x86 Intel Assembly - Registers and The Stack

Dan Flack

Roppers Academy - School of ROP

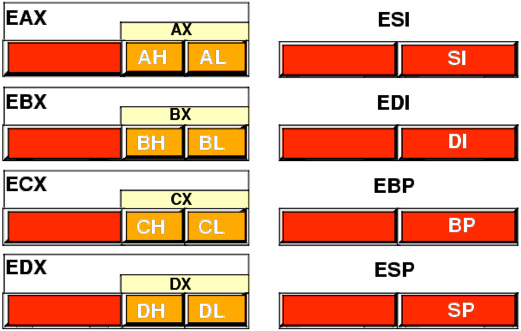
February 10, 2021

Objective

This tutorial/instruction slide-set is intended to provide a beginners look at registers and the stack as operated on by x86 instructions.

```
FunctionPrologue:
          push ebp       ; Comment
          mov  ebp, esp  ; Comment Two
          sub  esp, 8
```

32-bit Register Layout



32-bit Registers By Types

1. General Registers
   - EAX
   - EBX
   - ECX
   - EDX
2. Segment Registers
   - CS
   - DS
   - ES
   - FS
   - GS
   - SS
3. Index and Pointers
   - ESI
   - EDI
   - EBP
   - EIP
   - ESP
4. Indicator
   - EFLAGS

General Registers

General registers are used by most instructions in x86. These will be extremely common and can be broken down to 16 and 8 bit segments.

```
eax     ebx     ecx     edx     ; 32 bit registers
ax      bx      cx      dx      ; 16 bit registers
ah al   bh bl   ch cl   dh dl   ; 8 bit registers
```

The ‘h’ and ‘l’ suffixes on the 8 bit registers stands for the higher (h) and lower (l) bytes of the overall register.

Register Value Example

In the example below, assume `eax` contained the value 0 (zero) before the execution of the code below - for now, sign is ignored

```
add eax, 0xFF00h ;Add 65280 to eax                   Bits
;eax = 00000000 00000000 11111111 00000000             32
;ax  = 11111111 00000000                               16
;ah  = 11111111                                        8
;al  = 00000000                                        8
```

Segment Registers

For common reverse engineering the segment registers will not be of much interest. They are 16-bits in size and contain a 'segment selector.' A 'segment selector' is defined as a pointer to a place in memory where a segment exists.

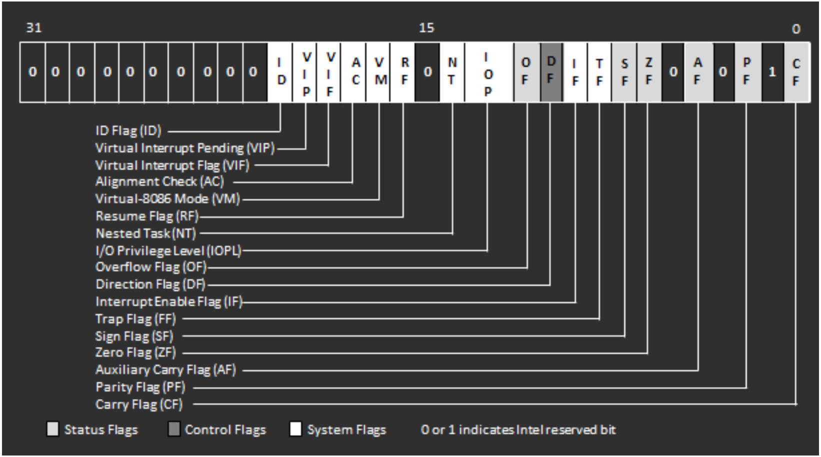
1. CS - Code segment - points to where the instructions are stored and executed
2. DS, ES, FS, GS - Points to the four data segments
3. SS - Points to the stack segment, where the procedure stack is stored

Pointers and Indexes

These registers are slightly misleading, the only register in this group that is classified separate from the General Registers is EIP. The other are used in convention for special purposes. EIP stores an offset to the next instruction to be executed. This register cannot be set by an application directly, it is set through implicit processes and functions.

EFLAGS Register

The EFLAGS register is different in operation than the others. It is 32-bits in size and stores a variety of 1-bit values, commonly called Flags.



Status Flags

The flags at bits 0, 2, 4, 6, 7, and 11 are called status flags. These are set after arithmetic to provide information about the 'status' of the computation. All flags are cleared if not set to 1 after a computation

1. Bit 0 - Carry Flag - Set to 1 is the computation generates a carry of the most significant bit(MSB). Indicates an overflow in unsigned arithmetic.
2. Bit 2 - Parity Flag - Set if the results least significant byte contains an even number of one bits.
3. Bit 4 - Aux. Carry Flag- Set if arithmetic generates a carry of bit 3 of the result - Used in Binary Coded Decimal arithmetic.
4. Bit 6 - Zero Flag - Set if the result is zero
5. Bit 7 - Sign Flag - Set equal to the MSB of the result (0 is positive, 1 is negative)
6. Bit 11 - Overflow Flag - Set if the result is too large of a positive number or too small of a negative number. Indicates overflow in signed arithmetic.

System and DF Flags

These flags generally control operating-system and executive operations. They are not usually modified by applications. The DF is the direction flag and is used in string operations. Unset, the DF string instructions will auto-increment (low-address to high-address) and with DF set string functions will auto-decrement (high-address to low-address).

Registers Overview

1. EAX - Commonly used for I/O Port Access, arithmetic, interrupts
2. EBX - Commonly used as memory access base pointer
3. ECX - Commonly used as a loop counter or a shift counter
4. EDX - Very similar to EAX in common usage
5. EDI - Used for string, memory array copying. Far pointer addressing
6. ESI - Used for string and memory array copying
7. EBP - Stores the stack base pointer
8. ESP - Stores a pointer to the top of the stack
9. EIP - Holds offset to next instruction

Stack overview

What is a stack?

- Data Structure
- FILO - First In, Last Out

Why?

Computers use the stack to keep track of relevant data and addresses during execution. Understanding the stack will be essential in reverse engineering programs.

Stack overview

What is a stack?

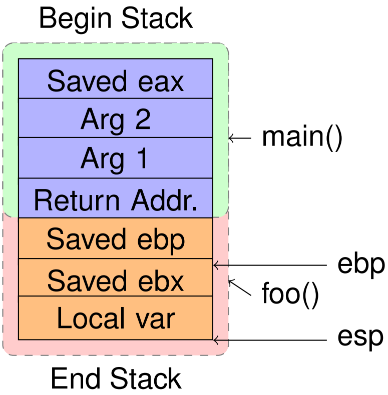
- Data Structure
- FILO - First In, Last Out

Why?

Computers use the stack to keep track of relevant data and addresses during execution. Understanding the stack will be essential in reverse engineering programs.

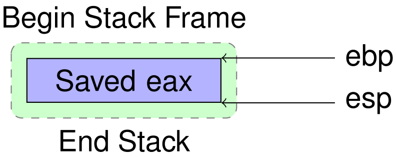
In other words, items are pushed onto the stack and then popped off of the top of the stack. The most recently added item will be the item that is popped.

CDECL Stack Arrangement



CDECL is called a caller clean-up convention. The calling function is responsible for pushing arguments onto the stack resizing the stack after the called function returns. This slide-set uses CDECL as it is the most common.

Stack Example - Part 1

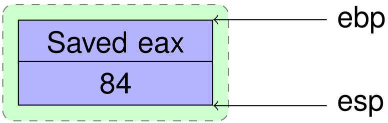


The Code

```
main:
    push eax      ;<--Here
    push 84
    push 21
    call foo
    add esp, 12
foo:
    push ebp
    mov ebp, esp
    push ebx
    sub esp, 4
; Some stuff here
    mov eax, 1337
    add esp, 4
    pop ebx
    pop ebp
    ret
```

Stack Example - Part 2

Begin Stack Frame

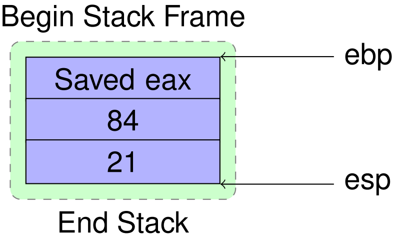


End Stack

The Code

```
main:
    push eax
    push 84       ;<--Here
    push 21
    call foo
    add esp, 12
foo:
    push ebp
    mov ebp, esp
    push ebx
    sub esp, 4
; Some stuff here
    mov eax, 1337
    add esp, 4
    pop ebx
    pop ebp
    ret
```

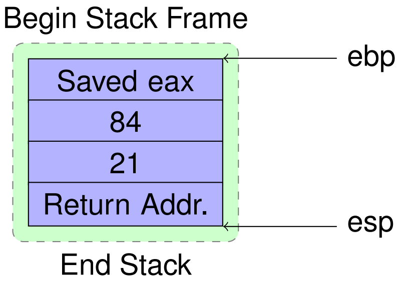
Stack Example - Part 3



The Code

```
main:
    push eax
    push 84
    push 21         ;<--Here
    call foo
    add esp, 12
foo:
    push ebp
    mov ebp, esp
    push ebx
    sub esp, 4
; Some stuff here
    mov eax, 1337
    add esp, 4
    pop ebx
    pop ebp
    ret
```

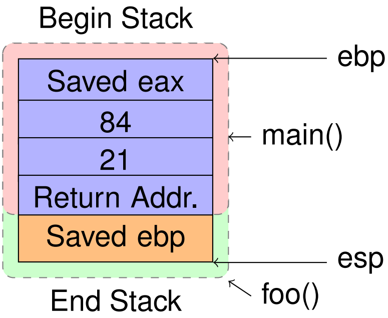
Stack Example - Part 4



The Code

```
main:
    push eax
    push 84
    push 21
    call foo     ;<--Here
    add esp, 12
foo:
    push ebp
    mov ebp, esp
    push ebx
    sub esp, 4
; Some stuff here
    mov eax, 1337
    add esp, 4
    pop ebx
    pop ebp
    ret
```

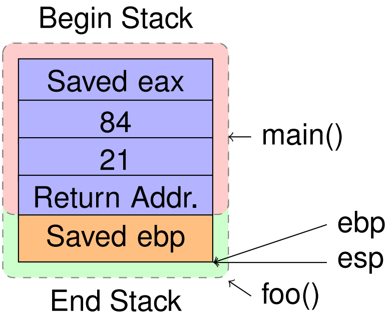
Stack Example - Part 5



The Code

```
main:
    push eax
    push 84
    push 21
    call foo
    add esp, 12
foo:
    push ebp      ;<--Here
    mov ebp, esp
    push ebx
    sub esp, 4
; Some stuff here
    mov eax, 1337
    add esp, 4
    pop ebx
    pop ebp
    ret
```

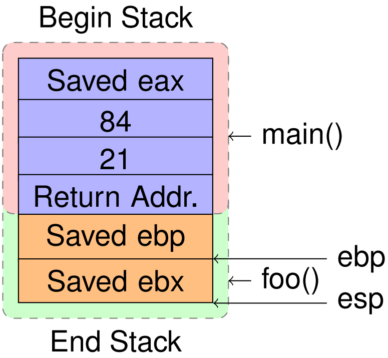
Stack Example - Part 6



The Code

```
main:
    push eax
    push 84
    push 21
    call foo
    add esp, 12
foo:
    push ebp
    mov ebp, esp ;<--Here
    push ebx
    sub esp, 4
; Some stuff here
    mov eax, 1337
    add esp, 4
    pop ebx
    pop ebp
    ret
```

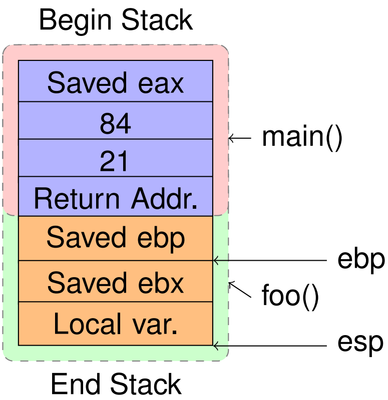
Stack Example - Part 7



The Code

```
main:
    push eax
    push 84
    push 21
    call foo
    add esp, 12
foo:
    push ebp
    mov ebp, esp
    push ebx        ;<--Here
    sub esp, 4
; Some stuff here
    mov eax, 1337
    add esp, 4
    pop ebx
    pop ebp
    ret
```

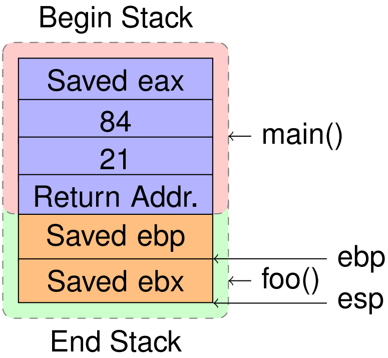
Stack Example - Part 8



The Code

```
main:
    push eax
    push 84
    push 21
    call foo
    add esp, 12
foo:
    push ebp
    mov ebp, esp
    push ebx
    sub esp, 4    ;<--Here
; Some stuff here
    mov eax, 1337
    add esp, 4
    pop ebx
    pop ebp
    ret
```

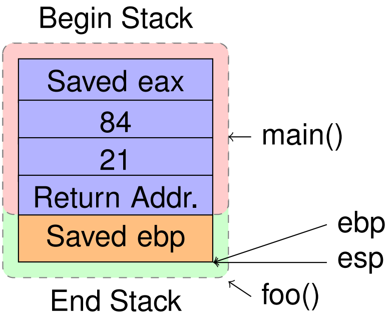
Stack Example - Part 9



The Code

```
main:
    push eax
    push 84
    push 21
    call foo
    add esp, 12
foo:
    push ebp
    mov ebp, esp
    push ebx
    sub esp, 4
; Some stuff here
    mov eax, 1337
    add esp, 4    ;<--Here
    pop ebx
    pop ebp
    ret
```

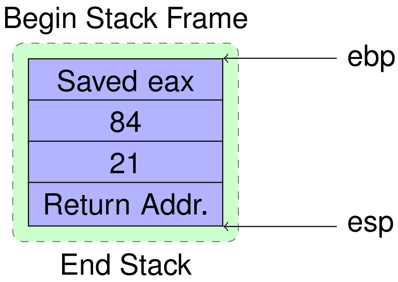
Stack Example - Part 10



The Code

```
main:
    push eax
    push 84
    push 21
    call foo
    add esp, 12
foo:
    push ebp
    mov ebp, esp
    push ebx
    sub esp, 4
; Some stuff here
    mov eax, 1337
    add esp, 4
    pop ebx          ;<--Here
    pop ebp
    ret
```

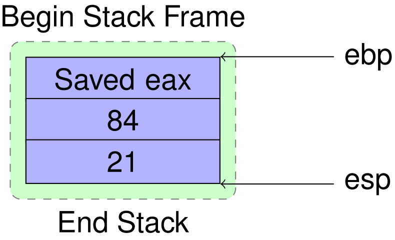
Stack Example - Part 11



The Code

```
main:
    push eax
    push 84
    push 21
    call foo
    add esp, 12
foo:
    push ebp
    mov ebp, esp
    push ebx
    sub esp, 4
; Some stuff here
    mov eax, 1337
    add esp, 4
    pop ebx
    pop ebp         ;<--Here
    ret
```

Stack Example - Part 12



The Code

```
main:
    push eax
    push 84
    push 21
    call foo
    add esp, 12
foo:
    push ebp
    mov ebp, esp
    push ebx
    sub esp, 4
; Some stuff here
    mov eax, 1337
    add esp, 4
    pop ebx
    pop ebp
    ret             ;<--Here
```

Stack Example - Part 13

The program is now finished. If there was a `ret`, the program would have continued backward into the next stack frame. Check out slide-set 2 to learn more about the x86 instructions and what they are used for.

The Code

```
main:
    push eax
    push 84
    push 21
    call foo
    add esp, 12  ;<--Here
foo:
    push ebp
    mov ebp, esp
    push ebx
    sub esp, 4
; Some stuff here
    mov eax, 1337
    add esp, 4
    pop ebx
    pop ebp
    ret
```