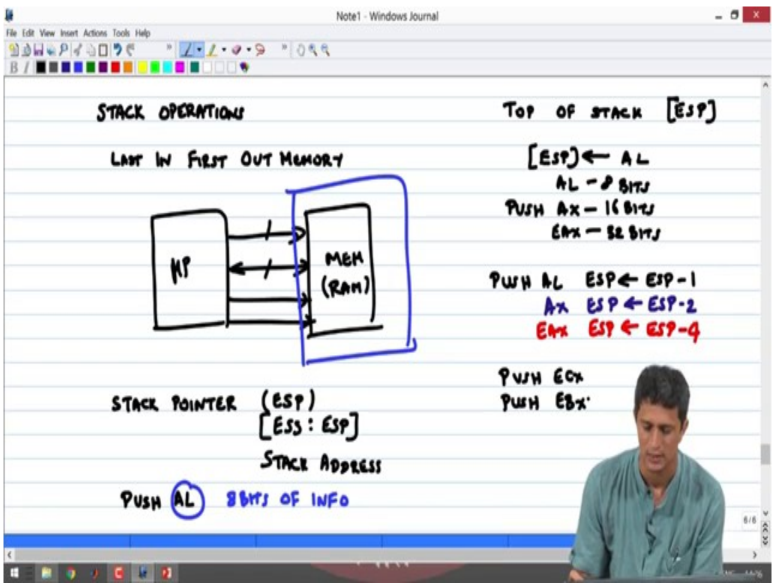
C Programming and Assembly language

Prof. Janakiraman Viraraghavan
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 05

So, welcome back to this course on C Programming and Assembly language. In this lecture we will discuss one of the most important operations that are exploited in a microprocessor both from the assembly language point of view and a C programming point of view and these are the stack related instructions.

(Refer Slide Time: 00:30)



So, the stack as we all know is as the name suggest impact is just a pile of books and instead of accessing a random book somewhere in between you always pick the book that is available on top. So, which means that the book that goes last is actually accessible first; so, this is a last in first out memory. So, let me remind you that in our abstraction of the memory, we said that there is an address which is some n bits and depending on the big combination 1 of the 2 power n locations will be accessed and that many bits of data will be made available on the data bus.

This is the abstraction of the memory that we discussed and really that does not change even when we are dealing with a stack. It is only how the microprocessor actually deals with the memory that enables or that differentiate if it is a random access memory or a stack memory. So, in our picture we had a memory which was random access and a mu P which communicated through a data bus an address bus a data bus and so on, read and write

And really none of this part actually changes when we are talking about a stack. So, what is it that enables microprocessor in order to treat a random access memory as a stack. So, this is enabled particularly by a register called the stack pointer and this we have alluded to it earlier is called ESP Extended Stack Pointer or the just the stack pointer .

And again what we do here is the memory is divided into various segments, the ESP is only the offset that is specified the base of that address comes from the stack segment ESS stack Extended Stack Segment register colon ESP is the stack address. So, what does this memory do? If you want to access a certain data or put something on to a stack all you have to do is exercise this instruction called a PUSH.

And you can specify let us say a register AL. What is AL? It is 8 bits; 8 bits of that has to be pushed on to the memory or the stack segment. Note that here I am not specifying any address of where this data has to go as opposed to a MOV instruction where I would have said MOV AL to maybe some contents pointed to by bx. So, I am explicitly specifying where this data has to go and sit in memory I am not doing that in the case of a PUSH instruction. So, what happens is the PUSH instruction simply looks at this top of stack (top of ESP) .

And, I am not going to mention always that it is a stack segment register which completes the address, we will only talk about the ESP as the address . So, the contents pointed to by ESP is going to get loaded with the contents of the AL register. So, what happens here? Contents pointed to by ESP will get loaded with AL register, similarly I could also PUSH AX or EAX. So, now, the difference between PUSH AX or PUSH EAX , so PUSH AL.

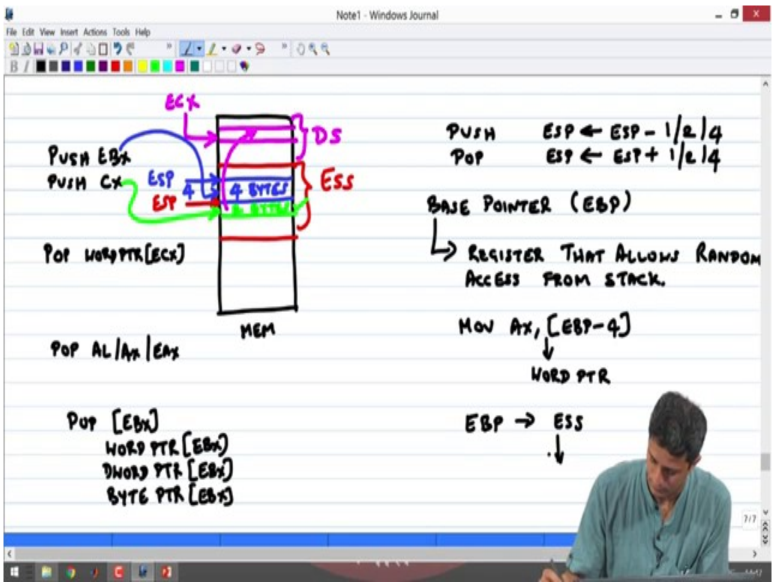
What is the difference? This is 8 bits, this is 16 bits and this is 32 bits. So now, let us go back to our logical memory map that we discussed in the initial lectures where we said that data is always accessed in chunks of 8 bits a particular address can store 8 bits, the next location will store the next 8 bits.

So, if I PUSH AL onto the stack, then my stack pointer has to be decremented by 1. So, after performing the PUSH AL my stack pointer has to get decremented by 1. WhyBecause I am now dealing with only 8 bits of information and the logical memory has to change only by 1.

So, this is how the location of the stack is eventually decided every time you PUSH something on the stack the stack pointer gets decremented by 1, 2 or 4. So, if you instead AX to stack then the ESP would get decremented by because this is now 2 bytes of information and if you PUSH EAX on to stack ESP would get decremented by ESP minus 4 because there are 4 bytes in the EAX register.

So, if you now look at a combination of two such instructions PUSH ECX, PUSH EBX. So, what would happen is the contents of ECX are actually loaded first and the contents of EBX are loaded next.

(Refer Slide Time: 08:15)


Note1 - Windows Journal

PUSH EBX
PUSH CX
POP WORDPTR[ECX]
POP AL/AX/EAX
POP [EBX]
WORD PTR [EBX]
DWORD PTR [EBX]
BYTE PTR [EBX]

PUSH ESP ← ESP - 1/2/4
POP ESP ← ESP + 1/2/4

BASE POINTER (EBP)
↳ REGISTER THAT ALLOWS RANDOM ACCESS FROM STACK.

MOV AX, [EBP-4]
↓
WORD PTR

EBP → ESS

So, if you look at the particular stack segment here this is my entire memory and let us assume that the stack segment starts here and finishes here. So, my ESS is going to determine how much of memory is available in the stack segment or at least when it starts.

So, if I now execute the instruction PUSH EBX, then 4 bytes of data of EBX will go and sit in 4 locations pointed to by the stack pointer. So, let us assume that my stack pointer is pointing somewhere here in memory ESP. So, when I PUSH EBX 4 locations will get return; will get return with the contents of EBX and in the process my stack pointer would have now been decremented by 4 and it goes to a new location as pointed to by as shown here in red.

Now, when I execute the next instruction which is PUSH ECX right or may be let us consider you know at different size here let us just PUSH CX 16 bits of information. Then what happens is you are going to decrement the stack pointer only by 2 now this distance is 4. Now, I am going to decrement my stack pointer only by 2 and CX will now come and sit here and this is 2 bytes let us say I want to access the data from the memory from the stack and bring it into a register I want to do the opposite of a push. So, that is called a POP operation right.

So, I could POP into AL right or I could POP into BL or I could POP into BX or I could POP into EBX. So, I could do the exact same operations AL, AX or EAX. So, remember here I can also PUSH or POP data from an other location right using a different addressing mode. For example, I could do POP of contents pointed to by EBX, where EBX is now a register which is going to point to an address and which segment does this point to EBX always is associate associated with the data segment register by default.

So, therefore, if this happens to be my data segment data segment, then EBX would be pointing to some location here and if I say POP into contents pointed to by EBX, then whatever my stack pointer is from that location a certain number of bytes would have to be popped into that location pointed to by EBX . Therefore, it is now important for us to specify how many such bytes we want to POP from the memory remember when we said PUSH AX or EAX, it was implicit from the op code or the mnemonic that we were pushing 16 bits or 32 bits or 8 bits.

Now, when we are popping from the memory this information is not obvious and needs to be specified explicitly and therefore, we have discussed this earlier we need to specify if this is a word pointer or is it a DWORD pointer or is it a BYTE, the default is a byte pointer. So, let us look at an example for this if after this PUSH EBX and, PUSH CX instruction we did a POP I just write the instruction lower here POP WORD pointer contents pointed to by ECX .

Then what would happen is the top of the stack is pointing to this green location here this location. So, 2 bytes from this location will be popped and return into the location that is pointed to by ECX. So, ECX for example, may actually be pointing to this location in memory we have majenta location here. So, the 2 bytes may go and just get return into this location that is pointed to assuming that ECX is pointing to this particular location..

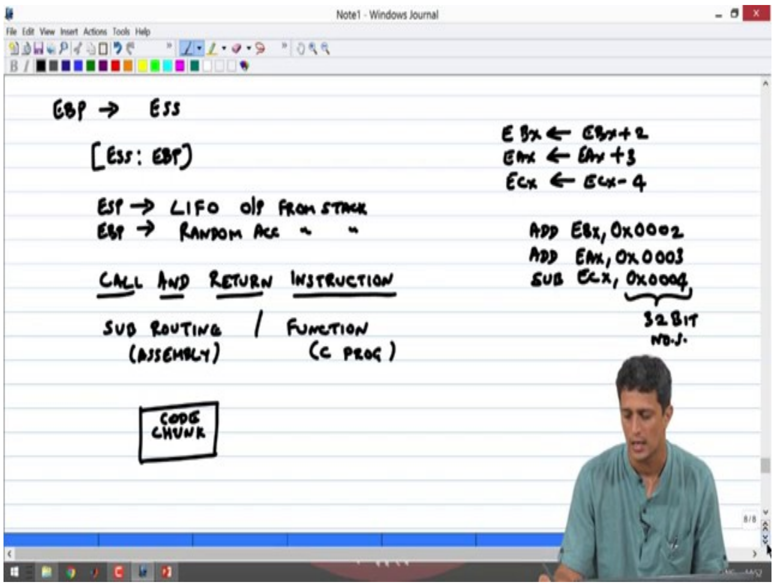
So, if instead of doing this POP word pointer I did POP DWORD pointer, then it would load 4 bytes from the top of stack into that location pointed to by ECX. So, 4 bytes there would get return. So, it is important to note that on implementing the PUSH the stack pointer is auto decremented . So, when we do a PUSH ESP is ESP minus 1, 2 or 4 which is determined by whether we want to PUSH a 8 byte data; 8 bit data, 16 bit data or a 32 bit data on to stack.

So, when we do a POP the ESP will now be the exact opposite will happen will get added with 1, 2 or 4 which is determined by whether we actually say WORD or DWORD or a byte. So, in this operation of PUSH and POP there is really no particular address that is that has to be explicitly mentioned, it is implicitly available from the top of the stack pointer which is given by the extended stack pointer register. Data from there is either accessed out or return to and the stack pointer register is altered by that amount that we are actually writing or reading from the memory.

So, this is the first, last in first out operation of the stack; however, it turns out that this is not sufficient for us we also need to be able to do random access even from the stack and therefore, we have another register which is known as the base pointer EBP. What does this do? It is nothing, but a register that allows random access from stack. So, for example, I could do MOV AX comma contents pointed to by EBP minus 4 .

So, you remember this register in directed indirect addressing with offset . So, that is what we are doing here we are calling the MOV instruction, we are going to load the data pointed to by EBP minus 4 and we have of course, I have to mention WORD pointer here right. So, data 16 bits of data pointed to EBP by minus 4 will get loaded into my AX register on executing this particular instruction . Again the EBP is associated with the extended stack segment register .

(Refer Slide Time: 18:52)


EBP → ESS

[ESS : EBP)

ESP → LIFO O/P FROM STACK
EBP → RANDOM ACC " "

CALL AND RETURN INSTRUCTION

SUB ROUTING / FUNCTION
(ASSEMBLY) (C PROG)

$\boxed{\text{CODE CHUNK}}$

EBX ← EBX + 2
EAX ← EAX + 3
ECX ← ECX - 4

ADD EBX, 0x0002
ADD EAX, 0x0003
SUB ECX, $\underbrace{\text{0x0004}}_{\text{32 BIT NOS.}}$

So, this is E is associated with the ESS which means that the address is always given by ESS colon EBP . So, in summary ESP enables last in first out operation from stack, EBP allows random access from stack. So, with that we move into another class of instructions and probably the most important class of instructions which are again a branching set of instructions, but slightly different from what we studied a little earlier.

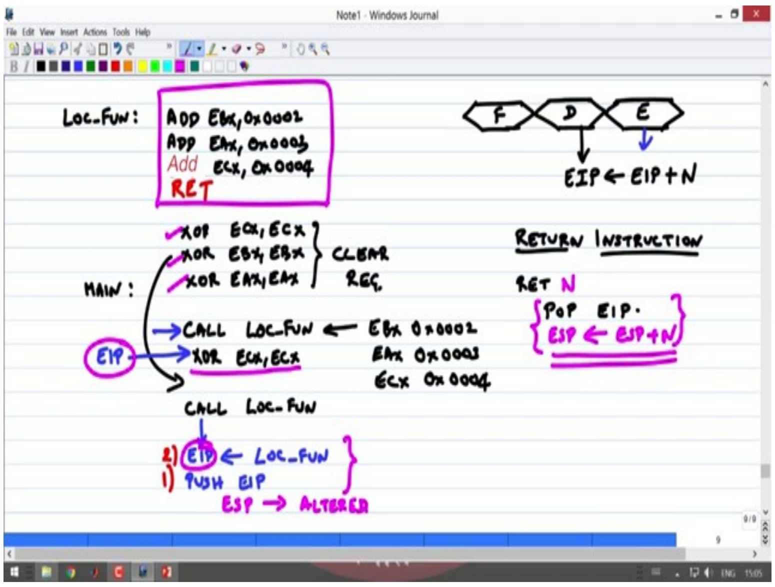
So, that takes us to the call and return instruction. So, the idea here is that if you want to do a particular operation repeatedly, then you do not have to re rewrite the same code again and again and again . So, you can actually invoke what is known as a subroutine in assembly language or a function in C programming . So, a good programming practice is if you find that you are reusing a particular segment of code again and again, then please do not copy paste that convert that to a function and call that function again and again .

So, we have the concept of a subroutine in assembly language or function in C programming So, the idea here is there is a particular chunk of code ; a CODE CHUNK which is going to be called again and again in order to perform a certain operation right. So, the simplest could be for example, I am just going to increment my BX register by 2.

Let us say that I want to do this operation again and again EBX should be EBX plus 2, EAX should simply be EAX plus 3 and ECX should be ECX minus 4. Let us just assume that this set of instructions has to be done again and again in a microprocessor . So, what do we do here? We actually if you wanted to write this instruction it would be the following , it would be add EBX comma 0x0002 .

Add EAX 0x0003 subtract ECX comma 0x0004 right. So, let us though I have written only 3 0s and its 16 bits let us assume that there are the you know its just z sign extended 0s are extended all the way to the 32 bits. So, these are all 32 bit numbers. So, let us assume that this sequence of instructions needs to be executed again and again in my program. So, what do? I load this into memory .

(Refer Slide Time: 23:50)



So, I have a location this is going to be some address in the code segment where I have these instructions add EBX comma 0x0002 add EAX, 0x0003 and subtract ECX comma 0x0004. So, this is sitting in my location function and my main code . So, these two labels LOC underscore FUN and main sitting before the colon are actually labels for my address in the code segment.

So, let us say I have some instructions and I want to be able to call this function multiple times in the process. So, I want do call of location underscore function then I execute some more instructions and I want to call it again maybe we can take a more concrete example here let us clear EAX comma EAX let us clear, EBX comma and let us clear ECX,ECX this is simply going to clear and then I am going to call the function LOC underscore FUN right as described above.

So, the idea is you clear it, you execute this function and when you want to come back and continue with the execution of these functions. So, let us may be fill in some more concrete instructions here where I just say I may be I will clear the instructions again the registers again , I am going to just put this code again here . So, what should happen in this sequence of instructions is you clear the registers call the function and when the function is invoked you are just going to increment EBX by 2, EAX by 3 and ECX by 4 at the end of this first call EBX should be 0x0002 .

EAX should be 0x0003 and ECX should be 0x0004. The same clearing happens again and again I should be able to call this function and the same result should repeat that is the intended idea behind this subroutine call. So, how is that implemented in assembly language? So, when you call a particular address location like LOC underscore function, the first thing that has to be done is the EIP has to be loaded with fun . So, this instruction should translate to this particular operation this is very clear and is no different from the unconditional or the conditional branch that we studied earlier. In fact, its just an unconditional branch that we are doing here.

So, what is the difference between a call and a unconditional branch that we studied earlier? The idea behind the call is it is a two way branching that is possible which means that you branch out into that subroutine call finish the execution and then you should be able to come back to where you left off. So, how does that gets executed in the microprocessor is the question. So, let us go back to our you know loop of instructions or loop of process that happens in a microprocessor right we said it does fetch decode and execute in a loop .

So, when it fetches the particular instruction and it decodes that instruction what happens is it actually knows where the next instruction is going to sit in memory. So, at this point, the EIP is automatically incremented to the next instruction . So, for example, if you do an add then you know that the add instruction takes maybe 16 bits of information in the code segment; that means, the next instruction is going to be 2 locations away. If another instruction needs 4 bits or 4 bytes of data in the code segment, then at decode time the microprocessor will know that the next instruction is going to be four locations away. Therefore N will be 4.

So, what happens is when you execute a particular instruction, the instruction pointer is auto incremented to point to the next instruction at that particular location. So, for example, when you are executing this particular call in the process of decoding that instruction after fetching, the instruction pointer has already been incremented to point to this instruction. So, when the call instruction is going to get executed , the fetch decode and execute process when the call instruction is going to get executed which happens here, the EPI has already being incremented to point to the next instruction or the location where we have to return after finishing the subroutine.

Therefore we implicitly have this information with us before we branch out. So, all we have to do is to put this information somewhere on in memory, so that we can just access it when we want to come back. So, it turns out that this return instruction data is pushed onto stack. So, when the call happens the other thing that implicitly happens is PUSH EIP on to stack. So, here remember this is another advantage of a stack that we discussed earlier, I do not have to worry about an exact location I just PUSH it on to stack and at an appropriate time I have to POP it out open out and so, that I can get the data of that instruction pointer back. So, when I do a call the instruction pointer is loaded with an new address that I have to go and execute my subroutine from and it also implicitly pushes my instruction pointer onto the stack.

So, which register actually changes in this process by the way just a correction, the order is quite the opposite. The PUSH happens first because you have to PUSH the next instruction location on to stack first and then load the instruction pointer with the new location that you want to go and execute a subroutine from. So, implicitly remember that the stack pointer ESP is altered and the EIP is also altered because you are loading a new address into that register. So, you call this function, it will now go into that location function after pushing the EIP onto stack, it executes that particular instruction set which is this three instructions shown here.

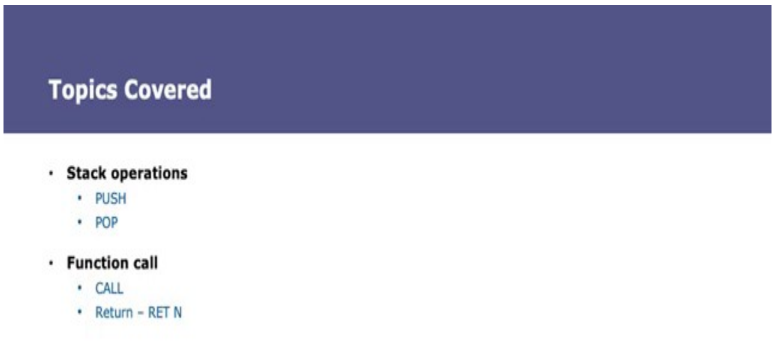
Now, I want to come back from this subroutine from where I left off that is enabled by what is known as the RET instruction R E T, it is a return instruction. What is the return instruction do RET . So, what does it do? It simply pops the top of stack data into the instruction pointer for it to continue from where it left off. So, it simply does the following it pops the top of stack into my instruction pointer. So, that it will now simply continue from where it left off.

So, in this particular sequence of instructions we clear the registers right we clear the registers one by one, then we do a call it goes to that particular location LOC underscore function executes the three instructions and when it does a ret remember that the EIP got pushed onto stack when we did the call. So, if you have done no other stack operation in between, then the top of stack will now contain the instruction pointer which is pointing to this particular location.

So, when you do a return which happens to just POP the instruction pointer into this the data of the top onto the stack into the instruction pointer; it will simply come and resume its execution from this particular instruction onwards. So, you can also do not just a plain return you can do what is known as a RENT N. So, what does the RENT N do? It not only will POP the top of stack into the instruction pointer it will also add ESP will get added plus N. It will just add this number N that is given to the stack pointer, I would like to reiterate that I am breaking down every assembly instruction here into multiple operations does not mean that each of these operations are executed as separate assembly instructions.

For example, the RET N is not broken down into these two assembly instructions. It is done in hardware and there is hardware support for it that is why it is that fast, if you do it in software it becomes very very slow , this is just a functional description of what happens all of these things that we discussed here or here happen in a single shot in hardware. So, we will come to why this particular addition is useful when we do a C program. So, with that we complete our discussion on the stack operations and the call and ret which are most critical for subroutine execution and returning in assembly language.

(Refer Slide Time: 36:46)



Topics Covered

- **Stack operations**
  - PUSH
  - POP
- **Function call**
  - CALL
  - Return – RET N