THE UNIVERSITY OF ARIZONA
UA South

CYBV 471 Assembly Programming for Security Professionals
Week 10

Stack, function, and assembly instructions

Agenda

- **What is the Stack?**
- **Calling Function Procedure**
- **Assembly Language Instructions**
  - Understand “MOV” instruction
  - Understand “NOP” instruction
  - Understand “LEA” instruction
  - Understand “NOP” instruction
  - Understand “LEA” instruction
  - Understand “PUSH” instruction
  - Understand “POP” instruction
  - Understand “CALL” instruction
  - Understand “RET” instruction

Mov vs LEA Instructions

- mov destination, source
  Moves (copies) data from one location to another (RAM/Register)
- Operand surrounded by brackets references to data located in memory location
  - [ebx] points to data located in the memory address stored in EBX
  - [0x4037c4] points to data located in memory address 0x4037c4

Instruction	Description
`mov eax, ebx`	Copies the contents of EBX into the EAX register
`mov eax, 0x42`	Copies the value 0x42 into the EAX register
`mov eax, [0x4037C4]`	Copies the 4 bytes at the memory location 0x4037C4 into the EAX register
`mov eax, [ebx]`	Copies the 4 bytes at the memory location specified by the EBX register into the EAX register
`mov eax, [ebx+esi*4]`	Copies the 4 bytes at the memory location specified by the result of the equation ebx+esi*4 into the EAX register

Load Effective Address (LEA) instruction

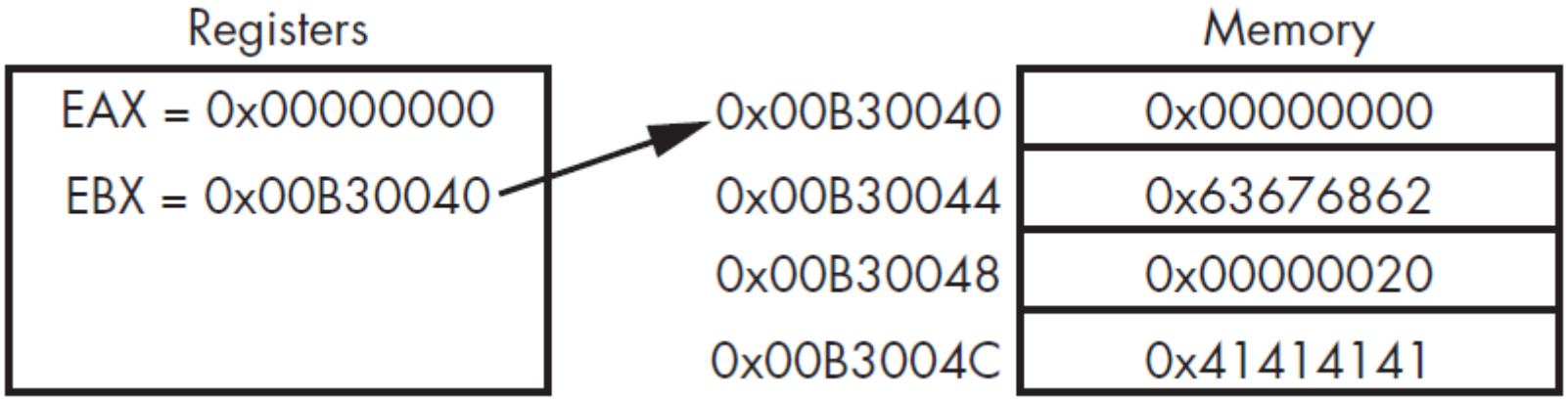
- lea destination, source
  - Put a memory address (located in source) into the destination
- lea eax, [ebx+8]
  - Puts memory address (ebx+8) into eax
- In contrast

  mov eax, [ebx+8]

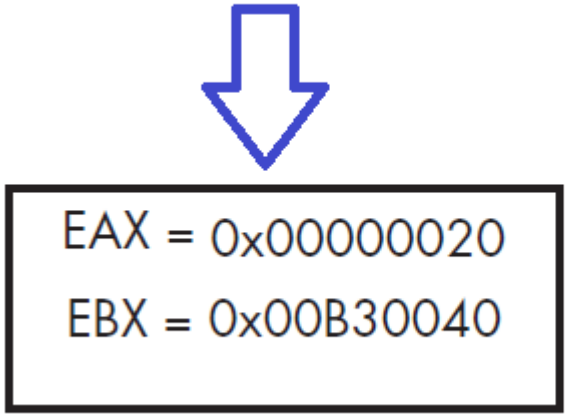
  Moves the data located in memory address pointed by (ebx+8) into eax

mov & lea Instruction Examples

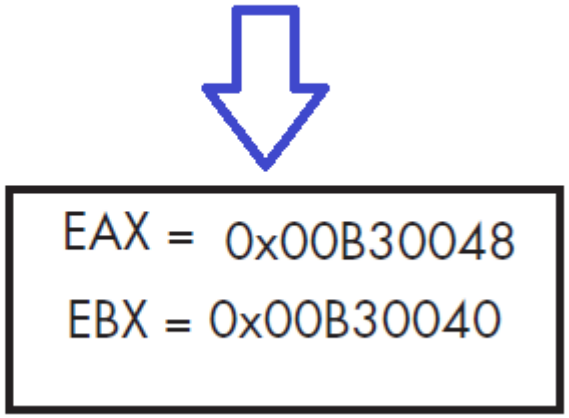
- Values for registers EAX and EBX on the left and the information contained in memory on the right



mov eax, [ebx+8]



lea eax, [ebx+8]

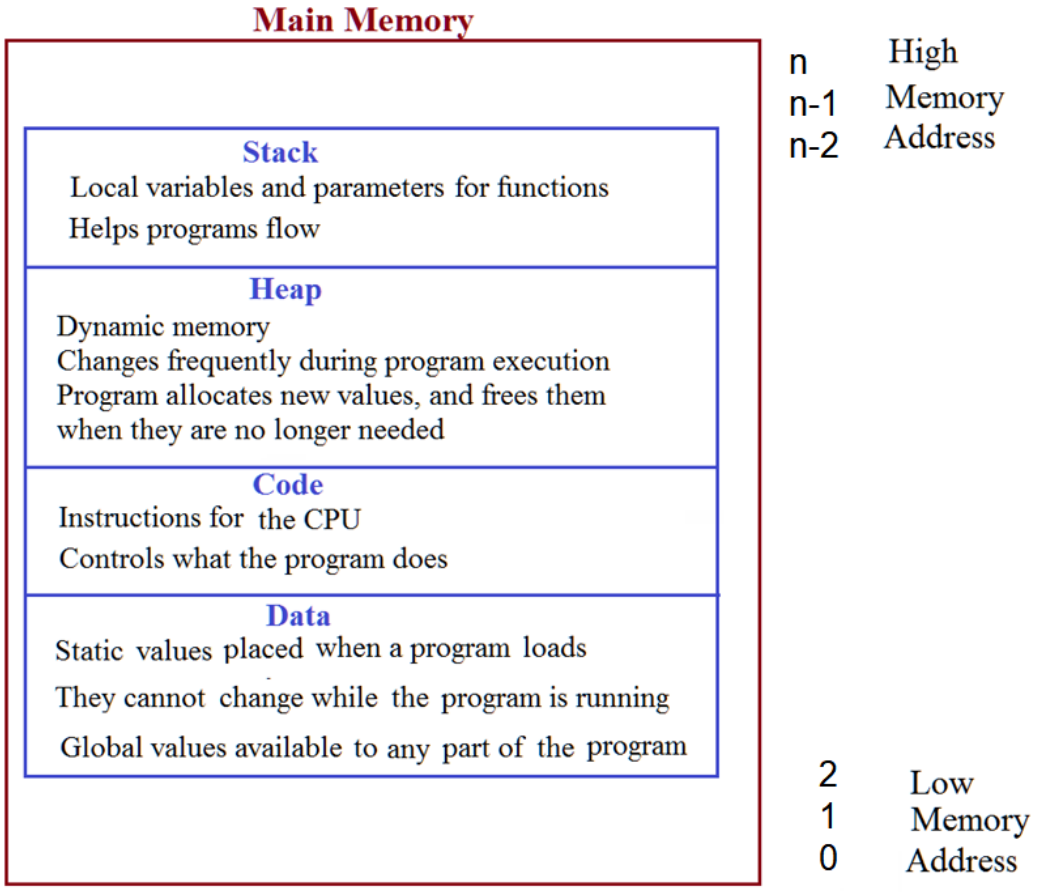


NOP instruction

- Does nothing, proceed to the next instruction
- The opcode for this instruction is 0x90
- Used to pad/align bytes, or to delay time

What is a Stack?

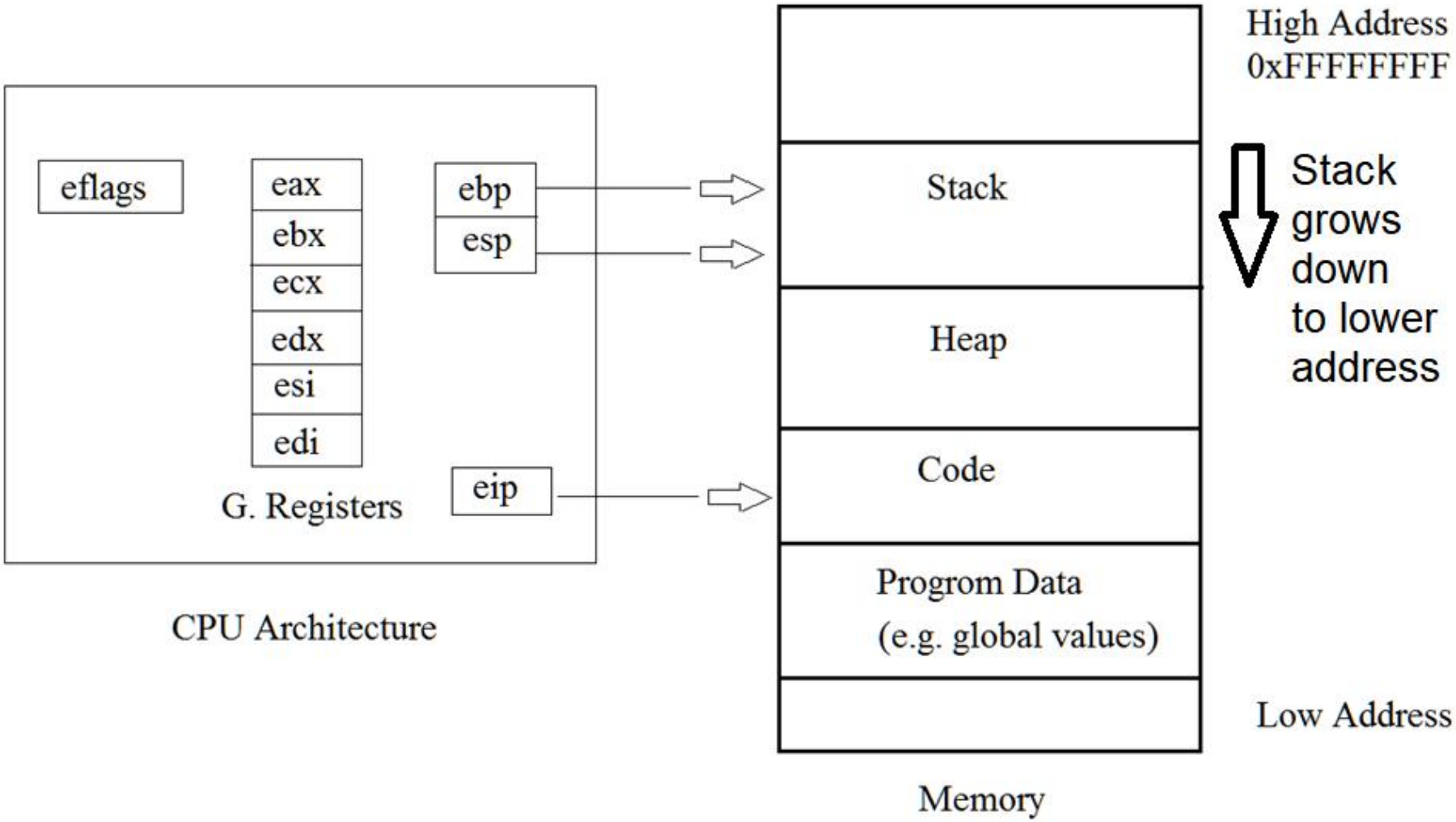
- A stack is a temporary memory storage region that holds a function variables, data, registers values.
- Every function has its stack frame



Basic memory layout for a program

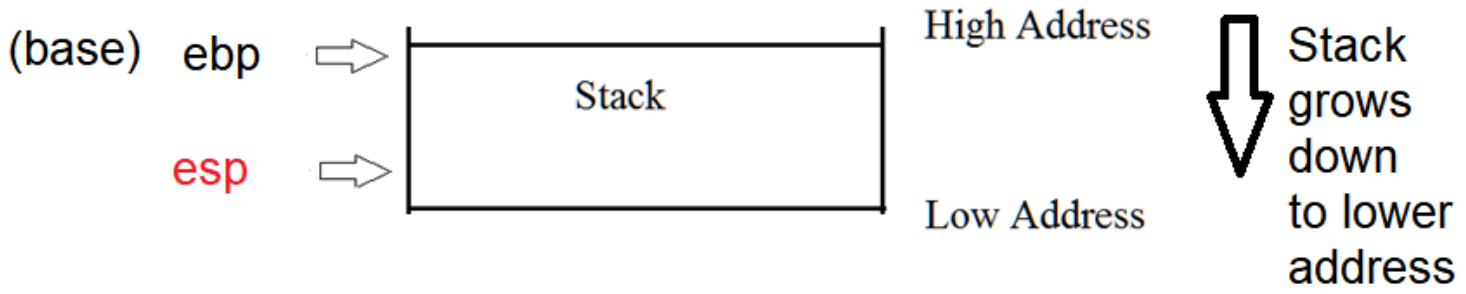
What is a Stack?

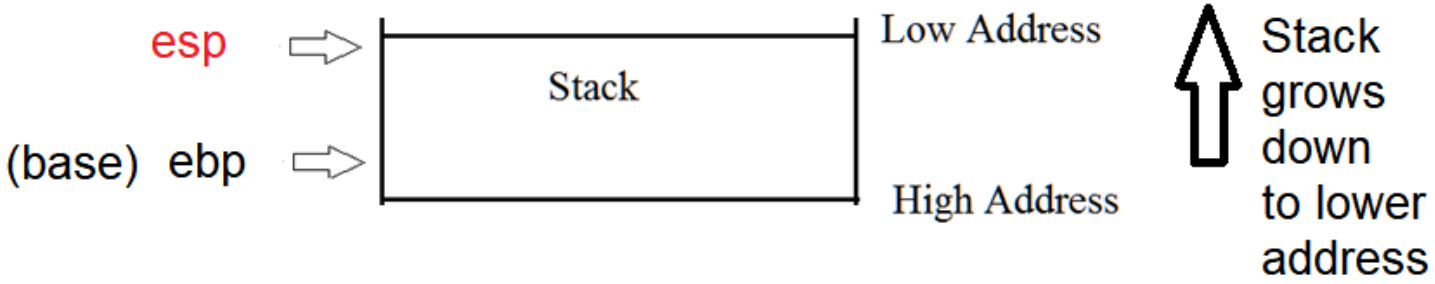
- The stack grows down towards lower addresses
- The value ESP is the "top" of the stack or the lowest address used by the stack



Visualizing the stack

- Some books show stack grows down towards low addresses
- Some books show stack grows up towards low addresses
- Debuggers show stack grows up towards low addresses

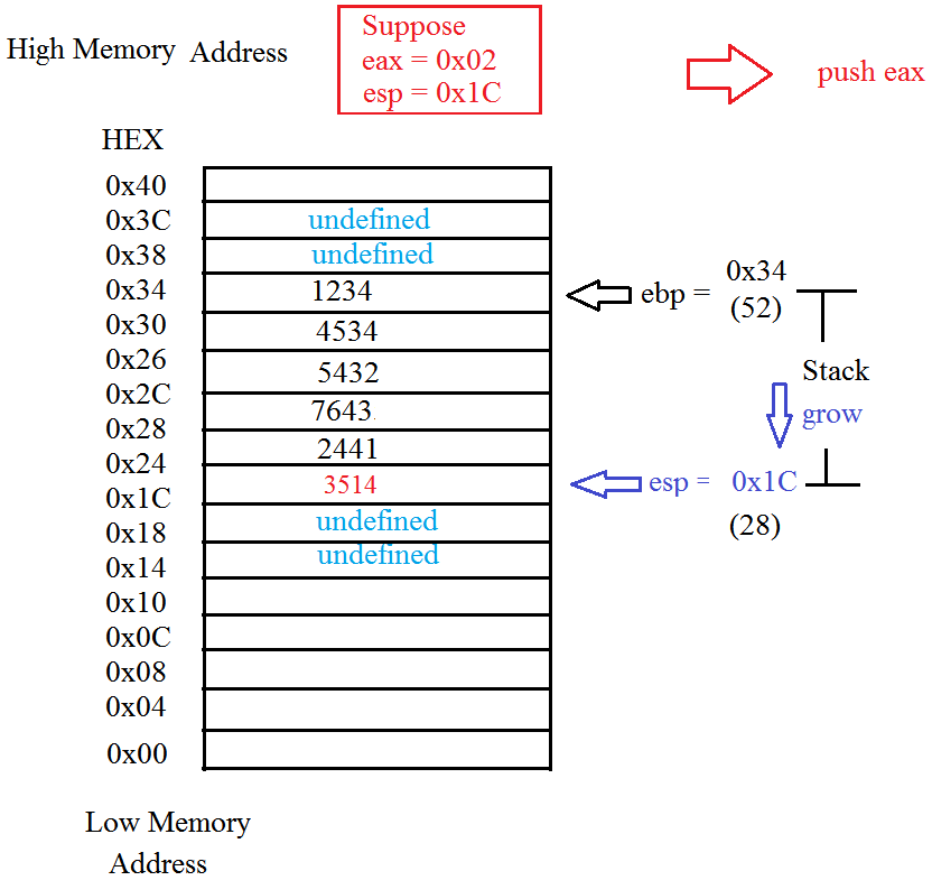


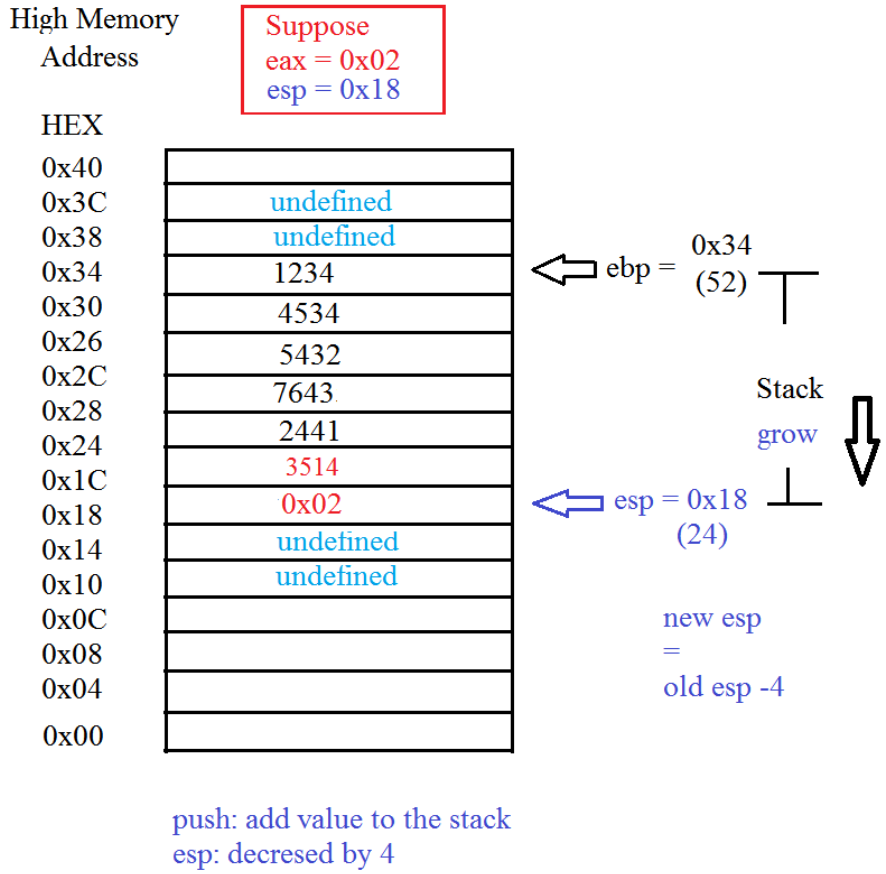


PUSH Instruction

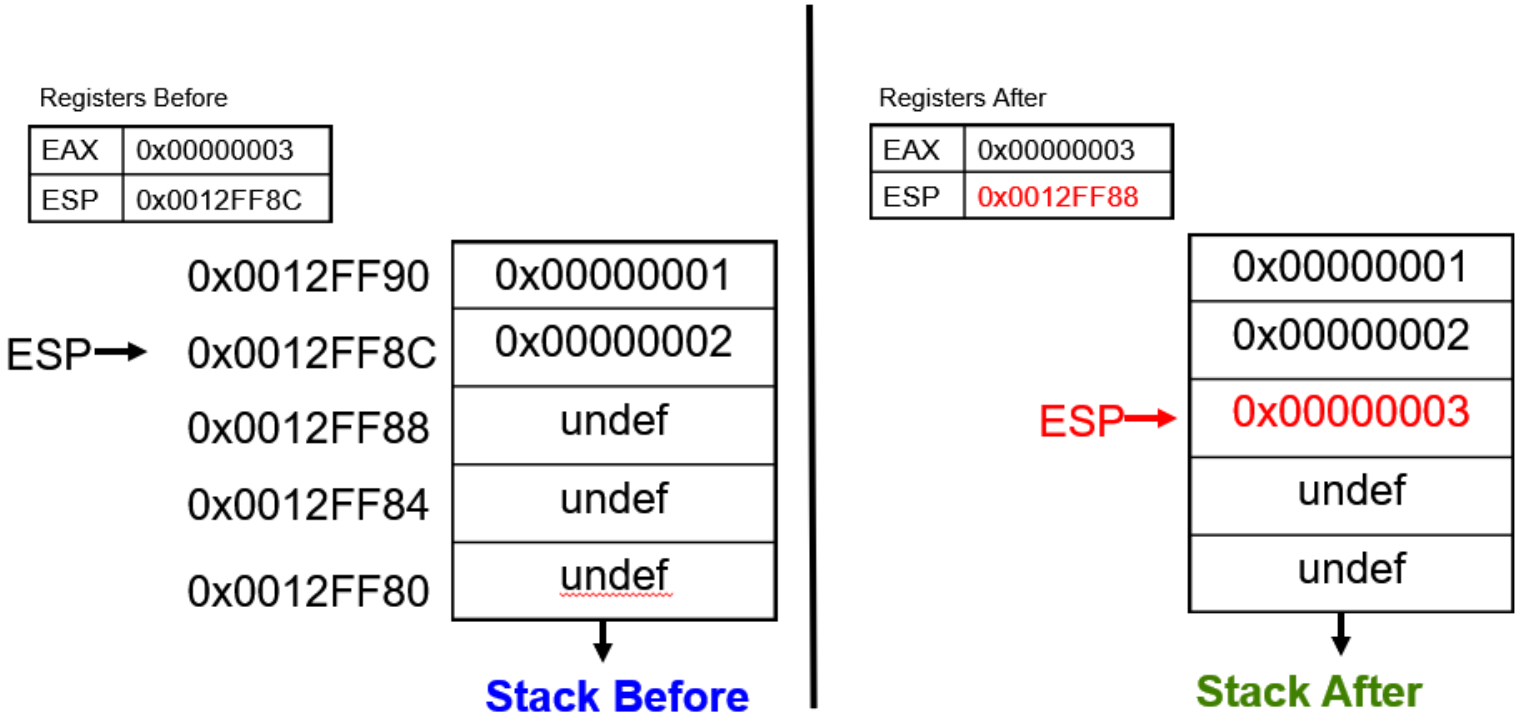
- Pushes a DWORD onto the stack
  - can either be an immediate (a numeric constant), or the value in a register
- Two operations in one (in order)
  1. Adjusts the stack pointer (New ESP = Old ESP - 4)
  2. Writes the new address value to ESP
- Note that if a register is pushed, the value in the register will not change

PUSH Instruction





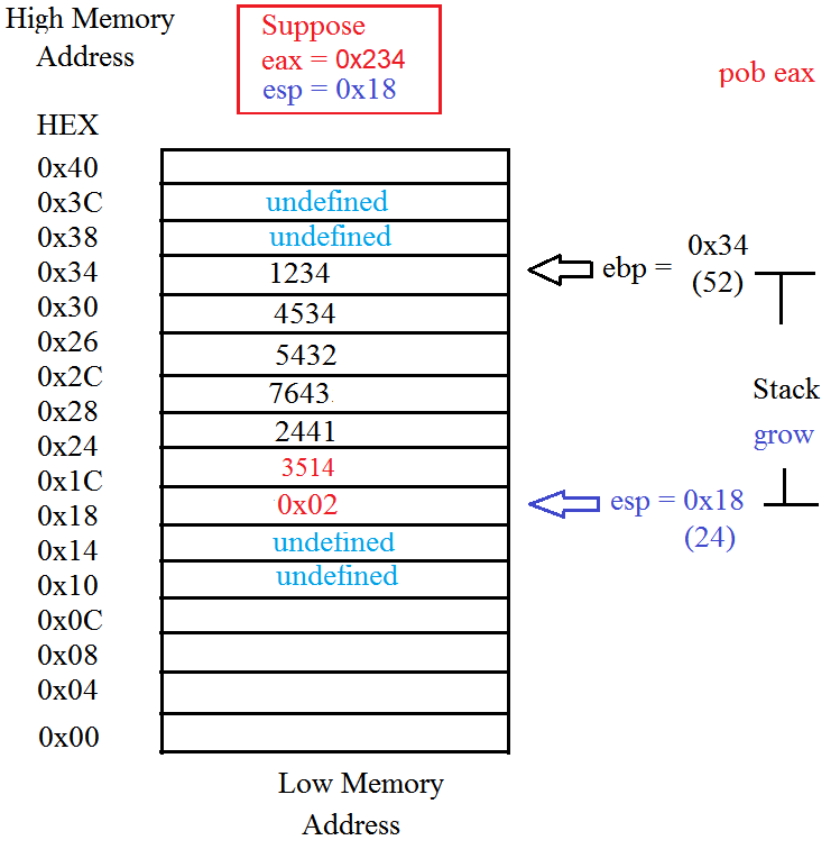
PUSH Instruction
Push eax

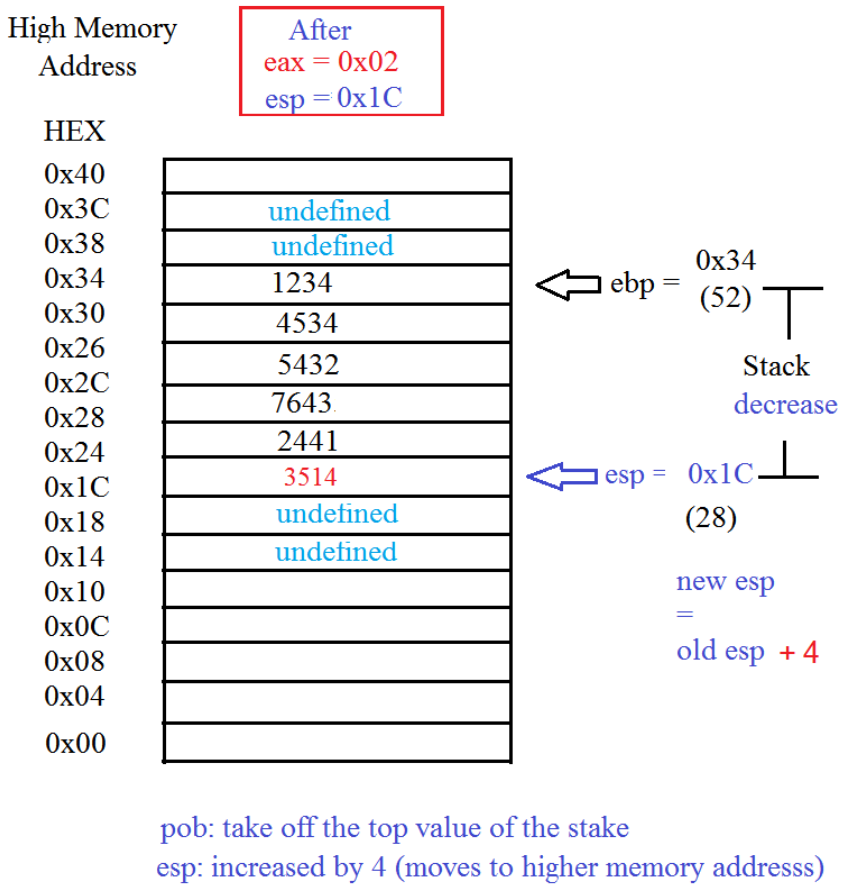


POP Instruction

- POPs a DWORD from the top of the stack INTO a register
- Two operations in one (in order)
  1. Adjusts the stack pointer (New ESP = Old ESP + 4)
  2. Writes the new address value to ESP
- Note that the value on the stack is not changed; only ESP changes

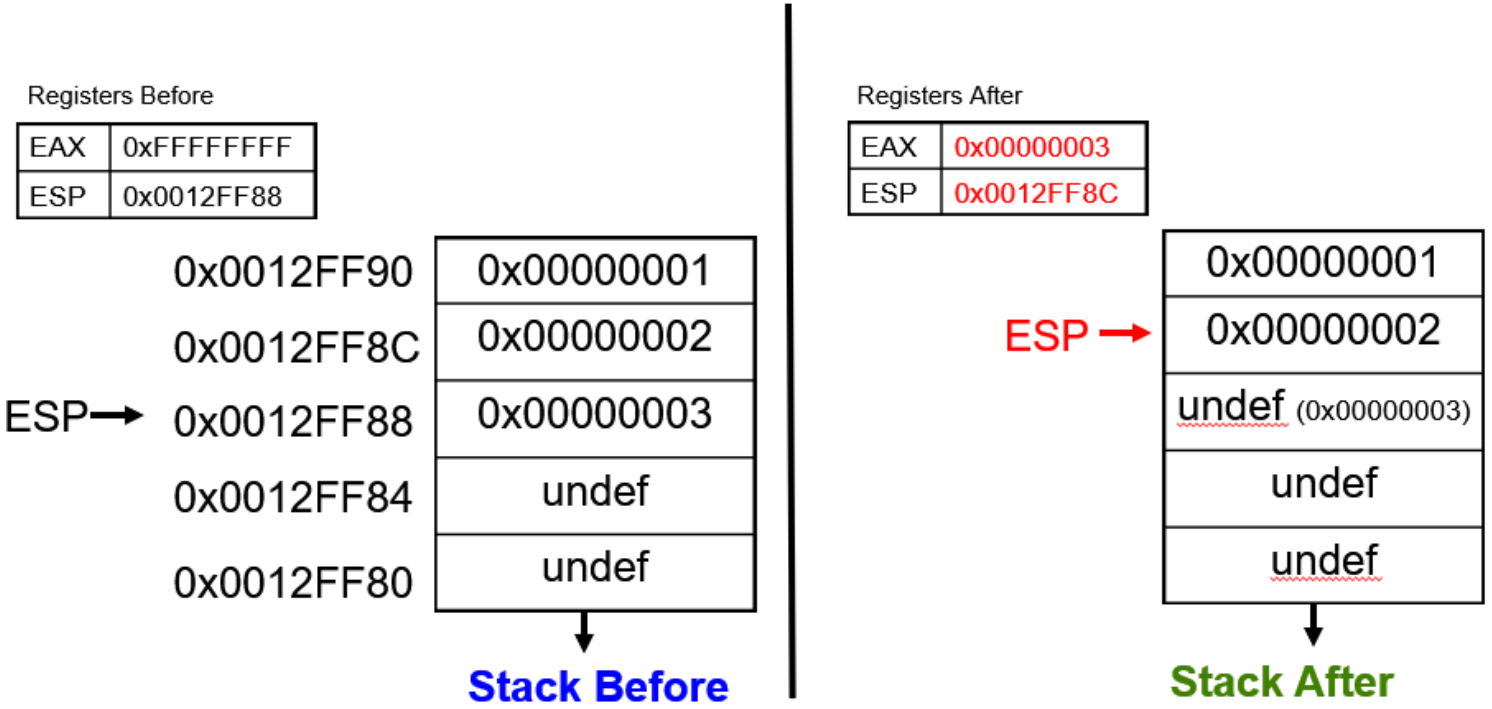
POP Instruction



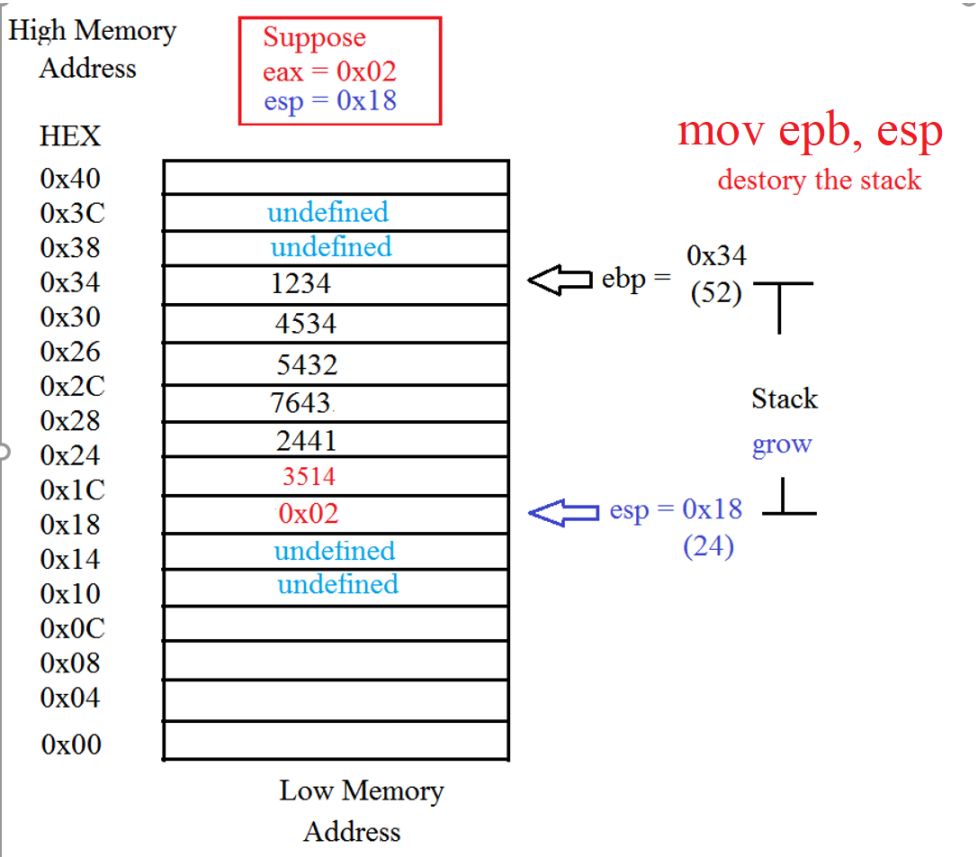


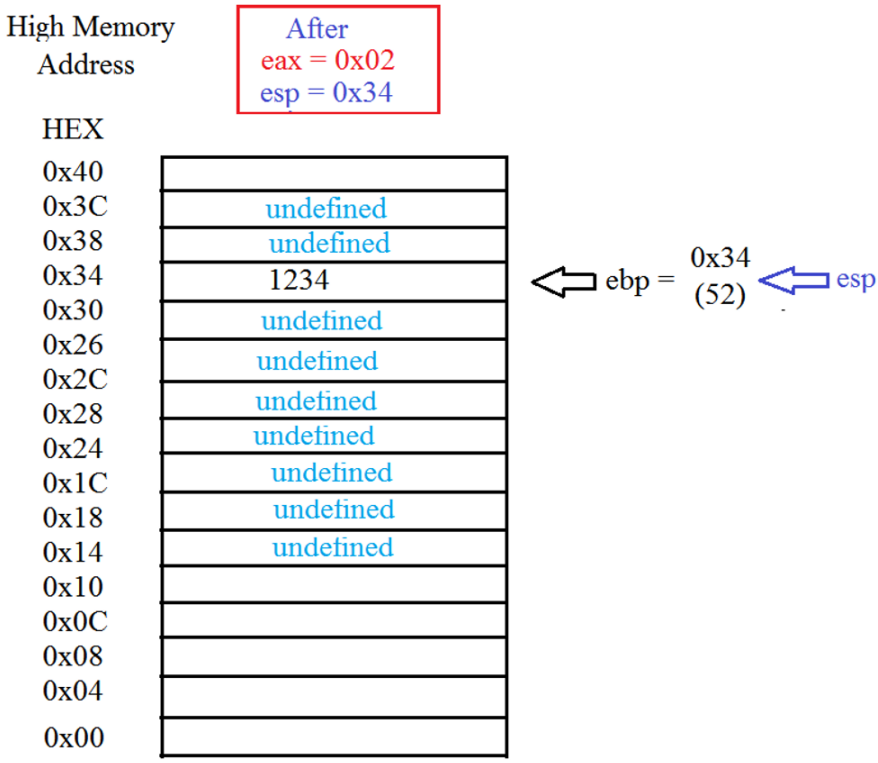
pob: take off the top value of the stake
esp: increased by 4 (moves to higher memory addresss)

POP Instruction
POP eax



Destroy a Function stack





Function Calls

- Functions are portions of code within a program that perform a specific task
- Functions are relatively independent of the remaining code.
- The main code calls and temporarily transfers execution to functions before returning to the main code.

- Each time a call is performed, a new stack frame is generated

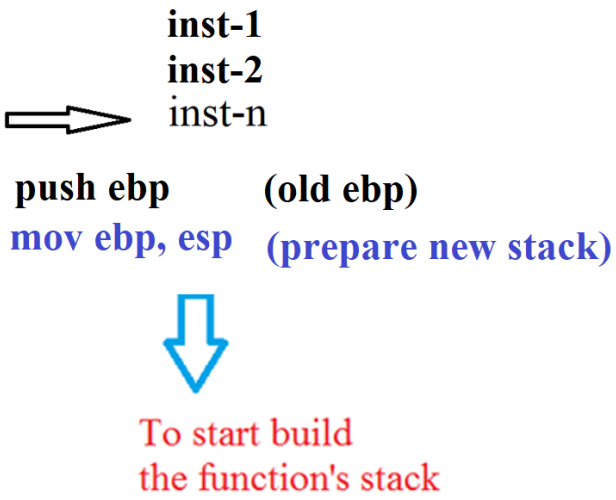
  Prologue: Instructions at the start of a function that prepare stack and registers for the function to use

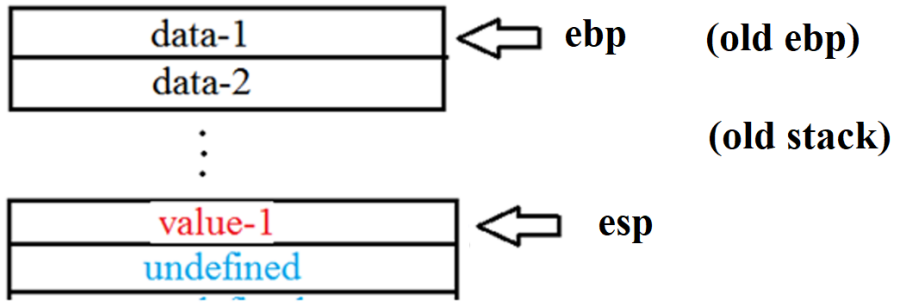
- A function maintains its own stack frame until it returns.

  Epilogue: Instructions at the end of a end of a function that restore the stack and registers to their state before the function was called

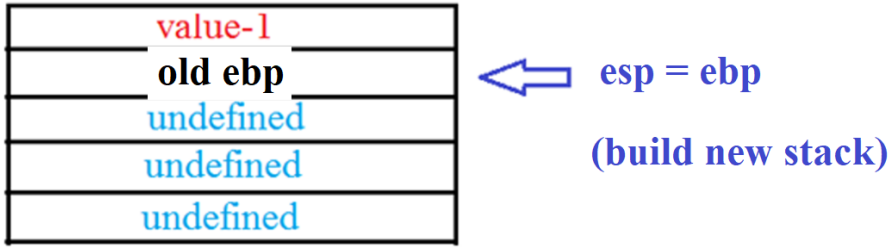
Code for Starting a Function stack

Prologue Part



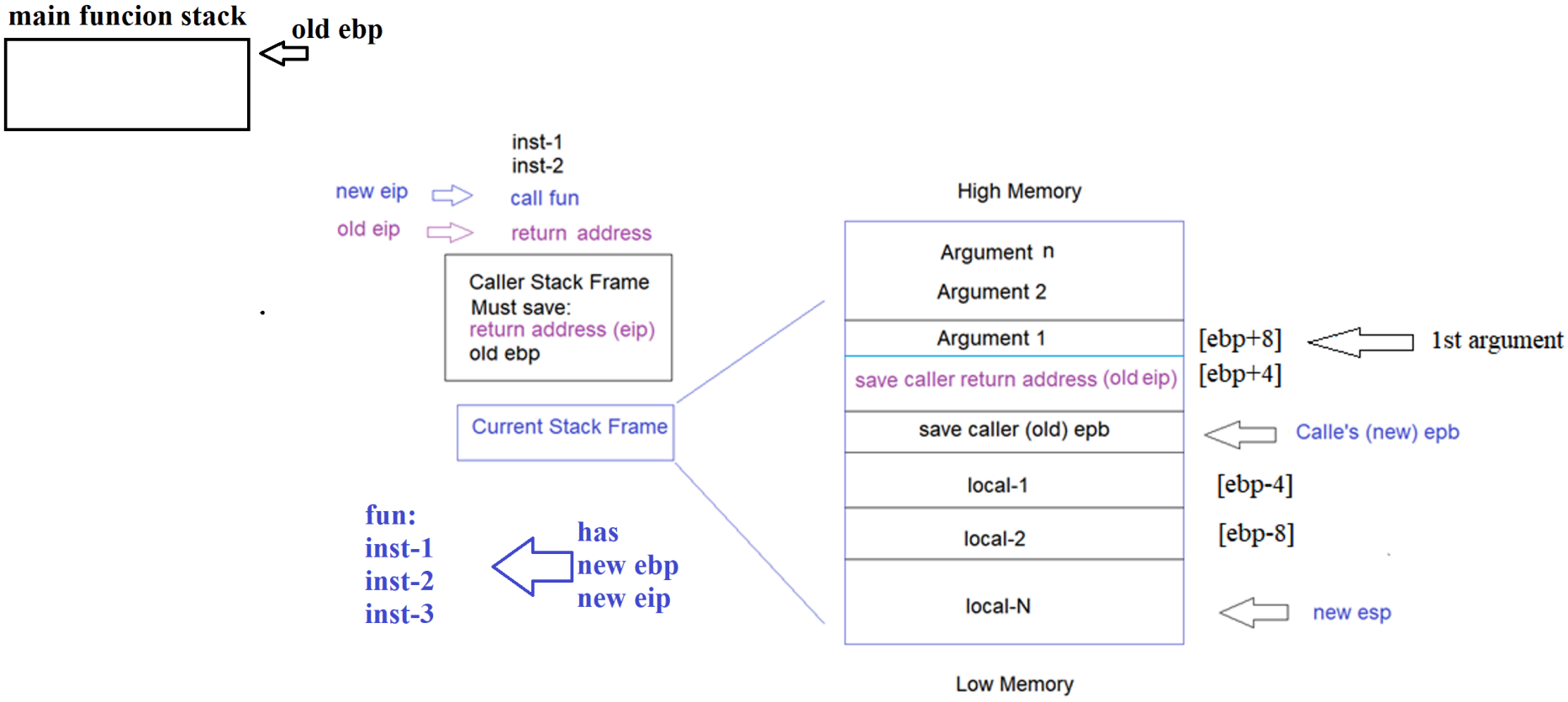


Start build a function stack



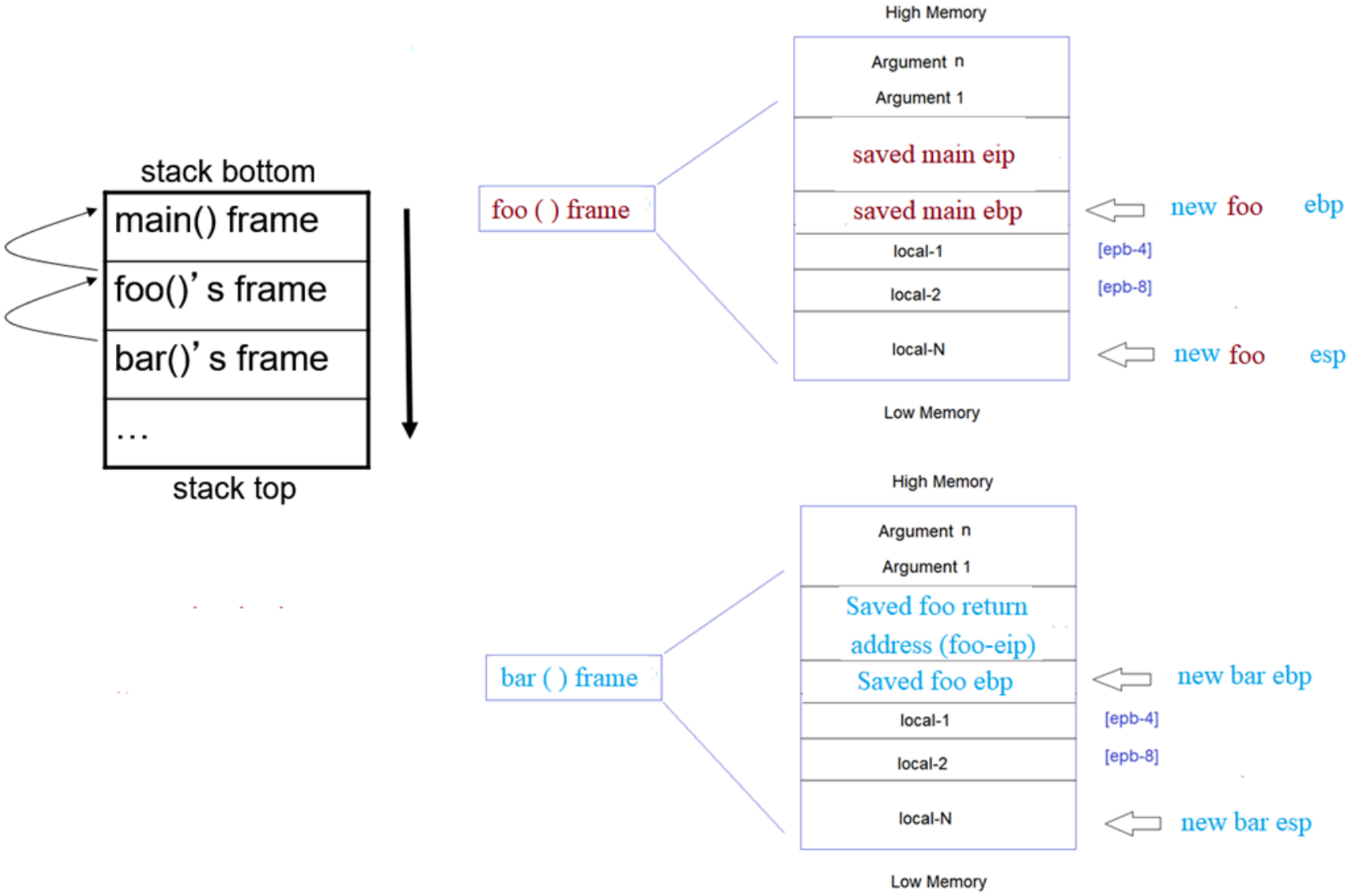
CALL – Call Procedure

- Two operations in one (in order):
  1. PUSH the address of the next instruction (old eip value) onto the stack
  2. Set new EIP to the CALL's destination address



Stack Frames are a Linked List

The EBP in the current frame points to the saved EBP of the previous frame.



Function epilogue

- Free the called function variables
- Restore the old pointer (old eip) of the calling function
- Return the control to the calling function

```
mov ebp, esp      ; destroy the stack
pob ebp           ;  get old ebp value
ret               ; pob old eip ; get old eip value
```

RET – Return from Procedure

- Ret: The function returns by calling the ret instruction. This pops the return address off the stack into EIP (get the old EIP value), so that the program will continue executing from where the original call was made.
- If "RET <number>", two steps:

  a- POPs the top of the stack into EIP as before

  b- Adds <number> to EAX

  - Examples: RET 0x8 or RET 0x20
- Leave: Sets ESP to equal EBP (destroy the stack) and pops EBP off the stack (get the old EBP value)

Registers and calling conventions

- Stack conventions:
  - ESP – Top of the stack
  - EBP – Pointer to last stack frame (next slide)
- Register saving ***conventions***
  - EBX, ESI, EDI and EBP persist across CALLs
  - Callers MUST ASSUME that EAX, ECX, EDX and ESP will change
  - Callers must save its register values to avoid any issues program crash

Example1.c

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

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

Example1.c

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

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

```
sub:
00401000  PUSH  EBP
00401001  MOV   ESP, EBP
00401003  MOV   EAX, 0BEEFh
00401008  POP   EBP
00401009  RET
```

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

```
main:
00401010  PUSH  EBP
00401011  MOV   ESP, EBP
00401013  CALL  SUB (401000h)
00401018  MOV   EAX,0F00Dh
0040101D  POP   EBP
0040101E  RET
```

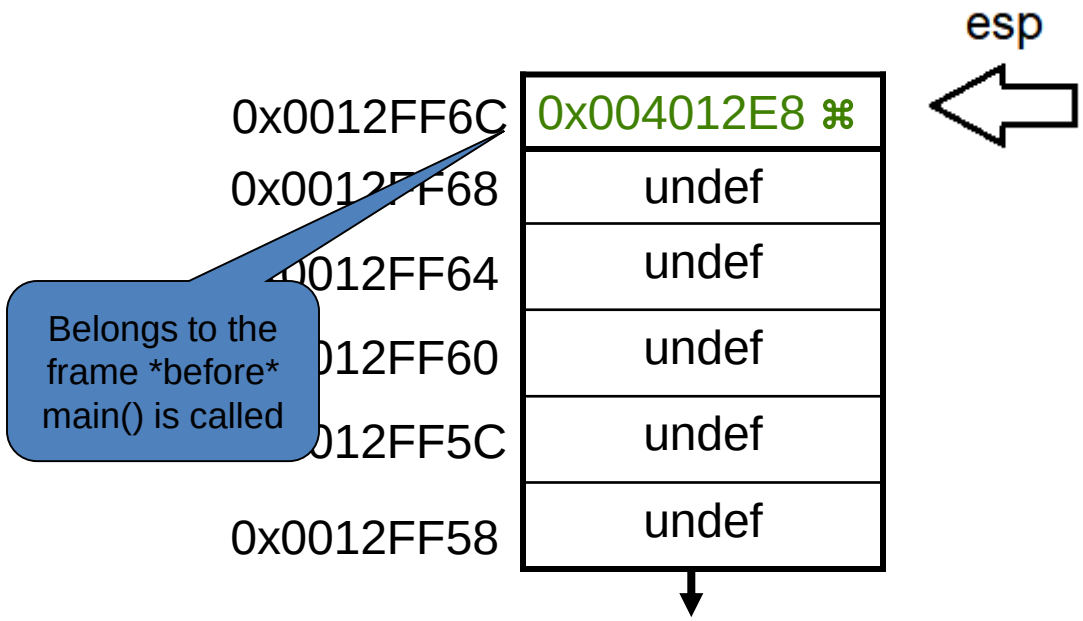
Example: Step-1

Assume: EIP = 00401010, but no instruction yet executed

eax	0x003435C0 ⌘
ebp	0x0012FFB8 ⌘
esp	0x0012FF6C ⌘

Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

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



Example: Step-2

Before

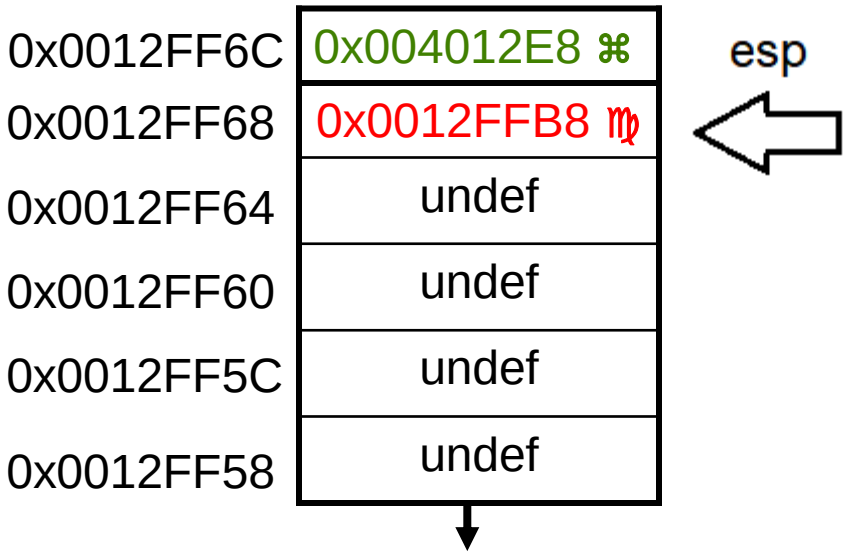
eax	0x003435C0 ⌘
ebp	0x0012FFB8 ⌘
esp	0x0012FF6C ⌘

Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

After

eax	0x003435C0
ebp	0x0012FFB8
esp	0x0012FF68 ♍

```
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
```



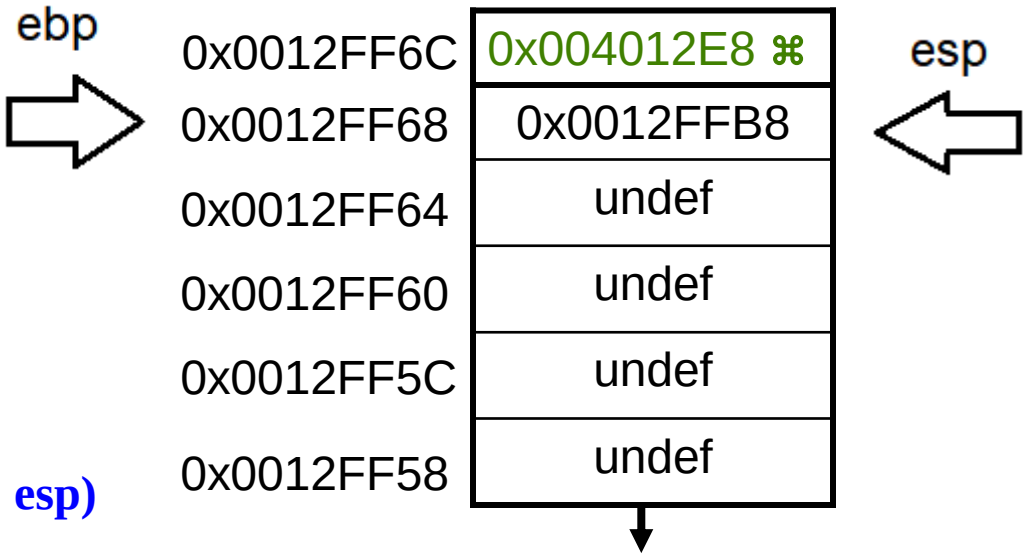
Example: Step-3

eax	0x003435C0 ⌘
ebp	0x0012FFB8 ⌘
esp	0x0012FF68 ⌘

Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

eax	0x003435C0 ⌘
ebp	0x0012FF68 ♍
esp	0x0012FF68

```
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 ☒  (ebp = esp)
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```



Example: Step-4

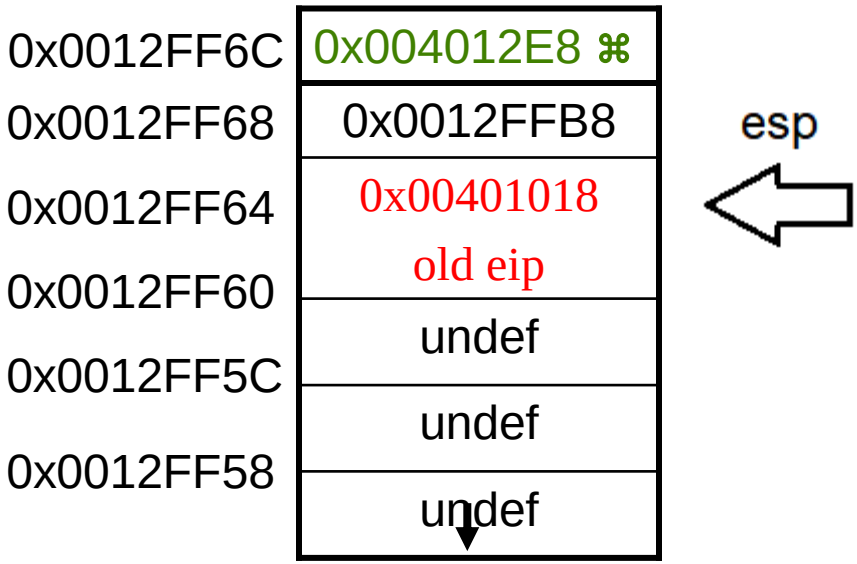
eax	0x003435C0 ⌘
ebp	0x0012FF68 ⌘
esp	0x0012FF68

Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

eax	0x003435C0 ⌘
ebp	0x0012FF68
esp	0x0012FF64 ♍

```
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
```

Save return address (old eip)



Example: Step-5

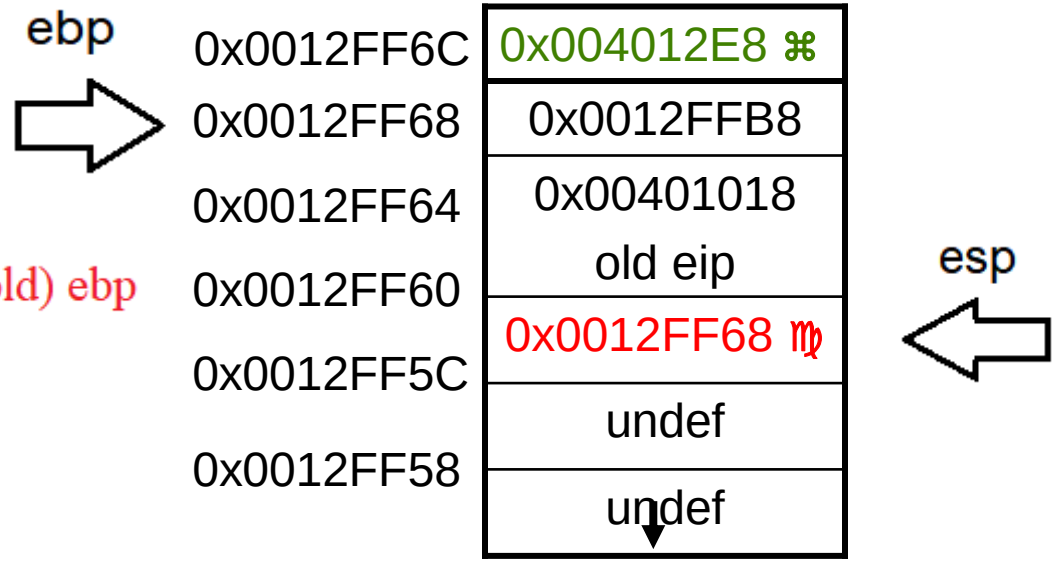
eax	0x003435C0 ⌘
ebp	0x0012FF68 ⌘
esp	0x0012FF64 ⌘

Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

eax	0x003435C0 ⌘
ebp	0x0012FF68
esp	0x0012FF60 ♍

```
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
```

Save caller (old) ebp



Example: Step-6

eax	0x003435C0 ⌘
ebp	0x0012FF68
esp	0x0012FF60 ⌘

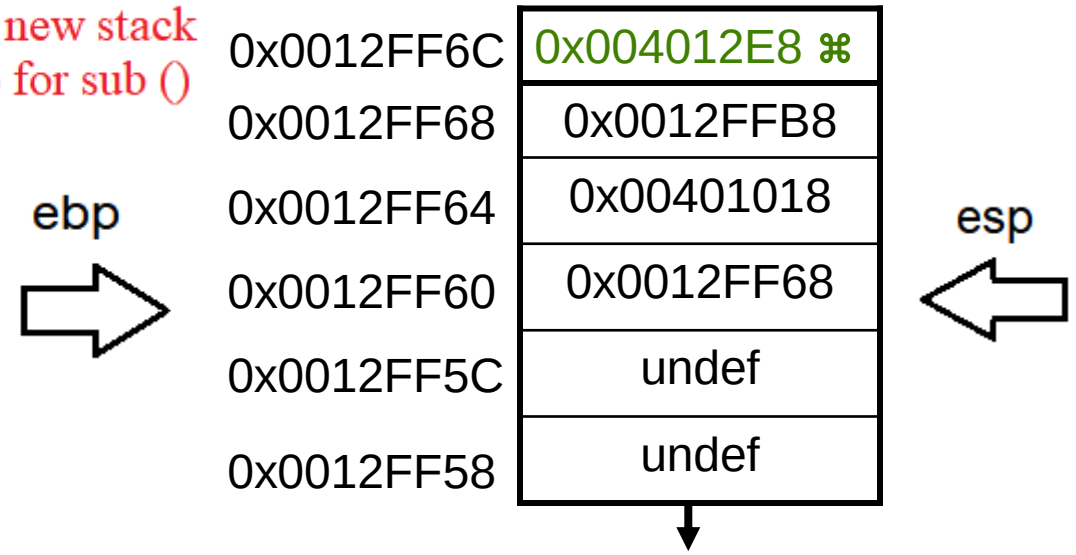
Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

eax	0x003435C0 ⌘
ebp	0x0012FF60 ♍
esp	0x0012FF60

```
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
```

eip = 00401003

Build new stack frame for sub ()



Example: Step-7

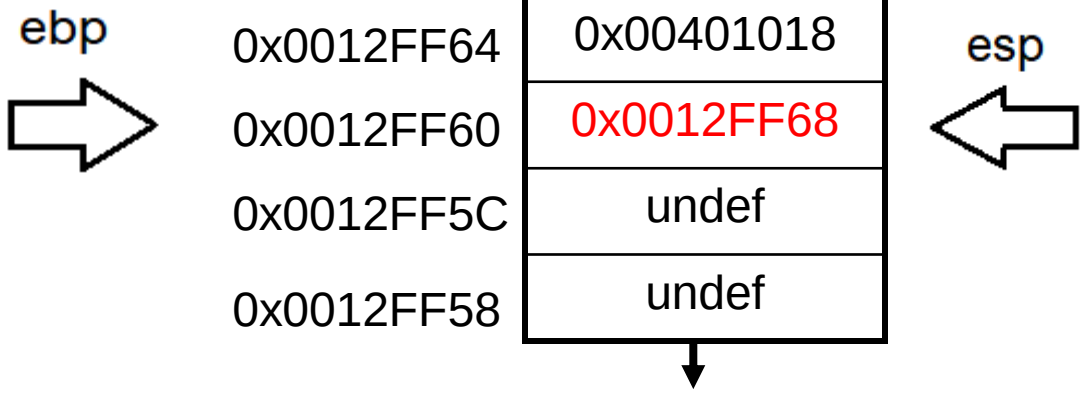
eax	0x003435C0 ⌘
ebp	0x0012FF60 ⌘
esp	0x0012FF60 ⌘

Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

eax	0x0000BEEF ♍
ebp	0x0012FF60
esp	0x0012FF60

```
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
```

eip= 00401008



Example: Step-8

eax	0x0000BEEF ⌘
ebp	0x0012FF60 ⌘
esp	0x0012FF60 ⌘

Key:
- ☒ **executed instruction,**
- ♍ **modified value**
- ⌘ **start value**

eax	0x0000BEEF
ebp	0x0012FF68 ♍
esp	0x0012FF64 ♍

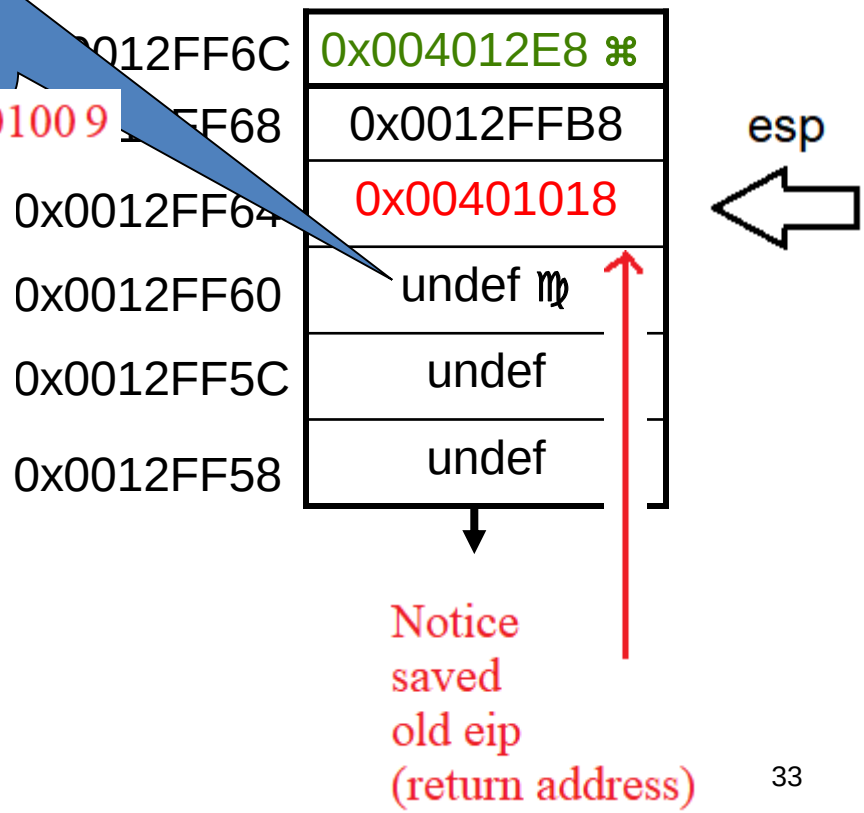
Note that a POP doesn't actually modify the value, but it should be considered undefined by prope

eip= 0040100 9

```
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
```

eip= 0040100 9

Notice that ebp got old value saved before calling sub()



Example: Step-9

eax	0x0000BEEF
ebp	0x0012FF68 ⌘
esp	0x0012FF64 ⌘

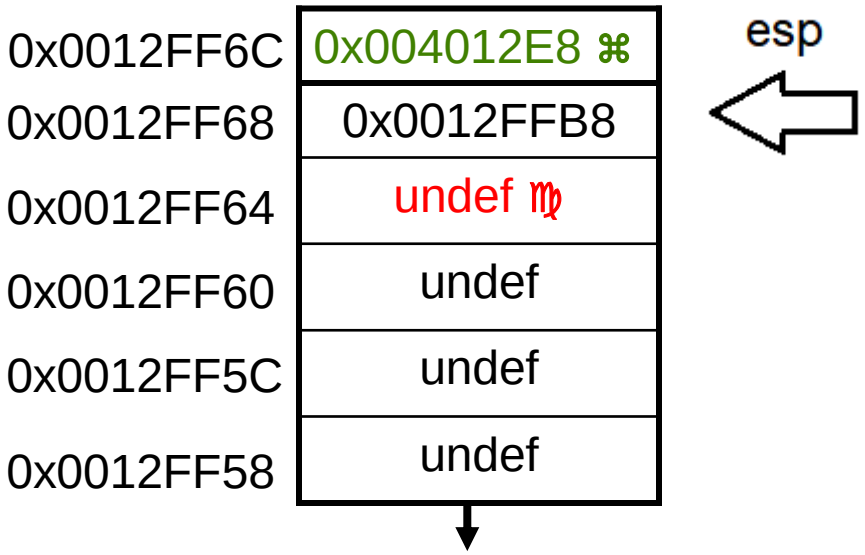
Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

eax	0x0000BEEF
ebp	0x0012FF68
esp	0x0012FF68 ♍

Notice that eip got old value saved before calling sub()

eip = 00401018

```
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
```



Example: Step-10

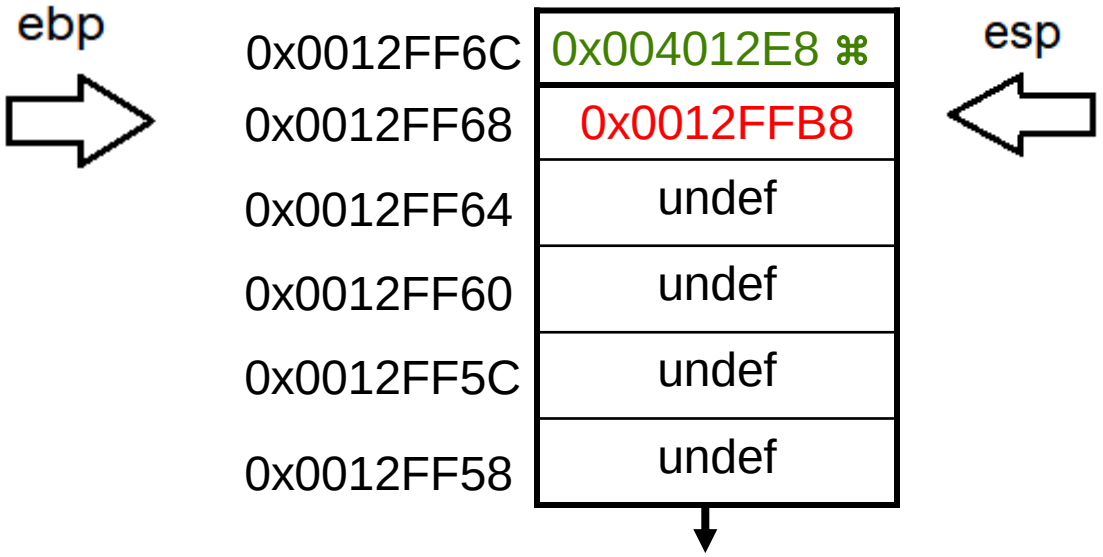
eax	0x0000BEEF⌘
ebp	0x0012FF68
esp	0x0012FF68

Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

eax	0x0000F00D ♍
ebp	0x0012FF68
esp	0x0012FF68

eip= 0040101D

```
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
```



Example: Step-11

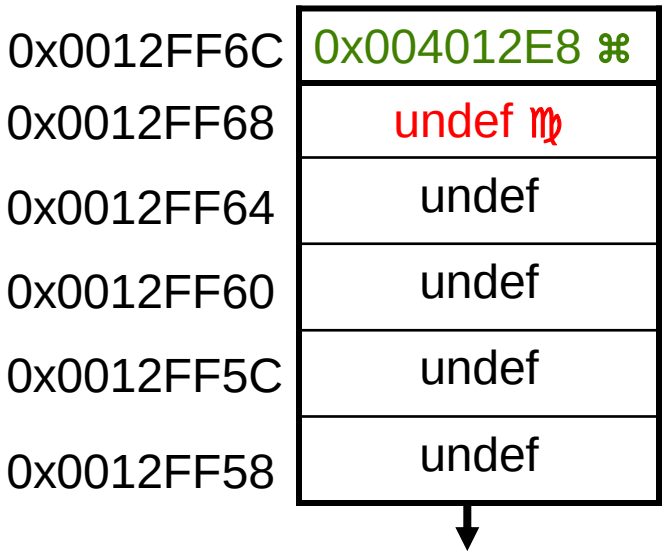
eax	0x0000F00D
ebp	0x0012FF68⌘
esp	0x0012FF68⌘

Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

eax	0x0000F00D
ebp	0x0012FFB8 ♍
esp	0x0012FF6C ♍

Notice that
ebp got old value
saved before
calling main ()

```
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
```



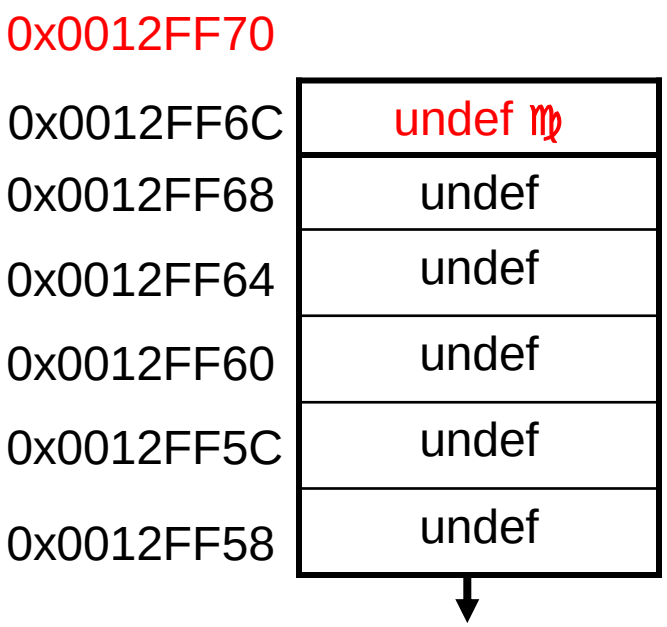
Example: Step-12

eax	0x0000F00D
ebp	0x0012FFB8 ⌘
esp	0x0012FF6C ⌘

Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

eax	0x0000F00D
ebp	0x0012FFB8
esp	0x0012FF70 ♍

```
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 ☒
```



Execution would continue at the value ret removed from the stack: 0x004012E8
New eip = 0x004012E8

Putting It All Together

You should know:

- **What is the Stack?**
- **Calling Function Procedure**
- **Assembly Language Instructions**
  - Understand “MOV” instruction
  - Understand “PUSH” instruction
  - Understand “POP” instruction
  - Understand “CALL” instruction
  - Understand “RET” instruction
  - Understand “NOP” instruction
  - Understand “LEA” instruction

Questions?

Coming Next Week
More Assembly Language Instructions

Week 10 Assignments

- **Learning Materials**

  1- Week 6 Presentation

  2- Read Pages 246-251 (Duntermann, Jeff. *Assembly Language Step by Step, Programming with Linux)*

  3- Read Pages 65-75 (PCASM textbook*)*

- **Assignment**

  1- Complete “Lab 10” by coming Sunday 11:59 PM.