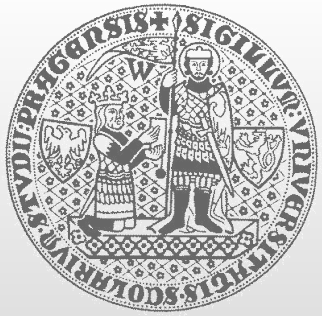
IA-32 & AMD64

Crash Dump Analysis 2015/2016



CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Department of Distributed and Dependable Systems

ORACLE®

SUSE®

redhat®

IA-32 Overview

- **32bit CISC architecture**
  - Starts with 80386
    - Also known as x86, i386, i586, i686, etc.
    - Strong inheritance from 8086, even 8080
  - Some RISC characteristics after Pentium (P5, i586)
  - Variable instruction size
  - Non-orthogonal instruction set
  - Most instructions can have memory operands

IA-32 Overview (2)

- **32bit CISC architecture (cont.)**
  - Very few GPRs (8)
    - Actually only 6 or 7 practically usable (ABI dependent)
  - Little-endian
  - Implicit stack
  - Complicated memory management
    - Several operational modes
      - Real mode (8086), V86 (virtual 8086), 16bit Protected mode (80286), 32bit Protected mode (80386+), SMM, …
    - Paging and segmentation

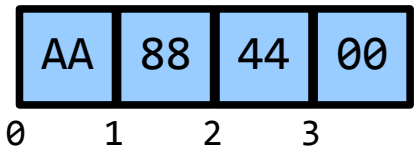
AMD64 Overview

- **Natural extension of IA-32**
  - Originally created by AMD (hence AMD64)
    - Later also adopted by Intel (as IA-32e, IA-64t, EM64T, Intel 64)
      - Vendor-neutral names such as x86-64, x64
  - Many properties of IA-32 apply also to AMD64
    - Key differences
      - 64bit architecture
      - 16 GPRs (14 practically usable)
      - Segmentation almost eliminated (except two remaining simplified segments)

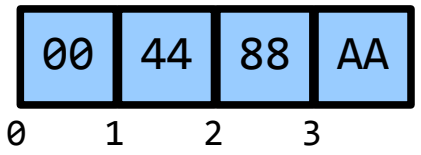
Aside: Little vs. Big Endian

- **Memory is usually addressed in bytes (8 bits)**
  - Several ways how to store values larger than one byte as a sequence of bytes (byte order)
    - *v* = `0xAA884400`

Big-Endian
big end first
most significant byte first

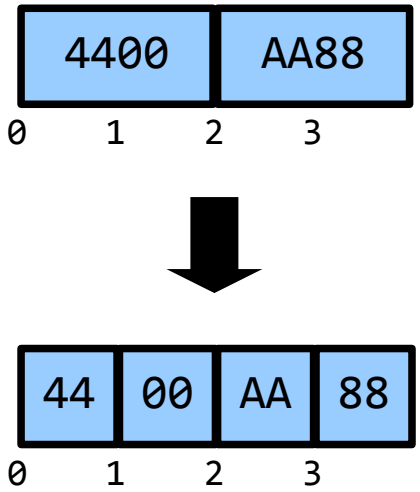


Litte-Endian
little end first
least significant byte first



Aside: Little vs. Big Endian (2)

- **Caution: Basic element size might vary**
  - Little-Endian example, byte addressing
    - Basic element size = 16 bits
    - *v* = 0xAA884400



Intel Manuals

- **Intel 64 and IA-32 Architectures Software Developer's Manual**
  - Volume 1: Basic Architecture
  - Volume 2A + 2B: Instruction Set Reference
  - Volume 3A + 3B: System Programming Guide
- **Intel 64 and IA-32 Architectures Optimization Reference Manual**

http://www.intel.com/products/processor/manuals

AMD Manuals

- **AMD64 Architecture Programmer's Manual**
  - Volume 1: Application Programming
  - Volume 2: System Programming
  - Volume 3: General-Purpose and System Instructions
- **Software Optimization Guide for AMD64 Processors**

http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/

IA-32 ABI

- **System V Application Binary Interface, Intel386 Architecture Processor Supplement**
  - Authoritative source for GNU/Linux (e.g. the GCC toolchain), *BSD, most UNIXes, etc.
  - We will use a simplified view sufficient for common cases (integer arguments, etc.)

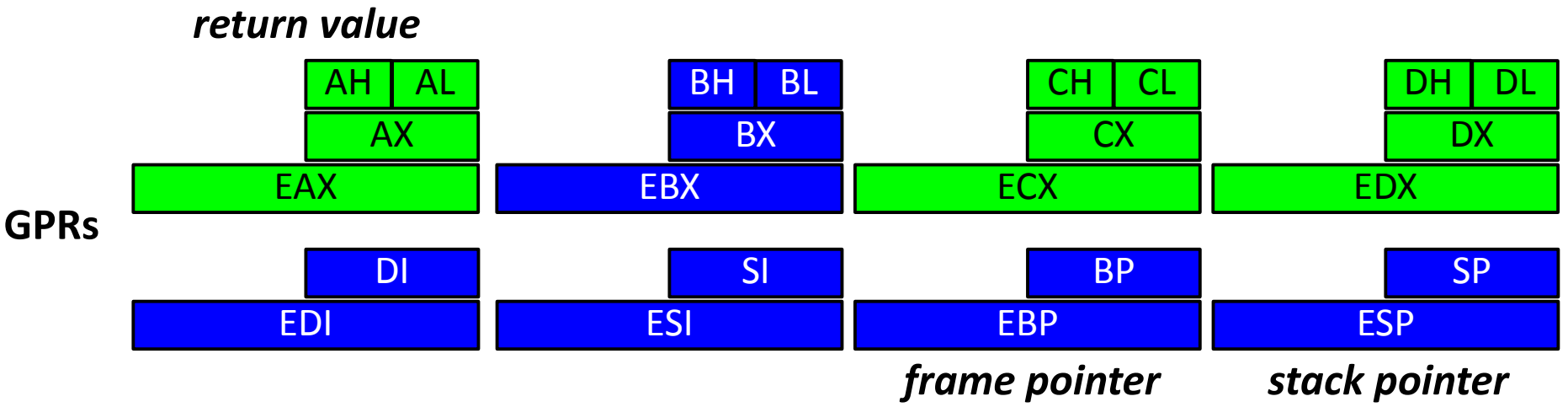
    http://www.sco.com/developers/devspecs/abi386-4.pdf

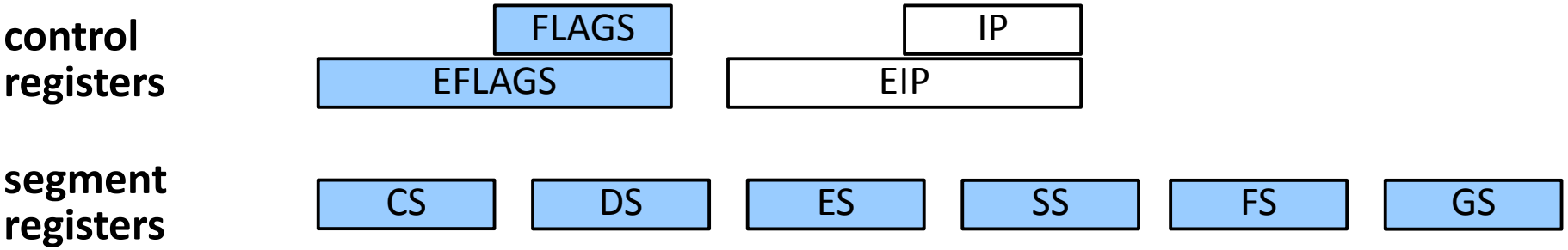
AMD64 ABI

- **System V Application Binary Interface, AMD64 Architecture Processor Supplement**
  - Authoritative source for GNU/Linux (e.g. the GCC toolchain), *BSD, most UNIXes, etc.
  - We will use a simplified view sufficient for common cases (integer arguments, etc.)

    http://www.x86-64.org/documentation/abi.pdf

IA-32 Registers





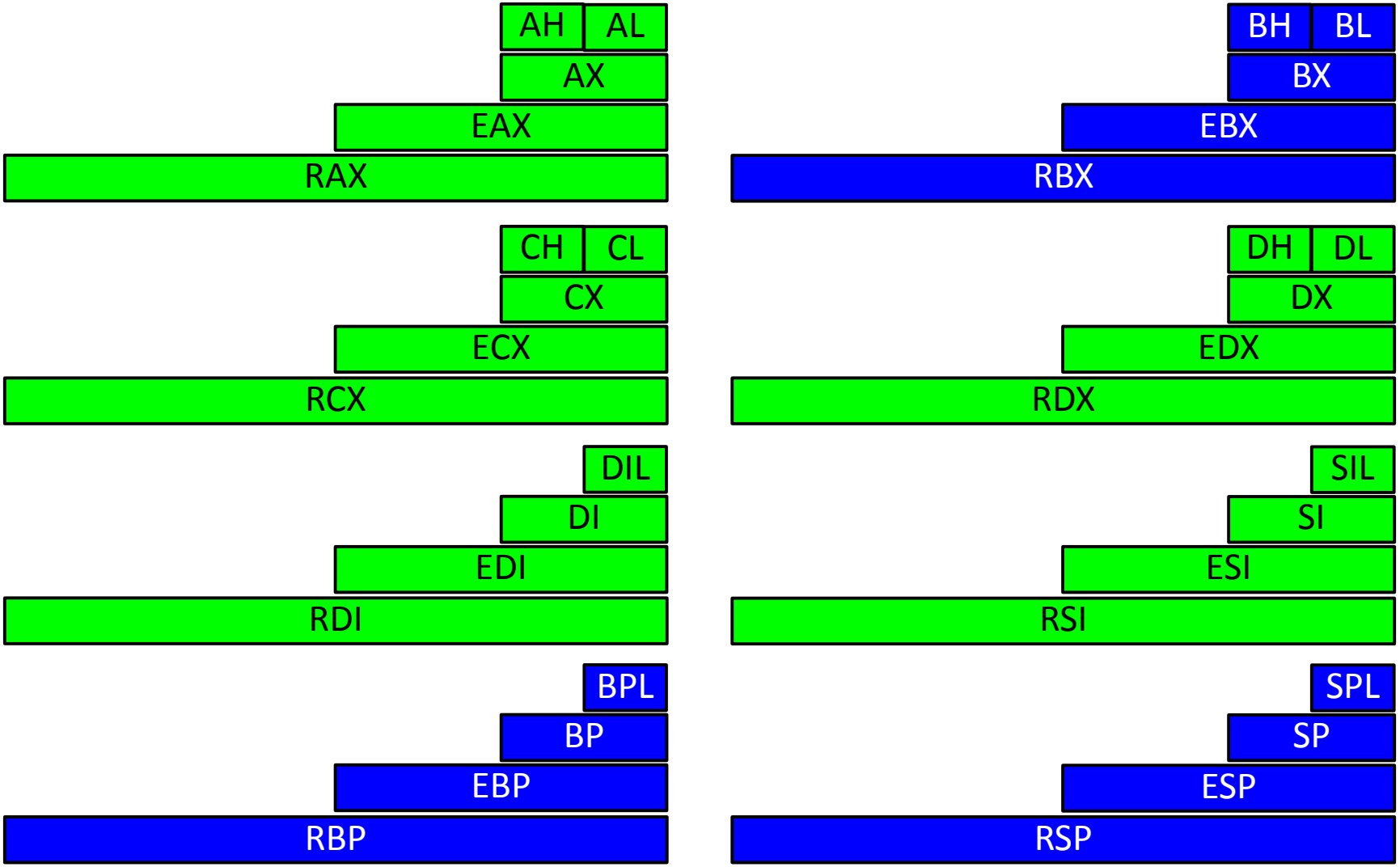
IA-32 ABI in a Nutshell

- **Arguments passed on stack**
  - In reverse order (the last argument is pushed first)
- **Return value**
  - For simple integer types in EAX
    - Otherwise on the stack
- **Implicit stack pointer**
  - Some instructions use ESP as implicit register operand

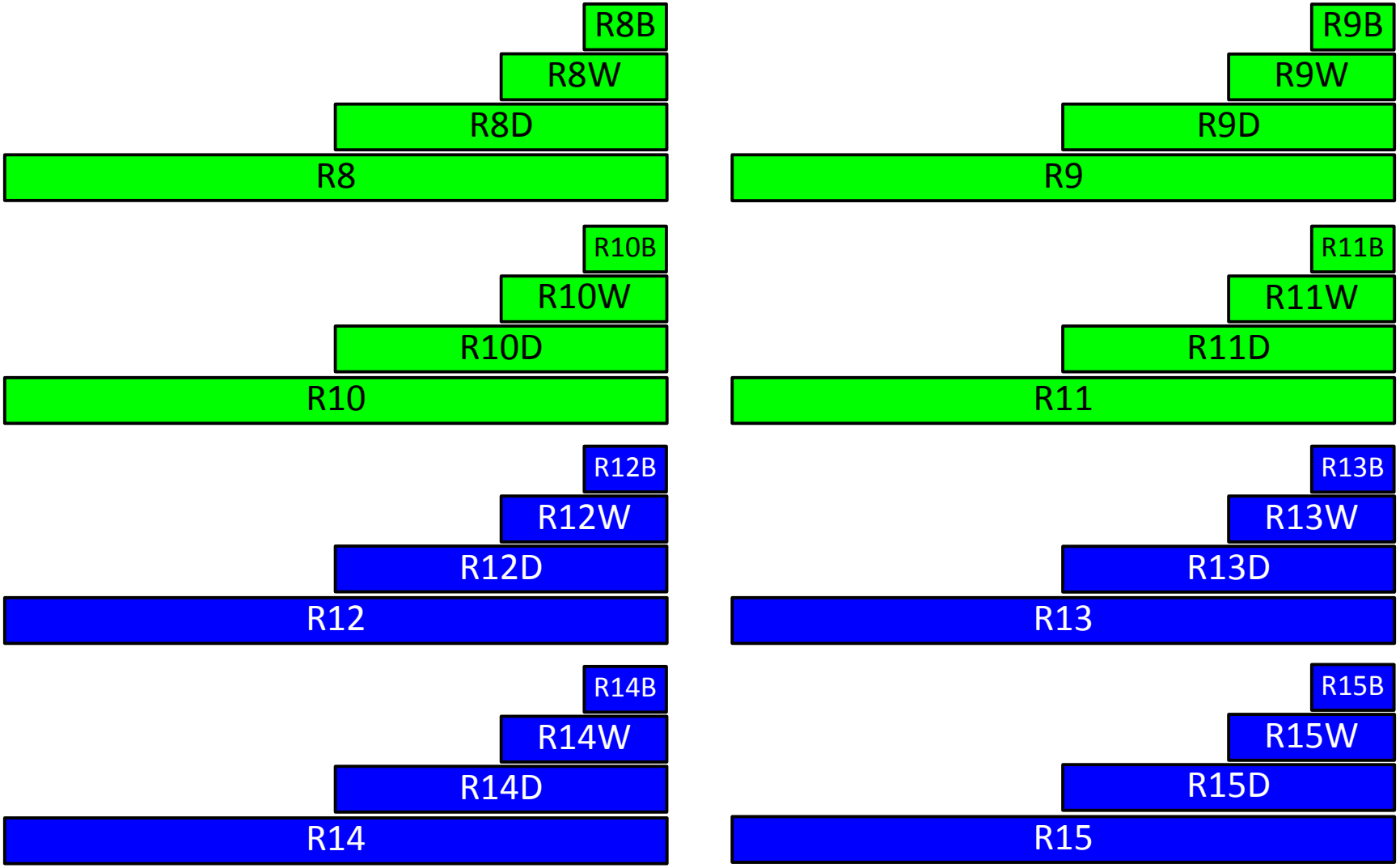
IA-32 ABI in a Nutshell (2)

- **Frame pointer**
  - Usually (not always) stored in EBP
- **Volatile (scratch, caller-saved) registers**
  - EAX, ECX, EDX
- **Non-volatile (preserved, callee-saved) registers**
  - EBX, EDI, ESI, EBP, ESP
- **Stack aligned on 4B boundary**
  - Some compilers use even larger alignment

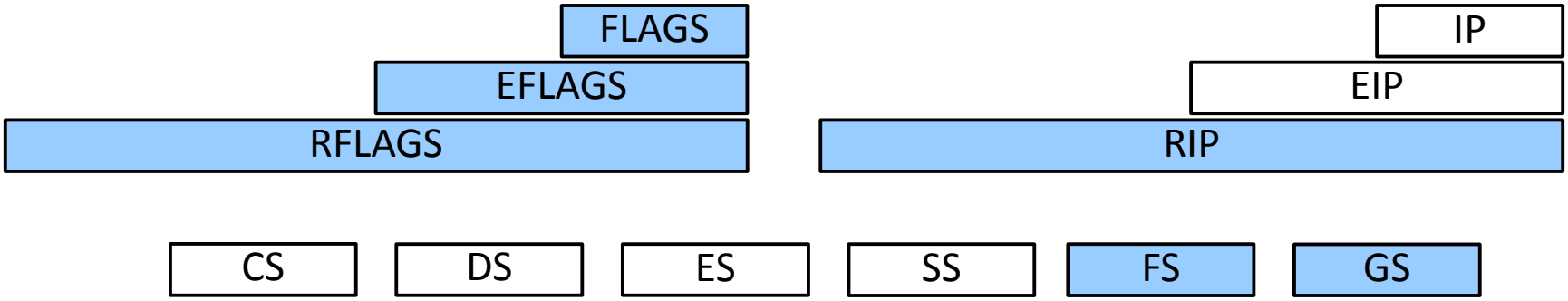
AMD64 Registers



AMD64 Registers (2)



AMD64 Registers (3)



AMD64 ABI in a Nutshell

- **First six integer arguments passed in registers**
  - RDI, RSI, RDX, RCX, R8, R9
- **Additional/complex arguments passed on stack**
  - In reverse order (the last argument is pushed first)
- **Return value**
  - For simple integer types in RAX
    - Otherwise on the stack
- **Implicit stack (RSP) and frame (RBP) pointer**

AMD64 ABI in a Nutshell (2)

- **Volatile (scratch, caller-saved) registers**
  - RAX, RCX, RDX, RDI, RSI, R8, R9, R10, R11
- **Non-volatile (preserved, callee-saved) registers**
  - RBX, RBP, RSP, R12, R13, R14, R15
- **Stack aligned on 8B boundary, but not 16B aligned on function's entry point**
  - Thus each stack frame is 16B aligned
    - Support for easy spilling of FPU and SSE registers
      - Some GCC builds apparently ignore this rule

AMD64 ABI in a Nutshell (3)

- **128B red-zone at RSP - 128**
  - Optimization
    - Functions do not need to allocate stack space
  - Signal and interrupt handlers must avoid this area
  - Sometimes the red-zone is disabled
    - `gcc -mno-red-zone`

IA-32 Instruction Set

- **Hundreds of instructions**
  - Most of them have several variants (operands as registers, operands as memory addresses, etc.)
- **Informal classification**
  - General purpose (arithmetic, logic, jumps, etc.)
  - System instructions (altering processor mode)
  - FPU instructions
  - SIMD and other instructions (MMX, SSE, etc.)

IA-32 Instruction Set (2)

- **Most general purpose instructions have two operands**
  - register – register
  - immediate – register
  - memory – register
  - immediate – memory

- *INST opl, opr*
  - AT&T syntax
    - *opr ← opr INST opl*
  - Intel syntax
    - *opl ← opl INST opr*

```
ADDL EAX, EBX
```

  - *EBX ← EBX + EAX*

IA-32 AT&T Syntax

- **Left operand: source**
- **Right operand: destination**
- **Register names prefixed by *%***
  - `%eax`
- **Immediate operands prefixed by *$***
  - `$0x1`
- **Operand size encoded as instruction suffix**
  - *b* (byte, 8 bit), *w* (word, 16 bit), *l* (long, 32 bit)
  - `movl $0x1, %eax`

IA-32 AT&T Syntax (2)

- **Memory operands**
  - Using implicit segment register
    - *displacement*(*base*, *index*, *scale*)
      - *base* and *index* are GPRs
      - *scale* is 1, 2, 4 or 8 (defaults to 1 if not specified)
      - *displacement* is an immediate offset
    - Effective address is calculated as

      EA = *displacement* + *base* + *index* * *scale*

```
0x8111f30, 0x8(%ebp), -0x28(%eax),
-0x2(%esi, %eax, 2)
mov (%esp), %edi
```

IA-32 AT&T Syntax (3)

- **Memory operands with explicit segment register**
  - *segment_register*:*displacement*(*base*, *index*, *scale*)
  - Segmentation is exploited only infrequently in modern OSes
    - Cannot be turned off
    - Mostly used for thread-local storage, current thread pointer in kernel, etc.
      - `movl %gs:0x10, %eax`
  - When accessing memory, the segment base is always applied (added) to the effective address
    - Also in the case of implicit segment registers

Common IA-32 Instructions

- **Real programs tend to use a limited set of instructions most of the time**
  - NOP, MOV, LEA
  - ADD, SUB, INC, DEC
  - XOR, AND, OR
  - PUSH, POP, CALL, RET
  - CMP, TEST
  - JMP, JE, JNE, JL, JB, JG, JA

Common IA-32 Instructions (2)

- **NOP**
  - Single byte instruction, opcode 0x90
    - There are multibyte variants using prefixes
  - No operation (actually XCHG EAX, EAX)
  - Important role for optimization and debugging
- **MOV**
  - Move between registers
  - Memory loads and stores

Common IA-32 Instructions (3)

- **LEA**
  - Evaluate effective address of a memory operand
  - Frequently used as a "fused multiply-add" operation
    - *EA = displacement + base + index * scale*

```
leal (%edx, %edx, 8), %eax
```

EAX ← *EDX* + 8 * *EDX* (= 9 * *EDX*)

Common IA-32 Instructions (4)

- **ADD, SUB, XOR, AND, OR**
  - Addition, subtraction, logical exclusive OR, logical AND, logical OR

    ```
    xorl %ebx, %ebx
    ```
- **INC, DEC**
  - Increment, decrement
  - Only one operand

    ```
    incb %al
    ```

Common IA-32 Instructions (5)

- **PUSH**
  - Store register value on stack

```
pushl %ecx
```

ESP ← *ESP* - 4

(*ESP*) ← *ECX*

- **POP**
  - Restore register value from stack

```
popl %edx
```

EDX ← (*ESP*)

ESP ← *ESP* + 4

Common IA-32 Instructions (6)

- **CALL**
  - Call subroutine (function)

```
call -0x8da0
```

ESP ← *ESP* - 4

(*ESP*) ← *EIP* + instr_size

EIP ← *EIP* - 0x8da0

- **RET**
  - Return from subroutine (function) call

```
ret
```

ESP ← *ESP* + 4

EIP ← (*ESP* - 4)

Common IA-32 Instructions (7)

- **CMP**
  - Compare two operands
    - Like SUB, but the result is discarded
    - Modifies bits in the *EFLAGS* register

      ```
      cmpb $0x2f, (%esi)
      ```
- **TEST**
  - Test bits
    - Like AND, but result is discarded and *EFLAGS* is modified

      ```
      test %eax, %eax
      ```

Common IA-32 Instructions (8)

- **JMP**
  - Unconditional jump
  - Several variants
    - Relative address as immediate operand
    - Long jump
- **JE, JNE, JL, JB, JA, ...**
  - Conditional branches (many mnemonics)
  - Condition: State of bits in *EFLAGS*
  - Relative address as immediate operand (±128 B)

AMD64 Instruction Set

- **Instructions have mostly the same syntax as in IA-32**
- **Notable differences**
  - New registers, new aliases, *RFLAGS*
  - New operand size *q* (quad, 64 bits)
  - 32bit operand instructions affect the upper 32 bits of registers
  - Effective address can use *RIP* as a base

```
cmpq +0x305f9e(%rip),%r13
```

IA-32 Function Prologue

```
pushl %ebp
movl %esp, %ebp
subl $imm, %esp
movl %ebx, 4(%esp)
pushl %edi
…
```

IA-32 Function Epilogue

```
…
popl %edi
movl 4(%esp), %ebx
movl %ebp, %esp
popl %ebp
ret
```

```
…
popl %edi
movl 4(%esp),%ebx
leave
ret
```

AMD64 Function Prologue

```
pushq %rbp
movq %rsp, %rbp
subq $imm, %rsp
movq %rdi, -8(%rbp)   # save the first argument on stack
pushq %r12            # save the preserved register
...
```

AMD64 Function Prologue (2)

- **Sometimes compilers generate code which saves arguments passed in registers into the stack frame**
  - Good for debugging
  - Bad for performance

```
gcc -msave-args
suncc -Wu,-save_args
```

AMD64 Function Epilogue

```
…
popq %r13
movq 8(%rsp), %r12
movq %rbp, %rsp
popq %rbp
ret
```

```
…
popq %r13
movq 8(%rsp),%r12
leave
ret
```

IA-32 Stack and Code Example

- **Recall: Functions `a()`, `b()`, `c()` from *Basics***
  - Compile using `gcc -O1` for IA-32
  - Disassemble and single-step `main()` and `a()`
  - Observe the stack

IA-32 Stack and Code Example (2)

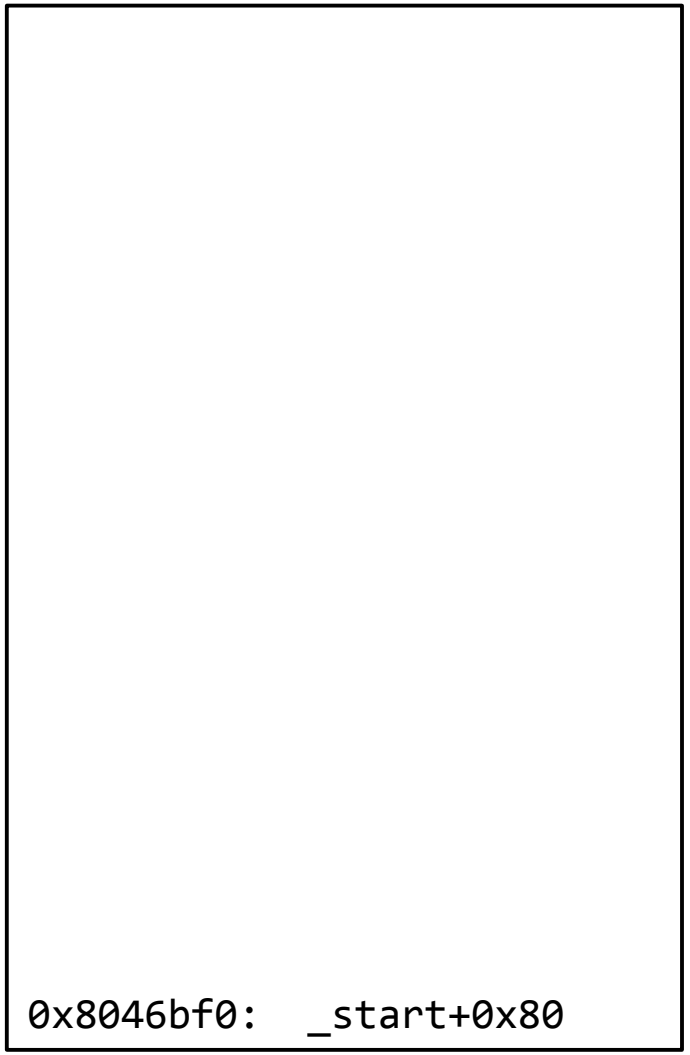
```
a:          pushl  %ebp
a+1:        movl   %esp,%ebp
a+3:        subl   $0x14,%esp
a+6:        pushl  0x8(%ebp)
a+9:        call   +0x5    <b>
a+0xe:      addl   $0x10,%esp
a+0x11:     leave
a+0x12:     ret
```

```
main:       pushl  %ebp
main+1:     movl   %esp,%ebp
main+3:     subl   $0x8,%esp
main+6:     andl   $0xfffffff0,%esp
main+9:     subl   $0x1c,%esp
main+0xc:   pushl  0x8(%ebp)
main+0xf:   call   -0x3f    <a>
main+0x14:  leave
main+0x15:  ret
```

IA-32 Stack and Code Example (2)

- **Initial state**
  - No instructions executed
  - Inherited stack pointer from `main()`'s caller

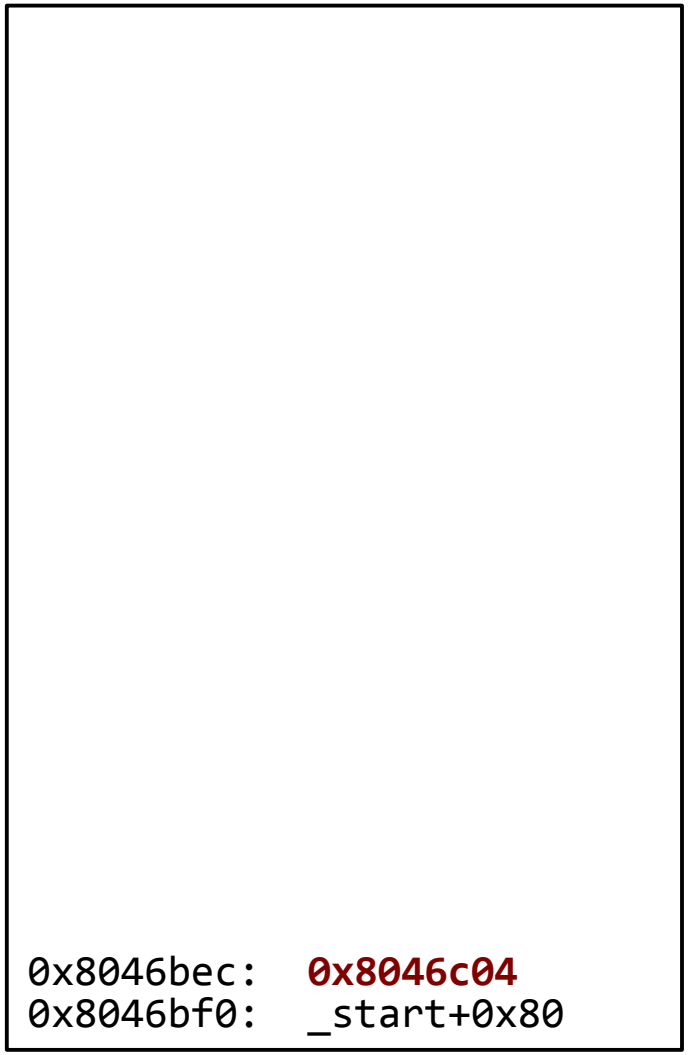
```
main:       pushl  %ebp
main+1:     movl   %esp,%ebp
main+3:     subl   $0x8,%esp
main+6:     andl   $0xfffffff0,%esp
main+9:     subl   $0x1c,%esp
main+0xc:   pushl  0x8(%ebp)
main+0xf:   call   -0x3f    <a>
main+0x14: leave
main+0x15: ret
```



IA-32 Stack and Code Example (2)

- **Save previous frame pointer on the stack**

```
main:      pushl  %ebp
main+1:    movl   %esp,%ebp
main+3:    subl   $0x8,%esp
main+6:    andl   $0xfffffff0,%esp
main+9:    subl   $0x1c,%esp
main+0xc:  pushl  0x8(%ebp)
main+0xf:  call   -0x3f    <a>
main+0x14: leave
main+0x15: ret
```



IA-32 Stack and Code Example (2)

- **Establish new fixed frame pointer in EBP**
  - It points to where we saved the previous one

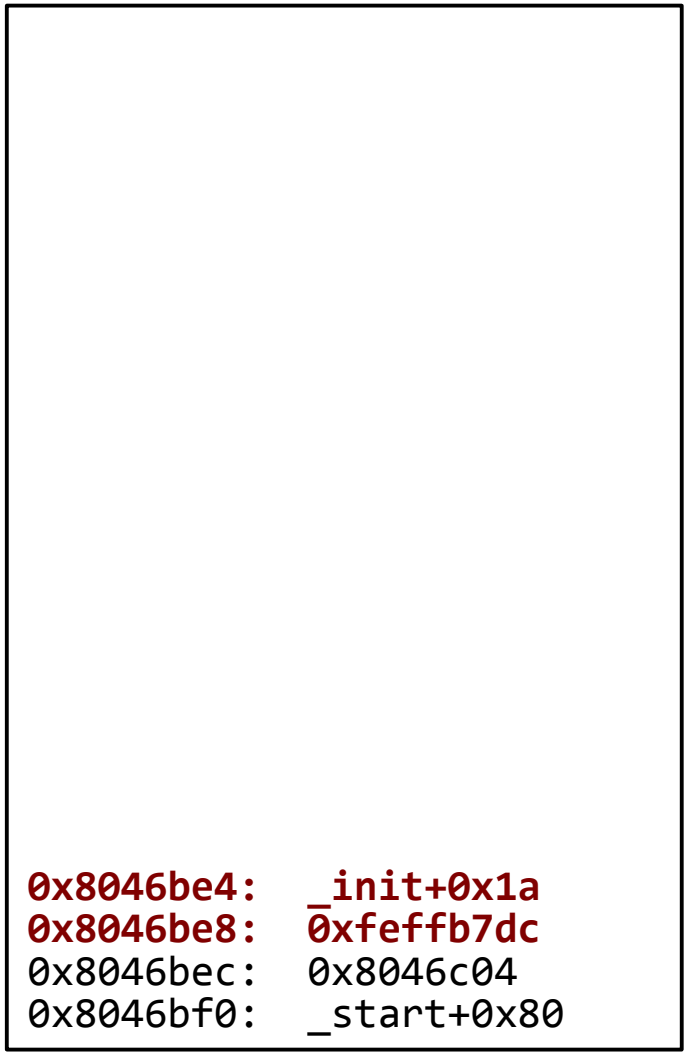
```
main:      pushl  %ebp
main+1:    movl   %esp,%ebp
main+3:    subl   $0x8,%esp
main+6:    andl   $0xfffffff0,%esp
main+9:    subl   $0x1c,%esp
main+0xc:  pushl  0x8(%ebp)
main+0xf:  call   -0x3f    <a>
main+0x14: leave
main+0x15: ret
```

```
0x8046bec:   0x8046c04
0x8046bf0:   _start+0x80
```

IA-32 Stack and Code Example (2)

- **Allocate some space on the stack**
  - Will not be used

```
main:      pushl  %ebp
main+1:    movl   %esp,%ebp
main+3:    subl   $0x8,%esp
main+6:    andl   $0xfffffff0,%esp
main+9:    subl   $0x1c,%esp
main+0xc:  pushl  0x8(%ebp)
main+0xf:  call   -0x3f    <a>
main+0x14: leave
main+0x15: ret
```



IA-32 Stack and Code Example (2)

- **Align the stack pointer on 16B boudary**
  - Not required by the ABI
  - Performance reasons

```
main:      pushl  %ebp
main+1:    movl   %esp,%ebp
main+3:    subl   $0x8,%esp
main+6:    andl   $0xfffffff0,%esp
main+9:    subl   $0x1c,%esp
main+0xc:  pushl  0x8(%ebp)
main+0xf:  call   -0x3f    <a>
main+0x14: leave
main+0x15: ret
```

```
0x8046be0:  0x8046bec
0x8046be4:  _init+0x1a
0x8046be8:  0xfeffb7dc
0x8046bec:  0x8046c04
0x8046bf0:  _start+0x80
```

IA-32 Stack and Code Example (2)

- **Allocate some more space on the stack**
  - Will not be used

```
main:      pushl  %ebp
main+1:    movl   %esp,%ebp
main+3:    subl   $0x8,%esp
main+6:    andl   $0xfffffff0,%esp
main+9:    subl   $0x1c,%esp
main+0xc:  pushl  0x8(%ebp)
main+0xf:  call   -0x3f    <a>
main+0x14: leave
main+0x15: ret
```

```
0x8046bc4:  _fpstart+0x2c
0x8046bc8:  0x29
0x8046bcc:  _fp_hw
0x8046bd0:  0x133f
0x8046bd4:  0x8050cda
0x8046bd8:  0x8060d3c
0x8046bdc:  0x8046bcc
0x8046be0:  0x8046bec
0x8046be4:  _init+0x1a
0x8046be8:  0xfeffb7dc
0x8046bec:  0x8046c04
0x8046bf0:  _start+0x80
```

IA-32 Stack and Code Example (2)

- Copy the incoming argument (argc) to the outgoing argument (a)

```
main:      pushl  %ebp
main+1:    movl   %esp,%ebp
main+3:    subl   $0x8,%esp
main+6:    andl   $0xfffffff0,%esp
main+9:    subl   $0x1c,%esp
main+0xc:  pushl  0x8(%ebp)
main+0xf:  call   -0x3f    <a>
main+0x14: leave
main+0x15: ret
```

```
0x8046bc0:  1
0x8046bc4:  _fpstart+0x2c
0x8046bc8:  0x29
0x8046bcc:  _fp_hw
0x8046bd0:  0x133f
0x8046bd4:  0x8050cda
0x8046bd8:  0x8060d3c
0x8046bdc:  0x8046bcc
0x8046be0:  0x8046bec
0x8046be4:  _init+0x1a
0x8046be8:  0xfeffb7dc
0x8046bec:  0x8046c04
0x8046bf0:  _start+0x80
```

IA-32 Stack and Code Example (2)

- Call a()

```
main:      pushl  %ebp
main+1:    movl   %esp,%ebp
main+3:    subl   $0x8,%esp
main+6:    andl   $0xfffffff0,%esp
main+9:    subl   $0x1c,%esp
main+0xc:  pushl  0x8(%ebp)
main+0xf:  call   -0x3f    <a>
main+0x14: leave
main+0x15: ret
```

```
0x8046bbc:  main+0x14
0x8046bc0:  1
0x8046bc4:  _fpstart+0x2c
0x8046bc8:  0x29
0x8046bcc:  _fp_hw
0x8046bd0:  0x133f
0x8046bd4:  0x8050cda
0x8046bd8:  0x8060d3c
0x8046bdc:  0x8046bcc
0x8046be0:  0x8046bec
0x8046be4:  _init+0x1a
0x8046be8:  0xfeffb7dc
0x8046bec:  0x8046c04
0x8046bf0:  _start+0x80
```

IA-32 Stack and Code Example (2)

```
a:          pushl  %ebp
a+1:        movl   %esp,%ebp
a+3:        subl   $0x14,%esp
a+6:        pushl  0x8(%ebp)
a+9:        call   +0x5    <b>
a+0xe:      addl   $0x10,%esp
a+0x11:     leave
a+0x12:     ret
```

- **Save the previous frame pointer to the stack**

```
0x8046bb8:  0x8046bec
0x8046bbc:  main+0x14
0x8046bc0:  1
0x8046bc4:  _fpstart+0x2c
0x8046bc8:  0x29
0x8046bcc:  _fp_hw
0x8046bd0:  0x133f
0x8046bd4:  0x8050cda
0x8046bd8:  0x8060d3c
0x8046bdc:  0x8046bcc
0x8046be0:  0x8046bec
0x8046be4:  _init+0x1a
0x8046be8:  0xfeffb7dc
0x8046bec:  0x8046c04
0x8046bf0:  _start+0x80
```

IA-32 Stack and Code Example (2)

```
a:          pushl   %ebp
a+1:        movl    %esp,%ebp
a+3:        subl    $0x14,%esp
a+6:        pushl   0x8(%ebp)
a+9:        call    +0x5    <b>
a+0xe:      addl    $0x10,%esp
a+0x11:     leave
a+0x12:     ret
```

- **Establish new frame pointer in EBP**
  - It points to the address where the previous one is stored

```
0x8046bb8:   0x8046bec
0x8046bbc:   main+0x14
0x8046bc0:   1
0x8046bc4:   _fpstart+0x2c
0x8046bc8:   0x29
0x8046bcc:   _fp_hw
0x8046bd0:   0x133f
0x8046bd4:   0x8050cda
0x8046bd8:   0x8060d3c
0x8046bdc:   0x8046bcc
0x8046be0:   0x8046bec
0x8046be4:   _init+0x1a
0x8046be8:   0xfeffb7dc
0x8046bec:   0x8046c04
0x8046bf0:   _start+0x80
```

IA-32 Stack and Code Example (2)

```
a:          pushl   %ebp
a+1:        movl    %esp,%ebp
a+3:        subl    $0x14,%esp
a+6:        pushl   0x8(%ebp)
a+9:        call    +0x5    <b>
a+0xe:      addl    $0x10,%esp
a+0x11:     leave
a+0x12:     ret
```

- **Allocate some space on the stack**
  - Will not be used

```
0x8046ba4:  0x80509e8
0x8046ba8:  dbg_desc
0x8046bac:  8
0x8046bb0:  1
0x8046bb4:  0
0x8046bb8:  0x8046bec
0x8046bbc:  main+0x14
0x8046bc0:  1
0x8046bc4:  _fpstart+0x2c
0x8046bc8:  0x29
0x8046bcc:  _fp_hw
0x8046bd0:  0x133f
0x8046bd4:  0x8050cda
0x8046bd8:  0x8060d3c
0x8046bdc:  0x8046bcc
0x8046be0:  0x8046bec
0x8046be4:  _init+0x1a
0x8046be8:  0xfeffb7dc
0x8046bec:  0x8046c04
0x8046bf0:  _start+0x80
```

IA-32 Stack and Code Example (2)

```
a:          pushl   %ebp
a+1:        movl    %esp,%ebp
a+3:        subl    $0x14,%esp
a+6:        pushl   0x8(%ebp)
a+9:        call    +0x5    <b>
a+0xe:      addl    $0x10,%esp
a+0x11:     leave
a+0x12:     ret
```

- Copy the incoming argument of a() to the outgoing argument for b()

```
0x8046ba0:  1
0x8046ba4:  0x80509e8
0x8046ba8:  dbg_desc
0x8046bac:  8
0x8046bb0:  1
0x8046bb4:  0
0x8046bb8:  0x8046bec
0x8046bbc:  main+0x14
0x8046bc0:  1
0x8046bc4:  _fpstart+0x2c
0x8046bc8:  0x29
0x8046bcc:  _fp_hw
0x8046bd0:  0x133f
0x8046bd4:  0x8050cda
0x8046bd8:  0x8060d3c
0x8046bdc:  0x8046bcc
0x8046be0:  0x8046bec
0x8046be4:  _init+0x1a
0x8046be8:  0xfeffb7dc
0x8046bec:  0x8046c04
0x8046bf0:  _start+0x80
```

IA-32 Stack and Code Example (2)

```
a:          pushl   %ebp
a+1:        movl    %esp,%ebp
a+3:        subl    $0x14,%esp
a+6:        pushl   0x8(%ebp)
a+9:        call    +0x5    <b>
a+0xe:      addl    $0x10,%esp
a+0x11:     leave
a+0x12:     ret
```

- Call b()

```
0x8046b9c:  a+0xe
0x8046ba0:  1
0x8046ba4:  0x80509e8
0x8046ba8:  dbg_desc
0x8046bac:  8
0x8046bb0:  1
0x8046bb4:  0
0x8046bb8:  0x8046bec
0x8046bbc:  main+0x14
0x8046bc0:  1
0x8046bc4:  _fpstart+0x2c
0x8046bc8:  0x29
0x8046bcc:  _fp_hw
0x8046bd0:  0x133f
0x8046bd4:  0x8050cda
0x8046bd8:  0x8060d3c
0x8046bdc:  0x8046bcc
0x8046be0:  0x8046bec
0x8046be4:  _init+0x1a
0x8046be8:  0xfeffb7dc
0x8046bec:  0x8046c04
0x8046bf0:  _start+0x80
```

IA-32 Stack and Code Example (2)

```
a:          pushl   %ebp
a+1:        movl    %esp,%ebp
a+3:        subl    $0x14,%esp
a+6:        pushl   0x8(%ebp)
a+9:        call    +0x5    <b>
a+0xe:      addl    $0x10,%esp
a+0x11:     leave
a+0x12:     ret
```

```
0x8046ba0:  1
0x8046ba4:  0x80509e8
0x8046ba8:  dbg_desc
0x8046bac:  8
0x8046bb0:  1
0x8046bb4:  0
0x8046bb8:  0x8046bec
0x8046bbc:  main+0x14
0x8046bc0:  1
0x8046bc4:  _fpstart+0x2c
0x8046bc8:  0x29
0x8046bcc:  _fp_hw
0x8046bd0:  0x133f
0x8046bd4:  0x8050cda
0x8046bd8:  0x8060d3c
0x8046bdc:  0x8046bcc
0x8046be0:  0x8046bec
0x8046be4:  _init+0x1a
0x8046be8:  0xfeffb7dc
0x8046bec:  0x8046c04
0x8046bf0:  _start+0x80
```

- **Step through and return from b( )**
  - b( )'s return value is in EAX

IA-32 Stack and Code Example (2)

```
a:          pushl   %ebp
a+1:        movl    %esp,%ebp
a+3:        subl    $0x14,%esp
a+6:        pushl   0x8(%ebp)
a+9:        call    +0x5    <b>
a+0xe:      addl    $0x10,%esp
a+0x11:     leave
a+0x12:     ret
```

- Destroy a()'s stack frame

```
0x8046bbc:  main+0x14
0x8046bc0:  1
0x8046bc4:  _fpstart+0x2c
0x8046bc8:  0x29
0x8046bcc:  _fp_hw
0x8046bd0:  0x133f
0x8046bd4:  0x8050cda
0x8046bd8:  0x8060d3c
0x8046bdc:  0x8046bcc
0x8046be0:  0x8046bec
0x8046be4:  _init+0x1a
0x8046be8:  0xfeffb7dc
0x8046bec:  0x8046c04
0x8046bf0:  _start+0x80
```

IA-32 Stack and Code Example (2)

```
a:          pushl   %ebp
a+1:        movl    %esp,%ebp
a+3:        subl    $0x14,%esp
a+6:        pushl   0x8(%ebp)
a+9:        call    +0x5    <b>
a+0xe:      addl    $0x10,%esp
a+0x11:     leave
a+0x12:     ret
```

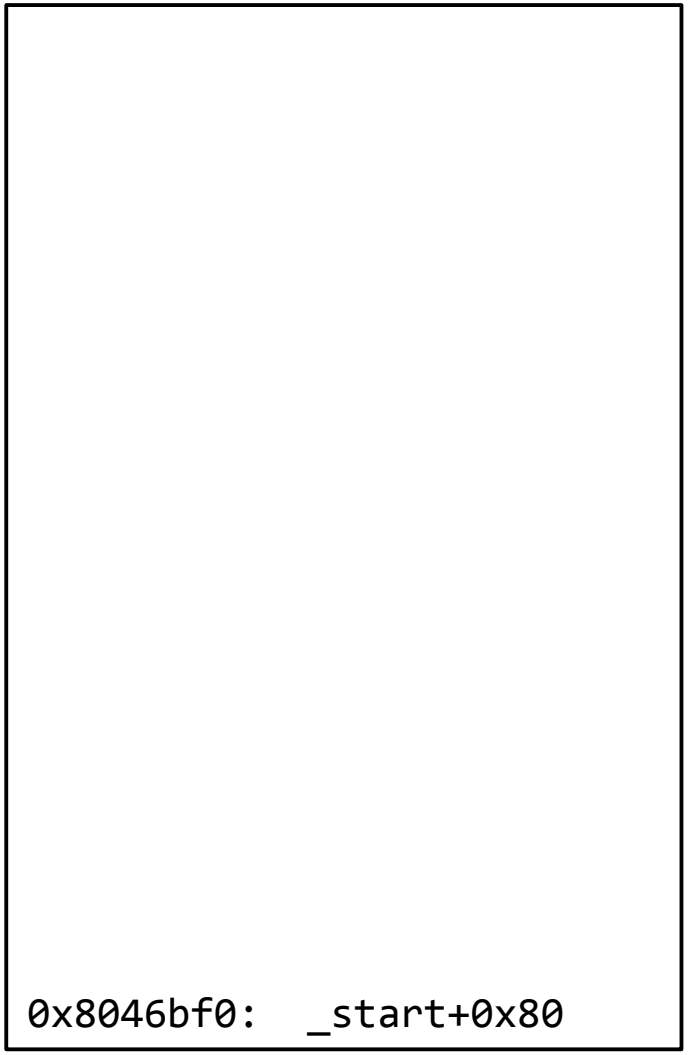
- **Return back to main( )**
  - Return value is again in EAX

```
0x8046bc0:  1
0x8046bc4:  _fpstart+0x2c
0x8046bc8:  0x29
0x8046bcc:  _fp_hw
0x8046bd0:  0x133f
0x8046bd4:  0x8050cda
0x8046bd8:  0x8060d3c
0x8046bdc:  0x8046bcc
0x8046be0:  0x8046bec
0x8046be4:  _init+0x1a
0x8046be8:  0xfeffb7dc
0x8046bec:  0x8046c04
0x8046bf0:  _start+0x80
```

IA-32 Stack and Code Example (2)

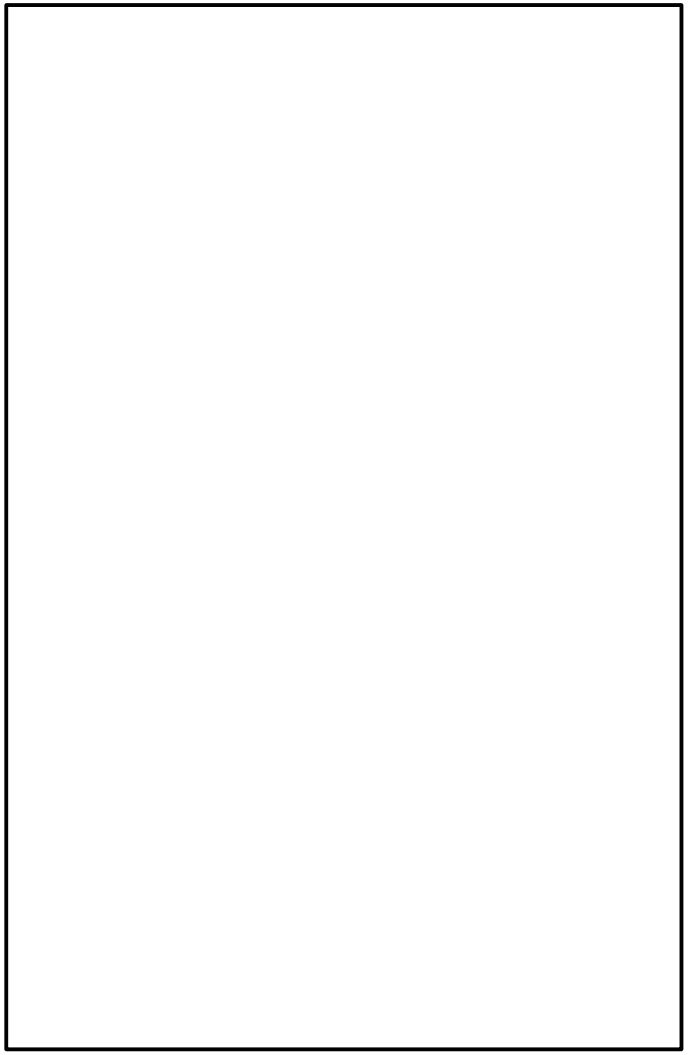
- Destroy `main()`'s stack frame

```
main:       pushl  %ebp
main+1:     movl   %esp,%ebp
main+3:     subl   $0x8,%esp
main+6:     andl   $0xfffffff0,%esp
main+9:     subl   $0x1c,%esp
main+0xc:   pushl  0x8(%ebp)
main+0xf:   call   -0x3f    <a>
main+0x14:  leave
main+0x15:  ret
```



IA-32 Stack and Code Example (2)

- Return from `main()`



```
main:      pushl  %ebp
main+1:    movl   %esp,%ebp
main+3:    subl   $0x8,%esp
main+6:    andl   $0xfffffff0,%esp
main+9:    subl   $0x1c,%esp
main+0xc:  pushl  0x8(%ebp)
main+0xf:  call   -0x3f    <a>
main+0x14: leave
main+0x15: ret
```

AMD64 Stack and Code Example

- **Recall: Functions `a()`, `b()`, `c()` from *Basics***
  - Compile using `gcc -O1 -m64` for AMD64
  - Disassemble and single-step `main()` and `a()`
  - Observe the stack

AMD64 Stack and Code Example (2)

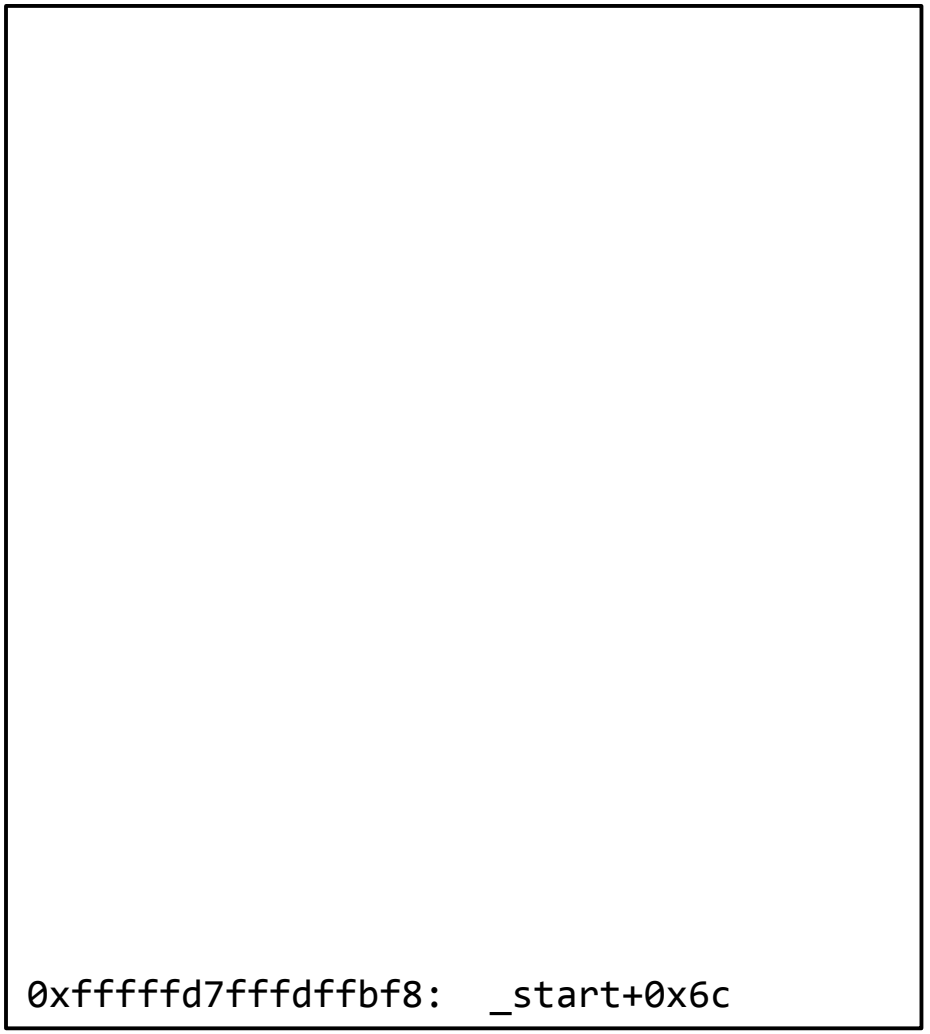
```
a:          pushq  %rbp
a+1:        movq   %rsp,%rbp
a+4:        movl   $0x0,%eax
a+9:        call   +0x2    <b>
a+0xe:      leave
a+0xf:      ret
```

```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     call   -0x2c    <a>
main+9:     leave
main+0xa:   ret
```

AMD64 Stack and Code Example (2)

- **Initial state**
  - No instructions executed
  - Inherited stack pointer from `main()`'s caller

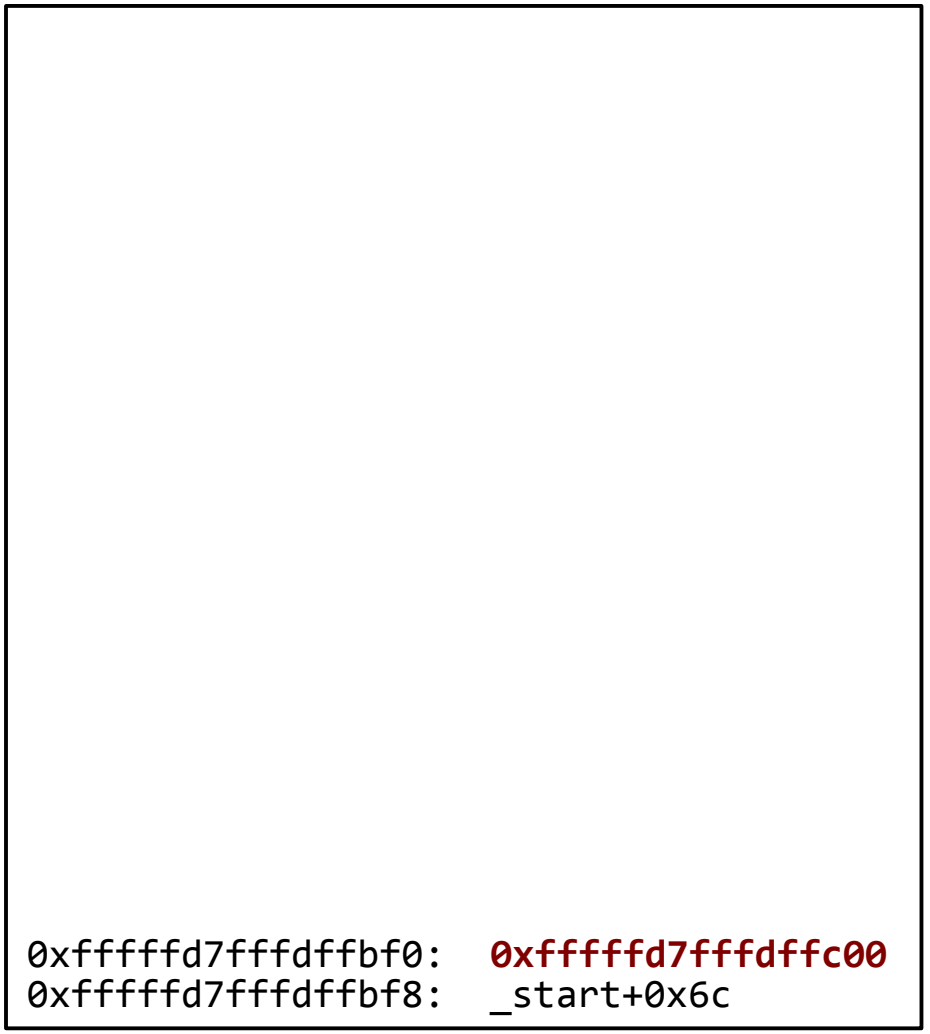
```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     call   -0x2c     <a>
main+9:     leave
main+0xa:   ret
```



AMD64 Stack and Code Example (2)

- **Save previous frame pointer on the stack**

```
main:      pushq  %rbp
main+1:    movq   %rsp,%rbp
main+4:    call   -0x2c     <a>
main+9:    leave
main+0xa:  ret
```



AMD64 Stack and Code Example (2)

- **Establish new fixed frame pointer in RBP**
  - It points to where we saved the previous one

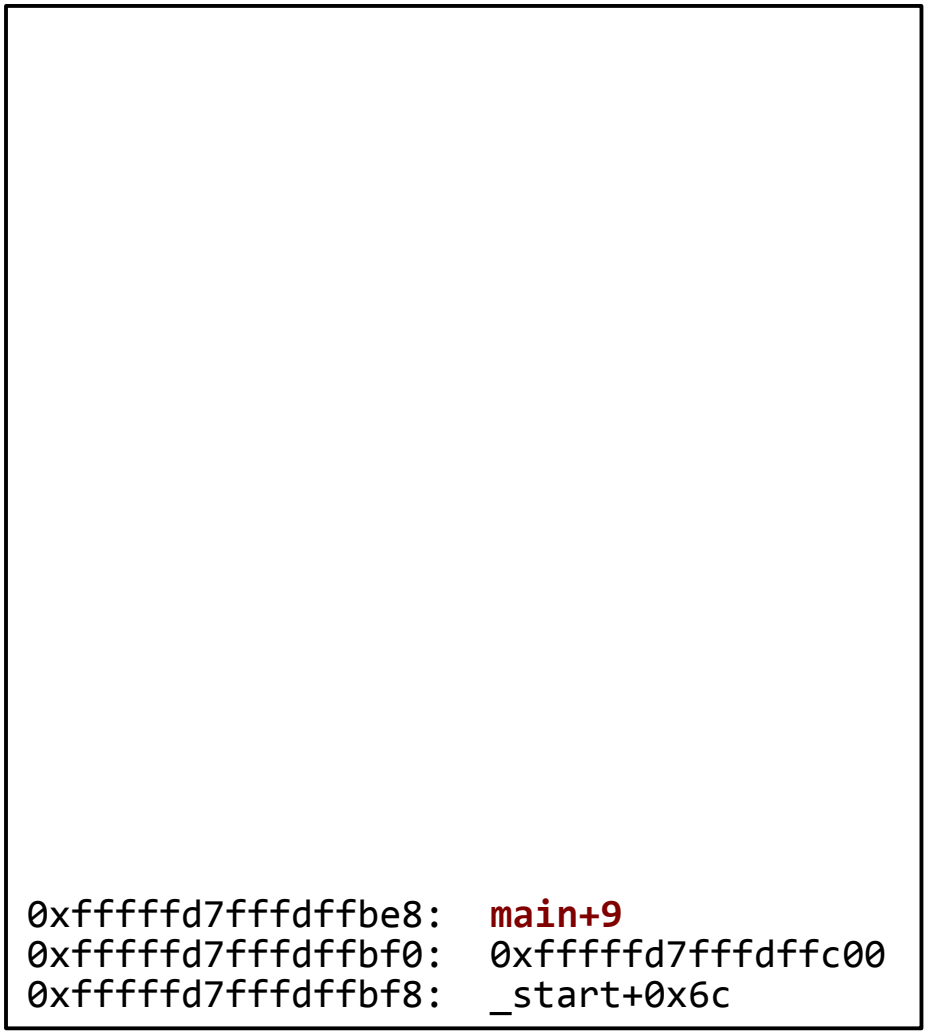
```
main:      pushq  %rbp
main+1:    movq   %rsp,%rbp
main+4:    call   -0x2c     <a>
main+9:    leave
main+0xa:  ret
```

```
0xffffffd7fffdffbf0:  0xffffffd7fffdffc00
0xffffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (2)

- **Call a()**
  - The argument is passed in RDI

```
main:      pushq  %rbp
main+1:    movq   %rsp,%rbp
main+4:    call   -0x2c     <a>
main+9:    leave
main+0xa:  ret
```



AMD64 Stack and Code Example (2)

```
a:          pushq   %rbp
a+1:        movq    %rsp,%rbp
a+4:        movl    $0x0,%eax
a+9:        call    +0x2    <b>
a+0xe:      leave
a+0xf:      ret
```

- Save the previous frame pointer to the stack

```
0xffffffd7fffdffbe0:   0xffffffd7fffdffbf0
0xffffffd7fffdffbe8:   main+9
0xffffffd7fffdffbf0:   0xffffffd7fffdffc00
0xffffffd7fffdffbf8:   _start+0x6c
```

AMD64 Stack and Code Example (2)

```
a:          pushq   %rbp
a+1:        movq    %rsp,%rbp
a+4:        movl    $0x0,%eax
a+9:        call    +0x2    <b>
a+0xe:      leave
a+0xf:      ret
```

- **Establish new frame pointer in RBP**
  - It points to the address where the previous one is stored

```
0xffffffd7fffdffbe0:   0xffffffd7fffdffbf0
0xffffffd7fffdffbe8:   main+9
0xffffffd7fffdffbf0:   0xffffffd7fffdffc00
0xffffffd7fffdffbf8:   _start+0x6c
```

AMD64 Stack and Code Example (2)

```
a:          pushq   %rbp
a+1:        movq    %rsp,%rbp
a+4:        movl    $0x0,%eax
a+9:        call    +0x2    <b>
a+0xe:      leave
a+0xf:      ret
```

- **Zero EAX**
  - Zero-extend to upper 32bits of RAX
    - Clears the whole RAX
  - Not needed

```
0xffffd7fffdffbd8:  a+0xe
0xffffd7fffdffbe0:  0xffffd7fffdffbf0
0xffffd7fffdffbe8:  main+9
0xffffd7fffdffbf0:  0xffffd7fffdffc00
0xffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (2)

```
a:          pushq   %rbp
a+1:        movq    %rsp,%rbp
a+4:        movl    $0x0,%eax
a+9:        call    +0x2    <b>
a+0xe:      leave
a+0xf:      ret
```

- Call b()
  - The argument is still in RDI

```
0xffffffd7fffdffbd8:   a+0xe
0xffffffd7fffdffbe0:   0xffffffd7fffdffbf0
0xffffffd7fffdffbe8:   main+9
0xffffffd7fffdffbf0:   0xffffffd7fffdffc00
0xffffffd7fffdffbf8:   _start+0x6c
```

AMD64 Stack and Code Example (2)

```
a:          pushq   %rbp
a+1:        movq    %rsp,%rbp
a+4:        movl    $0x0,%eax
a+9:        call    +0x2    <b>
a+0xe:      leave
a+0xf:      ret
```

- **Step through and return from b()**
  - b()'s return value is in RAX

```
0xffffffd7fffdffbe0:  0xffffffd7fffdffbf0
0xffffffd7fffdffbe8:  main+9
0xffffffd7fffdffbf0:  0xffffffd7fffdffc00
0xffffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (2)

```
a:          pushq   %rbp
a+1:        movq    %rsp,%rbp
a+4:        movl    $0x0,%eax
a+9:        call    +0x2    <b>
a+0xe:      leave
a+0xf:      ret
```

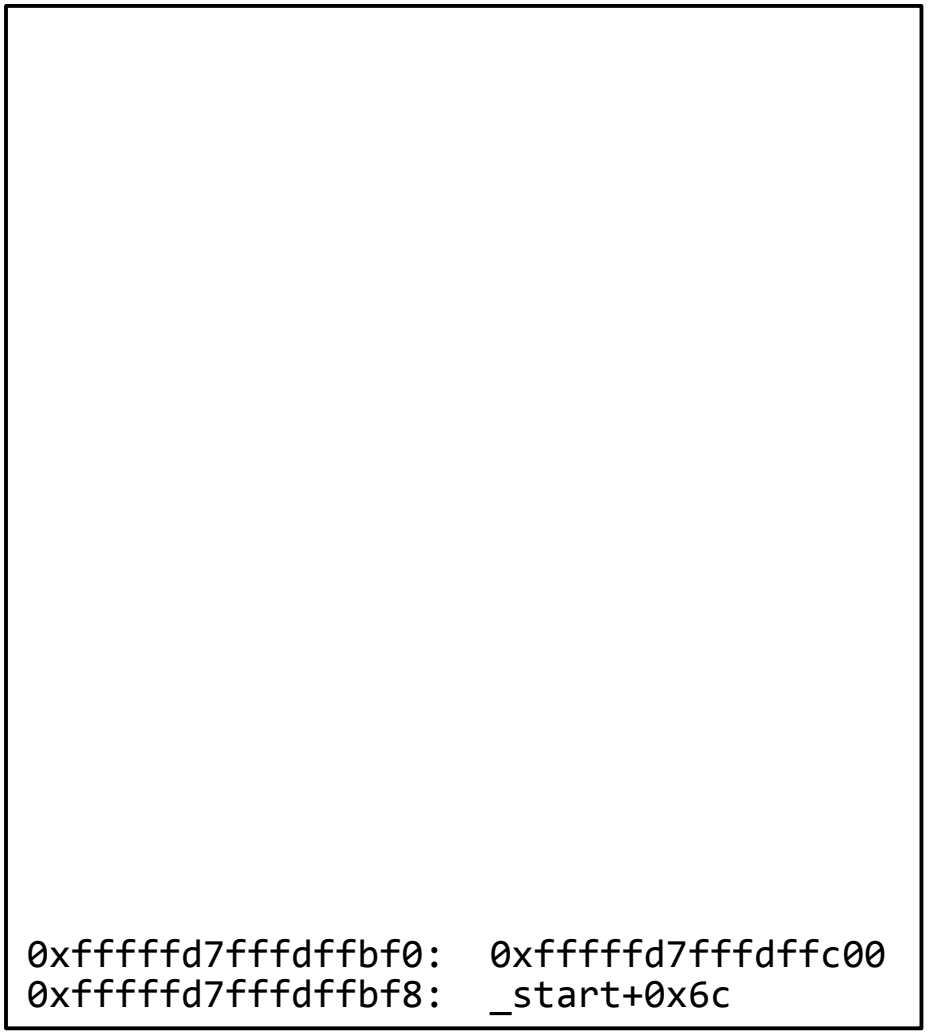
- Destroy a()'s stack frame

```
0xffffffd7fffdffbe8:  main+9
0xffffffd7fffdffbf0:  0xffffffd7fffdffc00
0xffffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (2)

```
a:          pushq   %rbp
a+1:        movq    %rsp,%rbp
a+4:        movl    $0x0,%eax
a+9:        call    +0x2    <b>
a+0xe:      leave
a+0xf:      ret
```

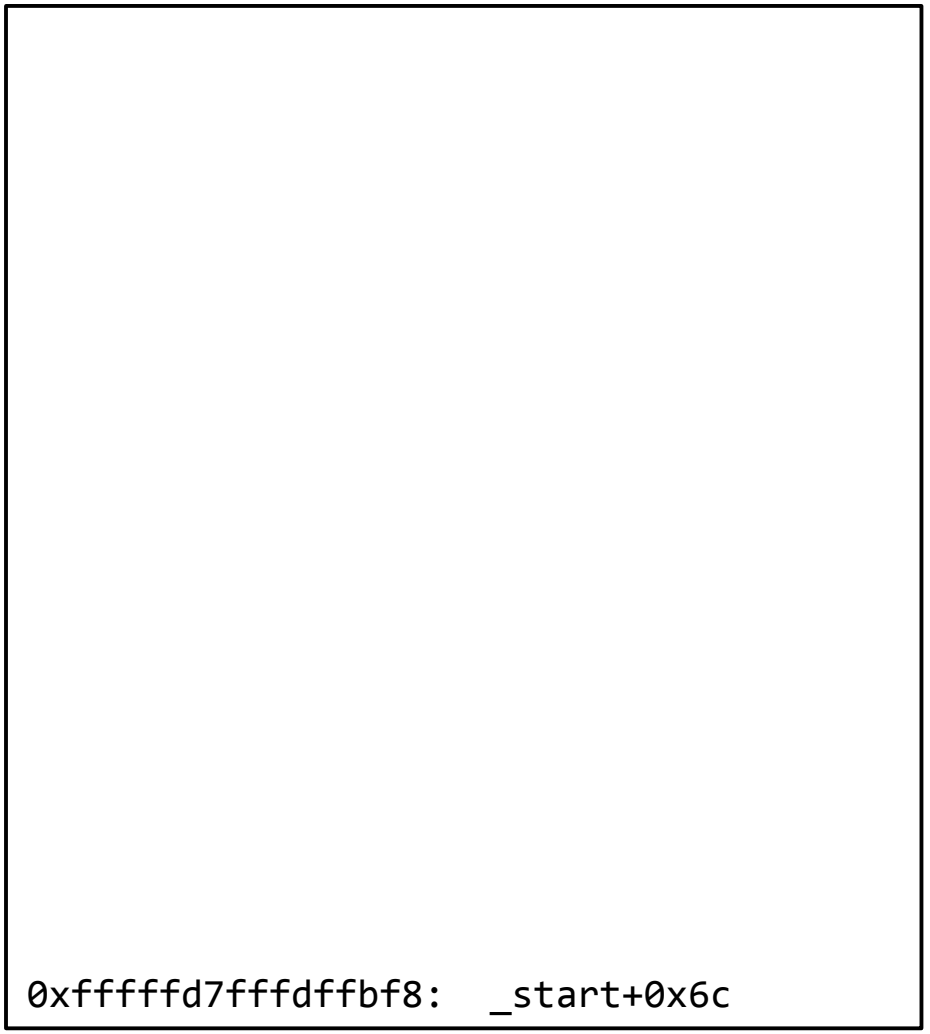
- **Return back to main()**
  - Return value is again in RAX



AMD64 Stack and Code Example (2)

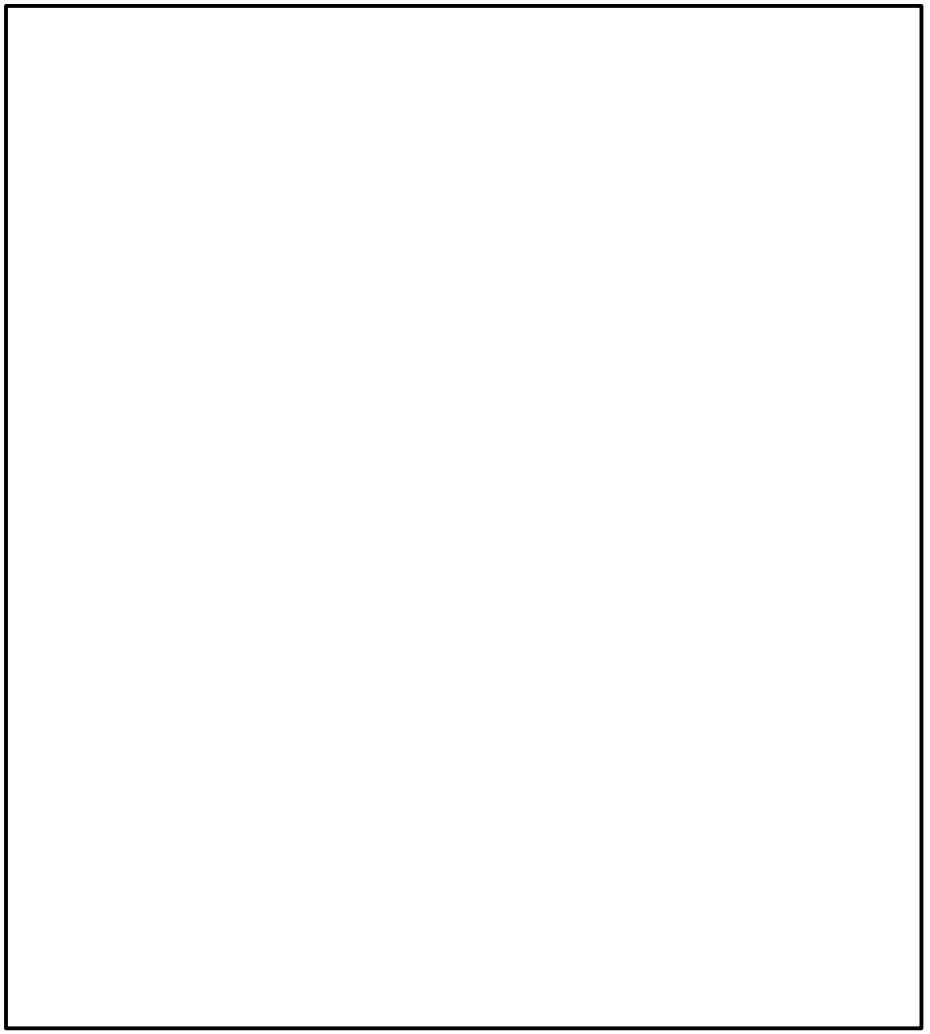
- Destroy `main()`'s stack frame

```
main:       pushq   %rbp
main+1:     movq    %rsp,%rbp
main+4:     call    -0x2c     <a>
main+9:     leave
main+0xa:   ret
```



AMD64 Stack and Code Example (2)

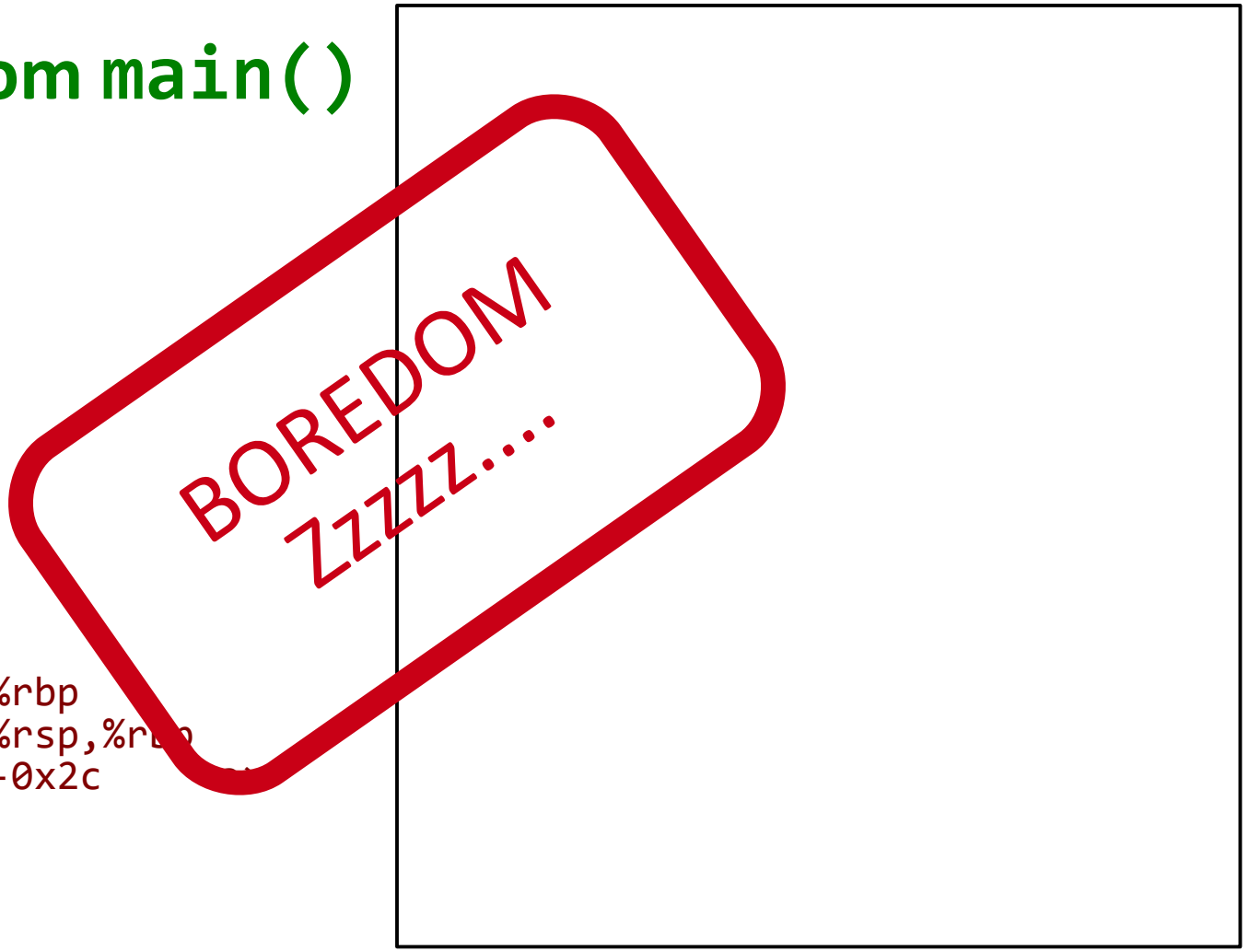
- Return from `main()`



```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     call   -0x2c     <a>
main+9:     leave
main+0xa:   ret
```

AMD64 Stack and Code Example (2)

- Return from `main()`




```
main:      pushq  %rbp
main+1:    movq   %rsp,%rbp
main+4:    call   -0x2c
main+9:    leave
main+0xa:  ret
```

AMD64 Stack and Code Example (3)

- **Let's try different compiler options**
  - Compile using `gcc -O0 -m64 -msave-args` for AMD64
  - Disassemble and single-step `main()` and `a()`
  - Observe the stack

AMD64 Stack and Code Example (4)

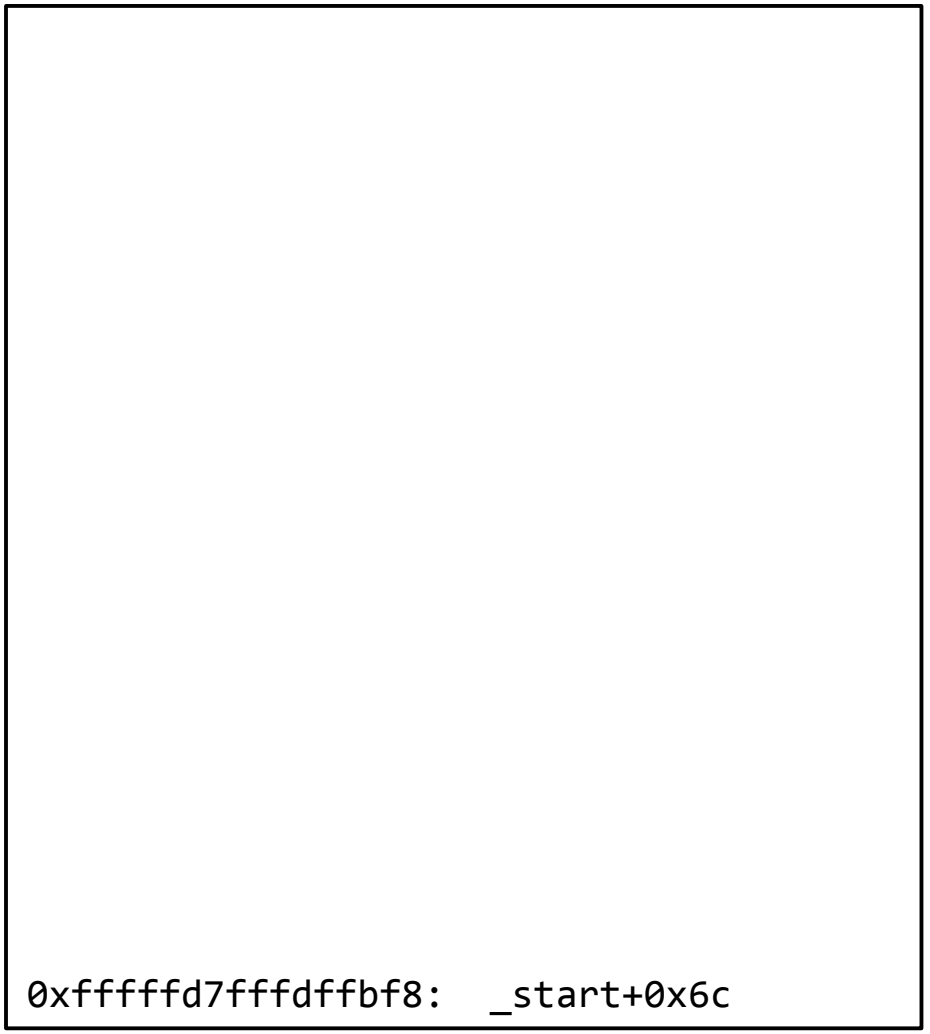
```
a:          pushq  %rbp
a+1:        movq   %rsp,%rbp
a+4:        movq   %rdi,-0x8(%rbp)
a+8:        subq   $0x20,%rsp
a+0xc:      movl   %edi,-0x14(%rbp)
a+0xf:      movl   -0x14(%rbp),%edi
a+0x12:     movl   $0x0,%eax
a+0x17:     call   +0x2    <b>
a+0x1c:     leave
a+0x1d:     ret
```

```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     movq   %rsi,-0x10(%rbp)
main+8:     movq   %rdi,-0x8(%rbp)
main+0xc:   subq   $0x20,%rsp
main+0x10:  movl   %edi,-0x14(%rbp)
main+0x13:  movq   %rsi,-0x20(%rbp)
main+0x17:  movl   -0x14(%rbp),%edi
main+0x1a:  call   -0x6b    <a>
main+0x1f:  leave
main+0x20:  ret
```

AMD64 Stack and Code Example (4)

- **Initial state**
  - No instructions executed
  - Inherited stack pointer from `main()`'s caller

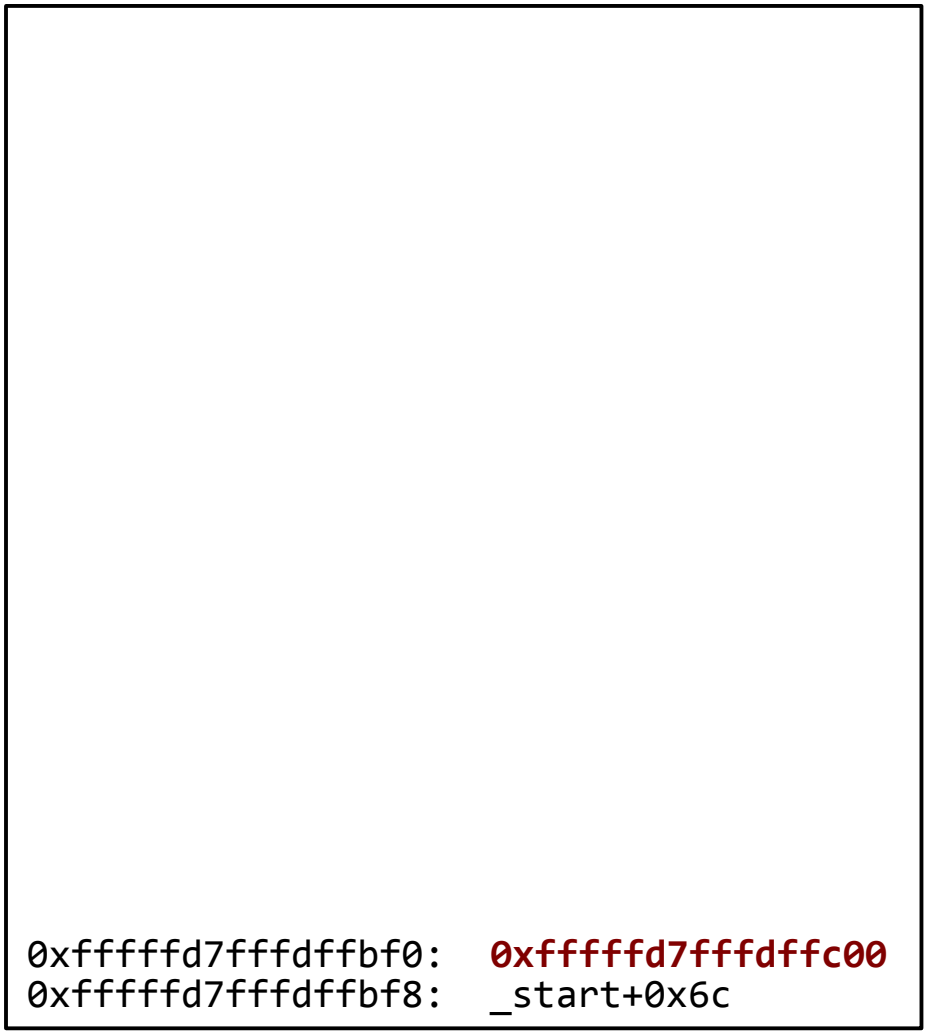
```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     movq   %rsi,-0x10(%rbp)
main+8:     movq   %rdi,-0x8(%rbp)
main+0xc:   subq   $0x20,%rsp
main+0x10:  movl   %edi,-0x14(%rbp)
main+0x13:  movq   %rsi,-0x20(%rbp)
main+0x17:  movl   -0x14(%rbp),%edi
main+0x1a:  call   -0x6b    <a>
main+0x1f:  leave
main+0x20:  ret
```



AMD64 Stack and Code Example (4)

- **Save previous frame pointer on the stack**

```
main:      pushq  %rbp
main+1:    movq   %rsp,%rbp
main+4:    movq   %rsi,-0x10(%rbp)
main+8:    movq   %rdi,-0x8(%rbp)
main+0xc:  subq   $0x20,%rsp
main+0x10: movl   %edi,-0x14(%rbp)
main+0x13: movq   %rsi,-0x20(%rbp)
main+0x17: movl   -0x14(%rbp),%edi
main+0x1a: call   -0x6b    <a>
main+0x1f: leave
main+0x20: ret
```



AMD64 Stack and Code Example (4)

- **Establish new fixed frame pointer in RBP**
  - It points to where we saved the previous one

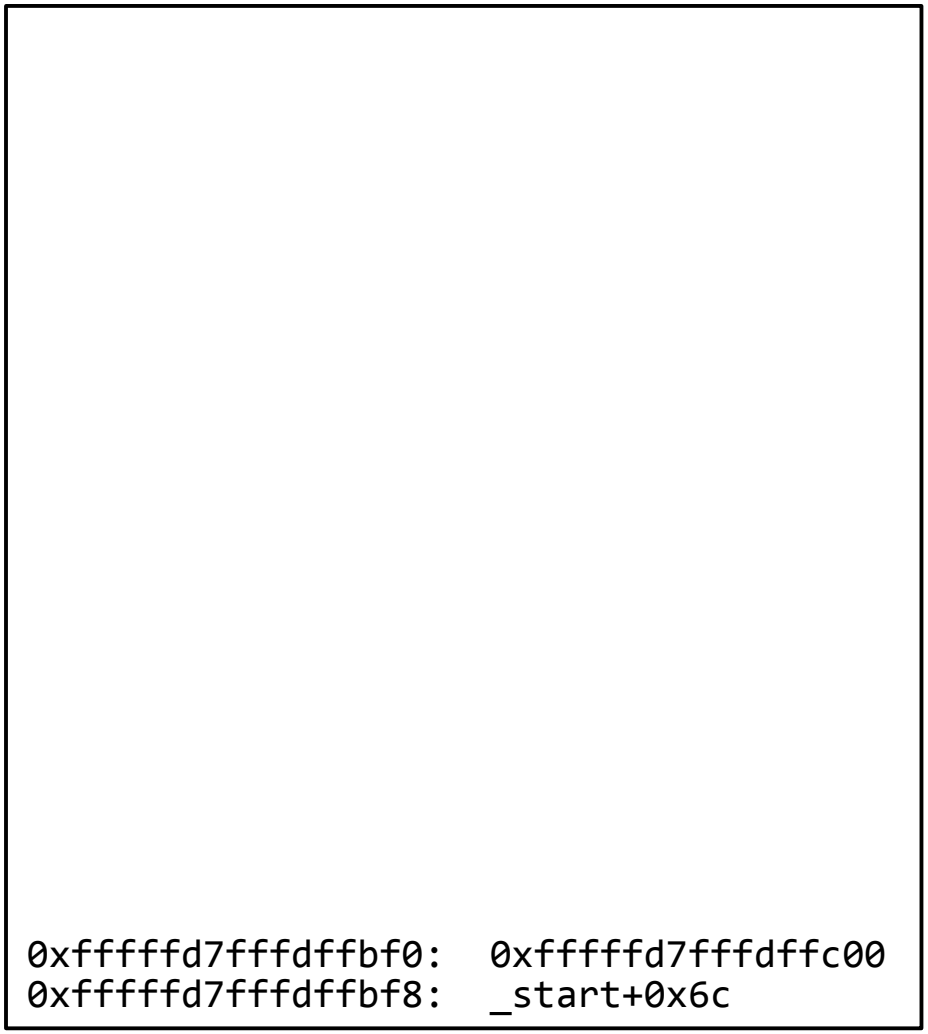
```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     movq   %rsi,-0x10(%rbp)
main+8:     movq   %rdi,-0x8(%rbp)
main+0xc:   subq   $0x20,%rsp
main+0x10:  movl   %edi,-0x14(%rbp)
main+0x13:  movq   %rsi,-0x20(%rbp)
main+0x17:  movl   -0x14(%rbp),%edi
main+0x1a:  call   -0x6b    <a>
main+0x1f:  leave
main+0x20:  ret
```

```
0xffffffd7fffdffbf0:  0xffffffd7fffdffc00
0xffffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (4)

- **Save the second argument on the stack**
  - Using the red zone

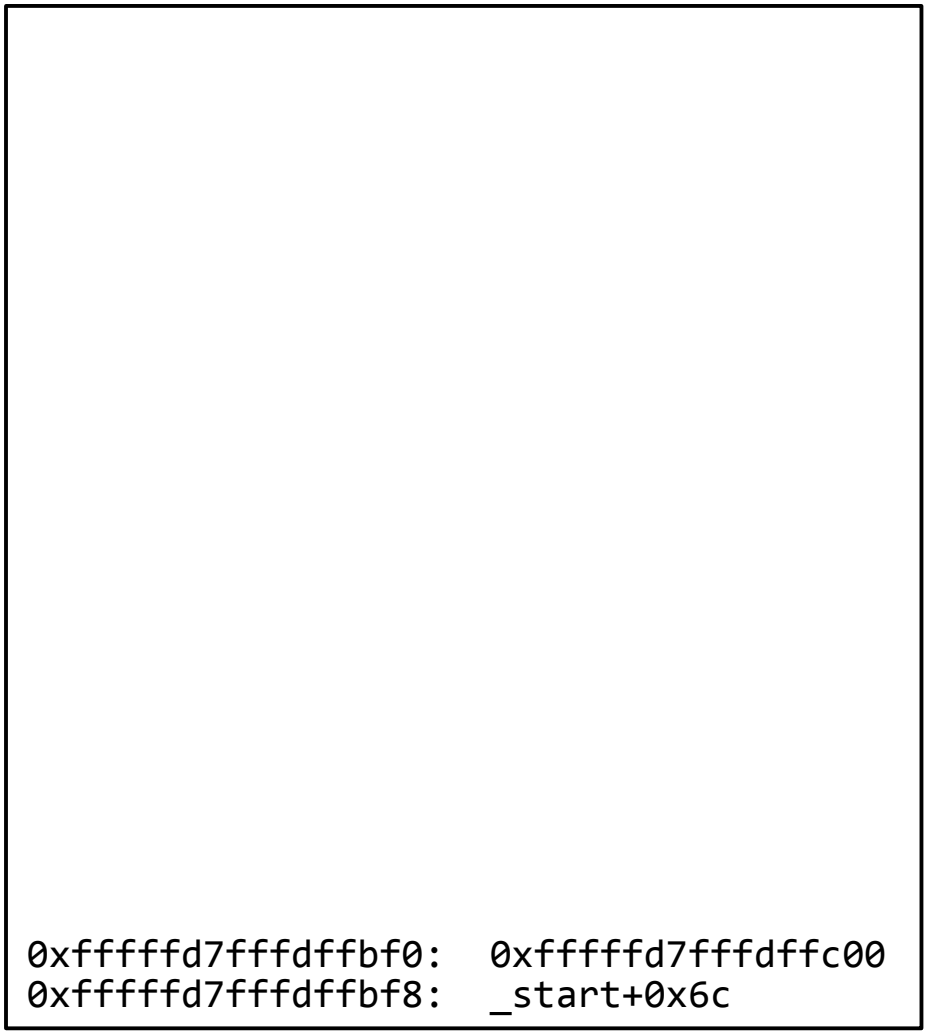
```
main:      pushq  %rbp
main+1:    movq   %rsp,%rbp
main+4:    movq   %rsi,-0x10(%rbp)
main+8:    movq   %rdi,-0x8(%rbp)
main+0xc:  subq   $0x20,%rsp
main+0x10: movl   %edi,-0x14(%rbp)
main+0x13: movq   %rsi,-0x20(%rbp)
main+0x17: movl   -0x14(%rbp),%edi
main+0x1a: call   -0x6b    <a>
main+0x1f: leave
main+0x20: ret
```



AMD64 Stack and Code Example (4)

- **Save the first argument on the stack**
  - Using the red zone

```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     movq   %rsi,-0x10(%rbp)
main+8:     movq   %rdi,-0x8(%rbp)
main+0xc:   subq   $0x20,%rsp
main+0x10:  movl   %edi,-0x14(%rbp)
main+0x13:  movq   %rsi,-0x20(%rbp)
main+0x17:  movl   -0x14(%rbp),%edi
main+0x1a:  call   -0x6b    <a>
main+0x1f:  leave
main+0x20:  ret
```



AMD64 Stack and Code Example (4)

- **Allocate stack space**
  - We can see the arguments now

```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     movq   %rsi,-0x10(%rbp)
main+8:     movq   %rdi,-0x8(%rbp)
main+0xc:   subq   $0x20,%rsp
main+0x10:  movl   %edi,-0x14(%rbp)
main+0x13:  movq   %rsi,-0x20(%rbp)
main+0x17:  movl   -0x14(%rbp),%edi
main+0x1a:  call   -0x6b    <a>
main+0x1f:  leave
main+0x20:  ret
```

```
0xffffd7fffdffbd0:  0xffffd7fffdffc00
0xffffd7fffdffbd8:  _start+0x63
0xffffd7fffdffbe0:  0xffffd7fffdffc18
0xffffd7fffdffbe8:  1
0xffffd7fffdffbf0:  0xffffd7fffdffc00
0xffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (4)

- Save away the first argument once more

```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     movq   %rsi,-0x10(%rbp)
main+8:     movq   %rdi,-0x8(%rbp)
main+0xc:   subq   $0x20,%rsp
main+0x10:  movl   %edi,-0x14(%rbp)
main+0x13:  movq   %rsi,-0x20(%rbp)
main+0x17:  movl   -0x14(%rbp),%edi
main+0x1a:  call   -0x6b    <a>
main+0x1f:  leave
main+0x20:  ret
```

```
0xffffd7fffdffbd0:  0xffffd7fffdffc00
0xffffd7fffdffbd8:  0x100400eb3
0xffffd7fffdffbe0:  0xffffd7fffdffc18
0xffffd7fffdffbe8:  1
0xffffd7fffdffbf0:  0xffffd7fffdffc00
0xffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (4)

- **Save away the second argument once more**

```
main:      pushq  %rbp
main+1:    movq   %rsp,%rbp
main+4:    movq   %rsi,-0x10(%rbp)
main+8:    movq   %rdi,-0x8(%rbp)
main+0xc:  subq   $0x20,%rsp
main+0x10: movl   %edi,-0x14(%rbp)
main+0x13: movq   %rsi,-0x20(%rbp)
main+0x17: movl   -0x14(%rbp),%edi
main+0x1a: call   -0x6b    <a>
main+0x1f: leave
main+0x20: ret
```

```
0xffffffd7fffdffbd0:   0xffffffd7fffdffc18
0xffffffd7fffdffbd8:   0x100400eb3
0xffffffd7fffdffbe0:   0xffffffd7fffdffc18
0xffffffd7fffdffbe8:   1
0xffffffd7fffdffbf0:   0xffffffd7fffdffc00
0xffffffd7fffdffbf8:   _start+0x6c
```

AMD64 Stack and Code Example (4)

- Just for sure, read the first argument back to EDI

```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     movq   %rsi,-0x10(%rbp)
main+8:     movq   %rdi,-0x8(%rbp)
main+0xc:   subq   $0x20,%rsp
main+0x10:  movl   %edi,-0x14(%rbp)
main+0x13:  movq   %rsi,-0x20(%rbp)
main+0x17:  movl   -0x14(%rbp),%edi
main+0x1a:  call   -0x6b    <a>
main+0x1f:  leave
main+0x20:  ret
```

```
0xffffffd7fffdffbd0:   0xffffffd7fffdffc18
0xffffffd7fffdffbd8:   0x100400eb3
0xffffffd7fffdffbe0:   0xffffffd7fffdffc18
0xffffffd7fffdffbe8:   1
0xffffffd7fffdffbf0:   0xffffffd7fffdffc00
0xffffffd7fffdffbf8:   _start+0x6c
```

AMD64 Stack and Code Example (4)

- **Call a()**
  - The argument is passed in RDI

```
main:      pushq  %rbp
main+1:    movq   %rsp,%rbp
main+4:    movq   %rsi,-0x10(%rbp)
main+8:    movq   %rdi,-0x8(%rbp)
main+0xc:  subq   $0x20,%rsp
main+0x10: movl   %edi,-0x14(%rbp)
main+0x13: movq   %rsi,-0x20(%rbp)
main+0x17: movl   -0x14(%rbp),%edi
main+0x1a: call   -0x6b    <a>
main+0x1f: leave
main+0x20: ret
```

```
0xffffd7fffdffbc8:  main+0x1f
0xffffd7fffdffbd0:  0xffffd7fffdffc18
0xffffd7fffdffbd8:  0x100400eb3
0xffffd7fffdffbe0:  0xffffd7fffdffc18
0xffffd7fffdffbe8:  1
0xffffd7fffdffbf0:  0xffffd7fffdffc00
0xffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (4)

```
a:          pushq  %rbp
a+1:        movq   %rsp,%rbp
a+4:        movq   %rdi,-0x8(%rbp)
a+8:        subq   $0x20,%rsp
a+0xc:      movl   %edi,-0x14(%rbp)
a+0xf:      movl   -0x14(%rbp),%edi
a+0x12:     movl   $0x0,%eax
a+0x17:     call   +0x2    <b>
a+0x1c:     leave
a+0x1d:     ret
```

- **Save the previous frame pointer to the stack**

```
0xffffffd7fffdffbc0:   0xffffffd7fffdffbf0
0xffffffd7fffdffbc8:   main+0x1f
0xffffffd7fffdffbd0:   0xffffffd7fffdffc18
0xffffffd7fffdffbd8:   0x100400eb3
0xffffffd7fffdffbe0:   0xffffffd7fffdffc18
0xffffffd7fffdffbe8:   1
0xffffffd7fffdffbf0:   0xffffffd7fffdffc00
0xffffffd7fffdffbf8:   _start+0x6c
```

AMD64 Stack and Code Example (4)

```
a:          pushq  %rbp
a+1:        movq   %rsp,%rbp
a+4:        movq   %rdi,-0x8(%rbp)
a+8:        subq   $0x20,%rsp
a+0xc:      movl   %edi,-0x14(%rbp)
a+0xf:      movl   -0x14(%rbp),%edi
a+0x12:     movl   $0x0,%eax
a+0x17:     call   +0x2    <b>
a+0x1c:     leave
a+0x1d:     ret
```

- **Establish new frame pointer in RBP**
  - It points to the address where the previous one is stored

```
0xffffffd7fffdffbc0:   0xffffffd7fffdffbf0
0xffffffd7fffdffbc8:   main+0x1f
0xffffffd7fffdffbd0:   0xffffffd7fffdffc18
0xffffffd7fffdffbd8:   0x100400eb3
0xffffffd7fffdffbe0:   0xffffffd7fffdffc18
0xffffffd7fffdffbe8:   1
0xffffffd7fffdffbf0:   0xffffffd7fffdffc00
0xffffffd7fffdffbf8:   _start+0x6c
```

AMD64 Stack and Code Example (4)

```
a:          pushq   %rbp
a+1:        movq    %rsp,%rbp
a+4:        movq    %rdi,-0x8(%rbp)
a+8:        subq    $0x20,%rsp
a+0xc:      movl    %edi,-0x14(%rbp)
a+0xf:      movl    -0x14(%rbp),%edi
a+0x12:     movl    $0x0,%eax
a+0x17:     call    +0x2    <b>
a+0x1c:     leave
a+0x1d:     ret
```

- **Save the argument on the stack**
  - Using the red zone

```
0xffffffd7fffdffbc0:   0xffffffd7fffdffbf0
0xffffffd7fffdffbc8:   main+0x1f
0xffffffd7fffdffbd0:   0xffffffd7fffdffc18
0xffffffd7fffdffbd8:   0x100400eb3
0xffffffd7fffdffbe0:   0xffffffd7fffdffc18
0xffffffd7fffdffbe8:   1
0xffffffd7fffdffbf0:   0xffffffd7fffdffc00
0xffffffd7fffdffbf8:   _start+0x6c
```

AMD64 Stack and Code Example (4)

```
a:          pushq   %rbp
a+1:        movq    %rsp,%rbp
a+4:        movq    %rdi,-0x8(%rbp)
a+8:        subq    $0x20,%rsp
a+0xc:      movl    %edi,-0x14(%rbp)
a+0xf:      movl    -0x14(%rbp),%edi
a+0x12:     movl    $0x0,%eax
a+0x17:     call    +0x2    <b>
a+0x1c:     leave
a+0x1d:     ret
```

- **Allocate stack space**
  - We can see the argument now

```
0xffffd7fffdffba0:  0
0xffffd7fffdffba8:  0
0xffffd7fffdffbb0:  0
0xffffd7fffdffbb8:  1
0xffffd7fffdffbc0:  0xffffd7fffdffbf0
0xffffd7fffdffbc8:  main+0x1f
0xffffd7fffdffbd0:  0xffffd7fffdffc18
0xffffd7fffdffbd8:  0x100400eb3
0xffffd7fffdffbe0:  0xffffd7fffdffc18
0xffffd7fffdffbe8:  1
0xffffd7fffdffbf0:  0xffffd7fffdffc00
0xffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (4)

```
a:          pushq  %rbp
a+1:        movq   %rsp,%rbp
a+4:        movq   %rdi,-0x8(%rbp)
a+8:        subq   $0x20,%rsp
a+0xc:      movl   %edi,-0x14(%rbp)
a+0xf:      movl   -0x14(%rbp),%edi
a+0x12:     movl   $0x0,%eax
a+0x17:     call   +0x2    <b>
a+0x1c:     leave
a+0x1d:     ret
```

- **Save away the first argument once again**

```
0xffffd7fffdffba0:  0
0xffffd7fffdffba8:  0x100000000
0xffffd7fffdffbb0:  0
0xffffd7fffdffbb8:  1
0xffffd7fffdffbc0:  0xffffd7fffdffbf0
0xffffd7fffdffbc8:  main+0x1f
0xffffd7fffdffbd0:  0xffffd7fffdffc18
0xffffd7fffdffbd8:  0x100400eb3
0xffffd7fffdffbe0:  0xffffd7fffdffc18
0xffffd7fffdffbe8:  1
0xffffd7fffdffbf0:  0xffffd7fffdffc00
0xffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (4)

```
a:          pushq   %rbp
a+1:        movq    %rsp,%rbp
a+4:        movq    %rdi,-0x8(%rbp)
a+8:        subq    $0x20,%rsp
a+0xc:      movl    %edi,-0x14(%rbp)
a+0xf:      movl    -0x14(%rbp),%edi
a+0x12:     movl    $0x0,%eax
a+0x17:     call    +0x2    <b>
a+0x1c:     leave
a+0x1d:     ret
```

- **Just for sure, read the first argument back to EDI**

```
0xffffd7fffdffba0:  0
0xffffd7fffdffba8:  0x100000000
0xffffd7fffdffbb0:  0
0xffffd7fffdffbb8:  1
0xffffd7fffdffbc0:  0xffffd7fffdffbf0
0xffffd7fffdffbc8:  main+0x1f
0xffffd7fffdffbd0:  0xffffd7fffdffc18
0xffffd7fffdffbd8:  0x100400eb3
0xffffd7fffdffbe0:  0xffffd7fffdffc18
0xffffd7fffdffbe8:  1
0xffffd7fffdffbf0:  0xffffd7fffdffc00
0xffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (4)

```
a:          pushq  %rbp
a+1:        movq   %rsp,%rbp
a+4:        movq   %rdi,-0x8(%rbp)
a+8:        subq   $0x20,%rsp
a+0xc:      movl   %edi,-0x14(%rbp)
a+0xf:      movl   -0x14(%rbp),%edi
a+0x12:     movl   $0x0,%eax
a+0x17:     call   +0x2    <b>
a+0x1c:     leave
a+0x1d:     ret
```

- **Zero EAX**
  - Zero-extend to upper 32bits of RAX
    - Clears the whole RAX
  - Not needed

```
0xffffd7fffdffba0:  0
0xffffd7fffdffba8:  0x100000000
0xffffd7fffdffbb0:  0
0xffffd7fffdffbb8:  1
0xffffd7fffdffbc0:  0xffffd7fffdffbf0
0xffffd7fffdffbc8:  main+0x1f
0xffffd7fffdffbd0:  0xffffd7fffdffc18
0xffffd7fffdffbd8:  0x100400eb3
0xffffd7fffdffbe0:  0xffffd7fffdffc18
0xffffd7fffdffbe8:  1
0xffffd7fffdffbf0:  0xffffd7fffdffc00
0xffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (4)

```
a:          pushq   %rbp
a+1:        movq    %rsp,%rbp
a+4:        movq    %rdi,-0x8(%rbp)
a+8:        subq    $0x20,%rsp
a+0xc:      movl    %edi,-0x14(%rbp)
a+0xf:      movl    -0x14(%rbp),%edi
a+0x12:     movl    $0x0,%eax
a+0x17:     call    +0x2    <b>
a+0x1c:     leave
a+0x1d:     ret
```

- **Call b( )**
  - The argument is still in RDI

```
0xffffd7fffdffb98:  a+0x1c
0xffffd7fffdffba0:  0
0xffffd7fffdffba8:  0x100000000
0xffffd7fffdffbb0:  0
0xffffd7fffdffbb8:  1
0xffffd7fffdffbc0:  0xffffd7fffdffbf0
0xffffd7fffdffbc8:  main+0x1f
0xffffd7fffdffbd0:  0xffffd7fffdffc18
0xffffd7fffdffbd8:  0x100400eb3
0xffffd7fffdffbe0:  0xffffd7fffdffc18
0xffffd7fffdffbe8:  1
0xffffd7fffdffbf0:  0xffffd7fffdffc00
0xffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (4)

```
a:          pushq  %rbp
a+1:        movq   %rsp,%rbp
a+4:        movq   %rdi,-0x8(%rbp)
a+8:        subq   $0x20,%rsp
a+0xc:      movl   %edi,-0x14(%rbp)
a+0xf:      movl   -0x14(%rbp),%edi
a+0x12:     movl   $0x0,%eax
a+0x17:     call   +0x2    <b>
a+0x1c:     leave
a+0x1d:     ret
```

- **Step through and return from b( )**
  - b( )'s return value is in RAX

```
0xffffd7fffdffba0:  0
0xffffd7fffdffba8:  0x100000000
0xffffd7fffdffbb0:  0
0xffffd7fffdffbb8:  1
0xffffd7fffdffbc0:  0xffffd7fffdffbf0
0xffffd7fffdffbc8:  main+0x1f
0xffffd7fffdffbd0:  0xffffd7fffdffc18
0xffffd7fffdffbd8:  0x100400eb3
0xffffd7fffdffbe0:  0xffffd7fffdffc18
0xffffd7fffdffbe8:  1
0xffffd7fffdffbf0:  0xffffd7fffdffc00
0xffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (4)

```
a:          pushq   %rbp
a+1:        movq    %rsp,%rbp
a+4:        movq    %rdi,-0x8(%rbp)
a+8:        subq    $0x20,%rsp
a+0xc:      movl    %edi,-0x14(%rbp)
a+0xf:      movl    -0x14(%rbp),%edi
a+0x12:     movl    $0x0,%eax
a+0x17:     call    +0x2    <b>
a+0x1c:     leave
a+0x1d:     ret
```

- Destroy a( )'s stack frame

```
0xffffd7fffdffbc8:  main+0x1f
0xffffd7fffdffbd0:  0xffffd7fffdffc18
0xffffd7fffdffbd8:  0x100400eb3
0xffffd7fffdffbe0:  0xffffd7fffdffc18
0xffffd7fffdffbe8:  1
0xffffd7fffdffbf0:  0xffffd7fffdffc00
0xffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (4)

```
a:          pushq  %rbp
a+1:        movq   %rsp,%rbp
a+4:        movq   %rdi,-0x8(%rbp)
a+8:        subq   $0x20,%rsp
a+0xc:      movl   %edi,-0x14(%rbp)
a+0xf:      movl   -0x14(%rbp),%edi
a+0x12:     movl   $0x0,%eax
a+0x17:     call   +0x2    <b>
a+0x1c:     leave
a+0x1d:     ret
```

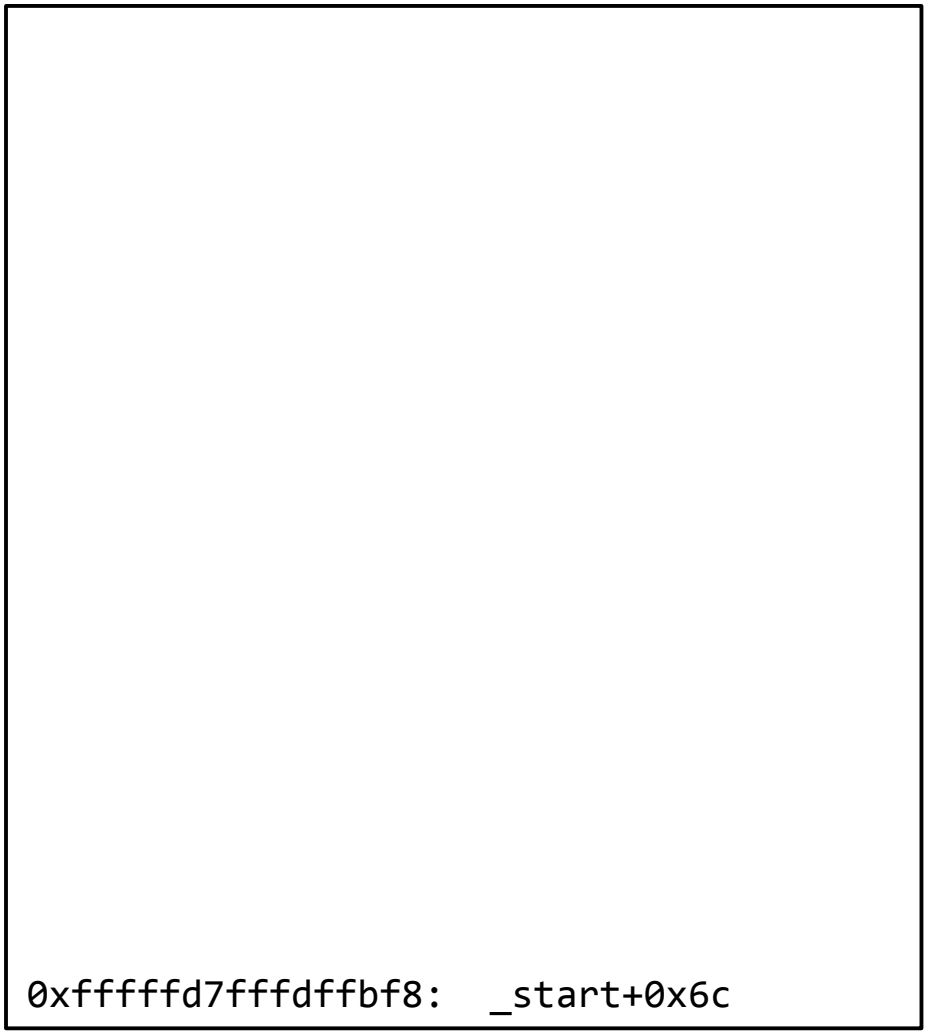
- **Return back to main()**
  - Return value is again in RAX

```
0xffffd7fffdffbd0:  0xffffd7fffdffc18
0xffffd7fffdffbd8:  0x100400eb3
0xffffd7fffdffbe0:  0xffffd7fffdffc18
0xffffd7fffdffbe8:  1
0xffffd7fffdffbf0:  0xffffd7fffdffc00
0xffffd7fffdffbf8:  _start+0x6c
```

AMD64 Stack and Code Example (4)

- Destroy `main()`'s stack frame

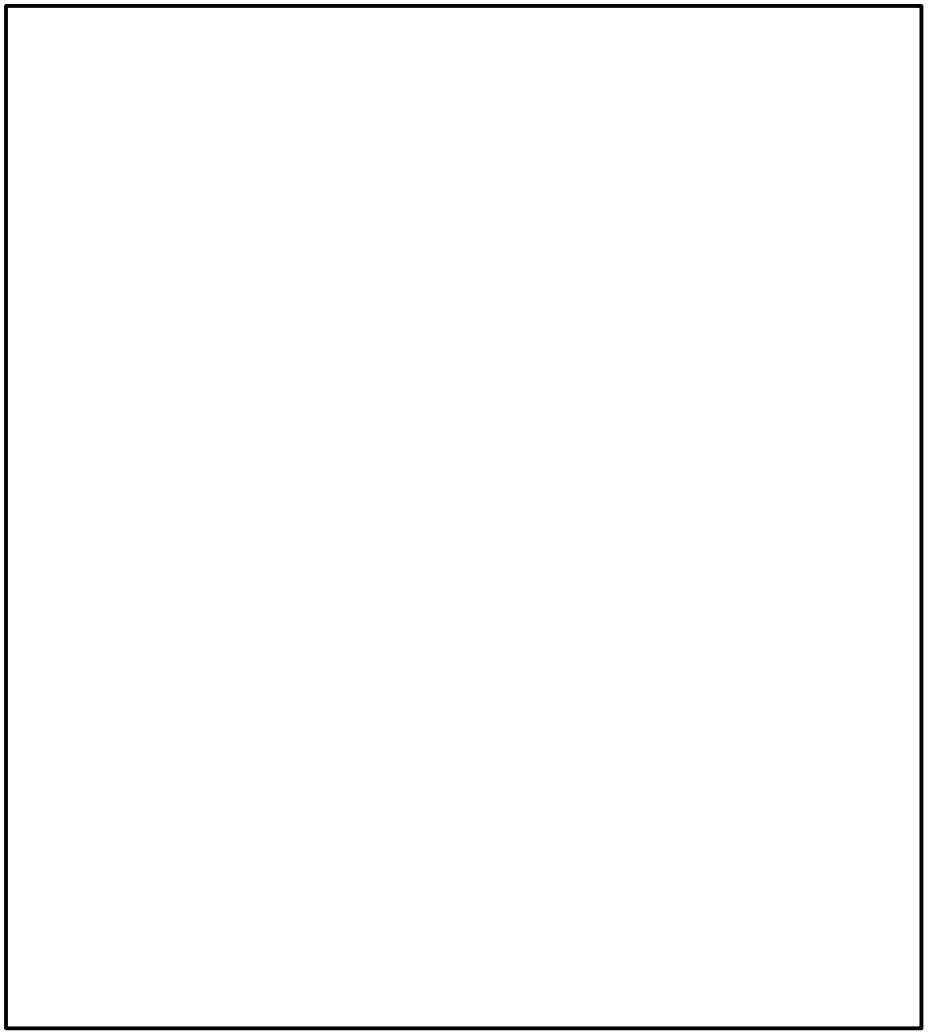
```
main:      pushq  %rbp
main+1:    movq   %rsp,%rbp
main+4:    movq   %rsi,-0x10(%rbp)
main+8:    movq   %rdi,-0x8(%rbp)
main+0xc:  subq   $0x20,%rsp
main+0x10: movl   %edi,-0x14(%rbp)
main+0x13: movq   %rsi,-0x20(%rbp)
main+0x17: movl   -0x14(%rbp),%edi
main+0x1a: call   -0x6b    <a>
main+0x1f: leave
main+0x20: ret
```



AMD64 Stack and Code Example (4)

- Return from `main()`

```
main:      pushq  %rbp
main+1:    movq   %rsp,%rbp
main+4:    movq   %rsi,-0x10(%rbp)
main+8:    movq   %rdi,-0x8(%rbp)
main+0xc:  subq   $0x20,%rsp
main+0x10: movl   %edi,-0x14(%rbp)
main+0x13: movq   %rsi,-0x20(%rbp)
main+0x17: movl   -0x14(%rbp),%edi
main+0x1a: call   -0x6b    <a>
main+0x1f: leave
main+0x20: ret
```



Compiling Without Frame Pointer

- **The frame pointer is actually not necessary**
  - Addressing variables by RSP/ESP is sufficient in many functions (unless e.g. `alloca()` is used)
    - The compilers are smart enough to track the offsets
    - The use of EBP/RBP is merely a convention
  - Extra general purpose register can be handy
    - So is saving one or two instructions on entry/exit
  - Downside: Obtaining stack traces and debugging is more difficult – no simple linked list traversal

AMD64 Stack and Code Example (5)

- **The compiler can omit frame pointers**
  - Actually the default behavior in GCC on AMD64 above `-O0`
    - And on IA-32 since GCC 4.6
    - Unless `-fno-omit-frame-pointer` is given
  - Example: Compile using `gcc -m64 -fomit-frame-pointer`
    - Using GCC 4.8.3 on Linux (more differences to previous examples)
      - Note: Using `alloca()` does result in frame pointer to be employed
    - Disassemble and single-step `main()` and `a()`
    - Observe the stack

AMD64 Stack and Code Example (6)

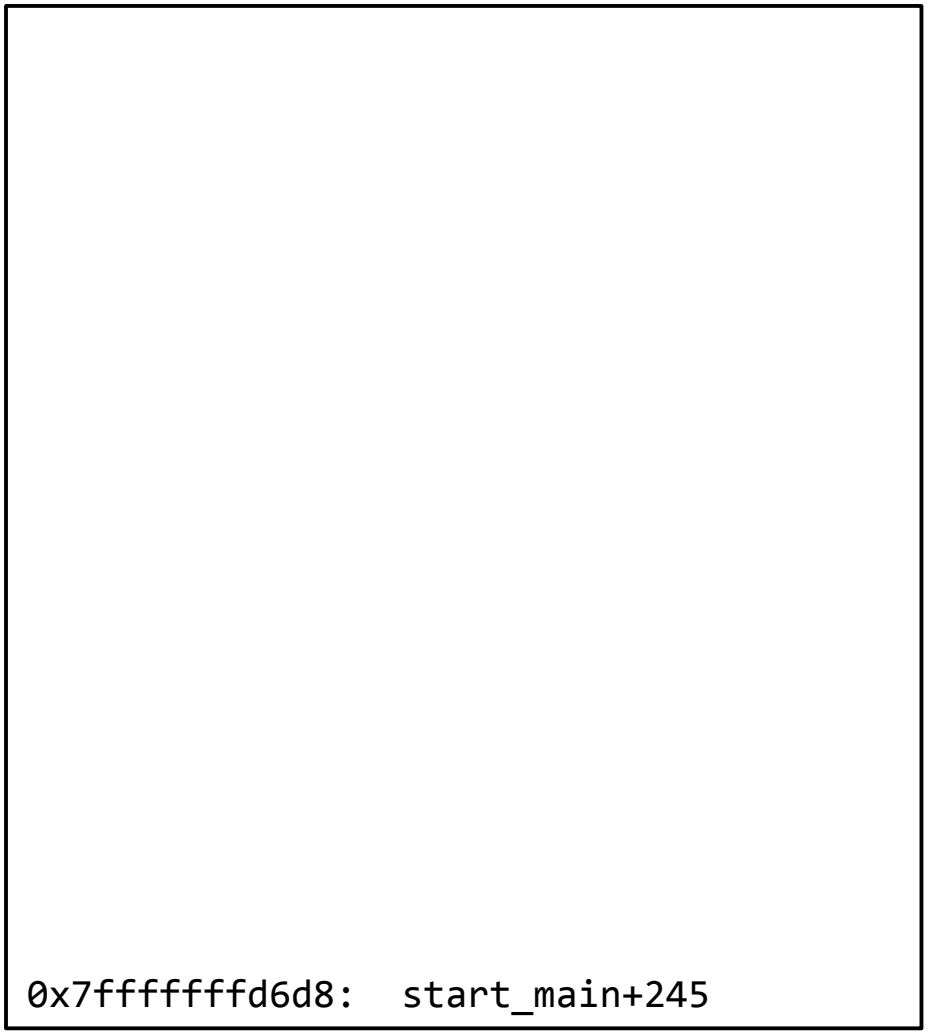
```
a+0:     sub    $0x8,%rsp
a+4:     mov    %edi,0x4(%rsp)
a+8:     mov    0x4(%rsp),%eax
a+12:    mov    %eax,%edi
a+14:    callq  0x400556    <b>
a+19:    add    $0x8,%rsp
a+23:    retq
```

```
main+0:    sub    $0x10,%rsp
main+4:    mov    %edi,0xc(%rsp)
main+8:    mov    %rsi,(%rsp)
main+12:   mov    0xc(%rsp),%eax
main+16:   mov    %eax,%edi
main+18:   callq  0x40056e    <a>
main+23:   add    $0x10,%rsp
main+27:   retq
```

AMD64 Stack and Code Example (6)

- **Initial state**
  - No instructions executed
  - Inherited stack pointer from `main()`'s caller

```
main+0:     sub     $0x10,%rsp
main+4:     mov     %edi,0xc(%rsp)
main+8:     mov     %rsi,(%rsp)
main+12:    mov     0xc(%rsp),%eax
main+16:    mov     %eax,%edi
main+18:    callq   0x40056e    <a>
main+23:    add     $0x10,%rsp
main+27:    retq
```



AMD64 Stack and Code Example (6)

- **Allocate stack space**
  - Only zeros so far

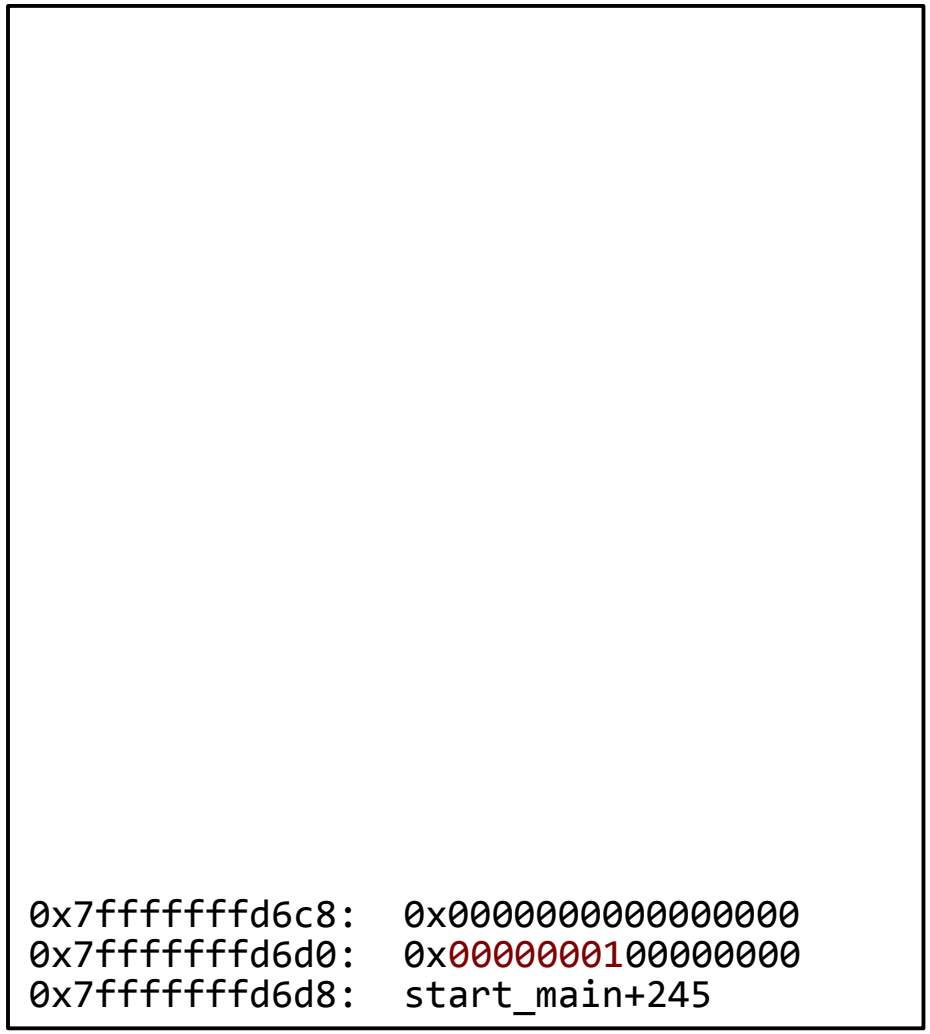
```
main+0:     sub     $0x10,%rsp
main+4:     mov     %edi,0xc(%rsp)
main+8:     mov     %rsi,(%rsp)
main+12:    mov     0xc(%rsp),%eax
main+16:    mov     %eax,%edi
main+18:    callq   0x40056e    <a>
main+23:    add     $0x10,%rsp
main+27:    retq
```

```
0x7fffffffd6c8:   0x0000000000000000
0x7fffffffd6d0:   0x0000000000000000
0x7fffffffd6d8:   start_main+245
```

AMD64 Stack and Code Example (6)

- **Save the first argument on the stack**
  - Not 8-bytes aligned

```
main+0:     sub     $0x10,%rsp
main+4:     mov     %edi,0xc(%rsp)
main+8:     mov     %rsi,(%rsp)
main+12:    mov     0xc(%rsp),%eax
main+16:    mov     %eax,%edi
main+18:    callq   0x40056e    <a>
main+23:    add     $0x10,%rsp
main+27:    retq
```



AMD64 Stack and Code Example (6)

- **Save the second argument on the stack**

```
main+0:     sub     $0x10,%rsp
main+4:     mov     %edi,0xc(%rsp)
main+8:     mov     %rsi,(%rsp)
main+12:    mov     0xc(%rsp),%eax
main+16:    mov     %eax,%edi
main+18:    callq   0x40056e    <a>
main+23:    add     $0x10,%rsp
main+27:    retq
```

AMD64 Stack and Code Example (6)

- **Read the saved first argument to EAX**
  - Because we can

```
main+0:     sub     $0x10,%rsp
main+4:     mov     %edi,0xc(%rsp)
main+8:     mov     %rsi,(%rsp)
main+12:    mov     0xc(%rsp),%eax
main+16:    mov     %eax,%edi
main+18:    callq   0x40056e    <a>
main+23:    add     $0x10,%rsp
main+27:    retq
```

```
0x7fffffffd6c8:   0x00007fffffffd7b8
0x7fffffffd6d0:   0x0000000100000000
0x7fffffffd6d8:   start_main+245
```

AMD64 Stack and Code Example (6)

- **Put the first argument for a() in EDI**
  - It's already there ...

```
main+0:     sub     $0x10,%rsp
main+4:     mov     %edi,0xc(%rsp)
main+8:     mov     %rsi,(%rsp)
main+12:    mov     0xc(%rsp),%eax
main+16:    mov     %eax,%edi
main+18:    callq   0x40056e    <a>
main+23:    add     $0x10,%rsp
main+27:    retq
```

```
0x7fffffffd6c8:   0x00007fffffffd7b8
0x7fffffffd6d0:   0x0000000100000000
0x7fffffffd6d8:   start_main+245
```

AMD64 Stack and Code Example (6)

- **Call a()**
  - The argument is passed in RDI

```
main+0:     sub     $0x10,%rsp
main+4:     mov     %edi,0xc(%rsp)
main+8:     mov     %rsi,(%rsp)
main+12:    mov     0xc(%rsp),%eax
main+16:    mov     %eax,%edi
main+18:    callq   0x40056e    <a>
main+23:    add     $0x10,%rsp
main+27:    retq
```

```
0x7fffffffd6c0:   main+23
0x7fffffffd6c8:   0x00007fffffffd7b8
0x7fffffffd6d0:   0x0000000100000000
0x7fffffffd6d8:   start_main+245
```

AMD64 Stack and Code Example (6)

```
a+0:     sub    $0x8,%rsp
a+4:     mov    %edi,0x4(%rsp)
a+8:     mov    0x4(%rsp),%eax
a+12:    mov    %eax,%edi
a+14:    callq  0x400556    <b>
a+19:    add    $0x8,%rsp
a+23:    retq
```

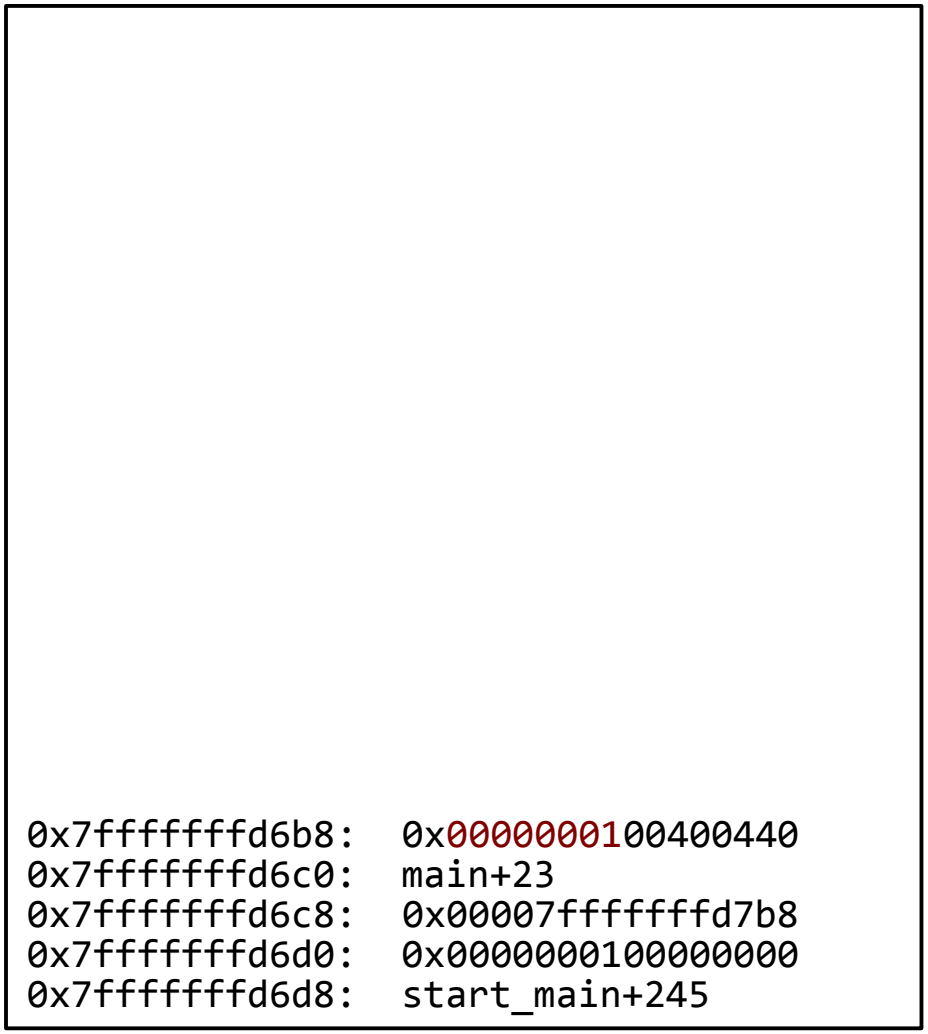
- **Allocate stack space**
  - Not zeroed out this time

```
0x7fffffffd6b8:   _start+0
              (raw: 0x0000000000400440)
0x7fffffffd6c0:   main+23
0x7fffffffd6c8:   0x00007fffffffd7b8
0x7fffffffd6d0:   0x0000000100000000
0x7fffffffd6d8:   start_main+245
```

AMD64 Stack and Code Example (6)

```
a+0:     sub    $0x8,%rsp
a+4:     mov    %edi,0x4(%rsp)
a+8:     mov    0x4(%rsp),%eax
a+12:    mov    %eax,%edi
a+14:    callq  0x400556    <b>
a+19:    add    $0x8,%rsp
a+23:    retq
```

- **Save the first argument on stack**
  - Again, not 8B aligned
  - The rest of old address remains



AMD64 Stack and Code Example (6)

```
a+0:     sub    $0x8,%rsp
a+4:     mov    %edi,0x4(%rsp)
a+8:     mov    0x4(%rsp),%eax
a+12:    mov    %eax,%edi
a+14:    callq  0x400556    <b>
a+19:    add    $0x8,%rsp
a+23:    retq
```

- **Fast-forward ... and return from b( )**
  - Nothing new there

```
0x7fffffffd6b8:   0x0000000100400440
0x7fffffffd6c0:   main+23
0x7fffffffd6c8:   0x00007fffffffd7b8
0x7fffffffd6d0:   0x0000000100000000
0x7fffffffd6d8:   start_main+245
```

AMD64 Stack and Code Example (6)

```
a+0:     sub    $0x8,%rsp
a+4:     mov    %edi,0x4(%rsp)
a+8:     mov    0x4(%rsp),%eax
a+12:    mov    %eax,%edi
a+14:    callq  0x400556    <b>
a+19:    add    $0x8,%rsp
a+23:    retq
```

- **Destroy a()'s stack frame**
  - No RBP handling necessary

```
0x7fffffffd6c0:   main+23
0x7fffffffd6c8:   0x00007fffffffd7b8
0x7fffffffd6d0:   0x0000000100000000
0x7fffffffd6d8:   start_main+245
```

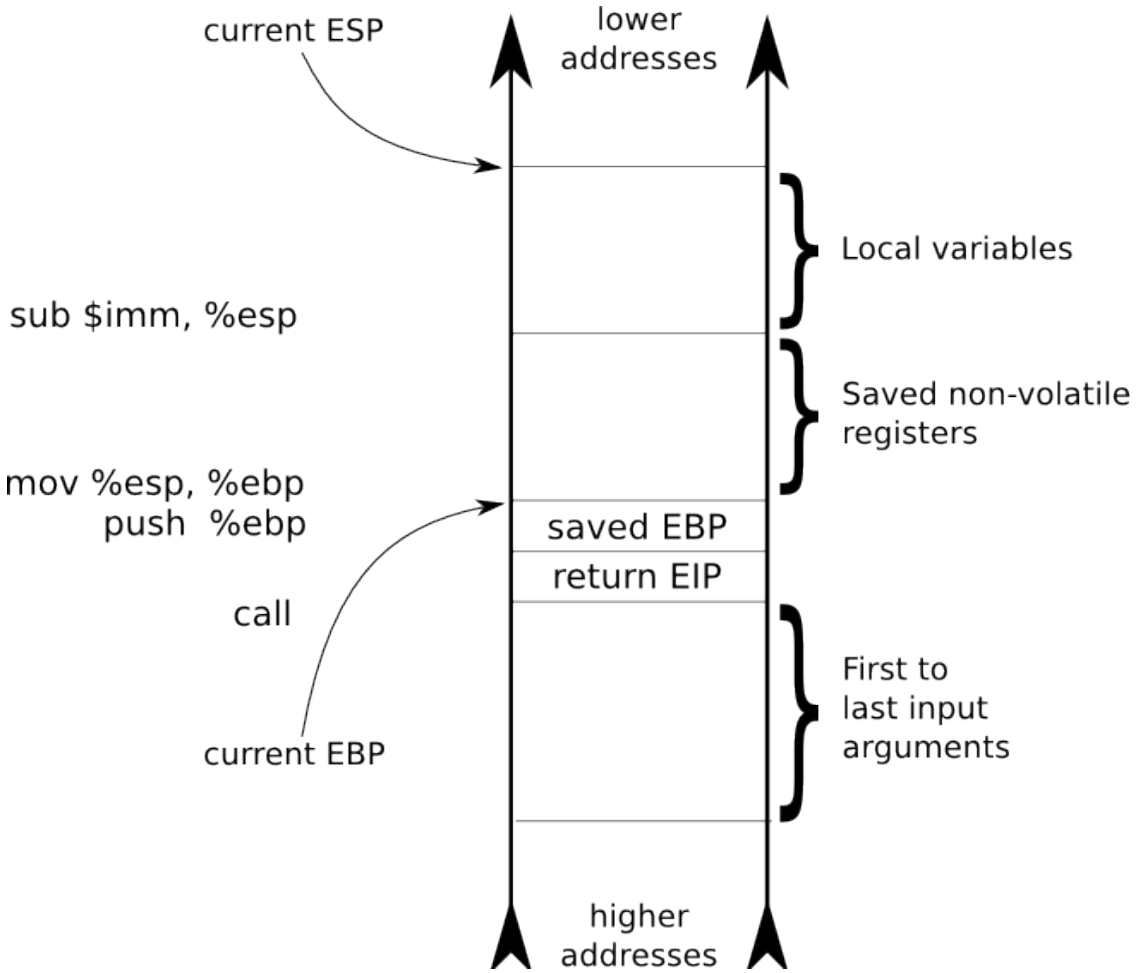
AMD64 Stack and Code Example (6)

```
a+0:     sub    $0x8,%rsp
a+4:     mov    %edi,0x4(%rsp)
a+8:     mov    0x4(%rsp),%eax
a+12:    mov    %eax,%edi
a+14:    callq  0x400556    <b>
a+19:    add    $0x8,%rsp
a+23:    retq
```

- **Return from `a()`**
  - Return value in RAX
  - Then the same ADD and RETQ in `main()`

```
0x7fffffffd6c8:   0x00007fffffffd7b8
0x7fffffffd6d0:   0x0000000100000000
0x7fffffffd6d8:   start_main+245
```

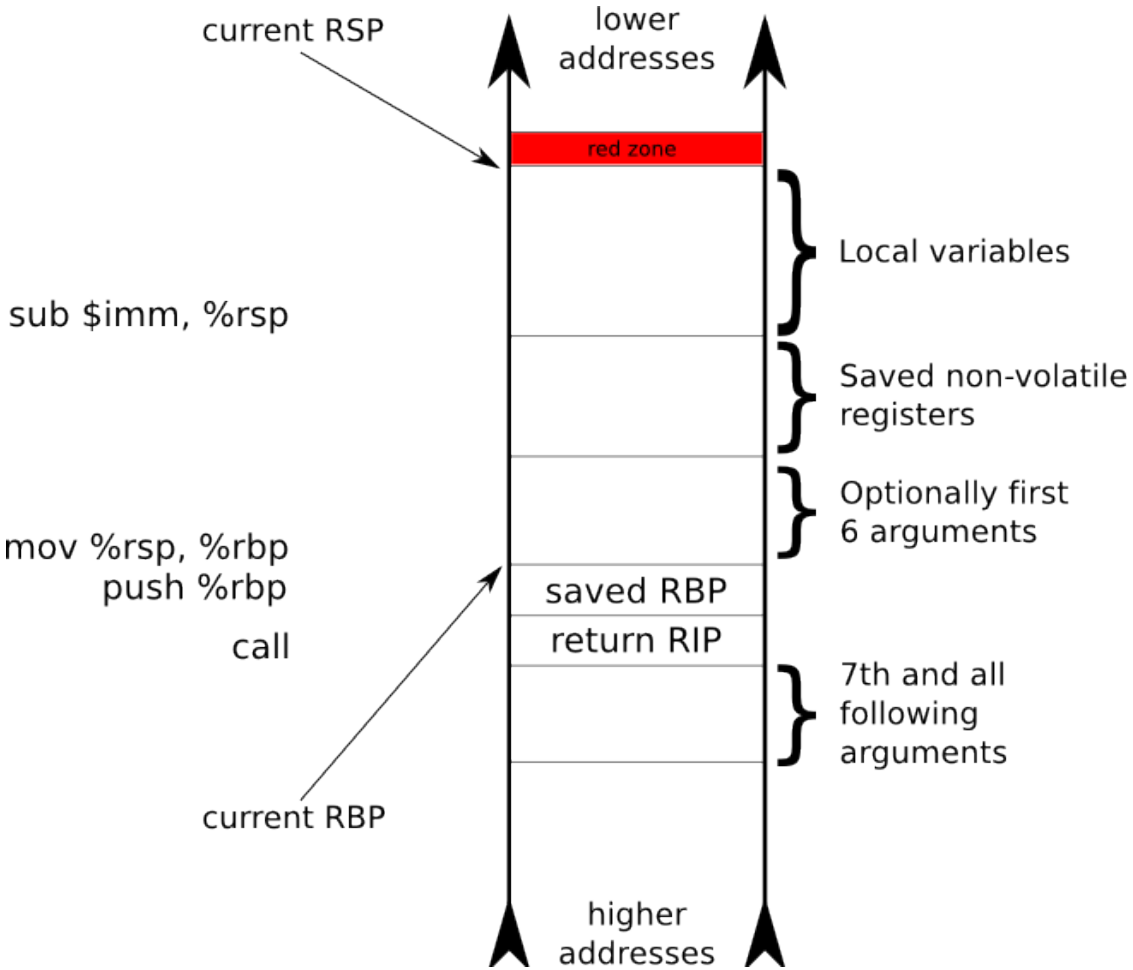
IA-32 ABI Cheat Sheet



EAX	return value
EBX	
ECX	
EDX	
ESI	
EDI	
EBP	frame pointer
ESP	stack pointer

non-volatile registers
volatile registers

AMD64 ABI Cheat Sheet



RAX	return value
RBX	
RCX	4th argument
RDX	3rd argument
RSI	2nd argument
RDI	1st argument
RBP	frame pointer
RSP	stack pointer
R8	5th argument
R9	6th argument
R10	
R11	
R12	
R13	
R14	
R15	

non-volatile registers
volatile registers