CSC 21100: Fundamentals of Computer Systems x86 Assembly

Instructor: Zheng Peng
Assistant Professor
Computer Science Department
City College of New York

This lecture slides contain materials from

- Assembly tutorial by Professor Ray Toal, Electrical Engineering & Computer Science, Loyola Marymount University
- University of Virginia, Computer Science, CS216: Program and Data Representation, Spring 2006, by Professor David Evans
- The x86 PC Assembly Language, Design and Interfacing, by Muhammad Ali Mazidi, Janice Gillespie Mazidi and Danny Causey
- NASM Assembly Language Tutorials - asmtutor.com

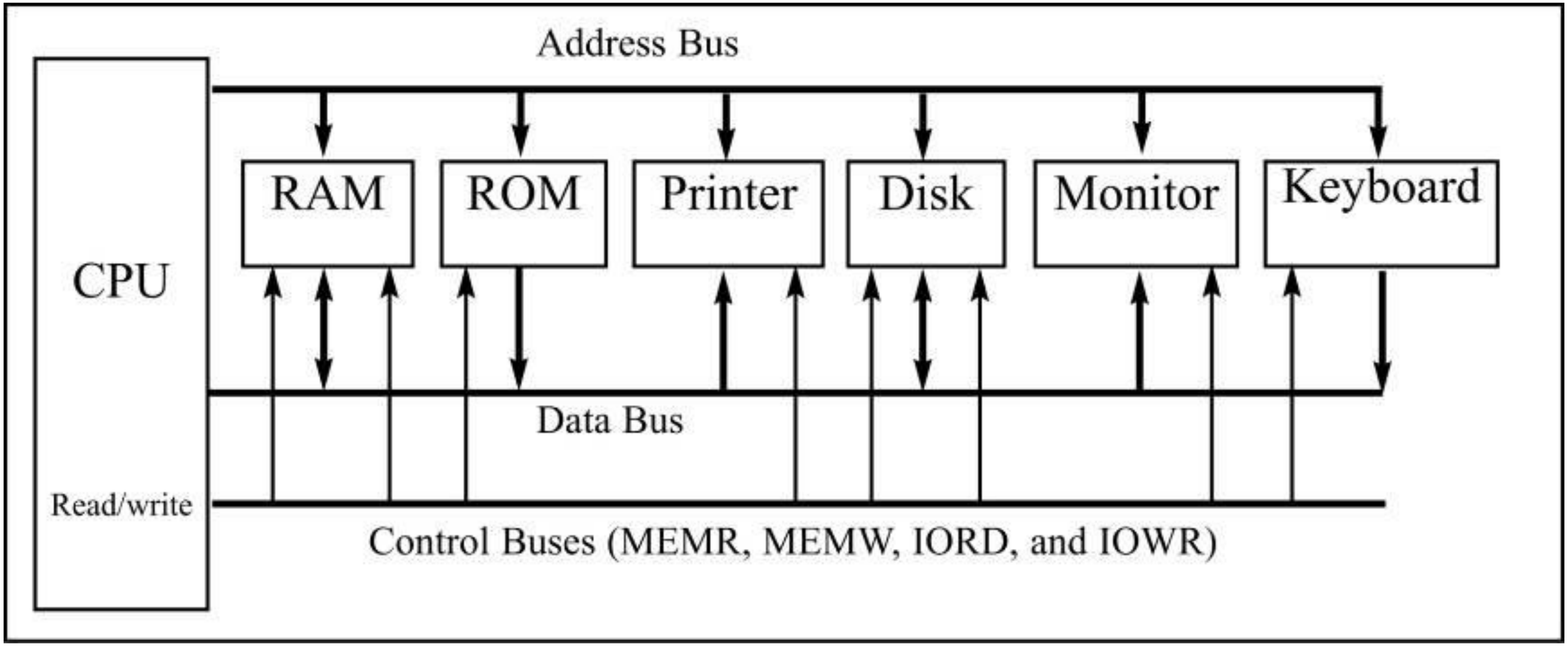
x86 Architecture Overview

- x86 is a family of instruction set architectures (ISA) based on the Intel 8086 CPU and its Intel 8088 variant.
- The architecture has been implemented in processors from Intel, Cyrix, AMD, VIA and many other companies.
- Currently, the majority of personal computers and laptops sold are based on the x86 architecture.
- x86 also dominates compute-intensive workstation and cloud computing segments.

Processor Modes

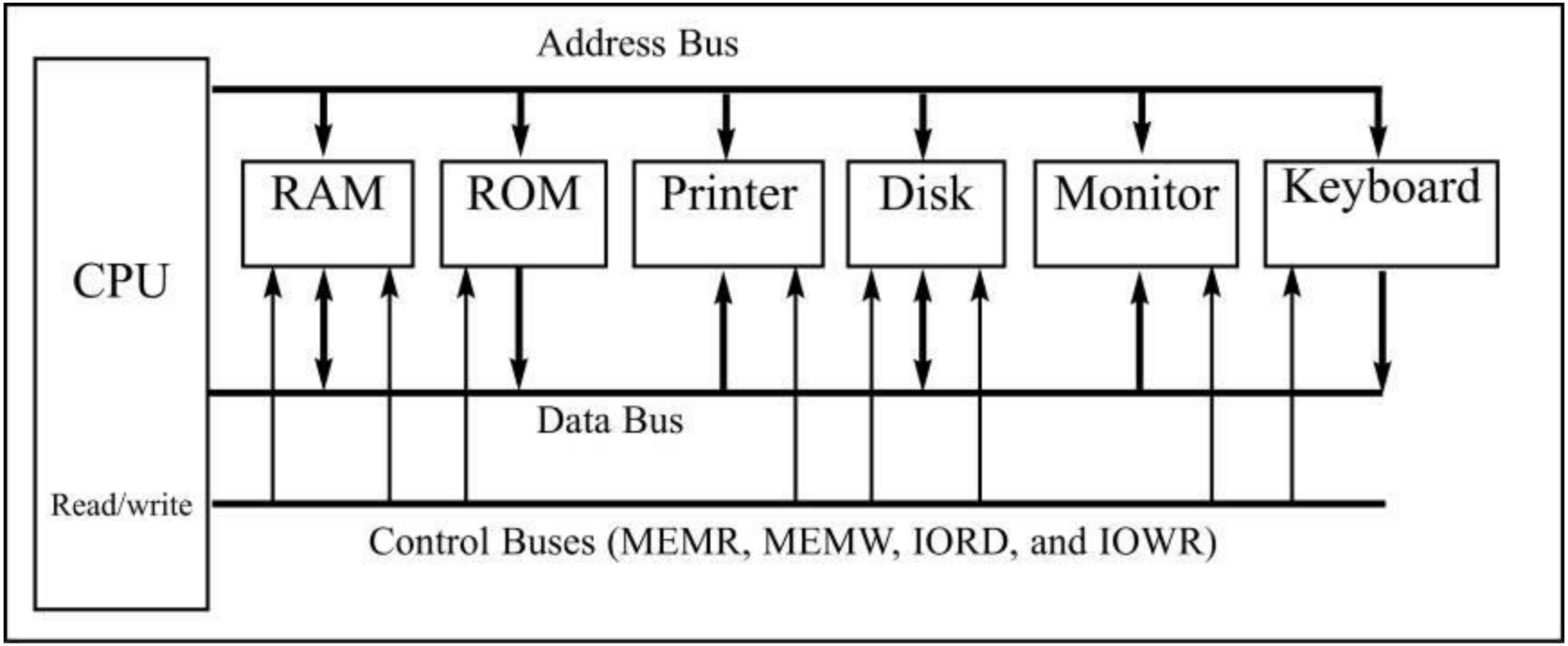
- A processor implementing the x86 architecture can execute in:
  - Real Mode: A highly restricted operating mode in which only the first 1 MB of physical memory is directly accessible.
  - Protected Mode: The normal mode of operation. This mode expands addressable physical memory to 16 MB and addressable virtual memory to 1 GB, and provides protected memory.
  - Other Modes: Not discussed in this class.

Inside a Computer (1)



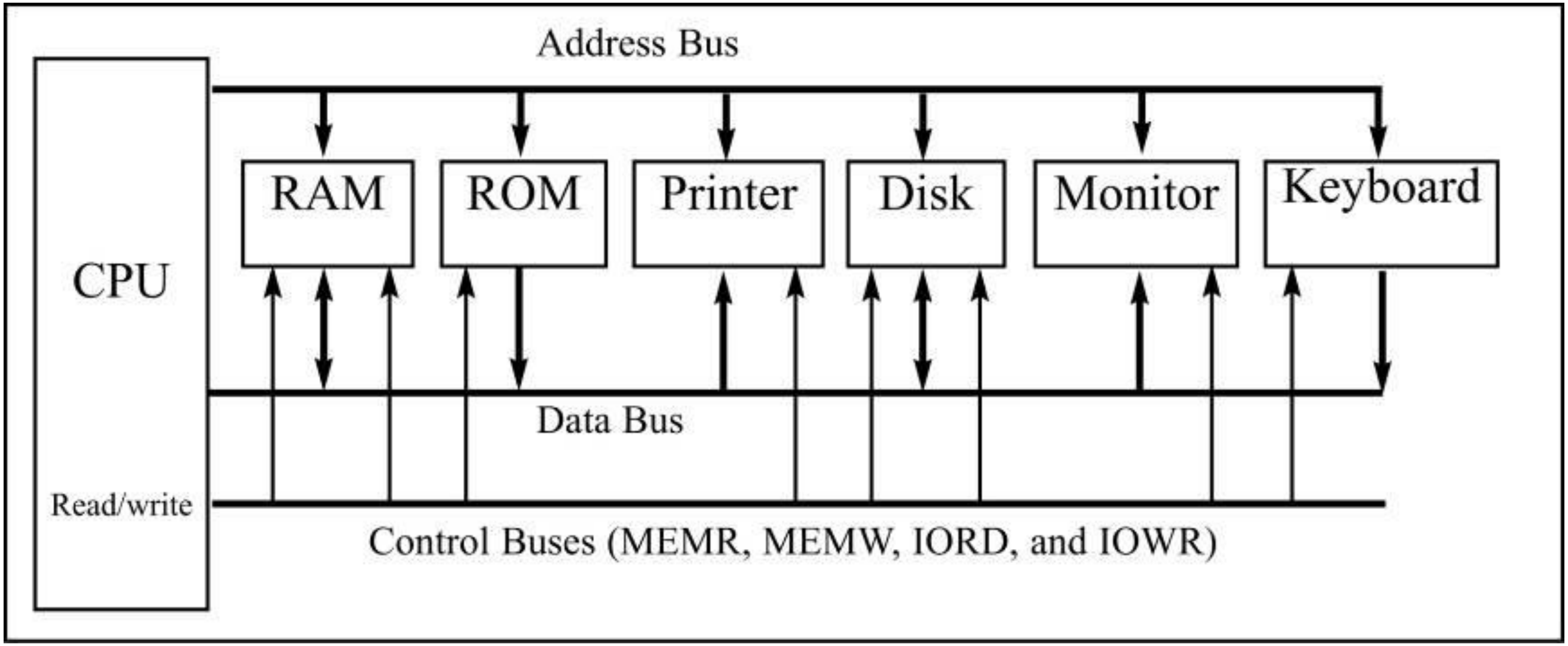
- CPU: “central processing unit” (also called a processor)
  - CPU function is to execute (process) information stored in memory.
  - I/O devices, such as keyboard & monitor provide a means of communicating with the CPU.
  - The CPU is connected to memory and I/O through a group of wires called a bus.

Inside a Computer (2)



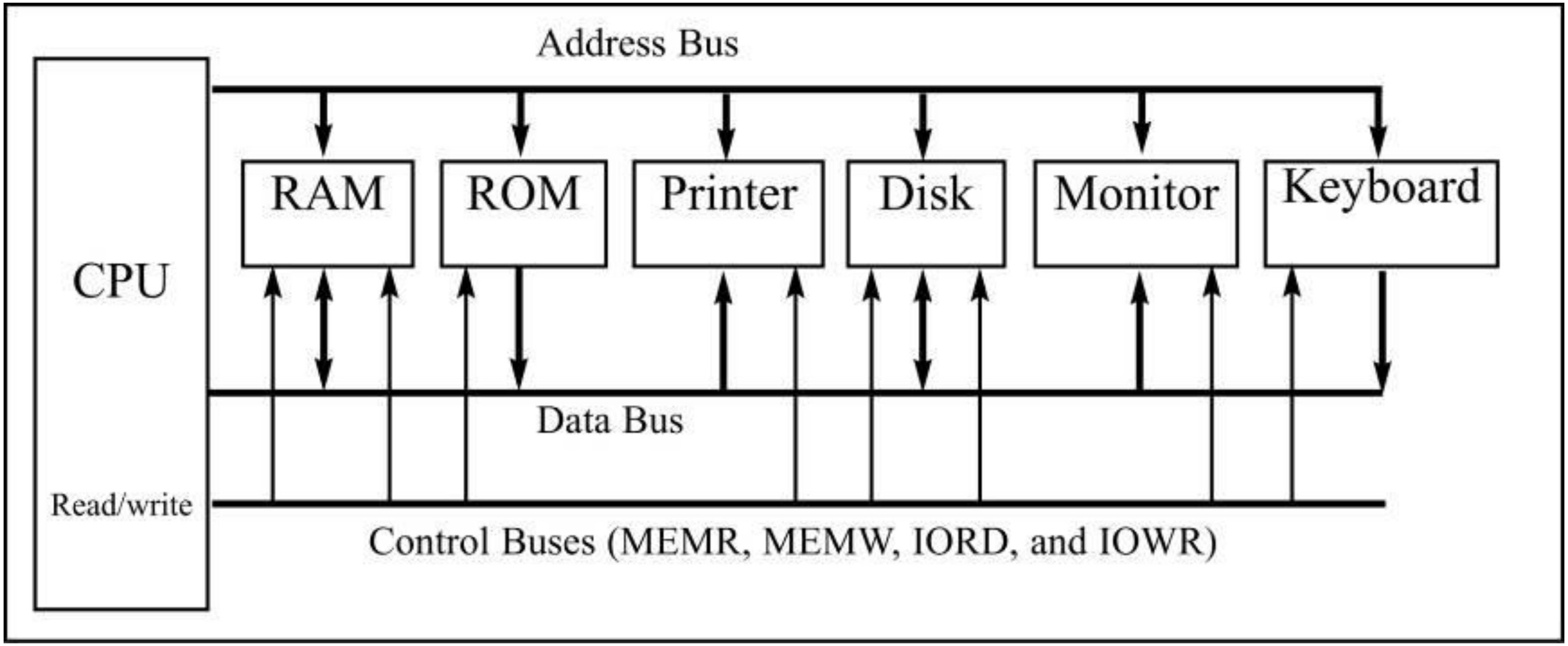
- Bus:
  - Data bus: it carries information in/out of a CPU
  - Address bus: it is used to identify devices and memory connected to the CPU
  - Control bus: it provides device control signals.

Inside a Computer (3)



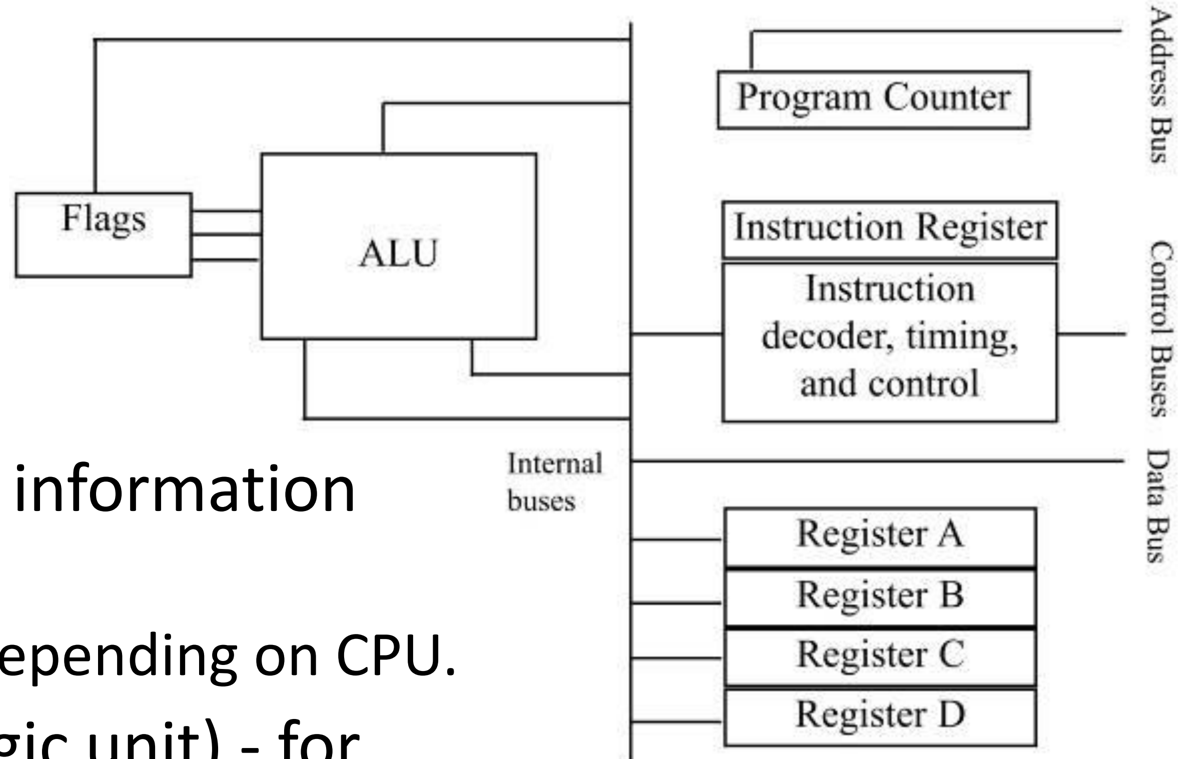
- RAM: “random access memory” (sometimes called read/write memory)
  - Used for temporary storage of programs while running.
  - Data is lost when the computer is turned off.
  - RAM is sometimes called volatile memory.

Inside a Computer (4)



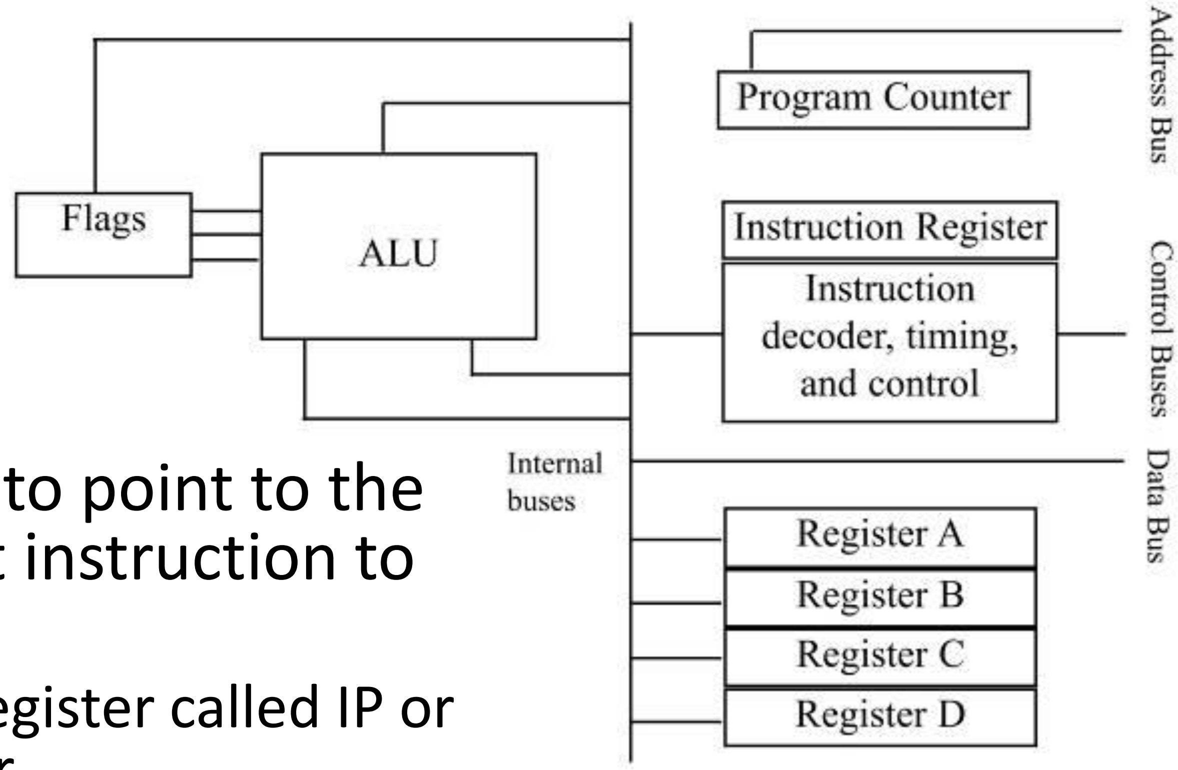
- ROM: "read-only memory"
  - Contains programs and information essential to the operation of the computer.
  - Information in ROM is permanent, cannot be changed by the user, and is not lost when the power is turned off.
  - ROM is called nonvolatile memory.

Inside a CPU (1)



- **Registers** - to store information temporarily.
  - 8, 16, 32, 64 bit, depending on CPU.
- **ALU** (arithmetic/logic unit) - for arithmetic functions such as add, subtract, multiply, and divide.
  - Also logic functions such as AND, OR, and NOT.

Inside a CPU (2)



- **Program counter** - to point to the address of the next instruction to be executed.
  - In the IBM PC, a register called IP or instruction pointer.
- **Instruction decoder** - to interpret the instruction fetched into the CPU.

CPU at Work (1)

- A step-by-step analysis of CPU processes to add three numbers, with steps & code shown.
  - Assume a CPU has registers A, B, C, and D.
    - An 8-bit data bus and a 16-bit address bus.
  - The CPU can access memory addresses 0000 to FFFFH.
    - A total of 10000H locations.

Action	***Code***	***Data***
Move value 21H into register A	B0H	21H
Add value 42H to register A	04H	42H
Add value 12H to register A	04H	12H

CPU at Work (2)

- If the program to perform the actions listed above is stored in memory locations starting at 1400H, the following would represent the contents for each memory address location…

Memory address	***Contents of memory address***
1400	(B0) code for moving a value to register A
1401	(21) value to be moved
1402	(04) code for adding a value to register A
1403	(42) value to be added
1404	(04) code for adding a value to register A
1405	(12) value to be added
1406	(F4) code for halt

CPU at Work (3)

- The CPU puts the address 1400H on the address bus and sends it out.
  - Memory finds the location while the CPU activates the READ signal, indicating it wants the byte at 1400H.
    - The content (B0) is put on the data bus & brought to the CPU.

Memory address	***Contents of memory address***
1400	(B0) code for moving a value to register A
1401	(21) value to be moved
1402	(04) code for adding a value to register A
1403	(42) value to be added
1404	(04) code for adding a value to register A
1405	(12) value to be added
1406	(F4) code for halt

CPU at Work (4)

- The CPU decodes the instruction B0 with the help of its instruction decoder dictionary.
  - Bring the byte of the next memory location into CPU Register A.

Memory address	***Contents of memory address***
1400	(B0) code for moving a value to register A
1401	(21) value to be moved
1402	(04) code for adding a value to register A
1403	(42) value to be added
1404	(04) code for adding a value to register A
1405	(12) value to be added
1406	(F4) code for halt

CPU at Work (5)

- From memory location 1401H, the CPU fetches code 21H directly to Register A.
  - After completing the instruction, the program counter points to the address of the next instruction - 1402H.
    - Address 1402H is sent out on the address bus, to fetch the next instruction.

Memory address	***Contents of memory address***
1400	(B0) code for moving a value to register A
1401	(21) value to be moved
1402	(04) code for adding a value to register A
1403	(42) value to be added
1404	(04) code for adding a value to register A
1405	(12) value to be added
1406	(F4) code for halt

CPU at Work (6)

- From 1402H, the CPU fetches code 04H.
  - After decoding, the CPU knows it must add the byte at the next address (1403) to the contents of register A.
    - After it brings the value (42H) into the CPU, it provides the contents of Register A, along with this value to the ALU to perform the addition.
    - Program counter becomes 1404, the next instruction address.

Memory address	***Contents of memory address***
1400	(B0) code for moving a value to register A
1401	(21) value to be moved
1402	(04) code for adding a value to register A
1403	(42) value to be added
1404	(04) code for adding a value to register A
1405	(12) value to be added
1406	(F4) code for halt

CPU at Work (7)

- Address 1404H is put on the address bus and the code is fetched, decoded, and executed.
  - Again adding a value to Register A.
    - The program counter is updated to 1406H

Memory address	***Contents of memory address***
1400	(B0) code for moving a value to register A
1401	(21) value to be moved
1402	(04) code for adding a value to register A
1403	(42) value to be added
1404	(04) code for adding a value to register A
1405	(12) value to be added
1406	(F4) code for halt

CPU at Work (8)

- The contents of address 1406 (HALT code) are fetched in and executed.
  - The HALT instruction tells the CPU to stop incrementing the program counter and asking for the next instruction.
    - Without HALT, the CPU would continue updating the program counter and fetching instructions.

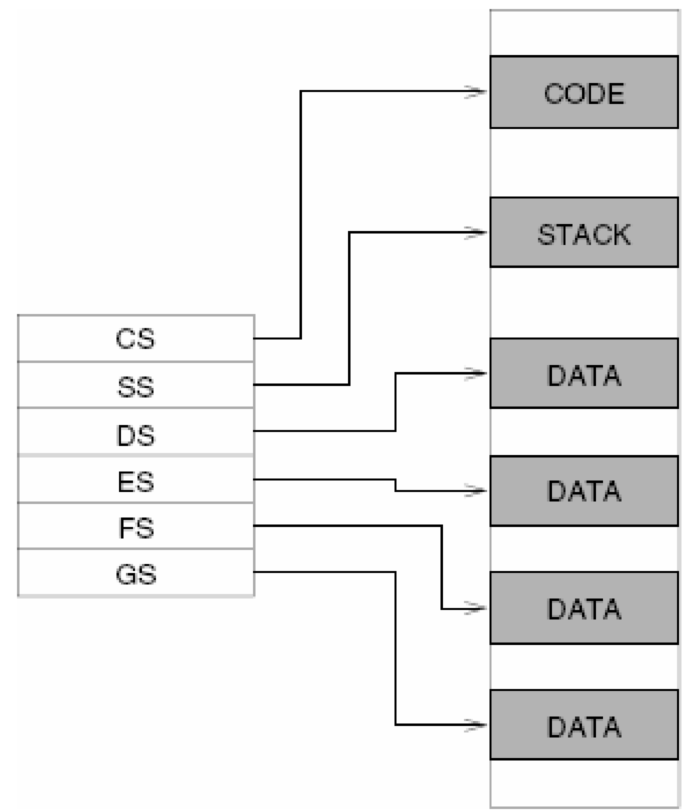
Memory address	*Contents of memory address*
1400	(B0) code for moving a value to register A
1401	(21) value to be moved
1402	(04) code for adding a value to register A
1403	(42) value to be added
1404	(04) code for adding a value to register A
1405	(12) value to be added
1406	(F4) code for halt

x86 Segments

- A typical Assembly language program consists of at least three segments:
  - A **code segment** - which contains the assembly language instructions that perform the tasks that the program was designed to accomplish.
  - A **data segment** - used to store information (data) to be processed by the instructions in the code segment. There may be many data segments.
  - A **stack segment** - used by the CPU to store information temporarily.
  - An <u>optional</u> **extra segment** - a spare segment that may be used for specifying a location in memory.

x86 Segment Registers

- All x86 segment registers are 16 bits in size, irrespective of the CPU:
  - CS: code segment register
  - DS: data segment register
  - SS: stack segment register
  - ES: extra segment register
  - FS and GS: clones of ES



Addressing Memory

- A memory reference has four parts and is often written as [SELECTOR : BASE + INDEX * SCALE + OFFSET]. Example: [FS:ECX+ESI*8+93221]
  - The selector is one of the six segment registers;
  - The base is one of the eight general purpose registers;
  - The index is any of the general purpose registers except ESP;
  - The scale is 0, 1, 2, 4, or 8;
  - The offset is any 32-bit number.
- The minimal reference consists of only a base register or only an offset; a scale can only appear if there is an index present.
- Sometimes the memory reference is written like this:

```
selector
offset(base,index,scale)
```

x86 Data Types

Type name	Number of bits	Bit indices
Byte	8	7..0
Word	16	15..0
Doubleword	32	32..0
Quadword	64	63..0
Doublequadword	128	127..0

Endianness

- The sequential order used to store a word in computer memory
  - Big Endian: Most-significant byte at least address of a word
  - Little Endian: Least-significant byte at least address
- As an example, suppose we have a hexadecimal number 0x12345678.

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

- The x86 is little endian, meaning the least significant bytes come first in memory

First Assembly Program

- "Hello World!"
  - Assembly version

```
; ----------------------------------------------------------------------------------------
; Writes "Hello, World" to the console using only system calls.
; To assemble and run:
;
;     nasm -f elf hello.asm & ld -m elf_i386 hello.o -o hello
; ----------------------------------------------------------------------------------------

        global    _start

        section   .text
_start: mov       eax, 4                  ; system call number for write
        mov       ebx, 1                  ; file handle 1 is stdout
        mov       ecx, message            ; address of string to output
        mov       edx, 13                 ; number of bytes
        int 80h                           ; request an interrupt on libc using INT 80h
exit:   mov       eax, 1                  ; syscam call number for exit
        mov       ebx, 0                  ; return 0 status on exit - 'No Errors'
        int 80h                           ; request an interrupt on libc using INT 80h

        section   .data
message: db       "Hello, World", 0Ah      ; note the newline at the end
```

First Assembly Program

- Most programs consist of **directives** followed by one or more **sections**.

```
; ----------------------------------------------------------------------------------------
; Writes "Hello, World" to the console using only system calls.
; To assemble and run:
;
;     nasm -f elf hello.asm & ld -m elf_i386 hello.o -o hello
; ----------------------------------------------------------------------------------------

        global    _start

        section   .text
_start: mov       eax, 4                  ; system call number for write
        mov       ebx, 1                  ; file handle 1 is stdout
        mov       ecx, message            ; address of string to output
        mov       edx, 13                 ; number of bytes
        int 80h                           ; request an interrupt on libc using INT 80h
exit:   mov       eax, 1                  ; syscam call number for exit
        mov       ebx, 0                  ; return 0 status on exit - 'No Errors'
        int 80h                           ; request an interrupt on libc using INT 80h

        section   .data
message: db       "Hello, World", 0Ah      ; note the newline at the end
```

Directives

Sections

First Assembly Program

- Lines can have an optional **label**. Most lines have an **instruction** followed by zero or more **operands**.

```
; ----------------------------------------------------------------------------------------
; Writes "Hello, World" to the console using only system calls.
; To assemble and run:
;
;     nasm -f elf hello.asm & ld -m elf_i386 hello.o -o hello
; ----------------------------------------------------------------------------------------

          global    _start

          section   .text
_start:   mov       eax, 4                  ; system call number for write
          mov       ebx, 1                  ; file handle 1 is stdout
          mov       ecx, message            ; address of string to output
          mov       edx, 13                 ; number of bytes
          int 80h                           ; request an interrupt on libc using INT 80h
exit:     mov       eax, 1                  ; syscam call number for exit
          mov       ebx, 0                  ; return 0 status on exit - 'No Errors'
          int 80h                           ; request an interrupt on libc using INT 80h

          section   .data
message:  db        "Hello, World", 0Ah      ; note the newline at the end
```

Labels Instructions Operands

Assemble and Run

```
zheng@Hudson:~/Assembly$
zheng@Hudson:~/Assembly$
zheng@Hudson:~/Assembly$ nasm -f elf hello.asm
zheng@Hudson:~/Assembly$ ls hello.o
hello.o
zheng@Hudson:~/Assembly$ ld -m elf_i386 hello.o -o hello
zheng@Hudson:~/Assembly$ ls hello
hello
zheng@Hudson:~/Assembly$ ./hello
Hello, World
zheng@Hudson:~/Assembly$
```

Your First Few Instructions

There are hundreds of instructions. You can't learn them all at once. Just start with these:

mov *x*, *y*	$x \leftarrow y$
and *x*, *y*	$x \leftarrow x$ and y
or *x*, *y*	$x \leftarrow x$ or y
xor *x*, *y*	$x \leftarrow x$ xor y
add *x*, *y*	$x \leftarrow x + y$
sub *x*, *y*	$x \leftarrow x - y$
inc *x*	$x \leftarrow x + 1$
dec *x*	$x \leftarrow x - 1$
int 80h	Request an interrupt on libc using INT 80h
db	A pseudo-instruction that declares bytes that will be in memory when the program runs

Complete list of instructions can be found here:

- https://software.intel.com/en-us/articles/intel-sdm

System Call Table

- System call: a special type of function call, in which a computer program requests a service from the kernel of the operating system it is executed on.
- The following table lists some of the system calls for the Linux kernel. It could also be thought of as an API for the interface between user space and kernel space.

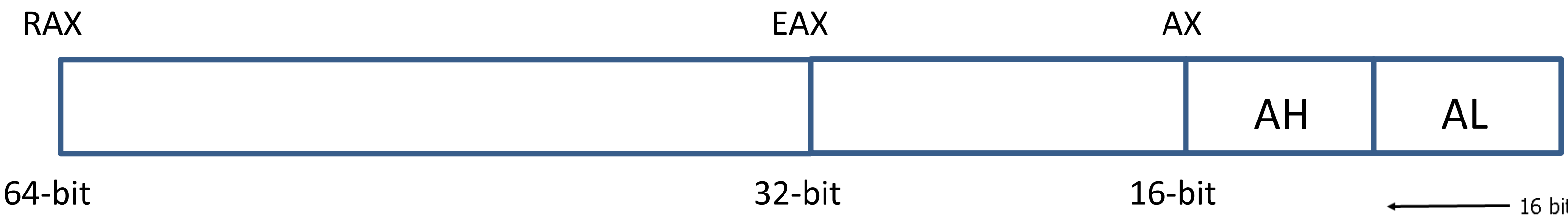
#	Name	Register			
		EAX	EBX	ECX	EDX
1	sys_exit	0x01	-	-	-
3	sys_read	0x03	unsigned int fd	char *buf	size_t count
4	sys_write	0x04	unsigned int fd	const char *buf	size_t count
5	sys_open	0x05	const char *name	int flags	int mode
6	sys_close	0x06	unsigned int fd	-	-

Complete list of Linux system calls can be found here:

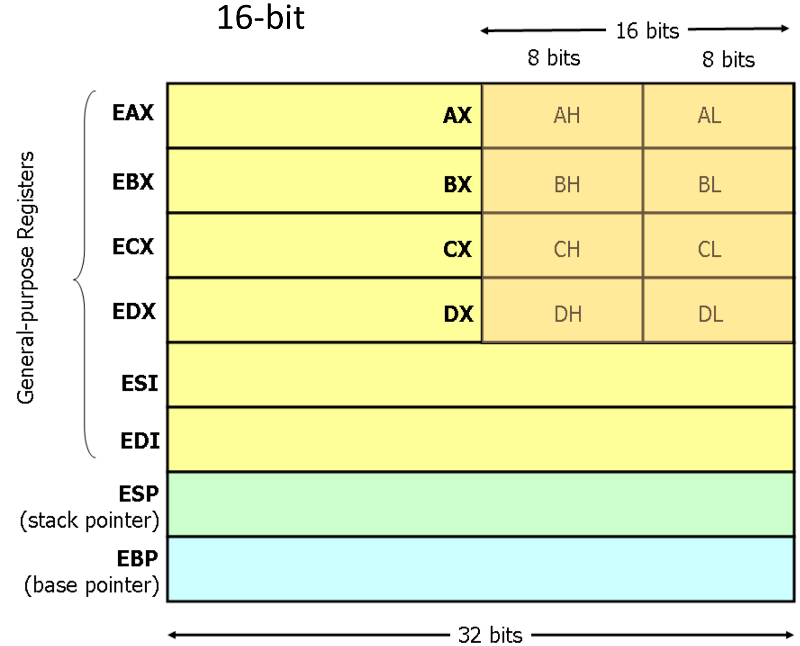
- http://shell-storm.org/shellcode/files/syscalls.html

The Three Kinds of Operands (1)

- Register Operands
  - General-purpose registers in x86 processors can be accessed as either 32-bit, 16-bit or 8-bit registers



  - Different purposes
    - A register is used for the accumulator.
    - B register is a base addressing register.
    - C register is a counter in loop operations.
    - D register points to data in I/O operations
    - ESI/EDI are index registers



The Three Kinds of Operands (2)

- Memory Operands
  - These are the basic forms of addressing:
    - [number]
    - [reg]
    - [reg + reg*scale]
    - [reg + number]
    - [reg + reg*scale + number]
  - The number is called the displacement; the plain register is called the base; the register with the scale is called the index.
  - Example

```
[750]                  ; displacement only
[ebp]                  ; base register only
[ecx + esi*4]          ; base + index * scale
[ebp + edx]            ; scale is 1
[ebx - 8]              ; displacement is -8
[eax + edi*8 + 500]    ; all four components
[ebx + counter]        ; uses the address of the variable 'counter' as the displacement
```

The Three Kinds of Operands (3)

- Immediate Operands
  - These can be written in many ways. Here are some examples from the official docs.

```
200           ; decimal
0200          ; still decimal - the leading 0 does not make it octal
0200d         ; explicitly decimal - d suffix
0d200         ; also decimal - 0d prefex
0c8h          ; hex - h suffix, but leading 0 is required because c8h looks like a var
0xc8          ; hex - the classic 0x prefix
0hc8          ; hex - for some reason NASM likes 0h
310q          ; octal - q suffix
0q310         ; octal - 0q prefix
11001000b     ; binary - b suffix
0b1100_1000   ; binary - 0b prefix, and by the way, underscores are allowed
```

Instructions formats

- Most of the basic instructions have only the following forms:

add *reg*, *reg*
add *reg*, *mem*
add *reg*, *imm*
add *mem*, *reg*
add *mem*, *imm*

Defining Data and Reserving Space

- To reserve space in memory, you can use the following pseudo instructions

```
db      0x55                ; just the byte 0x55
db      0x55,0x56,0x57      ; three bytes in succession
db      'a',0x55            ; character constants are OK
db      'hello',13,10,'$'   ; so are string constants
dw      0x1234              ; 0x34 0x12
dw      'a'                 ; 0x61 0x00 (it's just a number)
dw      'ab'                ; 0x61 0x62 (character constant)
dw      'abc'               ; 0x61 0x62 0x63 0x00 (string)
dd      0x12345678          ; 0x78 0x56 0x34 0x12
dd      1.234567e20         ; floating-point constant
dq      0x123456789abcdef0  ; eight byte constant
dq      1.234567e20         ; double-precision float
dt      1.234567e20         ; extended-precision float
```

Conditional Instructions (1)

- After an arithmetic or logic instruction, or the compare instruction, ***cmp***, the processor sets or clears bits in its ***eflags****. The most interesting flags are:
  - s (sign)
  - z (zero)
  - c (carry)
  - o (overflow)
- So after doing, say, an addition instruction, we can perform a jump, move, or set, based on the new flag settings. For example:

jz *label*	Jump to label L if the result of the operation was zero
cmovno *x*, *y*	*x* ← *y* if the last operation did *not* overflow
setc *x*	*x* ← *1* if the last operation had a carry, but *x* ← *0* otherwise (*x* must be a byte-size register or memory location)

*https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture#EFLAGS_Register

Conditional Instructions (2)

- The conditional instructions have three base forms:
  - *j* for conditional jump
  - *cmov* for conditional move
  - *set* for conditional set
- The suffix of the instruction can be one of many forms:

s	ns	z	nz	c	nc	o	no	p	np
pe	po	e	ne	l	nl	le	nle	g	ng
ge	nge	a	na	ae	nae	b	nb	be	nbe

Detailed example can be found here:

- https://en.wikibooks.org/wiki/X86_Assembly/Control_Flow

Understanding Function Calling

- Steps
  - Pass arguments through stack
  - Transfer control to the function
  - Perform function operations
  - Place result in register for caller
  - Return to place of call

Function Call Instructions

- Instructions:

push *x*	Decrement esp by the size of the operand, then store *x* in [esp]
pop *x*	Move [esp] into *x*, then increment esp by the size of the operand
call *label*	Push the address of the next instruction, then jump to the label
ret	Pop into the instruction pointer

  - Any register that your function needs to use should have it's current value put on the stack for safe keeping using the **PUSH** instruction.
  - Then after the function has finished it's logic, these registers can have their original values restored using the **POP** instruction.
  - When you **CALL** a subroutine, the address you called it from in your program is pushed onto the stack.
  - This address is then popped off the stack by **RET** and the program jumps back to that place in your code.

Calling Convention (1)

- The calling convention is a protocol about how to call and return from functions.
- Calling convention is based heavily on the use of the hardware-supported stack.
  - It is based on the `push`, `pop`, `call`, and `ret` instructions
- Arguments are passed to the functions by using the stack. Function return value is stored in the EAX register.
- Both function caller and callee need to follow the calling convention

Calling Convention (2)

- Caller rules:
  - To call a function, use the `call` instruction.
  - Before the function call,
    - Save EAX, ECX, EDX if the caller relies on the values in them.
    - Push function arguments onto the stack in inverted order
  - After the function call, the caller
    - Find the return value in the EAX register.
    - Remove the arguments from stack.
    - Restore the contents of caller-saved registers by popping them off of the stack.

Calling Convention (3)

- Caller rules (example)
  - To call a function: *`myfunc(arg1, arg2, arg3)`*

```
        section .text
_start:
;       ......
        push eax                ;save eax on stack
                                ;save ecx and edx on stack if needed
        push arg3               ;push last argument first
        push arg2               ;push the second argument
        push arg1               ;push first argument last
        call myfunc             ;call the function
        add  esp, 12            ;remove arguments
        mov [result], eax       ;the result of myfunc is now in eax
                                ;restore edx and ecx if previously saved
        pop eax                 ;restore eax
;       ......
        section .bss
result: resb 4                  ;reserve 4 bytes to store the result
```

Calling Convention (4)

- Callee rules
  - At the beginning of the function
    - Push the value of EBP onto the stack, and then copy the value of ESP into EBP.
    - Next, allocate local variables by making space on the stack.
    - Next, save the values of the callee-saved registers (EBX, EDI, and ESI) that will be used by the function.
  - At the end of the function
    - Leave the return value in EAX.
    - Restore the old values of any callee-saved registers that were modified.
    - Deallocate local variables.
    - Restore the caller's base pointer value by popping EBP off the stack.
    - Return to the caller by executing a RET instruction

Calling Convention (5)

- Callee rules (example)

```
;        ......
myfunc:
; Subroutine Prologue
        push ebp        ; Save the old base pointer value.
        mov  ebp, esp   ; Set the new base pointer value.
        sub  esp, 4     ; Make room for one 4-byte local variable.
        push edi        ; Save the values of registers that the function
        push esi        ; will modify. This function uses EDI and ESI.
                        ; (no need to save EBX, EBP, or ESP)

; Subroutine Body
        mov eax, [ebp+8]  ; Move value of parameter 1 into EAX
        mov esi, [ebp+12] ; Move value of parameter 2 into ESI
        mov edi, [ebp+16] ; Move value of parameter 3 into EDI

        mov [ebp-4], edi ; Move EDI into the local variable
        add [ebp-4], esi ; Add ESI into the local variable
        add eax, [ebp-4] ; Add the contents of the local variable
                         ; into EAX (final result)
```

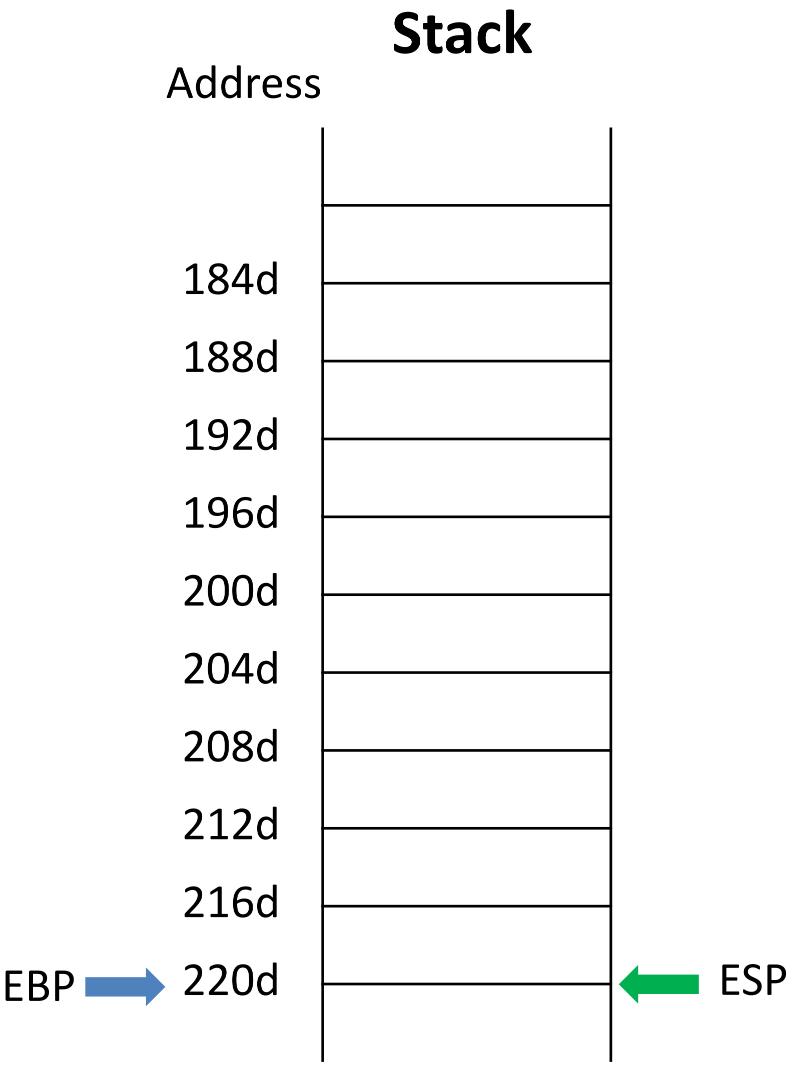
To be continued ...

Calling Convention (5)

- Callee rules (example)

```
; Subroutine Epilogue
        pop esi             ; Recover register values
        pop edi
        mov esp, ebp        ; Deallocate local variables
        pop ebp             ; Restore the caller's base pointer value
        ret
```

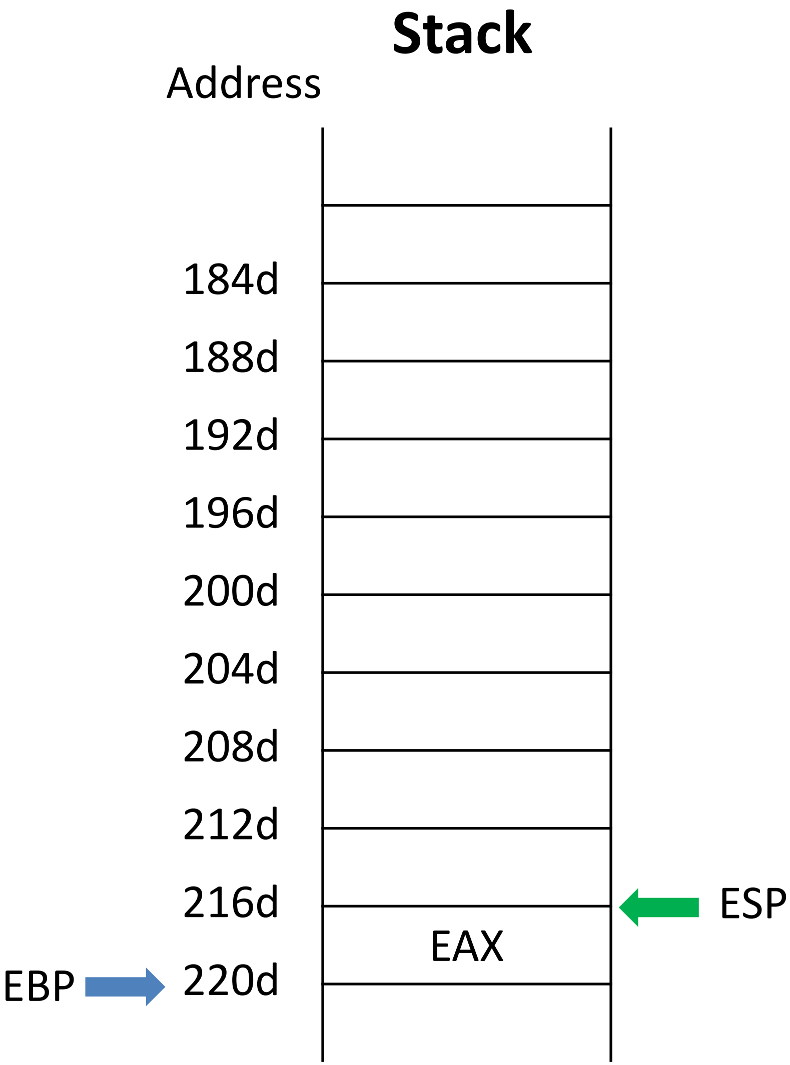
Function Call Example




```
    section .text
_start:
    push eax
    push arg3
    push arg2
    push arg1
    call myfunc
    add  esp, 12
    mov [result], eax
    pop eax
section .bss
result: resb 4
```

EAX
12345678h
ESI
13571357h
EDI
24682468h

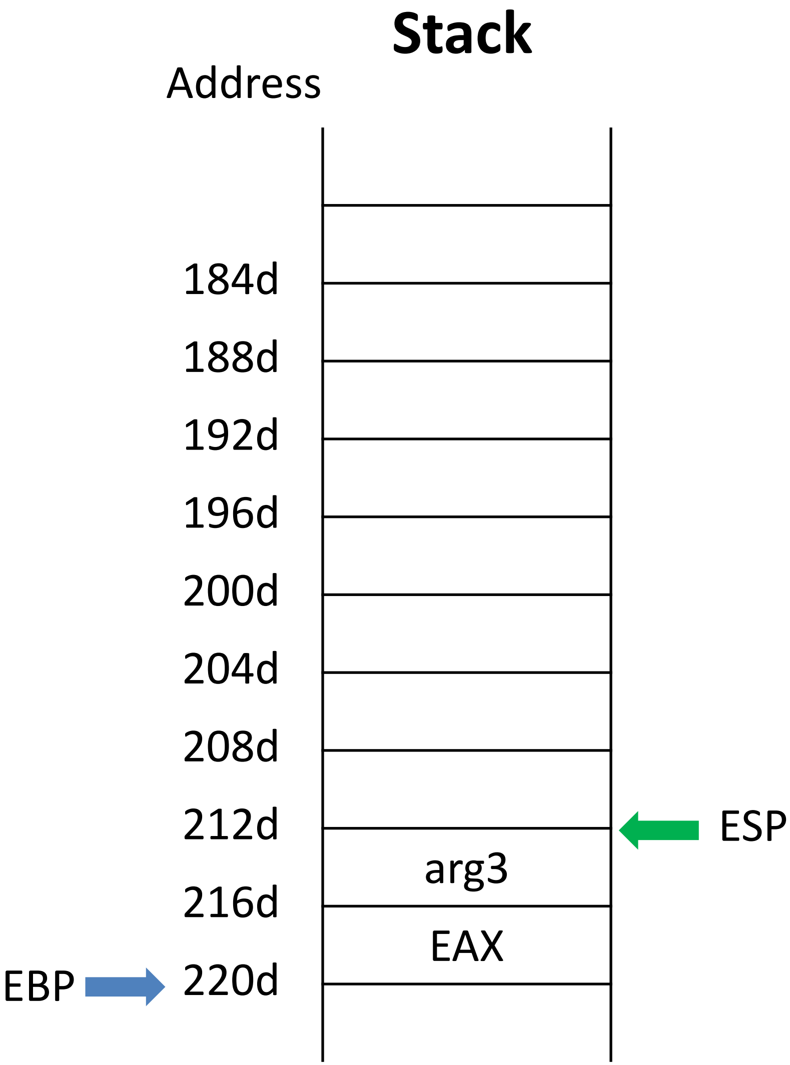
Function Call Example




```
    section .text
_start:
    push eax
    push arg3
    push arg2
    push arg1
    call myfunc
    add  esp, 12
    mov [result], eax
    pop eax
section .bss
result: resb 4
```

EAX
12345678h
ESI
13571357h
EDI
24682468h

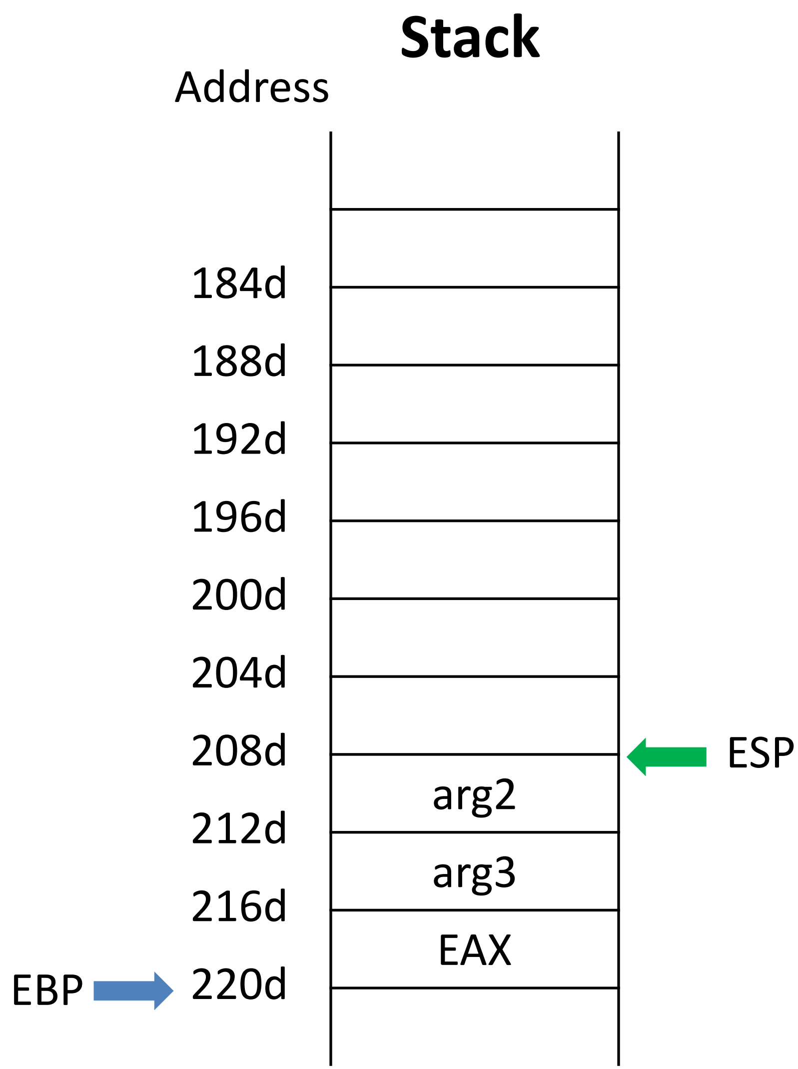
Function Call Example




```
    section .text
_start:
    push eax
    push arg3
    push arg2
    push arg1
    call myfunc
    add  esp, 12
    mov [result], eax
    pop eax
section .bss
result: resb 4
```

EAX
12345678h
ESI
13571357h
EDI
24682468h

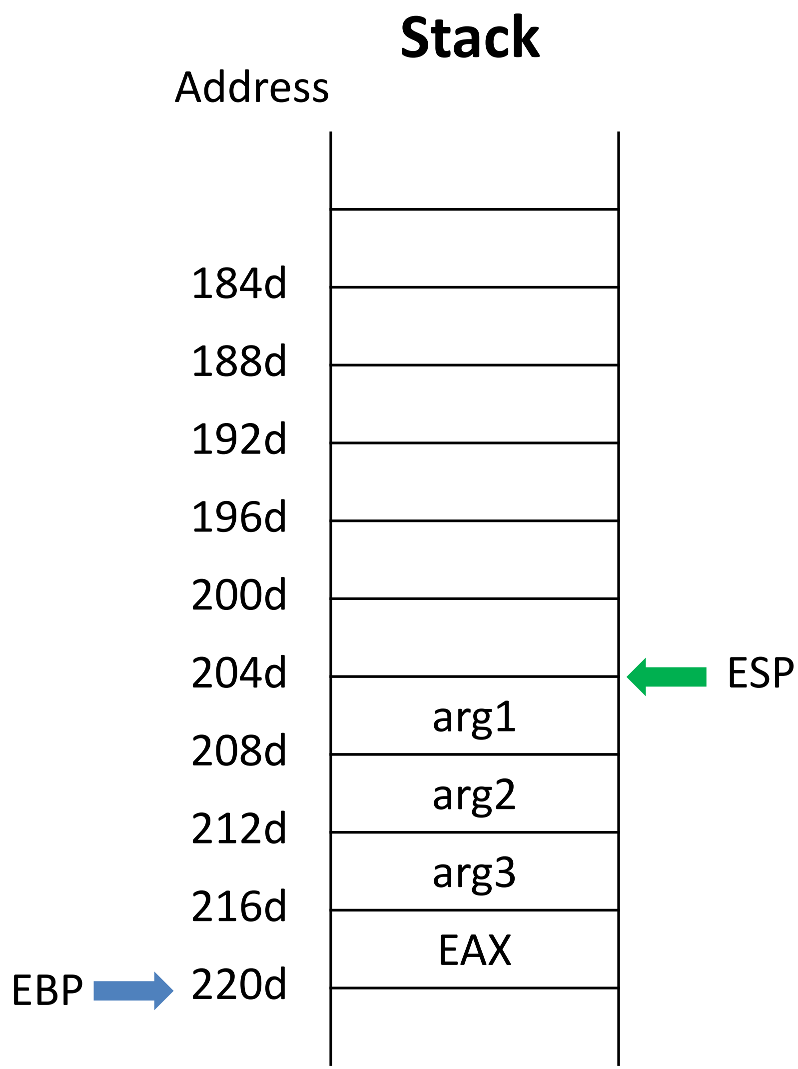
Function Call Example




```
    section .text
_start:
    push eax
    push arg3
    push arg2
    push arg1
    call myfunc
    add  esp, 12
    mov [result], eax
    pop eax
section .bss
result: resb 4
```

EAX	12345678h
ESI	13571357h
EDI	24682468h

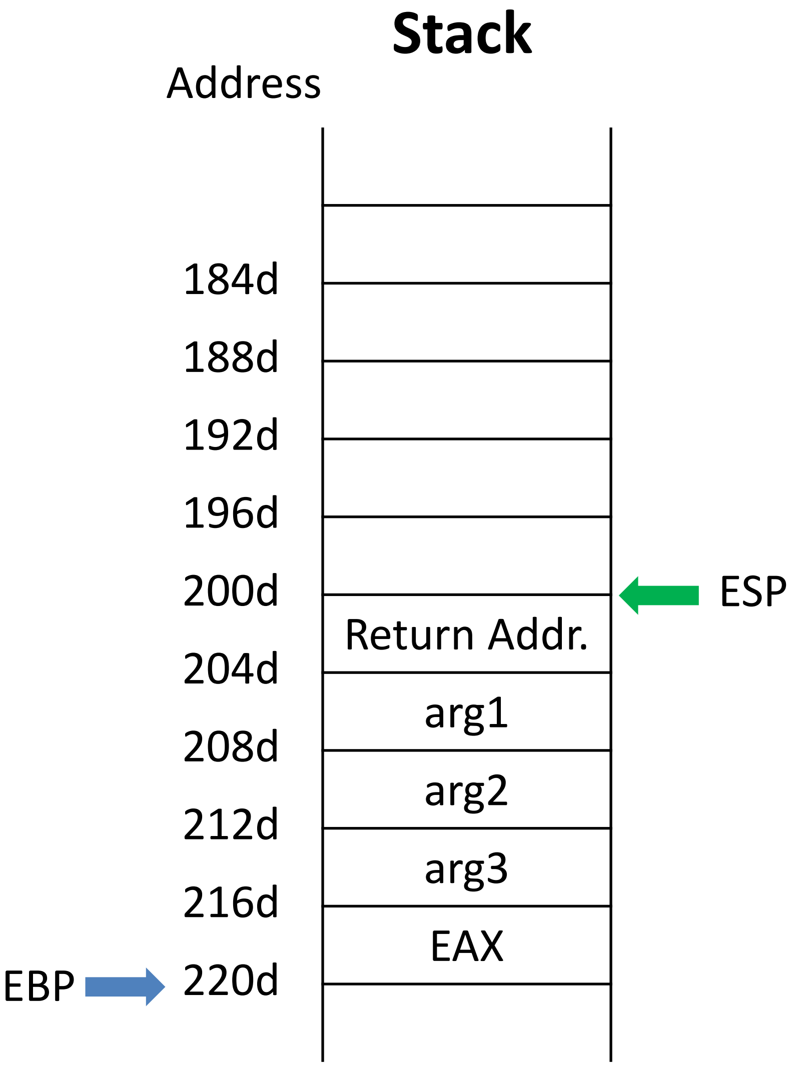
Function Call Example




```
    section .text
_start:
    push eax
    push arg3
    push arg2
    push arg1
    call myfunc
    add  esp, 12
    mov [result], eax
    pop eax
section .bss
result: resb 4
```

EAX
12345678h
ESI
13571357h
EDI
24682468h

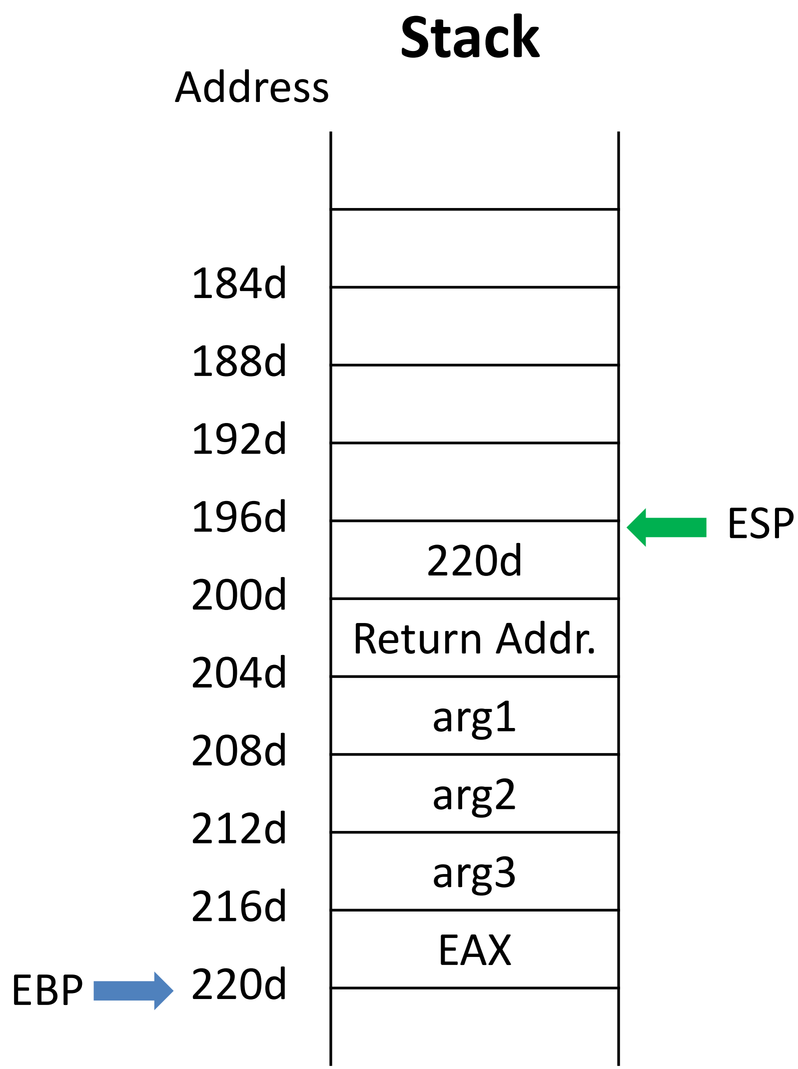
Function Call Example




```
    section .text
_start:
    push eax
    push arg3
    push arg2
    push arg1
    call myfunc
    add  esp, 12
    mov  [result], eax
    pop  eax
section .bss
result: resb 4
```

EAX
12345678h
ESI
13571357h
EDI
24682468h

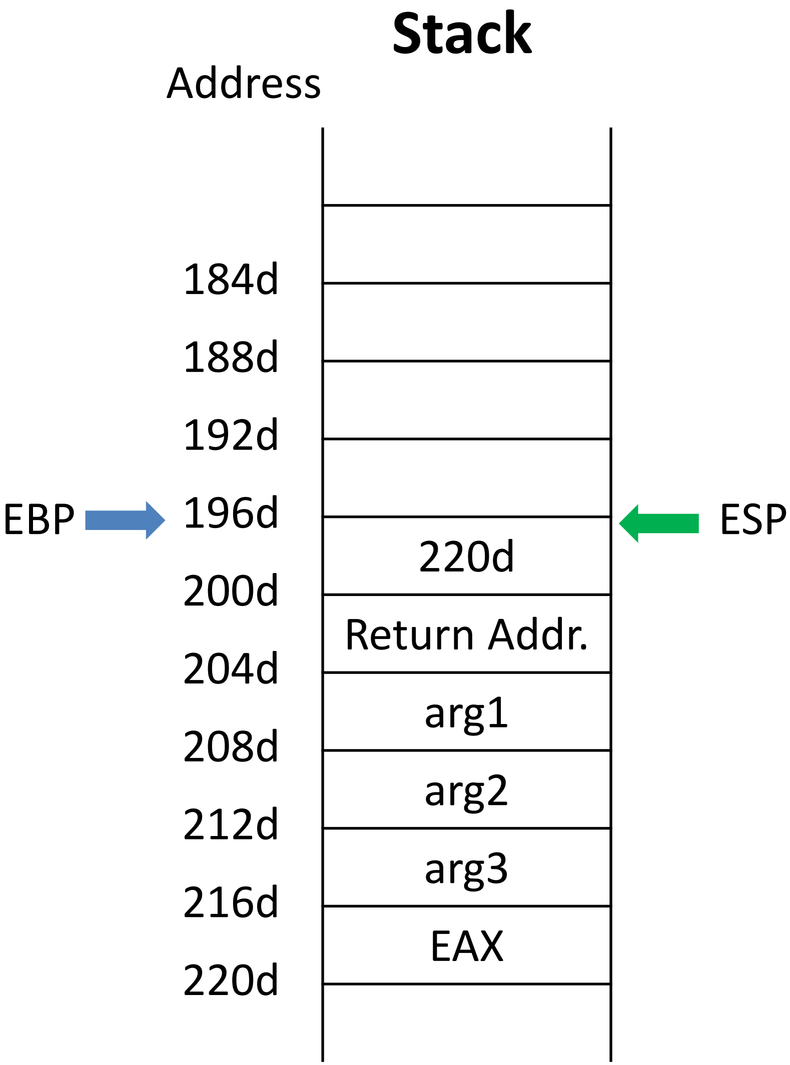
Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

EAX
12345678h
ESI
13571357h
EDI
24682468h

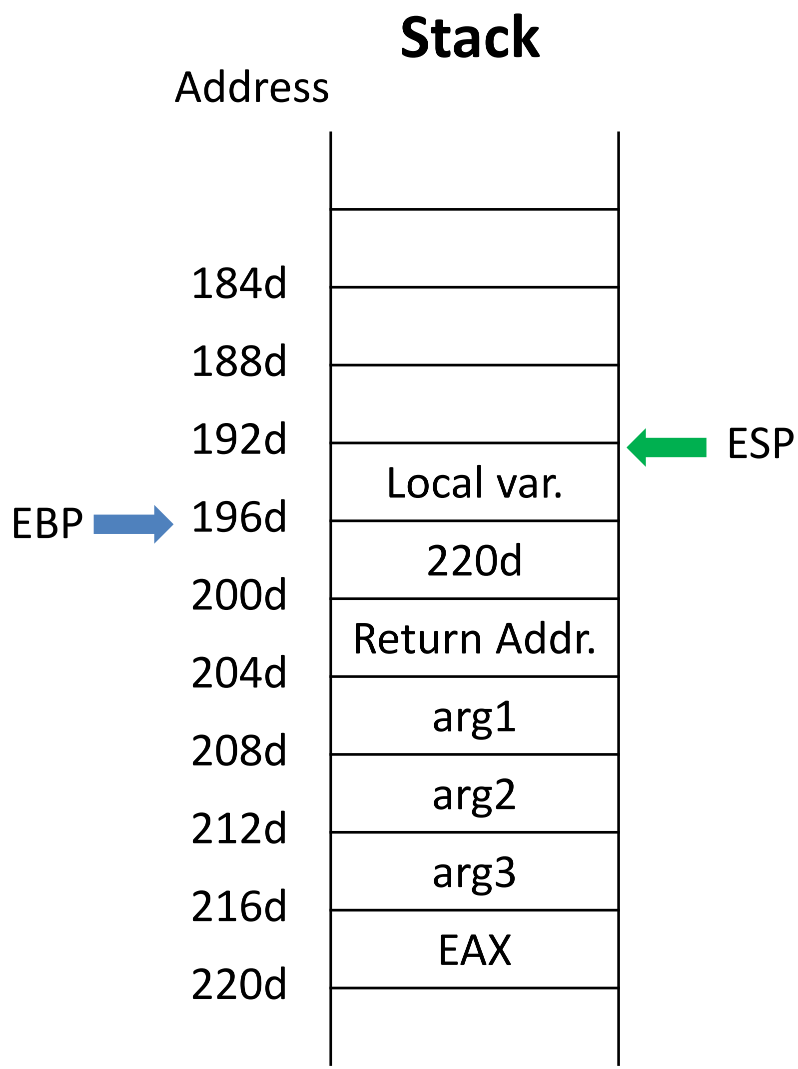
Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

EAX
12345678h
ESI
13571357h
EDI
24682468h

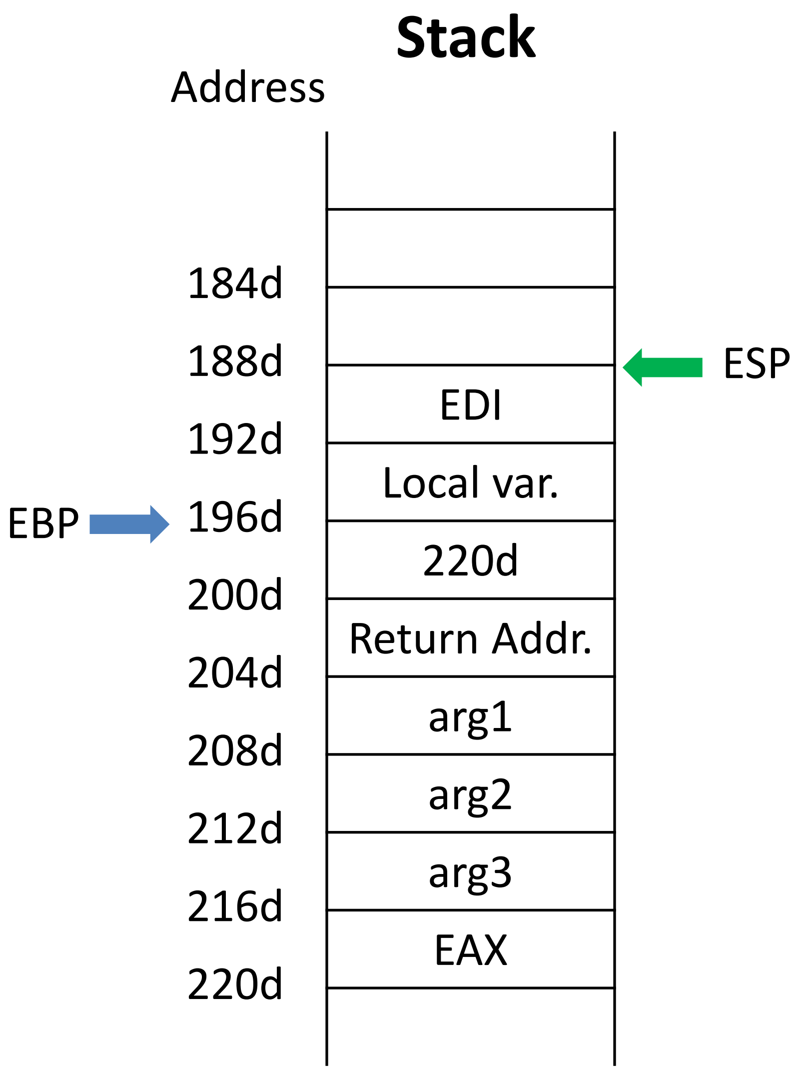
Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

EAX
12345678h
ESI
13571357h
EDI
24682468h

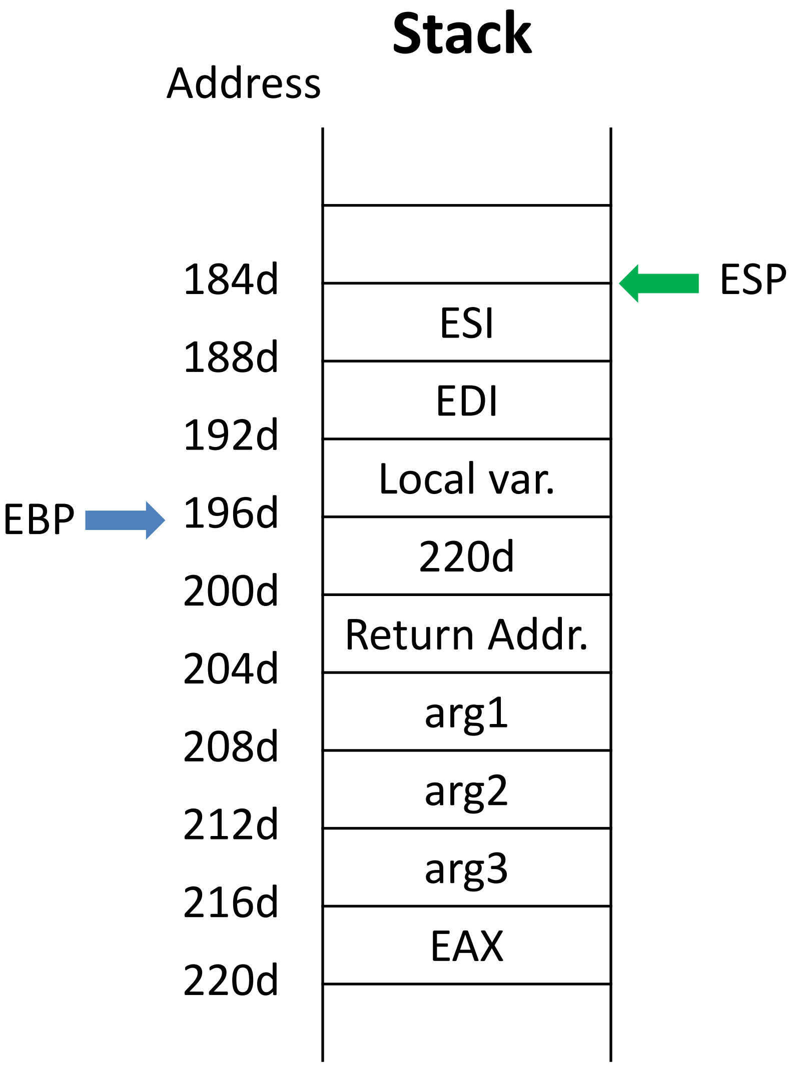
Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

EAX
12345678h
ESI
13571357h
EDI
24682468h

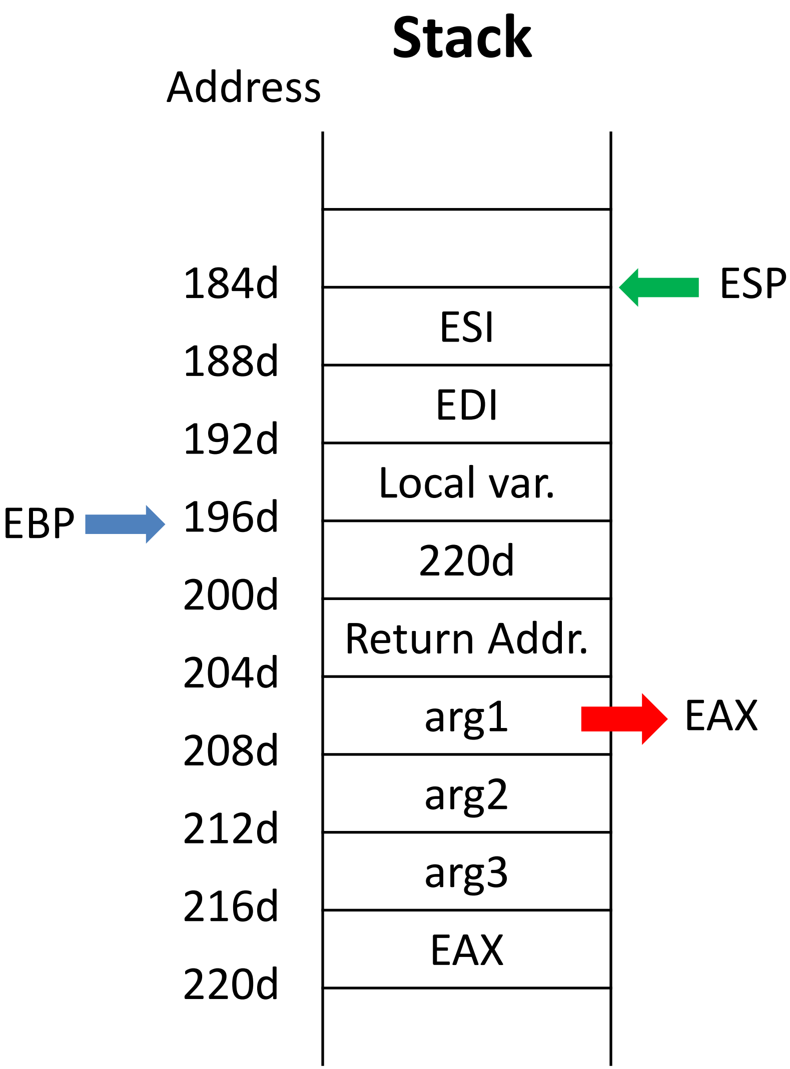
Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

EAX
12345678h
ESI
13571357h
EDI
24682468h

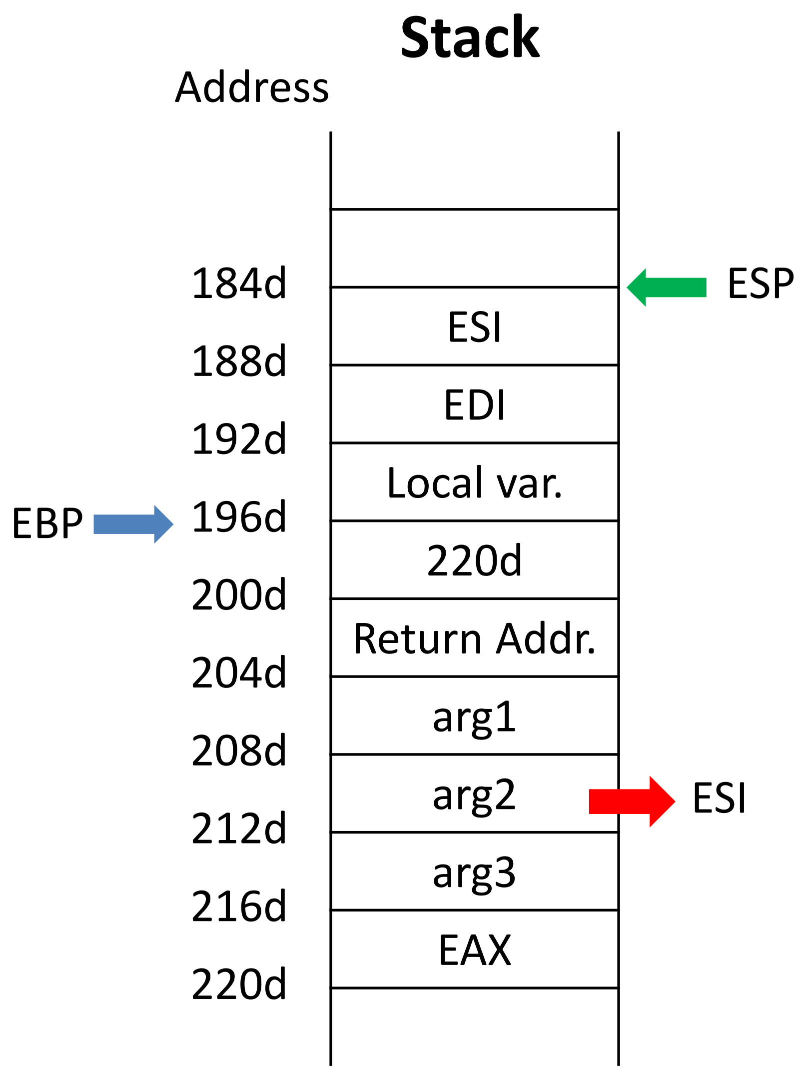
Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

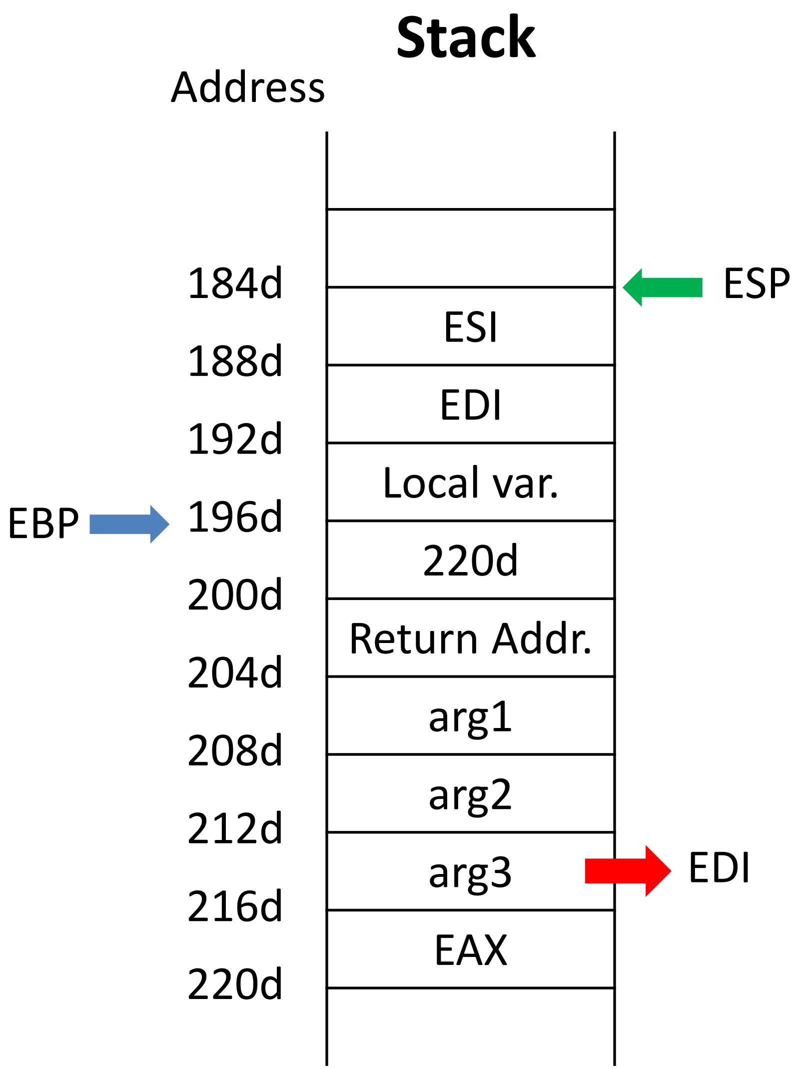
EAX
arg1
ESI
13571357h
EDI
24682468h

Function Call Example



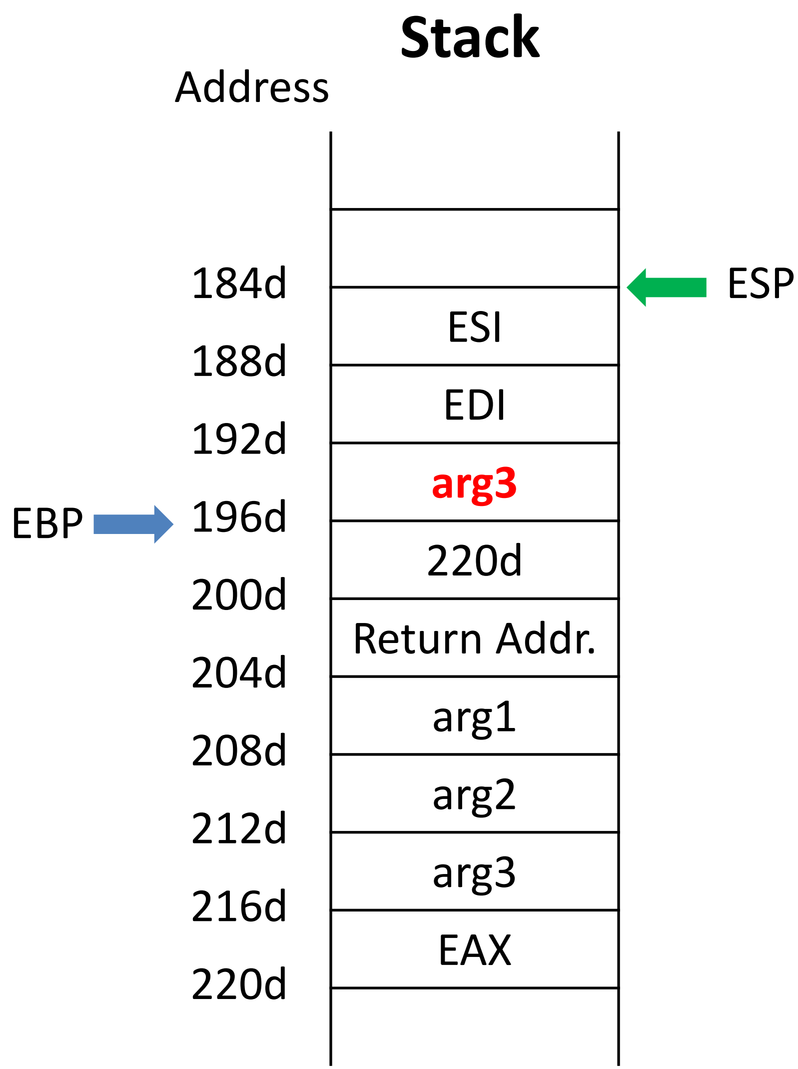

```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

Function Call Example



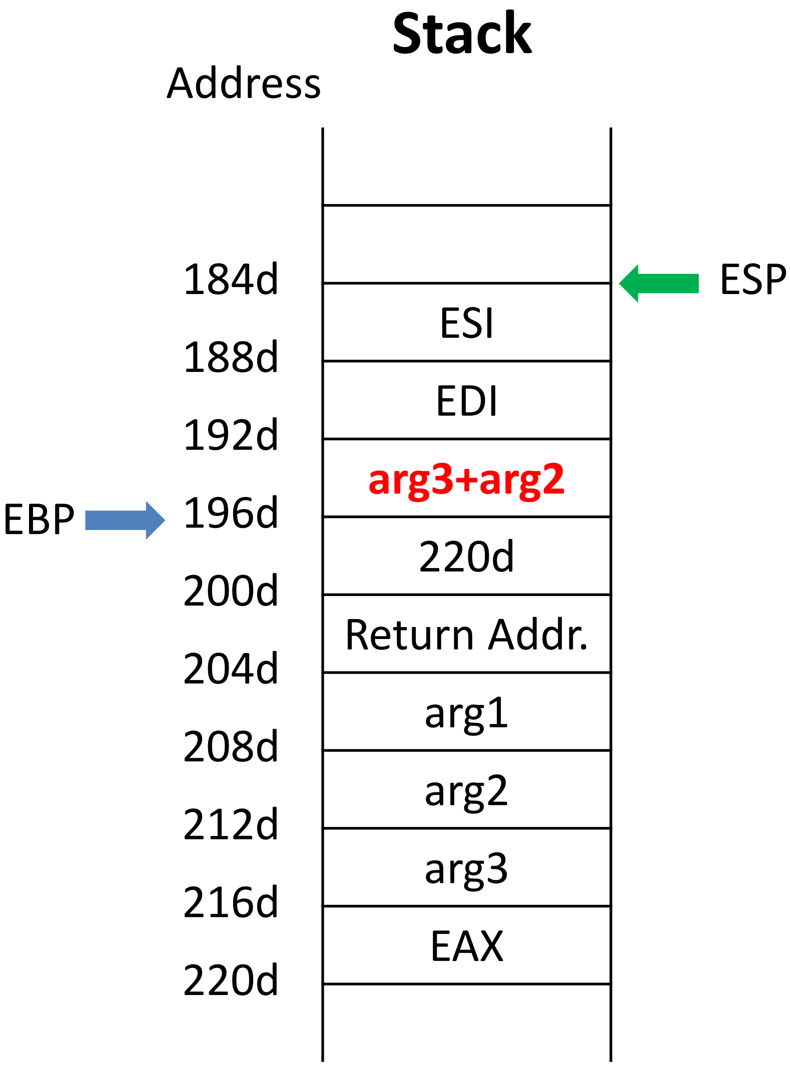

```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov  eax, [ebp+8]
    mov  esi, [ebp+12]
    mov  edi, [ebp+16]
    mov  [ebp-4], edi
    add  [ebp-4], esi
    add  eax, [ebp-4]
; Subroutine Epilogue
    pop  esi
    pop  edi
    mov  esp, ebp
    pop  ebp
    ret
```

Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

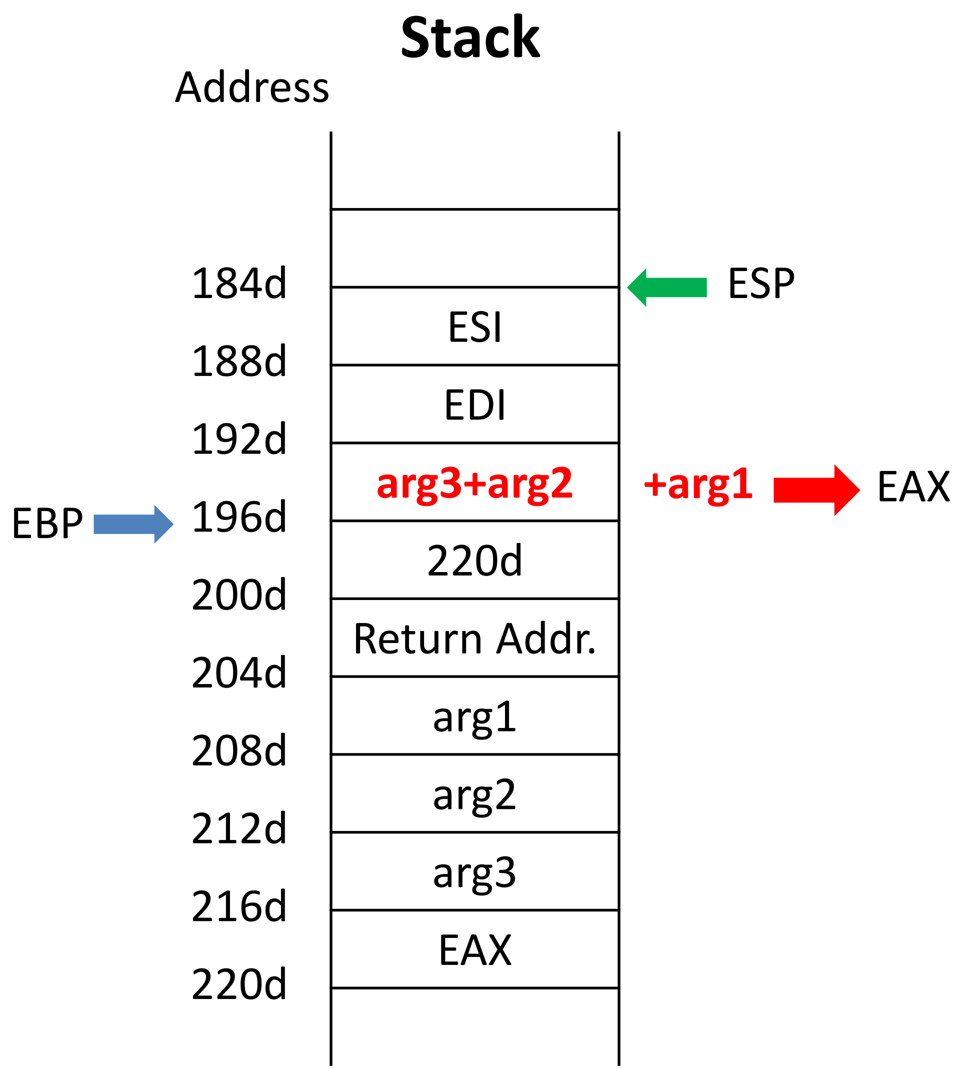
Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

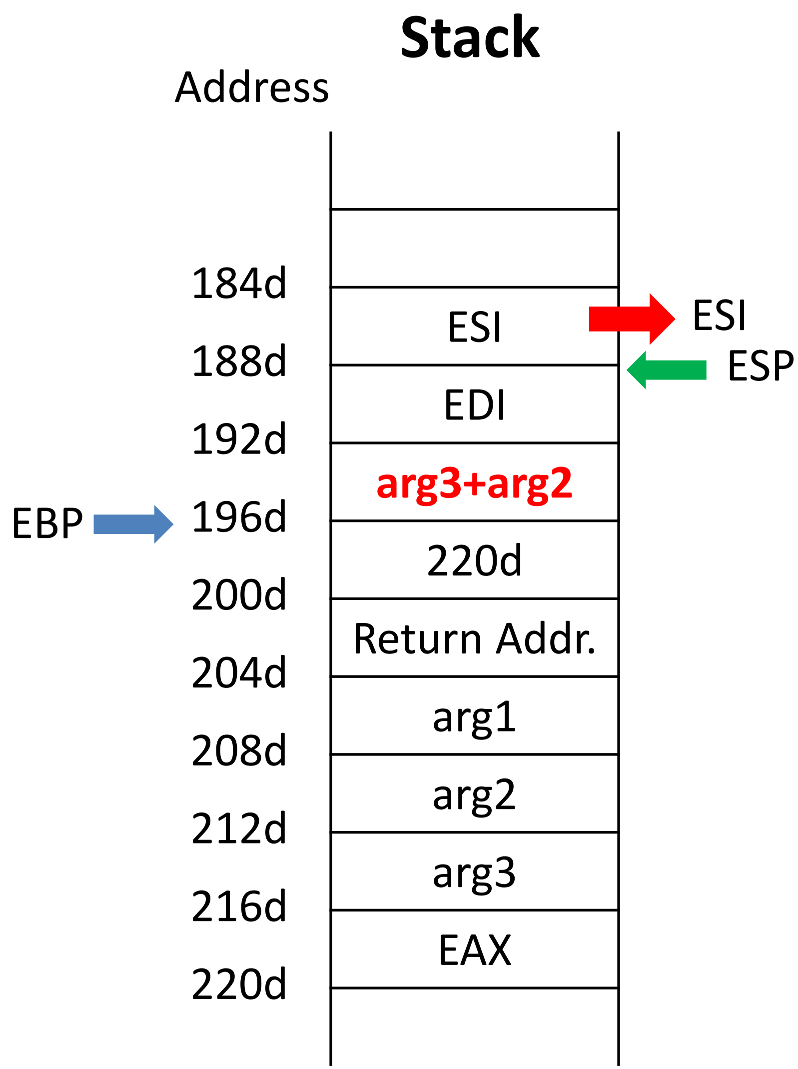
EAX
arg1
ESI
arg2
EDI
arg3

Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

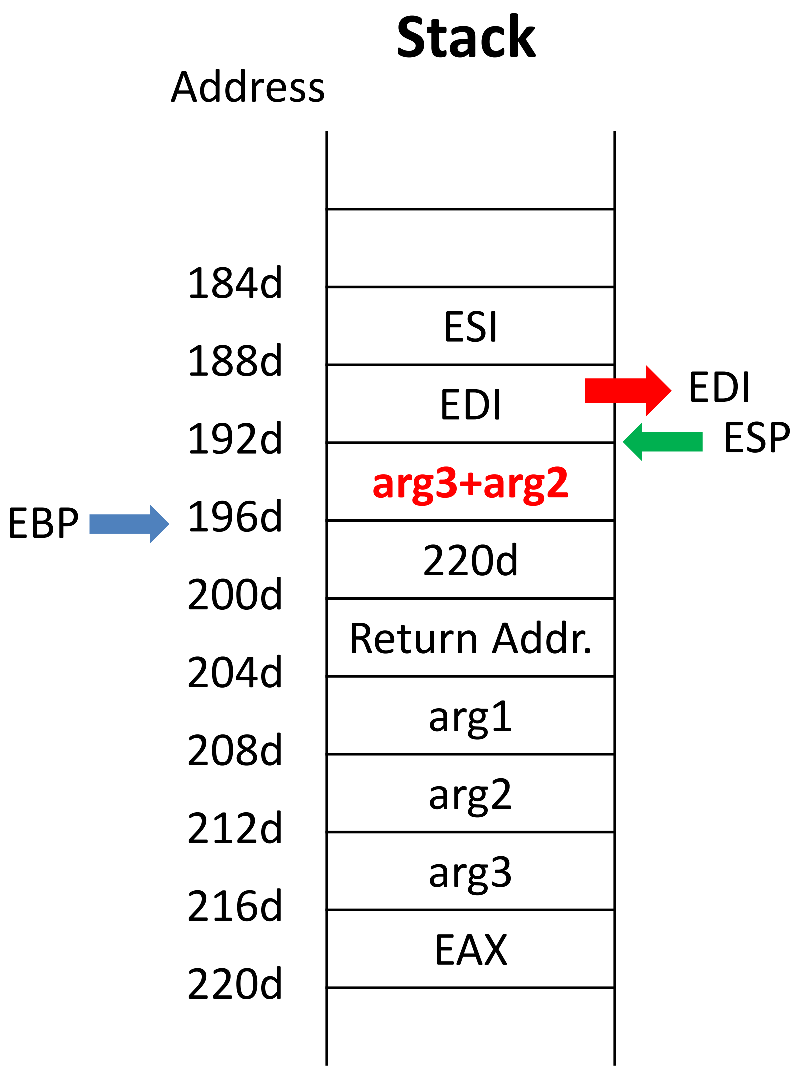
Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

EAX
arg1+arg2 +arg3
ESI
13571357h
EDI
arg3

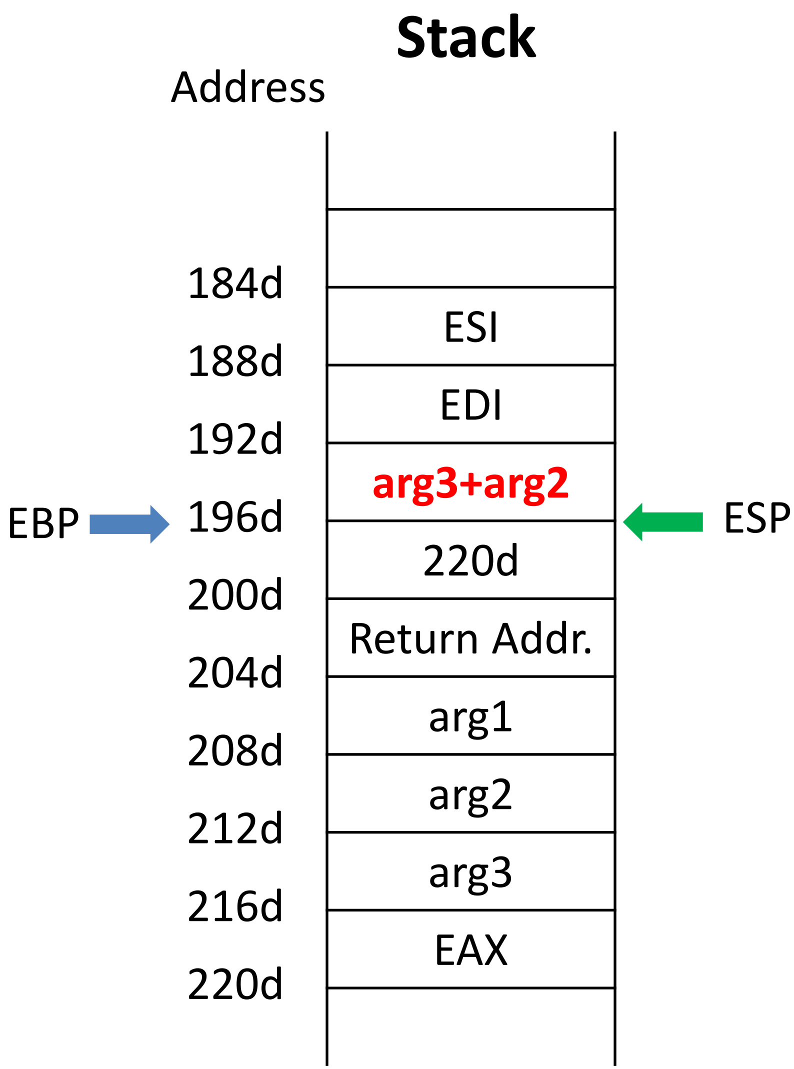
Function Call Example



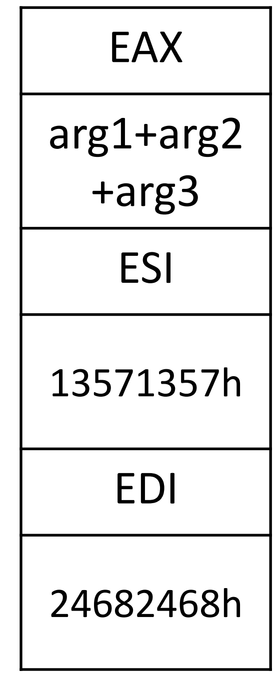

```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

EAX
arg1+arg2 +arg3
ESI
13571357h
EDI
24682468h

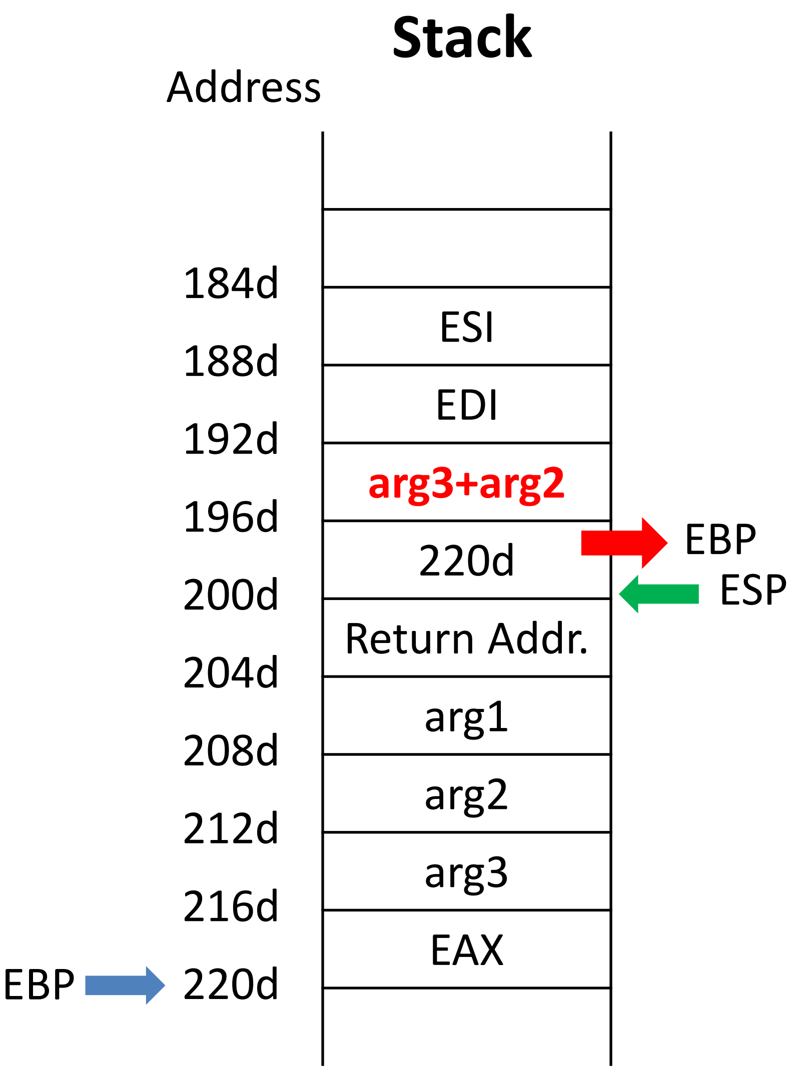
Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```



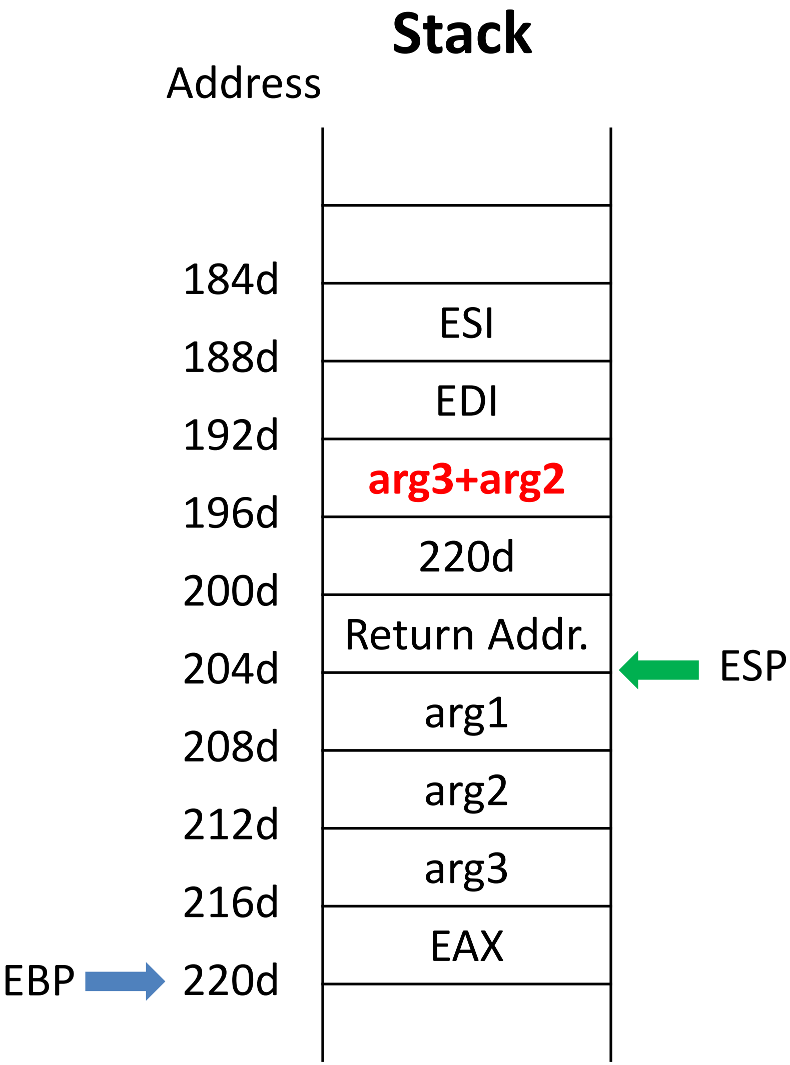
Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

EAX
arg1+arg2 +arg3
ESI
13571357h
EDI
24682468h

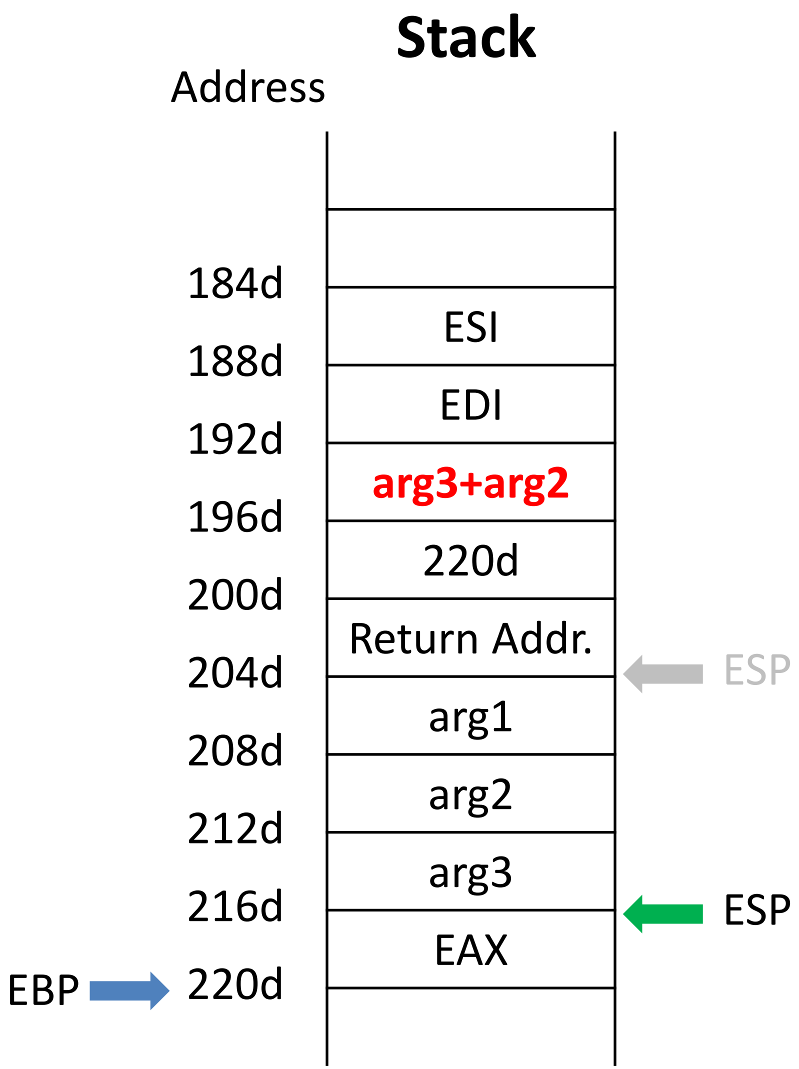
Function Call Example




```
myfunc:
; Subroutine Prologue
    push ebp
    mov  ebp, esp
    sub  esp, 4
    push edi
    push esi
; Subroutine Body
    mov eax, [ebp+8]
    mov esi, [ebp+12]
    mov edi, [ebp+16]
    mov [ebp-4], edi
    add [ebp-4], esi
    add eax, [ebp-4]
; Subroutine Epilogue
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

EAX	arg1+arg2 +arg3
ESI	13571357h
EDI	24682468h

Function Call Example

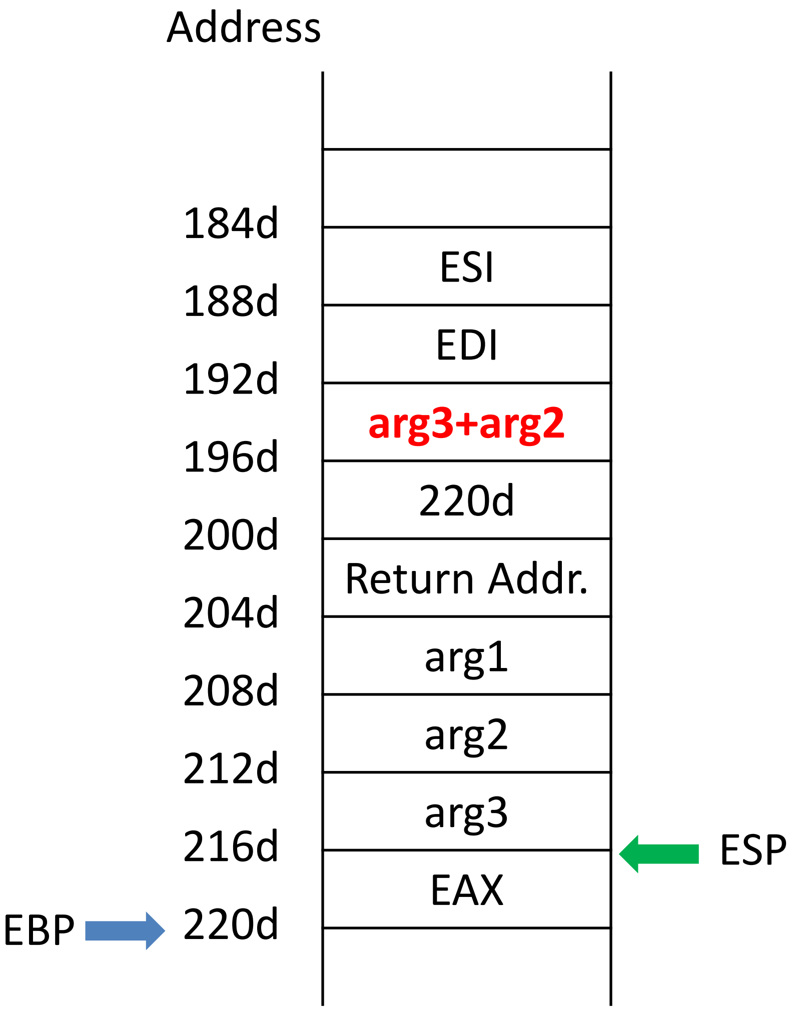



```
    section .text
_start:
    push eax
    push arg3
    push arg2
    push arg1
    call myfunc
    add  esp, 12
    mov [result], eax
    pop eax
section .bss
result: resb 4
```

EAX
arg1+arg2 +arg3
ESI
13571357h
EDI
24682468h

Function Call Example

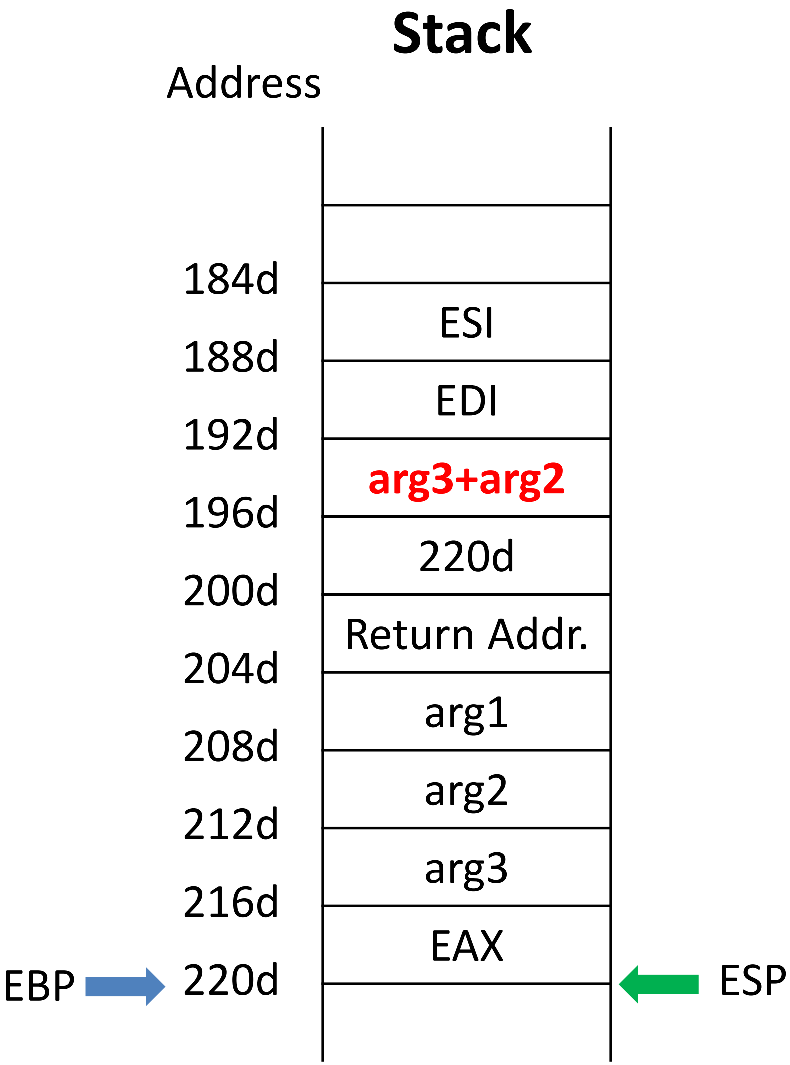
Stack




```
    section .text
_start:
    push eax
    push arg3
    push arg2
    push arg1
    call myfunc
    add  esp, 12
    mov [result], eax
    pop eax
section .bss
result: resb 4
```

EAX
arg1+arg2 +arg3
ESI
13571357h
EDI
24682468h

Function Call Example




```
    section .text
_start:
    push eax
    push arg3
    push arg2
    push arg1
    call myfunc
    add  esp, 12
    mov [result], eax
    pop eax
section .bss
result: resb 4
```

EAX
12345678h
ESI
13571357h
EDI
24682468h