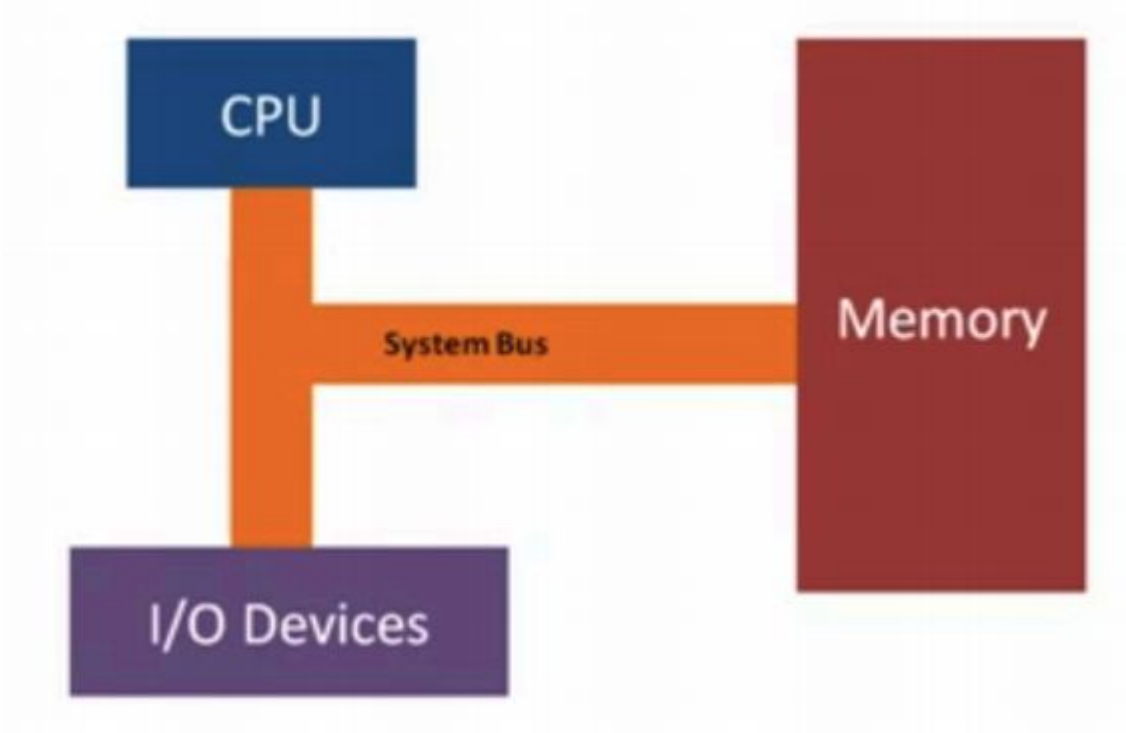
Assembly Primer

System Organization Basics

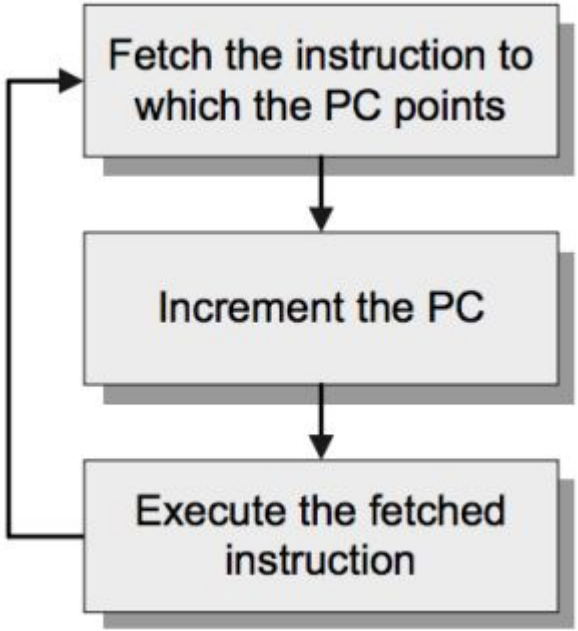


CPU

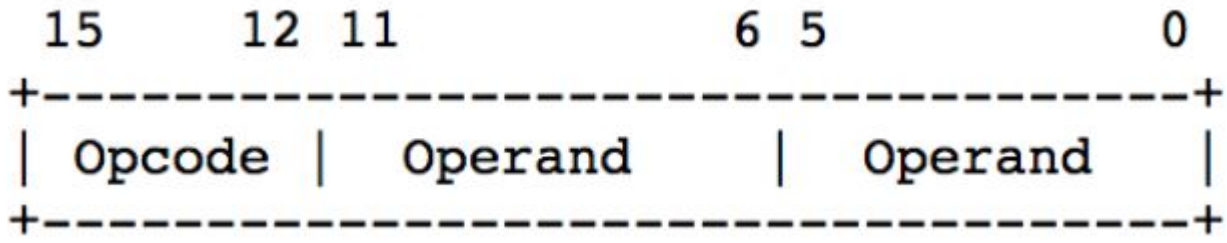
- Control Unit : Fetch & decode instructions , Fetch /store data in memory
- Execution Unit: Instruction Execution
- Registers: Internal memory (variables) of CPU
- Flags: Used to indicate specific events

CPU

- Fetch/execute cycle is the steps taken by CPU to execute an instruction
- Performing the action specified by an instruction is called executing an instruction
- Program Counter (PC) holds the memory address of next instruction



Instruction



- Instruction (aka binary instruction codes)
  - They are Macro operations consisting of several microoperations.
  - Microoperations fetch operands from memory or registers, perform arithmetic operations, stores results in registers
- Opcode
  - Designates the purpose of the instruction (add, sub, move etc).
  - No. of bits indicate no. of instructions supported by arch
- Operand
  - Registers or memory where data is located
  - No. of operands may vary: one operand for NOT and two for ADD

Instruction set, machine & assembly language

INSTRUCTION SET

- An instruction set is a list of commands ready to be executed directly by CPU.
- To instruct CPU with a set of instruction that can
  - tell the CPU where to find data
  - when to read the data
  - what to do with the data
- Types of instruction set.
  - Data transfer instruction
  - Arithmetic instruction
  - Logical instruction and bit manipulation
  - Program control instruction
  - Shift and rotate instruction

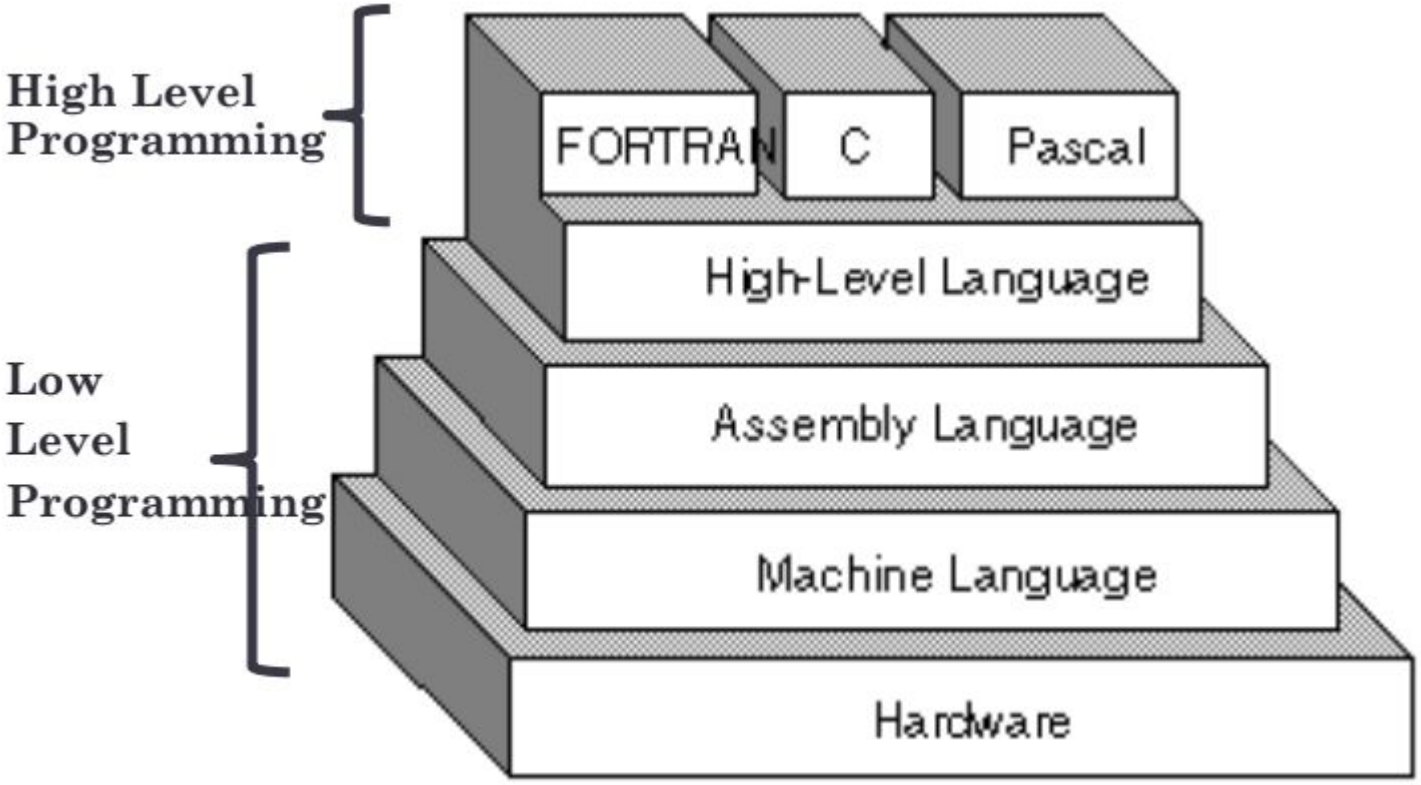
Assembly Language

- Low-level programming language for computers, microprocessors, microcontrollers and other programmable devices.
- Assembly language is just one level higher than machine language.
- Assembly language consists of simple codes.
- Each statement in an assembly language corresponds directly to a machine code understood by the microprocessor.
- The software used to convert an assembly program into machines codes is called an assembler.

What is the output of an assembler?

Machine Language

- A machine language is sometimes referred to as machine code or object code.
- Machine language is a collection of binary digits or bits that the computer reads and interprets.
- Machine language is the only language a computer is capable of understanding.
- Machine language consists of 0s and 1s.


High Level Programming
Low Level Programming
FORTRAN
C
Pascal
High-Level Language
Assembly Language
Machine Language
Hardware

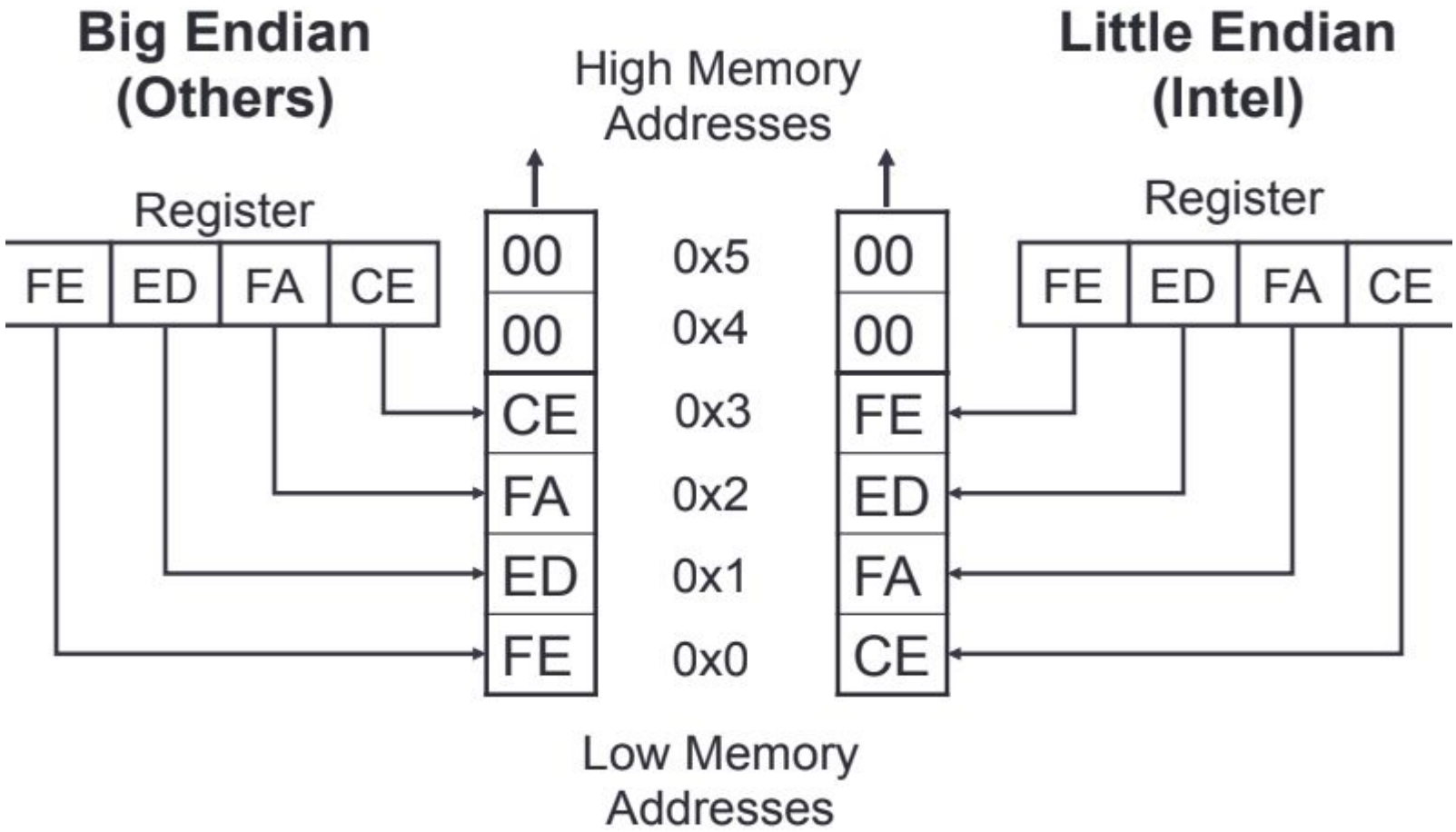
Architecture - CISC vs. RISC

- Intel is CISC - Complex Instruction Set Computer
  - Many very special purpose instructions that you will never see, and a given compiler may never use - just need to know how to use the manual
  - Variable-length instructions, between 1 and 16(?) bytes long.
  - 16 is max len in theory, I don't know if it can happen in practice
- Other major architectures are typically RISC -Reduced Instruction Set Computer
  - Typically more registers, less and fixed-size instructions
  - Examples: PowerPC, ARM, SPARC, MIPS

Architecture - Endian

- Endianness comes from Jonathan Swift's Gulliver's Travels.
- Little Endian - 0x12345678 stored in RAM “little end” first. The least significant byte of a word or larger is stored in the lowest address. E.g. 0x78563412
  - Intel is Little Endian
- Big Endian - 0x12345678 stored as is.
  - Network traffic is Big Endian
  - Most everyone else you've heard of (PowerPC, ARM, SPARC, MIPS) is either Big Endian by default or can be configured as either (Bi-Endian)

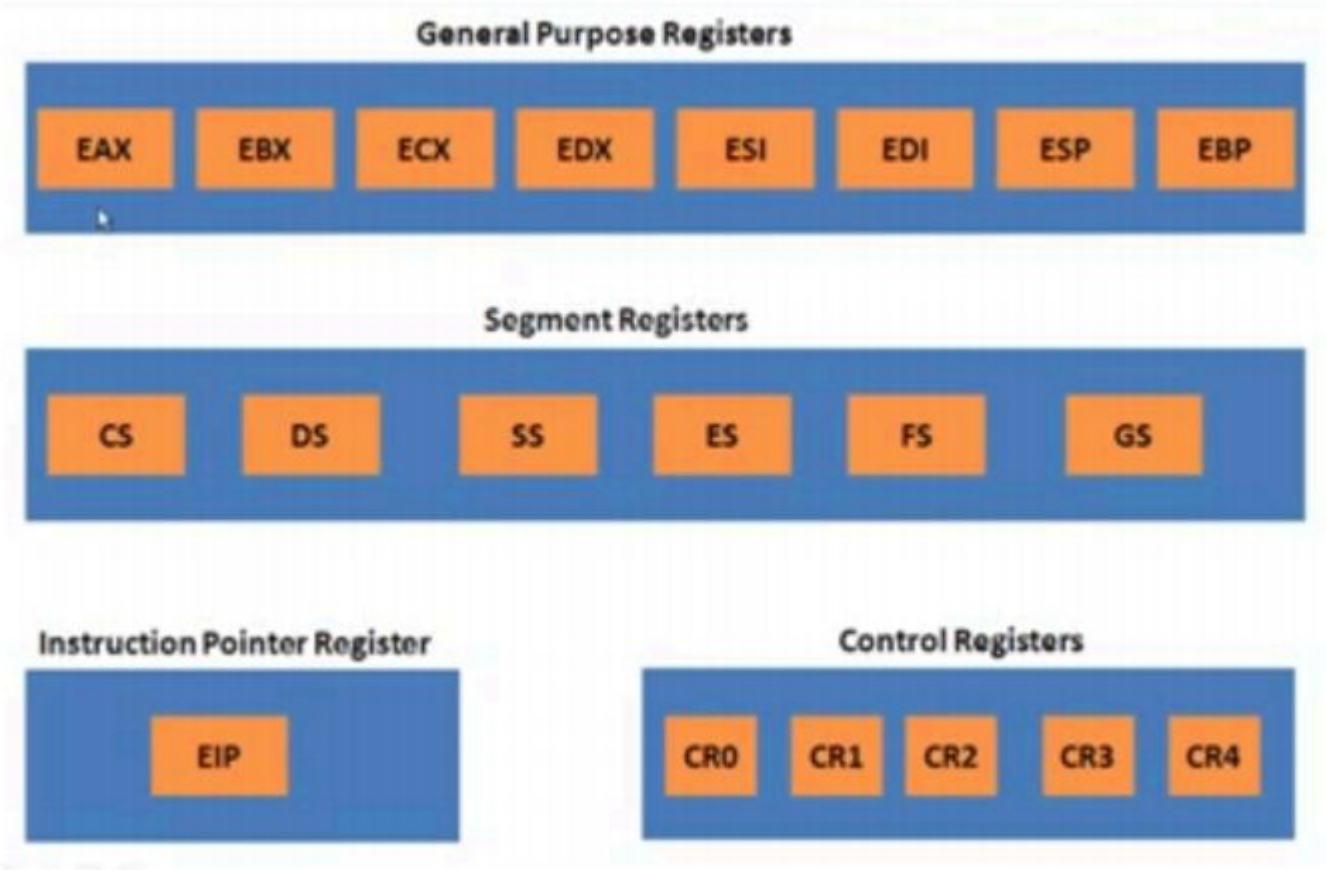
Endianess pictures



Architecture - Registers

- Registers are small memory storage areas built into the processor (still volatile memory)
- 8 “general purpose” registers + the instruction pointer which points at the next instruction to execute
  - But two of the 8 are not that general (ESP & EBP)
- On x86-32, registers are 32 bits long
- On x86-64, they're 64 bits

CPU Registers



Architecture - Register Conventions 1

- These are Intel's suggestions to compiler developers (and assembly handcoders). Registers don't have to be used these ways, but if you see them being used like this, you'll know why. I also color coded as **GREEN** for the ones which we will actually see in this class (as opposed to future ones), and **RED** for not.
- **EAX** - Stores function return values (used to be called accumulator; mainly for arithmetic operations)
- EBX - Base pointer to the data section
- **ECX** - Counter for string and loop operations (used to be called counter)

Architecture - Registers Conventions 2

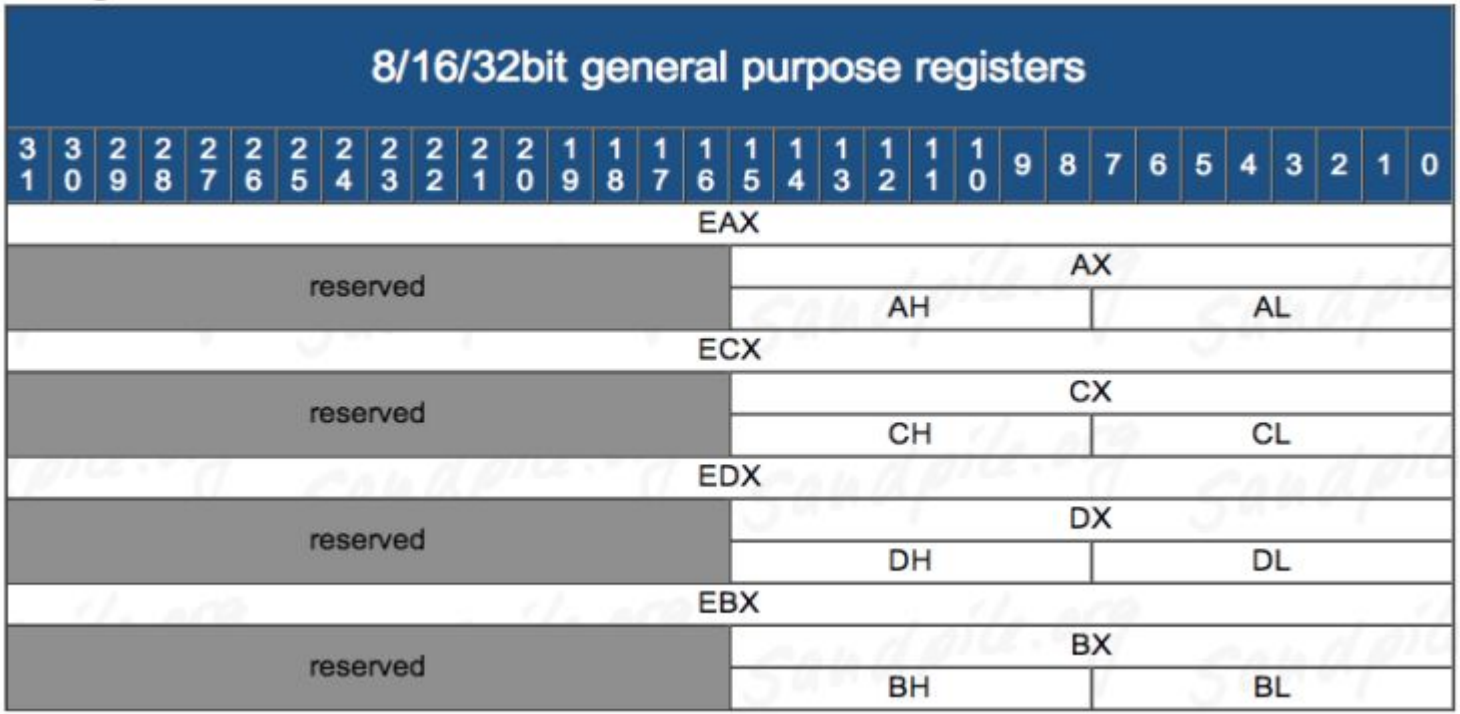
- **EDX** - I/O pointer
- **ESI** - Source pointer for string operations
- **EDI** - Destination pointer for string operations
- **ESP** - Stack pointer
- **EBP** - Stack frame base pointer
- **EIP** - Pointer to next instruction to execute (“instruction pointer”)

Architecture - Registers Conventions 3

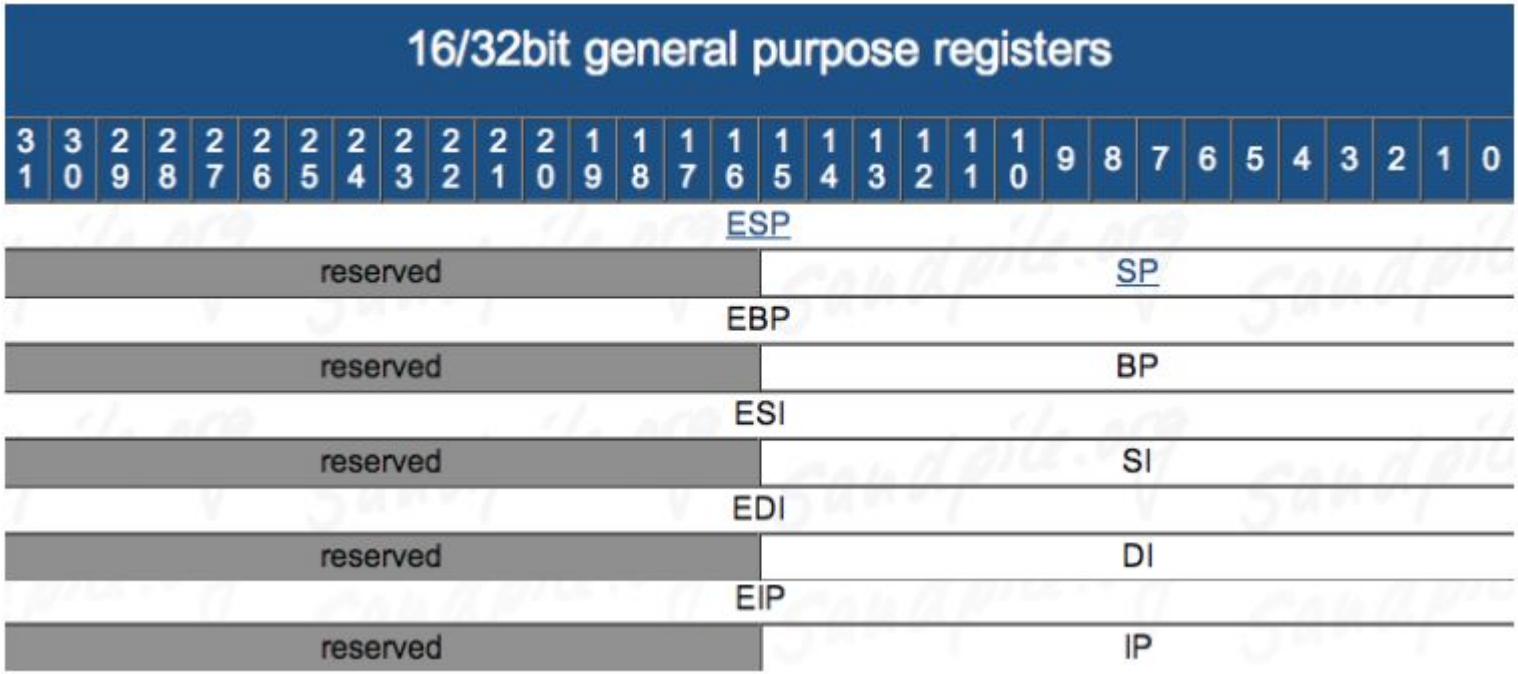
- Caller-save registers - eax, edx, ecx
  - If the caller has anything in the registers that it cares about, the caller is in charge of saving the value before a call to a subroutine, and restoring the value after the call returns
  - Put another way - the callee can (and is highly likely to) modify values in caller-save registers
- Callee-save registers - ebp, ebx, esi, edi
  - If the callee needs to use more registers than are saved by the caller, the callee is responsible for making sure the values are stored/restored
  - Put another way - the callee must be a good citizen and not modify registers which the caller didn't save, unless the callee itself saves and restores the existing values

Architecture - Registers - 8/16/32 bit addressing 1

EAX, ECX, EDX, EBX Allows selective access to lower order bits by using different name



Architecture - Registers - 8/16/32 bit addressing 2



Architecture - EFLAGS

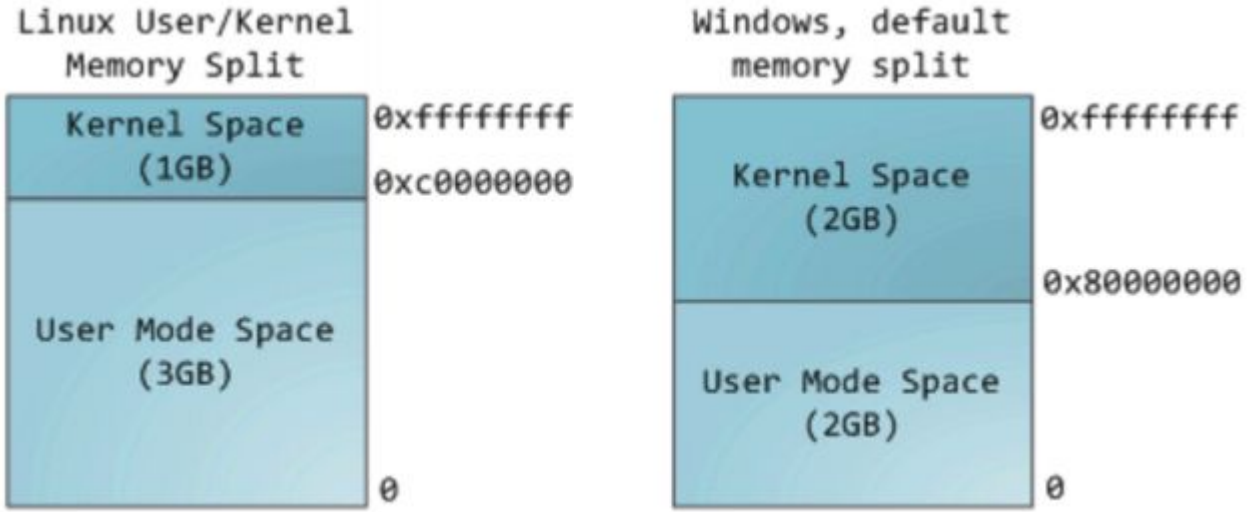
- EFLAGS register holds many single bit flags.
  - Zero Flag (ZF) - Set if the result of some instruction is zero; cleared otherwise.
  - Sign Flag (SF) - Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

Virtual Memory Model

- Each process in a multi-tasking OS runs in its own sandbox (what is a sandbox?)
- Sandbox is the **virtual address space** which in 32-bit mode is always a **4GB block of memory**.
- The virtual addresses are mapped to physical addresses using page tables that are maintained by the kernel
- Each process has its own set of page tables
- A portion of the VA space is allocated for the kernel

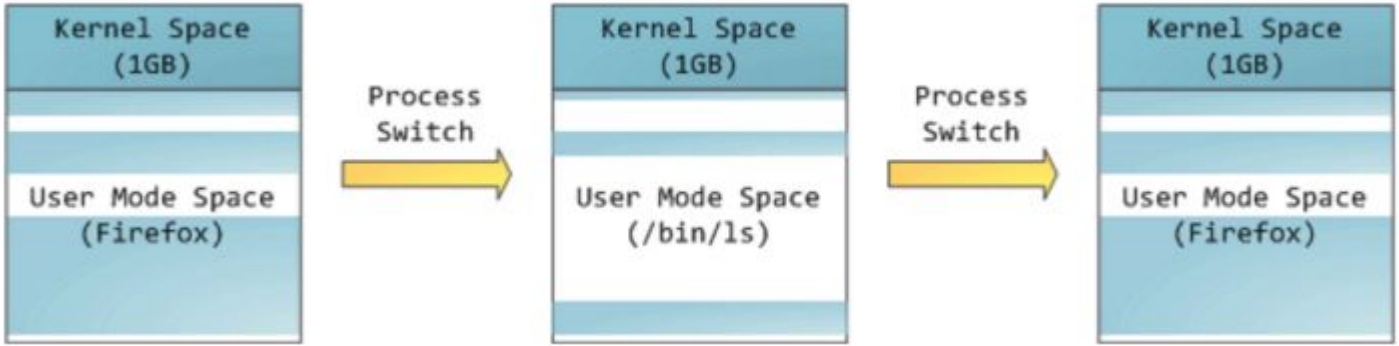
Virtual Memory Model

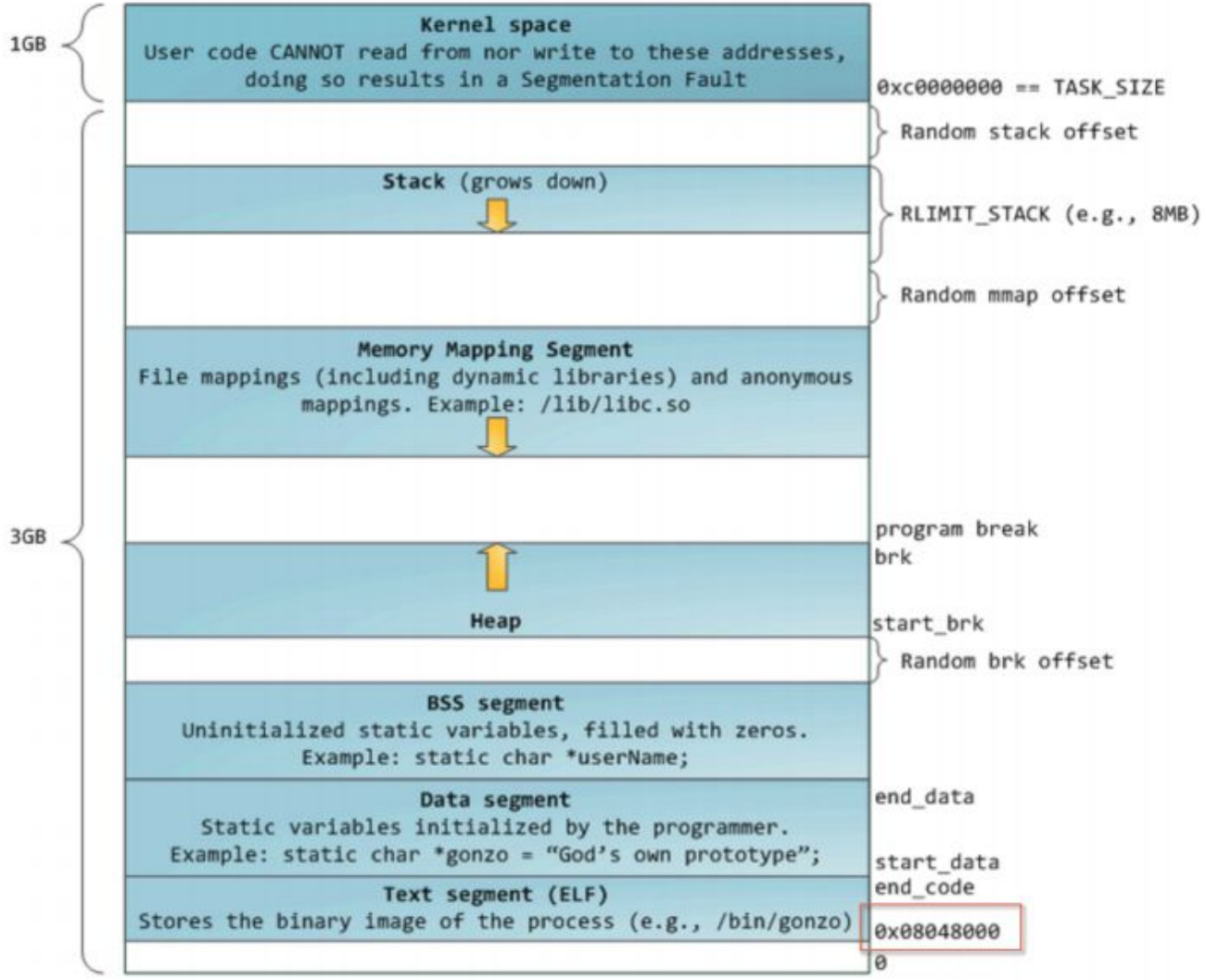
- Kernel space is flagged in the page tables as exclusive to privileged code (page fault when user-mode programs access it)
- In Linux, kernel space is always present and maps same physical memory in all processes



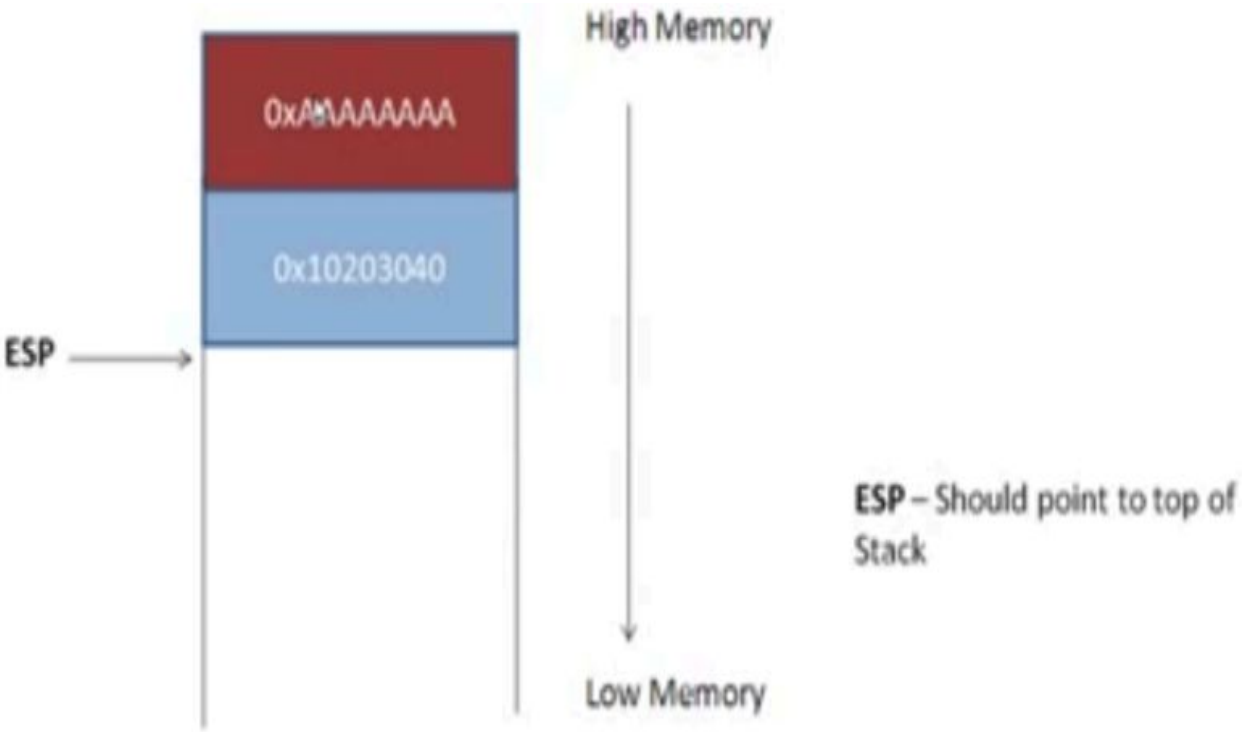
Virtual Memory Model

- Mapping for user-mode address space always changes whenever a process switch happens
- In Linux, kernel space is always present and maps same physical memory in all processes



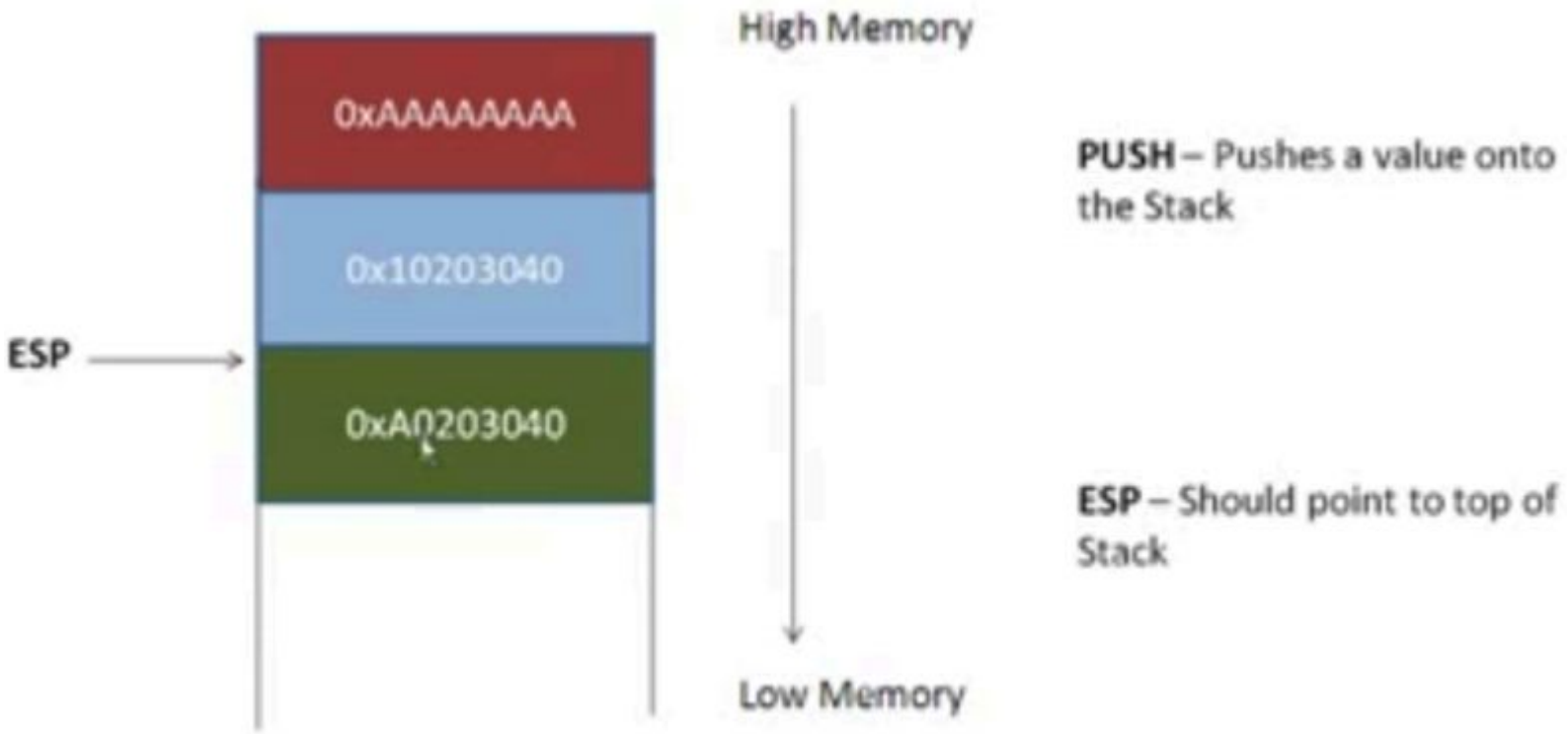

1GB
3GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
end_data
start_data
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
end_code
0x08048000
0

Stack is a LIFO

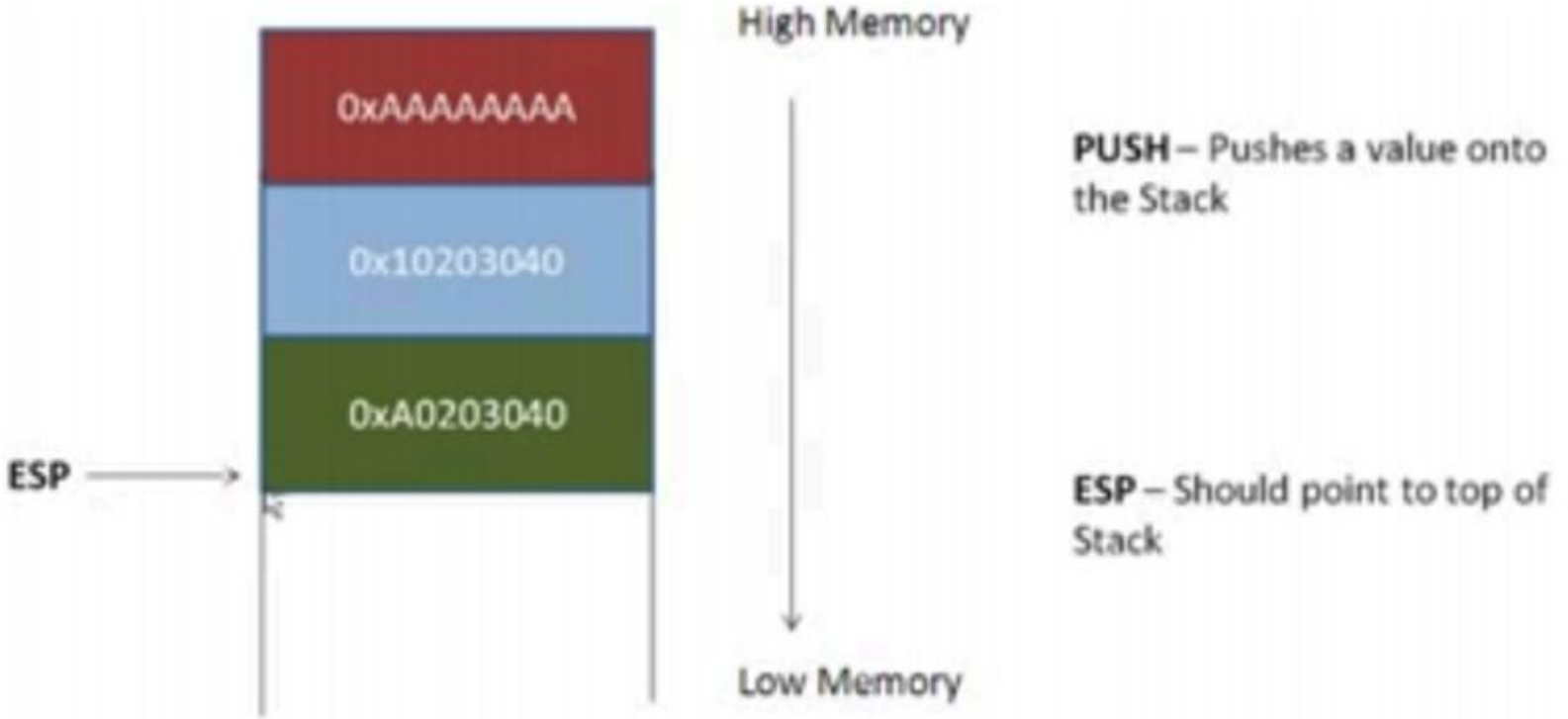


Two stack operations

Stack Push

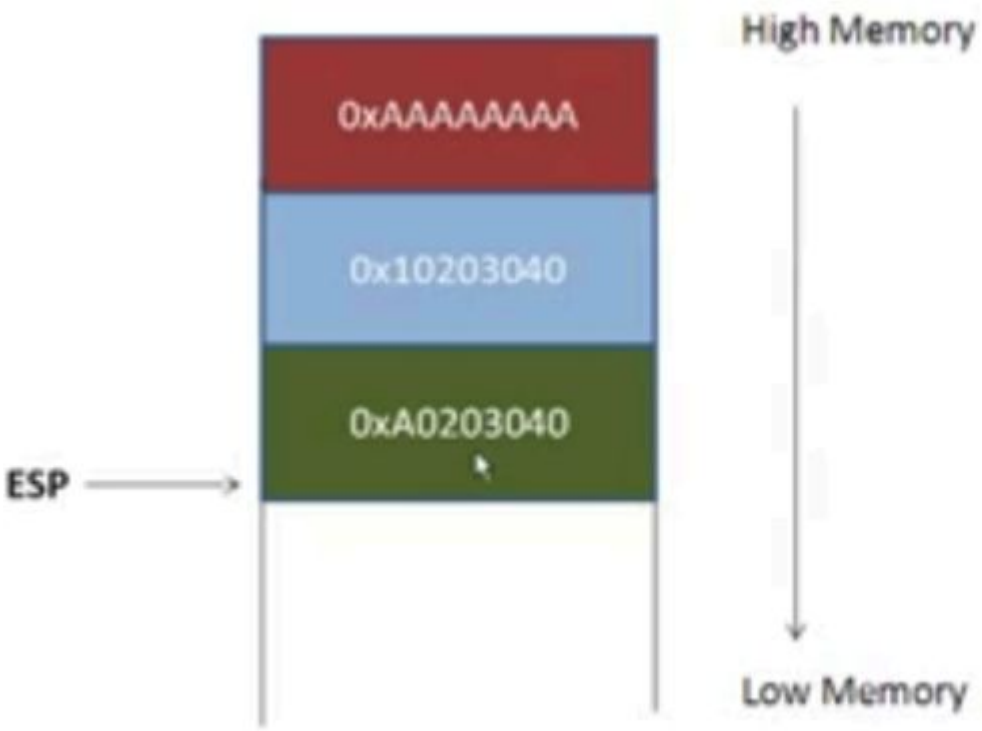

High Memory
0xAAAAAAAA
0x10203040
ESP
0xA0203040
Low Memory
PUSH – Pushes a value onto the Stack
ESP – Should point to top of Stack

Stack Push



ESP is updated

Stack Pop



PUSH – Pushes a value onto the Stack

POP – Removes the topmost value from the Stack

ESP – Should point to top of Stack

Example

```
#include<stdio.h>
Int main() {
 char ch[100];
 printf(“Hello World!\n”);
 gets(ch);
}
```

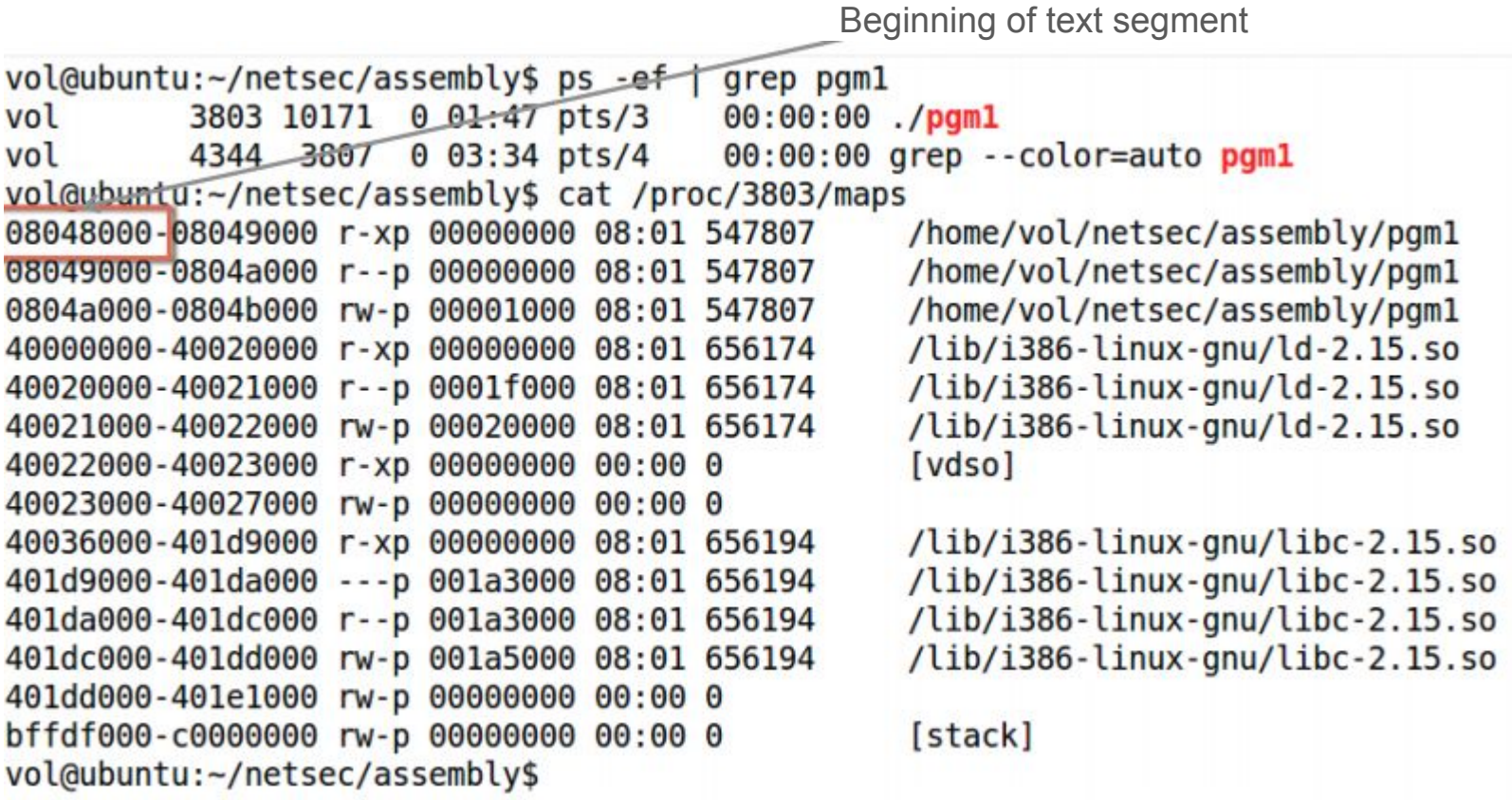
What happens here

Process Map

```
vol@ubuntu:~/netsec/assembly$ ps -ef | grep pgm1
vol       3803 10171  0 01:47 pts/3    00:00:00 ./pgm1
vol       4344  3807  0 03:34 pts/4    00:00:00 grep --color=auto pgm1
vol@ubuntu:~/netsec/assembly$ cat /proc/3803/maps
08048000-08049000 r-xp 00000000 08:01 547807     /home/vol/netsec/assembly/pgm1
08049000-0804a000 r--p 00000000 08:01 547807     /home/vol/netsec/assembly/pgm1
0804a000-0804b000 rw-p 00001000 08:01 547807     /home/vol/netsec/assembly/pgm1
40000000-40020000 r-xp 00000000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
40020000-40021000 r--p 0001f000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
40021000-40022000 rw-p 00020000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
40022000-40023000 r-xp 00000000 00:00 0          [vdso]
40023000-40027000 rw-p 00000000 00:00 0
40036000-401d9000 r-xp 00000000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401d9000-401da000 ---p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401da000-401dc000 r--p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401dc000-401dd000 rw-p 001a5000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401dd000-401e1000 rw-p 00000000 00:00 0
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
vol@ubuntu:~/netsec/assembly$
```

How the process is laid out in memory

Process Map



How the process is laid out in memory

Process Map

Beginning of text segment

```
vol@ubuntu:~/netsec/assembly$ ps -ef | grep pgm1
vol       3803 10171  0 01:47 pts/3    00:00:00 ./pgm1
vol       4344  3807  0 03:34 pts/4    00:00:00 grep --color=auto pgm1
vol@ubuntu:~/netsec/assembly$ cat /proc/3803/maps
08048000-08049000 r-xp 00000000 08:01 547807     /home/vol/netsec/assembly/pgm1
08049000-0804a000 r--p 00000000 08:01 547807     /home/vol/netsec/assembly/pgm1
0804a000-0804b000 rw-p 00001000 08:01 547807     /home/vol/netsec/assembly/pgm1
40000000-40020000 r-xp 00000000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
40020000-40021000 r--p 0001f000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
40021000-40022000 rw-p 00020000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
40022000-40023000 r-xp 00000000 00:00 0          [vdso]
40023000-40027000 rw-p 00000000 00:00 0
40036000-401d9000 r-xp 00000000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401d9000-401da000 ---p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401da000-401dc000 r--p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401dc000-401dd000 rw-p 001a5000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401dd000-401e1000 rw-p 00000000 00:00 0
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
vol@ubuntu:~/netsec/assembly$
```

How the process is laid out in memory

Process Map

Read only data segment

```
vol@ubuntu:~/netsec/assembly$ ps -ef | grep pgm1
vol       3803 10171  0 01:47 pts/3    00:00:00 ./pgm1
vol       4344  3807  0 03:34 pts/4    00:00:00 grep --color=auto pgm1
vol@ubuntu:~/netsec/assembly$ cat /proc/3803/maps
08048000-08049000 r-xp 00000000 08:01 547807     /home/vol/netsec/assembly/pgm1
08049000-0804a000 r--p 00000000 08:01 547807     /home/vol/netsec/assembly/pgm1
0804a000-0804b000 rw-p 00001000 08:01 547807     /home/vol/netsec/assembly/pgm1
40000000-40020000 r-xp 00000000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
40020000-40021000 r--p 0001f000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
40021000-40022000 rw-p 00020000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
40022000-40023000 r-xp 00000000 00:00 0          [vdso]
40023000-40027000 rw-p 00000000 00:00 0
40036000-401d9000 r-xp 00000000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401d9000-401da000 ---p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401da000-401dc000 r--p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401dc000-401dd000 rw-p 001a5000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401dd000-401e1000 rw-p 00000000 00:00 0
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
vol@ubuntu:~/netsec/assembly$
```

How the process is laid out in memory

Process Map

Read write data segment

```
vol@ubuntu:~/netsec/assembly$ ps -ef | grep pgm1
vol       3803 10171  0 01:47 pts/3    00:00:00 ./pgm1
vol       4344  3807  0 03:34 pts/4    00:00:00 grep --color=auto pgm1
vol@ubuntu:~/netsec/assembly$ cat /proc/3803/maps
08048000-08049000 r-xp 00000000 08:01 547807     /home/vol/netsec/assembly/pgm1
08049000-0804a000 r--p 00000000 08:01 547807     /home/vol/netsec/assembly/pgm1
0804a000-0804b000 rw-p 00001000 08:01 547807     /home/vol/netsec/assembly/pgm1
40000000-40020000 r-xp 00000000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
40020000-40021000 r--p 0001f000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
40021000-40022000 rw-p 00020000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
40022000-40023000 r-xp 00000000 00:00 0          [vdso]
40023000-40027000 rw-p 00000000 00:00 0
40036000-401d9000 r-xp 00000000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401d9000-401da000 ---p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401da000-401dc000 r--p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401dc000-401dd000 rw-p 001a5000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
401dd000-401e1000 rw-p 00000000 00:00 0
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
vol@ubuntu:~/netsec/assembly$
```

How the process is laid out in memory

What you're going to learn

```
#include <stdio.h>
int main(){
printf("Hello World!\n");
return 0x1234;
}
```

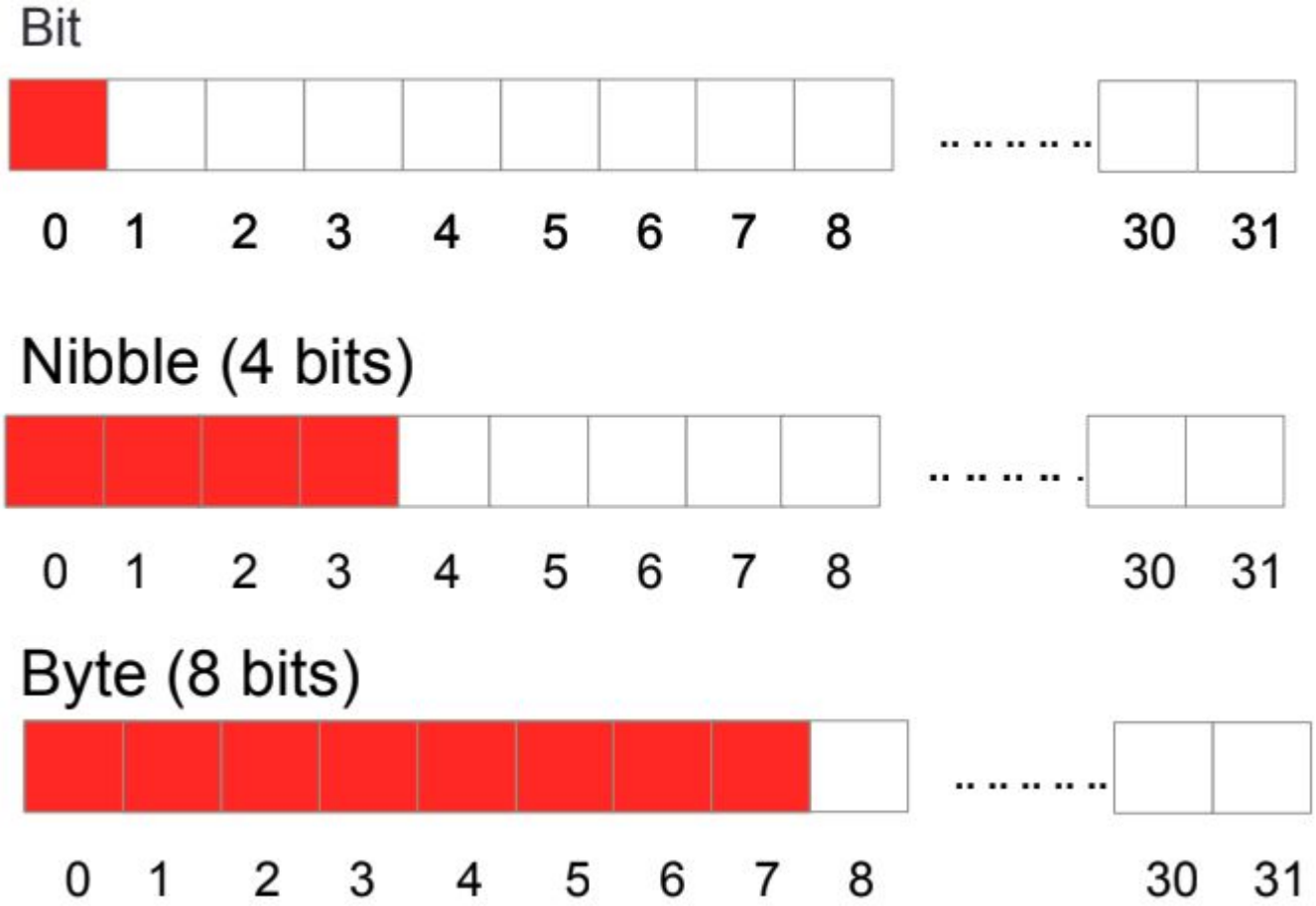
What is the instruction corresponding to printf

Ubuntu 8.04, GCC 4.2.4 Disassembled with

"objdump -d"

```
08048374 <main>:
 8048374:       8d 4c 24 04             lea    0x4(%esp),%ecx
 8048378:       83 e4 f0                and    $0xfffffff0,%esp
 804837b:       ff 71 fc                 pushl  -0x4(%ecx)
 804837e:       55                       push   %ebp
 804837f:       89 e5                    mov    %esp,%ebp
 8048381:       51                       push   %ecx
 8048382:       83 ec 04                 sub    $0x4,%esp
 8048385:       c7 04 24 60 84 04 08     movl   $0x8048460,(%esp)
 804838c:       e8 43 ff ff ff           call   80482d4 <puts@plt>
 8048391:       b8 2a 00 00 00           mov    $0x1234,%eax
 8048396:       83 c4 04                 add    $0x4,%esp
 8048399:       59                       pop    %ecx
 804839a:       5d                       pop    %ebp
 804839b:       8d 61 fc                lea    -0x4(%ecx),%esp
 804839e:       c3                       ret
 804839f:       90                       nop
```

Data sizes


Bit
0 1 2 3 4 5 6 7 8
30 31
Nibble (4 bits)
0 1 2 3 4 5 6 7 8
30 31
Byte (8 bits)
0 1 2 3 4 5 6 7 8
30 31

Data Sizes

Word (2 bytes or 16 bits)

0 15

31

Doubleword (4 bytes or 32 bits)

0 15

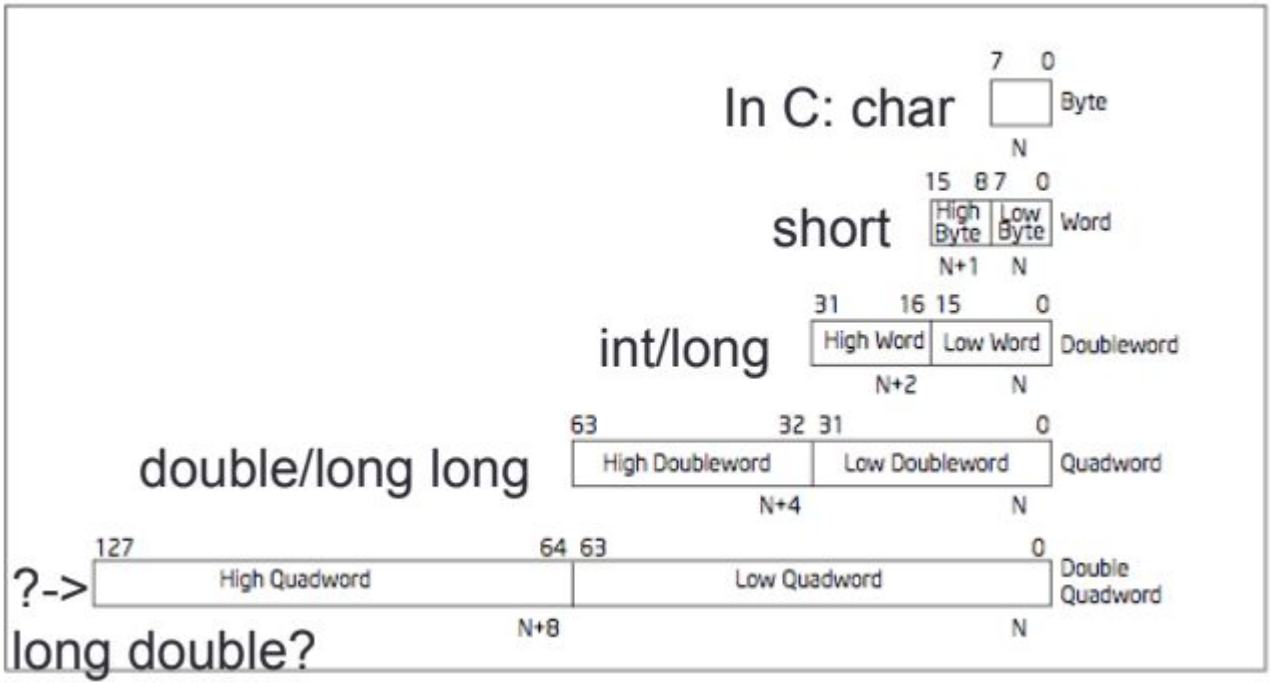
31

Quadword (8 bytes or 64 bits)

0 31

63

Data Types


In C: char
7 0
Byte
N
short
15 8 7 0
High Byte
Low Byte
Word
N+1 N
int/long
31 16 15 0
High Word
Low Word
Doubleword
N+2 N
double/long long
63 32 31 0
High Doubleword
Low Doubleword
Quadword
N+4 N
?->
127 64 63 0
High Quadword
Low Quadword
Double Quadword
N+8 N
long double?

Example – Data Types

```c
#include <stdio.h>

int main() {

 printf("sizeof char: %d \n", sizeof(char));
 printf("sizeof int: %d \n", sizeof(int));
 printf("sizeof unsigned int: %d \n", sizeof(unsigned int));
 printf("sizeof long: %d \n", sizeof(long));
 printf("sizeof long long: %d \n", sizeof(long long));
 printf("sizeof double: %d \n", sizeof(double));
 printf("sizeof long double: %d \n", sizeof(long double));
}
```

Negative Numbers

- “one's complement” = flip all bits. 0->1, 1->0
- “two's complement” = one's complement + 1
- Negative numbers are defined as the “two's complement” of the positive number

Number	One's Comp.	Two's Comp. (negative)
00000001b : 0x01	11111110b : 0xFE	11111111b : 0xFF : -1
00000100b : 0x04	11111011b : 0xFB	11111100b : 0xFC : -4
00011010b : 0x1A	11100101b : 0xE5	11100110b : 0xE6 : -26
?	?	10110000b : 0xB0 : -?

- 0x01 to 0x7F positive byte, 0x80 to 0xFF negative byte
- 0x00000001 to 0x7FFFFFFF positive dword
- 0x80000000 to 0xFFFFFFFF negative dword

x86 instruction: NOP

- NOP - No Operation! No registers, no values, no nothin'!
- Just there to pad/align bytes, or to delay time
- Bad guys use it to make simple exploits more reliable.
- “The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.”

The Stack

- The stack is a conceptual area of main memory (RAM) which is designated by the OS when a program is started.
  - Different OS start it at different addresses by convention
- A stack is a Last-In-First-Out (LIFO/FILO) data structure where data is "pushed" on to the top of the stack and "popped" off the top.
- By convention the stack grows toward lower memory addresses. Adding something to the stack means the top of the stack is now at a lower memory address.

The Stack

- As already mentioned, esp points to the top of the stack, the lowest address which is being used
  - While data will exist at addresses beyond the top of the stack, it is considered undefined
- The stack keeps track of which functions were called before the current one, it holds local variables and is frequently used to pass arguments to the next function to be called.
- A firm understanding of what is happening on the stack is essential to understanding a program's operation.

PUSH - Push Word, Doubleword or Quadword onto the Stack

- For our purposes, it will always be a DWORD (4 bytes).
  - Can either be an immediate (a numeric constant), or the value in a register
- The push instruction automatically decrements the stack pointer, esp, by 4.

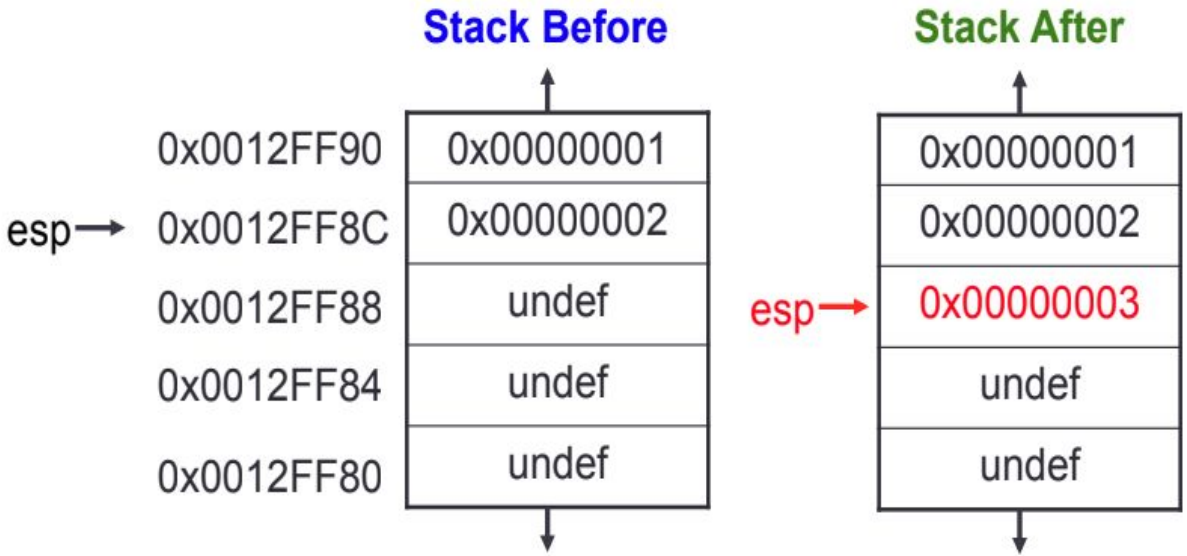
Registers Before

eax	0x00000003
esp	0x0012FF8C

push eax

Registers After

eax	0x00000003
esp	0x0012FF88



POP- Pop a Value from the Stack

- Take a DWORD off the stack, put it in a register, and increment esp by 4

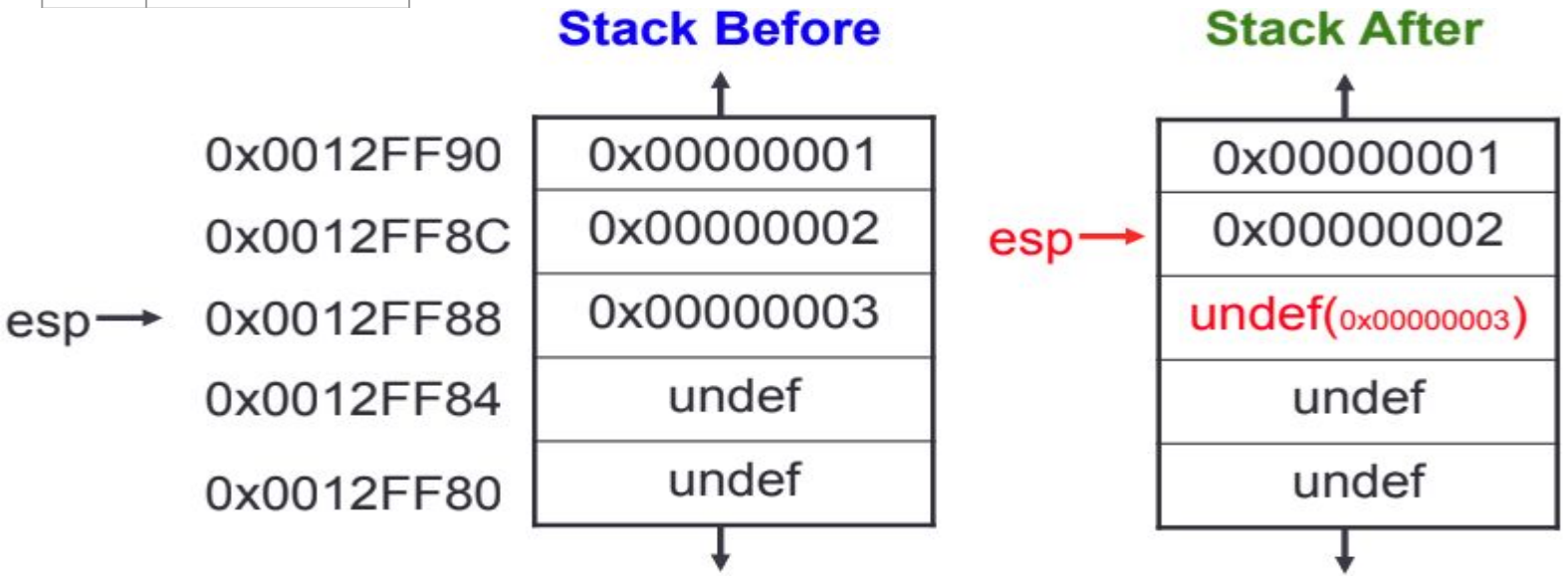
Registers Before

eax	0xFFFFFFFF
esp	0x0012FF88

pop eax

Registers After

eax	0x00000003
esp	0x0012FF8C



Assembly Language Syntax

- Intel
- AT&T

Assembly Language Syntax

- Register Prefixes
  - Intel has no register or immediate prefixes
  - AT&T registers are prefixed by ‘%’ and immediates are prefixed by$
  - Immediates in Intel
    - Hex or binary immediate data suffixed with ‘h’ or ‘b’
    - If the first hex digit is a letter, then the value is prefixed by a ‘0’

Intel Syntax	AT&T Syntax
mov eax, 1	mov $1, %eax
mov ebx, 0ffh	movl $0xff, %ebx
int 80h	int $0x80

Assembly Language Syntax

- Direction of Operands
  - Intel syntax is opposite that of ATT
  - Intel – first operand is destination, second is source
  - ATT – first operand is source, second is destination
    - Advantage of ATT- we read and write from left to right hence natural

Intel Syntax	AT&T Syntax
mov eax, 1	mov $1, %eax
mov ebx, 0ffh	movl $0xff, %ebx
int 80h	int $0x80

Assembly Language Syntax

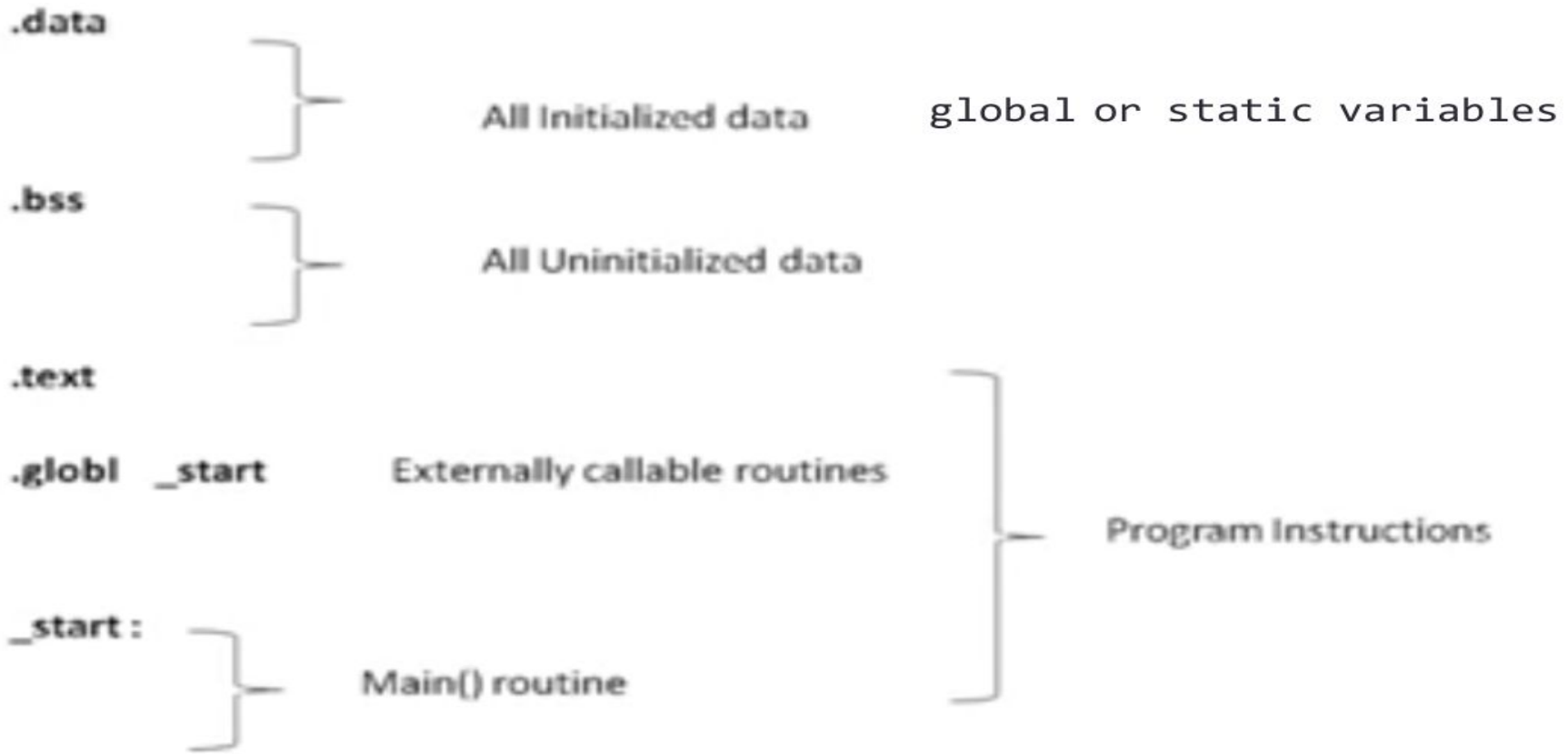
- Memory Operands
  - Intel syntax base register is enclosed in '[' ']'
  - In ATT syntax base register is enclosed in '(' ')'

Intel Syntax	AT&T Syntax
mov eax, [ebx]	mov (%ebx), %eax
mov eax, [ebx+3]	movl 3(%ebx), %eax

- Suffixes
  - In ATT syntax mnemonics have a suffix
  - Significance of suffix is operand size: 'l' for long 'w' for word, 'b' for byte

Intel Syntax	AT&T Syntax
mov al, bl	movb %bl, %al
mov ax, bx	movw %bx, %ax
mov eax, ebx	movl %ebx, %eax
mov eax, dword ptr [ebx]	movl (%ebx), %eax

Structure of an Assembly Program



Declaring Static Data Regions

- Analogous to global variables
- Must be preceded by .data directives
- .byte, .short, .long used to declare 1,2,4 byte locations
- Labels can be used to give names to memory locations

```
.data
var:
        .byte 64        /* Declare a byte, referred to as location var, containing the value 64. */
        .byte 10        /* Declare a byte with no label, containing the value 10. Its location is var + 1. */
x:
        .short 42       /* Declare a 2-byte value initialized to 42, referred to as location x. */
y:
        .long 30000     /* Declare a 4-byte value, referred to as location y, initialized to 30000. */
```

Declaring Static Data Regions

- Arrays are a number of cells in contiguous memory locations
  - Declared by simply listing the values
  - Array of bytes can be represented as string literals
  - .zero directive used to fill memory with zeros

```
s:
        .long 1, 2, 3          /* Declare three 4-byte values, initialized to 1, 2, and 3.
                                  The value at location s + 8 will be 3. */
barr:
        .zero 10               /* Declare 10 bytes starting at location barr, initialized to 0. */
str:
        .string "hello"        /* Declare 6 bytes starting at the address str initialized to
                                  the ASCII character values for hello followed by a nul (0) byte. */
```

Data Types in .DATA

- Data types that can be used in .data segment
- Space reserved at compile time
  - .byte = 1 byte
  - .ascii = string (typically used when aligning across word boundaries)
  - .asciz = Null terminated string (also called C strings)
  - .int = 32 bit integer
  - .short = 16 bit
  - .float = floating point number
  - .double = double precision floating point

Addressing Memory

- Memory addresses are 32 bit wides, x86 processors are capable of 2 32 bytes of memory.
- Labels are replaced by memory addresses

Instructions

- 3 categories – data movement, arithmetic/logic, control- flow.
- Notation:

`<reg32>`	Any 32-bit register (`%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%esp`, or `%ebp`)
`<reg16>`	Any 16-bit register (`%ax`, `%bx`, `%cx`, or `%dx`)
`<reg8>`	Any 8-bit register (`%ah`, `%bh`, `%ch`, `%dh`, `%al`, `%bl`, `%cl`, or `%dl`)
`<reg>`	Any register
`<mem>`	A memory address (e.g., `(%eax)`, `4+var(,1)`, or `(%eax,%ebx,1)`)
`<con32>`	Any 32-bit immediate
`<con16>`	Any 16-bit immediate
`<con8>`	Any 8-bit immediate
`<con>`	Any 8-, 16-, or 32-bit immediate

Data Movement Instructions

- Push
- Pop
- Mov
- lea

MOV - Move

- Can move:
  - mov <reg> <reg>
  - mov <reg> <mem>
  - mov <mem> <reg>
  - mov <const> <reg>
  - mov <const> <mem>
- Never memory to memory!
  - Source memory has to be copied to register first
- The instruction has 2 operands: first is source and second is destination

Mov- Example

- mov (%ebx), %eax ;Load contents (4 bytes) at memory address in %ebx into %eax
- mov -4(%esi), %eax ; Load contents at memory address esi+ (-4) into %eax
- mov (%esi, %ebx, 4), %edx ; Move 4 bytes of data at (ESI + (EBX * 4)) into EDX. The 4 is a scalar multiplier

Mov- Opcode Suffixes

- Previous cases the data sizes was implicit since memory and registers are 32 bit.
- What if you want to: mov $2 ,(%eax)
  - Should the assembler move 32 bit representation of 2 or a byte 2?
- 3 variants
  - movb $2, (%ebx) ; move single byte into addr stored in ebx
  - movw $2, (%ebx) ; move 16-bit representation of 2 into 2 bytes starting at address in %ebx
  - movl $2, (%ebx) ; move 32-bit representation

Demo

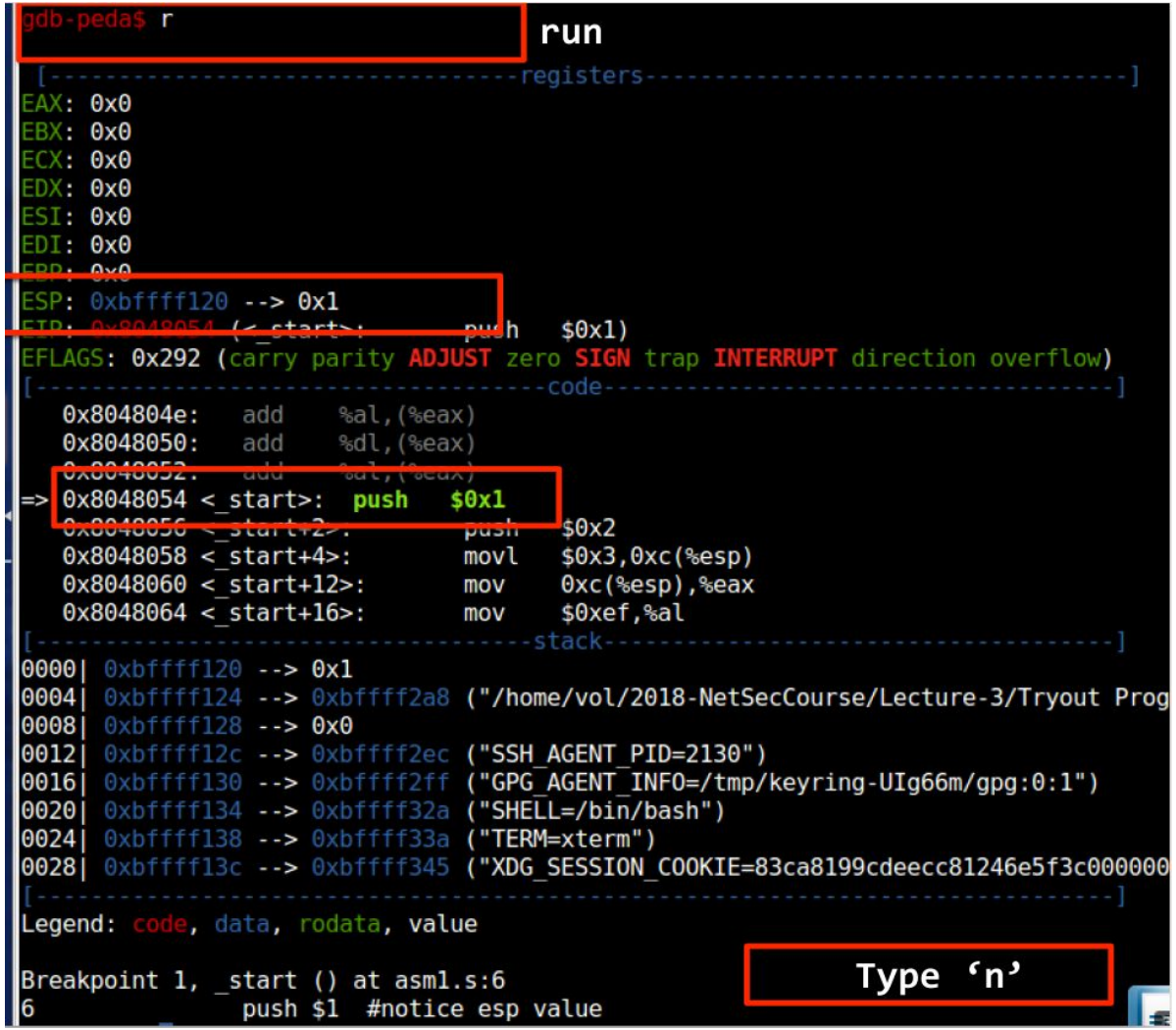
- Asm1.s
  - Difference between push and mov
  - Understanding mov instructions
  - Given Makefile to compile

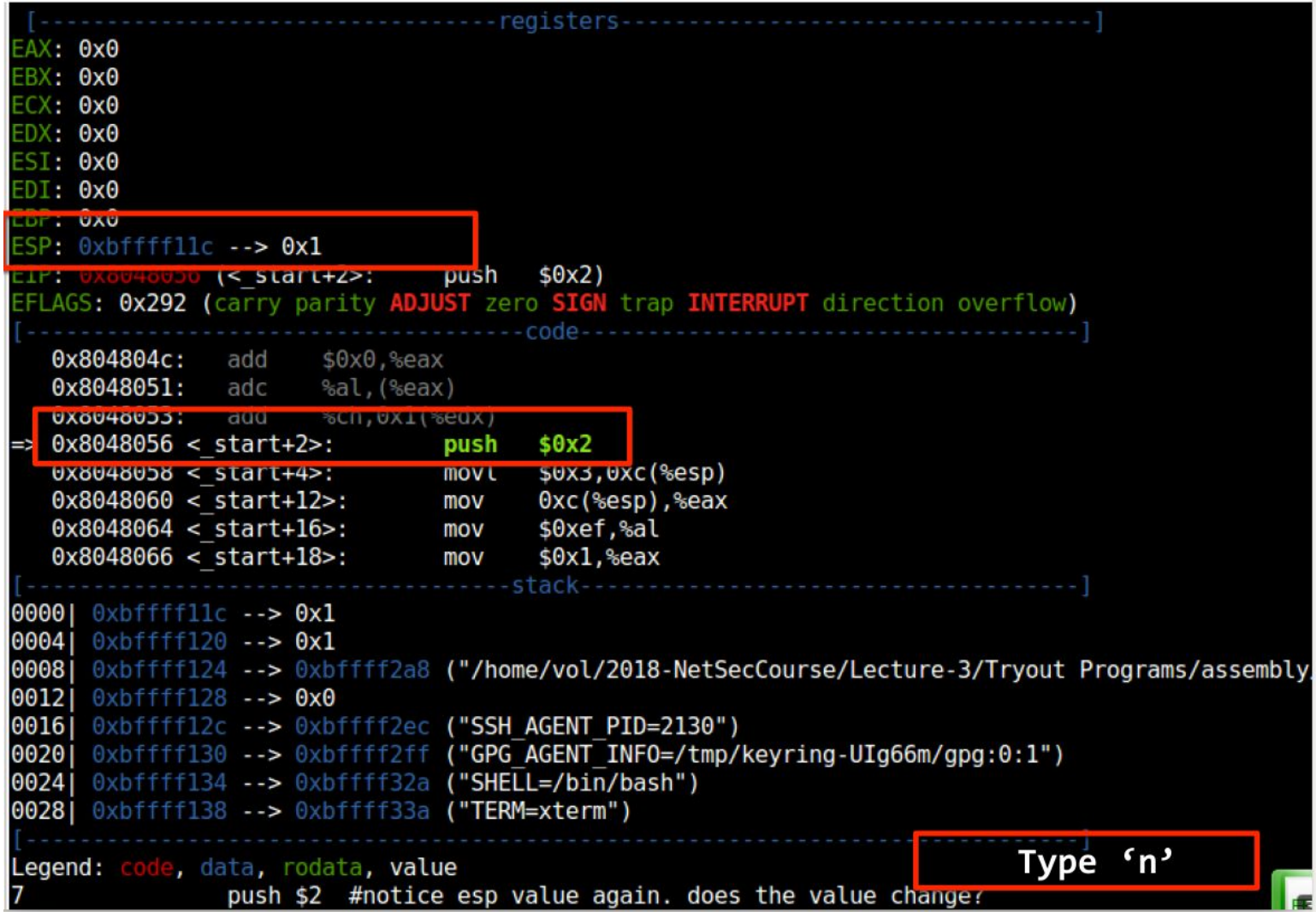
asm1

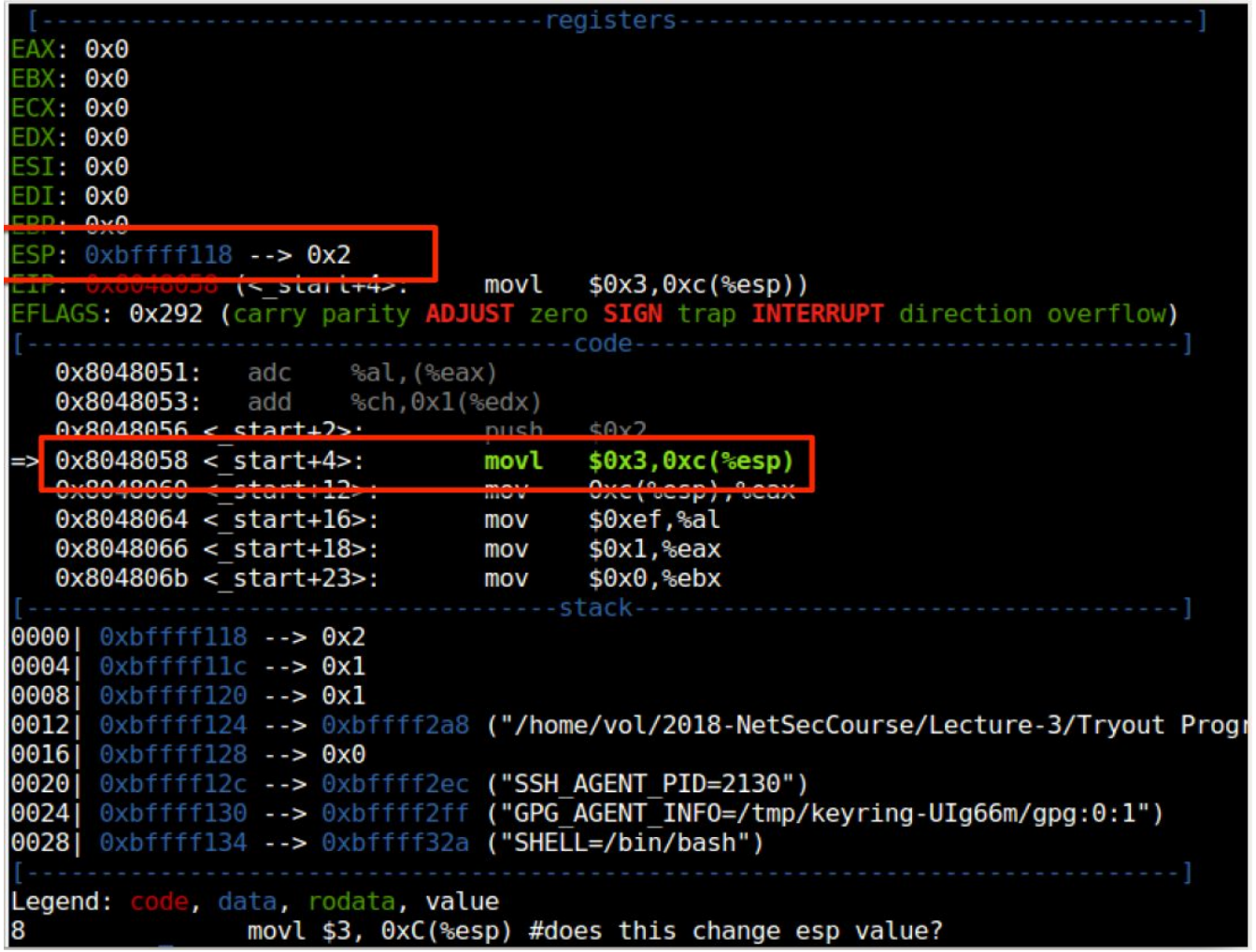
```
gdb-peda$ disass _start
Dump of assembler code for function _start:
   0x08048054 <+0>:     push   $0x1
   0x08048056 <+2>:     push   $0x2
   0x08048058 <+4>:     movl   $0x3,0xc(%esp)
   0x08048060 <+12>:    mov    0xc(%esp),%eax
   0x08048064 <+16>:    mov    $0xef,%al
   0x08048066 <+18>:    mov    $0x1,%eax
   0x0804806b <+23>:    mov    $0x0,%ebx
   0x08048070 <+28>:    int    $0x80
End of assembler dump.
gdb-peda$ b *0x08048054
Breakpoint 2 at 0x8048054: file asm1.s, line 6.
```

Disassemble start function and break at first instruction

asm1


gdb-peda$ r
run
[----------registers----------]
EAX: 0x0
EBX: 0x0
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x0
ESP: 0xbffff120 --> 0x1
EIP: 0x8048054 (<_start>: push $0x1)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[----------code----------]
0x804804e: add %al,(%eax)
0x8048050: add %dl,(%eax)
=> 0x8048054 <_start>: push $0x1
0x8048056 <_start+2>: push $0x2
0x8048058 <_start+4>: movl $0x3,0xc(%esp)
0x8048060 <_start+12>: mov 0xc(%esp),%eax
0x8048064 <_start+16>: mov $0xef,%al
[----------stack----------]
0000| 0xbffff120 --> 0x1
0004| 0xbffff124 --> 0xbffff2a8 ("/home/vol/2018-NetSecCourse/Lecture-3/Tryout Prog
0008| 0xbffff128 --> 0x0
0012| 0xbffff12c --> 0xbffff2ec ("SSH_AGENT_PID=2130")
0016| 0xbffff130 --> 0xbffff2ff ("GPG_AGENT_INFO=/tmp/keyring-UIg66m/gpg:0:1")
0020| 0xbffff134 --> 0xbffff32a ("SHELL=/bin/bash")
0024| 0xbffff138 --> 0xbffff33a ("TERM=xterm")
0028| 0xbffff13c --> 0xbffff345 ("XDG_SESSION_COOKIE=83ca8199cdeecc81246e5f3c000000
Legend: code, data, rodata, value
Breakpoint 1, _start () at asm1.s:6
6 push $1 #notice esp value
Type 'n'


[----------registers----------]
EAX: 0x0
EBX: 0x0
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x0
ESP: 0xbffff11c --> 0x1
EIP: 0x8048056 (<_start+2>: push $0x2)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[----------code----------]
0x804804c: add $0x0,%eax
0x8048051: adc %al,(%eax)
0x8048053: add %ch,0x1(%edx)
=> 0x8048056 <_start+2>: push $0x2
0x8048058 <_start+4>: movl $0x3,0xc(%esp)
0x8048060 <_start+12>: mov 0xc(%esp),%eax
0x8048064 <_start+16>: mov $0xef,%al
0x8048066 <_start+18>: mov $0x1,%eax
[----------stack----------]
0000| 0xbffff11c --> 0x1
0004| 0xbffff120 --> 0x1
0008| 0xbffff124 --> 0xbffff2a8 ("/home/vol/2018-NetSecCourse/Lecture-3/Tryout Programs/assembly
0012| 0xbffff128 --> 0x0
0016| 0xbffff12c --> 0xbffff2ec ("SSH_AGENT_PID=2130")
0020| 0xbffff130 --> 0xbffff2ff ("GPG_AGENT_INFO=/tmp/keyring-UIg66m/gpg:0:1")
0024| 0xbffff134 --> 0xbffff32a ("SHELL=/bin/bash")
0028| 0xbffff138 --> 0xbffff33a ("TERM=xterm")
[----------]
Legend: code, data, rodata, value
7 push $2 #notice esp value again. does the value change?
Type 'n'


[----------registers----------]
EAX: 0x0
EBX: 0x0
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x0
ESP: 0xbffff118 --> 0x2
EIP: 0x8048058 (<_start+4>: movl $0x3,0xc(%esp))
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[----------code----------]
0x8048051: adc %al,(%eax)
0x8048053: add %ch,0x1(%edx)
0x8048056 <_start+2>: push $0x2
=> 0x8048058 <_start+4>: movl $0x3,0xc(%esp)
0x8048060 <_start+12>: mov 0xc(%esp),%eax
0x8048064 <_start+16>: mov $0xef,%al
0x8048066 <_start+18>: mov $0x1,%eax
0x804806b <_start+23>: mov $0x0,%ebx
[----------stack----------]
0000| 0xbffff118 --> 0x2
0004| 0xbffff11c --> 0x1
0008| 0xbffff120 --> 0x1
0012| 0xbffff124 --> 0xbffff2a8 ("/home/vol/2018-NetSecCourse/Lecture-3/Tryout Progr
0016| 0xbffff128 --> 0x0
0020| 0xbffff12c --> 0xbffff2ec ("SSH_AGENT_PID=2130")
0024| 0xbffff130 --> 0xbffff2ff ("GPG_AGENT_INFO=/tmp/keyring-UIg66m/gpg:0:1")
0028| 0xbffff134 --> 0xbffff32a ("SHELL=/bin/bash")
[----------]
Legend: code, data, rodata, value
8 movl $3, 0xC(%esp) #does this change esp value?

asm1

Stack before execution of movl $0x3, 0xC(%esp)

```
gdb-peda$ x/20wx $esp
0xbffff118:     0x00000002      0x00000001      0x00000001      0xbffff2a8
0xbffff128:     0x00000000      0xbffff2ec      0xbffff2ff      0xbffff32a
0xbffff138:     0xbffff33a      0xbffff345      0xbffff396      0xbffff3a8
0xbffff148:     0xbffff3d2      0xbffff5e2      0xbffff5f0      0xbffff5fb
0xbffff158:     0xbffff604      0xbffffb25      0xbffffb36      0xbffffb41
```

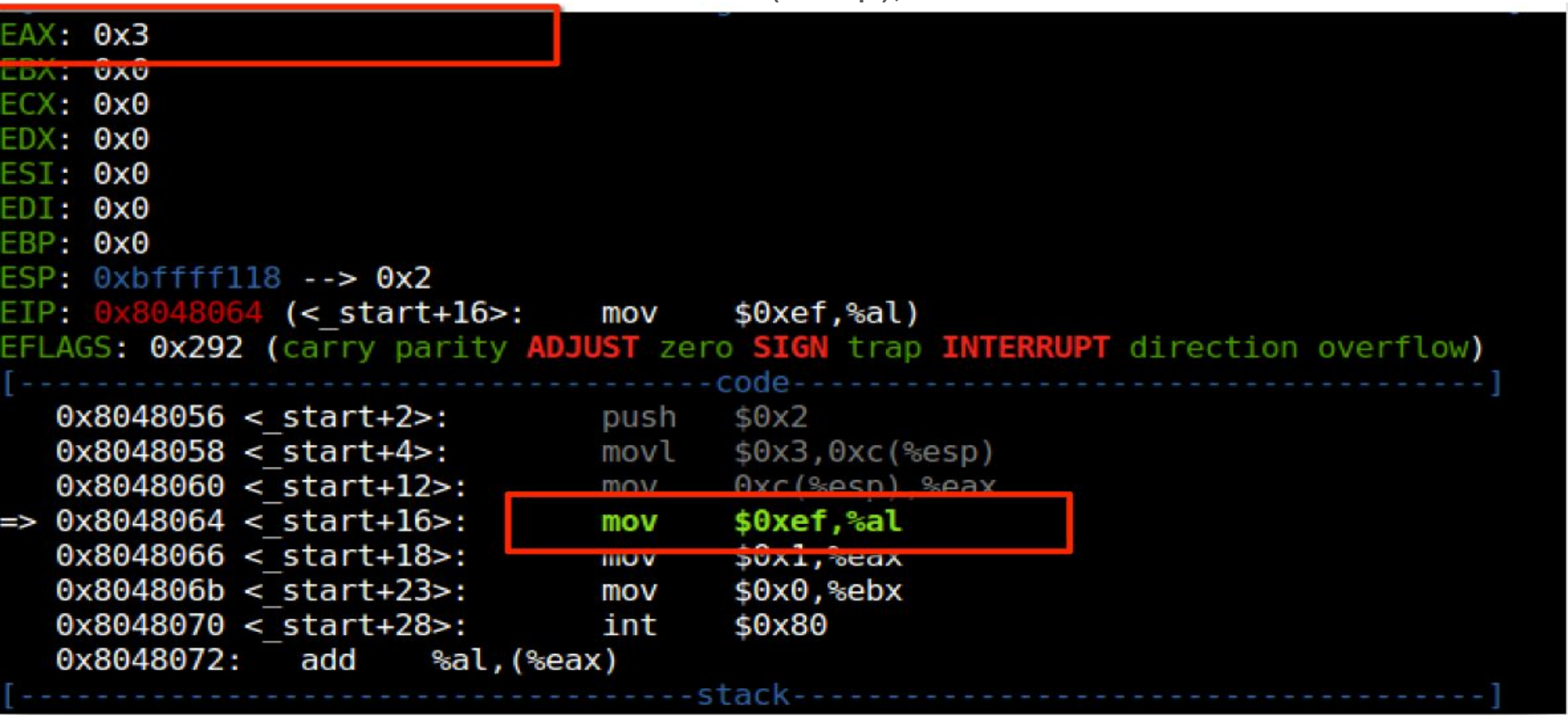
Next instruction is movl $0x3, 0xC(%esp)

```
gdb-peda$ p $esp
$1 = (void *) 0xbffff118
gdb-peda$ x/20wx $esp
0xbffff118:     0x00000002      0x00000001      0x00000001      0x00000003
0xbffff128:     0x00000000      0xbffff2ec      0xbffff2ff      0xbffff32a
0xbffff138:     0xbffff33a      0xbffff345      0xbffff396      0xbffff3a8
0xbffff148:     0xbffff3d2      0xbffff5e2      0xbffff5f0      0xbffff5fb
0xbffff158:     0xbffff604      0xbffffb25      0xbffffb36      0xbffffb41
```

What is the difference between push and mov?

asm1

After execution of next instruction movl 0xC(%esp), %eax



mov

- mov 0xC(%esp), %eax moves value in address %esp + 0xC to %eax
- E.g. Assume,

  %eax = 0xfffffe8

  %ebx = 0

  0xfffffe8 : 0xaabbccdd

mov %eax, %ebx changes %ebx = 0xfffffe8

mov (%eax), %ebx changes %ebx = 0xaabbccdd

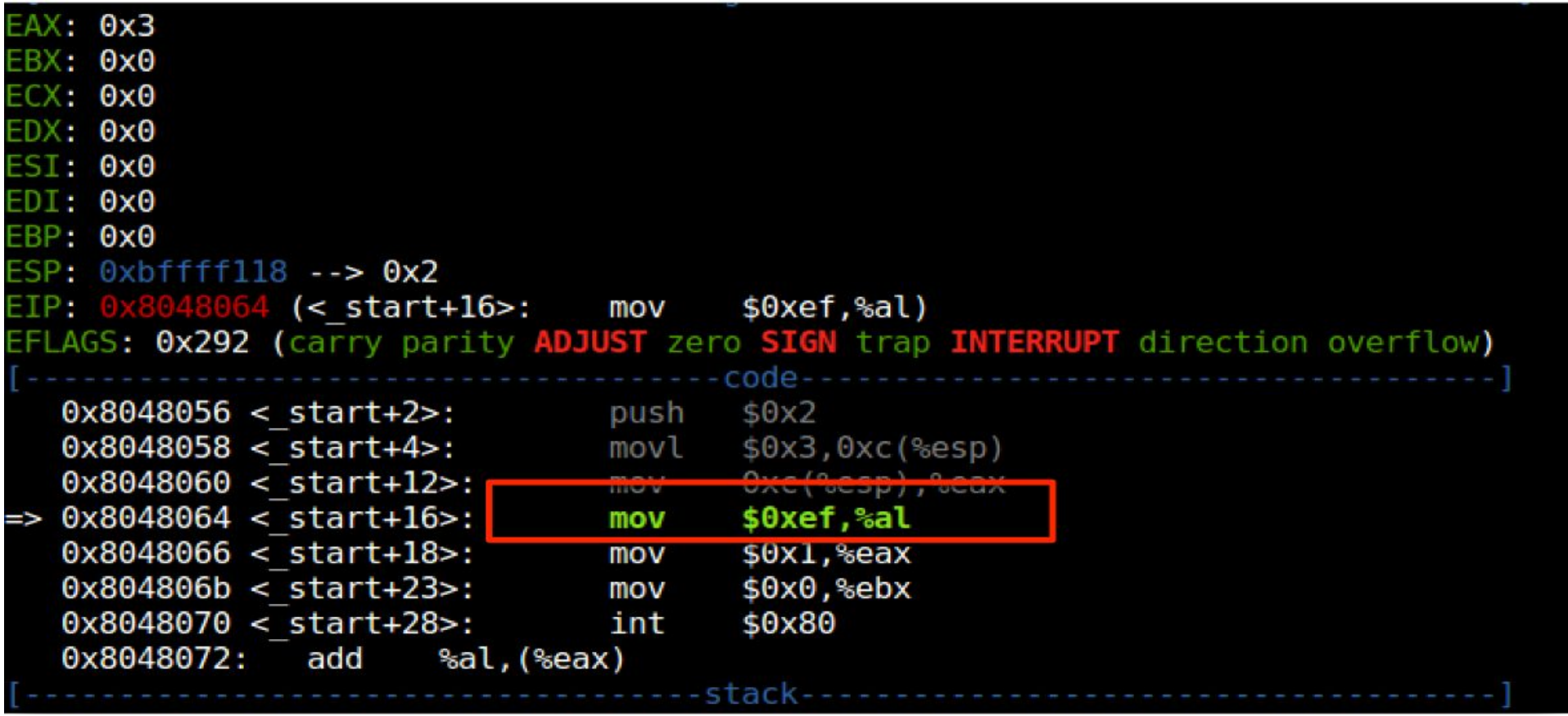
Demo asm1

- What will happen with this instruction?

  movb 0xdeadbeef into %eax

asm1

Next instruction is movb 0xdeadbeef, %eax



asm1

Next instruction is movb 0xdeadbeef, %eax is translated to mov 0xef, %al

```
EAX: 0xef
EBX: 0x0
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x0
ESP: 0xbffff118 --> 0x2
EIP: 0x8048066 (<_start+18>:    mov    $0x1,%eax)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048058 <_start+4>:        movl   $0x3,0xc(%esp)
   0x8048060 <_start+12>:       mov    0xc(%esp),%eax
   0x8048064 <_start+16>:       mov    $0xef,%al
=> 0x8048066 <_start+18>:       mov    $0x1,%eax
   0x804806b <_start+23>:       mov    $0x0,%ebx
   0x8048070 <_start+28>:       int    $0x80
   0x8048072:   add    %al,(%eax)
   0x8048074:   add    %al,(%eax)
```

Demo asm1

Making the file throws a warning

make asm1

```
as -o asm1.o -ggstabs asm1.s
asm1.s: Assembler messages:
asm1.s:12: Warning: using `%al' instead of `%eax' due to `b' suffix
asm1.s:12: Warning: ffffffffdeadbeef shortened to 00000000000000ef
```

Lea

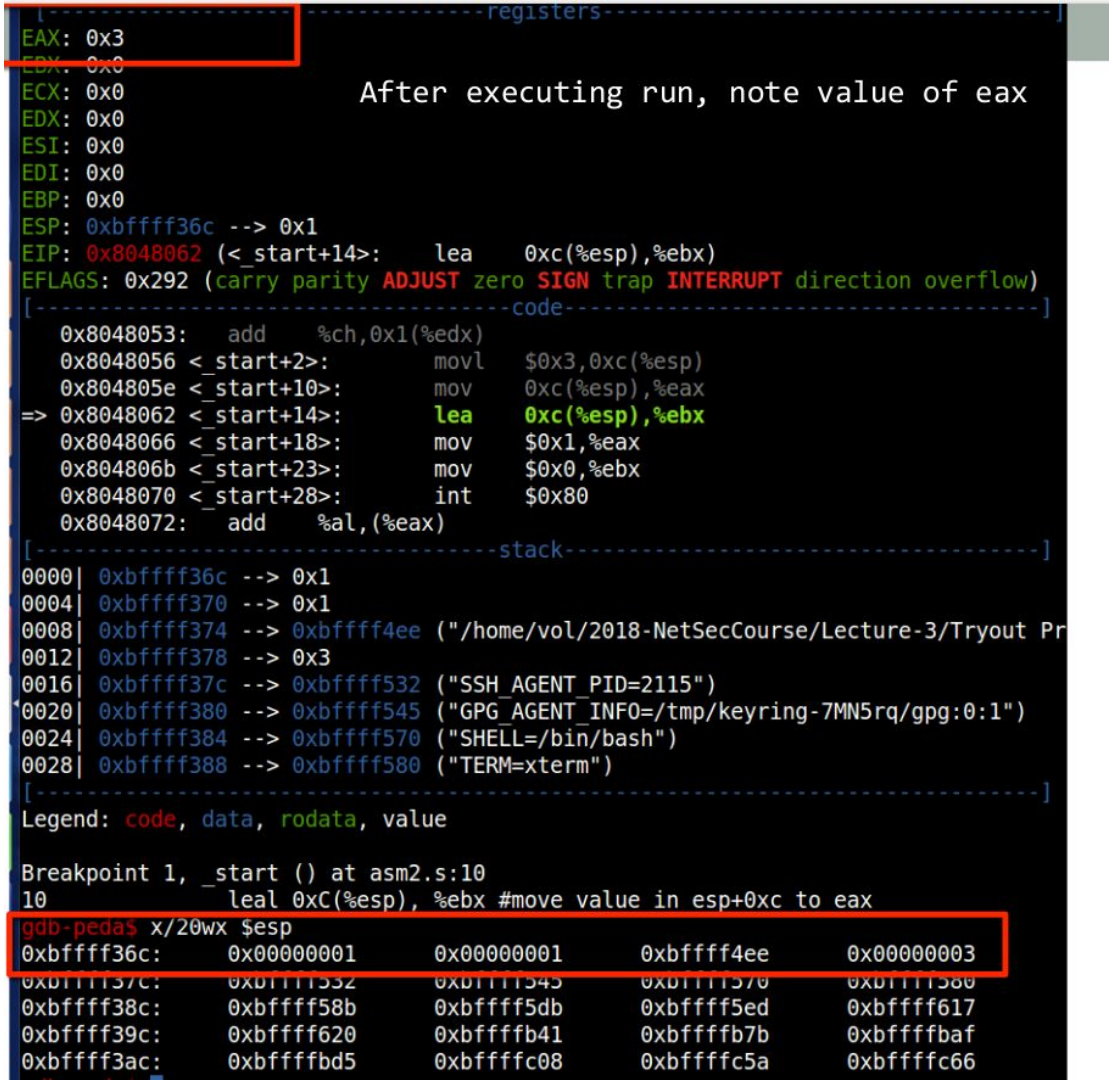
- Load Effective Address
- Place address specified by the first operand into the register specified by its second operand
- Mov instruction loads the value, Lea loads address
- Source operand is a memory address and destination is a register

Demo asm2

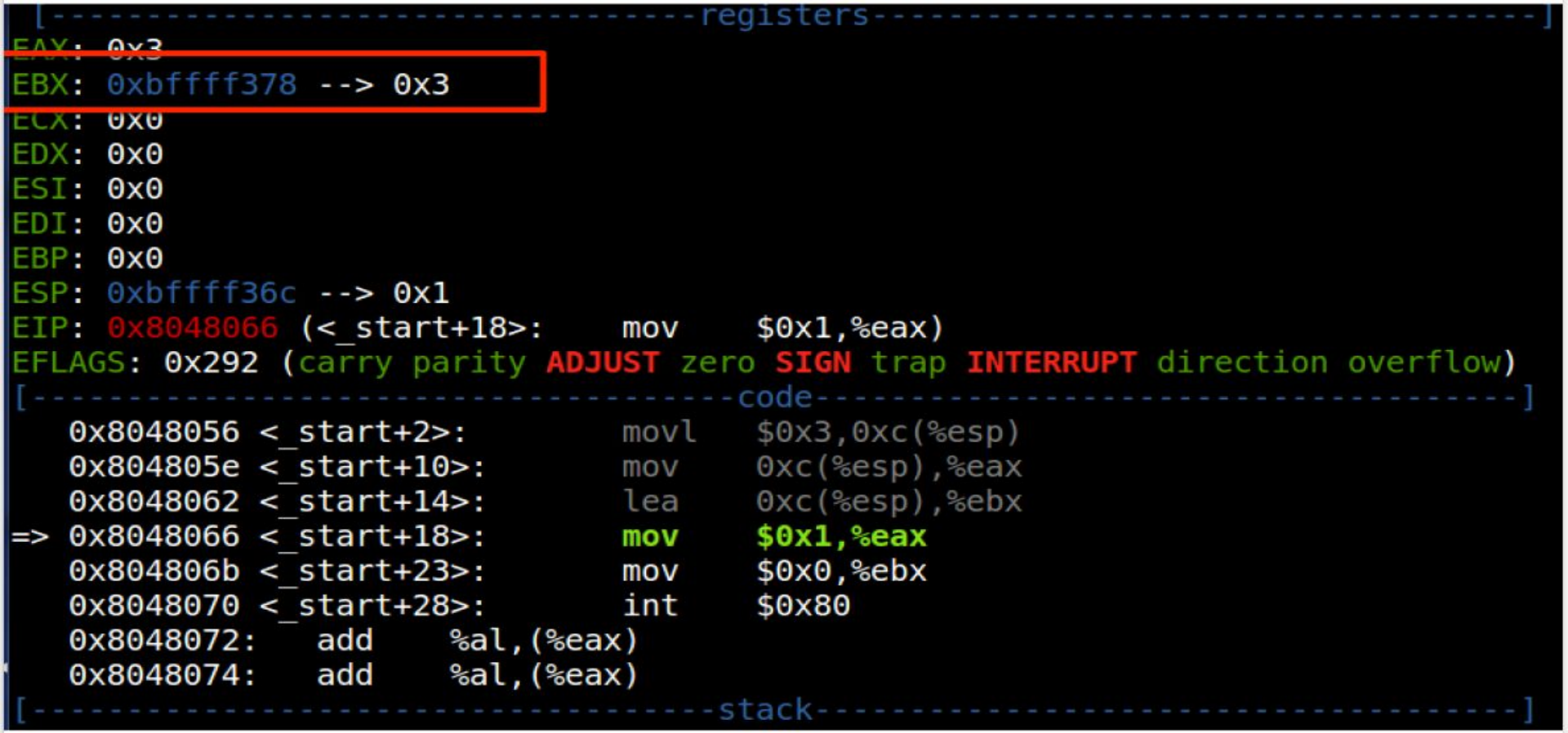
What is the difference between mov and lea

```
gdb-peda$ disass _start
Dump of assembler code for function _start:
   0x08048054 <+0>:     push   $0x1
   0x08048056 <+2>:     movl   $0x3,0xc(%esp)
   0x0804805e <+10>:    mov    0xc(%esp),%eax
   0x08048062 <+14>:    lea    0xc(%esp),%ebx
   0x08048066 <+18>:    mov    $0x1,%eax
   0x0804806b <+23>:    mov    $0x0,%ebx
   0x08048070 <+28>:    int    $0x80
End of assembler dump.
gdb-peda$ b *0x08048062
Breakpoint 2 at 0x8048062: file asm2.s, line 10.
```

asm2


After executing run, note value of eax
EAX: 0x3
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x0
ESP: 0xbffff36c --> 0x1
EIP: 0x8048062 (<_start+14>: lea 0xc(%esp),%ebx)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
0x8048053: add %ch,0x1(%edx)
0x8048056 <_start+2>: movl $0x3,0xc(%esp)
0x804805e <_start+10>: mov 0xc(%esp),%eax
=> 0x8048062 <_start+14>: lea 0xc(%esp),%ebx
0x8048066 <_start+18>: mov $0x1,%eax
0x804806b <_start+23>: mov $0x0,%ebx
0x8048070 <_start+28>: int $0x80
0x8048072: add %al,(%eax)
0000| 0xbffff36c --> 0x1
0004| 0xbffff370 --> 0x1
0008| 0xbffff374 --> 0xbffff4ee ("/home/vol/2018-NetSecCourse/Lecture-3/Tryout Pr
0012| 0xbffff378 --> 0x3
0016| 0xbffff37c --> 0xbffff532 ("SSH_AGENT_PID=2115")
0020| 0xbffff380 --> 0xbffff545 ("GPG_AGENT_INFO=/tmp/keyring-7MN5rq/gpg:0:1")
0024| 0xbffff384 --> 0xbffff570 ("SHELL=/bin/bash")
0028| 0xbffff388 --> 0xbffff580 ("TERM=xterm")
Legend: code, data, rodata, value
Breakpoint 1, _start () at asm2.s:10
10 leal 0xC(%esp), %ebx #move value in esp+0xc to eax
gdb-peda$ x/20wx $esp
0xbffff36c: 0x00000001 0x00000001 0xbffff4ee 0x00000003
0xbffff38c: 0xbffff58b 0xbffff5db 0xbffff5ed 0xbffff617
0xbffff39c: 0xbffff620 0xbffffb41 0xbffffb7b 0xbffffbaf
0xbffff3ac: 0xbffffbd5 0xbffffc08 0xbffffc5a 0xbffffc66

asm2


registers
EAX: 0x3
EBX: 0xbffff378 --> 0x3
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x0
ESP: 0xbffff36c --> 0x1
EIP: 0x8048066 (<_start+18>: mov $0x1,%eax)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
code
0x8048056 <_start+2>: movl $0x3,0xc(%esp)
0x804805e <_start+10>: mov 0xc(%esp),%eax
0x8048062 <_start+14>: lea 0xc(%esp),%ebx
=> 0x8048066 <_start+18>: mov $0x1,%eax
0x804806b <_start+23>: mov $0x0,%ebx
0x8048070 <_start+28>: int $0x80
0x8048072: add %al,(%eax)
0x8048074: add %al,(%eax)
stack

Lea

- LEA loads the pointer you are addressing and mov loads the actual value at that address
- LEA is used for address computation
- Provides two things that add does not provide:
  - Lea does not try to dereference an address => performs address calculations using two or three operands.
    - E.g. lea (%eax+%ebx+0x1234), %eax
  - The ability to store result in any register and not one of the operands in the instruction (unlike add)
  - Does not alter conditional flags (EFLAGS such as ZF)

Arithmetic and Logic Instructions

- Add
- Sub
- Inc,dec
- Imul
- Idiv
- And , or , xor
- Not
- Shl, shr

Add

- Adds two operands and places result in second operand
- Both operands maybe registers or one maybe memory

Syntax

```
add <reg>, <reg>
add <mem>, <reg>
add <reg>, <mem>
add <con>, <reg>
add <con>, <mem>
```

- add $10, %eax ; eax set to eax + 10
- addb $10, %eax ; add 10 to single byte stored at %eax

Sub

- Stores in the value of its second operand, the result of subtracting the value of its first operand from the value of second operand
- Both operands maybe registers or one maybe memory

Syntax

```
sub <reg>, <reg>
sub <mem>, <reg>
sub <reg>, <mem>
sub <con>, <reg>
sub <con>, <mem>
```

- sub %ah, %al ; al = al – ah
- subb $216, %eax ; eax = eax - 216

Demo asm3

- asm3

Inc, Dec

- Inc: Increments contents of its operand by one
- Dec: Decrements contents of its operand by one

Syntax

```
inc <reg>
inc <mem>
dec <reg>
dec <mem>
```

Demo asm4

- Run asm4

```
_start:
        movl $3, 0xC(%esp) #does this change esp value?
        addl $0x7, 0xC(%esp)
        subl $0x5, 0xC(%esp)
```

What is the value at 0xC(%esp) at this point?

```
        movl $3, %eax
        inc %eax #increment
        decb 0xc(%esp) #decrement
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

What is the value at 0xC(%esp) at this point?

Imul – Integer Multiplication

- 2 formats – 2 operand format and three operand format

Syntax

```
imul <reg32>, <reg32>
imul <mem>, <reg32>
imul <con>, <reg32>, <reg32>
imul <con>, <mem>, <reg32>
```

- imul (%ebx), %eax ; multiply contents of eax by the 32 bit contents of the memory at location ebx. Store result in eax
- imul $25, %edi, %esi ; esi is set to edi * 25

and, or, xor

Perform the specified logical operation on their operands

Syntax

```
and <reg>, <reg>
and <mem>, <reg>
and <reg>, <mem>
and <con>, <reg>
and <con>, <mem>

or <reg>, <reg>
or <mem>, <reg>
or <reg>, <mem>
or <con>, <reg>
or <con>, <mem>

xor <reg>, <reg>
xor <mem>, <reg>
xor <reg>, <mem>
xor <con>, <reg>
xor <con>, <mem>
```

Not

Logically negates the operand contents

Syntax

```
not <reg>
not <mem>
```

Negate

- Perform two’s complement negation of operand contents

Syntax

```
neg <reg>
neg <mem>
```

- neg %eax (EAX is set to –EAX)

Control Flow Instructions

- jmp
- jcondition
- cmp
- call
- ret

Flags Affected by Arithmetic

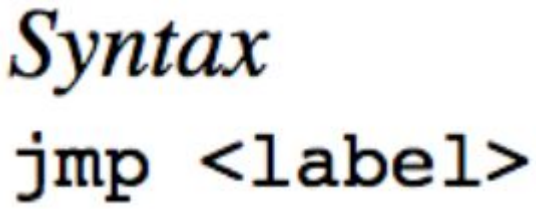
- EFLAGS has flags that reflect the outcome of arithmetic (and bitwise) operations
  - Based on the contents of the destination operand
- Essential Flags
  - Zero Flag – destination equals 0
  - Sign Flag – destination is negative
  - Carry Flag – unsigned value out of range
  - Overflow Flag – signed value out of rangeCF, OF out of scope of this class – feel free to dig in
- MOV and LEA instruction does not affect flags

asm7.s

```
.text

        .global _start

        _start:

                mov $1, %bx

                sub $1, %bx             #ZF=1

                mov $2, %al              #SF = 1

                sub $3, %al

                mov $0x7f, %al          #OF = 1

                add $1, %al

                #Exit

                mov $1, %eax

                mov $0, %ebx

                int $0x80
```

jmp

- Transfers control flow to the instruction at the memory location



- Unconditional jmp
  - E.g.

```
jmp begin
```

Demo asm5

Jmp demo

```
_start:

        movl $3, 0xC(%esp) #does this change esp value?

        addl $0x7, 0xC(%esp)

        jmp _decr

        movl $3, %eax

        inc %eax  #increment

        _decr: decb 0xc(%esp)        What is the value of %eax at this point?

        movl $1, %eax

        movl $0, %ebx

        int $0x80
```

Demo asm5

- gdb asm5
- Disass _start
  - What do you see?
- How to break at _decr ?

cmp

- Compare values of two specified operands, setting the condition codes in the machine status word
- Does a sub but does not store the result

Syntax

```
cmp <reg>, <reg>
cmp <mem>, <reg>
cmp <reg>, <mem>
cmp <con>, <reg>
```

- E.g.
  - Cmpb $10, (%ebx); if byte stored at location in ebx is equal to 10, then jump to loop
  - Jeq loop

jcondition

- Conditional jumps based on the status of a set of condition codes that are stored in a special register called machine status word
- Machine status word include information about last arithmetic operation performed
    - One bit indicates if last result was zero, another indicates if last result was one.
    - Conditional jumps performed based on the condition codes

jcondition

- E.g.

cmp %ebx, %eax ; if contents of eax is less than or equal to contents of ebx jump to label done

jle done

Syntax

`je <label>` (jump when equal)
`jne <label>` (jump when not equal)
`jz <label>` (jump when last result was zero)
`jg <label>` (jump when greater than)
`jge <label>` (jump when greater than or equal to)
`jl <label>` (jump when less than)
`jle <label>` (jump when less than or equal to)

Demo asm6

- Break at first instruction
- Use next to step through code.
- What happens when cmp instruction is executed?
  - Notice EFLAGS

Linux System Calls

- What are system calls?
  - Library calls that provide userland processes a way to get service from the kernel
  - Interface between the protected kernel mode and user mode
- What happens in a system call?
  - A kernel code snippet is run on request of a user process.
  - The code runs in ring 0 – the highest privilege in x86 arch
  - User processes run in ring 3.
- To implement system call we need a way to
  - Call ring 0 code from ring 3
- Two ways to execute syscalls
  - Use C library wrapper, libc (this works indirectly)
    - System calls available in /usr/include/asm/unistd.h
    - E.g. exit(), read(), write()
- Execute syscalls directly with assembly by loading the arguments into registers and then calling a software interrupt

Old School System Call

- System calls are invoked using a software interrupt – INT 0x80
- System call number copied to %eax
- Execute INT 0x80
  - CPU switches to kernel mode
  - Invokes an interrupt service routine that delegates call to the right system handler based on value in %eax
  - The routine is executed in ring 0
  - It is defined in the file /usr/src/linux/arch/i386/kernel/entry.s

Passing Arguments (Old School)

More arguments are passed as a data structure to the first argument

Register	**Purpose**
EAX	System Call Number
EBX	First Argument
ECX	Second Argument
EDX	Third Argument
ESI	Fourth Argument
EDI	Fifth Argument

Example: exit()

- Exit(0) exits a program
- Function definition
  - void _exit(int status);
- System call number is a unique integer that is assigned to each system call
- Syscall no. for exit is 1

Exit Assembly Program

```
.text

.global _start

_start:

movl $1, %eax          <eax stores sys call number>

movl $0, %ebx          <first argument>

int $0x80              <calls syscall>
```

Hello World (Old School)

- Use write syscall to print “Hello World”
- Use exit syscall to exit

Write Syscall

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

Argument	Description
fd	It is the file descriptor which has been obtained from the call to open. It is an integer value. The values 0, 1, 2 can also be given, for standard input, standard output & standard error, respectively .
buf	It points to a character array, which can be used to store content obtained from the file pointed to by fd.
nbytes	It specifies the number of bytes to be written from the character array into the file pointed to by fd.

Write syscall no. is 4

fd=1 for standard output

Hello World

- Storing initialized data
  - In .data segment

.data

hwstr:

.asciz “Hello World\n”

What goes in text for write

EAX?

EBX?

ECX?

EDX?

Hello World

```
.data
        hwstr:
                .ascii “Hello World\n”
.text
.global _start
_start:
#write     <comment>
movl $4, %eax      <sys call#>
movl $1, %ebx      <stdout>
```

```
movl $hwstr, %ecx       <buffer>
movl $12, %edx    <length of string>
#exit
movl $1, %eax      <eax stores sys call number>
movl $0, %ebx      <first argument>
int $0x80        <calls syscall>
```

Tryout More Commands

- gdb helloworld
- Type following commands
- b *_start+1 (to break after first instruction from start)
- Info variables (to list all variables)
- x/s <addr> (to examine value of variables)
- x/12cb <addr> (of hello world)
- x/1db <addr> (one decimal, byte for In32)
- x/1dh <addr> (one decimal, halfword for Int 16)
- x/1xh <addr> (one hex, halfword for Int 16)
- x/1fw <addr> (address of float)
- x/4d <addr_of_array>

Calling Conventions

- How code calls a subroutine is compiler-dependent and configurable. But there are a few conventions.
- We will only deal with the “cdecl” and “stdcall” conventions.
- More info at
  - http://en.wikipedia.org/wiki/X86_calling_conventions
  - http://www.programmersheaven.com/2/Calling-conventions

Calling Conventions - cdecl

- “C declaration” - most common calling convention
- Function parameters pushed onto stack right to left
- Saves the old stack frame pointer and sets up a new stack frame
- eax or edx:eax returns the result for primitive data types
- Caller is responsible for cleaning up the stack

Calling Conventions - stdcall

- This convention used by Microsoft C++ code - e.g. Win32 API
- Function parameters pushed onto stack right to left
- Saves the old stack frame pointer and sets up a new stack frame
- eax or edx:eax returns the result for primitive data types
- Callee responsible for cleaning up any stack parameters it takes
- Aside: typical MS, “If I call my new way of doing stuff 'standard' it must be true!”

call, ret

- Subroutine call and return
- Call saves the return location on stack (unlike jmp)
- Overcome limitations of old Intel processors (INT 0x80 is more than 500 lines of code)
- Return pops the code location and jumps to the location

Syntax

```
call <label>
ret
```

CALL - Call Procedure

- 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 for when the procedure is done
- Then it changes eip to the address given in the instruction
- Destination address can be specified in multiple ways
  - Absolute address
  - Relative address (relative to the end of the instruction)

RET - Return from Procedure

- Pop the top of the stack into eip (remember pop increments stack pointer)
- In this form, the instruction is just written as “ret”
- Typically used by cdecl functions

Example1.c – What does this program do?

Function x:

```
0x804845c push   ebp

0x804845d mov    esp,ebp

0x804845f mov    0xbeef,%eax

0x8048464 pop    %ebp

0x8048465 ret
```

main:

```
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <x>
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```

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
int add(){
  return 0xbeef;
}
int main(){
  add();
  return 0xdead;
}
```

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp
0x8048465 ret
main:
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```

Example1.c 1:

EIP = 0x8048466, but no instruction yet executed

eax	0x01⌘
ebp	0x0⌘
esp	0xBFFFF6BC⌘

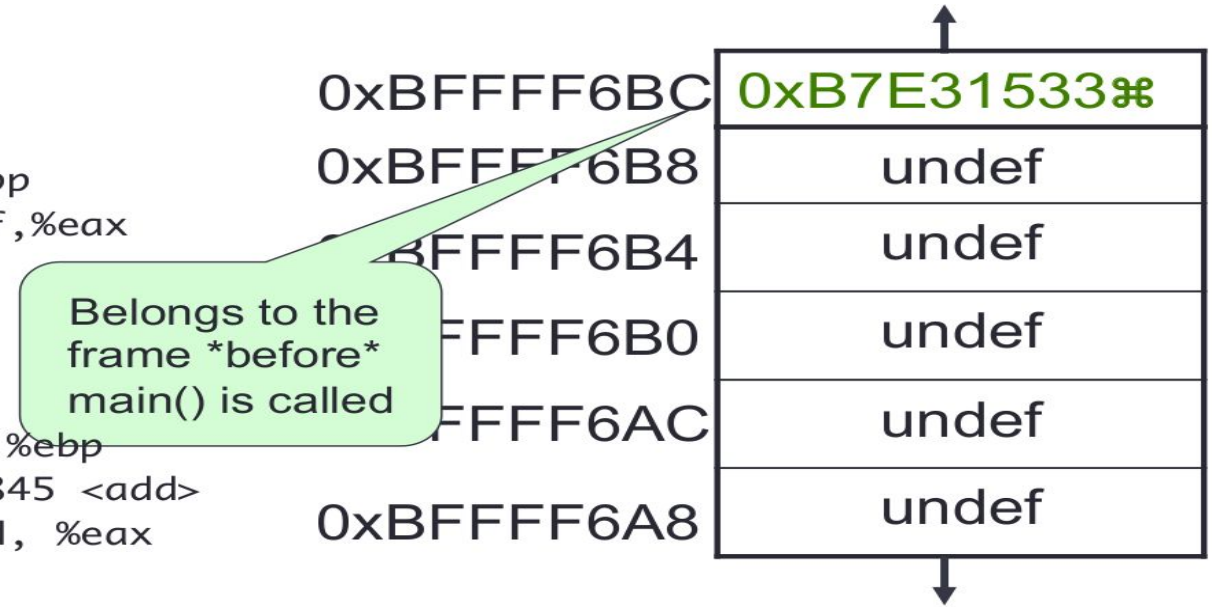
Key:

☒ executed instruction

♍ modified value

⌘ start value

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp
0x8048465 ret
main:
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```

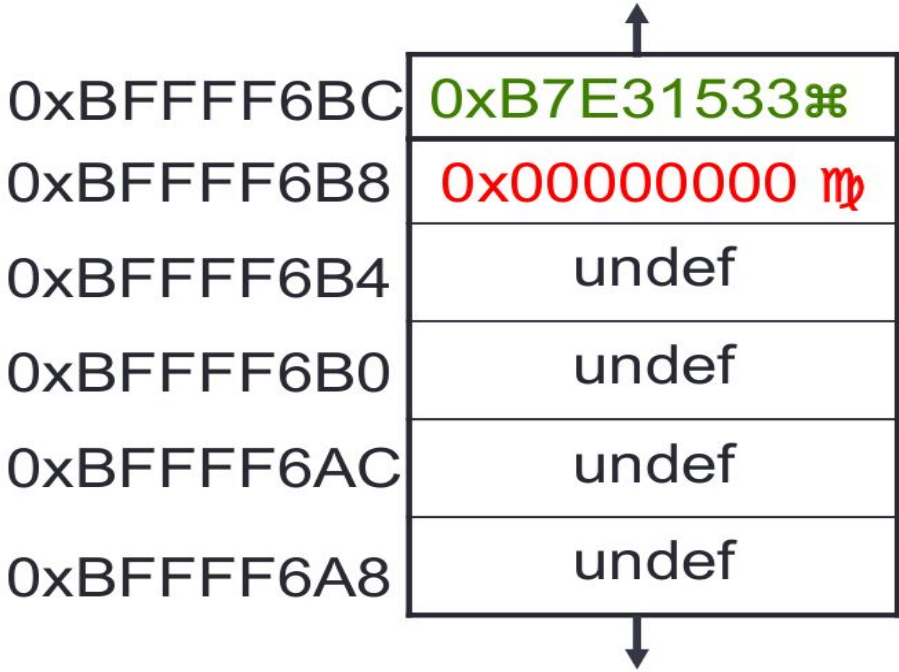


Example1.c 2

eax	01⌘
ebp	0⌘
esp	0xBFFFF6B8♍

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

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp
0x8048465 ret
main:
0x8048466 push    ebp   ☒
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```

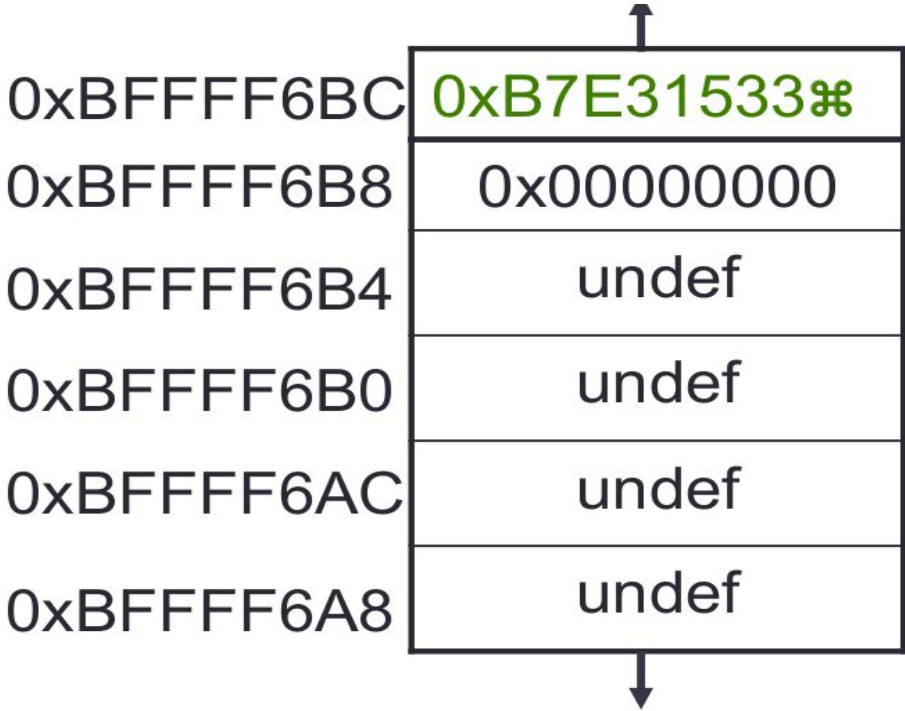


Example1.c 3

eax	0x1⌘
ebp	0xBFFFF6B8♍
esp	0xBFFFF6B8

Key:
⌧ executed instruction
♍ modified value
⌘ start value

```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0xbeef,%eax
0x8048464 pop         %ebp
0x8048465 ret
main:
0x8048466 push        %ebp
0x8048467 mov         esp,ebp ⌧
0x8048469 call        0x804845 <add>
0x804846e mov         0xdead, %eax
0x8048466 pop         %ebp
0x8048466 ret
```



Example1.c 4

eax	01⌘
ebp	0xBFFFF6B8
esp	0xBFFFF6B4♍

Key:
⊠ executed instruction
♍ modified value
⌘ start value

```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0xbeef,%eax
0x8048464 pop         %ebp
0x8048465 ret
main:
0x8048466 push        %ebp
0x8048467 mov         %esp, %ebp
0x8048469 call        0x804845 <add> ⊠
0x804846e mov         0xdead, %eax
0x8048466 pop         %ebp
0x8048466 ret
```

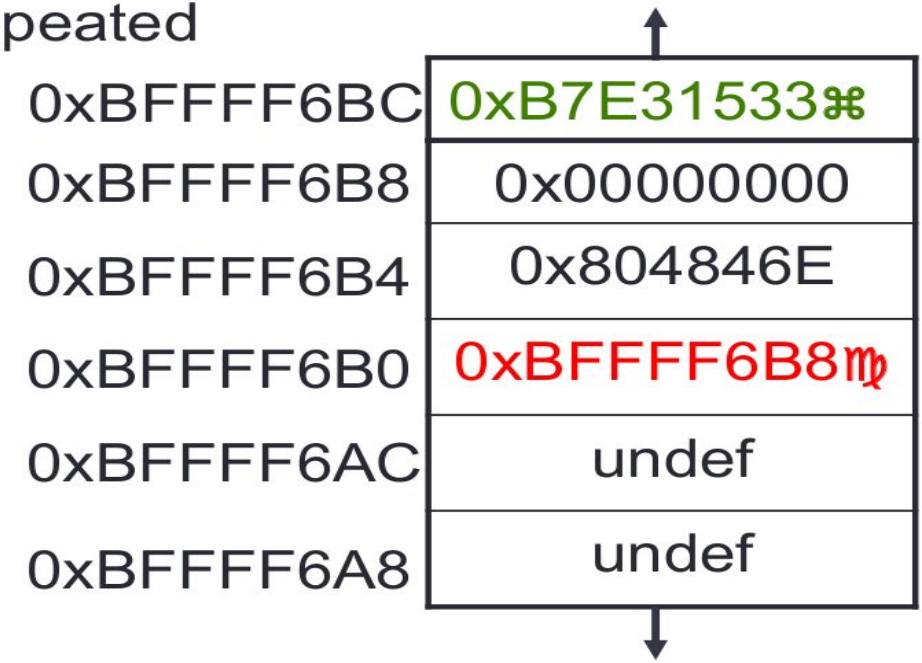
Example1.c 5

eax	01⌘
ebp	0xBFFFF6B8
esp	0xBFFFF6B0♍

Key:
⌧ executed instruction
♍ modified value
⌘ start value

Beginning is repeated

```
add:
0x804845c push   ebp ⌧
0x804845d mov    esp,ebp
0x804845f mov    0xbeef,%eax
0x8048464 pop    %ebp
0x8048465 ret
main:
0x8048466 push   %ebp
0x8048467 mov    %esp, %ebp
0x8048469 call   0x804845 <add>
0x804846e mov    0xdead, %eax
0x8048466 pop    %ebp
0x8048466 ret
```



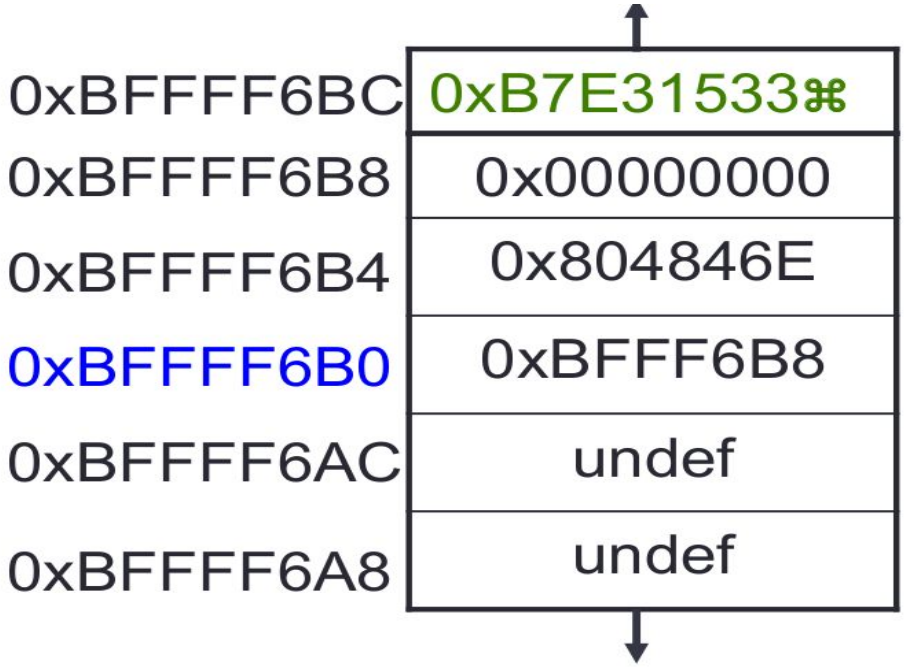
ebp belongs to which function?

Example1.c 6

eax	01⌘
ebp	0xBFFFF6B0♍
esp	0xBFFFF6B0

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

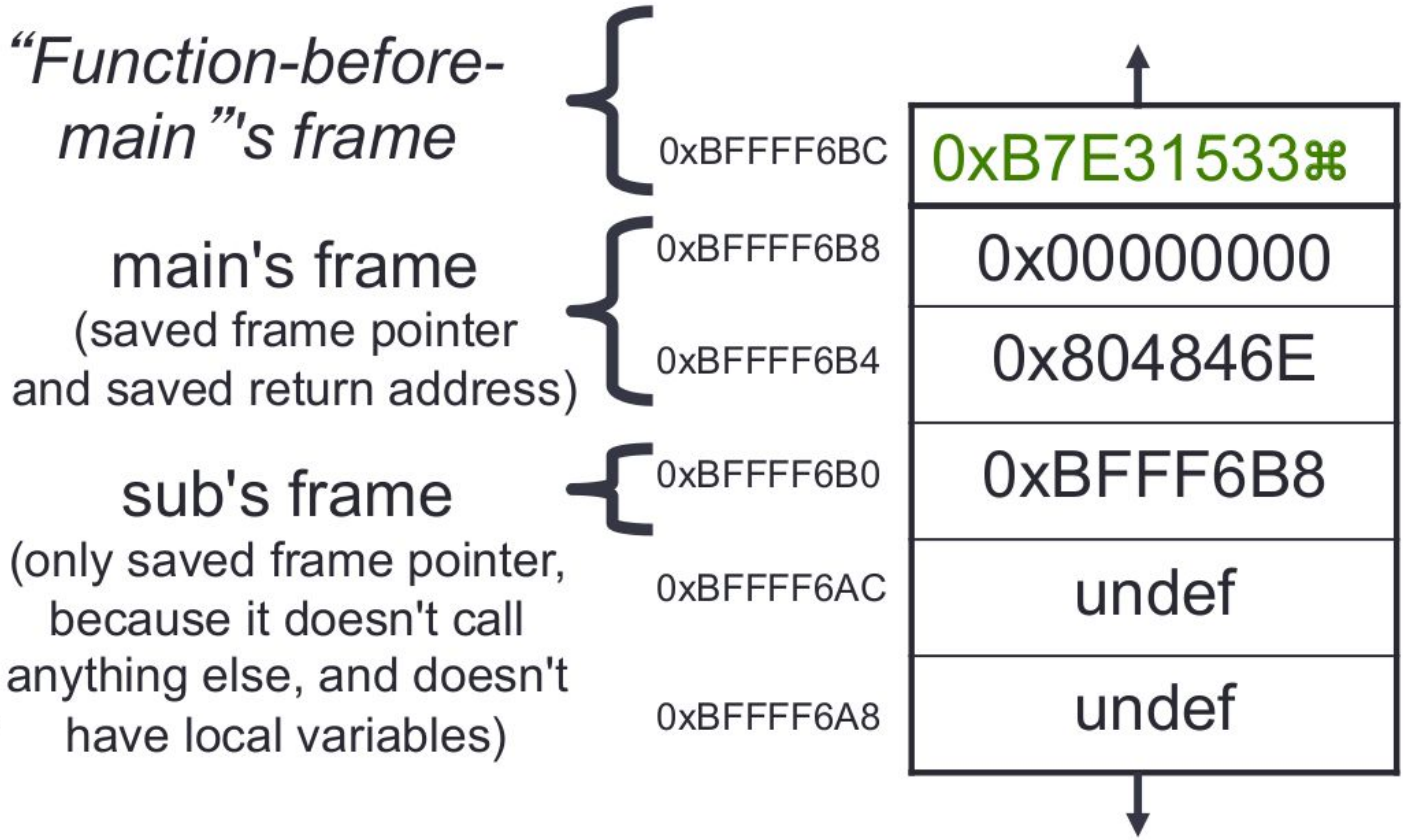
```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp ☒
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp
0x8048465 ret
main:
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```



What is this instruction doing?

Example1.c 6 STACK FRAME TIME OUT

```
add:
push ebp
mov  esp,ebp
mov  0xbeef,%eax
pop  %ebp
ret
main:
push %ebp
Mov  %esp, %ebp
call 0x804845 <add>
mov  0xDEAD, %eax
pop  %ebp
ret
```



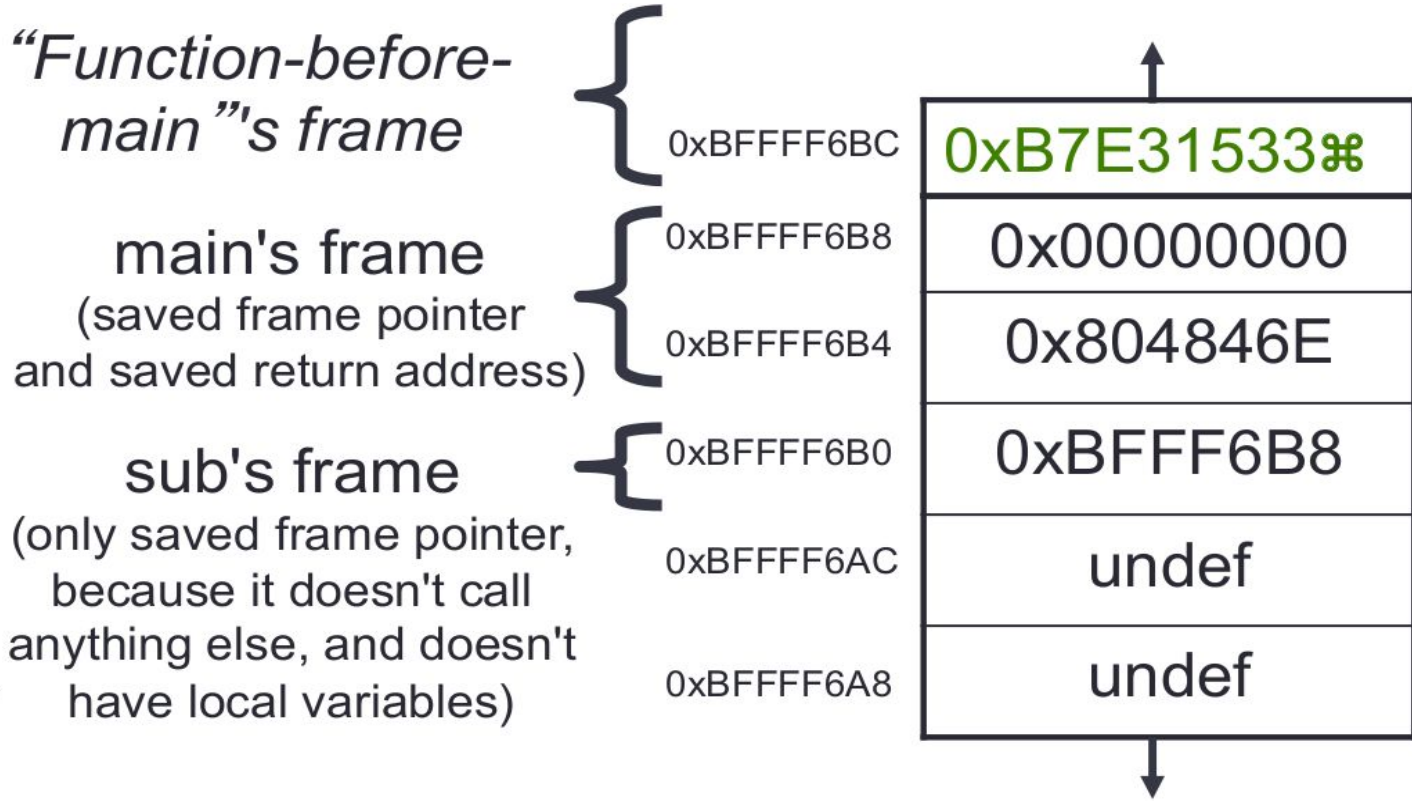
Stack Frames and Function Invocation

- The stack is composed of frames
- Frames are pushed onto stack by a function prologue
- The address of the current frame is stored in the Frame Pointer (FP) register
  - On intel architectures %ebp is used for this purpose
- Each frame contains
  - The function parameters
  - The return address to jump to at the end of the function
  - The pointer to the previous frame (old ebp)
  - Functions local variables

Example1.c 6 STACK FRAME TIME OUT

Representation1: The frame contains everything a function pushes onto the stack

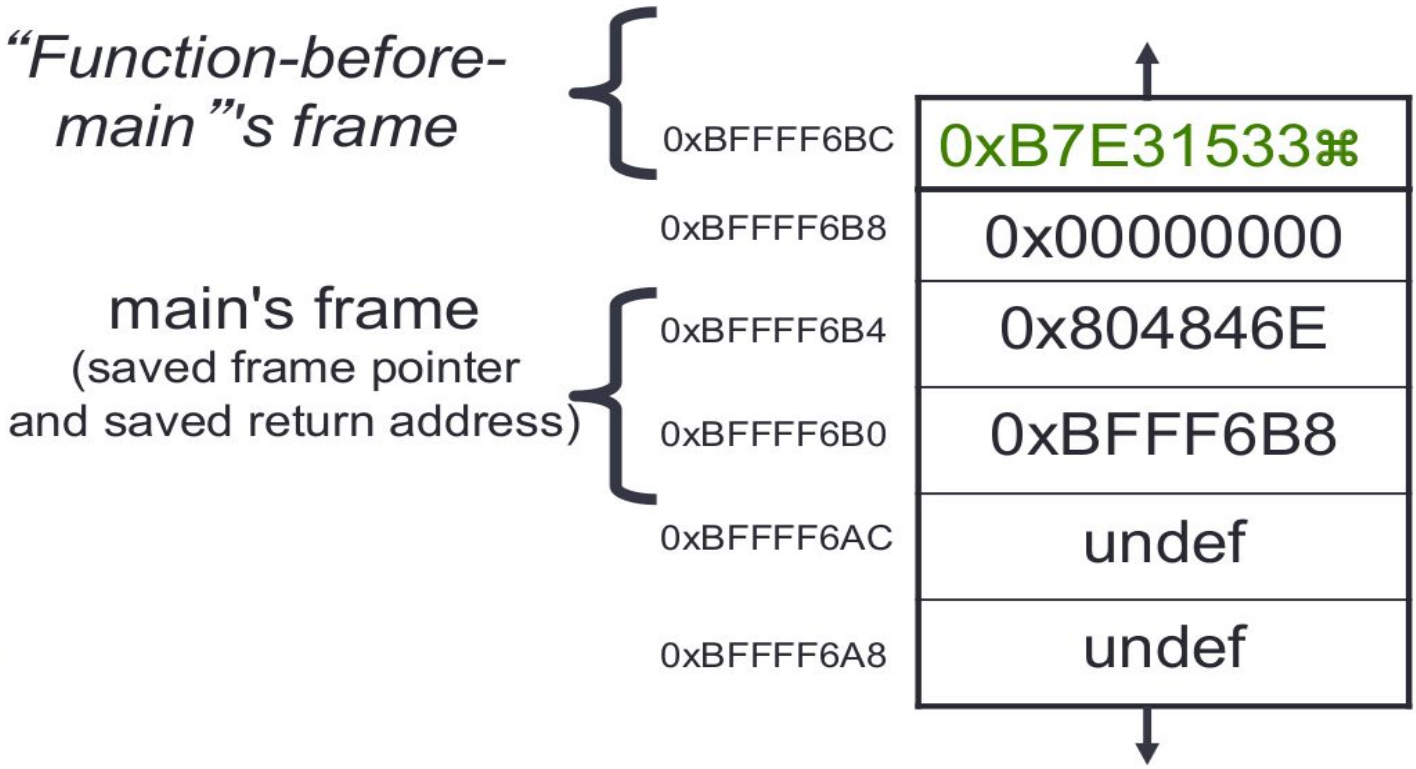
```
add:
push ebp
mov  esp,ebp
mov  0xbeef,%eax
pop  %ebp
ret
main:
push %ebp
Mov  %esp, %ebp
call 0x804845 <add>
mov  0xDEAD, %eax
pop  %ebp
ret
```



Example1.c 6 STACK FRAME TIME OUT

Representation2: The frame contains everything that belongs to a function

```
add:
push ebp
mov  esp,ebp ⌧
mov  0xbeef,%eax
pop  %ebp
ret
main:
push %ebp
Mov  %esp, %ebp
call 0x804845 <add>
mov  0xDEAD, %eax
pop  %ebp
ret
```



Example1.c 6 STACK FRAME TIME OUT

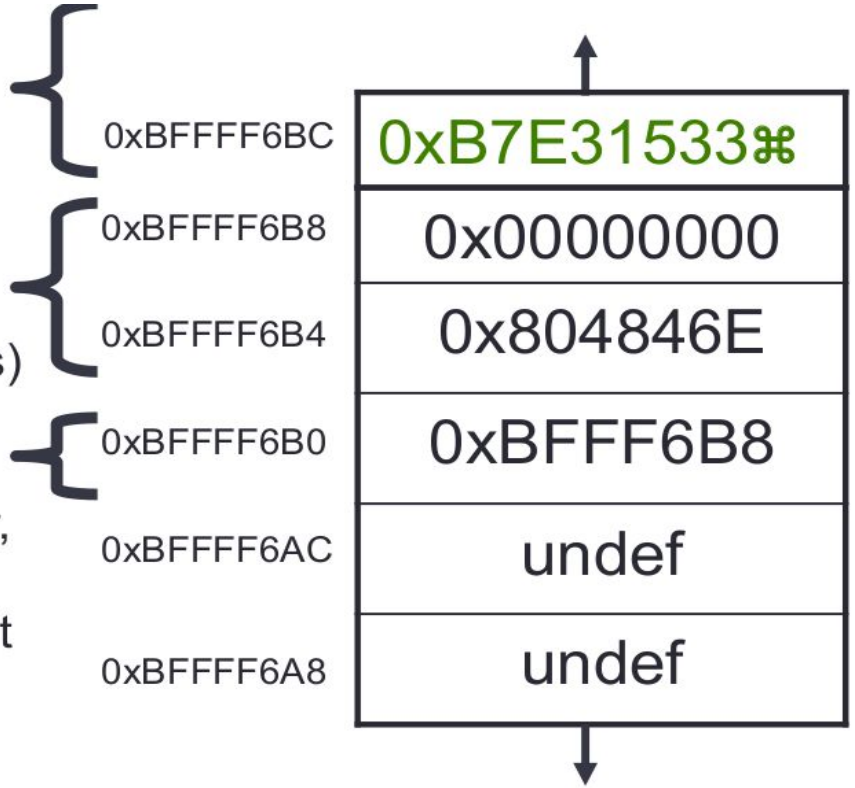
Chosen Convention: The frame contains everything a function pushes onto the stack

```
add:
push ebp
mov  esp,ebp
mov  0xbeef,%eax
pop  %ebp
ret
main:
push %ebp
Mov  %esp, %ebp
call 0x804845 <add>
mov  0xDEAD, %eax
pop  %ebp
ret
```

"Function-before-main"'s frame

main's frame
(saved frame pointer and saved return address)

sub's frame
(only saved frame pointer, because it doesn't call anything else, and doesn't have local variables)

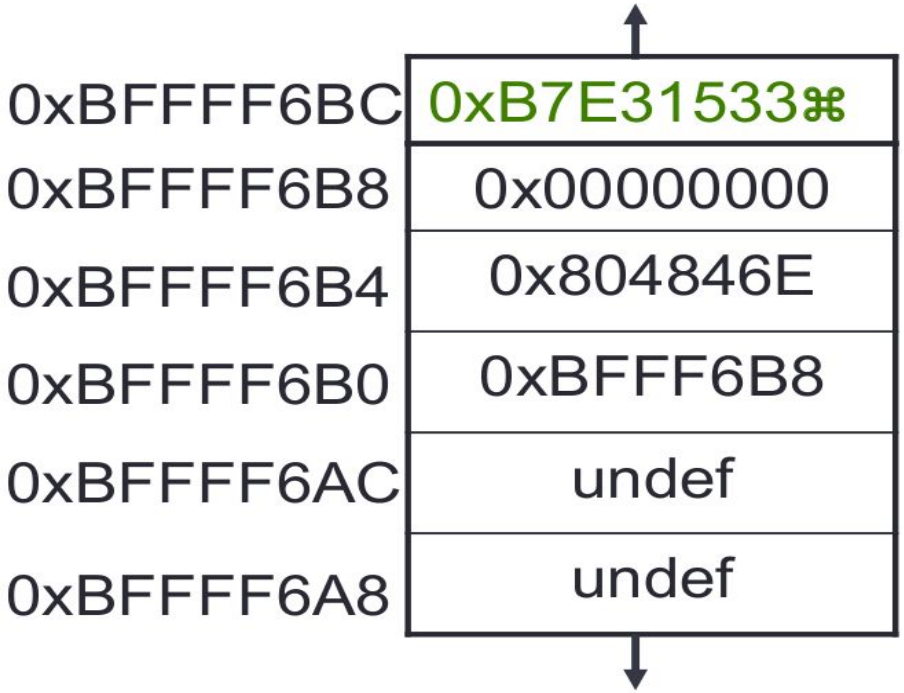


Example1.c 7

eax	0x0000BEEF
ebp	0xBFFFF6B0
esp	0xBFFFF6B0

Key:
⌧ executed instruction
♍ modified value
⌘ start value

```
add:
0x804845c push      %ebp
0x804845d mov       %esp, %ebp
0x804845f mov       0xBEEF, %eax ⌧
0x8048464 pop       %ebp
0x8048465 ret
main:
0x8048466 push      %ebp
0x8048467 mov       %esp, %ebp
0x8048469 call      0x804845 <add>
0x804846e mov       0xdead, %eax
0x8048466 pop       %ebp
0x8048466 ret
```



Example1.c 8

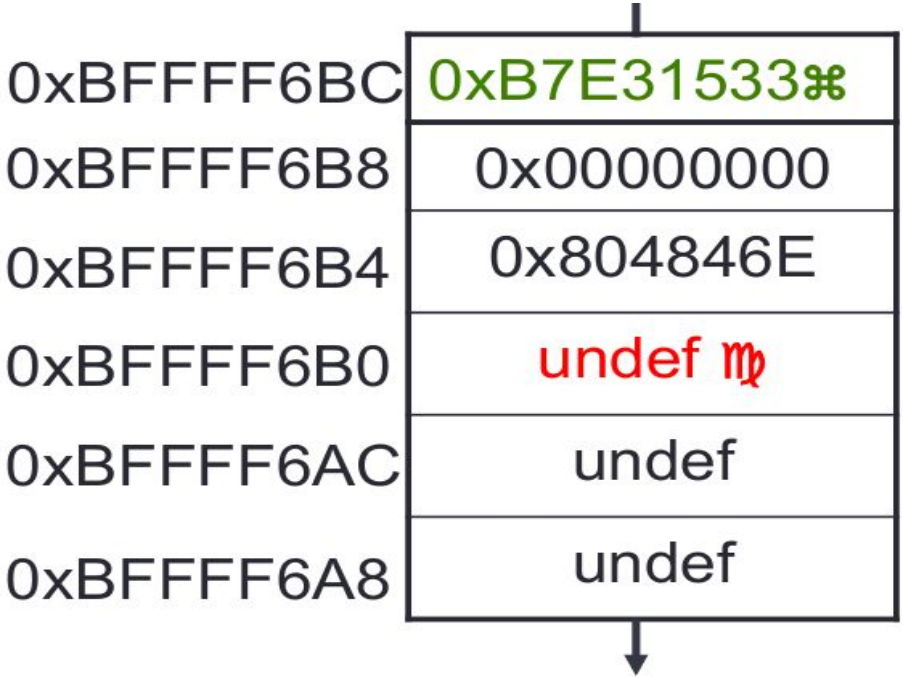
eax	0xBEEF
ebp	0xBFFFF6B8♍
esp	0xBFFFF6B4♍

Key:

☒ executed instruction

♍ modified value

```
add:
0x804845c push    %ebp
0x804845d mov     %esp, %ebp
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp ☒
0x8048465 ret
main:
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```



Example1.c 9

eax	0xBEEF
ebp	0xBFFFF6B8
esp	0xBFFFF6B8℔

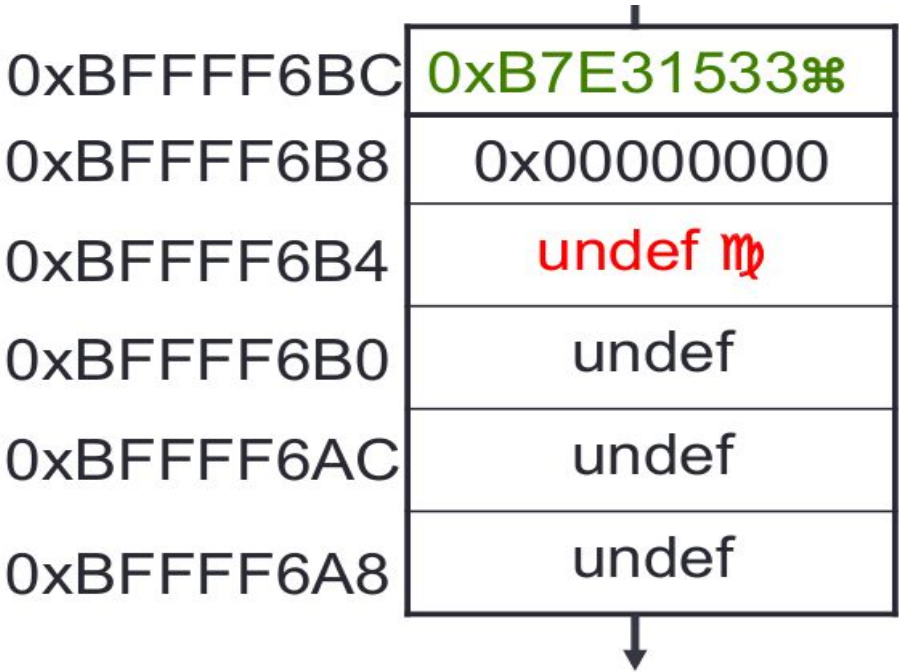
Key:

⌧ executed instruction

℔ modified value

⌘ start value

```
add:
0x804845c push    %ebp
0x804845d mov     %esp, %ebp
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp
0x8048465 ret ⌧
main:
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```

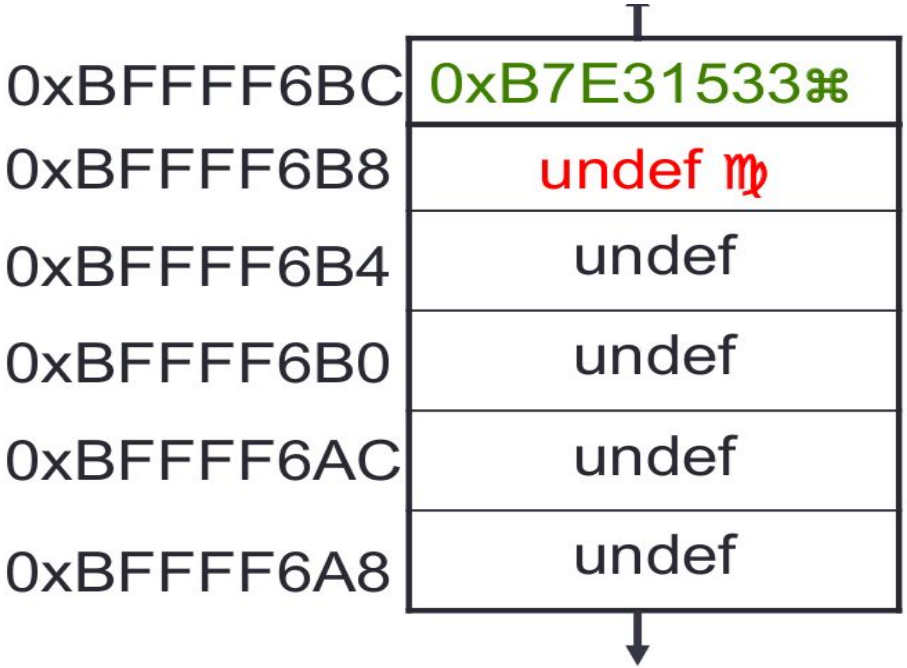


Example1.c 10

eax	0xDEAD
ebp	0x00000000♍
esp	0xBFFFF6BC♍

Key:
⌧ executed instruction
♍ modified value
⌘ start value

```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0xbeef,%eax
0x8048464 pop         %ebp
0x8048465 ret
main:
0x8048466 push        %ebp
0x8048467 mov         %esp, %ebp
0x8048469 call        0x804845 <add>
00401018  mov         0xDEAD, %eax
0040101D  pop         ebp ⌧
0040101E  ret
```

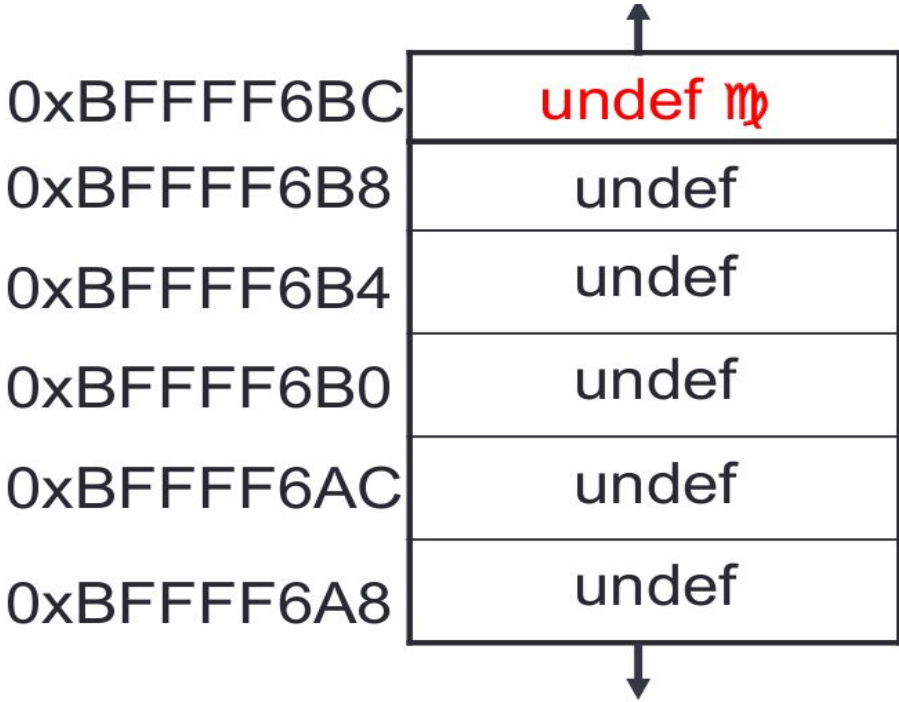


Example1.c 11

eax	0xDEAD
ebp	0x00000000
esp	0xBFFFF6C0♍

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:
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: **0xB7E31533**

Example2.c

The stack frame now also contains arguments to functions

Learn how arguments are accessed

```
//Example2 – understanding
  function calls
int add(int x, int y){
   return x + y;
}
int main(){
  add(6,8);
  return 0xf00d;
}
```

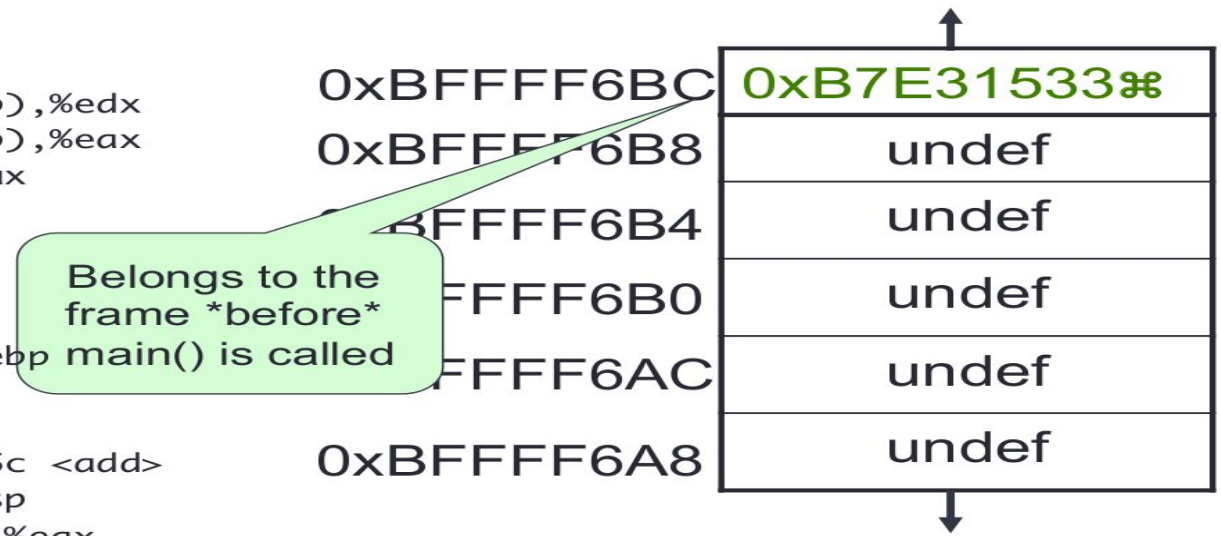
```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0x8(%ebp),%edx
0x8048462 mov         0xC(%ebp),%eax
0x8048465 add         %edx,%eax
0x8048467 pop         %ebp
0x8048465 ret
main:
0x8048469 push        %ebp
0x804846a mov         %esp, %ebp
0x804846c push        $0x8
0x804846e push        $0x6
0x8048470 call        0x804845c <add>
0x8048475 add         0x8, %esp
0x804846e mov         0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Example2.c 1

eax	0x01⌘
ebp	0x0⌘
esp	0xBFFFF6BC⌘

Key:
☒ executed instruction
♍ modified value
⌘ start value

```
add:
0x804845c push          ebp
0x804845d mov           esp,ebp
0x804845f mov           0x8(%ebp),%edx
0x8048462 mov           0xC(%ebp),%eax
0x8048465 add           %edx,%eax
0x8048467 pop           %ebp
0x8048465 ret
main:
0x8048469 push          %ebp
0x804846a mov           %esp, %ebp
0x804846c push          $0x8
0x804846e push          $0x6
0x8048470 call          0x804845c <add>
0x8048475 add           0x8, %esp
0x804846e mov           0xf00d, %eax
0x8048466 leave
0x8048466 ret
```



Example2.c 2

eax	01⌘
ebp	0⌘
esp	0xBFFFF6B8♍

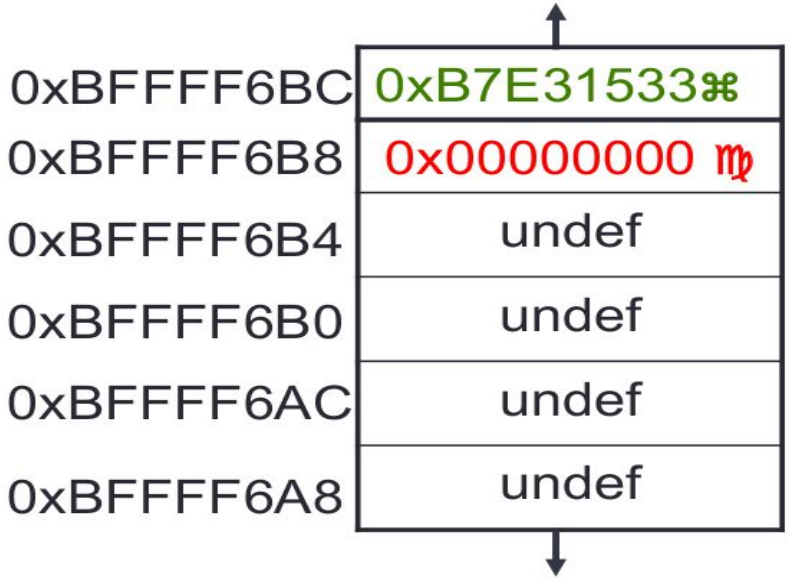
```
add:
0x804845c push          ebp
0x804845d mov           esp,ebp
0x804845f mov           0x8(%ebp),%edx
0x8048462 mov           0xC(%ebp),%eax
0x8048465 add           %edx,%eax
0x8048467 pop           %ebp
0x8048465 ret
main:
0x8048469 push          ebp  ☒
0x804846a mov           %esp, %ebp
0x804846c push          $0x8
0x804846e push          $0x6
0x8048470 call          0x804845c <add>
0x8048475 add           0x8, %esp
0x804846e mov           0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:

☒ executed instruction,

♍ modified value

⌘ start value



Example2.c 3

eax	0x1⌘
ebp	0xBFFFF6B8♍
esp	0xBFFFF6B8

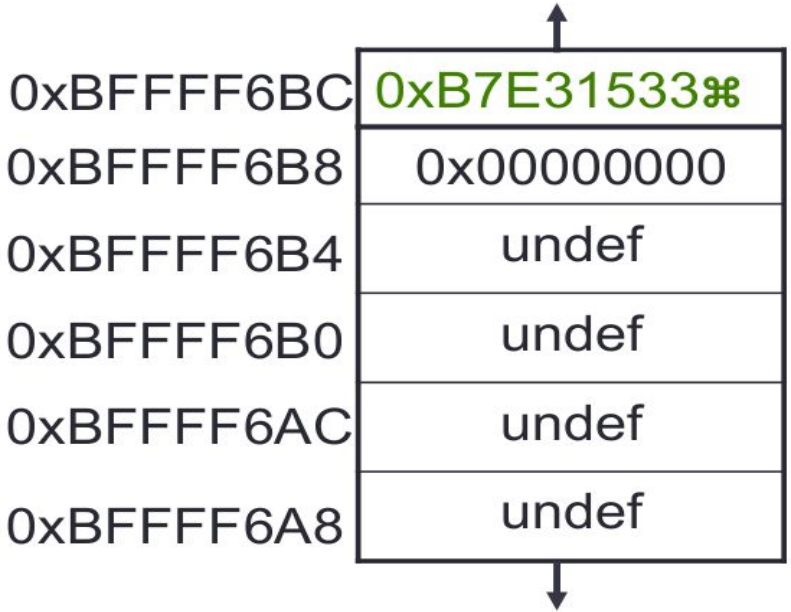
```
add:
0x804845c push          ebp
0x804845d mov           esp,ebp
0x804845f mov           0x8(%ebp),%edx
0x8048462 mov           0xC(%ebp),%eax
0x8048465 add           %edx,%eax
0x8048467 pop           %ebp
0x8048465 ret
main:
0x8048469 push          %ebp
0x804846a mov           esp,ebp ⌧
0x804846c push          $0x8
0x804846e push          $0x6
0x8048470 call          0x804845c <add>
0x8048475 add           0x8, %esp
0x804846e mov           0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:

⌧ executed instruction,

♍ modified value

⌘ start value



Example2.c 4

eax	01⌘
ebp	0xBFFFF6B8
esp	0xBFFFF6B4♍

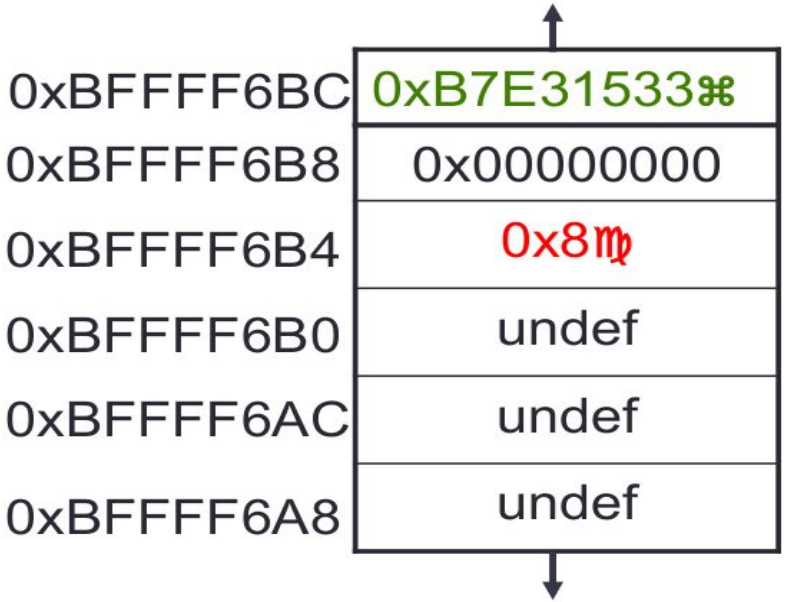
```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0x8(%ebp),%edx
0x8048462 mov         0xC(%ebp),%eax
0x8048465 add         %edx,%eax
0x8048467 pop         %ebp
0x8048465 ret
main:
0x8048469 push        %ebp
0x804846a mov         %esp, %ebp
0x804846c push        $0x8 ⌧
0x804846e push        $0x6
0x8048470 call        0x804845c <add>
0x8048475 add         0x8, %esp
0x804846e mov         0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:

⌧ executed instruction,

♍ modified value

⌘ start value



Example2.c 5

eax	01⌘
ebp	0xBFFFF6B8
esp	0xBFFFF6B0♍

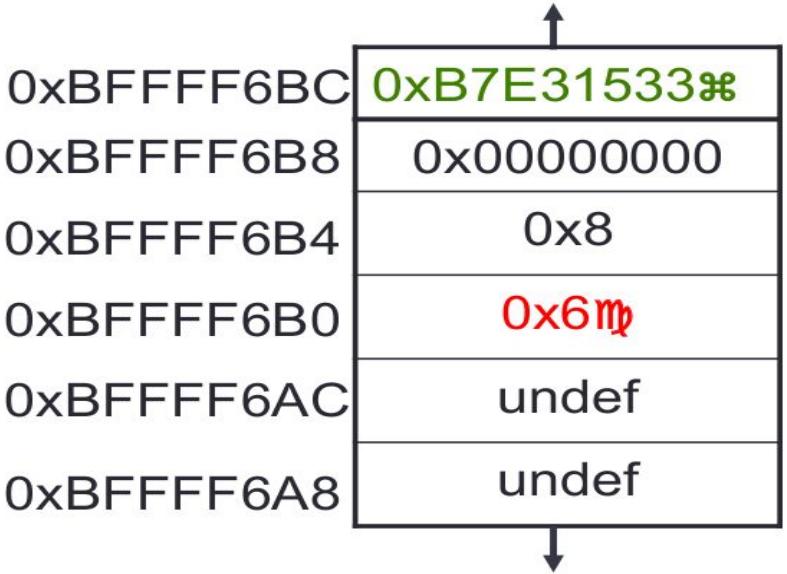
```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0x8(%ebp),%edx
0x8048462 mov         0xC(%ebp),%eax
0x8048465 add         %edx,%eax
0x8048467 pop         %ebp
0x8048465 ret
main:
0x8048469 push        %ebp
0x804846a mov         %esp, %ebp
0x804846c push        $0x8
0x804846e push        $0x6   ⌧
0x8048470 call        0x804845c <add>
0x8048475 add         0x8, %esp
0x804846e mov         0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:

⌧ executed instruction,

♍ modified value

⌘ start value



Example2.c 6

eax	01⌘
ebp	0xBFFFF6B8
esp	0xBFFFF6AC♍

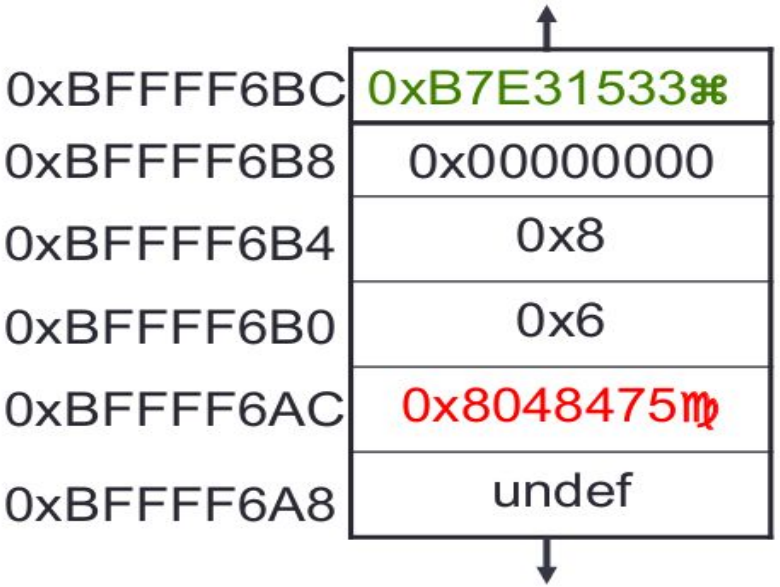
```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0x8(%ebp),%edx
0x8048462 mov         0xC(%ebp),%eax
0x8048465 add         %edx,%eax
0x8048467 pop         %ebp
0x8048465 ret
main:
0x8048469 push        %ebp
0x804846a mov         %esp, %ebp
0x804846c push        $0x8
0x804846e push        $0x6
0x8048470 call        0x804845c <add> ⌧
0x8048475 add         0x8, %esp
0x804846e mov         0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:

⌧ executed instruction,

♍ modified value

⌘ start value



Example2.c 7

eax	01⌘
ebp	0xBFFFF6B8
esp	0xBFFFF6A8♍

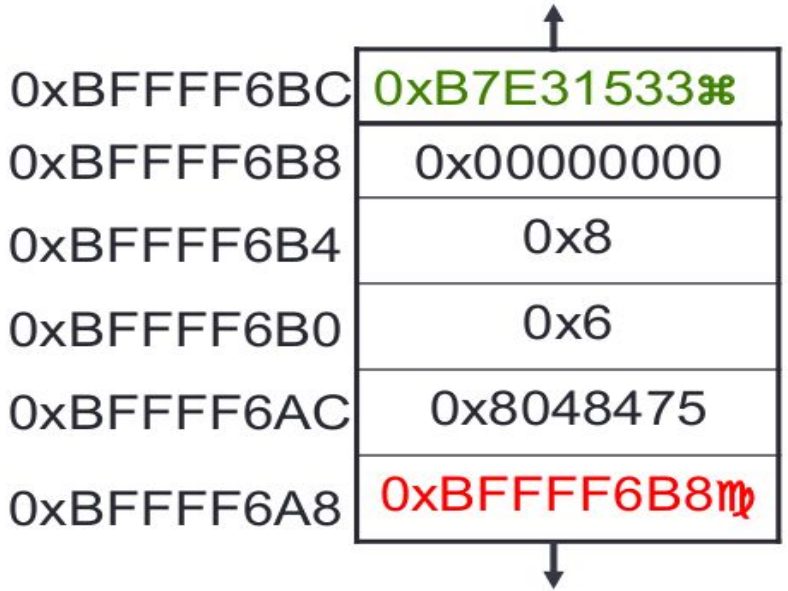
```
add:
0x804845c push        ebp  ⌧
0x804845d mov         esp,ebp
0x804845f mov         0x8(%ebp),%edx
0x8048462 mov         0xC(%ebp),%eax
0x8048465 add         %edx,%eax
0x8048467 pop         %ebp
0x8048465 ret
main:
0x8048469 push        %ebp
0x804846a mov         %esp, %ebp
0x804846c push        $0x8
0x804846e push        $0x6
0x8048470 call        0x804845c <add>
0x8048475 add         0x8, %esp
0x804846e mov         0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:

⌧ executed instruction,

♍ modified value

⌘ start value



Example2.c 8

eax	01⌘
ebp	0xBFFFF6A8♍
esp	0xBFFFF6A8

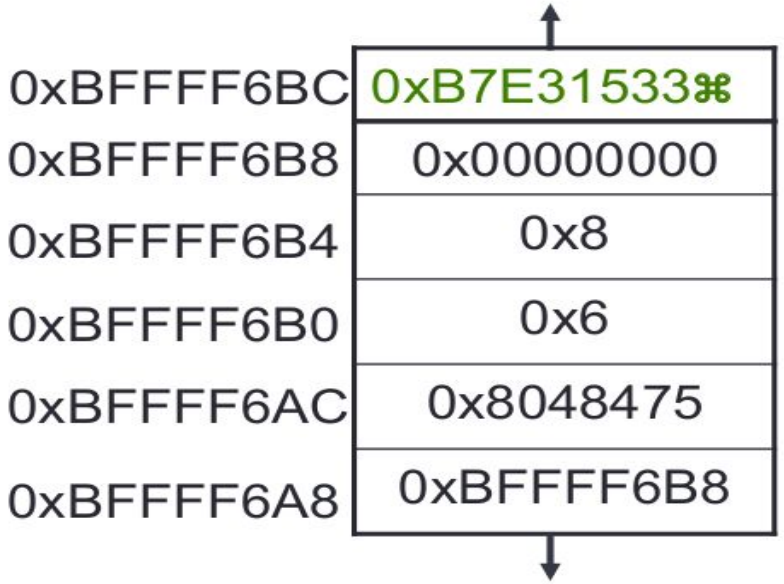
```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp ⌧
0x804845f mov         0x8(%ebp),%edx
0x8048462 mov         0xC(%ebp),%eax
0x8048465 add         %edx,%eax
0x8048467 pop         %ebp
0x8048465 ret
main:
0x8048469 push        %ebp
0x804846a mov         %esp, %ebp
0x804846c push        $0x8
0x804846e push        $0x6
0x8048470 call        0x804845c <add>
0x8048475 add         0x8, %esp
0x804846e mov         0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:

⌧ executed instruction,

♍ modified value

⌘ start value



Example2.c 9

eax	01⌘
edx	0x6♍
ebp	0xBFFFF6A8
esp	0xBFFFF6A8

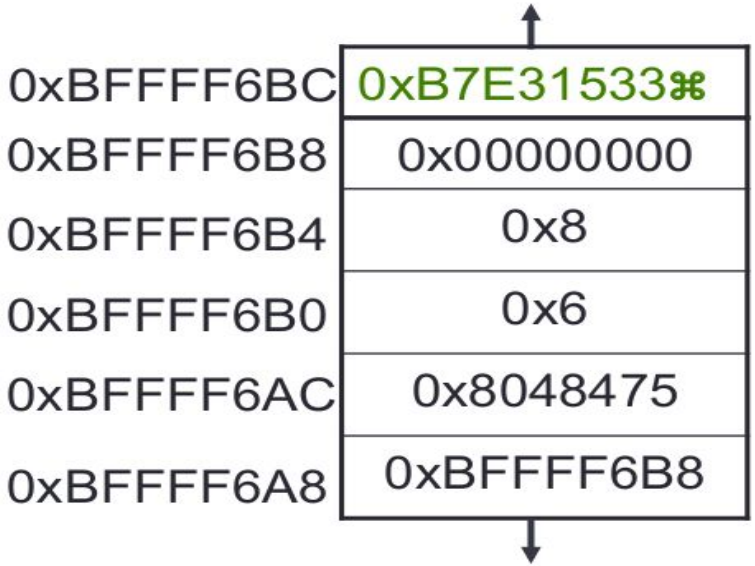
```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0x8(%ebp),%edx   ☒
0x8048462 mov         0xC(%ebp),%eax
0x8048465 add         %edx,%eax
0x8048467 pop         %ebp
0x8048465 ret
main:
0x8048469 push        %ebp
0x804846a mov         %esp, %ebp
0x804846c push        $0x8
0x804846e push        $0x6
0x8048470 call        0x804845c <add>
0x8048475 add         0x8, %esp
0x804846e mov         0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:

☒ executed instruction,

♍ modified value

⌘ start value



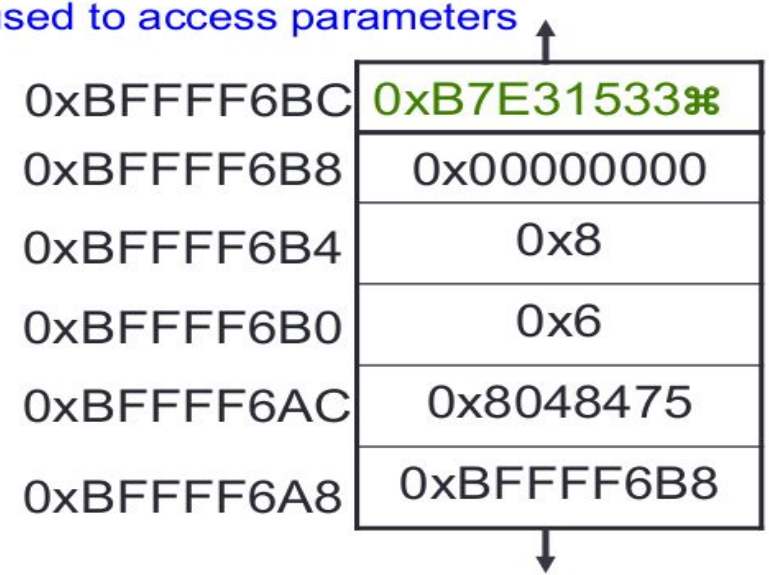
Example2.c 10

eax	0x8♍
edx	0x6
ebp	0xBFFFF6A8
esp	0xBFFFF6A8

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f mov     0x8(%ebp),%edx
0x8048462 mov     0xC(%ebp),%eax   ⊠
0x8048465 add     %edx,%eax
0x8048467 pop     %ebp
0x8048465 ret
main:
0x8048469 push    %ebp
0x804846a mov     %esp, %ebp
0x804846c push    $0x8
0x804846e push    $0x6
0x8048470 call    0x804845c <add>
0x8048475 add     0x8, %esp
0x804846e mov     0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:
⊠ executed instruction,
♍ modified value
⌘ start value

ebp used to access parameters



Example2.c 11

eax	0xE♍
edx	0x6
ebp	0xBFFFF6A8
esp	0xBFFFF6A8

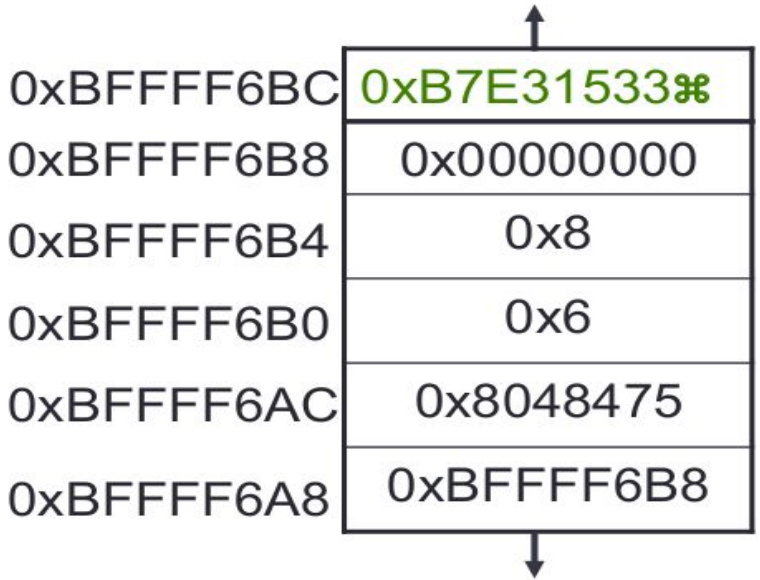
```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0x8(%ebp),%edx
0x8048462 mov         0xC(%ebp),%eax
0x8048465 add         %edx,%eax   ☒
0x8048467 pop         %ebp
0x8048465 ret
main:
0x8048469 push        %ebp
0x804846a mov         %esp, %ebp
0x804846c push        $0x8
0x804846e push        $0x6
0x8048470 call        0x804845c <add>
0x8048475 add         0x8, %esp
0x804846e mov         0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:

☒ executed instruction,

♍ modified value

⌘ start value



Example2.c 12

eax	0xE
edx	0x6
ebp	0xBFFFF6B8
esp	0xBFFFF6AC♍

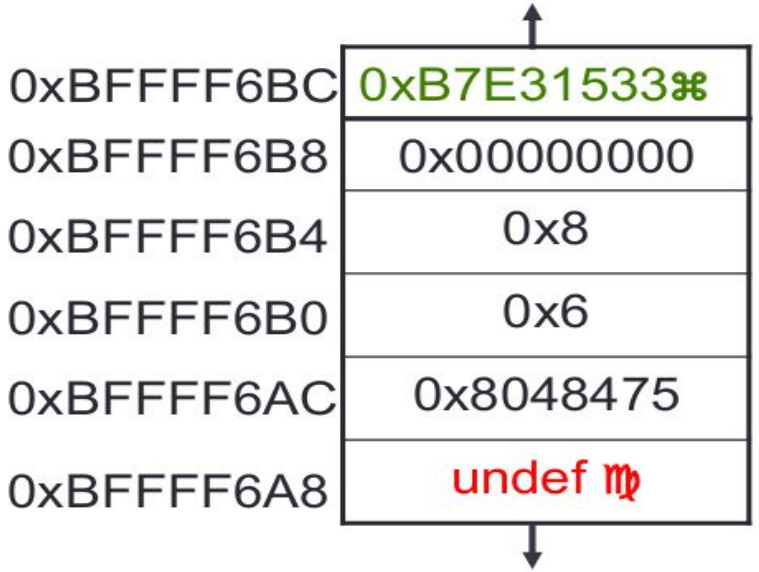
```
add:
0x804845c push          ebp
0x804845d mov           esp,ebp
0x804845f mov           0x8(%ebp),%edx
0x8048462 mov           0xC(%ebp),%eax
0x8048465 add           %edx,%eax
0x8048467 pop           %ebp    ⌧
0x8048465 ret
main:
0x8048469 push          %ebp
0x804846a mov           %esp, %ebp
0x804846c push          $0x8
0x804846e push          $0x6
0x8048470 call          0x804845c <add>
0x8048475 add           0x8, %esp
0x804846e mov           0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:

⌧ executed instruction,

♍ modified value

⌘ start value



Example2.c 13

eax	0xE♍
edx	0x6
ebp	0xBFFFF6B8
esp	0xBFFFF6B0♍

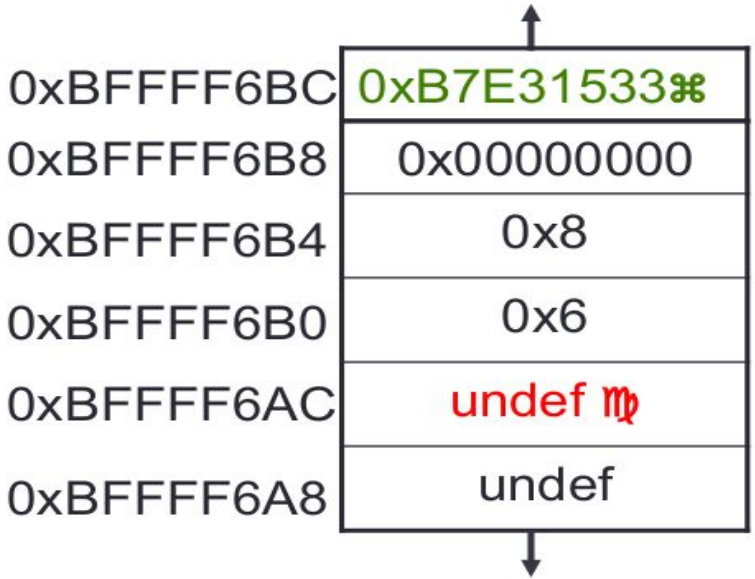
```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0x8(%ebp),%edx
0x8048462 mov         0xC(%ebp),%eax
0x8048465 add         %edx,%eax
0x8048467 pop         %ebp
0x8048465 ret   ☒
main:
0x8048469 push        %ebp
0x804846a mov         %esp, %ebp
0x804846c push        $0x8
0x804846e push        $0x6
0x8048470 call        0x804845c <add>
0x8048475 add         0x8, %esp
0x804846e mov         0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:

☒ executed instruction,

♍ modified value

⌘ start value



Example2.c 14

eax	0xE♍
edx	0x6
ebp	0xBFFFF6B8♍
esp	0xBFFFF6B0♍

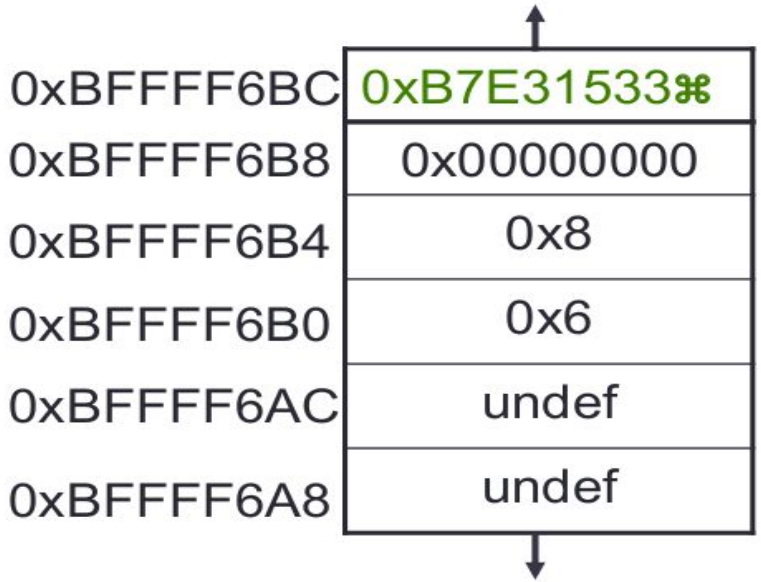
```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0x8(%ebp),%edx
0x8048462 mov         0xC(%ebp),%eax
0x8048465 add         %edx,%eax
0x8048467 pop         %ebp
0x8048465 ret
main:
0x8048469 push        %ebp
0x804846a mov         %esp, %ebp
0x804846c push        $0x8
0x804846e push        $0x6
0x8048470 call        0x804845c <add>
0x8048475 add         0x8, %esp   ⊠
0x804846e mov         0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

Key:

⊠ executed instruction,

♍ modified value

⌘ start value



Example3.c

The stack frame now also contains local variables

```
//Example1 - using the stack
//to call subroutines
#include <stdio.h>
int add(int x, int y){
        int a = 0xfeed;
        int b=0xdead, c = 0xbeef;
        int z = x + y;
        return z;
}

int main(){
        add(6,8);
        return 0xf00d;
}
```

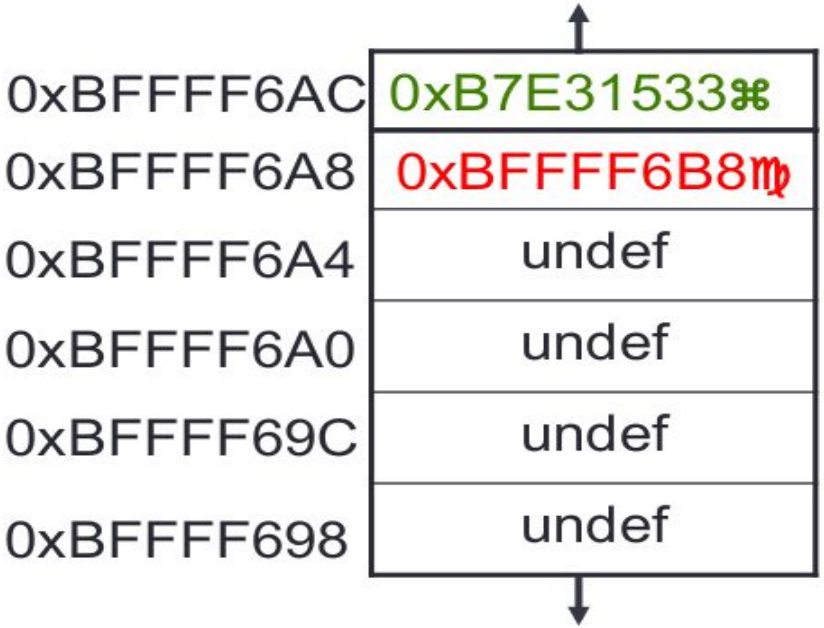
```
add:
0x804845c push   ebp
0x804845d mov    esp,ebp
0x804845f sub    $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)
0x8048469 movl   0xdead,-0xC(%ebp)
0x8048470 movl   0xbeef,-0x8(%ebp)
0x8048477 mov    0x8(%ebp),%edx
0x804847a mov    0xC(%ebp),%eax
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp)
0x8048482 mov    -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```

Example3.c 1

eax	0x01⌘
ebp	0xBFFFF6B8
esp	0xBFFFF6A8♍

Key:
⊠ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push   ebp   ⊠
0x804845d mov    esp,ebp
0x804845f sub        $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)
0x8048469 movl   0xdead,-0xC(%ebp)
0x8048470 movl   0xbeef,-0x8(%ebp)
0x8048477 mov    0x8(%ebp),%edx
0x804847a mov    0xC(%ebp),%eax
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp)
0x8048482 mov    -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```

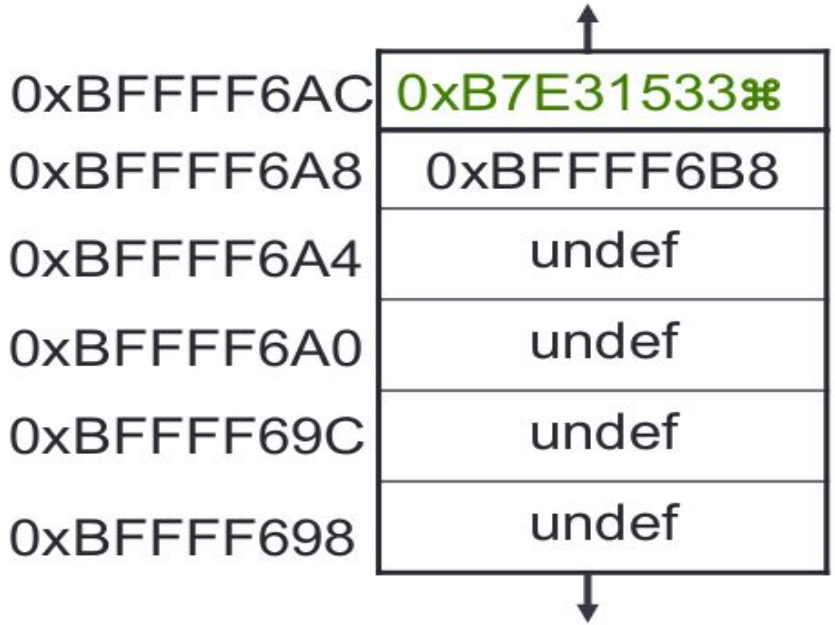


Example3.c 2

eax	0x01⌘
ebp	0xBFFFF6A8♍
esp	0xBFFFF6A8

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp ⊠
0x804845f sub     $0x10,%esp
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```

Key:
⊠ executed instruction,
♍ modified value
⌘ start value

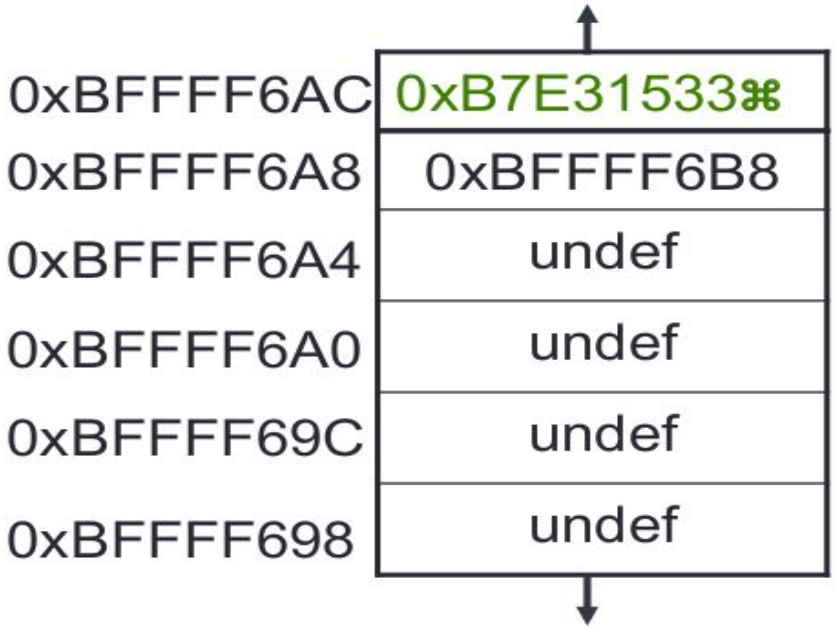


Example3.c 3

eax	0x01⌘
ebp	0xBFFFF6A8
esp	0xBFFFF698♍

Key:
⌧ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub     $0x10,%esp ⌧
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```



Example3.c 4

eax	0x01⌘
ebp	0xBFFFF6A8
esp	0xBFFFF698

```
add:
0x804845c push   ebp
0x804845d mov    esp,ebp
0x804845f sub        $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)⊠
0x8048469 movl   0xdead,-0xC(%ebp)
0x8048470 movl   0xbeef,-0x8(%ebp)
0x8048477 mov    0x8(%ebp),%edx
0x804847a mov    0xC(%ebp),%eax
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp)
0x8048482 mov    -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```

Key:

⊠ executed instruction,

♍ modified value

⌘ start value

Example3.c 5

eax	0x01⌘
ebp	0xBFFFF6A8
esp	0xBFFFF698

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

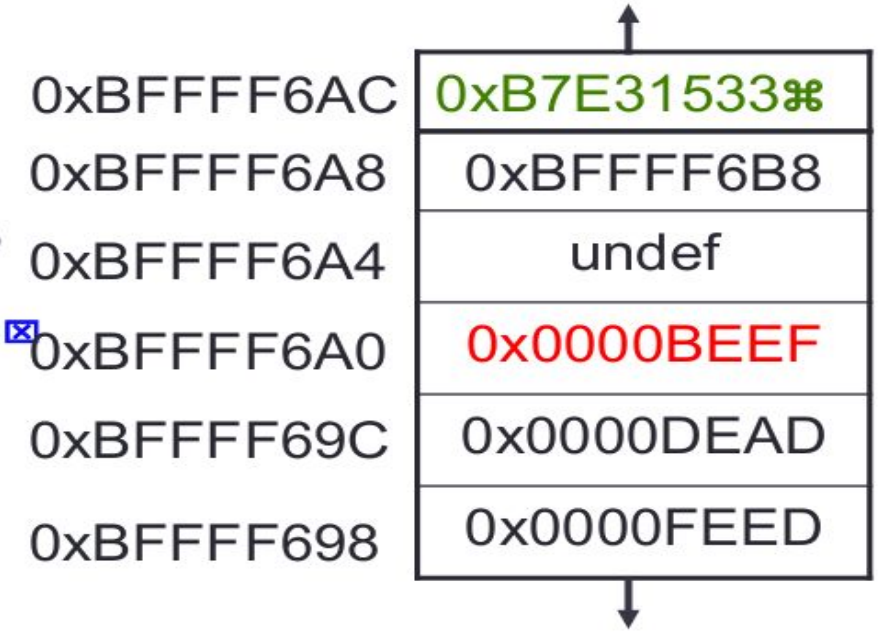
```
add:
0x804845c push   ebp
0x804845d mov    esp,ebp
0x804845f sub    $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)
0x8048469 movl   0xdead,-0xC(%ebp)   ☒
0x8048470 movl   0xbeef,-0x8(%ebp)
0x8048477 mov    0x8(%ebp),%edx
0x804847a mov    0xC(%ebp),%eax
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp)
0x8048482 mov    -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```

Example3.c 6

eax	0x01⌘
ebp	0xBFFFF6A8
esp	0xBFFFF698

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

```
add:
0x804845c push   ebp
0x804845d mov    esp,ebp
0x804845f sub    $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)
0x8048469 movl   0xdead,-0xC(%ebp)
0x8048470 movl   0xbeef,-0x8(%ebp) ☒
0x8048477 mov    0x8(%ebp),%edx
0x804847a mov    0xC(%ebp),%eax
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp)
0x8048482 mov    -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```



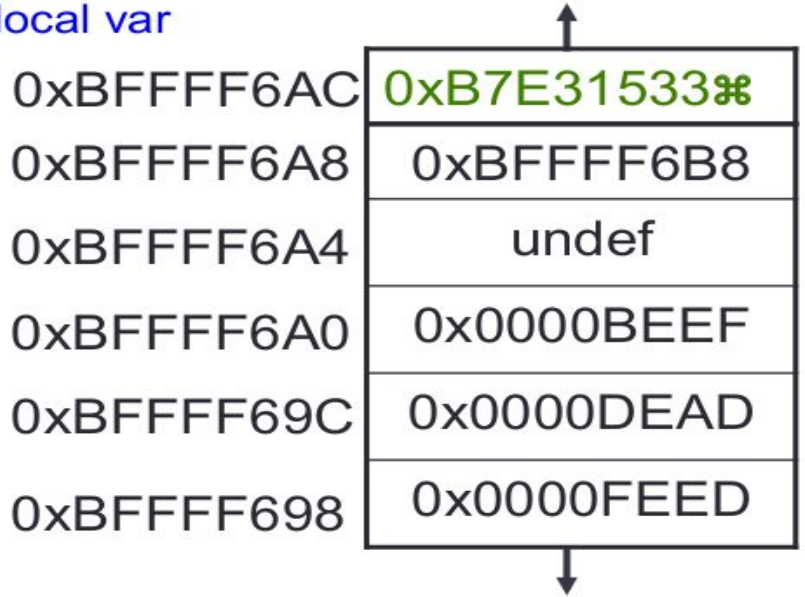
Example3.c 7

eax	01⌘
edx	0x6♍
ebp	0xBFFFF6A8
esp	0xBFFFF698

Key:
⌧ executed instruction,
♍ modified value
⌘ start value

ebp used to access local var

```
add:
0x804845c push   ebp
0x804845d mov    esp,ebp
0x804845f sub        $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)
0x8048469 movl   0xdead,-0xC(%ebp)
0x8048470 movl   0xbeef,-0x8(%ebp)
0x8048477 mov    0x8(%ebp),%edx   ⌧
0x804847a mov    0xC(%ebp),%eax
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp)
0x8048482 mov    -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```

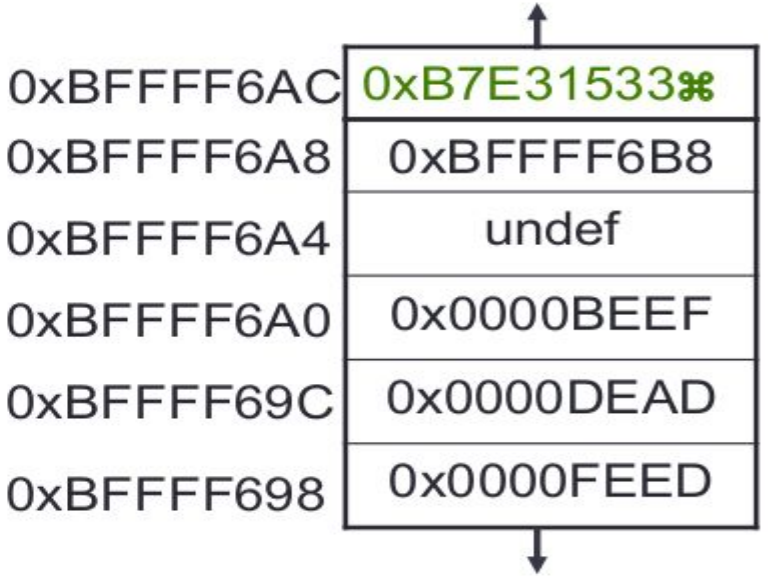


Example3.c 8

eax	0x8♍
edx	0x6
ebp	0xBFFFF6A8
esp	0xBFFFF698

Key:
⌧ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push   ebp
0x804845d mov    esp,ebp
0x804845f sub    $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)
0x8048469 movl   0xdead,-0xC(%ebp)
0x8048470 movl   0xbeef,-0x8(%ebp)
0x8048477 mov    0x8(%ebp),%edx
0x804847a mov    0xC(%ebp),%eax ⌧
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp)
0x8048482 mov    -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```

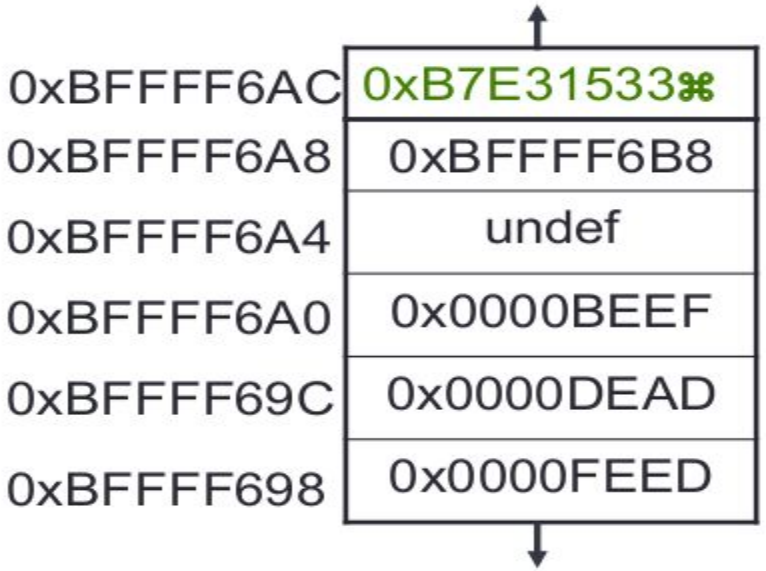


Example3.c 9

eax	0x8♍
edx	0x6
ebp	0xBFFFF6A8
esp	0xBFFFF698

Key:
⌧ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push   ebp
0x804845d mov    esp,ebp
0x804845f sub    $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)
0x8048469 movl   0xdead,-0xC(%ebp)
0x8048470 movl   0xbeef,-0x8(%ebp)
0x8048477 mov    0x8(%ebp),%edx
0x804847a mov    0xC(%ebp),%eax ⌧
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp)
0x8048482 mov    -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```

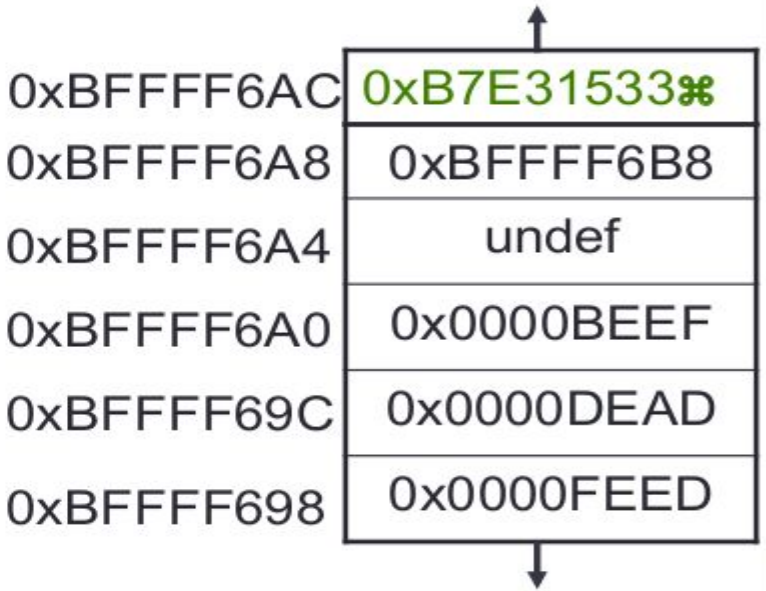


Example3.c 10

eax	0xE♍
edx	0x6
ebp	0xBFFFF6A8
esp	0xBFFFF698

Key:
⌧ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub         $0x10,%esp
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax ⌧
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```



Example3.c 11

eax	0xE
edx	0x6
ebp	0xBFFFF6A8♍
esp	0xBFFFF698♍

Key:
⌧ executed instruction,
♍ modified value
⌘ start value

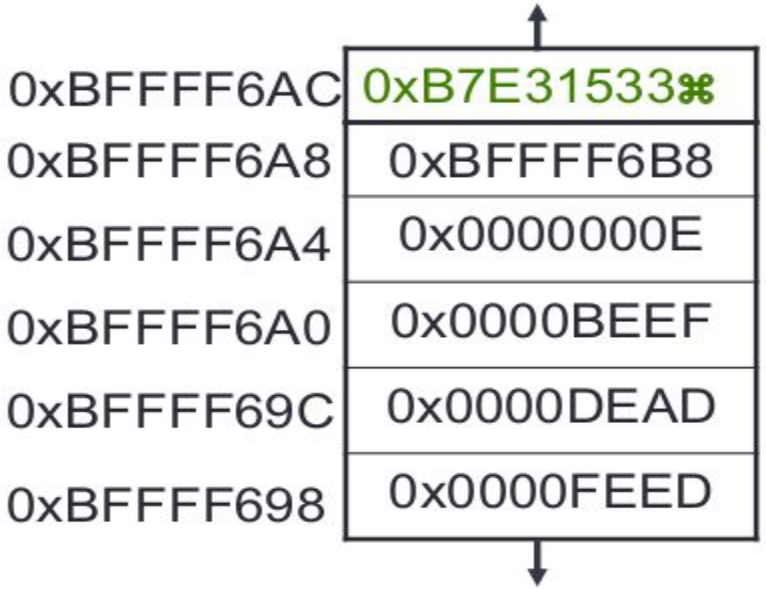
```
add:
0x804845c push   ebp
0x804845d mov    esp,ebp
0x804845f sub    $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)
0x8048469 movl   0xdead,-0xC(%ebp)
0x8048470 movl   0xbeef,-0x8(%ebp)
0x8048477 mov    0x8(%ebp),%edx
0x804847a mov    0xC(%ebp),%eax
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp) ⌧
0x8048482 mov    -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```

Example3.c 12

eax	0xE♍
edx	0x6
ebp	0xBFFFF6A8♍
esp	0xBFFFF698♍

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

```
add:
0x804845c push   ebp
0x804845d mov    esp,ebp
0x804845f sub    $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)
0x8048469 movl   0xdead,-0xC(%ebp)
0x8048470 movl   0xbeef,-0x8(%ebp)
0x8048477 mov    0x8(%ebp),%edx
0x804847a mov    0xC(%ebp),%eax
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp)
0x8048482 mov    -0x4(%ebp),%eax ☒
0x8048485 leave
0x8048486 ret
```

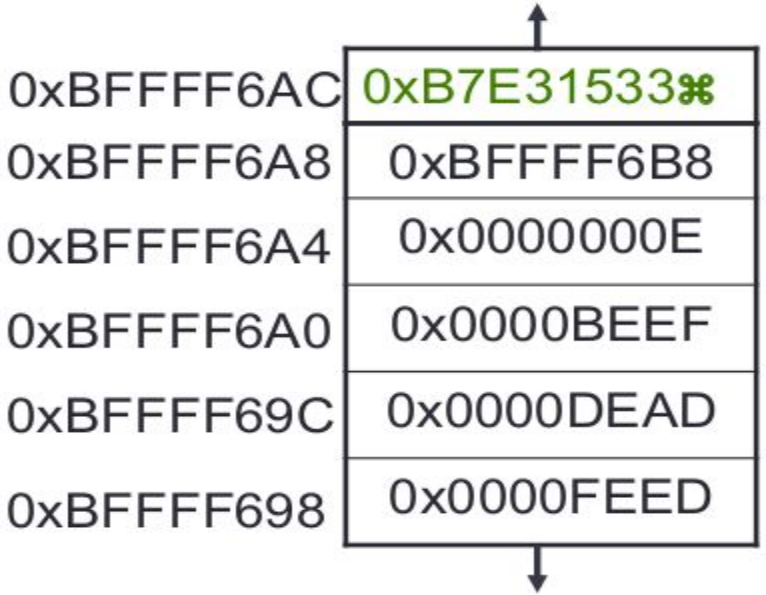


Example3.c 13

eax	0xE
edx	0x6
ebp	0xBFFFF6B8♍
esp	0xBFFFF6AC♍

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

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub     $0x10,%esp
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave   ☒
0x8048486 ret
```

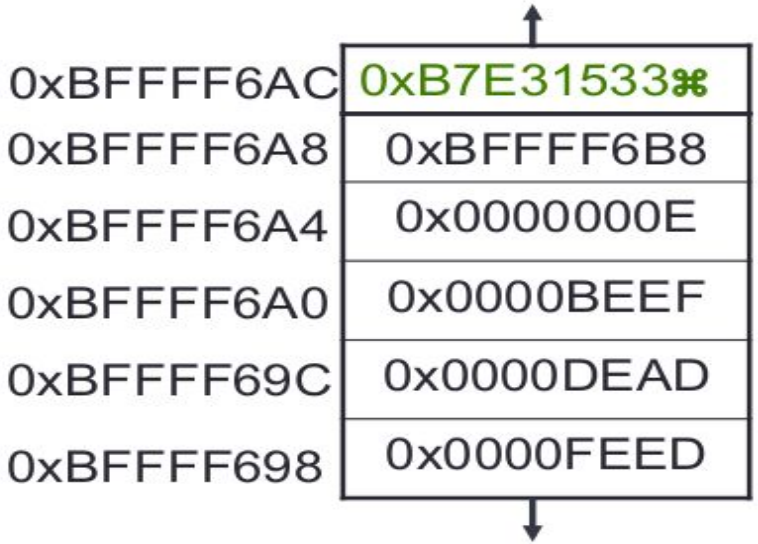


Example3.c 14

eax	0xE
edx	0x6
ebp	0xBFFFF6B8♍
esp	0xBFFFF6AC♍

```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f sub             $0x10,%esp
0x8048462 movl        0xfeed,-0x10(%ebp)
0x8048469 movl        0xdead,-0xC(%ebp)
0x8048470 movl        0xbeef,-0x8(%ebp)
0x8048477 mov         0x8(%ebp),%edx
0x804847a mov         0xC(%ebp),%eax
0x804847d add         %edx,%eax
0x804847f mov         %eax, -0x4(%ebp)
0x8048482 mov         -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret  ☒ updates EIP to return
   address in main
```

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



Tryout

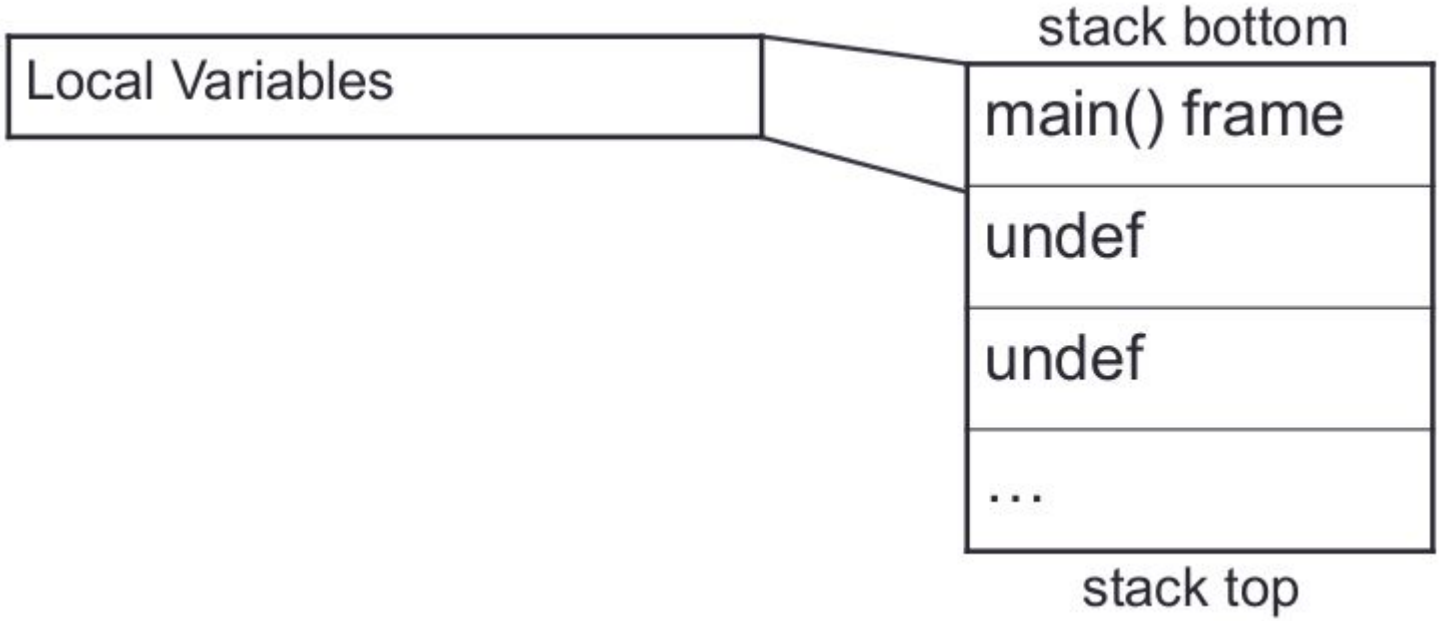
Example2.c compile : gcc –ggdb example2.c –o example2

```
add:
0x804845c push   ebp
0x804845d mov    esp,ebp
0x804845f sub    $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)
0x8048469 movl   0xdead,-0xC(%ebp)
0x8048470 movl   0xbeef,-0x8(%ebp)
0x8048477 mov    0x8(%ebp),%edx
0x804847a mov    0xC(%ebp),%eax
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp)
0x8048482 mov    -0x4(%ebp),%eax
0x8048485 leave
```

Why are these two instructions there?

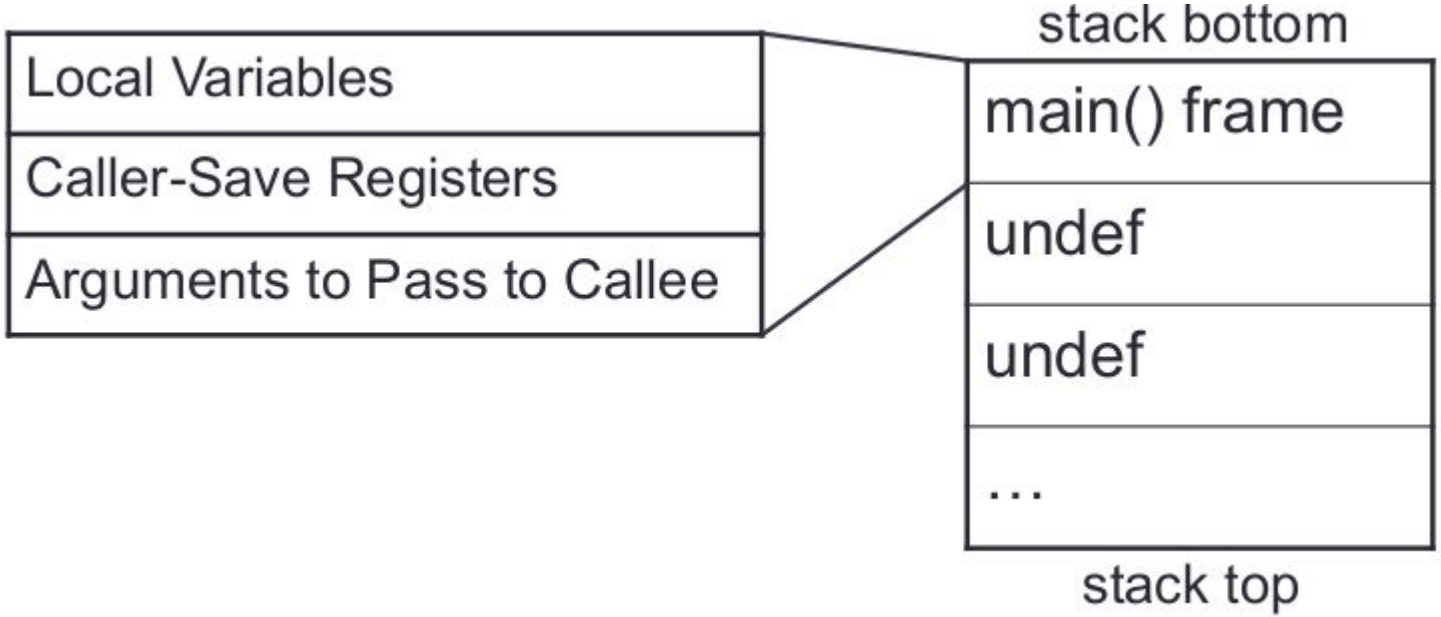
General Stack Frame Operation

We are going to pretend that main() is the very first function being executed in a program. This is what its stack looks like to start with (assuming it has any local variables).



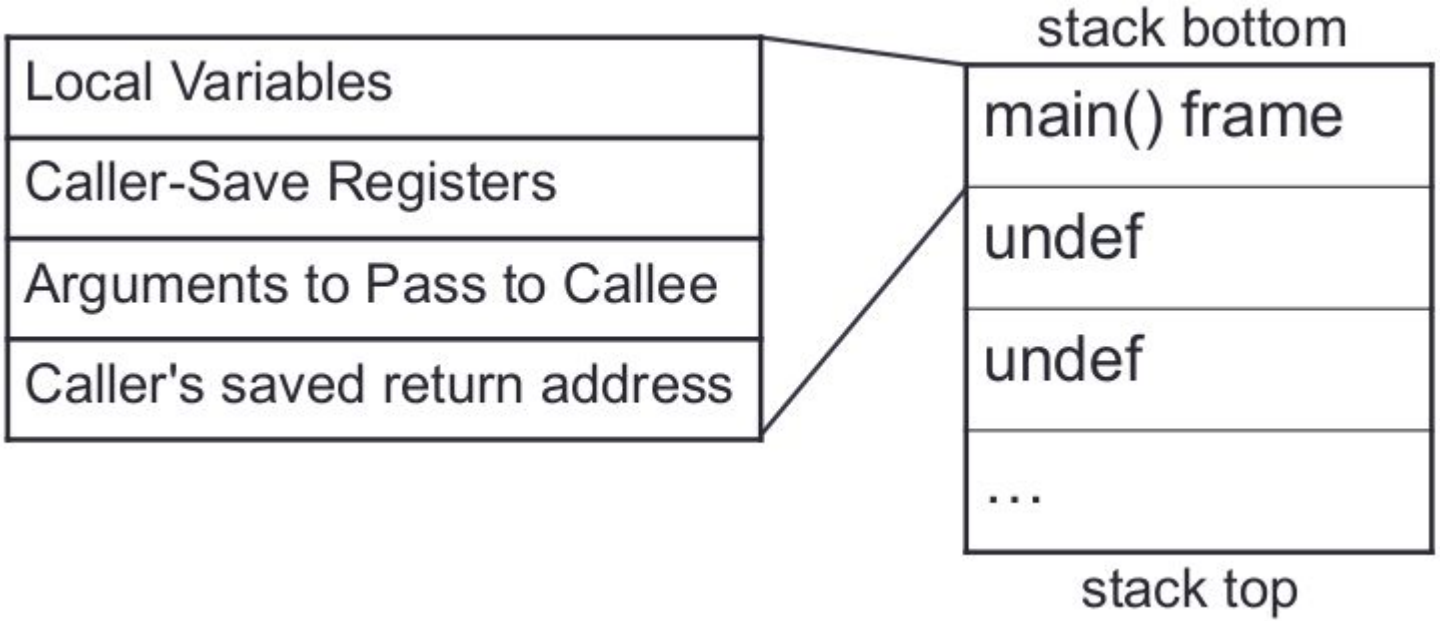
General Stack Frame Operation 2

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 takes some input arguments.



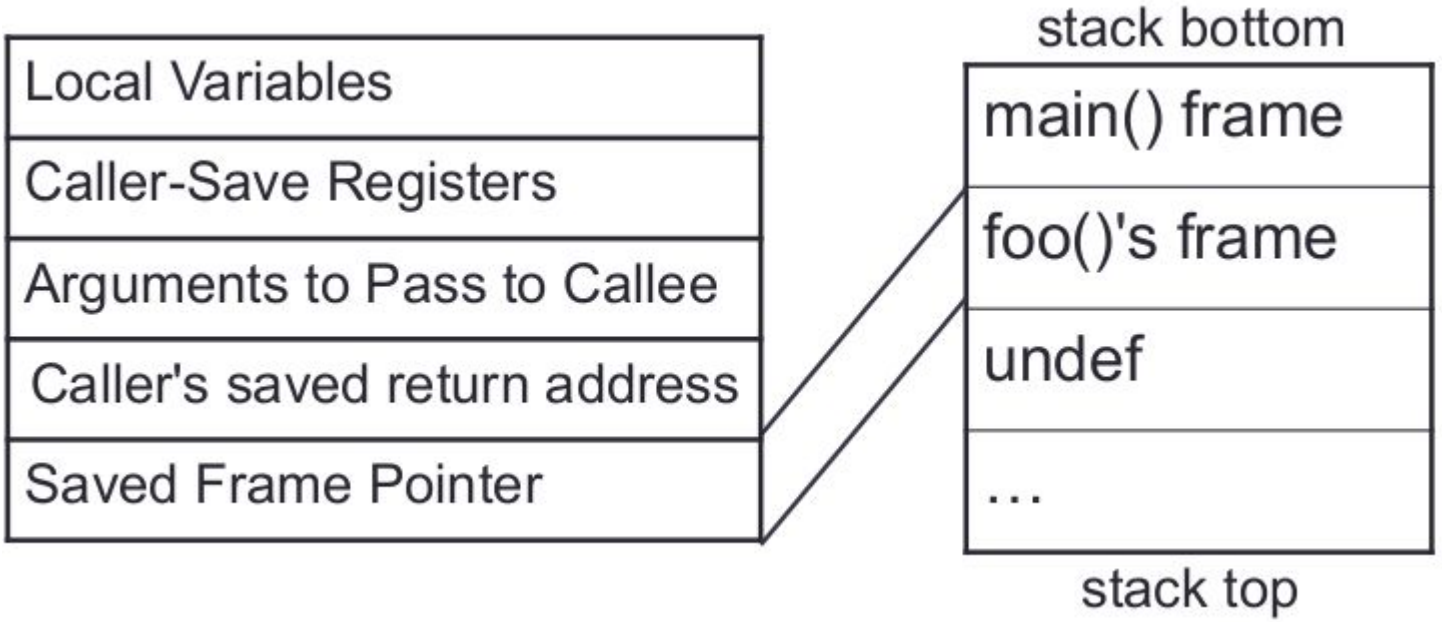
General Stack Frame Operation 3

When main() actually issues the CALL instruction, the return address gets saved 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.



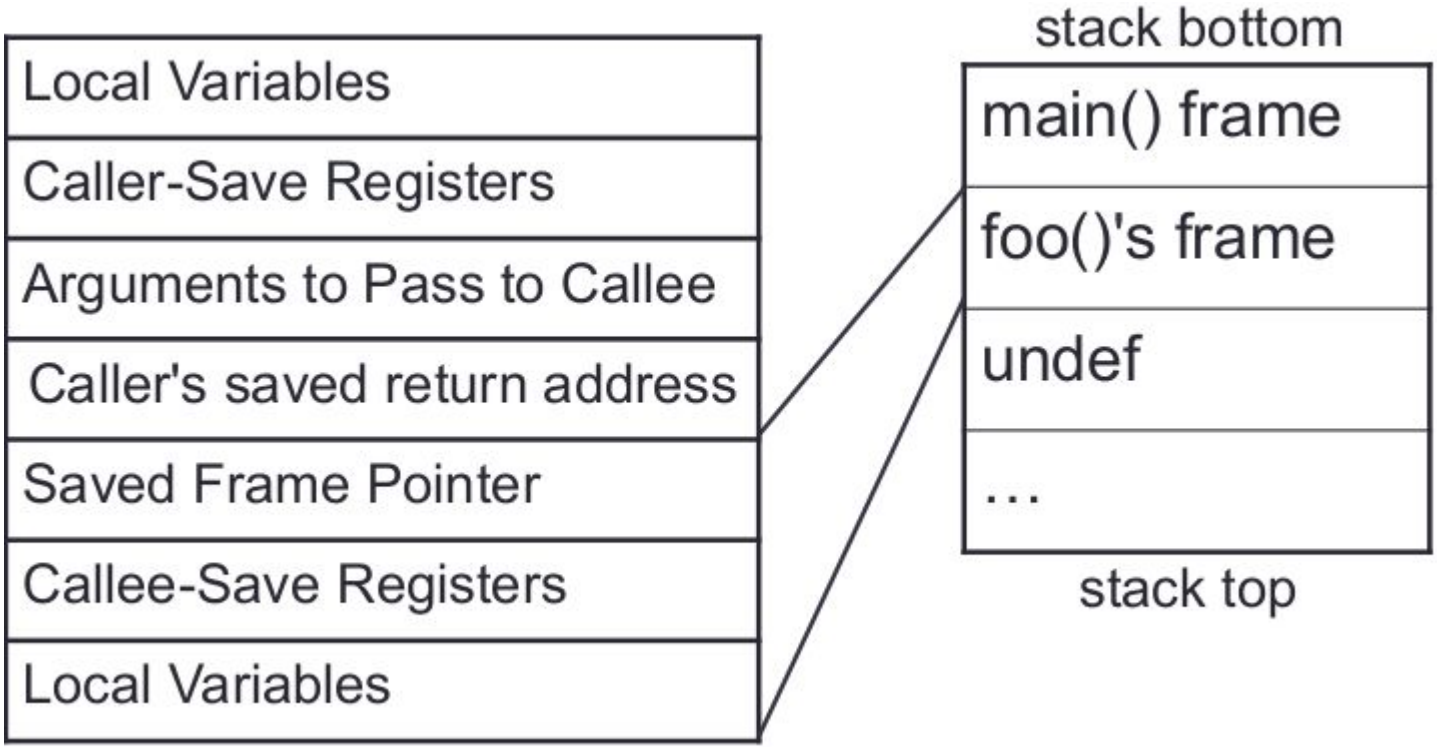
General Stack Frame Operation 4

When foo() starts, the frame pointer (ebp) still points to 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.



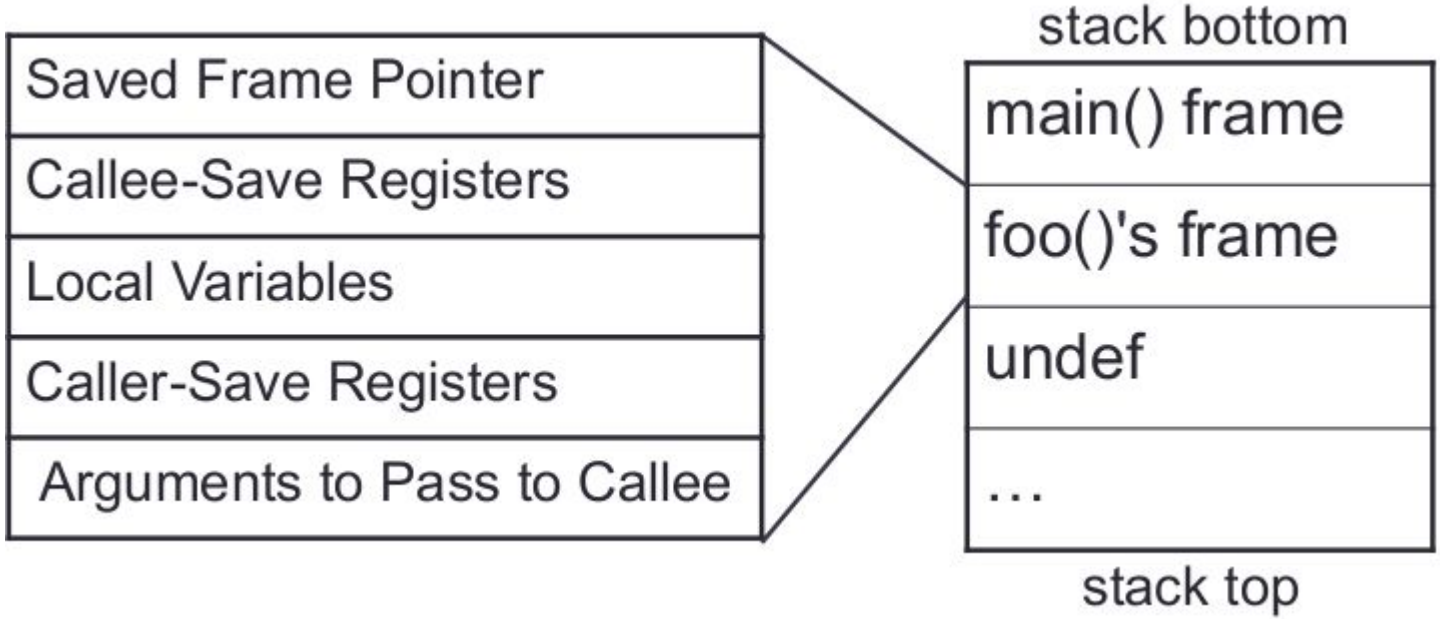
General Stack Frame Operation 5

Next, we'll assume the the callee foo() would like to use all the registers,and must therefore save the callee-save registers. Then it will allocate space for its local variables.



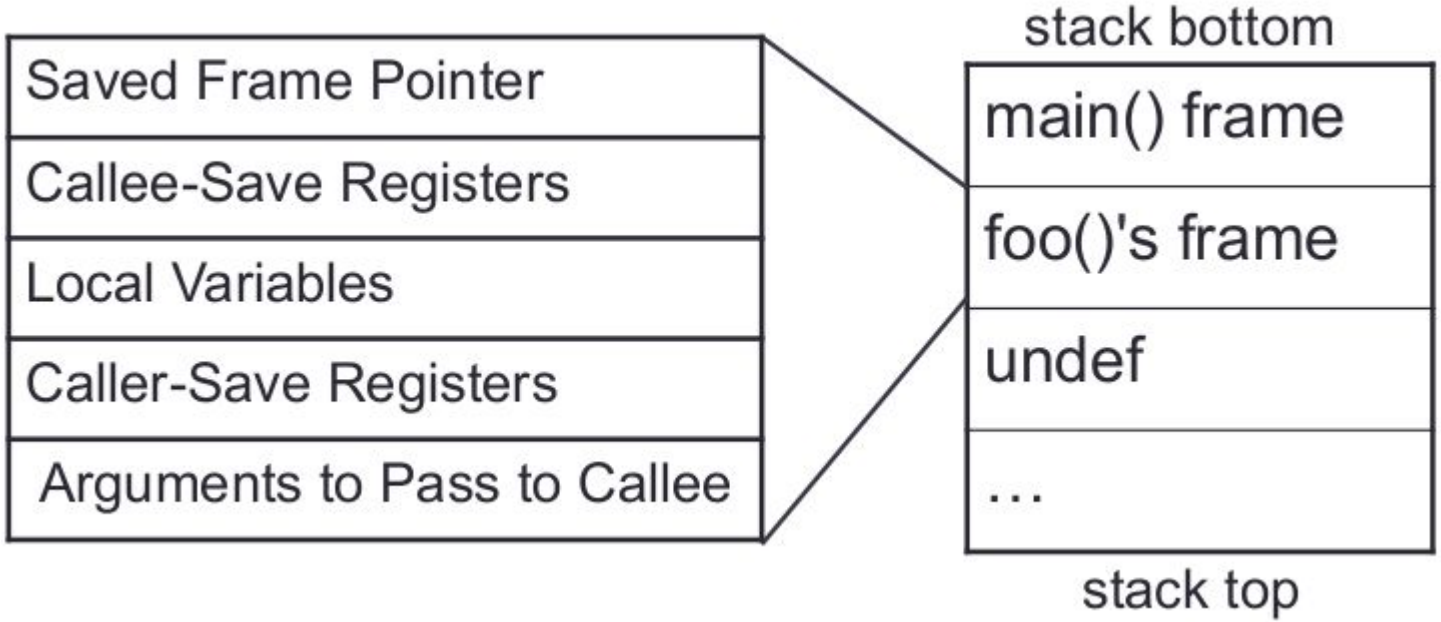
General Stack Frame Operation 6

At this point, foo() decides it wants to call bar(). It is still the callee-of-main(), but it will now be the caller-of-bar. So it saves any caller-save registers that it needs to. It then puts the function arguments on the stack as well.



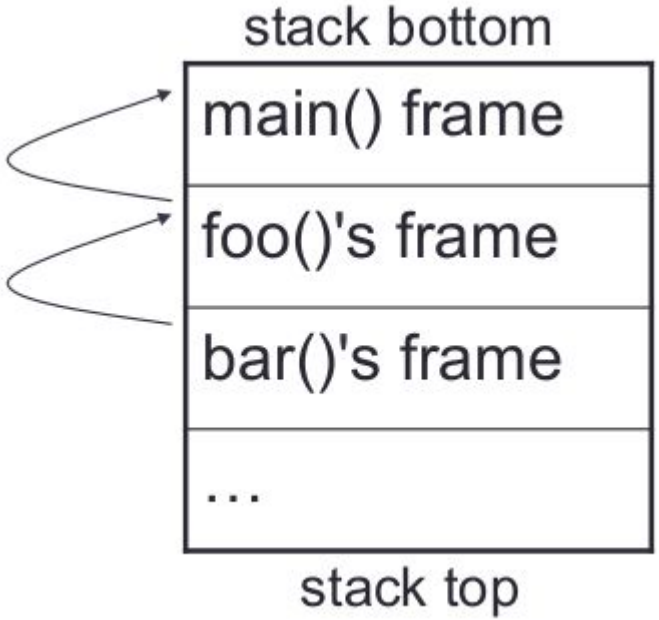
General Stack Frame Layout

Every part of the stack frame is technically optional (that is, you can hand code asm without following the conventions.) But compilers generate code which uses portions if they are needed. Which pieces are used can sometimes be manipulated with compiler options. (E.g. omit frame pointers, changing calling convention to pass arguments in registers, etc.)



Stack Frames are a Linked List!

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



Function Prologue

- Before calling a function the caller prepares the parameters by either setting specific registers or by pushing them on the stack.
- The prologue of the called function
  - Pushes the current base pointer onto a stack
  - Sets the base pointer to the current stack pointer
  - Moves the stack pointer onward to make room for local variables
    - push %ebp
    - mov %esp, %ebp
    - sub $n, %esp /* n is the size of local vars */
  - Assembly ENTER opcode is a short hand for this.

Epilogue

- The epilogue of the called function
  - Saves the result (if any) in the %eax register
  - Stores the base pointer into the stack pointer (deletes the current stack frame)
  - Pops a value from the stack, restoring the saved base pointer
  - Executes a ret
- The second and third operations are equivalent to LEAVE opcode.
  - mov ebp, esp
  - pop ebp

Tryout1 – What does this program do?

```
0x080483e4 <+0>:     push   %ebp
0x080483e5 <+1>:     mov    %esp,%ebp
0x080483e7 <+3>:     and    $0xfffffff0,%esp
0x080483ea <+6>:     sub    $0x20,%esp
0x080483ed <+9>:     movl   $0x1,0x1c(%esp)
0x080483f5 <+17>:    jmp    0x8048411 <main+45>
0x080483f7 <+19>:    mov    $0x80484f0,%eax
0x080483fc <+24>:    mov    0x1c(%esp),%edx
0x08048400 <+28>:    mov    %edx,0x4(%esp)
0x08048404 <+32>:    mov    %eax,(%esp)
0x08048407 <+35>:    call   0x8048300 <printf@plt>
0x0804840c <+40>:    addl   $0x1,0x1c(%esp)
0x08048411 <+45>:    cmpl   $0xa,0x1c(%esp)
0x08048416 <+50>:    jle    0x80483f7 <main+19>
0x08048418 <+52>:    leave
0x08048419 <+53>:    ret
```

Tryout1 – What does this program do?

Identify the function prolog, how many bytes allocated for local vars?

```
0x080483e4 <+0>:     push   %ebp
0x080483e5 <+1>:     mov    %esp,%ebp
0x080483e7 <+3>:     and    $0xfffffff0,%esp
0x080483ea <+6>:     sub    $0x20,%esp
0x080483ed <+9>:     movl   $0x1,0x1c(%esp)
0x080483f5 <+17>:    jmp    0x8048411 <main+45>
0x080483f7 <+19>:    mov    $0x80484f0,%eax
0x080483fc <+24>:    mov    0x1c(%esp),%edx
0x08048400 <+28>:    mov    %edx,0x4(%esp)
0x08048404 <+32>:    mov    %eax,(%esp)
0x08048407 <+35>:    call   0x8048300 <printf@plt>
0x0804840c <+40>:    addl   $0x1,0x1c(%esp)
0x08048411 <+45>:    cmpl   $0xa,0x1c(%esp)
0x08048416 <+50>:    jle    0x80483f7 <main+19>
0x08048418 <+52>:    leave
0x08048419 <+53>:    ret
```

Tryout1 – What does this program do?

How many parameters to printf? What are the possible parameters?

```
0x080483e4 <+0>:     push   %ebp
0x080483e5 <+1>:     mov    %esp,%ebp
0x080483e7 <+3>:     and    $0xfffffff0,%esp
0x080483ea <+6>:     sub    $0x20,%esp
0x080483ed <+9>:     movl   $0x1,0x1c(%esp)
0x080483f5 <+17>:    jmp    0x8048411 <main+45>
0x080483f7 <+19>:    mov    $0x80484f0,%eax
0x080483fc <+24>:    mov    0x1c(%esp),%edx
0x08048400 <+28>:    mov    %edx,0x4(%esp)
0x08048404 <+32>:    mov    %eax,(%esp)
0x08048407 <+35>:    call   0x8048300 <printf@plt>
0x0804840c <+40>:    addl   $0x1,0x1c(%esp)
0x08048411 <+45>:    cmpl   $0xa,0x1c(%esp)
0x08048416 <+50>:    jle    0x80483f7 <main+19>
0x08048418 <+52>:    leave
0x08048419 <+53>:    ret
```

Tryout1 – What does this program do?

What do you think the program does overall?

Cmpl $0xa, 0x1c (%esp) if value in stack is less than 0xa

```
0x080483e4 <+0>:     push   %ebp
0x080483e5 <+1>:     mov    %esp,%ebp
0x080483e7 <+3>:     and    $0xfffffff0,%esp
0x080483ea <+6>:     sub    $0x20,%esp
0x080483ed <+9>:     movl   $0x1,0x1c(%esp)
0x080483f5 <+17>:    jmp    0x8048411 <main+45>
0x080483f7 <+19>:    mov    $0x80484f0,%eax
0x080483fc <+24>:    mov    0x1c(%esp),%edx
0x08048400 <+28>:    mov    %edx,0x4(%esp)
0x08048404 <+32>:    mov    %eax,(%esp)
0x08048407 <+35>:    call   0x8048300 <printf@plt>
0x0804840c <+40>:    addl   $0x1,0x1c(%esp)
0x08048411 <+45>:    cmpl   $0xa,0x1c(%esp)
0x08048416 <+50>:    jle    0x80483f7 <main+19>
0x08048418 <+52>:    leave
0x08048419 <+53>:    ret
```

Tryout1- reversing/example1.c

```
#include <stdio.h>
int main() {
int i = 1;
for (; i <= 10; i++)
printf("%d\t",i);
}
```

Tryout1- reversing/example2.c

```
#include <stdio.h>
void main()
{
int num=7;
if (num > 0)
printf("%d is a positive number \n", num);
else if (num < 0)
printf("%d is a negative number \n", num);
else
printf("0 is neither positive nor negative");
}
```

```
0x080483e4 <+0>:     push   %ebp
0x080483e5 <+1>:     mov    %esp,%ebp                Function Prolog
0x080483e7 <+3>:     and    $0xfffffff0,%esp
0x080483ea <+6>:     sub    $0x20,%esp
0x080483ed <+9>:     movl   $0x7,0x1c(%esp)
0x080483f5 <+17>:    cmpl   $0x0,0x1c(%esp)
0x080483fa <+22>:    jle    0x8048413 <main+47>
0x080483fc <+24>:    mov    $0x8048510,%eax
0x08048401 <+29>:    mov    0x1c(%esp),%edx
0x08048405 <+33>:    mov    %edx,0x4(%esp)
0x08048409 <+37>:    mov    %eax,(%esp)
0x0804840c <+40>:    call   0x8048300 <printf@plt>
0x08048411 <+45>:    jmp    0x804843e <main+90>
0x08048413 <+47>:    cmpl   $0x0,0x1c(%esp)
0x08048418 <+52>:    jns    0x8048431 <main+77>
0x0804841a <+54>:    mov    $0x804852a,%eax
0x0804841f <+59>:    mov    0x1c(%esp),%edx
0x08048423 <+63>:    mov    %edx,0x4(%esp)
0x08048427 <+67>:    mov    %eax,(%esp)
0x0804842a <+70>:    call   0x8048300 <printf@plt>
0x0804842f <+75>:    jmp    0x804843e <main+90>
0x08048431 <+77>:    mov    $0x8048544,%eax
0x08048436 <+82>:    mov    %eax,(%esp)
0x08048439 <+85>:    call   0x8048300 <printf@plt>
0x0804843e <+90>:    leave
0x0804843f <+91>:    ret
```

```
0x080483ed    <+9>:     movl  $0x7,0x1c(%esp)
0x080483f5    <+17>:    cmpl  $0x0,0x1c(%esp)
0x080483fa    <+22>:    jle   0x8048413 <main+47>
0x080483fc    <+24>:    mov         $0x8048510,%eax
0x08048401    <+29>:    mov         0x1c(%esp),%edx
0x08048405    <+33>:    mov         %edx,0x4(%esp)
0x08048409    <+37>:    mov         %eax,(%esp)
0x0804840c    <+40>:    call        0x8048300 <printf@plt>
0x08048411    <+45>:    jmp         0x804843e <main+90>
0x08048413    <+47>:    cmpl  $0x0,0x1c(%esp)
0x08048418    <+52>:    jns         0x8048431 <main+77>
0x0804841a    <+54>:    mov         $0x804852a,%eax
0x0804841f    <+59>:    mov         0x1c(%esp),%edx
0x08048423    <+63>:    mov         %edx,0x4(%esp)
0x08048427    <+67>:    mov         %eax,(%esp)
0x0804842a    <+70>:    call        0x8048300 <printf@plt>
0x0804842f    <+75>:    jmp         0x804843e <main+90>
0x08048431    <+77>:    mov         $0x8048544,%eax
0x08048436    <+82>:    mov         %eax,(%esp)
0x08048439    <+85>:    call        0x8048300 <printf@plt>
0x0804843e    <+90>:    leave
0x0804843f    <+91>:    ret
```

First control flow path

```
0x080483ed <+9>:     movl   $0x7,0x1c(%esp)
0x080483f5 <+17>:    cmpl   $0x0,0x1c(%esp)
0x080483fa <+22>:    jle    0x8048413 <main+47>
0x080483fc <+24>:    mov    $0x8048510,%eax
0x08048401 <+29>:    mov    0x1c(%esp),%edx
0x08048405 <+33>:    mov    %edx,0x4(%esp)
0x08048409 <+37>:    mov    %eax,(%esp)
0x0804840c <+40>:    call   0x8048300 <printf@plt>
0x08048411 <+45>:    jmp    0x804843e <main+90>
0x08048413 <+47>:    cmpl   $0x0,0x1c(%esp)
0x08048418 <+52>:    jns    0x8048431 <main+77>
0x0804841a <+54>:    mov    $0x804852a,%eax
0x0804841f <+59>:    mov    0x1c(%esp),%edx
0x08048423 <+63>:    mov    %edx,0x4(%esp)
0x08048427 <+67>:    mov    %eax,(%esp)
0x0804842a <+70>:    call   0x8048300 <printf@plt>
0x0804842f <+75>:    jmp    0x804843e <main+90>
0x08048431 <+77>:    mov    $0x8048544,%eax
0x08048436 <+82>:    mov    %eax,(%esp)
0x08048439 <+85>:    call   0x8048300 <printf@plt>
0x0804843e <+90>:    leave
0x0804843f <+91>:    ret
```

Second control flow path

Jump if not sign

```
0x080483ed <+9>:     movl   $0x7,0x1c(%esp)
0x080483f5 <+17>:    cmpl   $0x0,0x1c(%esp)
0x080483fa <+22>:    jle    0x8048413 <main+47>
0x080483fc <+24>:    mov    $0x8048510,%eax  Third control
0x08048401 <+29>:    mov    0x1c(%esp),%edx  flow path
0x08048405 <+33>:    mov    %edx,0x4(%esp)
0x08048409 <+37>:    mov    %eax,(%esp)
0x0804840c <+40>:    call   0x8048300 <printf@plt>
0x08048411 <+45>:    jmp    0x804843e <main+90>
0x08048413 <+47>:    cmpl   $0x0,0x1c(%esp)
0x08048418 <+52>:    jns    0x8048431 <main+77>
0x0804841a <+54>:    mov    $0x804852a,%eax
0x0804841f <+59>:    mov    0x1c(%esp),%edx
0x08048423 <+63>:    mov    %edx,0x4(%esp)
0x08048427 <+67>:    mov    %eax,(%esp)
0x0804842a <+70>:    call   0x8048300 <printf@plt>
0x0804842f <+75>:    jmp    0x804843e <main+90>
0x08048431 <+77>:    mov    $0x8048544,%eax
0x08048436 <+82>:    mov    %eax,(%esp)
0x08048439 <+85>:    call   0x8048300 <printf@plt>
0x0804843e <+90>:    leave
0x0804843f <+91>:    ret
```

```
0x080483ed <+9>:     movl   $0x7,0x1c(%esp)
0x080483f5 <+17>:    cmpl   $0x0,0x1c(%esp)
0x080483fa <+22>:    jle    0x8048413 <main+47> not true
0x080483fc <+24>:    mov    $0x8048510,%eax  First control
0x08048401 <+29>:    mov    0x1c(%esp),%edx  flow path
0x08048405 <+33>:    mov    %edx,0x4(%esp)
0x08048409 <+37>:    mov    %eax,(%esp)
0x0804840c <+40>:    call   0x8048300 <printf@plt>
0x08048411 <+45>:    jmp    0x804843e <main+90>
0x0804843e <+90>:    leave
0x0804843f <+91>:    ret
```

```
gdb-peda$ x/s 0x8048510
0x8048510: "%d is a positive number \n"
```

```
jle if (destination <= source)
```

```
0x080483ed <+9>:      movl   $0x7,0x1c(%esp)
0x080483f5 <+17>:     cmpl   $0x0,0x1c(%esp)
0x080483fa <+22>:     jle    0x8048413 <main+47>
0x08048413 <+47>:     cmpl   $0x0,0x1c(%esp)
0x08048418 <+52>:     jns    0x8048431 <main+77>
0x0804841a <+54>:     mov    $0x804852a,%eax
0x0804841f <+59>:     mov    0x1c(%esp),%edx
0x08048423 <+63>:     mov    %edx,0x4(%esp)
0x08048427 <+67>:     mov    %eax,(%esp)
0x0804842a <+70>:     call   0x8048300 <printf@plt>
0x0804842f <+75>:     jmp    0x804843e <main+90>
0x0804843e <+90>:     leave
0x0804843f <+91>:     ret
```

Second control flow path

Jump if not sign

```
gdb-peda$ x/s 0x804852a
0x804852a: "%d is a negative number \n"
```

Here assume num = -7

compare will yield a signed number which sets SF

```
0x080483ed <+9>:     movl   $0x7,0x1c(%esp)
0x080483f5 <+17>:    cmpl   $0x0,0x1c(%esp)
0x080483fa <+22>:    jle    0x8048413 <main+47>
0x08048413 <+47>:    cmpl   $0x0,0x1c(%esp)
0x08048418 <+52>:    jns    0x8048431 <main+77>
0x08048431 <+77>:    mov    $0x8048544,%eax
0x08048436 <+82>:    mov    %eax,(%esp)
0x08048439 <+85>:    call   0x8048300 <printf@plt>
0x0804843e <+90>:    leave
0x0804843f <+91>:    ret
```

Third control flow path

```
gdb-peda$ x/s 0x8048544
0x8048544:    "0 is neither positive nor negative"
```

Tryout1- reversing/example3.c

```
#include <stdio.h>
int main() {
int arr[] = {1, 2, 3, 4, 5, 6};
int i = 0;
for (; i < 6; i++)
printf("%d\t",arr[i]);
}
```

```
0x080483e4 <+0>:     push   %ebp
0x080483e5 <+1>:     mov    %esp,%ebp
0x080483e7 <+3>:     and    $0xfffffff0,%esp
0x080483ea <+6>:     sub    $0x30,%esp
0x080483ed <+9>:     movl   $0x1,0x14(%esp)
0x080483f5 <+17>:    movl   $0x2,0x18(%esp)
0x080483fd <+25>:    movl   $0x3,0x1c(%esp)
0x08048405 <+33>:    movl   $0x4,0x20(%esp)
0x0804840d <+41>:    movl   $0x5,0x24(%esp)
0x08048415 <+49>:    movl   $0x6,0x28(%esp)
0x0804841d <+57>:    movl   $0x0,0x2c(%esp)
0x08048425 <+65>:    jmp    0x8048445 <main+97>
0x08048427 <+67>:    mov    0x2c(%esp),%eax
0x0804842b <+71>:    mov    0x14(%esp,%eax,4),%edx
0x0804842f <+75>:    mov    $0x8048520,%eax
0x08048434 <+80>:    mov    %edx,0x4(%esp)
0x08048438 <+84>:    mov    %eax,(%esp)
0x0804843b <+87>:    call   0x8048300 <printf@plt>
0x08048440 <+92>:    addl   $0x1,0x2c(%esp)
0x08048445 <+97>:    cmpl   $0x5,0x2c(%esp)
0x0804844a <+102>:   jle    0x8048427 <main+67>
0x0804844c <+104>:   leave
0x0804844d <+105>:   ret
```

Tryout Example3

How much space is allocated for local variables etc.?

```
int convert (char * str)
{
int result = atoi(str);
return result;
}
int main(int argc, char **argv)
{
int i, sum;
for (i=0; i < argc; i++)
sum += convert(argv[i]);
printf(“sum=%d\n”, sum);
return 0;
}
```

```
Dump of assembler code for function main:
   0x0804842d <+0>:     push   %ebp
   0x0804842e <+1>:     mov    %esp,%ebp
   0x08048430 <+3>:     and    $0xfffffff0,%esp
   0x08048433 <+6>:     sub    $0x20,%esp
   0x08048436 <+9>:     movl   $0x0,0x18(%esp)
   0x0804843e <+17>:    jmp    0x804845d <main+48>
   0x08048440 <+19>:    mov    0x18(%esp),%eax
   0x08048444 <+23>:    shl    $0x2,%eax
   0x08048447 <+26>:    add    0xc(%ebp),%eax
   0x0804844a <+29>:    mov    (%eax),%eax
   0x0804844c <+31>:    mov    %eax,(%esp)
   0x0804844f <+34>:    call   0x8048414 <convert>
   0x08048454 <+39>:    add    %eax,0x1c(%esp)
   0x08048458 <+43>:    addl   $0x1,0x18(%esp)
   0x0804845d <+48>:    mov    0x18(%esp),%eax
   0x08048461 <+52>:    cmp    0x8(%ebp),%eax
   0x08048464 <+55>:    jl     0x8048440 <main+19>
   0x08048466 <+57>:    mov    $0x8048560,%eax
   0x0804846b <+62>:    mov    0x1c(%esp),%edx
   0x0804846f <+66>:    mov    %edx,0x4(%esp)
   0x08048473 <+70>:    mov    %eax,(%esp)
   0x08048476 <+73>:    call   0x8048320 <printf@plt>
   0x0804847b <+78>:    mov    $0x0,%eax
   0x08048480 <+83>:    leave
   0x08048481 <+84>:    ret
```

Function prolog

32 bytes

Tryout Example3

How much of this can you reverse?

```
int convert (char * str)
{
int result = atoi(str);
return result;
}
int main(int argc, char **argv)
{
int i, sum;
for (i=0; i < argc; i++)
sum += convert(argv[i]);
printf(“sum=%d\n”, sum);
return 0;
}
```

Frames and Function Invocation

```
int convert (char * str)
{
int result = atoi(str);
return result;
}
int main(int argc, char **argv)
{
int i, sum;
for (i=0; i < argc; i++)
sum += convert(argv[i]);
printf("sum=%d\n", sum);
return 0;
}
```