

# Intel X86 Processors

only 14 assembly instructions make up to 90 percent of the code!

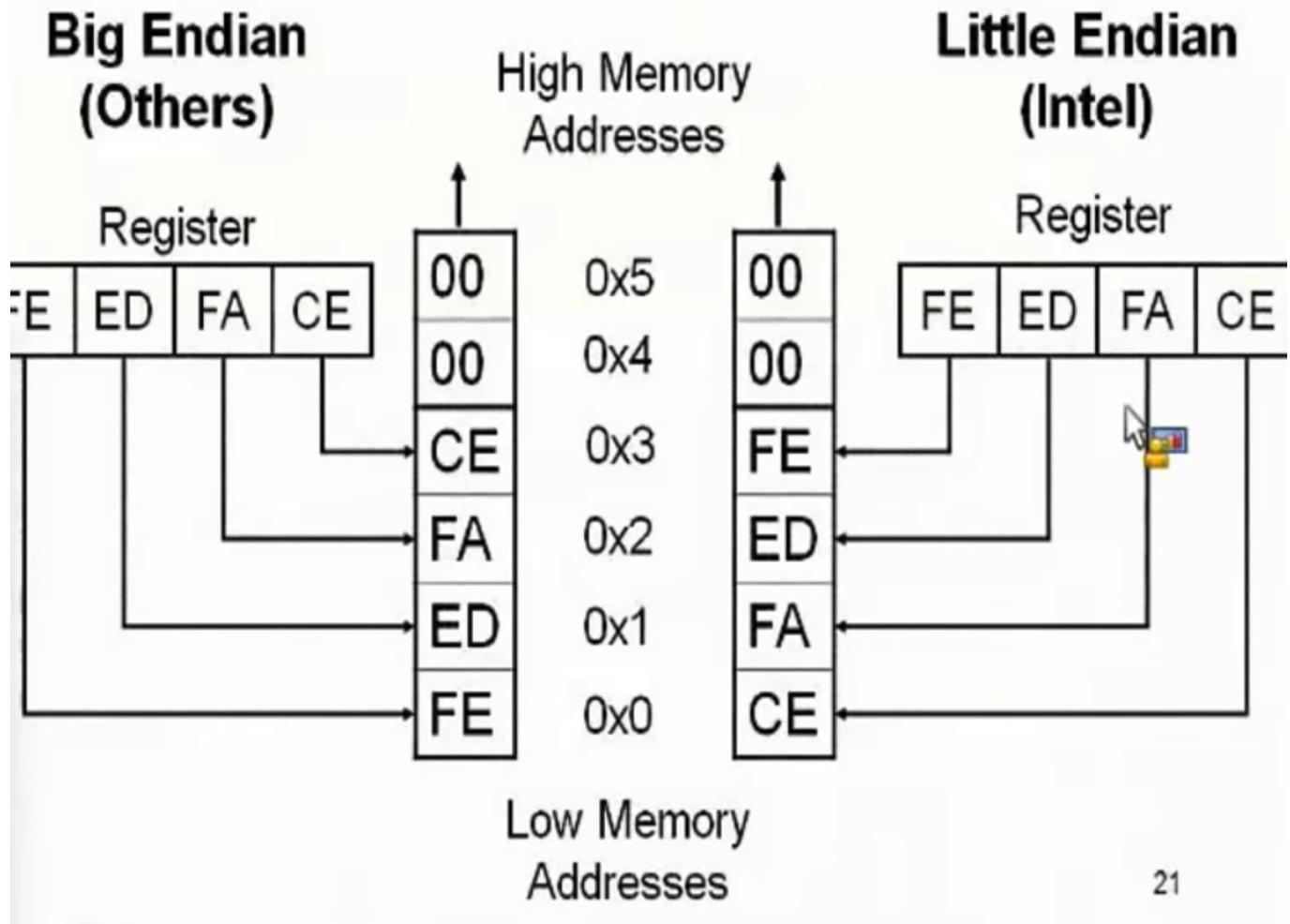
in intel terminology : a byte is a char(8) , a word is a short(16), int/long is double word(32), double long(64)

If you don't know this, you must memorize tonight

Decimal (base 10)	Binary (base 2)	Hex (base 16)
00	0000b	0x00
01	0001b	0x01
02	0010b	0x02
03	0011b	0x03
04	0100b	0x04
05	0101b	0x05
06	0110b	0x06
07	0111b	0x07
08	1000b	0x08
09	1001b	0x09
10	1010b	0x0A
11	1011b	0x0B
12	1100b	0x0C
13	1101b	0x0D
14	1110b	0x0E
15	1111b	0x0F

Intel is little endian. 0x12345678, least significant bit is registered first, then more significant and more significant.

# Endianess pictures



Endians are only meaningful in byte level not in bit level!

Register Conventions!

**EAX** ==> stores function return value **EBX** ==> Base pointer to the data section **ECX**==>counter for string and loop ops **EDX** ==> I/O pointer **ESI**==>source pointer for string operations **EDI**==>Destination pointer for string operations **ESP**==>stack pointer **EBP**==>Stack frame base pointer **EIP**==>pointer to next instruction to execute(instruction pointer)

There are also other type of registers.

**EAX, EDX , ECX** == > Caller-save registers

I am a function and I will call another function. And when I call I am sure I am going to destroy the current registers(eax edx ecx) so we save the copy of the registers before calling the function so we do not lose the register

**EBP, EBX, ESI, EDI** ==> Callee-save registers

If I call a function, it won't change and modify the info in the register of ebp, ebx etc.

this E at the head of registers mean Extended. Since these are written originally for 16 bit , AX became Extended AX with 32 bits. etc.

## EFLAGS

EFLAGS register holds many single bit flags. Will only ask you to remember the following for now :

Zero Flag(ZF)- Set if the result of some instruction is zero, cleared otherwise Sign Flah(SF) - Set equal to the most significant bit of the result.

## INSTRUCTIONS

**NOP** No Operation. No registers no values nothing. Just there to pad bytes or delay time. Bad guys use it to make simple exploits more reliable. But that's another class.

NOP actually takes a register and re-registers it on itself 😊

**PUSH** ==> push word, double word or quadword onto the STACK. can either be immediate(a numeric constant) or the value in a register.

For our purposes, it will always be a DWORD(for this course) the push instruction automatically decrements the stack pointer, esp, by 4. Why decrements? Because its stack, its FILO.

**POP** ==> Pop a value from the Stack

take a Dword off the stack, put it in a register and increment esp by 4 ( reverse of Push)

**CALL** ==> Call's job is to transfer control to a different function in a way that control can later be resumed where it left off.

First it pushes the address of the next instruction onto the stack. for use by RET(return from procedure) for when the procedure is done.

Because idea behind a function is , do this and I will keep executing from the next line. So call also pushes the address of the next instruction set, so that once the called function is completed, the program keeps continuing from where it left off. Its a reminder to where to return to after executing the function.

Then it changes eip to the address given in the instruction.

Destination address can be specified in a multiple ways.

- Absolute address ==> address 0x0030434
- Relative address(relative to the end of the instruction) ==> some address that hex 50 bytes ta ta ta.

**RET**==> Return from Procedure.

Two forms :

1. pop the top of the stack into the eip

in this form, the instruction is just written as ret.

typically used by cdecl functions

2. pop the top of the stack into eip and add a constant number of bytes to esp.

in this form, the instruction is written as ret 0x8 or 0x20 etc. typically used by stdcall functions

**MOV** ==> Move.

Can move:

register to register

memory to register, register to memory, immediate to register immediate to memory.

immediate is hardcoded value.

### NEVER MEMORY TO MEMORY

Memory addresses are given in r/m32 form.

**ADD** ==> a = a + b for example add eax,ebx means eax = eax + ebx. so it sums them and writes on what is on the left side

**SUB** ==> subtraction. sub eax, [ebx\*2]

**LEA** ==> Load Effective Address. ==> frequently used with pointer arithmetic sometimes for just arithmetic in general. uses the r/m32 form but **is the exception to the rule**

Lea makes the calculation and appends to the destination without disturbing the memory. Does the calc on the spot.

that the [] syntax means dereference("value at")

example: ebx = 0X2, edx = 0x1000 --- lea eax,[edx+ebx\*2] **eax = 0X1004, not the value at 0x1004!!**

## THE STACK

stack is the conceptual area of main memory which is designated by OS when program is started. Stack is LIFO/FIFO data structure where data is pushed on to top of the stack and popped off the top

Stack grows over lower memory addresses. Adding something to the stack means the top of the stack is now at a lower memory address.

ESP points to the top of the stack. The lowest address which is being used(since it is LIFO)

Stack keeps track of which functions were called before the current one. It holds LOCAL VARIABLES and is used to pass arguments to the next function to be called

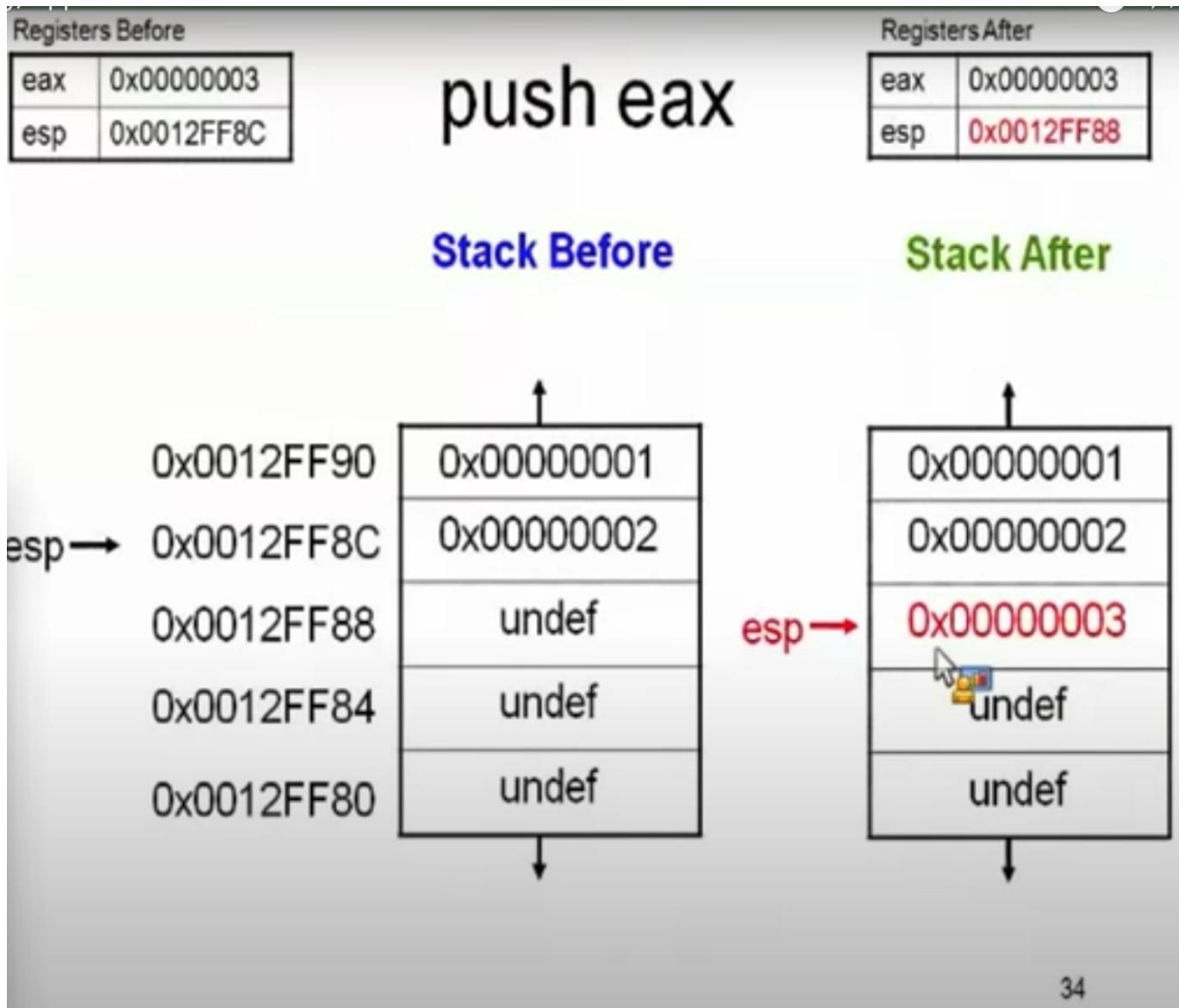
Register in time 1 :

eax : 0x00000003 esp : 0x0012FF8C

So when we call lets say, **push eax**, we expect the eax to be written 4 bytes after(decrement) the last esp register in the stack.

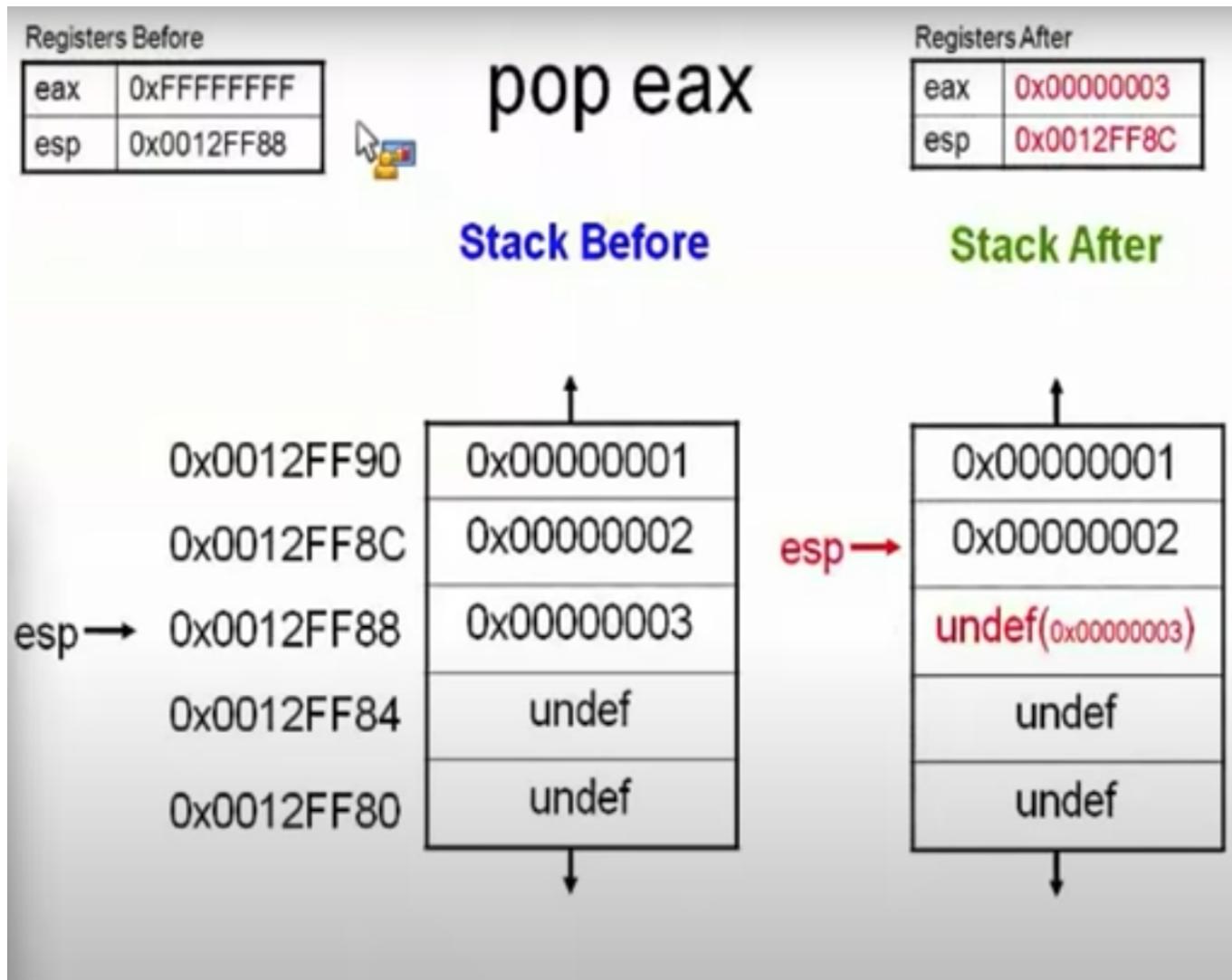
Register after push eax:

eax: 0x00000003 esp:0x0012FF88 ==> here, we jumped 4 bytes from FF8C to FF88. (where FF8C>FF88)



to reverse it , we use **POP**

in order to reverse the action above, we **pop eax** and voila, esp jumped by 4 bytes up ( incremented) and top shelf of the stack is emptied.



Everything numerically under `esp` is considered to be undefined. Although there is a data over there, it is not of our business.

once we call `pop eax`, the the value at the top of the stack (`eax`) is taken to the register and that memory spot is not undefined.

## Calling Convention

calling convention is about how to pass parameters and how to get parameters back.

how code calls a subroutine is compiler-dependent and configurable. But there are few conventions. We will deal with cdecl and stdcall conventions.

### cdecl

C Declaration. The most common calling convention. Function parameters pushed (using push) onto stack RIGHT TO LEFT. Saves the old stace frame pointer and sets up a new stack frame.

`eax` or `edx:eax` returns the result for primitive data types.

Most significant 32 bits go into `edx`, least significant bits go into `eax`

Also values are always big endian in registers, in memory they are little endian.

so for example : `printf(%d\n, myVariable)` ==> from right to left: push my variable to the stack then push the pointer and then call the function.

function here, saves the old frame pointer, saves the address to the stack.

**Caller** is responsible for cleaning up the stack. ==>

so we have 2 parameters and 1 function call. push push and call. so whatever register is calling the function is also responsible to clean up the stack

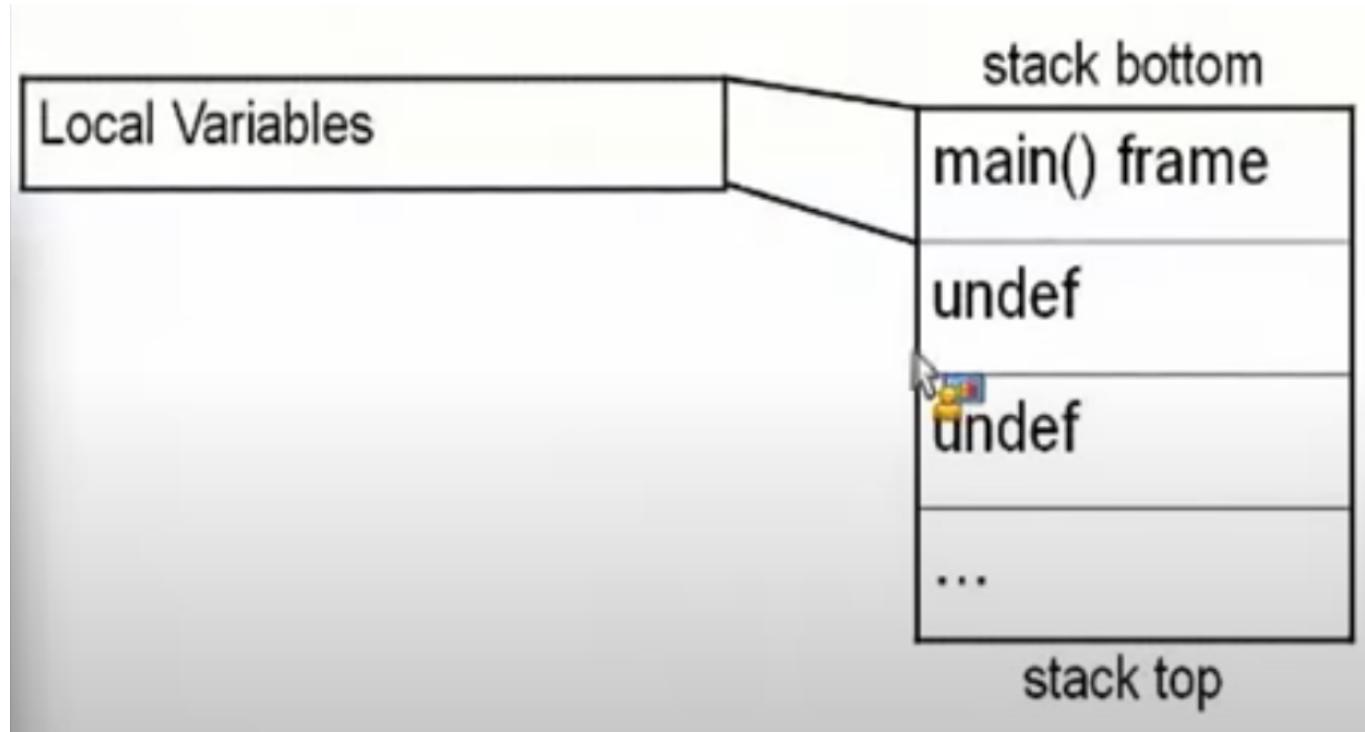
stdcall

Microsoft C++ code e.g. Win32API parameters pushed onto stack right to left saves the old stack frame pointer and sets up a new stack frame pointer

HERE, CALLEE is responsible for cleaning up any stack parameters it takes not the CALLER!

## General Stack Frame Operation

We are going to assume that the `main()` is the very first function being executed in a program. This is what its stack looks like to start with.



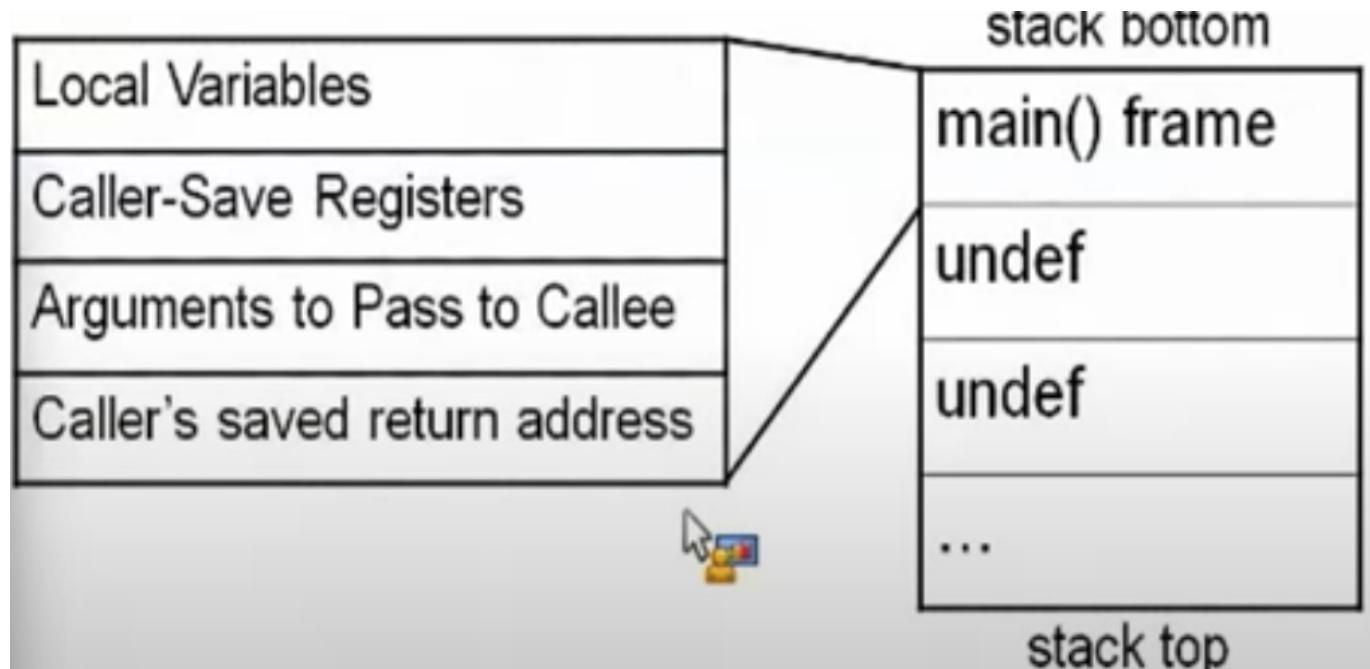
Low addresses at the bottom, high addresses at the top.

The more I add to the stack, the more I expect it to grow to the bottom.

So first thing `main` does is to preserve space on the stack for the local variables. When `main()` decides to call a subroutine, `main()` becomes the caller. We will assume `main()` has some registers it would like to remain the same, so it will save them. We will also assume that the callee function stakes some input args.

so , local variables, caller-save registers, and Args to pass to callee are in `main()` stack frame at the bottom of the stack.

so right to left, local variables, caller-save registers, args to pass to callee and then function call. git when we execute the call instruction, the return address(RET) gets save onto the stack and because the next instruction after the call will be the beginning of the called function, we consider the frame to have changed to the callee.



next, when subroutine starts, the frame pointer(ebp) still points to the main()'s frame. so the first thing it does is to save the old frame pointer on the stack and set the new value to point to its own frame.

so at the main() function, ebp is pointing at the top of the frame. when a subroutine is called, it will save this address to stack , take the ebp and continue its function, once the function is finished, it will restore the value of ebp so that the next subroutines can take up from original ebp register.

### STACK FRAMES ARE A LINKED LIST!

The ebp in the current frame points at the saved ebp of the previous frame

# Example1.c

The stack frames in this example will be very simple.  
Only saved frame pointer (ebp) and saved return addresses (eip).

```
//Example1 - using the stack
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int sub(){
    return 0xbeef;
}
int main(){
    sub();
    return 0xf00d;
}

sub:
00401000 push    ebp
00401001 mov     ebp,esp
00401003 mov     eax,0BEEFh
00401008 pop    ebp
00401009 ret

main:
00401010 push    ebp
00401011 mov     ebp,esp
00401013 call    sub (401000h)
00401018 mov     eax,0F00Dh
0040101D pop    ebp
0040101E ret
```

```
int sub(){
    return 0xbeef;
}

int main(){
    sub();
    return 0xf00d;
}
```

NOTE : ON INTEL REGISTERS AND CALLS, DESTINATION IS ON THE LEFT AND SOURCE IS ON THE RIGHT

SO

MOV EBP,ESP MEANS

Move esp TO EBP!!!

IN AT&T syntax, this is the opposite tho.

sub:

```
00401000 push    ebp
00401001 mov     ebp,esp
```

```

00401003 mov eax,0BEEFh
00401008 pop ebp
00401009 ret

main:
00401010 push ebp
00401011 mov ebp,esp
00401013 call sub(401000h)
00401018 mov eax,0F00Dh
0040101D pop ebp
0040101E ret

```

note : push ebp mov ebp,esp codes are generated by compiler automatically

```

main:
00401010 push ebp

```

when code started to executing, there was some value in ebp. so initially, main pushes ebp(base pointer). so when we push ebp, since esp always points to the top of the stack, we also see that esp is also changed.

stack is decremented by 4 bytes with this push since it pushes stack to down(LIFO)

**mov ebp,esp** means take whatever value in the esp and put it onto ebp. because intel x86 syntax is **instruction destination,source**

so now ebp and esp are the same. so whatever is at the address of ebp and low is the new stack frame.

**00401013 call sub(401000h)** in the main function is very self-explanatory. call the function called sub at the address of 401000h which is the beginning of the address of sub.

so what is at the stack now ?

```

00401013 call sub(401000h)
00401018 mov eax,0F00Dh

```

subroutine(function sub) needs to register the address of the next register so that after the execution of the subroutine, main function picks up from where it left off. hence, the address of mov eax,0F00Dh is registered to the stack.

now the stack is hosting 00401018 right after ebp.

next step is to calling sub routine. when subroutine is called, it will save the ebp to the register (push ebp) this ebp is pointing at the main function normally. then it will move esp to ebp like main function(mov ebp,esp). now ebp and esp are pointing at the same place. why **mov ebp,esp** is important? **by setting esp to ebp, sub is creating its own stack!!** next step, sub can start functioning, which is writing 0XBEEF to eax.

why? because eax holds the return values and the sub() function was only returning 0xbeef;. thus it makes sense!

thus , in the stack, eax is changed to 0x0000BEEF address since this is our return value.

now sub() function is executed and we do not need the base pointer anymore. next step is pop ebp since this is the end of the stack.

end we will exit the sub() function with **ret**.

what **ret** does is that whatever is on the top iof the stack, go ahead and take it from there and I will put there the insturction pointer and so this will be our next destination.

remember the code :

```
sub:  
00401000 push ebp  
00401001 mov ebp,esp  
00401003 mov eax,0BEEFh  
00401008 pop ebp  
00401009 ret  
  
main:  
00401010 push ebp  
00401011 mov ebp,esp  
00401013 call sub(401000h)  
00401018 mov eax,0F00Dh  
0040101D pop ebp  
0040101E ret
```

so when **ret** (0x000401009) is called, the top of the stack will have the value of will be replaced to what?

it will be replaced with **0x000401018** since this is the first address after the sub() function call. thus main() function will keep goint from here. so next step is **00401018 mov eax,0F00Dh**

whici moves the value 0x00F00D to the eax because is the return value. if we did not put this register, the eax would be stil under the effect of sub() function so would keep pointing at 0x00BEEF. but main() wants a return of 0xFOOD so we immediately overwrite it.

next, we tear down the stack with **pop ebp** and returning to the function.

with pop ebp and ret, we cleared the last 2 remaining registers of stack. now stack is all undefined.

Some notes:

since sub() is a deadcode , meaning its return value 0xFOOD, is not used for anything, compiler with optimization option on would delete sub() function.

because there are not input parameters to sub(), there is no diff whether we compile cdecl or stdcall calling convention.

in the debugger when we jump to the assembly debugger here is the C code and the corresponding assembly code :

```

int main(){
011E1010 push      ebp
011E1011 mov       ebp,esp
    sub();
011E1013 call      sub (11E1000h)
    return 0xf00d;
011E1018 mov       eax,0F00Dh
}
011E101D pop      ebp
011E101E ret

```

Lets do another example.

```

#include <stdlib.h>
int sub(int x, int y){
    return 2*x+y;
}

int main(int argc, char ** argv){
    int a;
    a = atoi(argv[1]);
    return sub(a,a);
}

.text:00000000 _sub:    push  ebp
.text:00000001          mov   ebp,esp
.text:00000003          mov   eax,[ebp+8]
.text:00000006          mov   ecx,[ebp+0Ch]
.text:00000009          lea   eax,[ecx+eax*2]
.text:0000000C          pop   ebp
.text:0000000D          retn 
.text:00000010 _main:   push  ebp
.text:00000011          mov   ebp,esp
.text:00000013          push  ecx
.text:00000014          mov   eax,[ebp+0Ch]
.text:00000017          mov   ecx,[eax+4]
.text:0000001A          push  ecx
.text:0000001B          call  dword ptr ds:_imp_atoi
.text:00000021          add   esp,4
.text:00000024          mov   [ebp-4],eax
.text:00000027          mov   edx,[ebp-4]
.text:0000002A          push  edx
.text:0000002B          mov   eax,[ebp+8]
.text:0000002E          push  eax
.text:0000002F          call  _sub
.text:00000034          add   esp,8
.text:00000037          mov   esp,ebp
.text:00000039          pop   ebp
.text:0000003A          retn 

```

88

## r/m32 Addressing Forms

Anywhere you see an r/m32 it means it could be taking a value either from a register or a memory address. (r/m == register/memory) in **Intel Syntax**, most of the time square brackets([]) means to treat the value within as a memory address and fetch the value at the address (like dereferencing pointer)

- mov eax,ebx
- mov eax, [ebx]
- mov eax, [ebx+ecx\*X] => X can be 1,2,4 or 8.

so for example here, it means, take the ecx(lets say 0x00344F) and multiply it with X, go to the address in this multiplication, add it to ebx, and write the cumulation to eax.

the most complicated form is : [base + index \* scale + disp]

base +(index \* scale ) + displacement.

mov eax ,[ebx+ecx\*X+Y] where Y is one byte(0-255) or 4 bytes(0-2^32-1)

Scenario : data type we have is an array. Its consequential data type. So ebx points to the base of the Array, beginning point of the array. Ecx is the index number(so A[0]) in this case what should be the X? it could be a 4 byte element, 2 bytes elements. so to jump to correct next position, X is the length of the element.

base + 0x4

base + 1x4

base + 2x4

so 4 is the elements length. hence, ecx \* X is jumping to the next element in each iteration.

so what is displacement then ? Why is is used?

When the given array is **multidimensional** array. so to give an idea:

Scenario : in the C code I have 2 arrays of 10 bytes. I want to iterate over each array. using **base +(index \* scale )**, I started at the base0 of the array1 and went through all the elements. Now I want to jump to the second array right? So in order to move the ebx to the base of the second array, I use displacement constant.

[0,1,2,3,4,5,6,7,8,9] [B0,B1,B2,B3,B4,B5,B6,B7,B8,B9]

when ebp is pointing to **0**, **base +(index \* scale )** will iterate through the array. now I eax is pointing to **9** but I want to jump to **B0**. Addinng the **displacement** constant to ebp will help pointer to jump to **B0**

what does this mean ? ==> – **mov eax, [ebx]** go to ebx, read what is in that memory address, read 4 bytes from memory and write it on eax.

Note that brackets can be on one side or on the other side but never on both sides because there is not memory to memory operation!

just before exiting a function, generally the main function, we generally see the register mnemonic **mov ebp, esp** meaning level the base and the top pointers of the stack.

In practice this means destroy all the local variables in the stack. We are closing down the shop, nothing should be left in the stack.

The next move, of course, is to destroy the pointers from the stack which is **pop ebp** and finally **ret**.