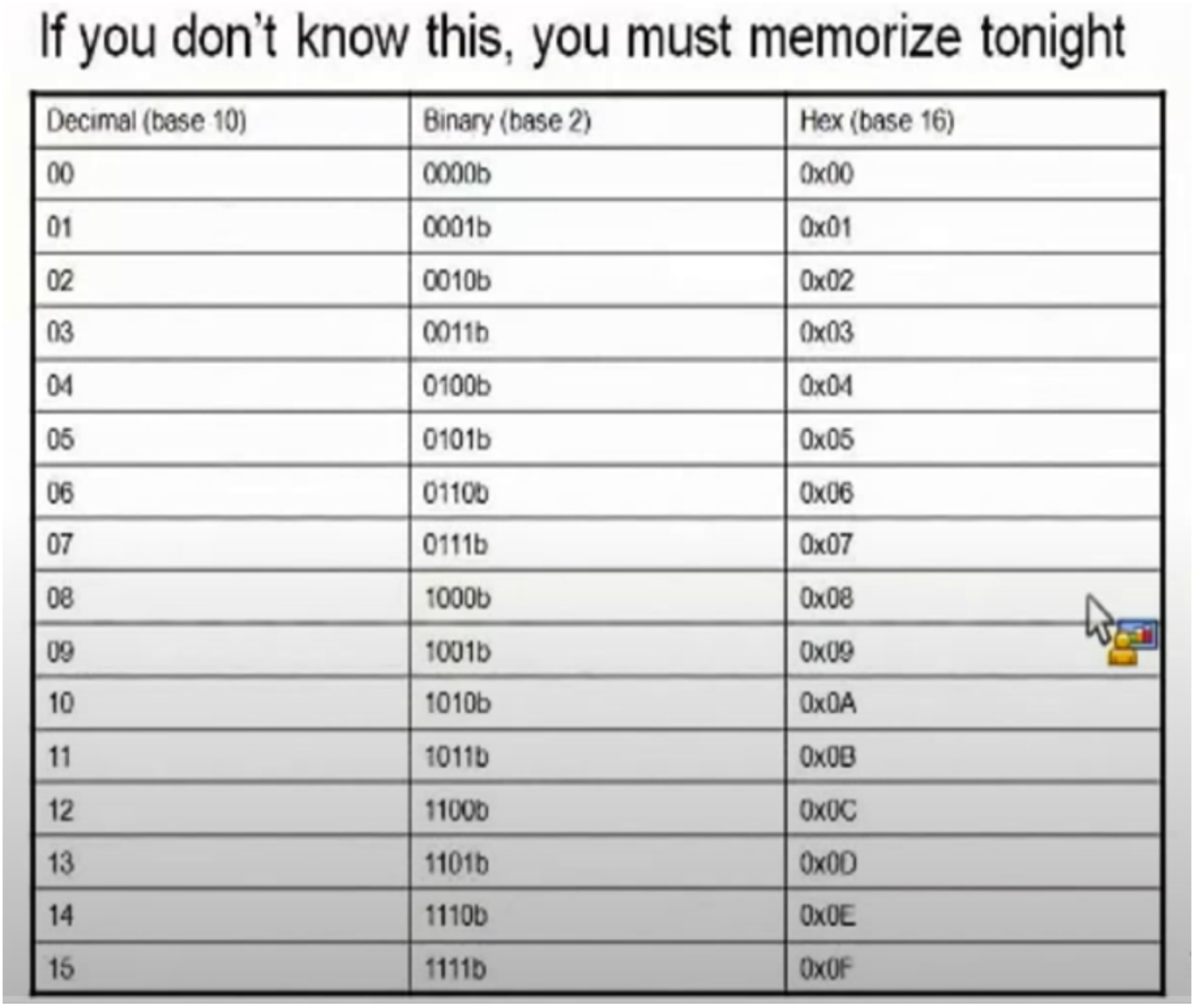
Intel X86 Processors

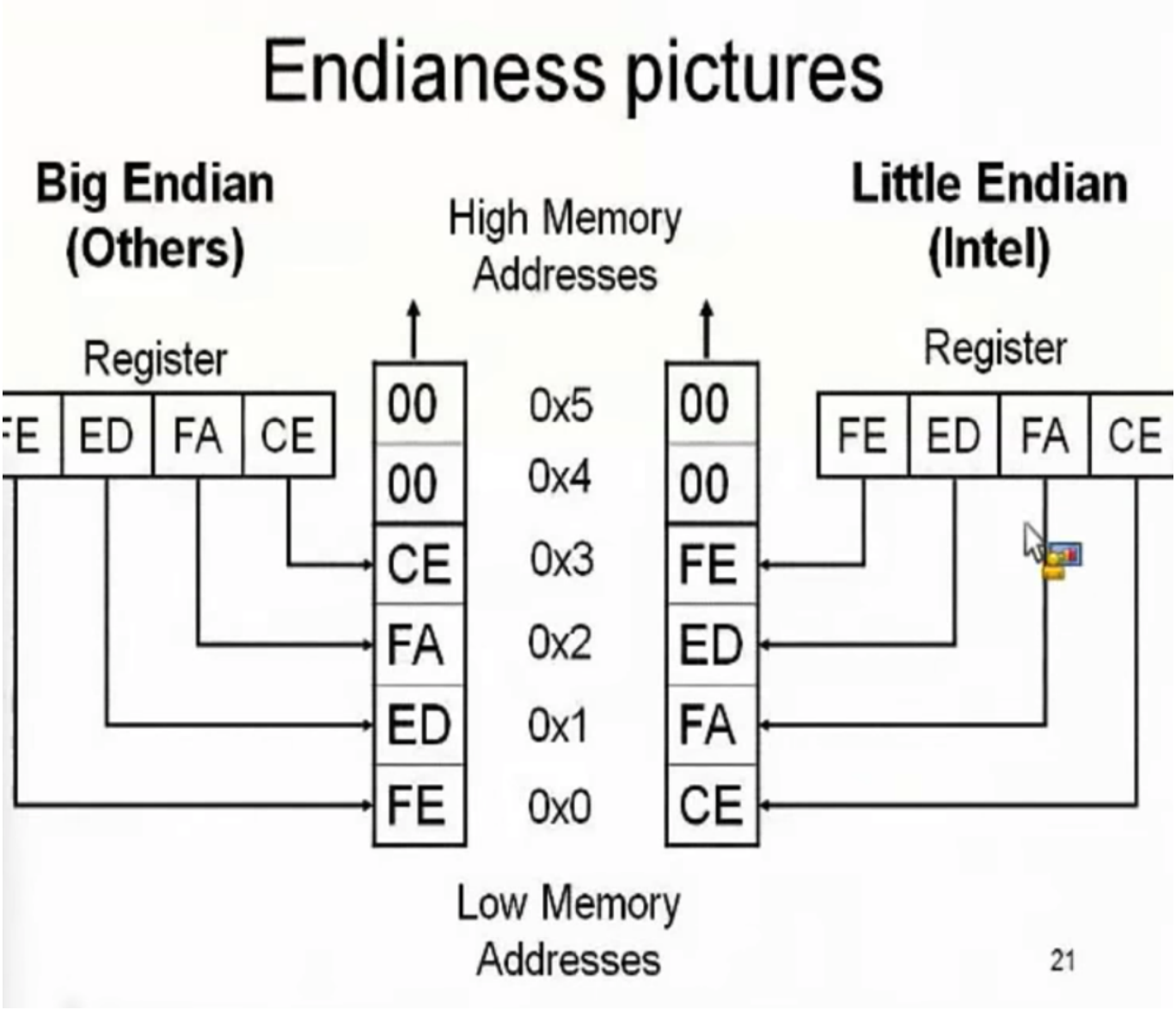
> only 14 assembkly instructions make up to 90 percent of the code!

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



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.



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 regusters(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 functiion, it wont 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 resuklt of some instruction is zer, cleared otherwise Sign Flah(SF) - Set equal to the most significant bit of the result.

INSTURCTIONs

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 thats another class.

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

PUSH ==> push word, double word ort quadword onto the STACK. can either be emmediuate(a numeric constant) of the value in a register.

fror our purposes, it will; always be a DWORD(for this course) the push instruction aautomatically decrements the stack pointter, esp, by 4. Why decrements? bcs its stack, its FILO.

POP ==> Pop a value from the Stack

take a Dword off the stack, put it ina 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.

DEstionation address can be specified in a multiple ways.

- Absolute address ==> address 0x0030434
- Relative address(relative the the end of the instruction) ==> some adress 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 instructuon ius just written as ret.

typically used by cdecl functiuons

2. pop the top of the stacj into eip and add a constatn number of bytes to esp.

in this form , the instruction is written as ret 0x8 of 0x20 etc. typically used bty stdcall fucntions

`MOV` ==> Move.

Can move:

register to register

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

immediate is hardcoded value.

> NEVER MEMORY TO MEMORY

> Memory addresses are given in r/m32 from.

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

THE STACK

stack is the conceptiual area of main memory which is designated by OS when program is started. Stack is LIFO/FILO 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 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

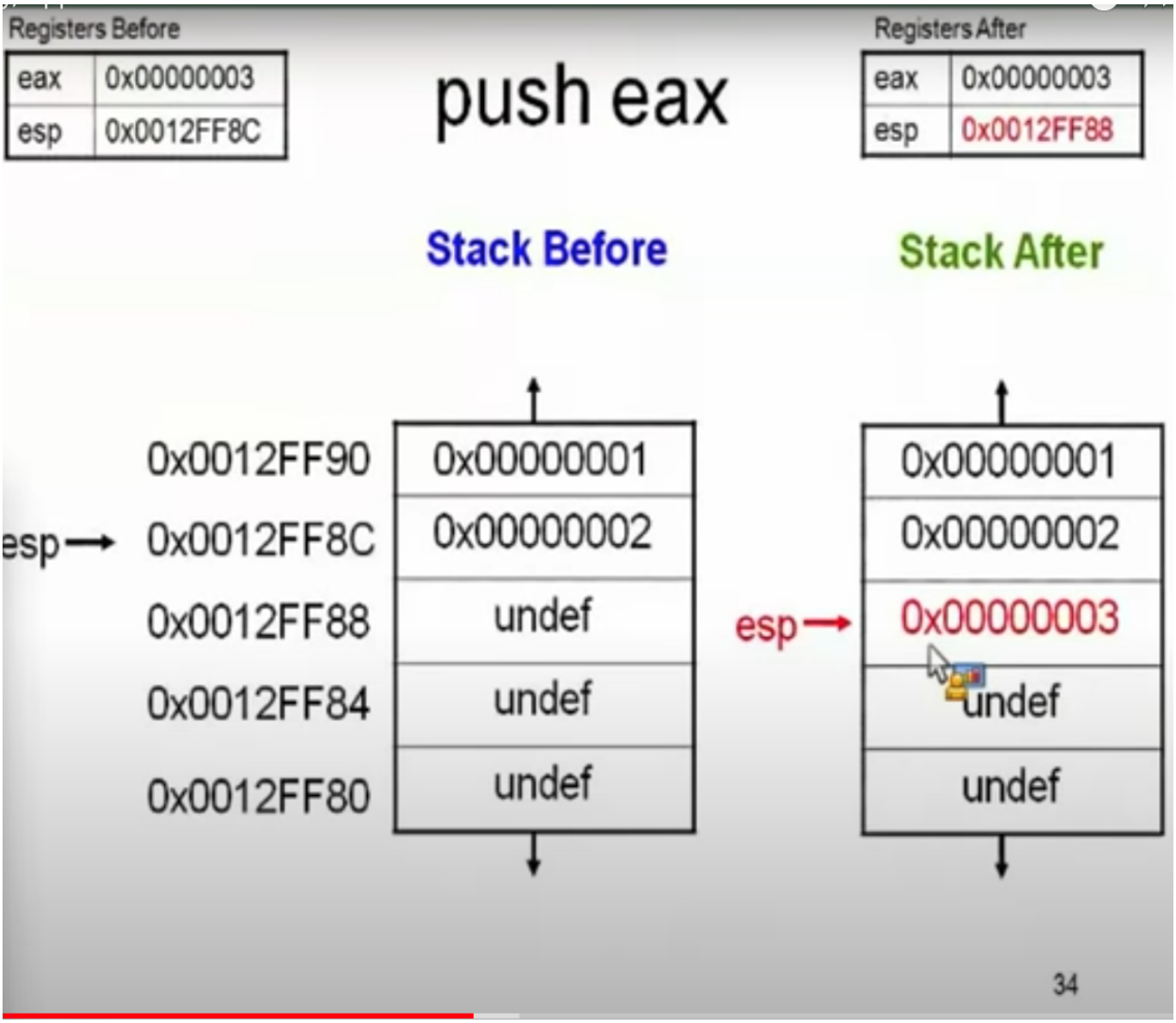
Register in time 1 :

eax : 0x00000003 esp : 0x0012FF8C

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

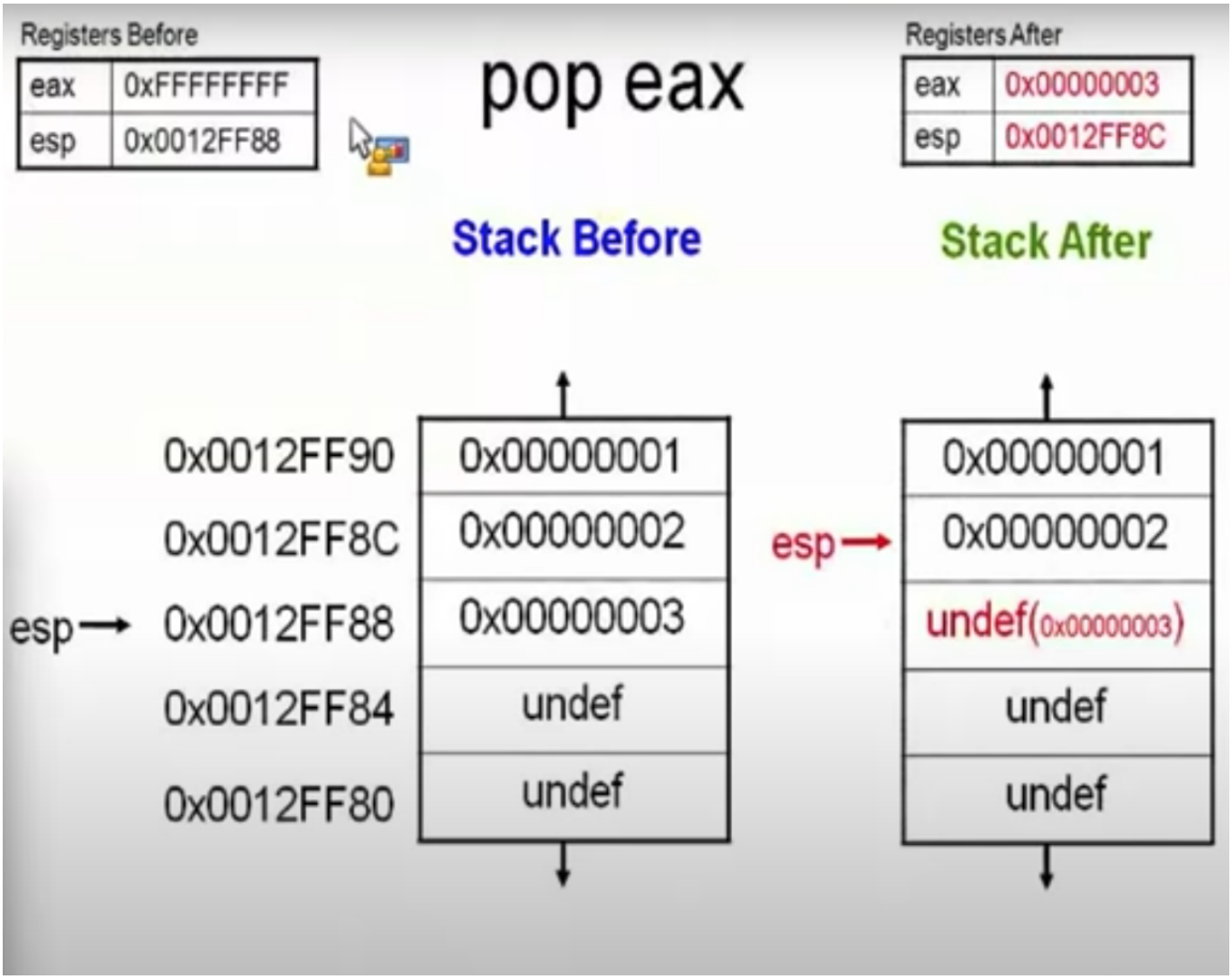
Register after push eax:

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 ppusth the pointer and trhen 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 regiester is calling the function is also responsible to clean up the stack

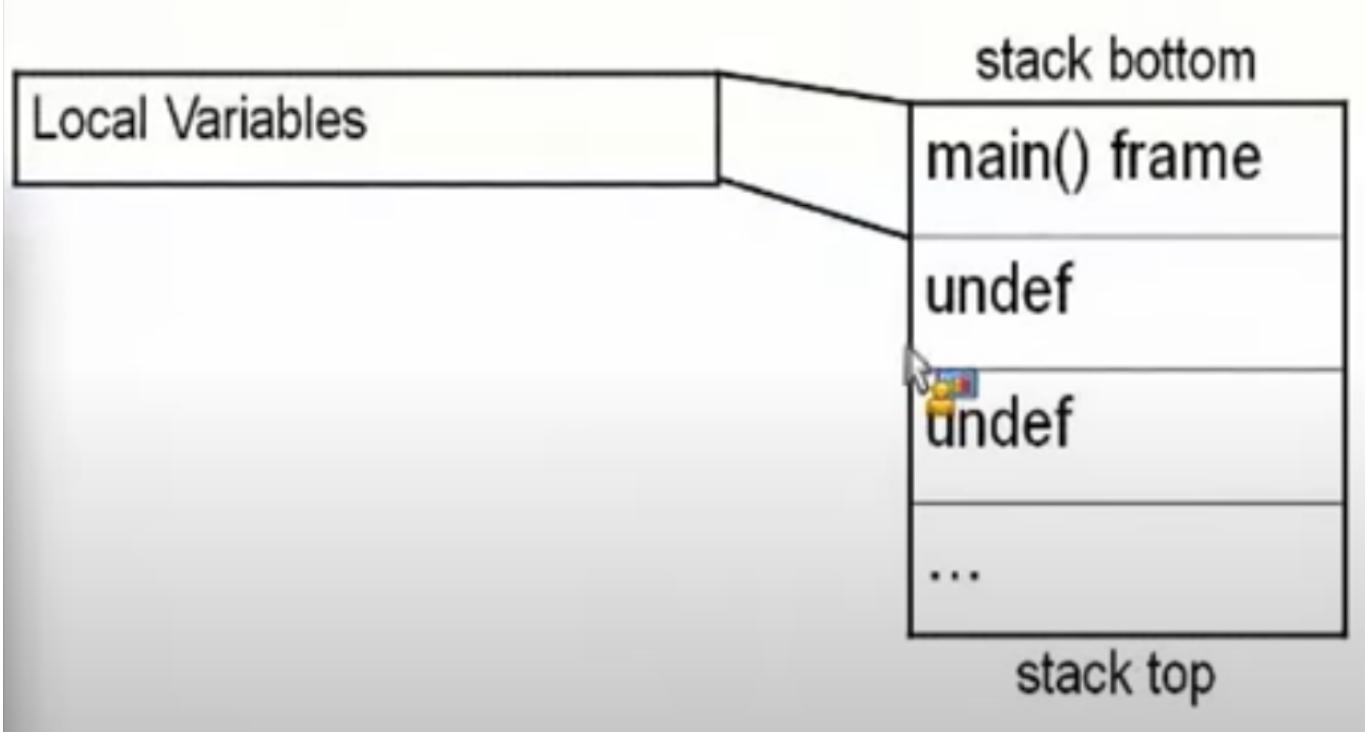
stdcall

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 responsibnle for cleaning ip 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 witrh.



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

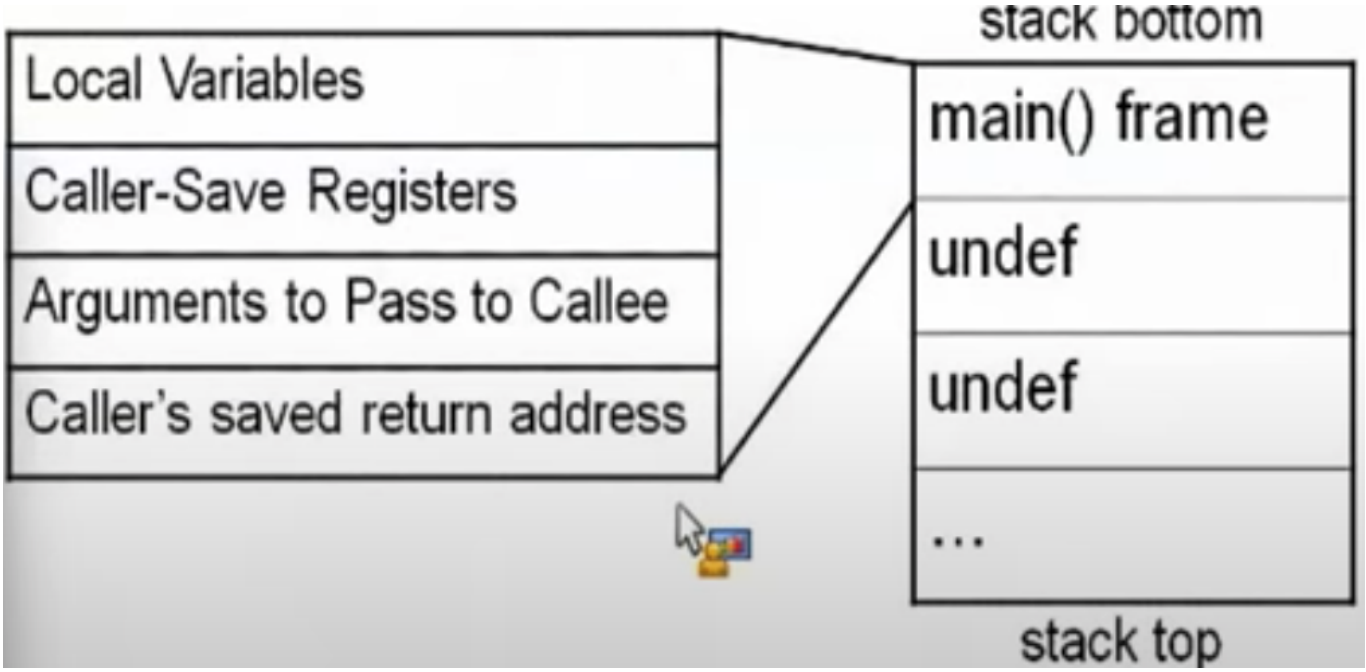
The more I add to the stack, the more I expect it to grow to the bottom.

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.

when we execute the call instruction, the return address(RET) gerts save onto the stackm and becuse the next instruction after the call will be the beginning of the called function, wer consiter the frame to have changed to the callee.



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

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

> STACK FRAMES AR 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
```

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

int main(){
    sub();
    return 0xf00fd;
}
```

```
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 automaticaly

```
main:
00401010 push ebp
```

when code started to executing, there was some value in ebp. so initially, main pusheh 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 teh 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 registrered 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 0X0BEEF 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.