How Software Executes

CS-576 Systems Security

Instructor: Georgios Portokalidis

Fall 2018

Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety

Intel x86 Processors

Dominate laptop/desktop/server market

Evolutionary design

- Backwards compatible up until 8086, introduced in 1978
- Added more features as time goes on

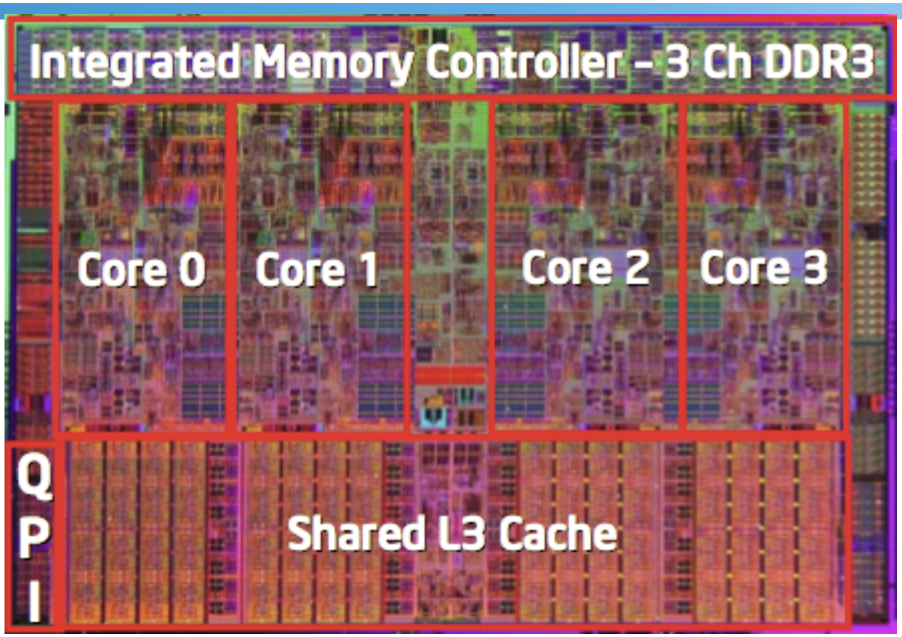
Complex instruction set computer (CISC)

- Many different instructions with many different formats
  - But, only small subset encountered with Linux programs
- Hard to match performance of Reduced Instruction Set Computers (RISC)
- But, Intel has done just that!
  - In terms of speed. Less so for low power.

Intel x86 Processors

Machine Evolution

386	1985	0.3M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7	2008	731M



Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores

x86 Integer Registers

General purpose registers

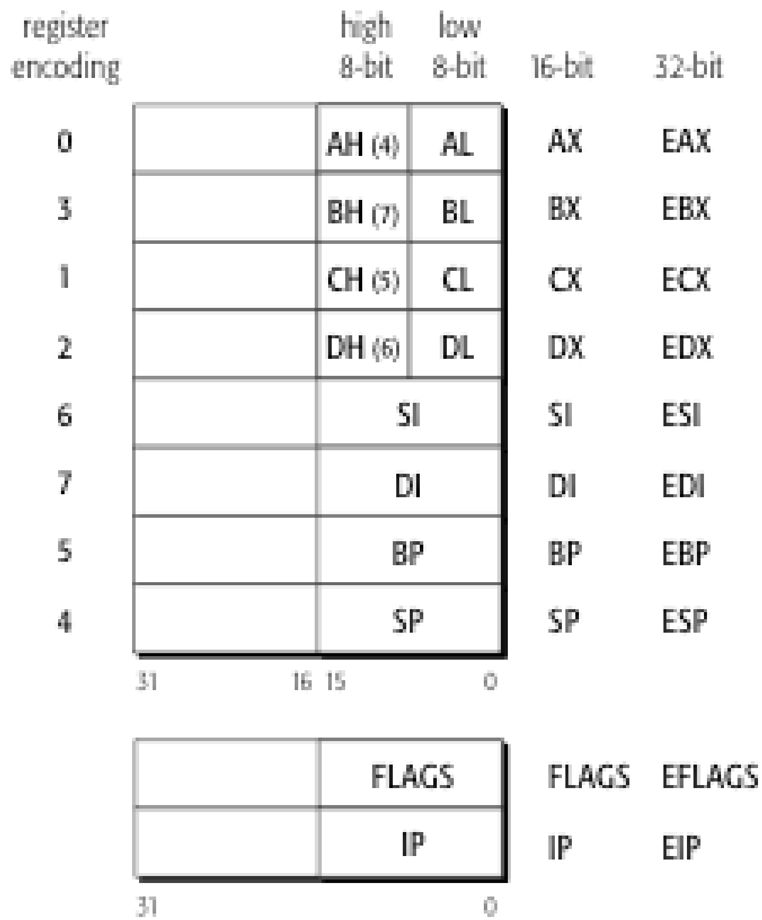
- On 32-bit architectures EAX, EBX, ECX, EDX, EDI, ESI, ESP, EBP

The instruction pointer (IP)

- Also referred to as program counter (PC)
- EIP on 32-bit

FLAGS register

- Used for control flow operations, etc.
- EFLAGS



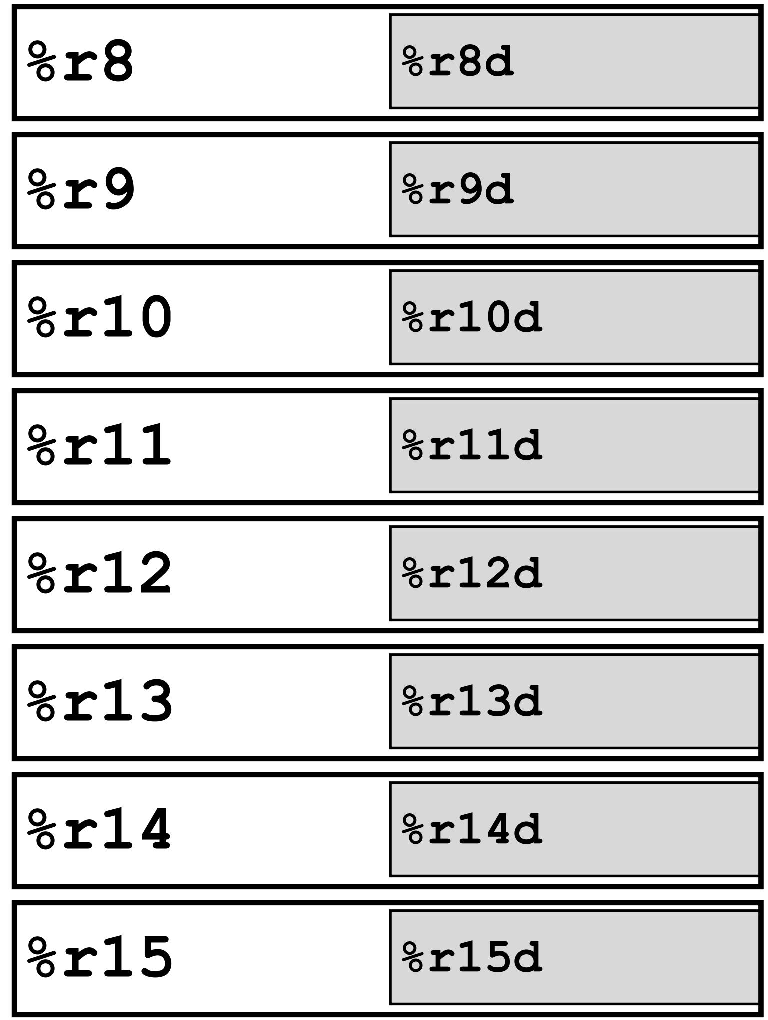
x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

x86-64 Integer Registers

Can reference low-order bytes too

- d suffix for lower 32-bits (r8d)
- w suffix for lower 16-bits (r8w)
- b suffix for lower 8-bits (r8b)



Typical Register Uses

EAX: accumulator

EBX : Pointer to data

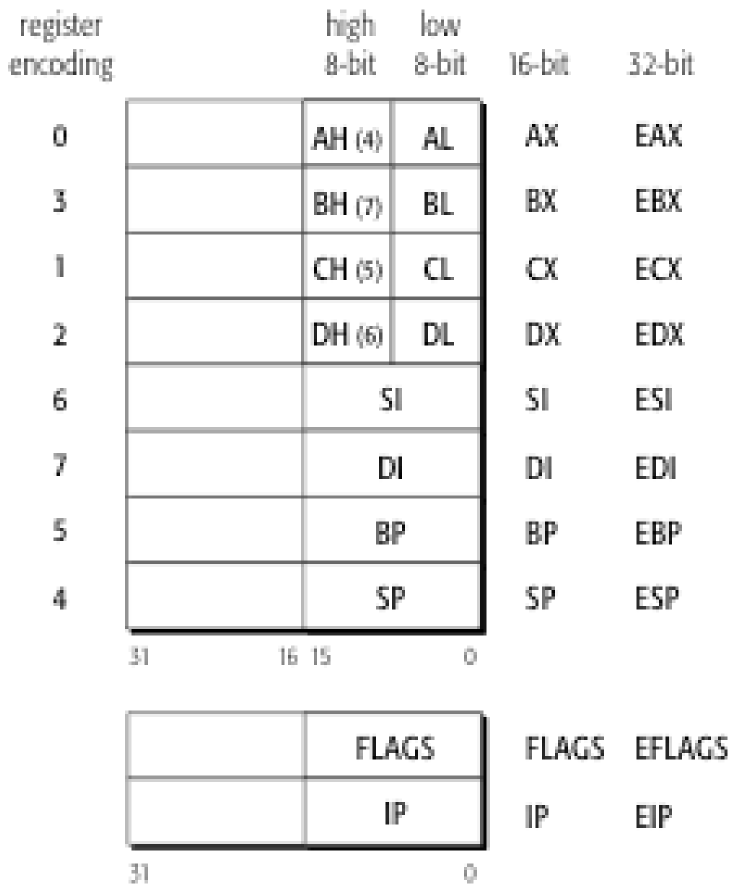
ECX: Counter for string operations and loops

EDX: I/O Operations

EDI: Destination for string operations

ESP: Stack pointer

EBP: Frame pointer



Assembly Syntax

Intel: OP dest, src

AT&T: OP src, dest

Unix systems prefer AT&T

- We are going to use the same as the GNU assembler (gas syntax)

Assembly Instructions

pushq: push quad word to stack

movq: Move quad word

imull: Signed multiply long

addl: Add long

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -20(%rbp)
movl    %esi, -24(%rbp)
movl    %edx, -28(%rbp)
movl    -20(%rbp), %eax
imull   -28(%rbp), %eax
movl    %eax, %edx
movl    -24(%rbp), %eax
addl    %edx, %eax
imull   -28(%rbp), %eax
```

Operand Sizes

Instructions include a suffix that indicates the size of the operand(s)

Register operands are prefixed with a **%**

pushq %rax

Intel syntax does not include a suffix, size depends on the size of the operand

Register operands must match size
For example,

- quad → rax
- long → eax
- word → ax
- byte → ah or al

Memory Operands

Parentheses indicate a memory operand

Each memory address can be defined as: **B**ase+**I**ndex***S**cale+**D**isp

- In AT&T syntax:
  disp(base, index, scale)
- disp, index, and scale are optional

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -20(%rbp)
movl    %esi, -24(%rbp)
movl    %edx, -28(%rbp)
movl    -20(%rbp), %eax
imull   -28(%rbp), %eax
movl    %eax, %edx
movl    -24(%rbp), %eax
addl    %edx, %eax
imull   -28(%rbp), %eax
```

Memory Addressing Modes

Normal (B) Mem[Reg[R]]

- Register R specifies memory base address
- Pointer dereferencing in C

```
movq (%rcx),%rax
```

Displacement D(B) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Memory Addressing Modes

Most General Form

D(B,I,S) Mem[Reg[R_b]+S*Reg[R_i]+ D]

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register
- Ri: Index register: Any, except for **%rsp**
- S: Scale: 1, 2, 4, or 8

```
movq 8(%rbp, %rax, 4),%rdx
```

Immediates

Constants or immediates are defined using **$**

In decimal, unless:

- 0x prefix is used → hexadecimal
- 0 prefix is used → octal

```
addl    $1, %eax
```

Immediates can help you identify the syntax

Endianness

Memory representation of multi-byte integers

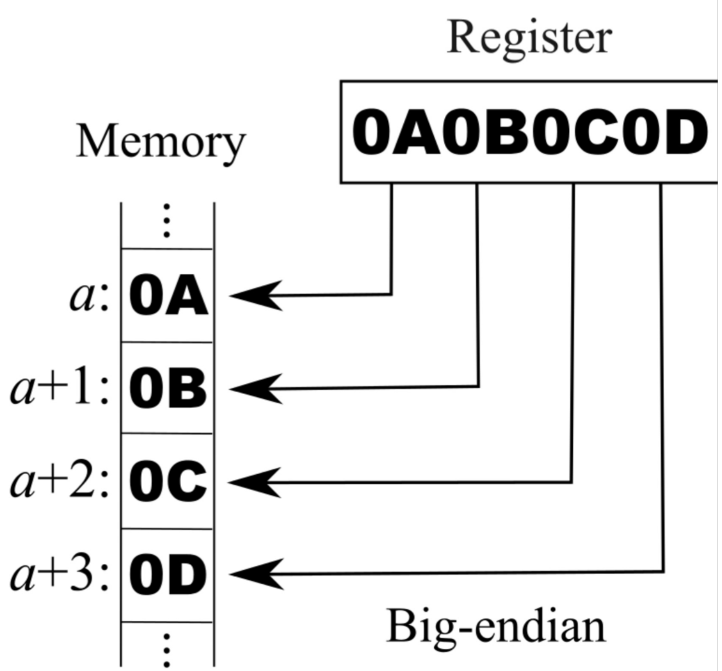
For example the integer: 0x0A0B0C0Dh

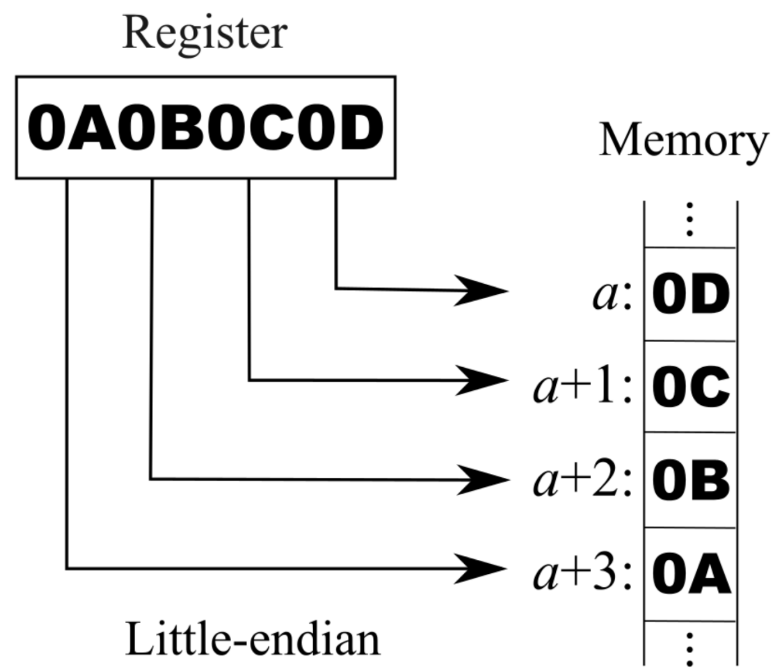
Big-endian↔highest order byte first

0A 0B 0C 0D

Little-endian↔lowest order byte first (X86)

0D 0C 0B 0A


Register
0A0B0C0D
Memory
a: 0A
a+1: 0B
a+2: 0C
a+3: 0D
Big-endian


Register
0A0B0C0D
Memory
a: 0D
a+1: 0C
a+2: 0B
a+3: 0A
Little-endian

Load Effective Address

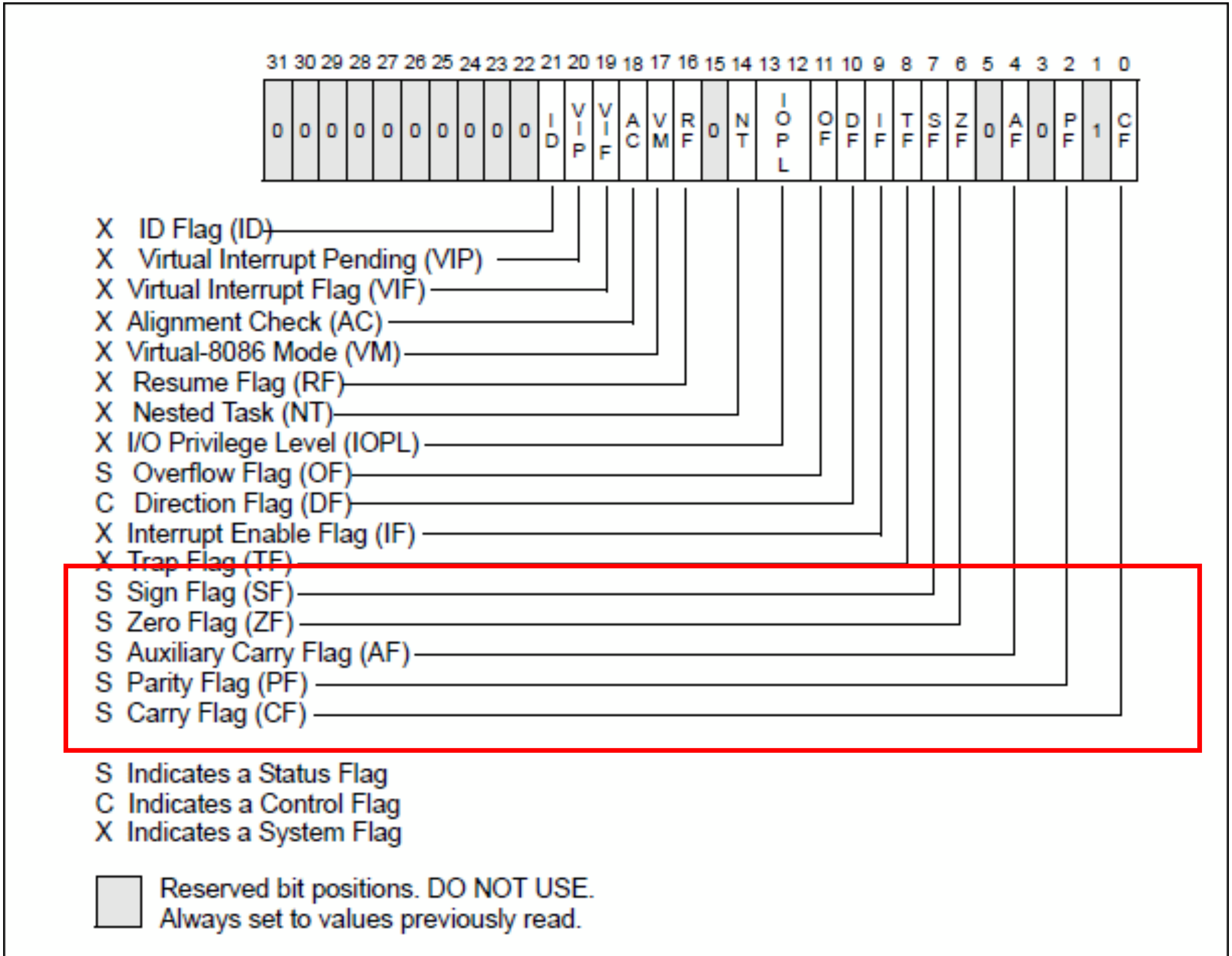
- `leaq` Src, Dst
  - Src is address mode expression
  - Set Dst to address denoted by expression
- Computing addresses without a memory reference
  - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form x + k*y
  - k = 1, 2, 4, or 8
- Example

```
leaq (%rdi,%rdi,2), %rax
```

Control Flow

```
if (a > b) {
        c = d;
} else {
        d = c;
}
```

```
13:   cmp     -0x8(%rbp),%eax
16:   jle     0xe
18:   mov     -0x10(%rbp),%eax
1b:   mov     %eax,-0xc(%rbp)
1e:   jmp     0x6
20:   mov     -0xc(%rbp),%eax
23:   mov     %eax,-0x10(%rbp)
26:   mov     -0xc(%rbp),%eax
```



EFLAGS Register

Condition Codes Set by Compare

Explicit Setting by Compare Instruction

- **`cmpq`** Src2, Src1
- **`cmpq b,a`** like computing **`a-b`** without setting destination

- CF set if carry out from most significant bit (used for unsigned comparisons)
- ZF set if **`a == b`**
- SF set if **`(a-b) < 0`** (as signed)
- OF set if two's-complement (signed) overflow
  `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

Condition Codes (Explicit Setting: Test)

Explicit Setting by Test instruction

- **`testq`** Src2, Src1
  - **`testq b,a`** like computing **`a&b`** without setting destination

- Sets condition codes based on value of Src1 & Src2
- Useful to have one of the operands be a mask

- ZF set when **`a&b == 0`**
- SF set when **`a&b < 0`**

Common Conditional Jumps

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)

Overview

Introduction

Anatomy of a program

Basic assembly

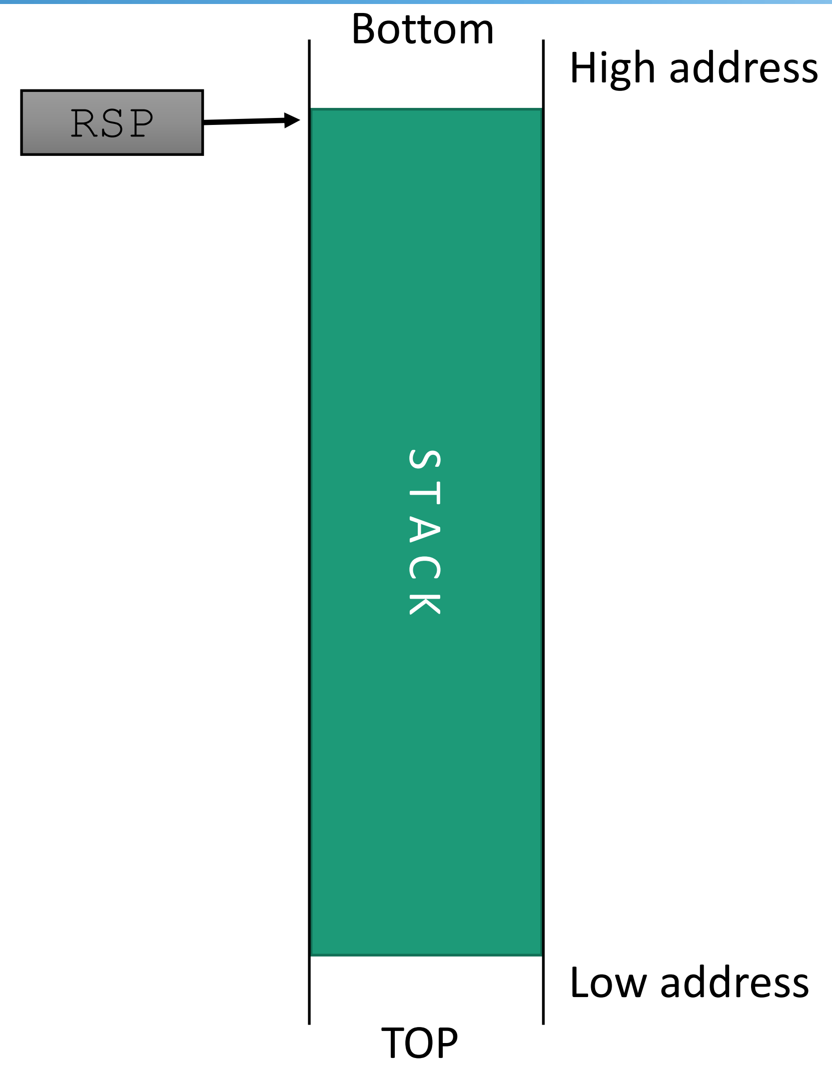
Anatomy of function calls (and returns)

Memory Safety

The Stack Pointer (SP)

The stack pointer points to the first element in the stack.

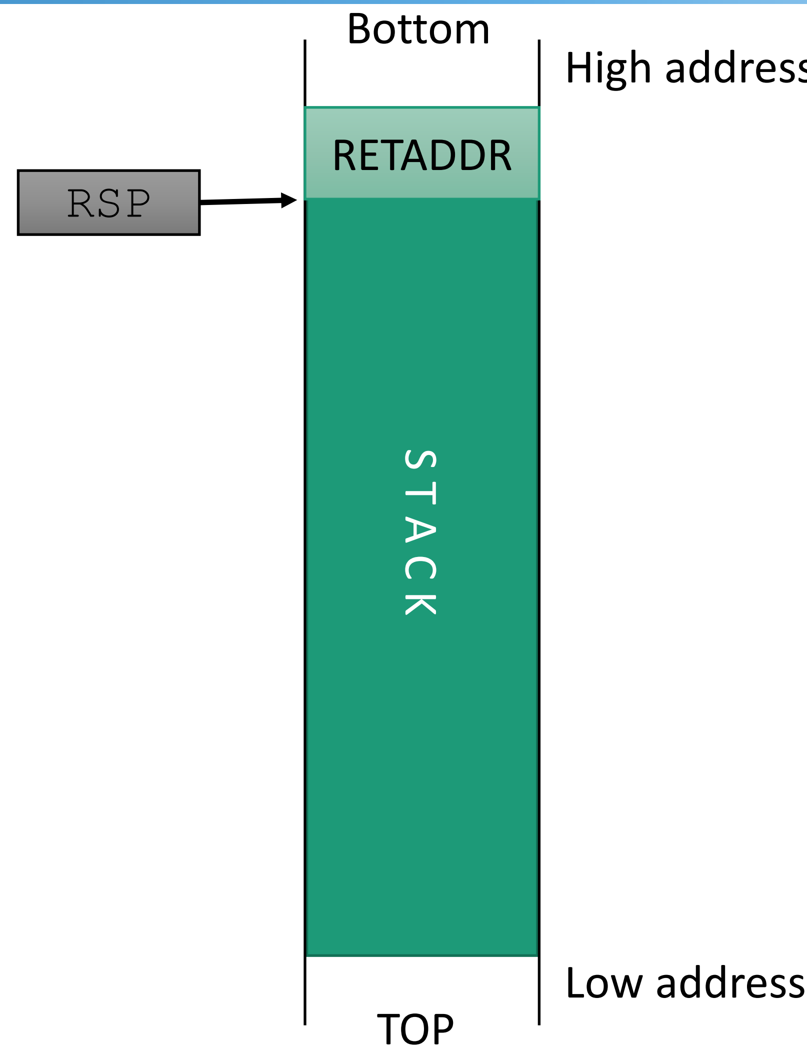
Usually the RSP/ESP is used to store the SP.



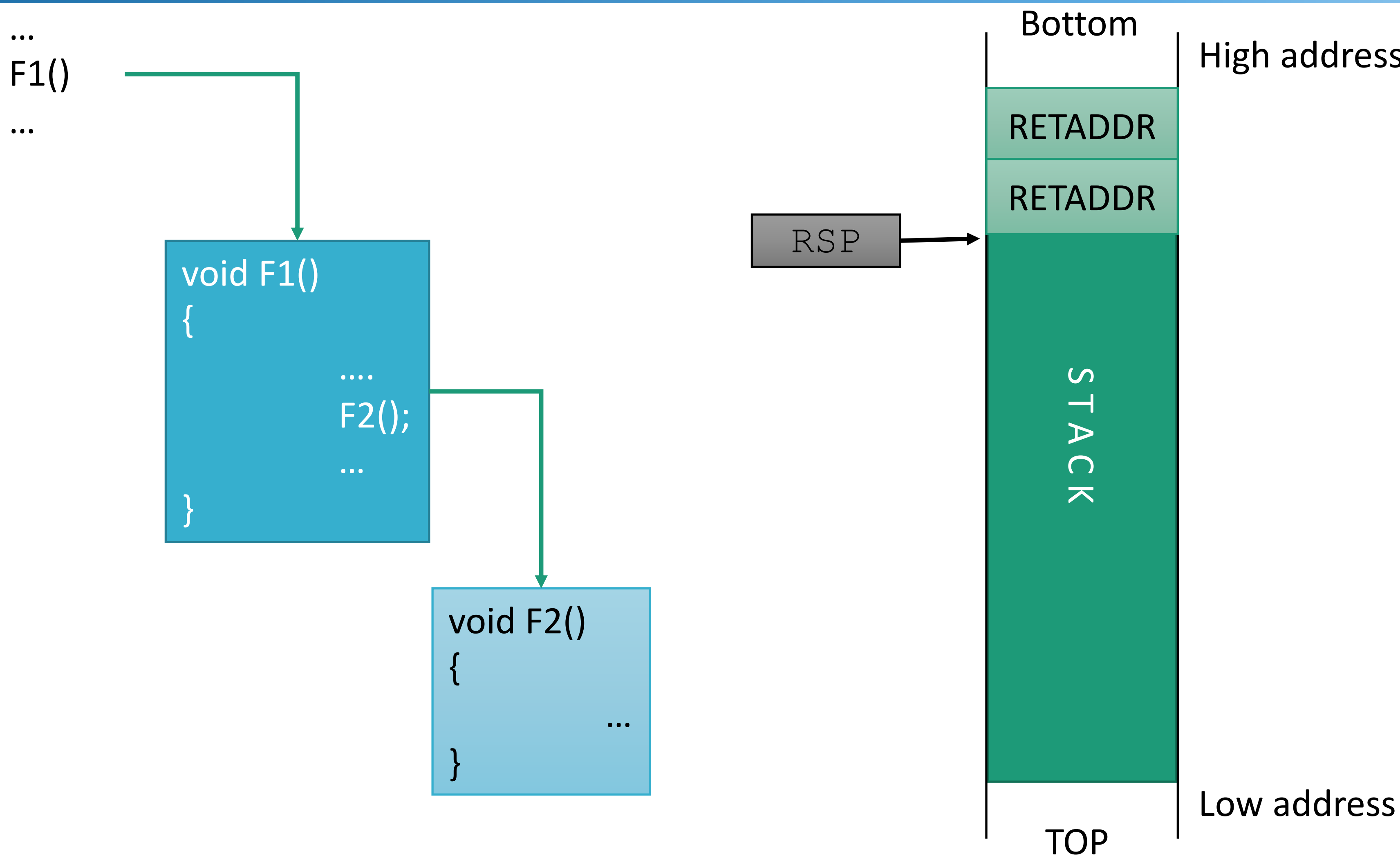
Simple Function Call

```
...
F1()
...
```

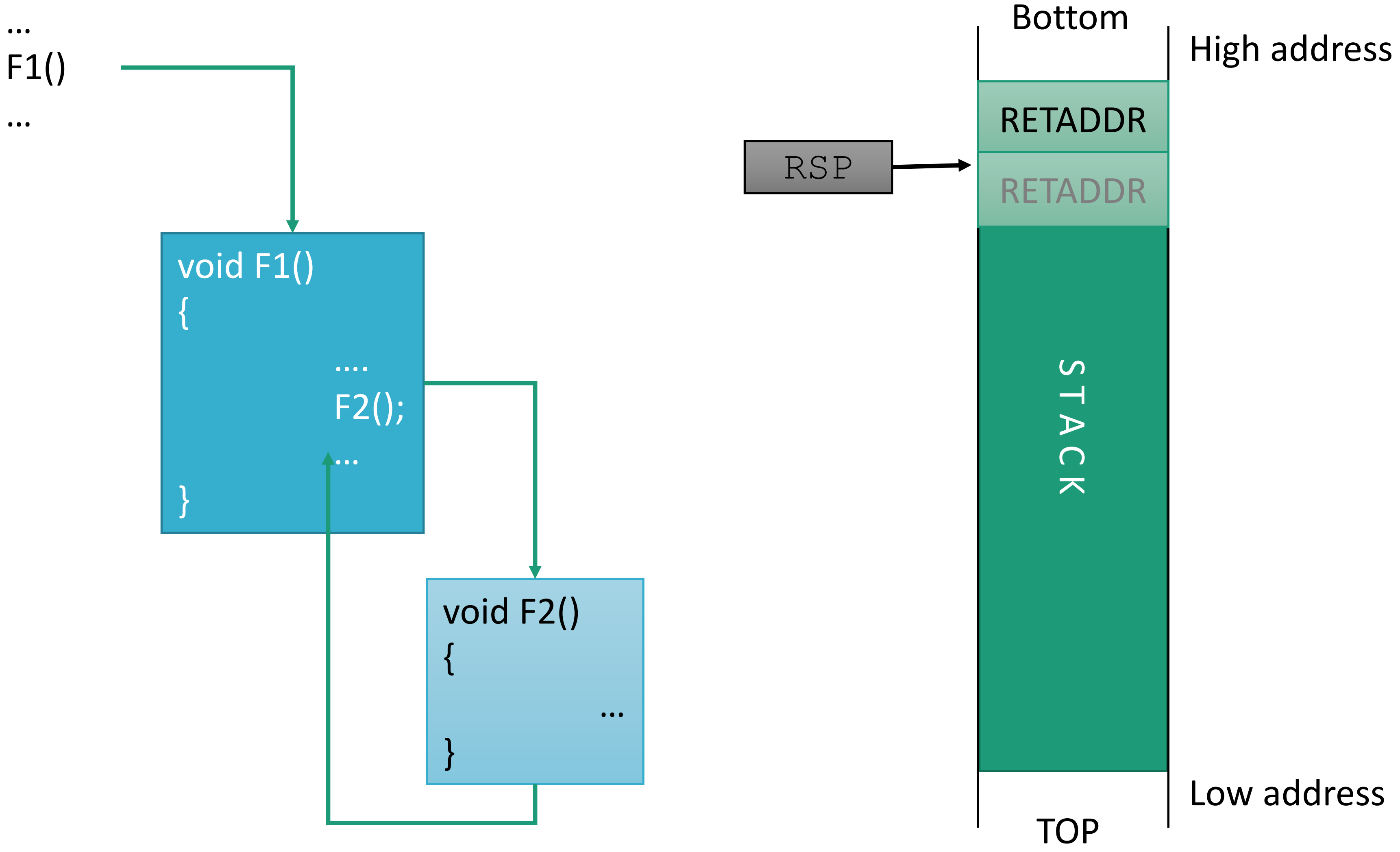
```
void F1()
{
        ....
        F2();
        ...
}
```



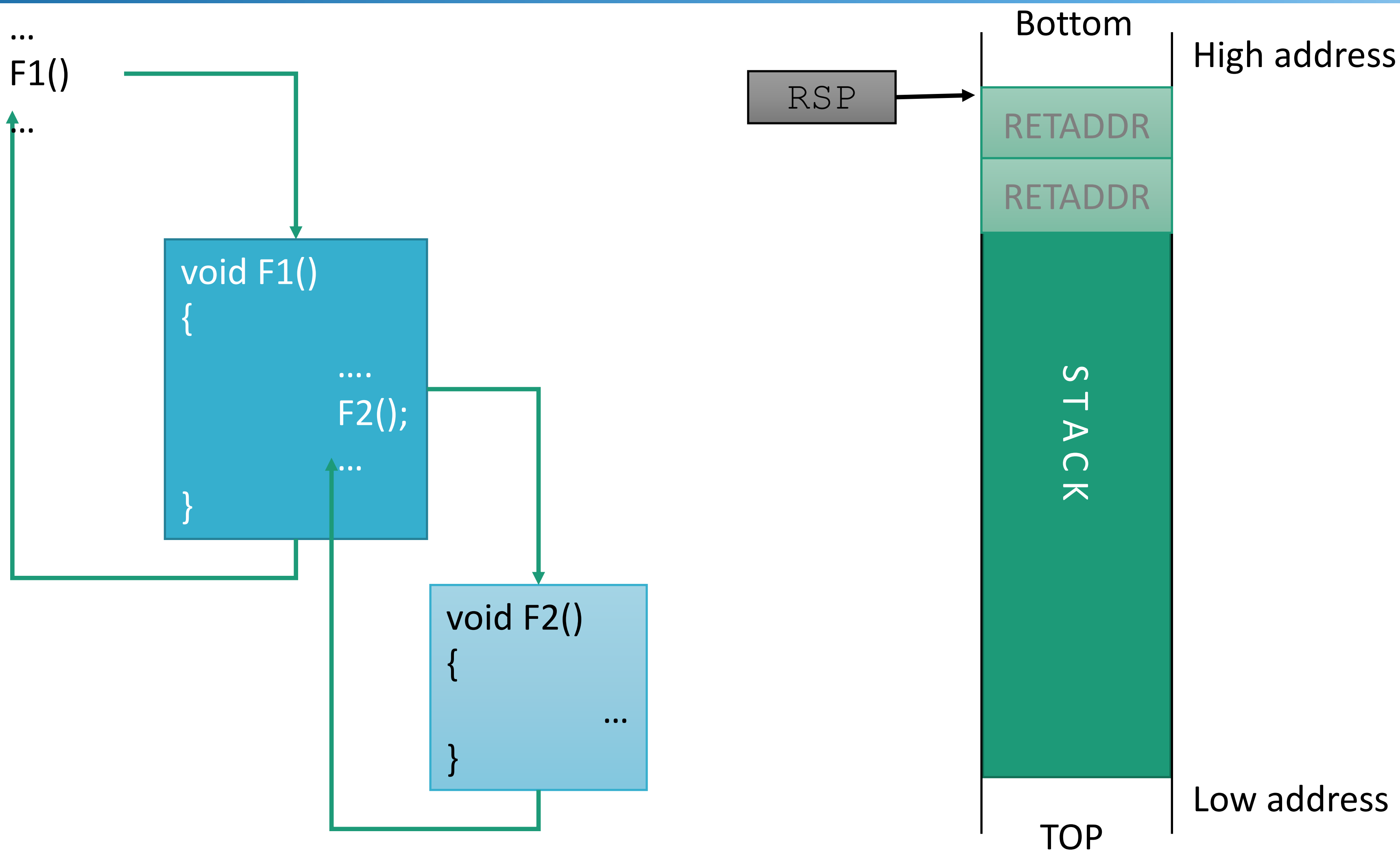
Simple Function Call



Simple Function Call



Simple Function Call



Function Calls

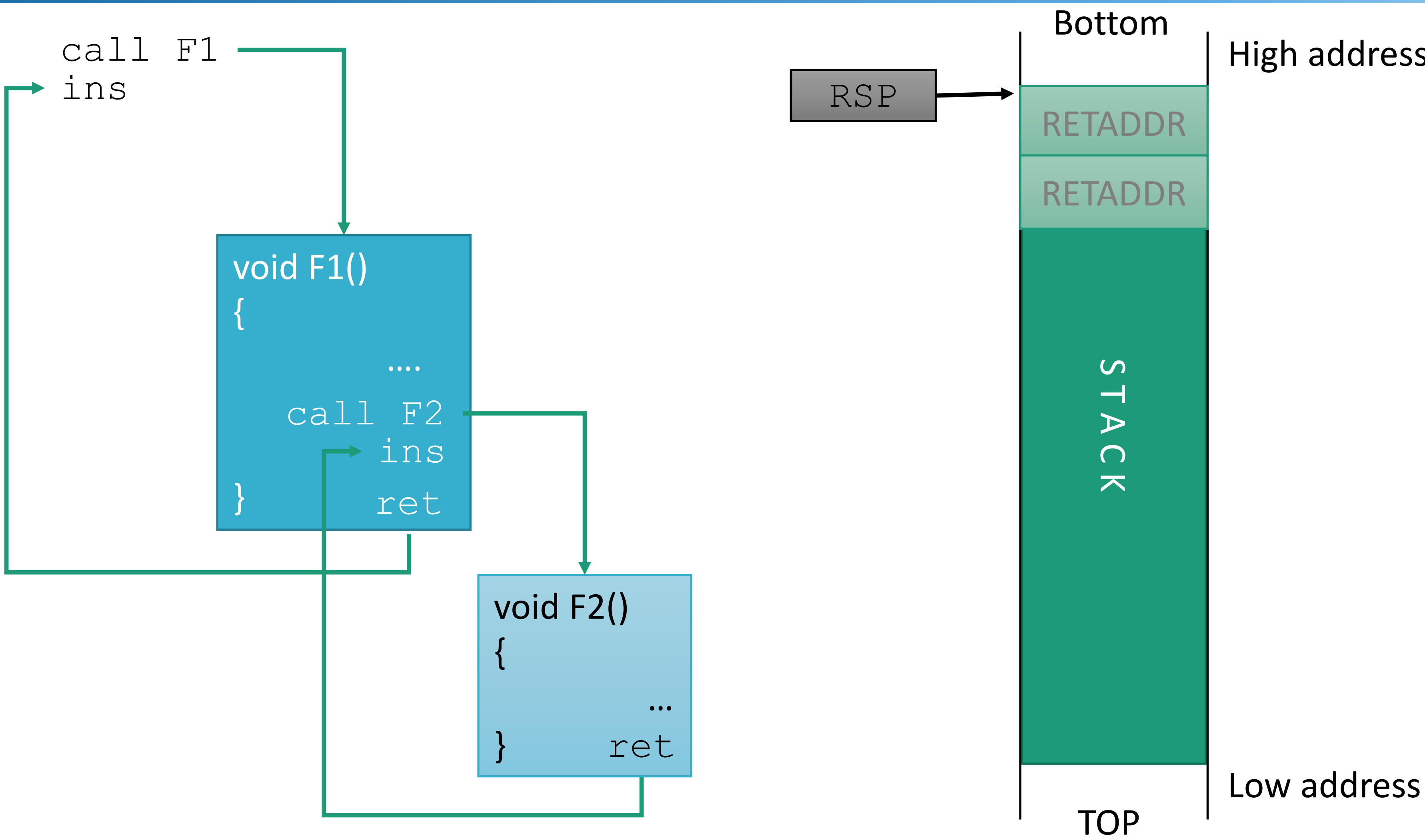
Function calls transfer control and use the stack to keep track of callees

- `call <address>` Transfer control to address and save the address of the next instruction on the stack
- `ret` Pop the address from the stack and transfer control to it

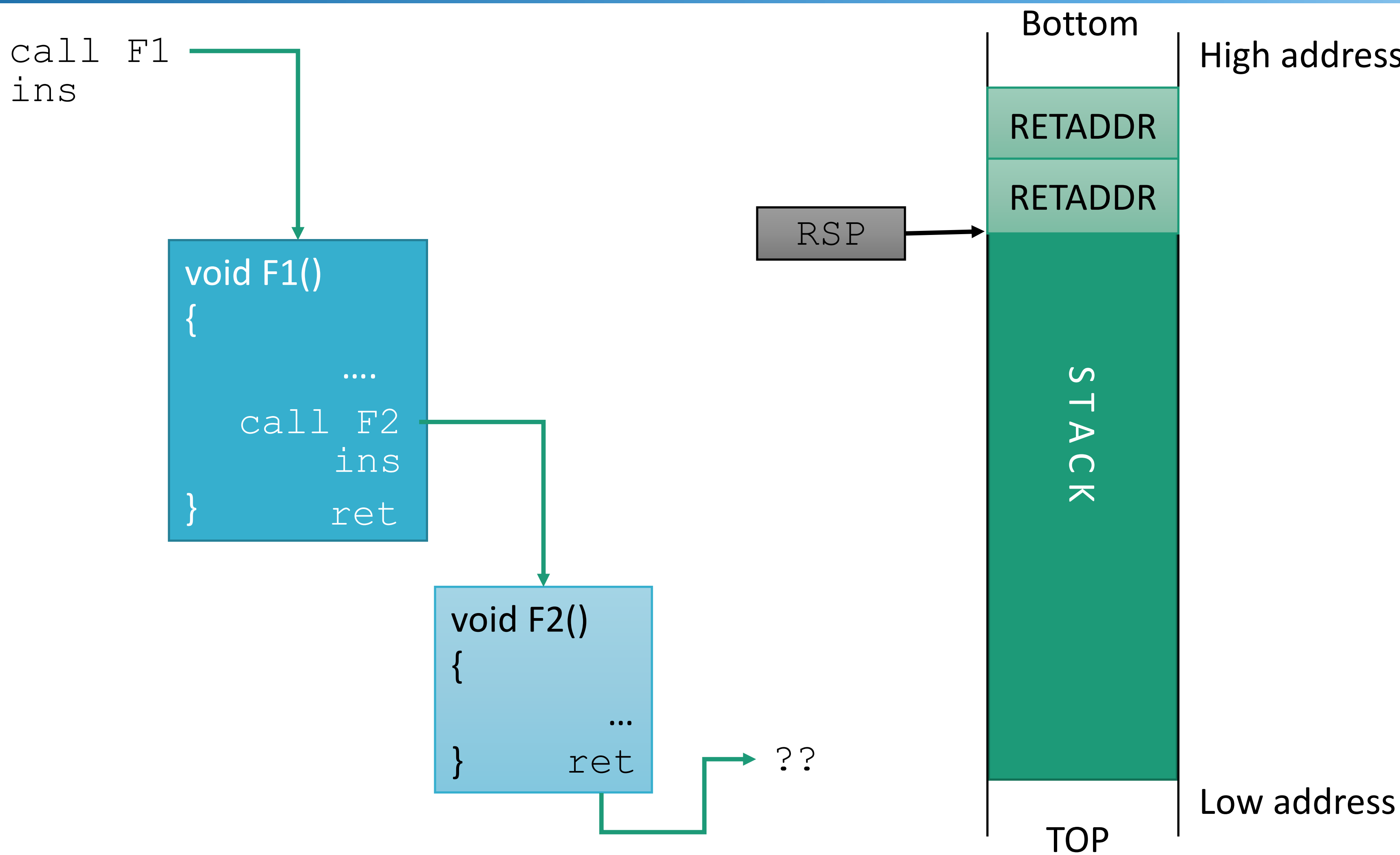
`call` and `ret` implicitly use the RSP register

- So does `push`/`pop`

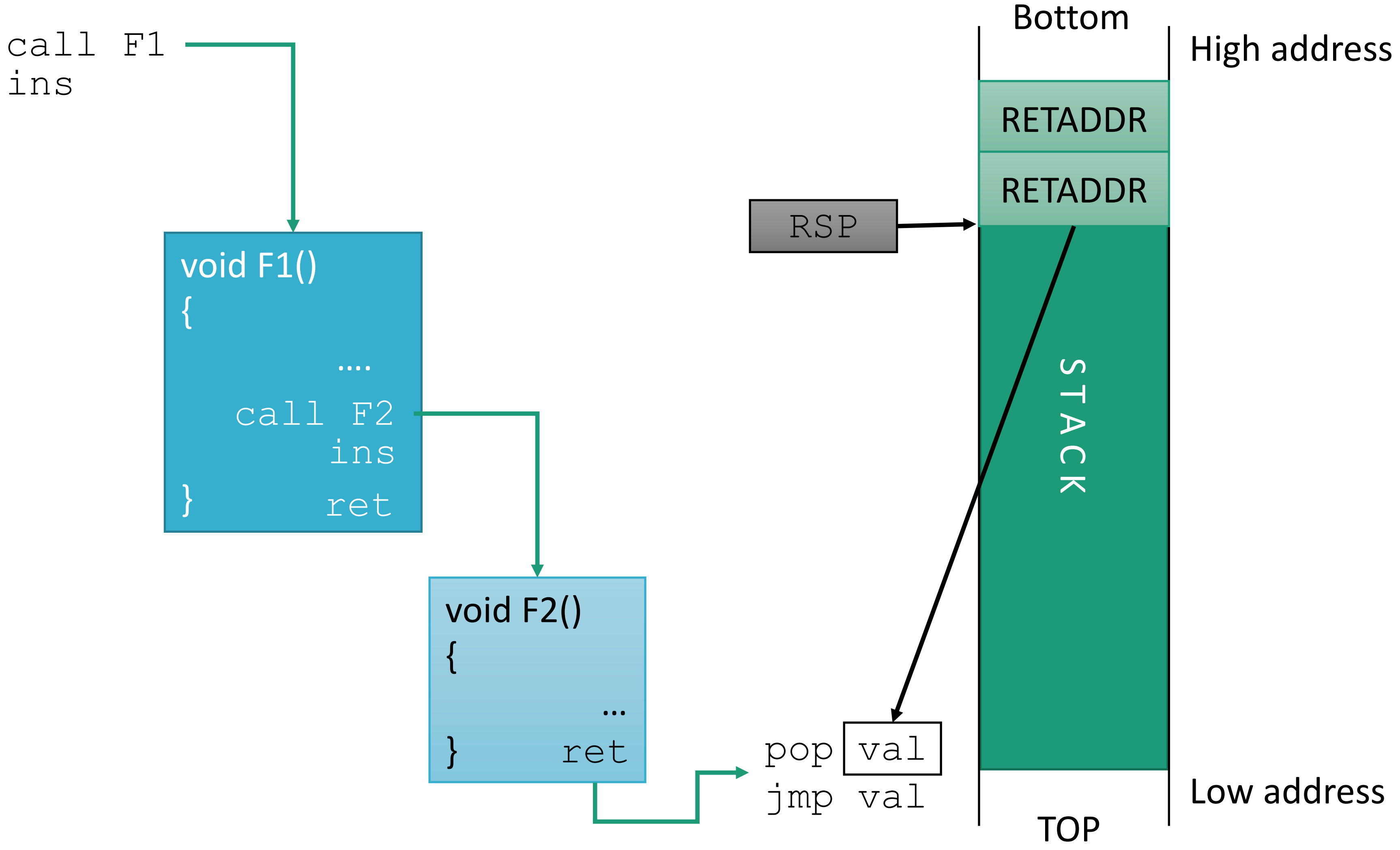
Simple Function Call



Simple Function Call

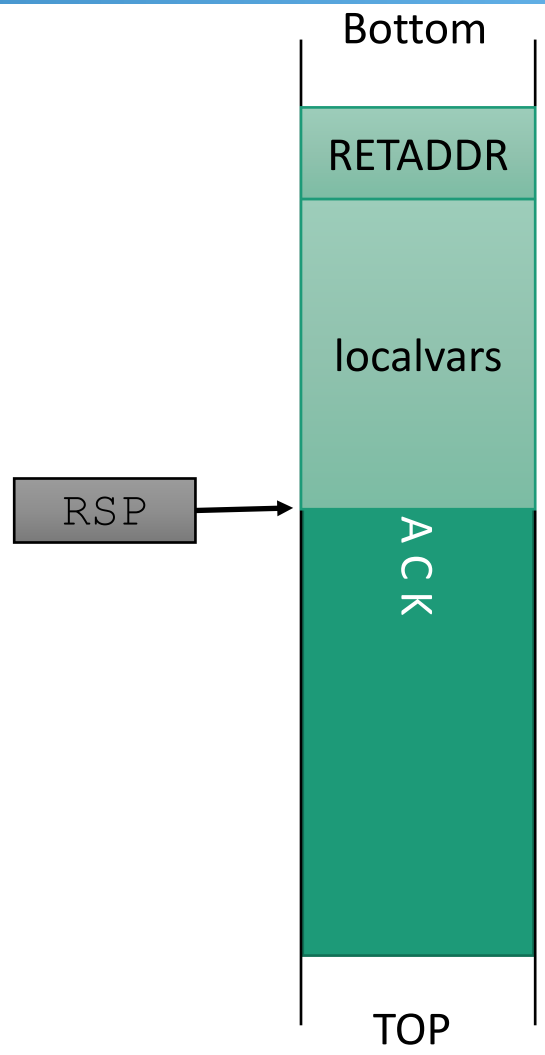


Simple Function Call



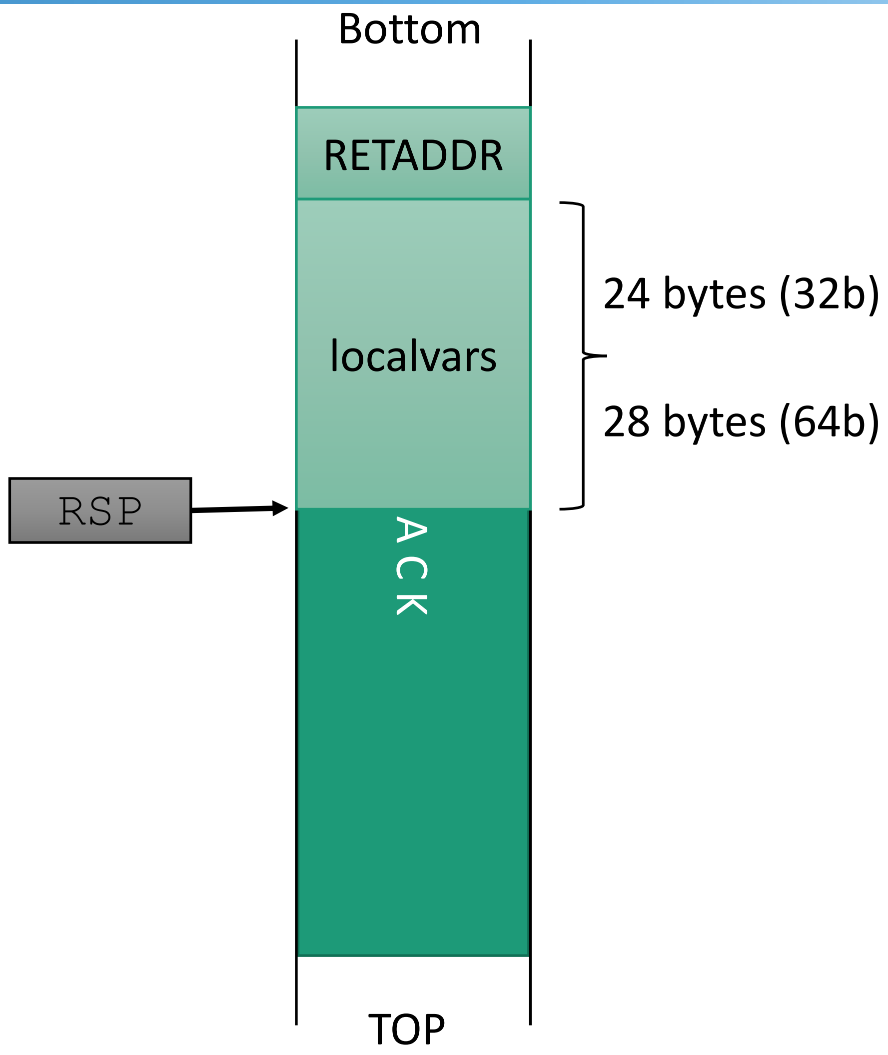
Local Variables

```
void F1()
{
        int a;
        char buf[16];
        unsigned long l;
        ….
        F2();
        …
}
```

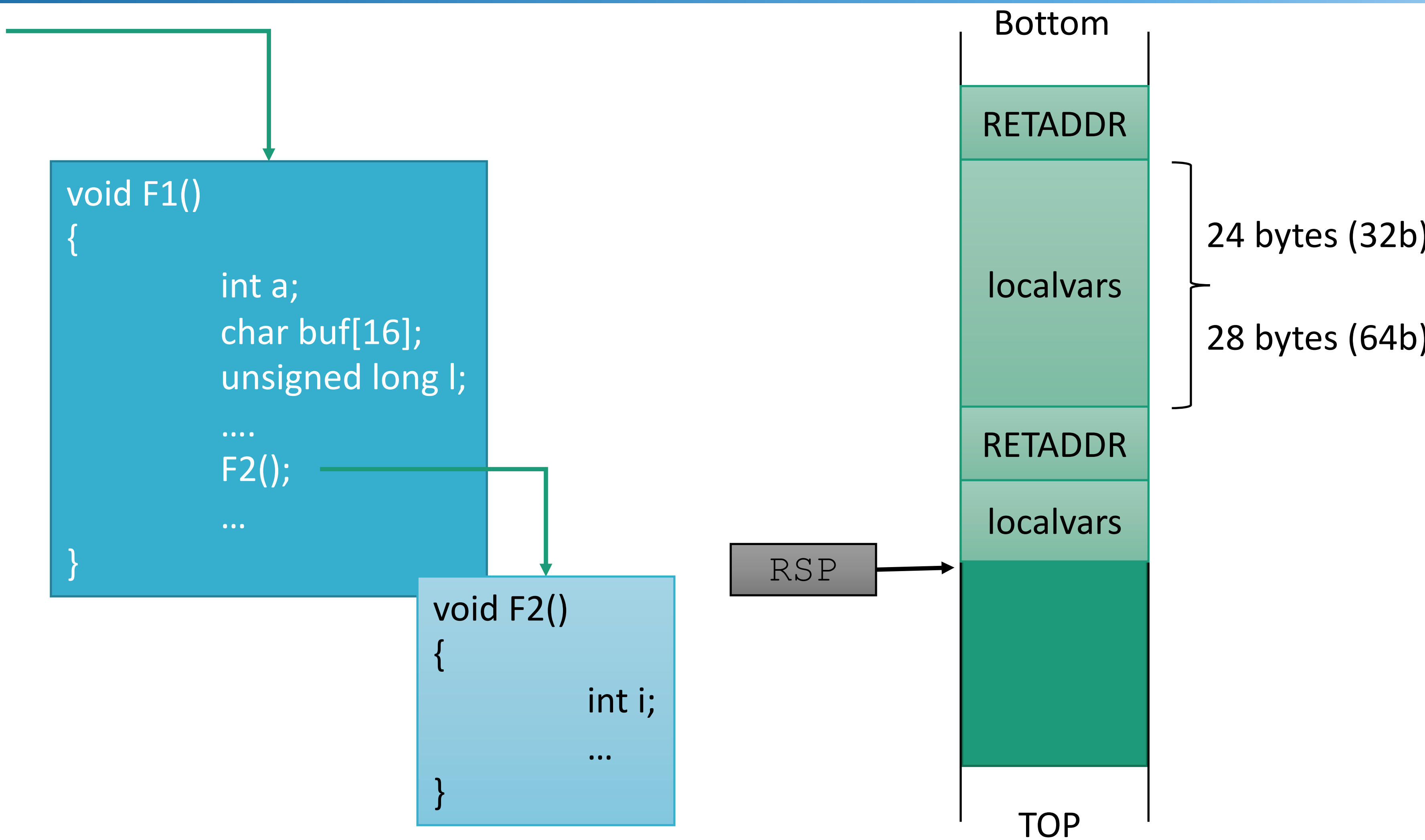


Local Variables

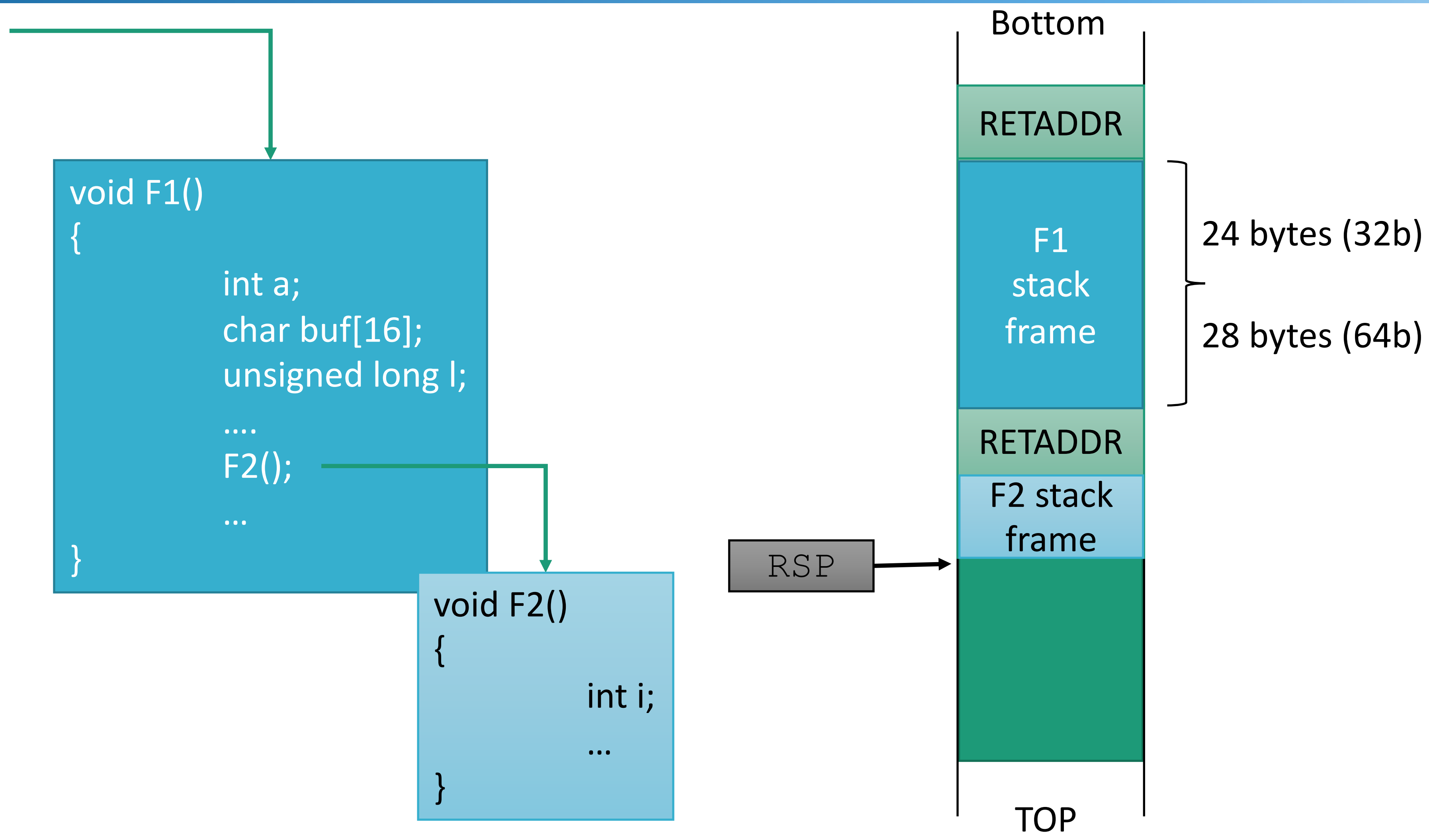
```
void F1()
{
        int a;
        char buf[16];
        unsigned long l;
        ....
        F2();
        ...
}
```



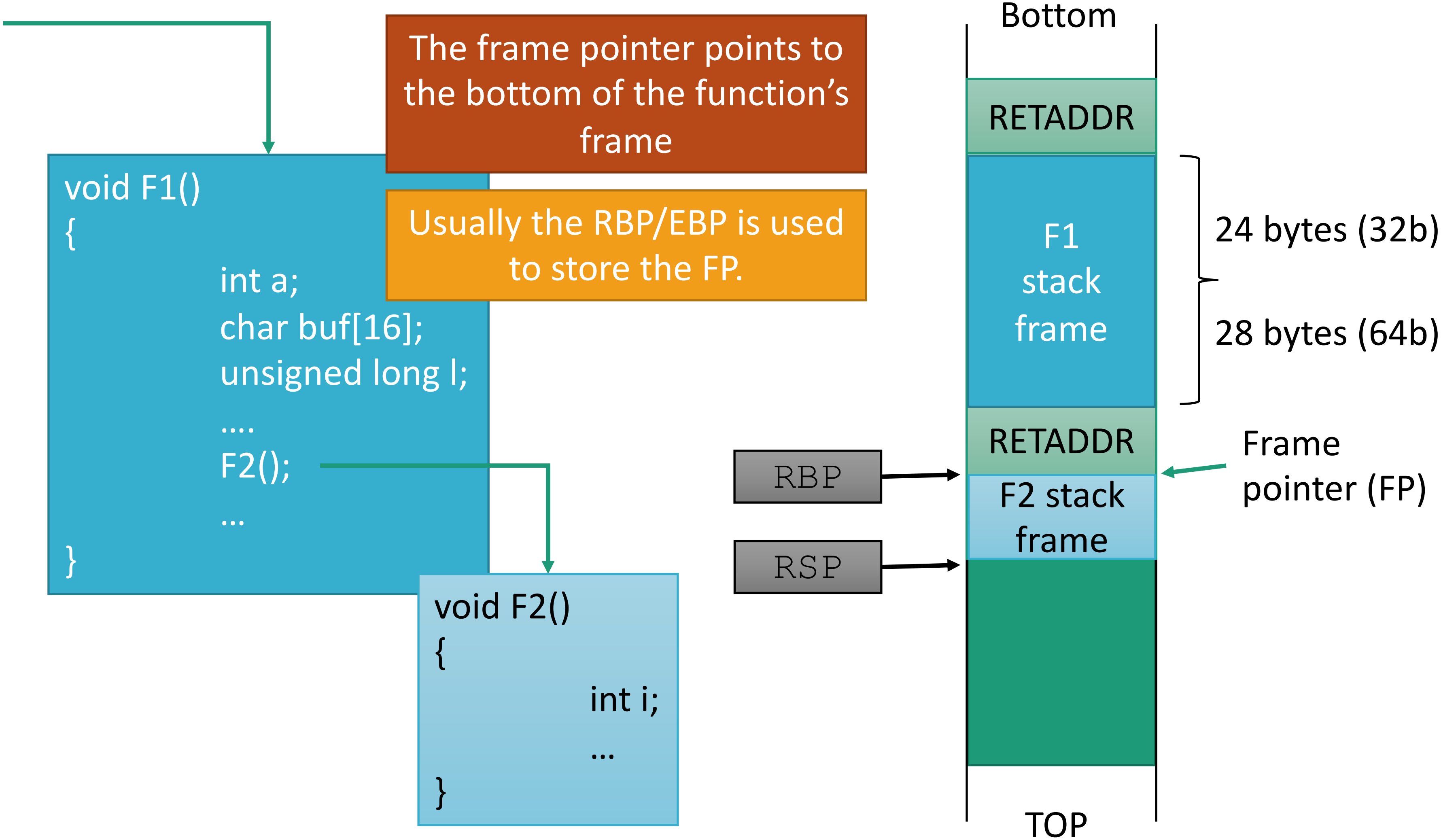
Local Variables



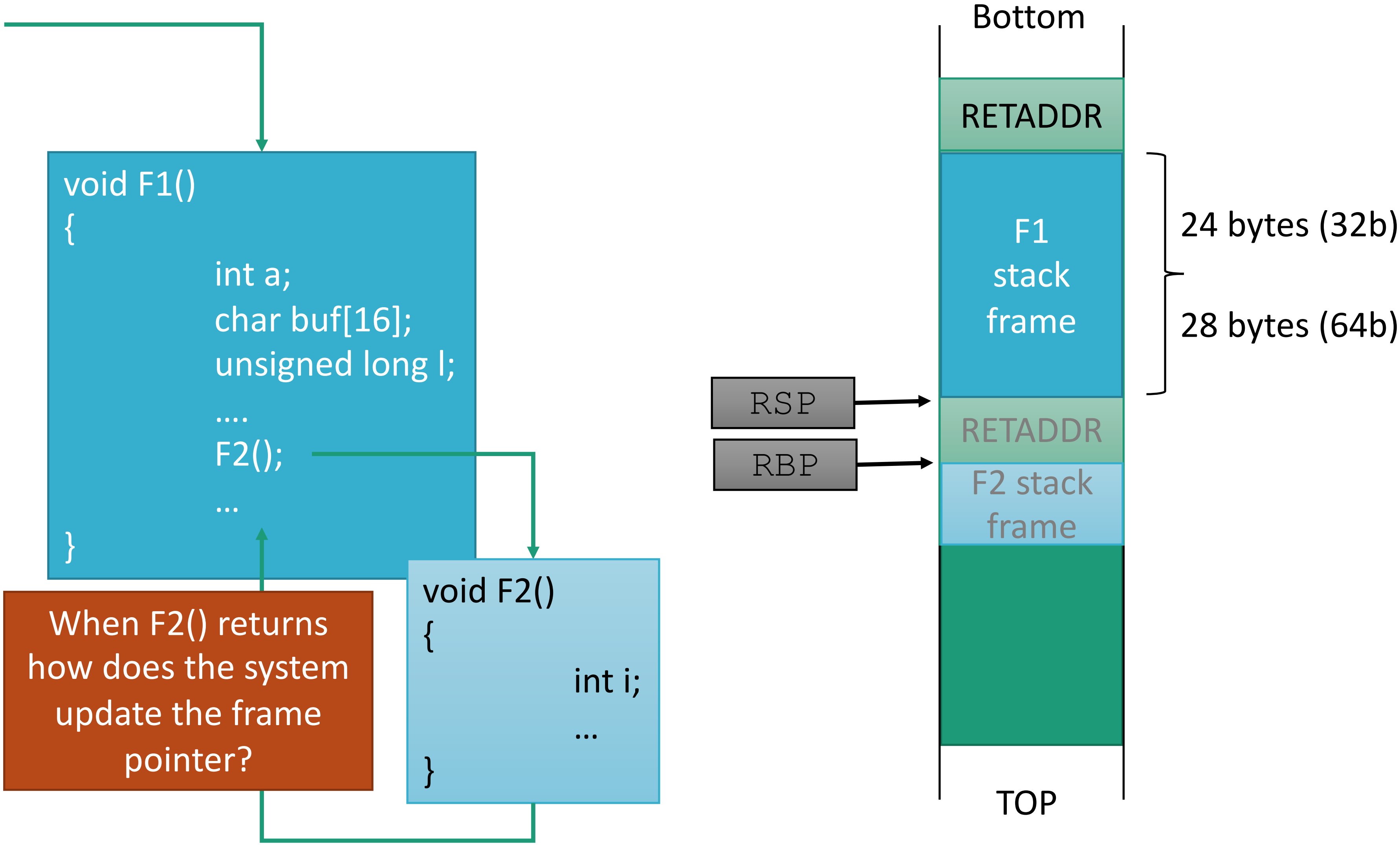
Stack Frames



Frame Pointer (FP)



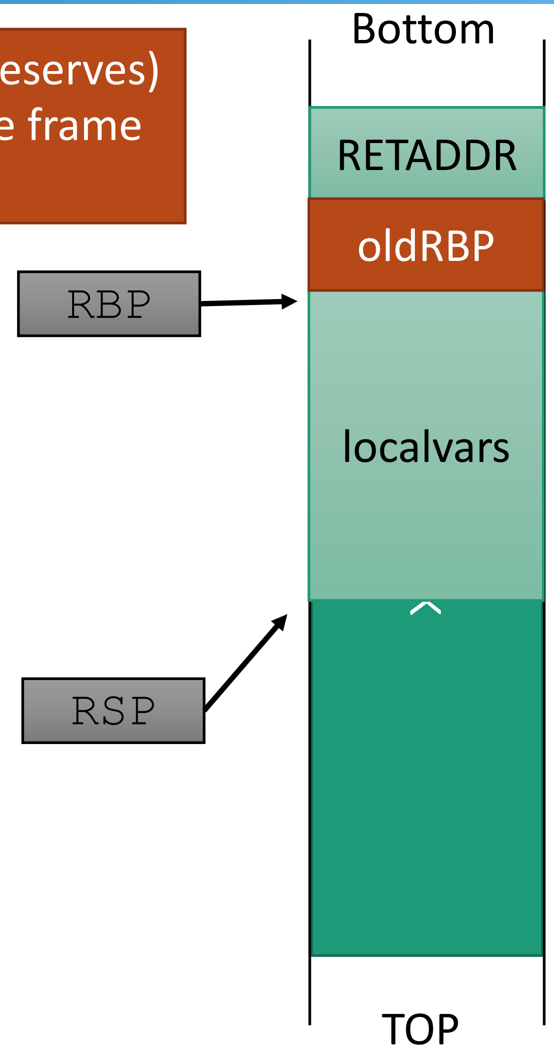
Frame Pointer (FP)



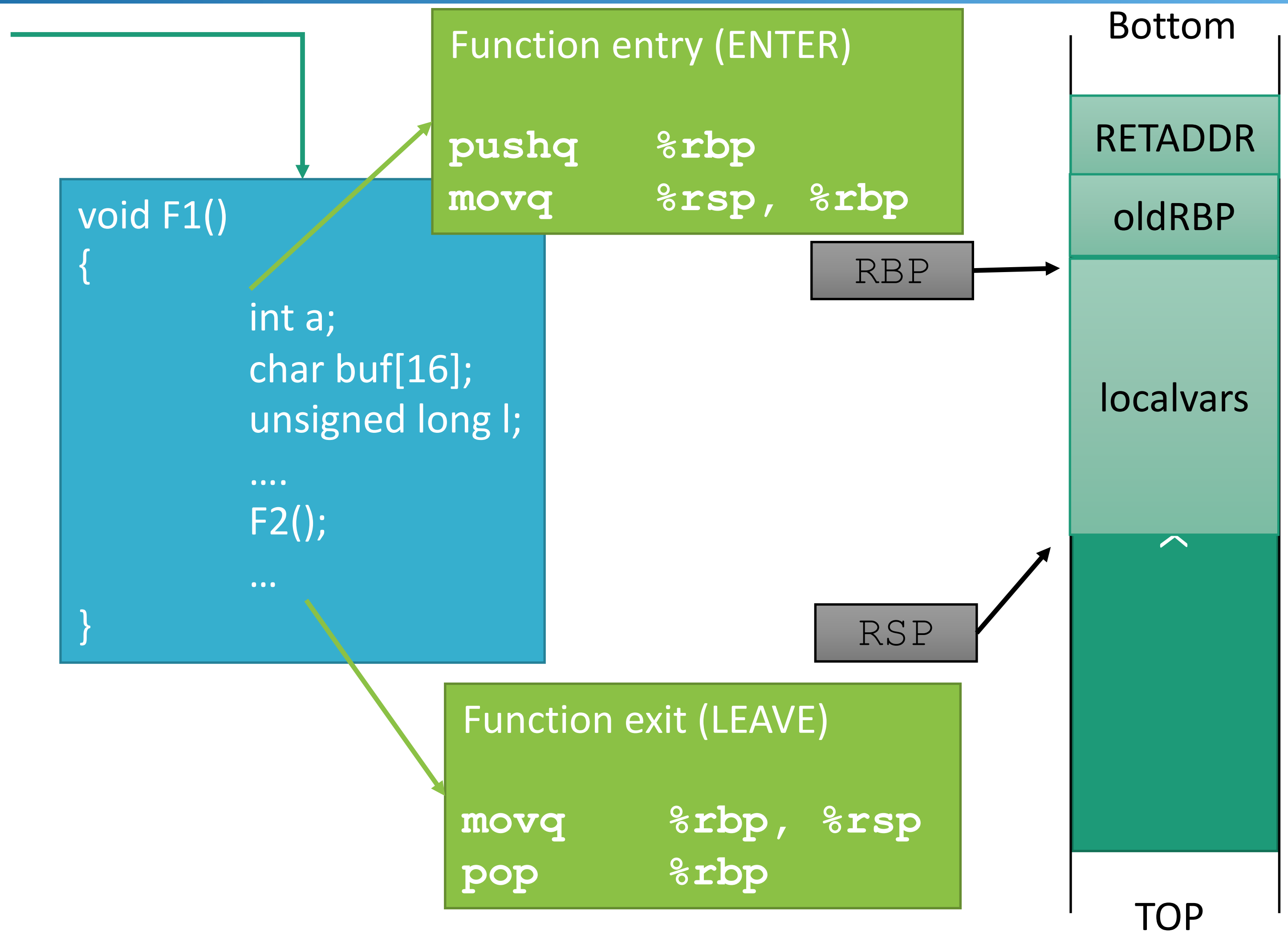
Saved Frame Pointer

The **callee** saves (preserves) the old value of the frame pointer.

```
void F1()
{
        int a;
        char buf[16];
        unsigned long l;
        ....
        F2();
        ...
}
```



Function Prologue/Epilogue



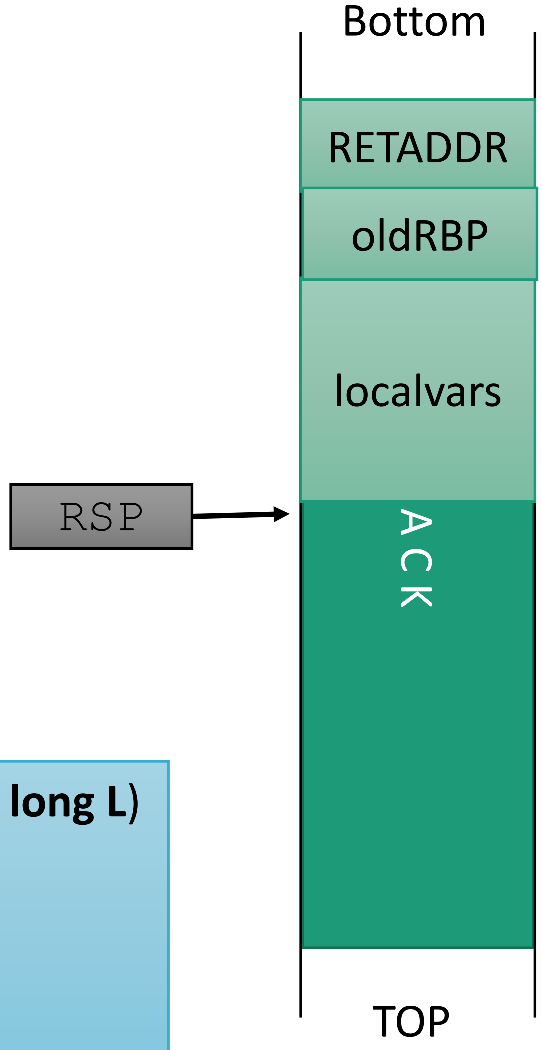
Frame pointers are optional

```
gcc -fomit-frame-pointer test.c
```

Function Arguments

```
void F1()
{
        int a;
        char buf[16];
        unsigned long l;
        ....
        F2(a, buf, l);
        ...
}
```

```
void F2(int A, char *BUF unsigned long L)
{
        int i;
        ...
}
```



Calling Conventions

Defines the standard for passing arguments

Caller and callee need to agree

Enforced by compiler

Important when using 3rd party libraries

Different styles ↔ different advantages

Popular Conventions

cdecl (mostly 32-bit)

Arguments are passed on the stack

- Pushed right to left

eax, edx, ecx are caller saved

- callee can overwrite without saving

ebx, esi, edi are callee saved

- callee must ensure they have same value on return

eax used for function return value

System V AMD64 ABI

Arguments are passed using registers

- First 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9

RBP, RBX, and R12–R15 are callee saved

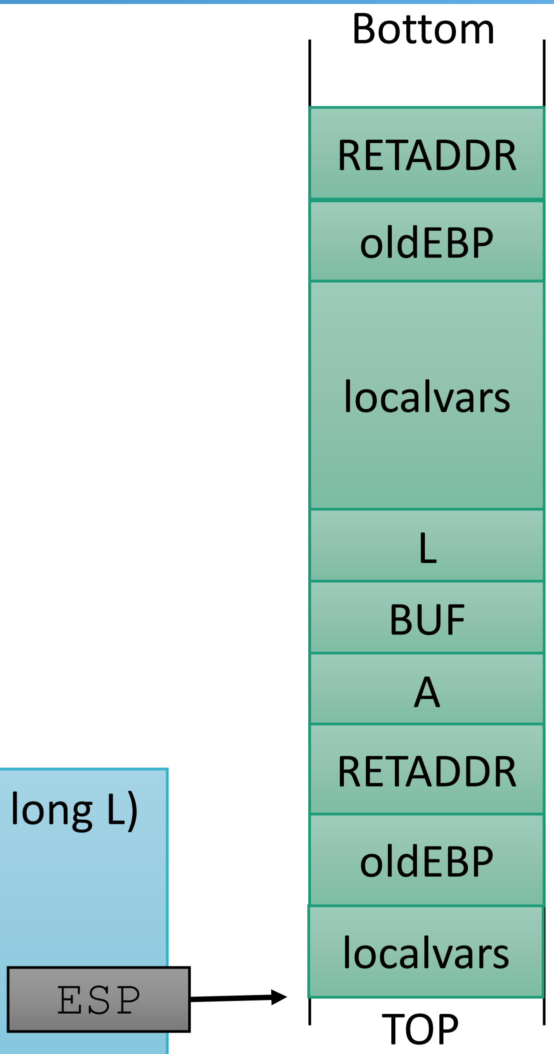
RAX used for function return

https://en.wikipedia.org/wiki/X86_calling_conventions

cdecl Example

```
void F1()
{
        int a;
        char buf[16];
        unsigned long l;
        ....
        F2(a, buf, l);
        ...
}
```

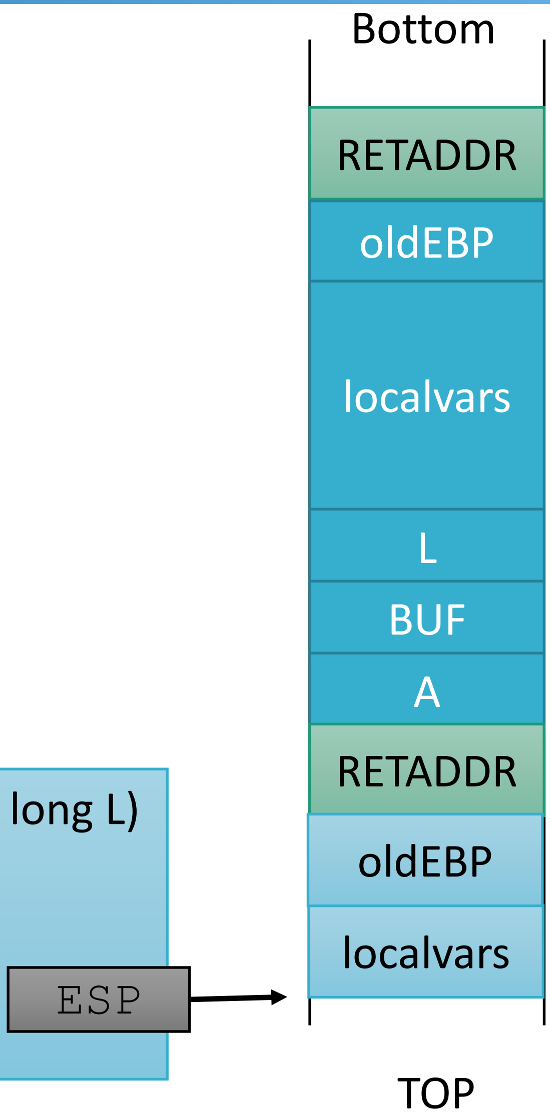
```
void F2(int A, char *BUF unsigned long L)
{
        int i;
        ...
}
```



cdecl Example

```
void F1()
{
        int a;
        char buf[16];
        unsigned long l;
        ….
        F2(a, buf, l);
        …
}
```

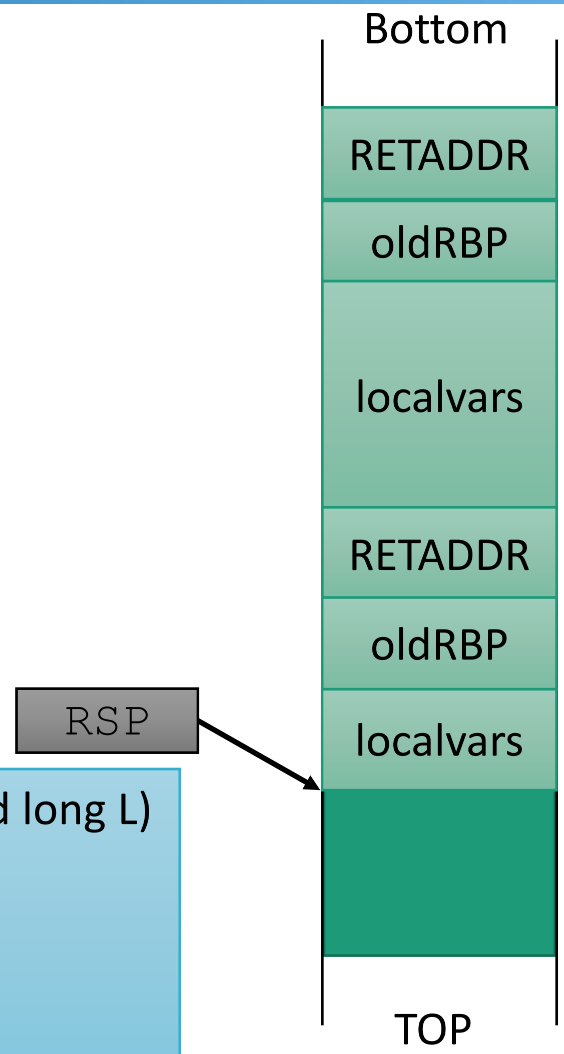
```
void F2(int A, char *BUF unsigned long L)
{
        int i;
        …
}
```



System V AMD64 ABI Example

```
void F1()
{
        int a;
        char buf[16];
        unsigned long l;
        ....
        F2(a, buf, l);
        ...
}
```

```
void F2(int A, char *BUF unsigned long L)
{
        int i;
        ...
}
```



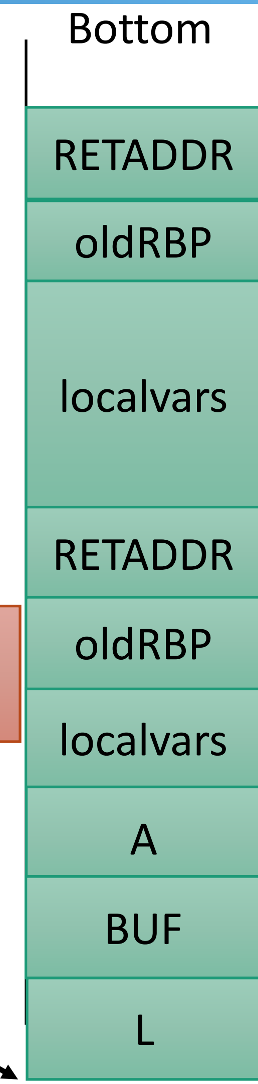
Register/Argument Spilling

```
void F1()
{
        int a;
        char buf[16];
        unsigned long l;
        ....
        F2(a, buf, l);
        ...
}
```

Arguments may be also saved in the stack by the compiler

```
void F2(int A, char *BUF unsigned long L)
{
        int i;
        ...
}
```

RSP



Variable Number of Arguments

Used in variadic functions, like

```
int printf(const char *format, ...);
```

Arguments passed in the stack

- Order right-to-left

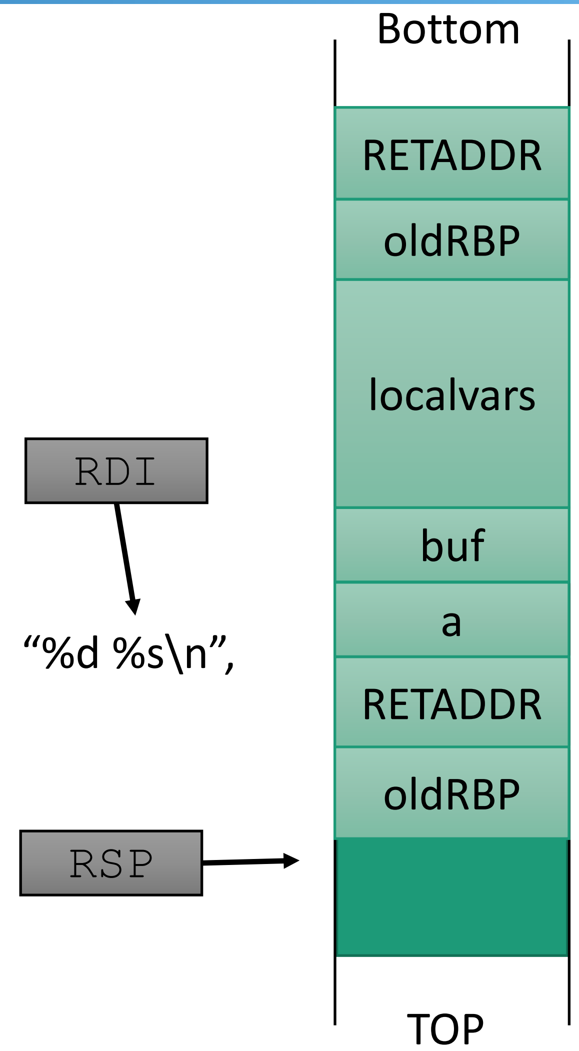
Only caller knows exact number of arguments

- Caller responsible for cleaning

Example on 64-bit

```
void F1()
{
        int a;
        char buf[16];
        unsigned long l;
        ....
        printf("%d %s\n", a, buf);
        ...
}
```

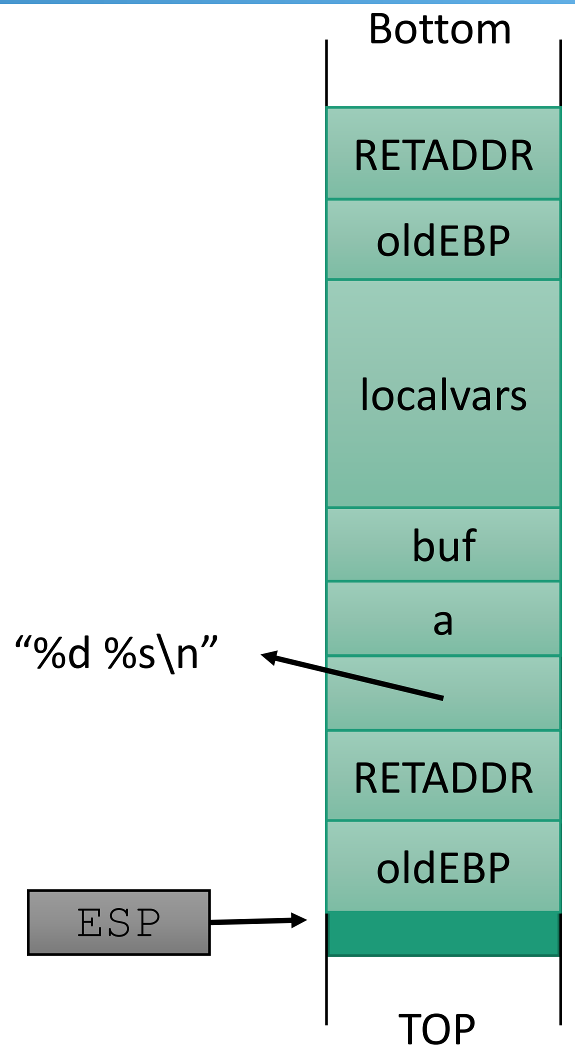
```
void printf(const char *fmt, ...)
{
        ...
}
```



Example on 32-bit

```
void F1()
{
        int a;
        char buf[16];
        unsigned long l;
        ....
        printf("%d %s\n", a, buf);
        ...
}
```

```
void printf(const char *fmt, ...)
{
        ...
}
```

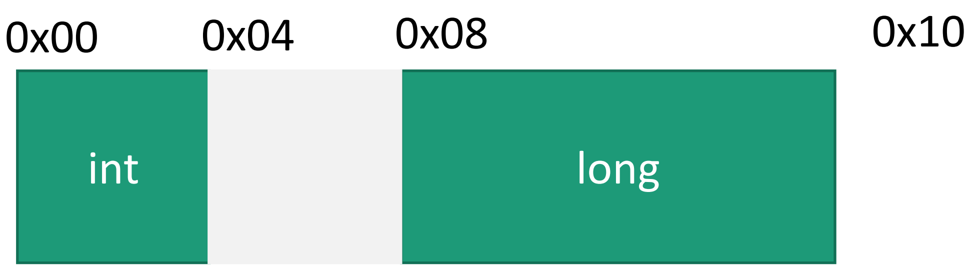


Alignment

CPUs like aligned data

- Better performance

Compilers try to align data



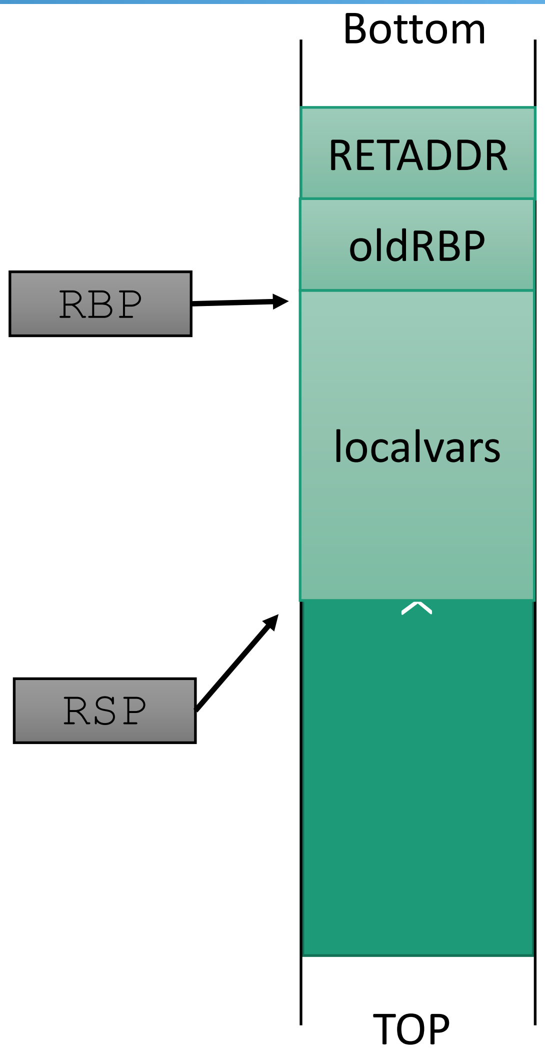
Accessing Stack Variables

With frame pointer

```
mov     -0x18(%rbp),%eax
```

With stack pointer

```
mov     0xc(%rsp),%eax
```



Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety

Variables and Memory

C, C++

int n = 0xdeadbeef;

char str[16] = “Hello”;

Python

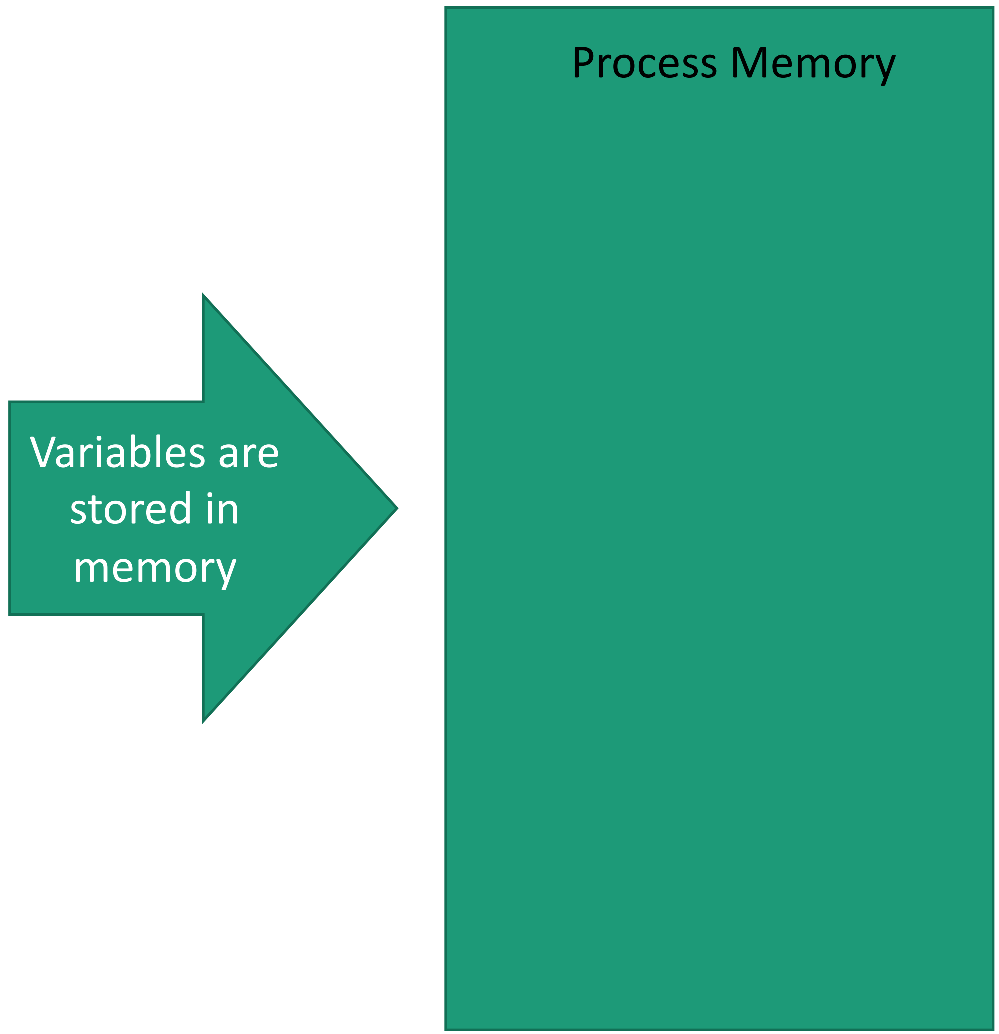
n = 0xdeadbeef

str = “Hello”

Java

int n = 0xdeadbeef;

String str = “Hello”;



Variables and Memory

C, C++

```
int n = 0xdeadbeef;

char str[16] = "Hello";
```

Python

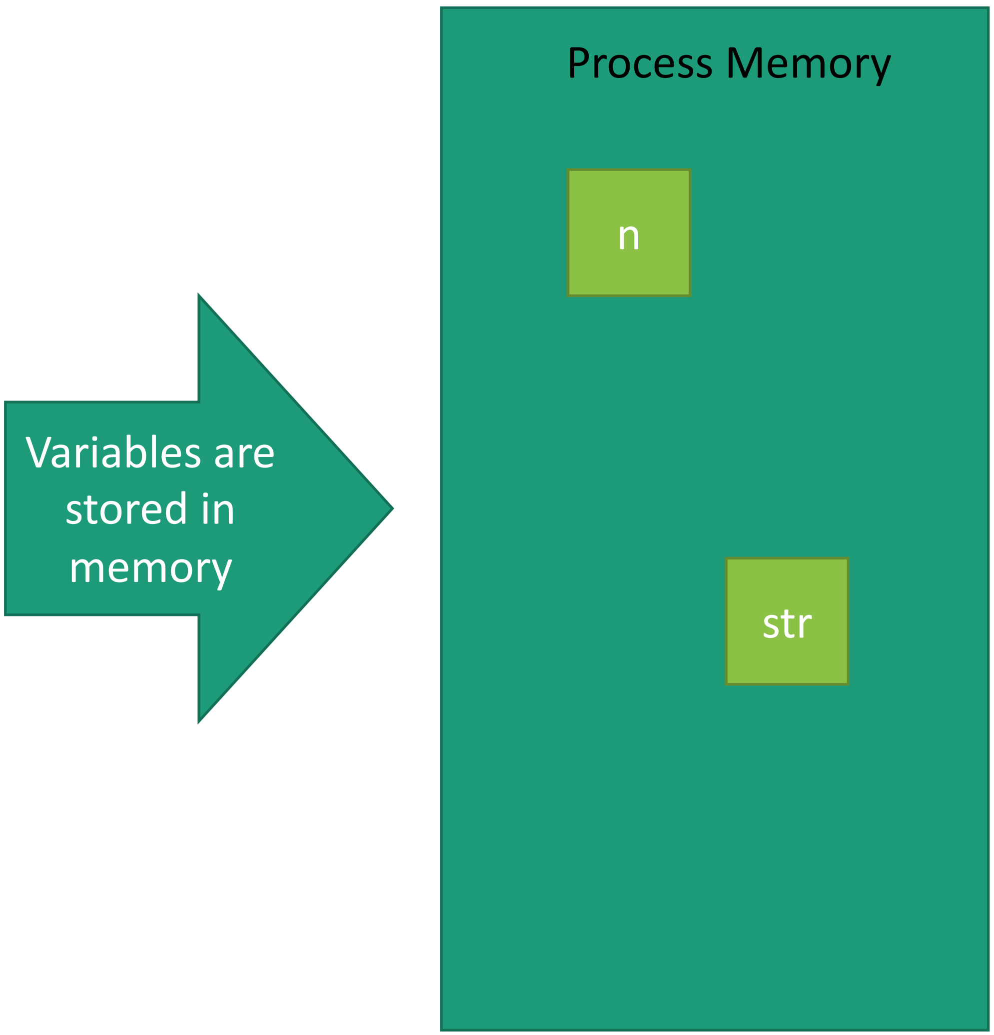
```
n = 0xdeadbeef

str  = "Hello"
```

Java

```
int n = 0xdeadbeef;

String str = "Hello";
```



Variables and Memory

The location and actual storage characteristics are hidden by the programmer

Python

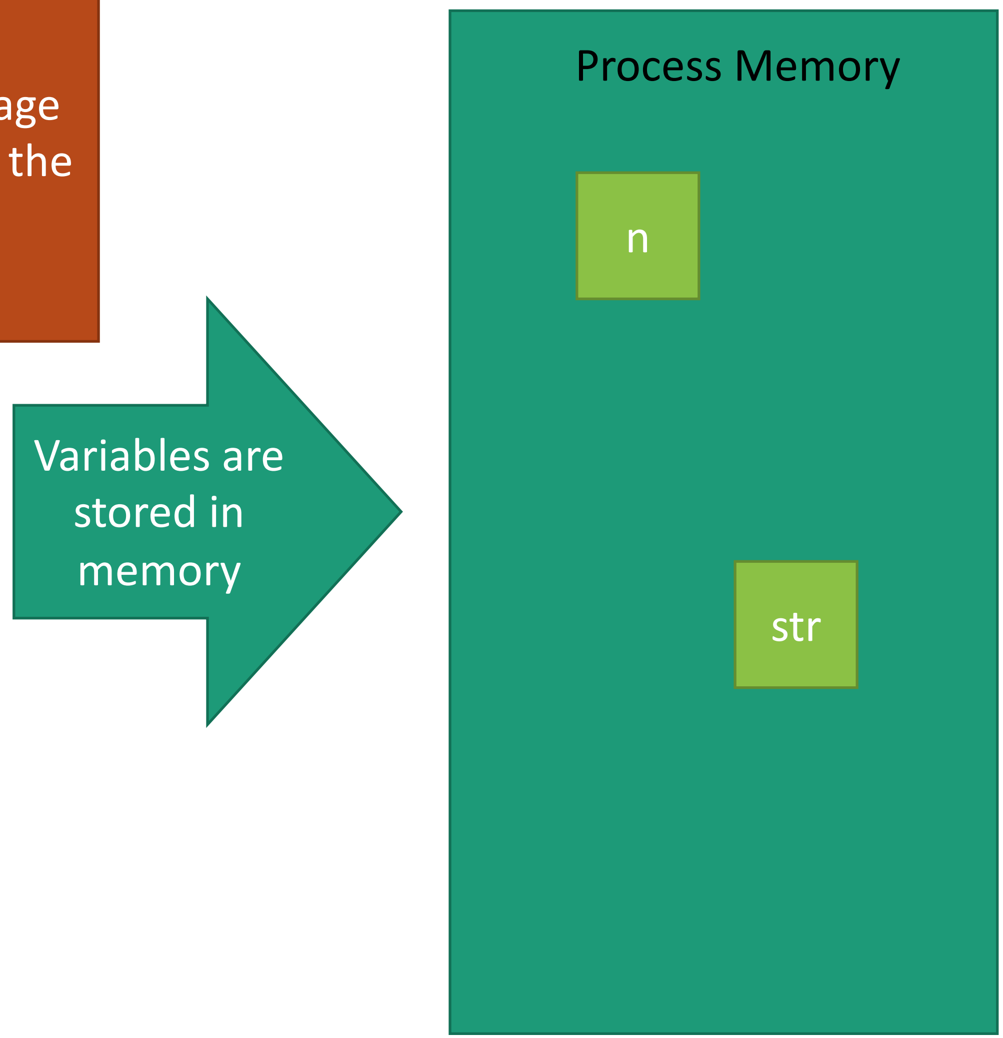
```
n = 0xdeadbeef

str  = “Hello”
```

Java

```
int n = 0xdeadbeef;

String str = “Hello”;
```



Variables and Memory

C, C++

```
int n = 0xdeadbeef;

char str[16] = "Hello";
```

The location and storage characteristics are transparent to the programmer

Variables are stored in memory

Process Memory

n

str

Variables in C and C++

int n = 0xdeadbeef;

0xef
0xbe
0xed
0xde

char str[16] = “Hello”;

It’s All Bytes

‘H’
‘e’
‘l’
‘l’
‘o’
0x00
~~

~~

10 uninitialized bytes

Pointers in C and C++

```
int n = 0xdeadbeef;
int  n_p = &n;
```

0xef
0xbe
0xed
0xde

```
char str[16] = “Hello”;
char *str_p = str;
```

‘H’
‘e’
‘l’
‘l’
‘o’
0x00
~~

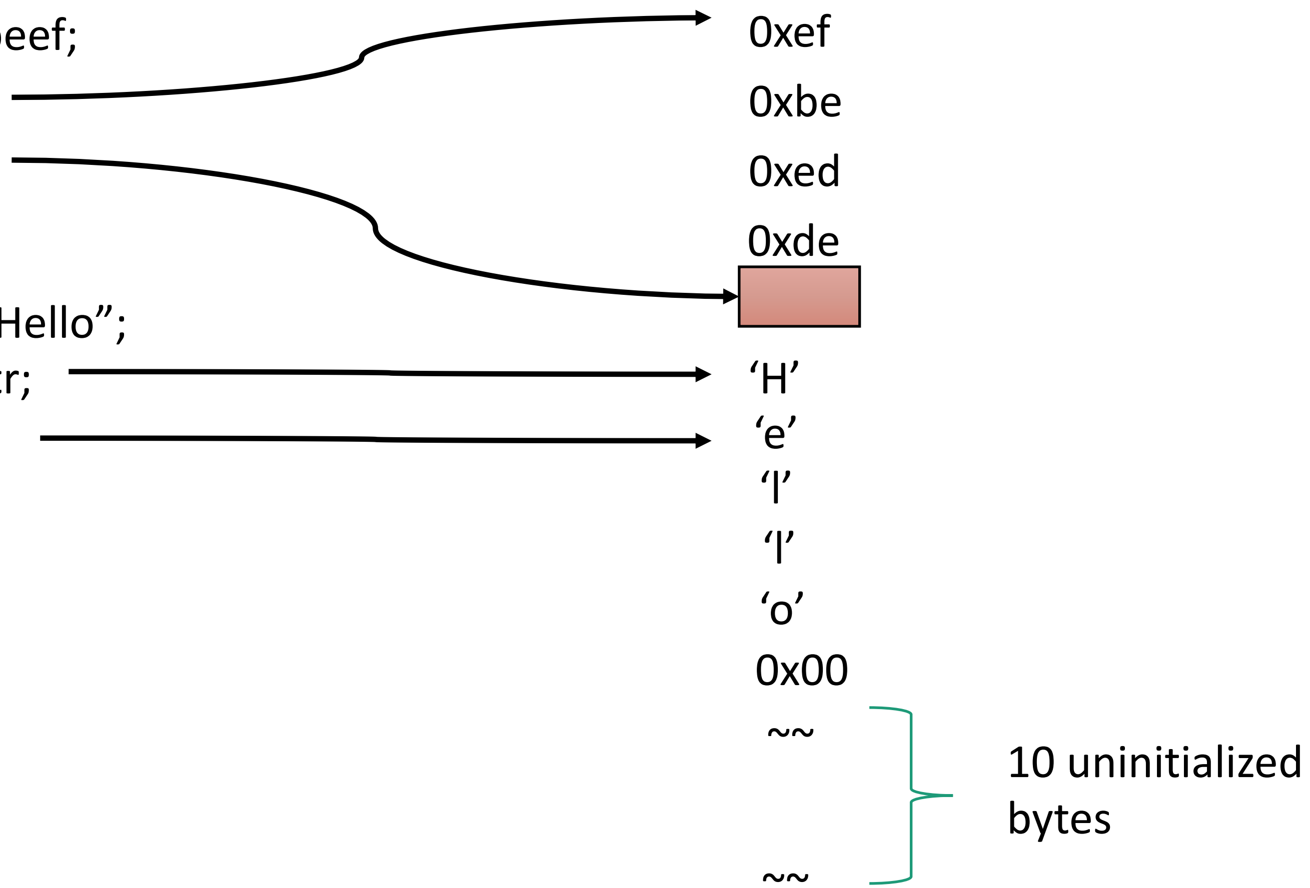
~~

10 uninitialized bytes

Pointers in C and C++

```
int n = 0xdeadbeef;
int  n_p = &n;
n_p++;

char str[16] = “Hello”;
char *str_p = str;
str_p++;
```



Pointers in C and C++

```
int n = 0xdeadbeef;
int  n_p = &n;
n_p++;
long n_addr = (long)&n;

char str[16] = "Hello";
char *str_p = str;
str_p++;
long str_addr = (long)str;
```

0xef
0xbe
0xed
0xde

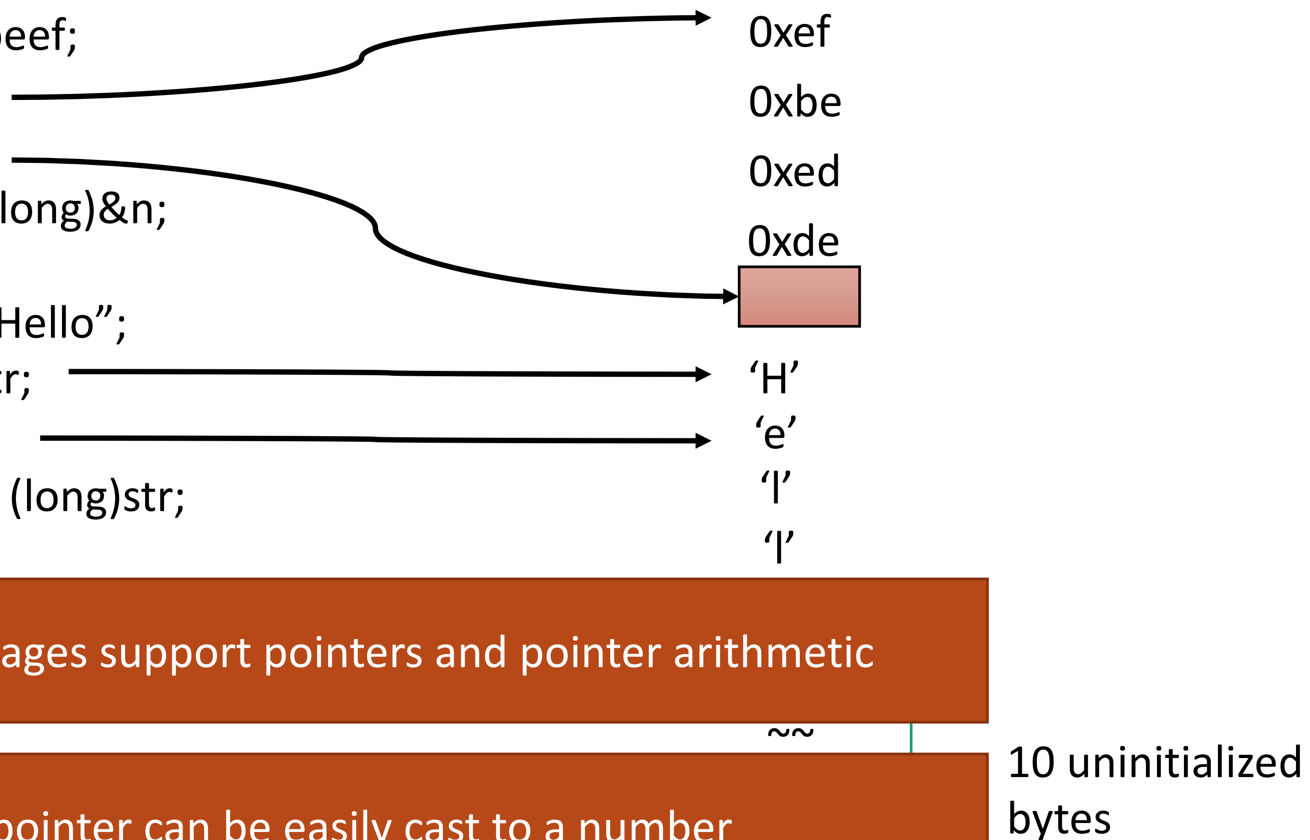
'H'
'e'
'l'
'l'
'o'
0x00
~~
~~

10 uninitialized bytes

Pointers in C and C++

```
int n = 0xdeadbeef;
int  n_p = &n;
n_p++;
long n_addr = (long)&n;

char str[16] = “Hello”;
char *str_p = str;
str_p++;
long str_addr = (long)str;
```



The languages support pointers and pointer arithmetic

A pointer can be easily cast to a number

Pointer Example

```
char year[4] = ”2001”;

simple_function(year);

int simple_function(char *str)
{
        char *c;

        for (c = str; c != '\0'; c++) {
                if (*c == '0')
                        *c = '1';
        }
}
```

Pointer Example

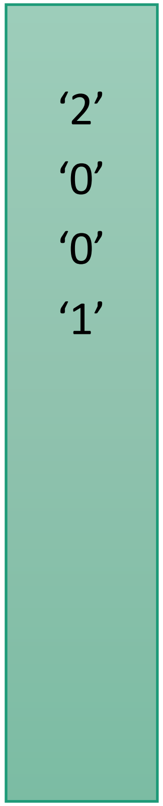
```
char year[4] = ”2001”;

simple_function(year);

int simple_function(char *str)
{
        char *c;

        for (c = str; c != '\0'; c++) {
                if (*c == '0')
                        *c = '1';
        }
}
```

Memory



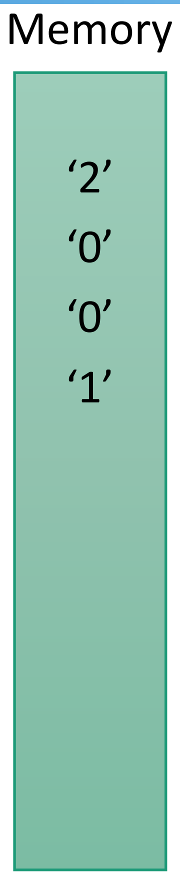
Pointer Example

C/C++ does not perform bounds checking to ensure accesses through remain within range or are performed on the correct memory object.
This task falls on the programmer.

C/C++ are memory unsafe!

```
{
        char *c;

        for (c = str; c != '\0'; c++) {
                if (*c == '0')
                        *c = '1';
        }
}
```



Can the Hardware Help?

Modern hardware has limited support for isolated memory coarsely

Memory is organized into pages

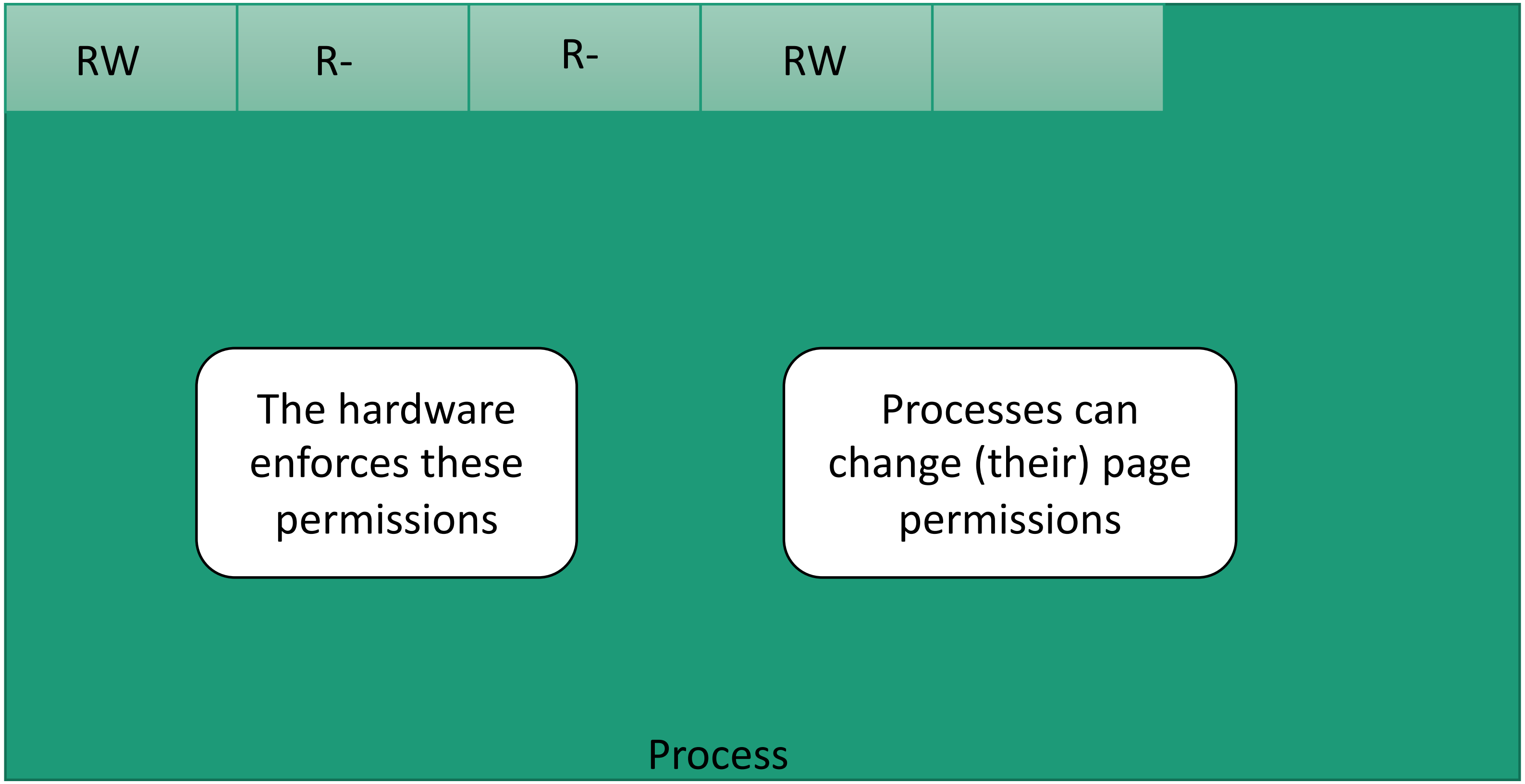
- Contiguous 4KB or 2MB chunks of memory

Each page can be configured as being readable or readable and writable

- All readable pages used to be executable
- Now they can be also marked as non-executable
  - We’ll get back to this later

Paging and Permissions

Page permissions: readable (implicit, writable



Segment Permissions