original designers of the ARM instruction set, for the early ARM instruction sets, not the later ones, did not have divided instructions, because you know it is somewhat complicated. We will see why and we will also correlate this with what we had done in simple RISC, to sort of resolve the issue.

So, here the idea is that we take a single argument which can be registered on memory. So, what would an example be? Something like, idiv ebx. So, what we do, is that we consider a 64 bit number, in the case of a 32 bit instruction set, where the number is stored in edx and eax, edx stores the upper 4 bytes, eax stores the lower 4 bytes.

So, same see the logic is similar, if you multiply to 4 byte numbers the product can at most be 8 bytes. So, here what we do is, that we can, thats the reason considered a larger number and divided by a smaller number, and it is sort of an exact analogue of the single address form of the imul instruction were assuming that the dividend the number that, we will divide is an eight bit number, it is stored in edx and eax with the upper 4 byte. So, let me maybe draw it, it will be easier to understand that way, if this is the dividend, we are

assuming that the dividend itself in this case. Case meaning the 32 bit instruction case is 8 bytes, the upper 4 bytes are in edx, and the lower 4 bytes are in eax.

Now, in this case we are dividing it by the cons. what is the devisor, and the divisor is this argument, which in this case is ebx; such a divisor is source one. So, once we divide. So, the main difficulty in a division instruction is that, a division instruction actually produces two results. So, let us try to divide integer divide, 10 divided by seven. So, in this case the results, there are two results one is the quotient, which is one, the other is the remainder which is three. Another example fifteen divided by seven quotient is two and remainder is one.

So, as you can see there are two results, essentially we need two registers to store them; one register to store the quotient, and one register store the remainder. So, this makes our life very simple. So, by default eax will store the quotient after an idiv instruction, and edx will contain the remainders. can sort of think as eax and edx as a pair, that will come together very very often. And so in this case edx and eax are containing the dividend, then they are getting overwritten. So, eax will ultimately contain the quotient at the end, and edx will contain the remainder all right. So, let me just summarize whats written here. The dividend contains is contain an edx eax. In a 32 bit instruction set edx contains the upper 32 bits, eax contains the lower 32 bits, or the input operant contains the divisor.

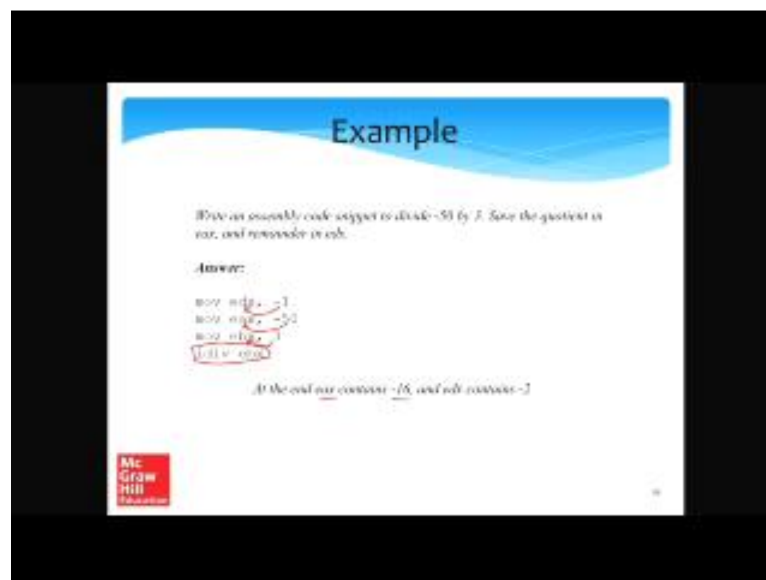
So, which is basically either a register or a memory address? After the end of the division eax will contain the quotient, edx will contain the remainder right. So, this is very simple, but here is one mistake that most students do, and this is a very very very very important point to keep in mind. So, let me put a star and this is, so important. Let me put on one more star right. So, let me also write importance in a just in case, you know. So, basically what is the idea, the idea is if the dividend is positive. So, let us assume the dividend is 10, So, in this case edx would contain all zeros, and eax would contain 10, and let us write 10 and x which is 0 x a.

What if we have a number? So, we have inside eax, we have minus 10, what do we do then. So, here is the mistake that students to, what should edx contain. edx should contain, is edx contain all ones, because we are essentially sign extending the number, or extending the sign of the number, which in this case is minus 10 minus 10 represented in binary of course,. So, we are extending its sign from 32 bit to 64 bit. So, that is the

reason edx should contain all ones right. and if this is not the case, then we will have a lot of trouble, in the sense if you set edx to all zeros that is going to be wrong, and this is a large source of bugs in the idiv instruction, and setting any number to all ones is actually very easy, all that we do is mov edx minus 1. This is all that needs to be done.

So, if a number is negative and it is fitting within 32 bits and the number is within the eax all that needs to be done. in the case of a negative dividend is that we need to extend the sine of eax. say eax is positive what we do, is that we set edx to zero, and if edx if eax is then negative what we do, is that we set edx to all ones, extend the sign of eax pretty much, and the way that this can be done is, we move minus 1 into edx. So, this is very very important, it is a big source of bugs, I request all readers listeners to keep this in mind.

(Refer Slide Time: 42:20)



So, the learning is never complete without an example. So, let us write an assembly code snippet to divide minus 50 by 3, and let us say the quotient in eax and remainder in edx. So, minus 50 what do we need to do, we mov minus 50 into eax, we extend it sign we mov minus 1 into edx. We mov three into ebx and we call the idiv instruction on ebx. Automatically the eax will contain the quotient, which in this case is minus 3, and edx will contain the remainder which in this case is minus 2.

(Refer Slide Time: 43:06)

Now let us take a look at logical instructions. So, logical instructions will do a boolean bit manipulation; the logical of the instructions that we have standard instructions that we have seen in the case of simple RISC and in the case of ARM also. So, they are, and or in xor. So, these are the standard logical instructions, that take two source operands, and we have the not instruction which takes a single source operand.

So, the way that they work is very similar. The and or and xor instructions work exactly the same way as three other arithmetic counterparts such as add and sub work. So, the format is exactly the same. The not instruction works the same way as ink and deck, and other single source arithmetic instructions. So, in this case not of eax, would basically take the value of eax compute bitwise complement, and store it in eax itself.

(Refer Slide Time: 44:15)

Now let us take a look at shift instructions. So, similar to simple RISC and ARMS I am not, I am deliberately going slightly fast here, because my assumption here is, that readers have an idea of what shift instructions are, and so this idea has come from chapters two, and chapter three, because you know I have been seeing all the time it is not possible to explain the concepts or the fundamentals of assembly language, and a complex assembly language together at the same time. It will just make understanding very hard. So, if listeners are still having difficulty, they can always go back to chapter three when simple RISC was introduced, take a look at shift instructions and come back.

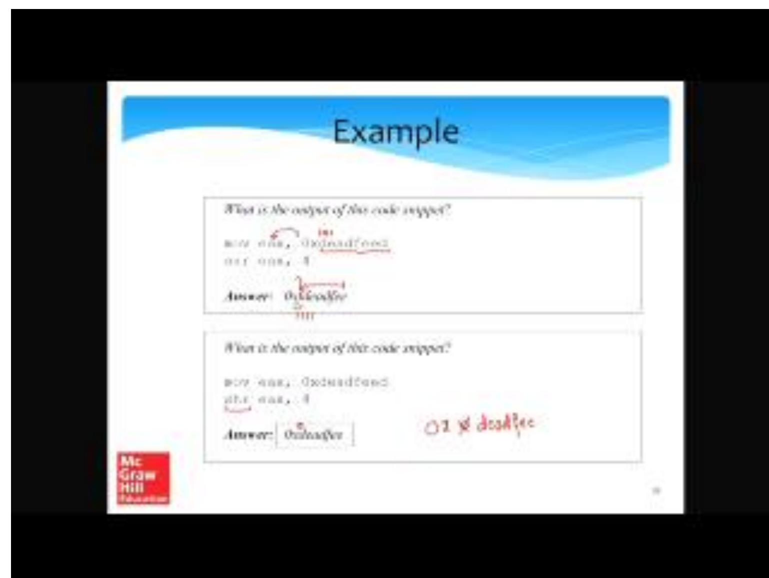
So, we have three shift instructions; one is an s a r instruction which is a arithmetic shift right, shift arithmetic right, which is essentially the same as dividing a number by a power of 2. So, in this case what we do is that the format is that the first operand which is the source, as well as the destination, can be shifted with an immediate number, with a number of positions as you need to shift with. For example, writing s a r, s a r eax 3 will set eax to eax right shifted by three places.

Similarly, we have the s h r which is the logical shift right, which is the same as dividing the unsigned representation by a power of 2. So, let me write, it may be unsigned. So, in this case we have always been using the three right arrow conventions. So, the format is exactly the same s h r eax 3 as an example. Here we set the value of eax, the eax right shifted in a logical manner, where the m s b s are being replaced with zeros, not the sign

bit all right. Then the shift left instruction does not have when arithmetic and logical counterpart, it is only one instruction. So, the one instruction has actually two codes and both of them are equivalent s a l and s h l, essentially they are the same instruction. So, this can be arithmetic left and logical left are the same. So, the same instruction s a l or s h l, format is the same, first operand can be a register memory, and the second operand can be an immediate.

So, what we do is, we log do a logical shift left, we take the register, and we take a number, we set the register as eax left shifted by two places. So, we use the left shift, operand here. So, it is important that the second operand the shift amount, needs to be an immediate.

(Refer Slide Time: 47:18)



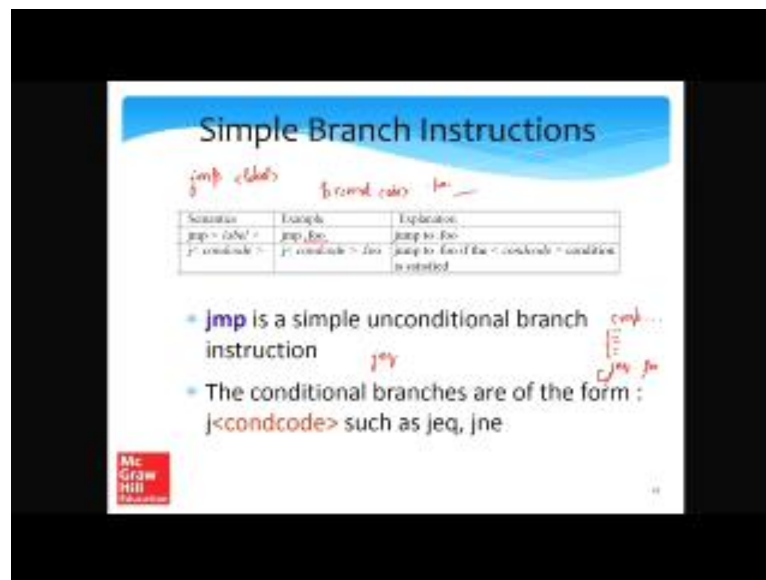
So, nothing is again complete without an example. So, what is the output of this particular code snippet? So, let us consider this is a eax number actually it sounds like an English word, but if we take a look each and every digit over here is the hexadecimal digit. So, it is dead feed. So, let us move this to eax, and let us shift it to the right by four places. So, four places means, pretty much by 1 eax digit. So, then the remaining the seven digits of the n become dead fee, the question is what is the m s b.

So, this is easy to find out, because d is actually 13, and 13 representation is 1 1 0 1. So, the m s b is 1. Say the m s b is 1 we in the case of an arithmetic shift right, which is replicate the m s b. So, we put in an f, which is four ones. Now let us do the same thing,

but with a logical shift right. In this case the only difference is, that the m s the most significant digit is actually zero the answer is zero x zero dead fee. Since zero need not be explicitly mentioned, we have not mentioned it. So, the answer is 0 x dead fee.

So, now, we have taken a look at the machine model of x 86 simple integer instructions. So, now, we will proceed and take a look at branch instructions.

(Refer Slide Time: 49:11)



The first thing that we need to note about x 86 is branch instructions, is that branches are actually called jumps. Anyway there is nothing in a name Shakespeare, says called a rose chrysanthemum a rose will remain a rose, the branch will remain the branch.

So, branches pretty much work the same way as they worked in simple RISC and in ARM. So, here we have the jump instruction which is the unconditional branch. So, we always jump to a label, and the label essentially indicates the instruction that we want to jump to. So, in this case if you are jumping to label dot foo, it essentially means that we are jumping to the instruction; that is pointed to by the label dot foo x 86, also has a lot of conditional instructions. So, the conditional branches are very similar. So, recall that in ARM and simple and x 86, sorry in ARM and simple RISC are the conditional branches, where the form b and then the condition code.

So, there were instructions of the form b e q b n e and so on. So, in this case instead of a b, we have it a say an instructional from j e q, basically means a jumped if equal and

equal, so where does equal come from. Well the last instruction that has set the flags. So, basically there are some flags setting instructions, compared to one of those instructions which can set the flags after that there can be many instructions. If these instructions the only condition is that, they should not set the flag for this example, then we have j e q and then the name of a label.

So, if they compare instruction here led to an equality, then the j e q condition will be true, and we will branch to the label dot foo; otherwise the program will just fall through in a sense of the next instruction will be executed.

(Refer Slide Time: 51:34)

Condition Codes in x86

Condition code	Meaning
o	Overflow
no	No overflow
b	Below (unsigned less than)
nb	Not below (unsigned greater than or equal to)
e/z	Equal or zero
ne/nz	Not equal or not zero
be	Below or equal (unsigned less than or equal)
s	Sign bit is 1 (negative)
ns	Sign bit is 0 (0 or positive)
l	Less than (signed less than)
le	Less than or equal (signed)
g	Greater than (signed)
ge	Greater than or equal (signed)

So, there are many kinds of conditions the same way as an ARM. We had 15 conditions, we have many conditions also in x 86. So, the condition codes are like this. So, all of these conditions code codes come after j. For example, j o would mean that branch if there has been an overflow right, in the last instruction which is doing some kind of arithmetic of operation.

Similarly, j n o means no overflow. There are some comparison operators for unsigned comparison; so for unsigned less than that is called below. So, j b basically means jump, if it is less than in an unsigned sense, and n b means not below. Then we have two conditions e and z which are actually the same conditions. So, they can be used interchangeably, no problem in that j e and j z, which means that the last comparison or the last flag setting instruction, should have resulted in an equality.

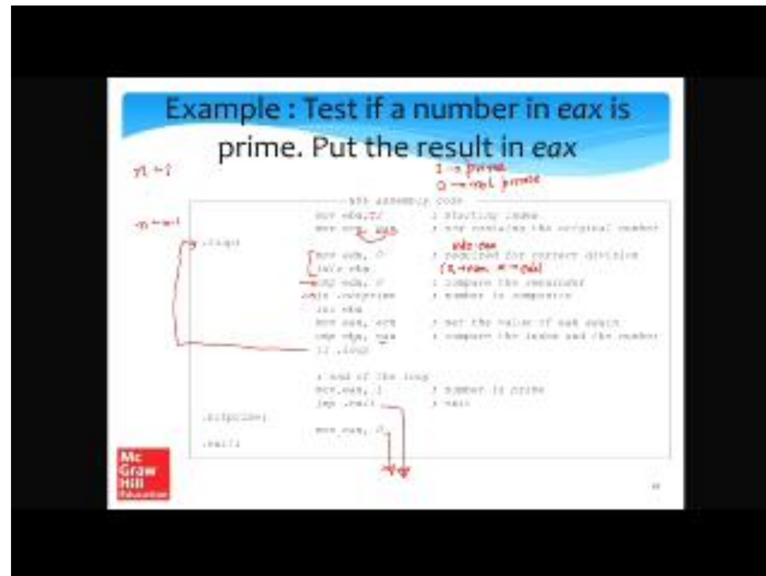
So, recall that how do we check for equality, essentially when we are comparing two values a and b, and when two values a and b have been compared, what the processor actually does is that it subtracts b from a. So, it computes a minus b. So, the a minus b is equal to 0, it essentially means that a is equal to b. So, we have equality here, and it is same as a minus b being equal to 0 c equal or 0 is sort of the same condition. There are other flag setting instructions as well. So, essentially there we see the value of the result, if the result is zero this particular flag will be z. So, the opposite of e and z is n e and n z. So, that is, we will have two instructions j n e and j n z which are also totally equivalent.

So, we introduced the b flag here, below and sign less than. So, b e is below or equal and sign less than or equal. So, now, we have after these comparisons we have another class of condition codes. The condition codes are s and n s, s is basically means the sign bit is one, or alternatively the number is negative, n s means that the sign bit is zero, which means as the number is zero or positive. So, recall that we had similar flags in ARMS also p l and m i. So, we have you know very similar flags in x 86.

So, this is the similarity between the instruction sets, between simple RISC ARM and x 86. So, what we get to see over here, is that most of the core mechanism such that branches and condition codes, are similar, and to a large extent to this, they are also inspired from some earlier designs, but the basic point to note is, that the difference between instruction sets, in a lot of cases is only superficial, lot of their fundamental underlined mechanisms are still the same.

Now, we have some signed operators l and l e. So, l means less than signed less than, is a signed comparison, l e means less than or equal signed, g and g e mean greater than, and greater than or equal all right. So, basically a branch of the form j g would mean jump if greater than, and j g e would mean jump if greater than or equal in a signed sense.

(Refer Slide Time: 55:33)



So, now that we have seen this basic condition codes, we are all set to write a program and so this is an assembly program. My assumption here again, is that readers have some familiarity of assembly programs, from chapter three. Otherwise in this program, I will still explain all the steps, but it might be slightly difficult. So, the example here is test if a number eax is prime, and they put the result back in eax results, what is the result typically it is a Boolean result if the number is prime, you essentially put one right one indicates for the number is prime. And zero indicates for the number is not prime, that is a Boolean result.

So, what we do. So, what is the basic algorithm, how do we find if a number is prime or not. So, what we do is, let us say given a number n. So, here the assumption is the number is large enough. So, we do not want any corner cases. So, given a number m v n we divided by 2 we divided by 3. So, we keep on dividing it till we reach n minus 1. If any one of the remainders of these divisions is zero, this means that it has a factor, and which is not either one or n itself, as a result a number is not prime.

So, let us start with the first index which is two. So, see I am setting ebx equal to 2. So, now, we the original number, which we need to test if it is prime or not, we are transferring that to ecx, the ecx will contain the original number and then we start a loop. So, what we do is that since the original number is in eax now. So, this is required for correct divisions, recall that the idiv instruction, actually needs a 64 bit number in edx

and eax. So, what we do is, since the original number is in eax also. We can go ahead with a division, but we need to set edx. So, the number is positive, the value of edx will be 0, because we are just extending the sign of a positive numbers, and the sign bit of a positive number is 0. So, it moves zero to edx, this is needed for division, and then we go ahead with the divide, and we divided with the value stored in ebx which is 2, for the first iteration.

Subsequently, we take a look at the remainder which is stored in edx. So, we compare this with 0. If it is 0 which means j e, we jump to dot not prime. So, not prime the answer is very simple, we move 0 to eax, which means the number is not prime and we exit. Otherwise we increment ebx; so from to the significant 3 and 4 and 5 and so on. Then the value of eax has also changed, because the core the idiv instruction puts the quotient in eax. So, recall that, the quotient is being put in eax and the remainder is being put in edx.

So, what we do is, since the value of eax change we put back though, set the original value of eax again we are transferring the contents of ecx back to eax. So, eax gets restored. Then we compare ebx and eax, which is essentially the index, and the original number. As long as the index that we are dividing the original number by the divisor is less than the original number j l, we jump back over here, jump if less than. So, we jumped back over here.

Otherwise we reach the end of the loop. At this point we can compute that since we did not find any factors the number is prime they put one there, and we jump to exit which means we move out of the program. So, this program is simple, the program is not very difficult, in terms of assembly statements how many do we have; 1 2 3 4 5 6 7 8 9 10 11 12 13. So, we have thirteen assembly statements, it is not that difficult, and now after have explained to it, it must be appearing slightly more straightforward, but the question is that can students were listening to this lecture.

Can they write such a program from scratch right. Given any problem can they write such a program, and that is unfortunately not going to come by listening to such lectures or reading books. The only way that it is going to actually happen is by practicing as much as possible, because all of these things are pretty much a function of practice. So, what listeners need to do, is that they need to write hundreds of assembly programs; such

that they can write fairly good bug free code, almost all the time, in a very repeatable reproducible fashion.

(Refer Slide Time: 61:22)

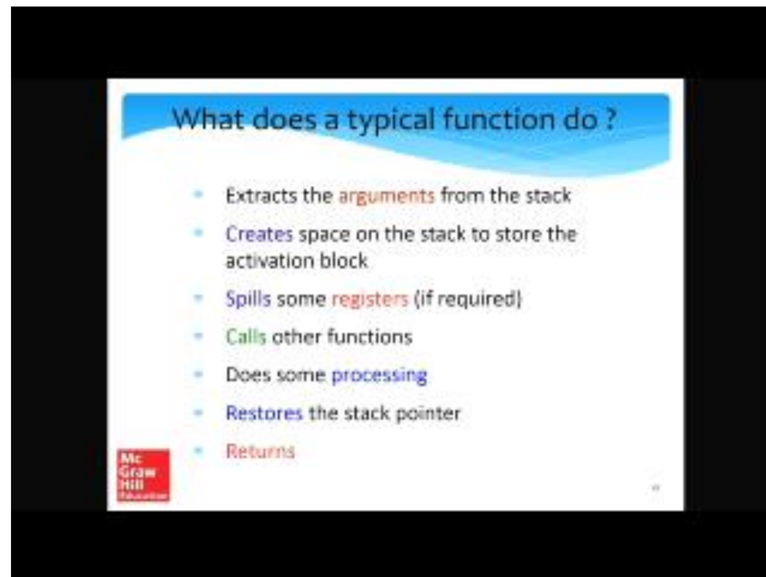
So, let us move ahead. So, what is left? What is left is function call and return instructions. So, the call and return instructions are exactly the same as simple RISC with some differences. So, the differences are like this. So, lets us find out what is similar. So, what is similar is. So, we have a call instruction, we always call the format is call any labels. So, let us say we go to label dot foo, dot foo is the beginning of the function right. And so recall that what was simple RISC and ARM doing, what they were doing, say call instruction is just not a jump; something needs to be done with the return address.

So, what simple RISC an ARM we are doing, is that the return address was. So, basically call has something to do with the return address. So, return address in those i s c s was being put in the return address registers, x 86 does not do that, it is rather stack way. So, it pushes the return address on the stack. So, it knows the value, it knows which register contains the stack pointer, see it pushes the value of the return address that is on the stack, and then it jumps to the label dot foo.

Similarly, we have is view addresses format instruct instruction called ret. So, here we return to the address; that is saved on the top of the stack and we pop the entry. There is completely a stack based system, so assume that this is a stack, and then we have a call. So, the return address gets saved on the top of the stack, and then we keep moving.

Finally, what we need to do is we need to kill the stack; the return instruction will take the value of the return address and pop the stacks.

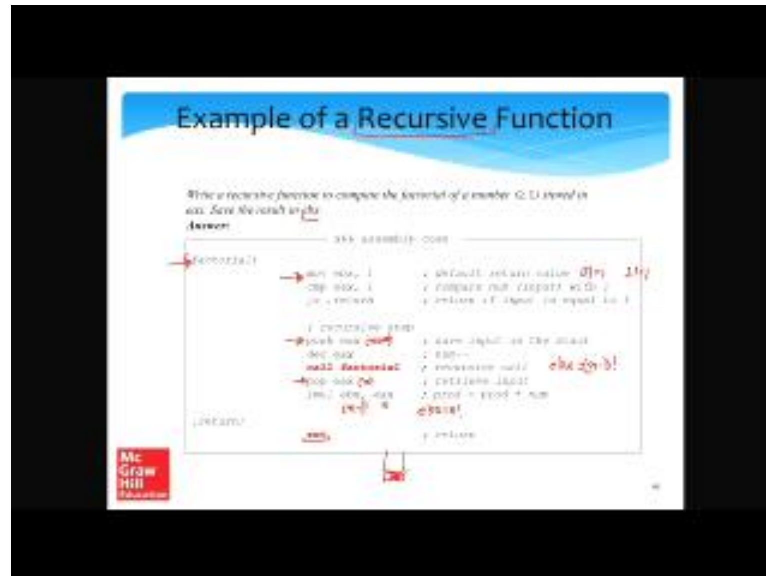
(Refer Slide Time: 63:15)



So, what does a typical function in x 86 do. So, well it needs to do similar things. So, we recall that we have discussed everything about functions, in a chapter three. So, the idea is that first before calling a function, we need to put all the arguments and so on, on the stack or in registers. So, once the function begins, it extracts arguments from the stack it creates space on the stack to store it is own activation block. It spills registers if required. So, there these are the issue of call is saved, and call is saved comes into play. It calls other functions does whatever processing it, it does it is own logic, it implements it is own logic.

Finally, after doing that it puts the return values, either you know in designated places of memory or in registers, it restores the stack pointed and it returns.

(Refer Slide Time: 64:19)



So, now let us take a look at an example. So, this is the example. So, the idea here is to write a recursive function. The users might wonder that why do we always give recursive function to the examples. Well our aim of giving an example, is to give the most difficult examples. So, it is not a difficult example is hopefully stimulate the reader to go into depth. So, the idea here is, write a recursive function, to compute the factorial of a number which is greater than equal to 1, and the number is stored in eax and save the result in ebx. Essentially compute eax factorial, save it in ebx, and the function has to be recursive. What does the recursive function mean; it basically means that the function is calling itself.

So, let us start. So, the name of the function is factorials, we will start when whenever we jump, we will essentially jump to this point. So, here we put 1 in ebx. So, this is the default return value. So, recall that zero factorial is one. So, maybe I can write it over here zero factorial is, what is this? This is equal to 1, 1 factorial is also equal to 1 right. So, basically the default return value we put it in ebx, and then we compare eax is 1. So, j is z, it can be j z here or j e does not matter at all. So, if there is equality we return. So, we return if the input is equal to 1, so which means that the answer is 1. So, we can just return and answer will be there ebx; otherwise we go to the recursive step.

So, in the recursive step we push the value of eax which is the value of the number n right, that we got. So, we are actually calling it num over here; so the number num on the

stack. We decrement eax, and then we call the factorial function. After calling the factorial function what is the expectation. The expectation is that. So, the reason this is made in red is, because this is these are the important instructions. So, the expectation is that ebx will contain n minus 1 factorial.

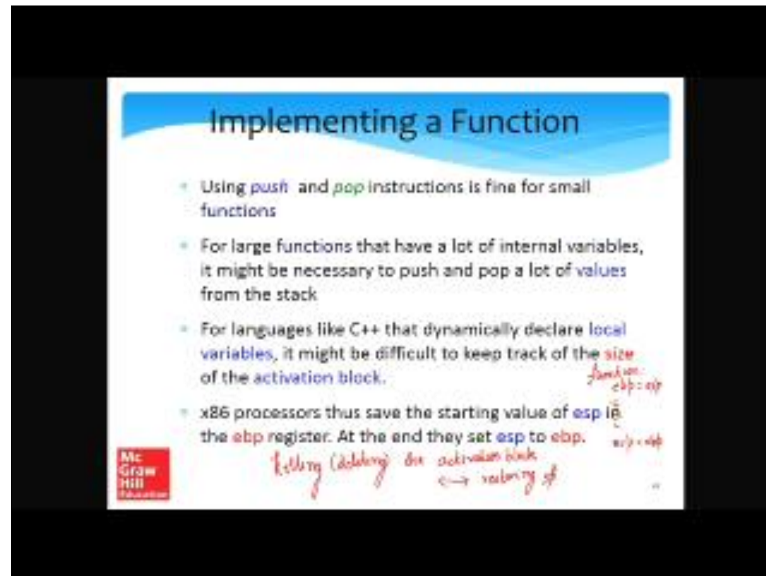
So, what do we do now we pop x. So, bring it read it back from the stack, and we multiply. So, what is eax contain at this point, eax contains the number n, and what is the ebx contain ebx contains n minus 1 factorial times n. So, we multiply n minus factorial times n. So, ebx will now contain n factorial right, a regular multiply instruction, and then we return. So, this is you know as simple as it is a very simple function, where we have a single push in a single pop, and what is the reason for actually pushing eax, because we are changing it is value and we are sending it to other function, and we need the value at the end. So, you can think of this as. Well it is can be that the sense of a 1 value, is being used does not matter if it is called, it call is save it or call is saved, but essentially the value of eax is required, and the value is being changed. So, before calling the function, we essentially store it on the stack and later on restore it.

The reason the read function works is, because we are otherwise every push has an accompanying pop instruction. So, the size of the stack is otherwise not being changed. So, when we do a read instruction, when you call the read instruction, it will find the return address at the stack top, which is this point, it will pop the return address and go back to it is original parent function. So, this is a simple program, it has 3 4 5 6 7 8 and 9 instructions to compute the factorial.

Even if I were to write it in c it would still take three four lines, depending upon how we count. So, x 86 assembly in that sense, is not all that inefficient, it has many instructions, so a lot of cool things can be done. It also has conditional instructions similar to ARM, for example, in conditional add or so on. So, they are called c moves instructions. So, I am I am not discussing them. So, you can see the x 86 manual, that has hundreds of instructions and a lot of them have very cool features, but I am only discussing the main instructions.

So, here what is the trick, well there are hardly any important tricks, the only trick is we are using the imul instruction, and we are pushing and you know saving and restoring the value of eax via the stack, those are the only take home points from this algorithm.

(Refer Slide Time: 70:01)

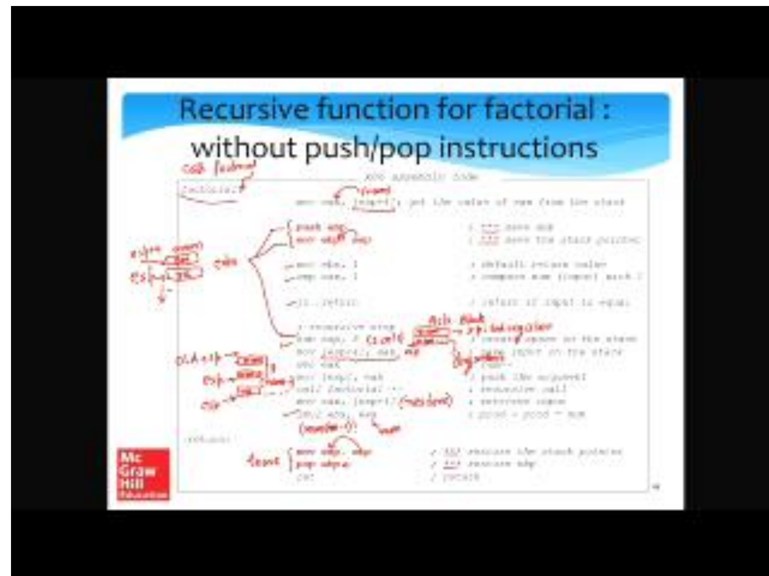


Say implementing a function what do we do, we use push and pop instructions in small functions like the one that we showed, which is fine, because it worked. So, if you see in the last one, we used a push instruction and a pop. For large functions that having a lots of internal variables and arguments, it might be necessary to push and pop a lot of values from the stack; fair enough. For languages such as C ++ that dynamically declare local variables. So, what you have is in languages like C ++. A lot of local variables can be declared even within the body of the function. So, it sometimes becomes difficult to keep track of the size of the activation block, because even at compile time you would not know what is the size of the activation block right, how many internal variables of the function actually use. So, at runtime the size of the stack needs to be varied.

So, x 86 processors have a mechanism of doing this. So, what I do is, that we, this is where we use the ebp register. So, we save the starting value of esp. So, whenever a function, whenever we enter a function the value of the stack pointer is called the starting value. So, we save the starting value of esp in the ebp register, this is the stack pointer is stored in a base pointer register. So, let me maybe this might require a little bit of explanation. So, whenever we start a function, the first thing that we do, is that we set the ebp equal to the esp, and then we keep on growing the stack, that a lot of instructions. at the end to restore the stack pointer back all that we do is esp is equal to ebp.

So, that keeps our life very simple, because note that in x 86 not. In fact, in any other assembly language, before leaving a function we need to kill the activation block. Killing the activation block is same as restoring the stack pointer. So, let me write it killing, it should not be killing, it should actually be deleting, but that is ok, we will be deleting, is a signal restoring the stack pointer.

(Refer Slide Time: 72:58)



So, let us write recursive functions for factorial without pushing pop instructions right. So, let us not use any push and pop instructions, and let us write a recursive function for factorial. All right, so let us do one thing, let us assume that whoever calls the factorial function, it puts the value of eax which contains the number whose factorial needs to be computed, on the stack.

So, so whatever is the stack pointer, on the stack pointer we do that. So, the assumption in x 86 is like this, slightly complicated if this is the stack pointer. So, at the beginning of a function if we read what is there inside the stack pointer, you only get the return address because this is; what is the last thing that is done before calling a function. The last thing that is done is that you will have an instruction of the form call factorial, and immediately the control will jump to the factorial function, and that is when the return address will be pushed on the stack, and a stack pointer will point to it. Just above the return address which means, because the stack is downward growing. So, pretty much esp plus four, which is this address. We have the opportunity of pushing in arguments.

So, we push in the value of eax, we can the moment we enter the factorial function, which is at this point, we can read s b plus for the stack pointer plus four. It is contents would contain the value of eax, which is essentially the value of the number, and this we can transfer to the registers eax. So, let me repeat, consider the function that is calling the factorial function, what is the last thing that it will do before calling. It will invoke the call instruction. What is the call instruction do? The call instruction decrement the stack pointer and pushes the value of the return address on the stack.

So, before that if we need to access the stack of the caller function, it is possible for the caller to push the value of eax on the stack, which is push the value of any register or any memory address, but in this case we are only concerned with the eax. So, push the value of eax, which is a number num on the stack, and then invoke the call instruction. So, when we reach here esp will point to the return address, but esp plus 4, would point to a location within the callers activation block, and there we can read the value of num and store it in eax. Then what we can do recall the star star star which are the new instructions, we can push ebp. So, we can store the value of the base pointer that was there, and transfer the value of the stack pointer into the base pointer.

Well subsequent instructions are the same, setting the default return value comparing the number with one, jumping if there is equality, this is all the same. Now let us come to the recursive step, in the recursive step what I do is I create some space on the stack. So, you subtract eight from esp, which gives me space to put in several things between two integers. So, at esp plus 4, what I do is, I put in the value of eax which is a number, and I decrement eax, and at esp which is essentially. So, let us me consider, this is the stack pointer what I do. So, we will assume that this is the old value of the stack pointer, which is this instruction prior to this instruction, this is pointing over here. So, we are subtract it by 8, and creating positions, I am creating actually two spaces, two slots to save two integers right, if I am subtracting it by 8, and a new stack pointer will point over here.

So, what I can do is at esp plus 4 which is this location. Let me save the value of the current numbers right, and then at esp which is this location. Let me decrement the number by 1 and save num minus 1 right, it might not be visible. So, I will just write once again. Then I call the factorial function, what that would do is, that would do exactly the same go to 8 esp plus four which is this location sorry. So, basically what that would do is, we will call the factorial functions. So, when you are calling the factorial

function what will the stack look like at this point when you are doing the call instruction? Additionally the return address will be pushed, and the esp the stack pointer will point over here. Inside the called function we will add four to esp and read num minus 1 and compute it is factorial.

So, after calling factorial what do we need to do, we need to restore the value of eax. So, just take this instruction, and flip the arguments. So, s b plus 4 is restore, retrieving the input. Finally, we do the multiplication prod equal prod times num then we have these two extra lines here. So, we restore the stack pointer. So, basically whatever was there an ebp we put it into esp, and we restore the value of ebp by popping it, and we return. So, in a nutshell what did we achieve this program, is exactly the same as this program right, but here we are the only difference is, we are not using push and pop instructions, instead what we are doing is that very explicitly managing the stack. So, in terms of hardware performance, it will not be any different, because push and pop instructions actually encapsulate a lot of functionality. So, what we do is, that we first. So, the main magic lies here, is that we create an activation block, with space to store two integers. Let me call it the activation block, activation block, it is space to store two integers; one of them is the esp plus 4. So, new stack pointer would point over here esp.

One of them is used for register spilling. So, we are spilling the value of eax which essentially contains the original number. The other one is being used to pass an argument to the next function that will be called. So, this has num minus 1 right, only two things right. In the activation block this is a spilled register, and this is an argument for the next function that we are calling, then we call the next function it works in exactly the same way, then we restore eax, and we multiply. So, ebx at this point is supposed to contain n minus 1 factorial eax. So, n sorry num minus 1 minus take, num minus 1 factorial right and eax is supposed to contain num. We multiply and then we get num factorial.

Subsequently the, since we change the stack pointer whatever was the original value saved in ebp we move it to esp, and we restore the value of the base pointer and we return. So, what are the extra instructions, extra instructions are used to right, these two over here and the instructions to create the activation block and manage it, which is subtract eight from esp and mov eax to esp plus 4 mov eax to esp, and restore the value of eax right, couple of extra instructions are there.

(Refer Slide Time: 82:24)

Enter and Leave Instructions

Semantics	Example	Explanation
enter imm, 0	enter 32, 0	push ebp (push the value of ebp on the stack); mov ebp, esp (save the stack pointer in ebp); esp ← esp - 32
leave	leave	mov esp, ebp (restore the value of esp); pop ebp (restore the value of ebp)

- push ebp ; mov ebp, esp ; sub esp, <stack size> is a standard sequence of operations
  - The enter instruction does all the three operations
- mov esp, ebp ; pop ebp
  - Standard sequence at the end of a function
  - Both the operations are done by the leave instruction

Mc Graw Hill Education

So, do we need that many extra instructions, if you see we have seven instructions that we are changing. Do we need them, the answer is probably no say x 86 is great in providing shortcuts. So, the complexity that we saw in the last example, that will not happen that is not required, we have an entire instruction. So, what the entire instruction does is that it does a lot of things in one instruction. It pushes the value of ebp onto the stack, the value of the base pointer on the stack, it mov esp to ebp, so saves the value of stack pointer in the base pointer, and it subtracts from esp whatever argument has been specified here.

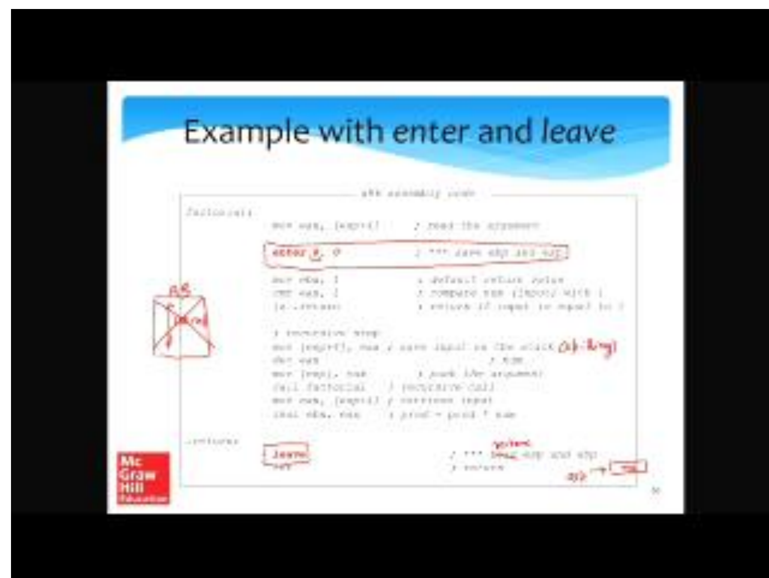
So, we are here we are specified 32 bytes, certain subtract 32 from esp. So, is it doing it is doing a lot of things. First it is doing the job of pushing ebp, putting esp to ebp, and, the job of this instruction, this instruction, and this instruction are all being done by one instruction, which is essentially storing the value of ebp on the stacks, are setting ebp to the stack pointer, and creating the activation block by deleting the size, sorry by subtracting the size of act activation block from the stack pointer. So, this is like a three in one kind of instruction. So, basically this is what it does, push ebp, mov esp ebp and subtract the stack size from esp. It is a standard sequence of operations we instead of having three instructions for it, the entire instruction I will do actually all three.

Similarly, we have the leave instruction which will do the reverse. So, it will move the value of the base pointer into the stack pointer, or in other words restore the value of the

stack pointers, it will pop ebp, which means restore the value of the base pointer, which is also a standard sequence at the end of a function right. So, what does the leave do, the leave instruction does the job of these two instructions over here; restore the stack pointer and the base pointers, and the enter instruction does the job of actually storing the base pointer setting the base pointer equal to the stack pointer, and creating the activation block.

So, pretty much these three instructions are folded into one instruction called enter, and the only argument. Well the two arguments. The second argument is slightly more complicated, but for all practical purposes we always set it to zero; the first argument is the size of the activation block, which in this example is 32.

(Refer Slide Time: 85:15)



So, if we were to make write this program, the previous program what do we have to do. All that we have to we need to add only two instructions, and remove all the other instructions; one instruction would be this, one more instruction would be this. So, let me go back to the complicated figure that we have complicated by writing a lot of things. Essentially these three instructions get fused into one, and these two get fused into one. So, we can call the entire instruction, which saves the base pointer and the stack pointer, and also creates an activation block of size 8 bytes.

The rest of the code is the same, then we save the input on the stack, right saving the input on the stack is essentially same as spilling right. So, we know that the calling will

change it. So, we are pretty much spilling it, and then decrementing eax and pushing the argument calling the function, all of this is fine, and at the end we call the leave instruction through the only job is to load or the instead of load maybe the word restore would be better. So, restore esp and ebp. So now, you see that push and pop are not always required. Well they are good ideas if the program is small and simple, and recalls that pushes and pops have to be exactly matched. So, if we have n pushes, we need to have n pops right; otherwise the stack will not remain the same, and a return instruction will not work, and keeping track of the number of pushes and pops in a complex function is hard; that is the reason what you do is, that we you save the starting state at the beginning with the entire instruction, then you do whatever you want to do with the stack, and just before leaving called the leave instruction, which will restore the value of the stack pointer and the base pointers, and then you return.

So, why is this needed the reason this is needed, is basically because we want to completely delete the activation block, and when you return the value of the stack pointer should be pointing to the return address, because that is where we will read the return address from all right. So, after taking a look at enter and leaves. So, let me actually come back for two seconds, with enter and leave and then I will again proceed further.

So, in complicated programs, where a lot of variables are allocated space on the activation block, and activation block per say, you know is a large structure. So, let me call it the a b, and it can have like hundred integers or maybe in the more right. So, depending upon the paths that the program takes the size of the activation block can vary. So, instead of monitoring it in very fine grained manners, what is the ultimate thing that we need to ensure that, when the function returns this entire structure is killed. This is exactly what the entire instruction and a leave instruction allow you to do.

So, using them, using the, enter as one of the first instructions and a function, and leave as one of the last instructions, like the instruction just before read, is actually a good idea and good x 86 programming practice. Now, we will discuss advanced memory instructions.