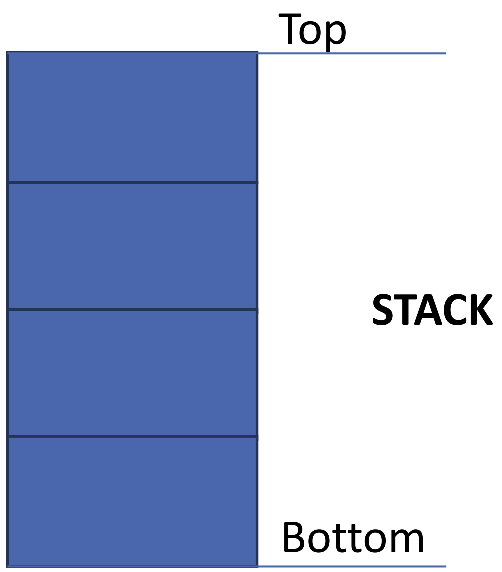
CS 576 – Systems Security
Smashing the Stack

Georgios (George) Portokalidis

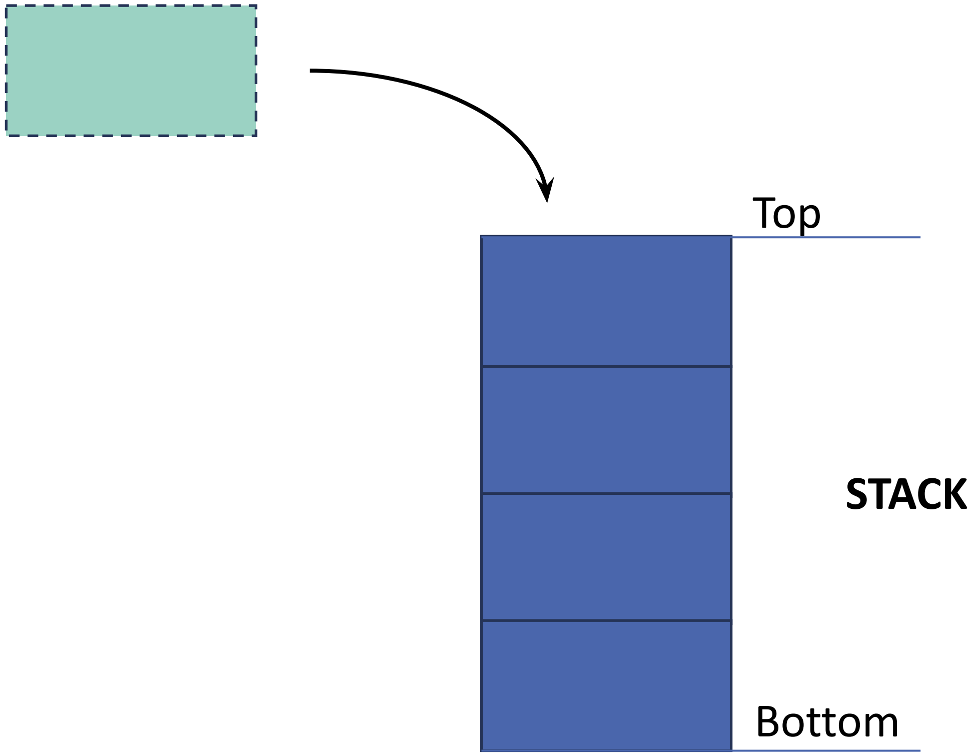
How Do Function Calls Work

Stack Data Structure

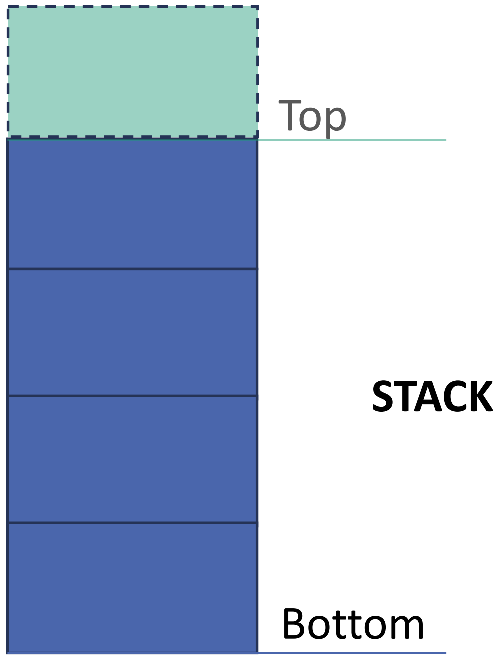
- Stack plays a crucial role in supporting functions
  - Follows last-in first-out semantic



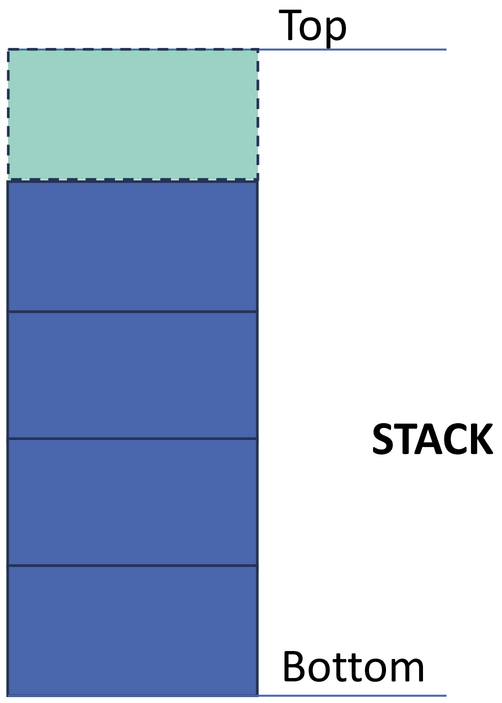
Stack Operation Push



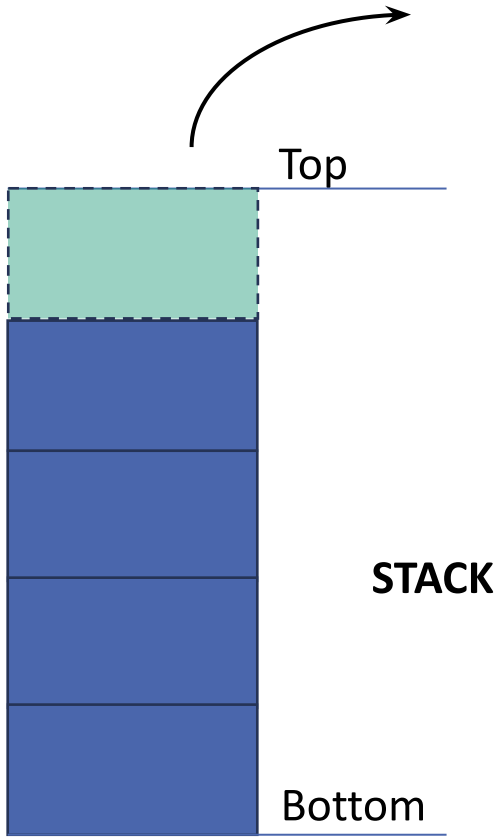
Stack Operation Push



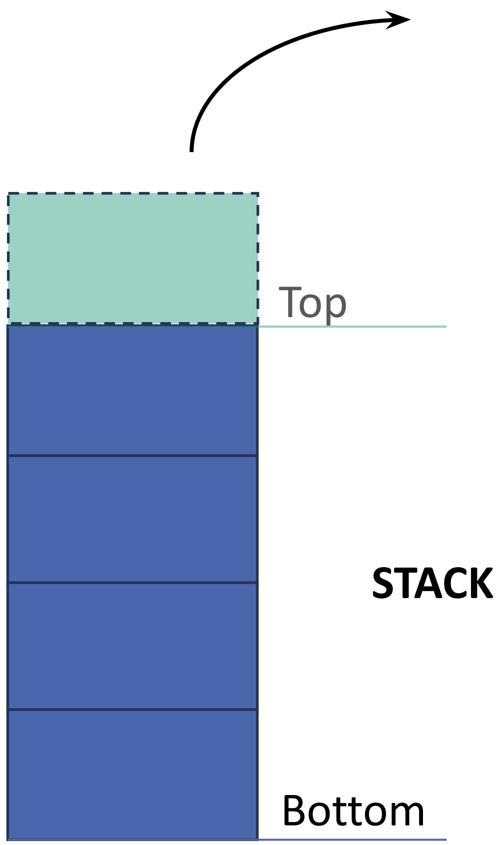
Stack Operation Push



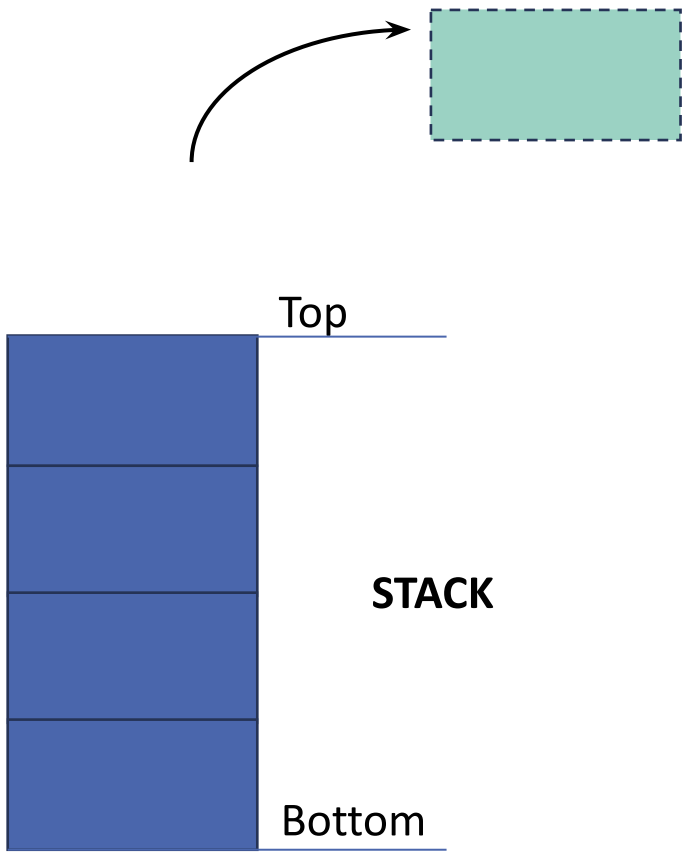
Stack Operation Pop



Stack Operation Pop

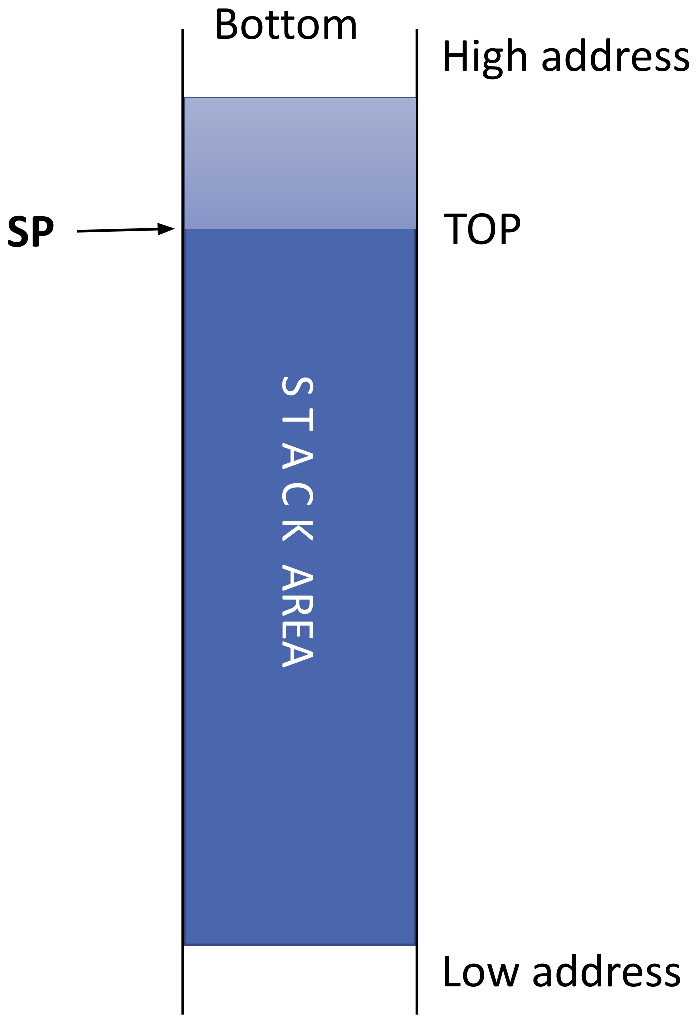


Stack Operation Pop



The Stack Pointer (SP)

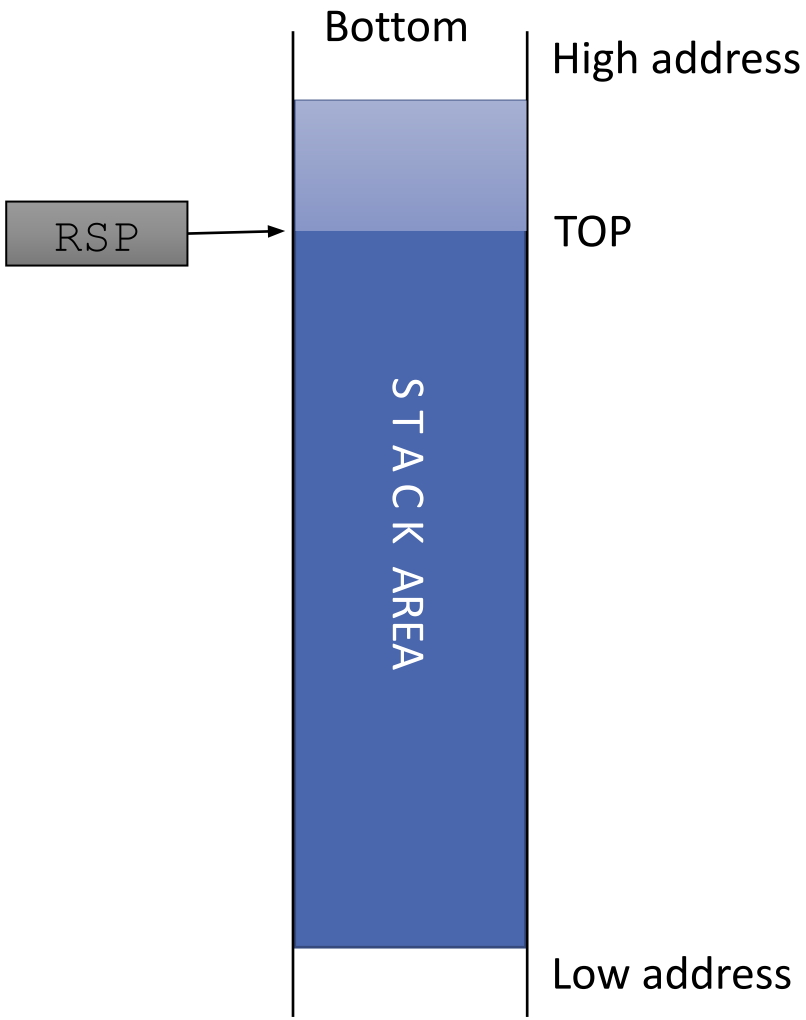
The stack pointer points to the first element in the stack (the top).



The Stack Pointer (SP)

The stack pointer points to the first element in the stack (the top).

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

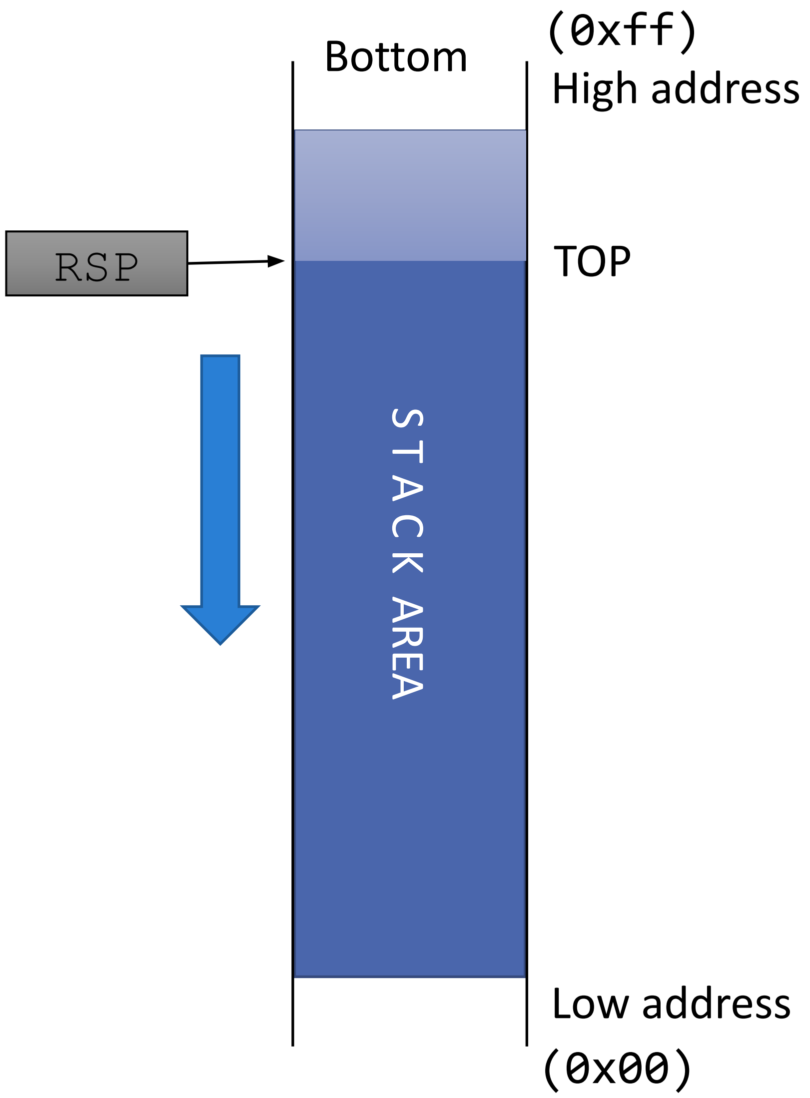


The Stack Pointer (SP)

The stack pointer points to the first element in the stack (the top).

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

The stack grows towards lower addresses



Simple Function Call

C

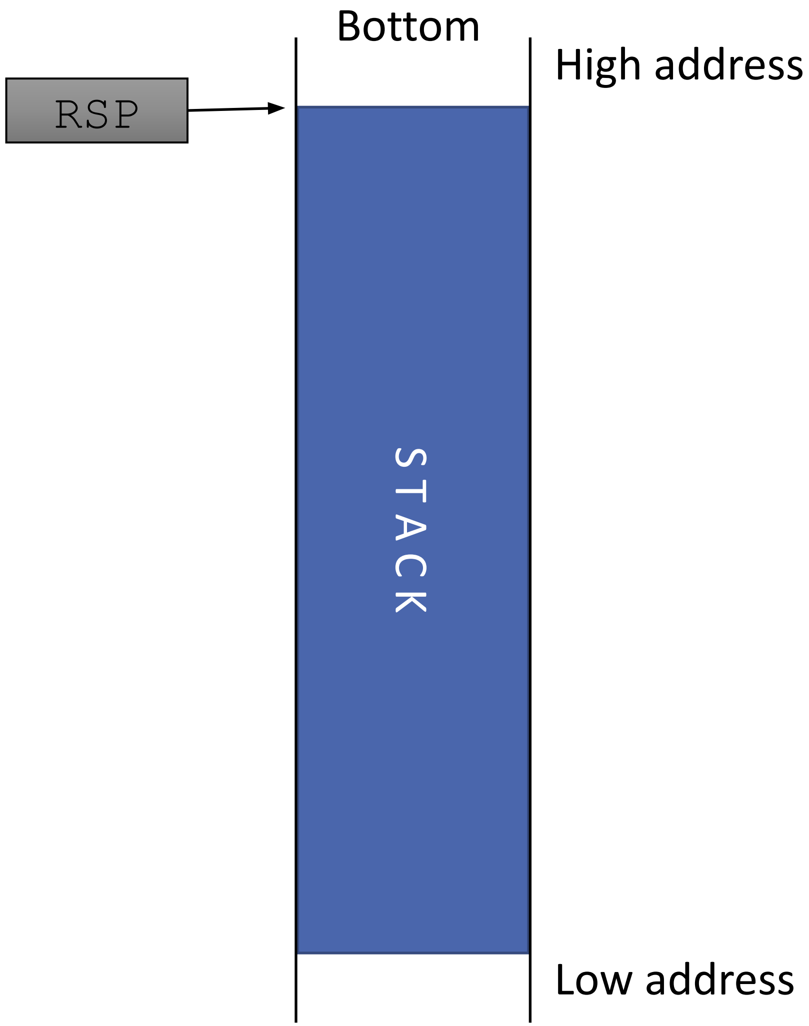
```
…
F1();
…
```

```
void F1()
{
      ….
}
```

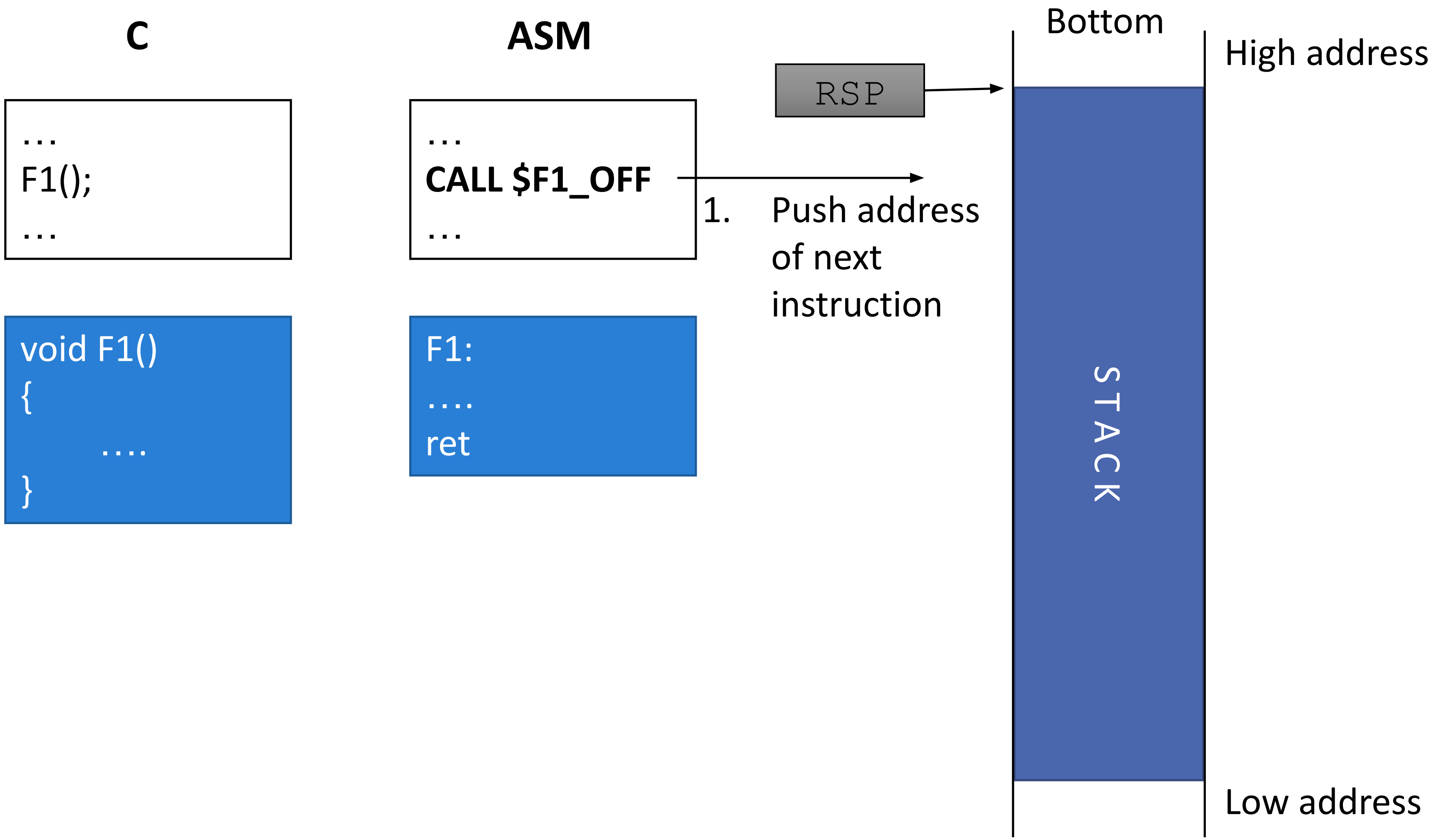
ASM

```
…
CALL $F1_OFF
…
```

```
F1:
….
ret
```



Simple Function Call



Simple Function Call

C

```
…
F1();
…
```

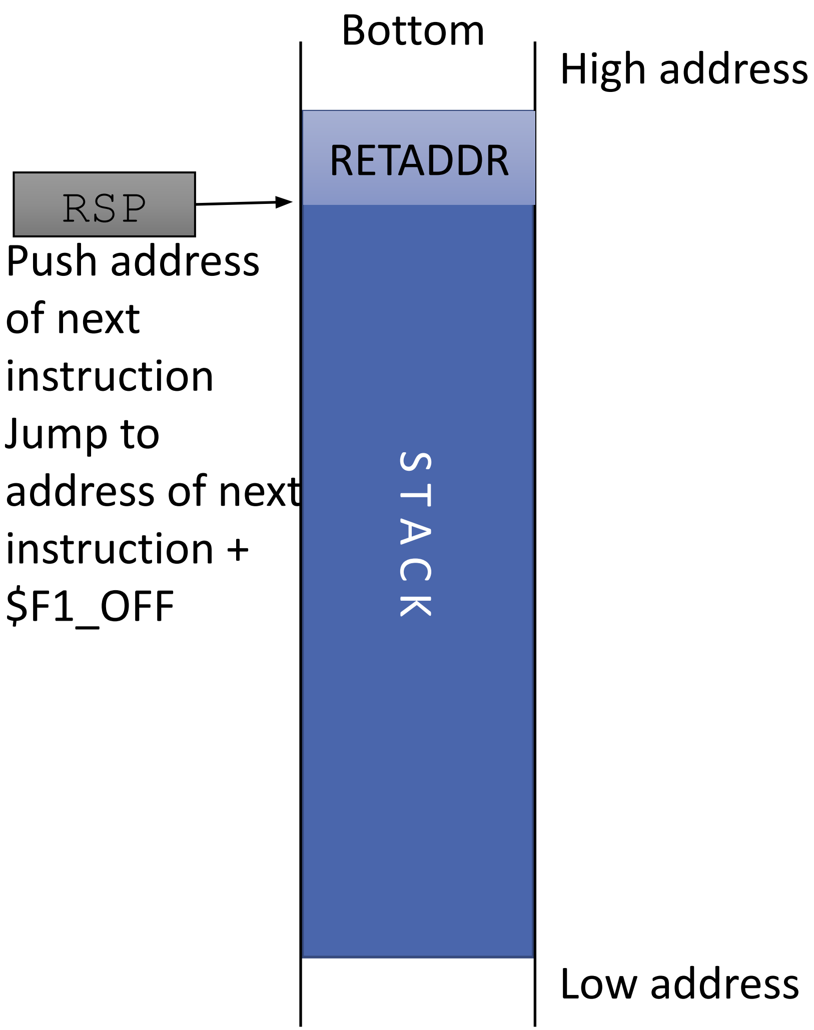
```
void F1()
{
        ….
}
```

ASM

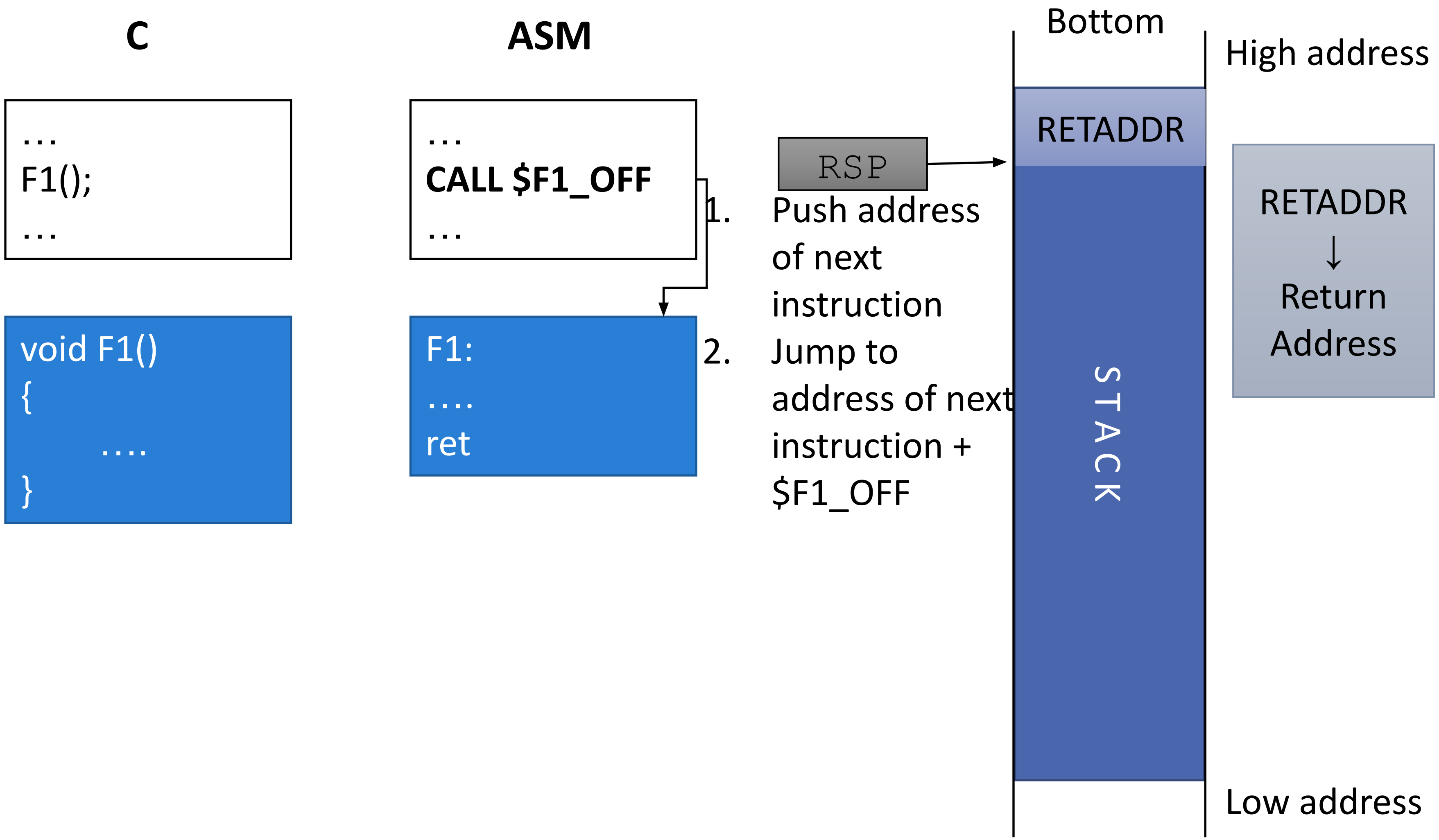
```
…
CALL $F1_OFF
…
```

```
F1:
….
ret
```

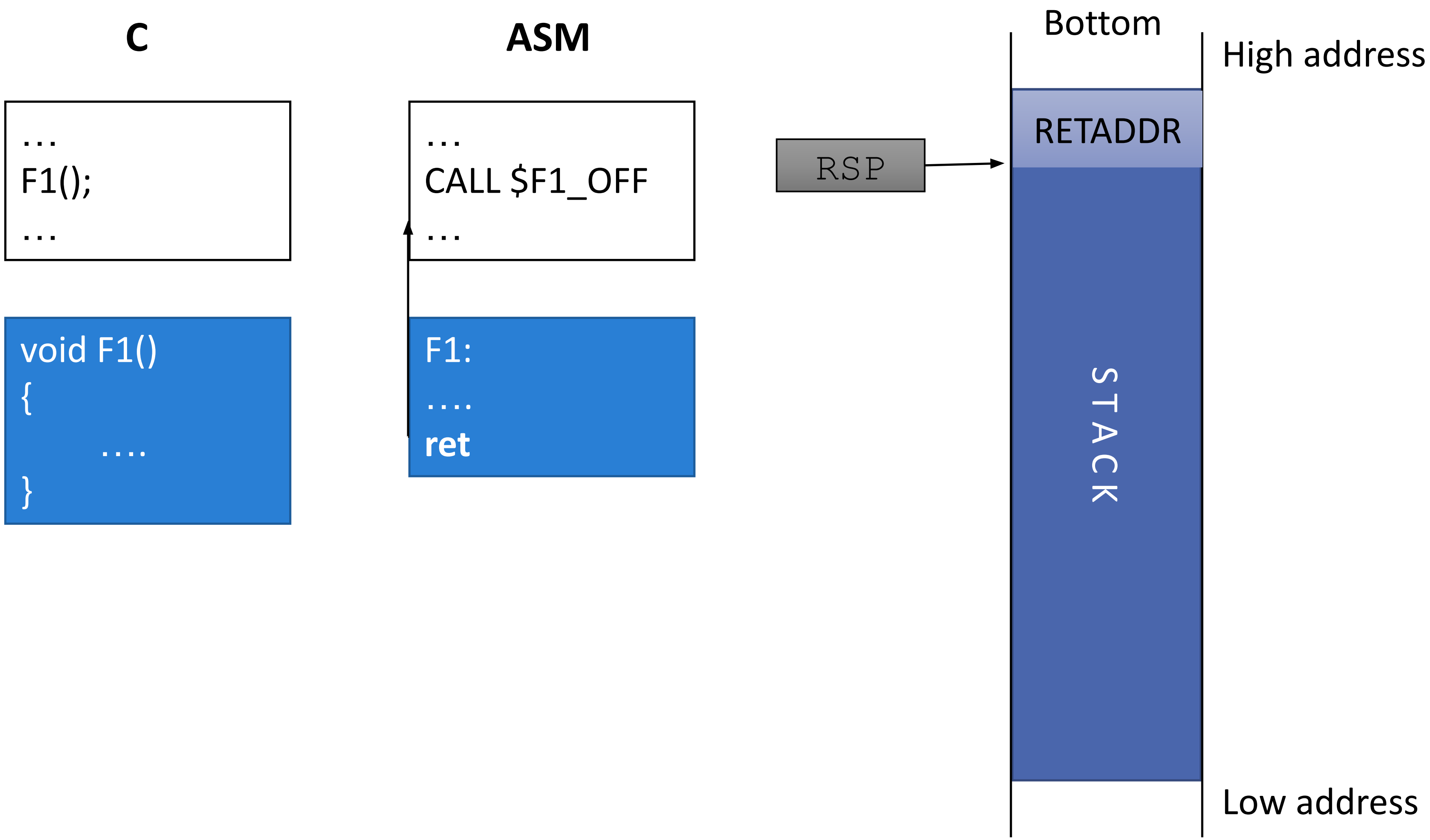
1. Push address of next instruction
2. Jump to address of next instruction + $F1_OFF



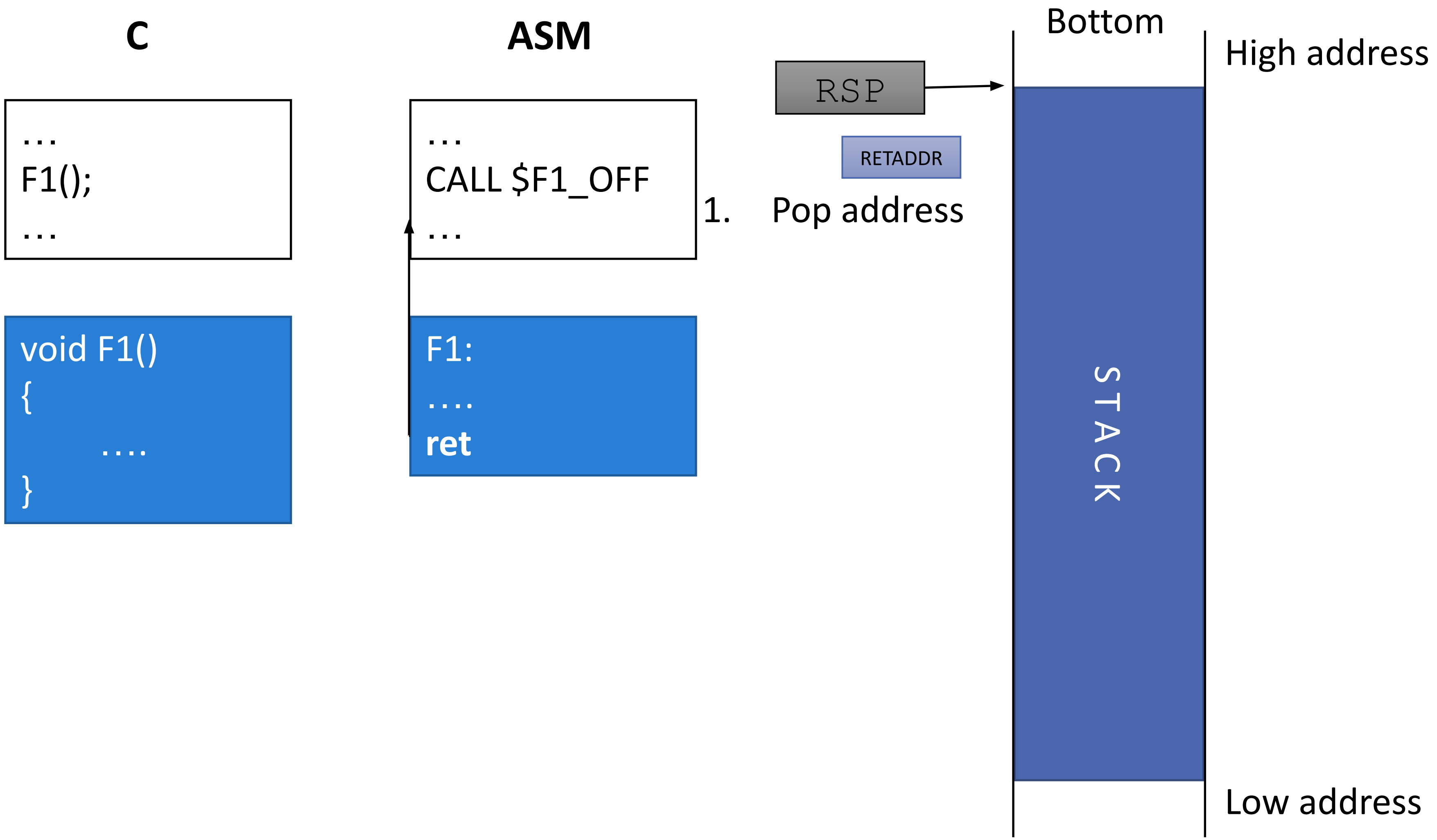
Simple Function Call



Simple Function Return



Simple Function Return



Simple Function Return

C

```
…
F1();
…
```

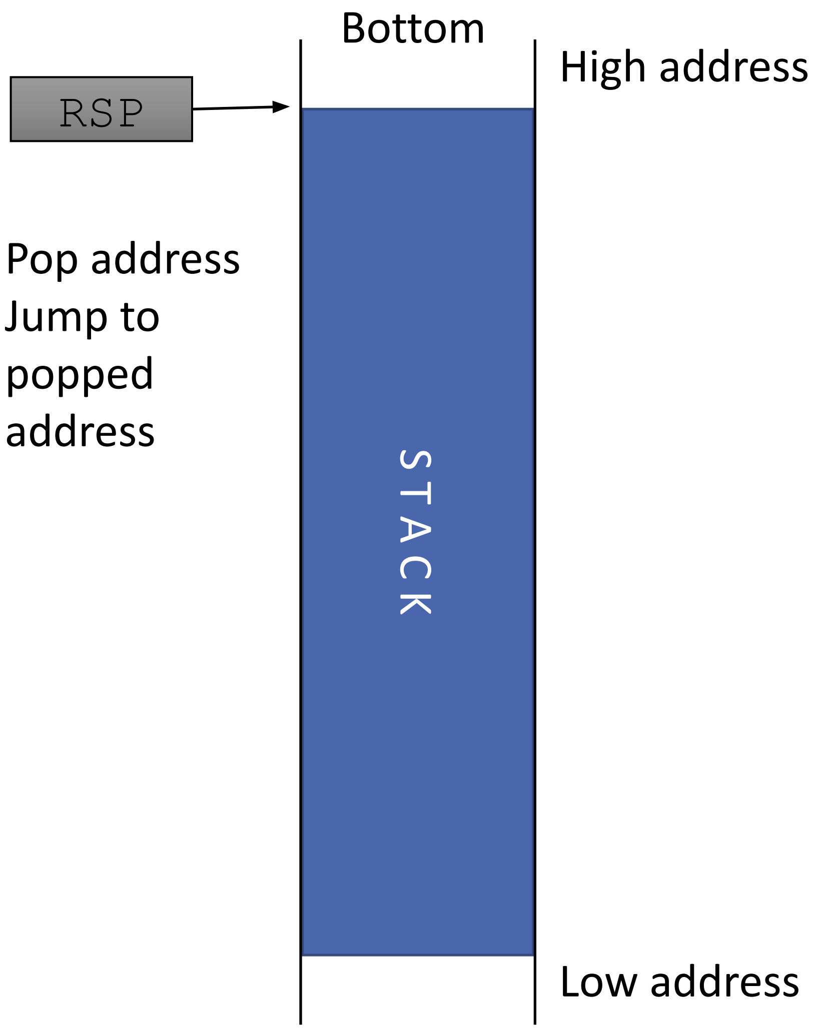
```
void F1()
{
      ….
}
```

ASM

```
…
CALL $F1_OFF
…
```

```
F1:
….
ret
```

1. Pop address
2. Jump to popped address



Function Calls and Returns

Calling a function (the callee)

- CALL instruction
  - Pushes next_ins_addr on stack and transfers control to address described by operand
- Most common syntax: CALL OFFSET
  - Target is next_ins_addr + OFFSET

Returning to caller

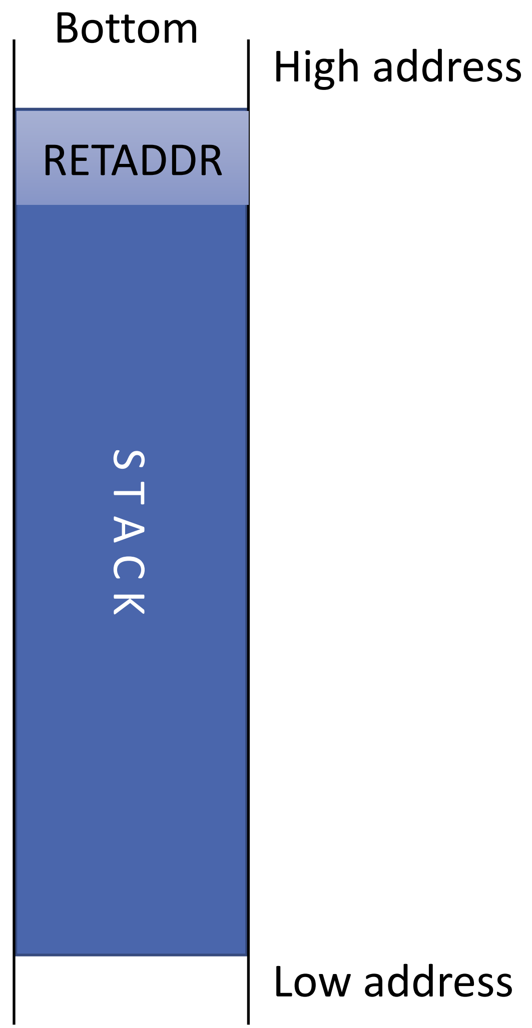
- RET instruction
  - Pop return address from stack and transfers control to it

```
CALL tgt  push next_ins_addr; jmp tgt
RET  pop retaddr; jmp retaddr
```

`call` and `ret` implicitly use the SP

The Stack Is Used…

- …to store the return address of caller functions
  - Code pointers!
- …to store local variables
  - Aka stack variables



Local Variables

C

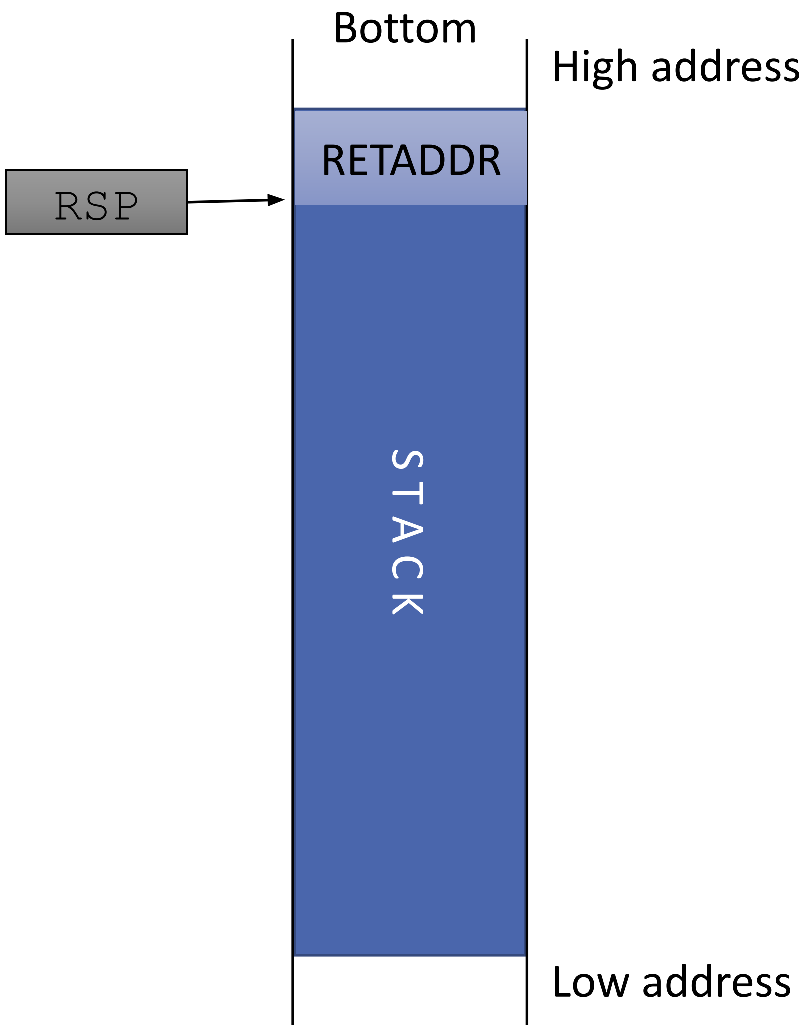
```
…
F1();
…
```

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

ASM

```
…
CALL $F1_OFF
…
```

```
F1:
...

...
ret
```



Local Variables

C

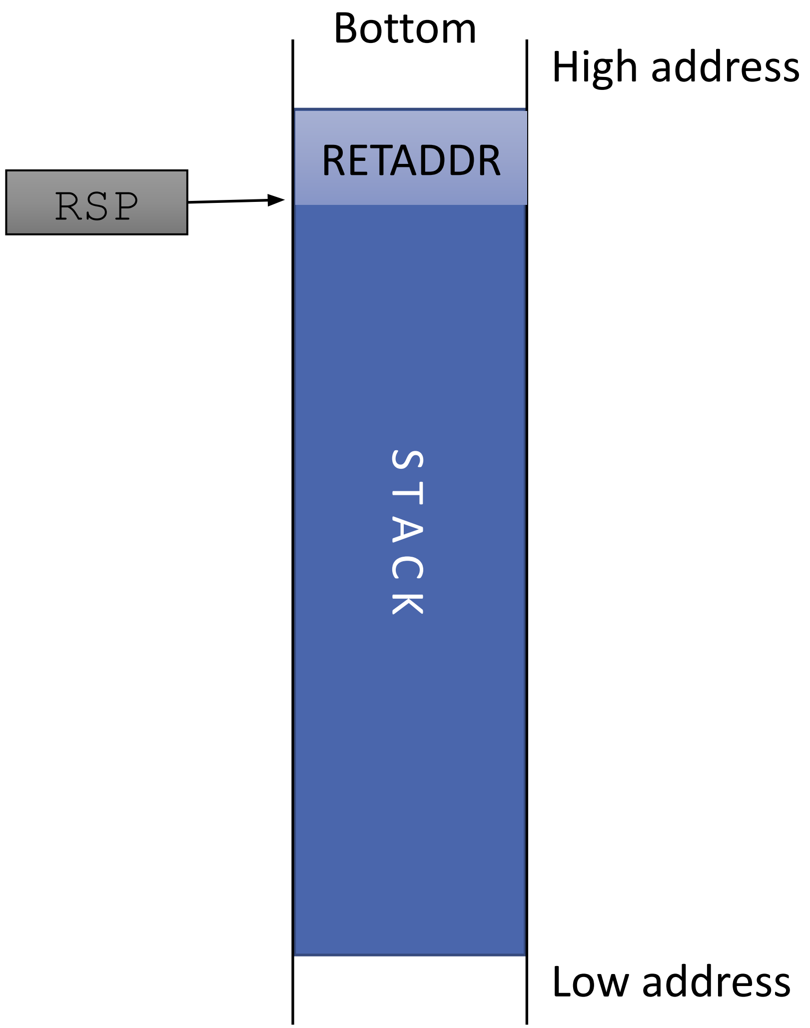
```
…
F1();
…
```

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

ASM

```
…
CALL $F1_OFF
…
```

```
F1:

subq $48,%rsp

...
ret
```



Local Variables

C

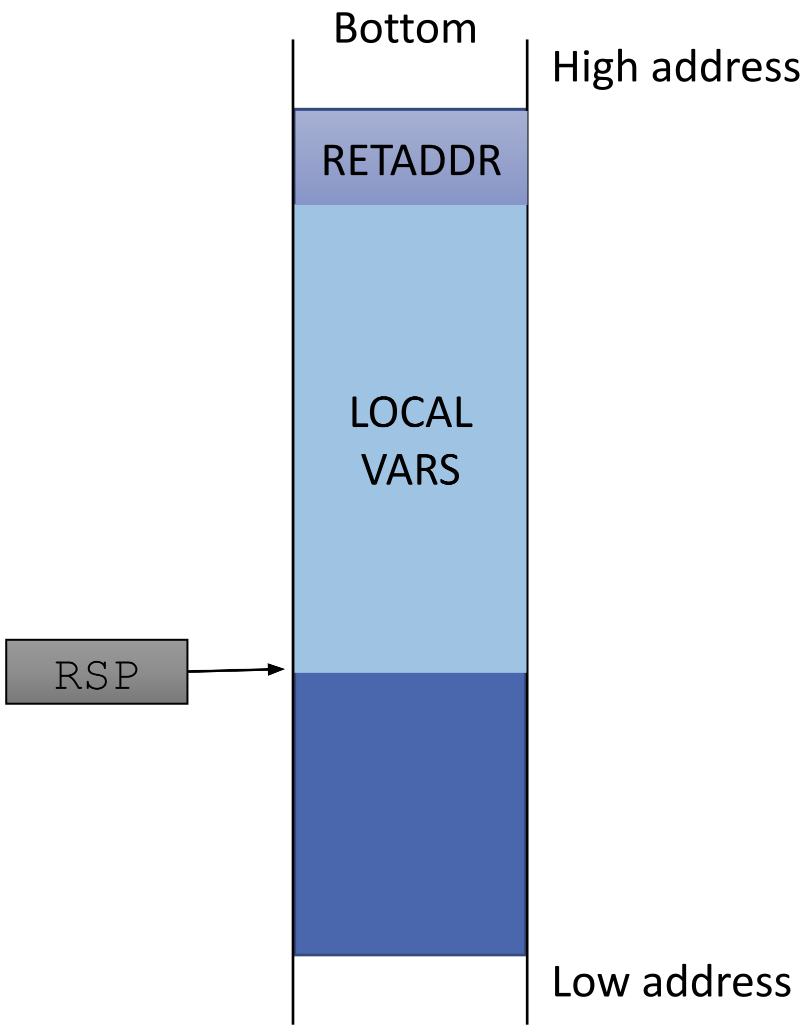
```
…
F1();
…
```

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

ASM

```
…
CALL $F1_OFF
…
```

```
F1:

subq $48,%rsp

...
ret
```



Local Variables

C

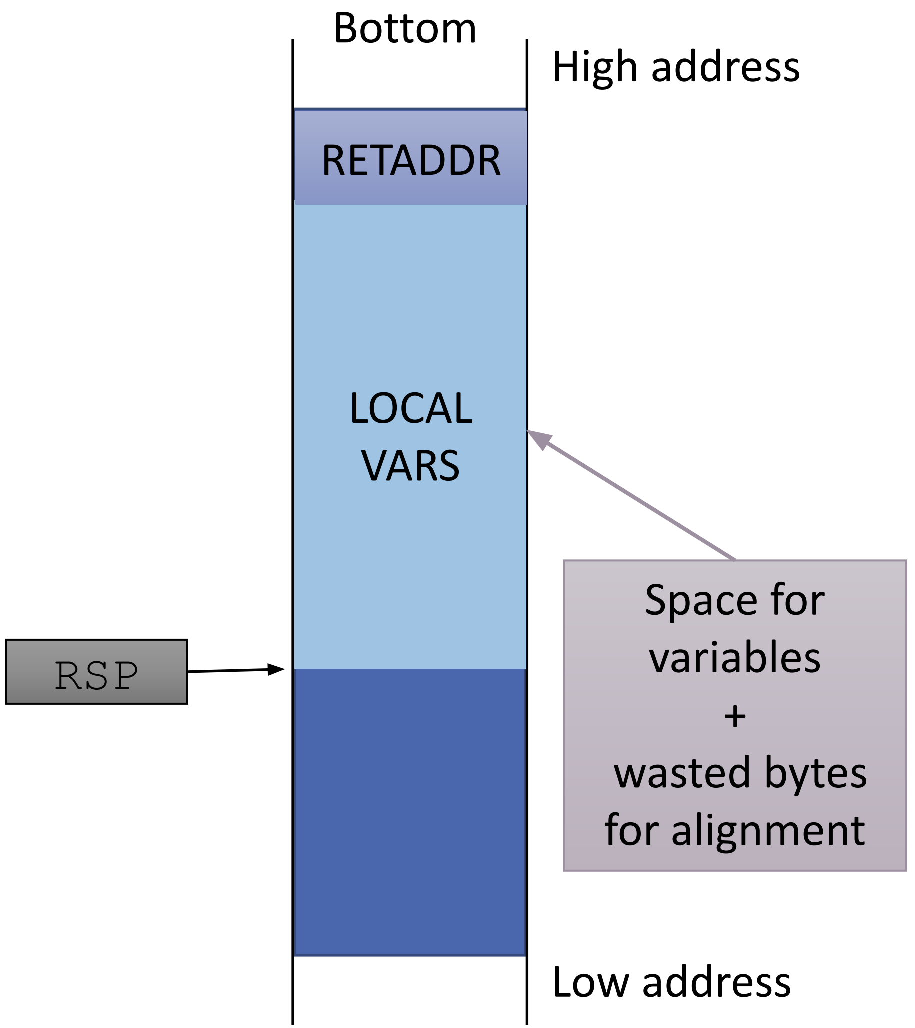
```
…
F1();
…
```

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

ASM

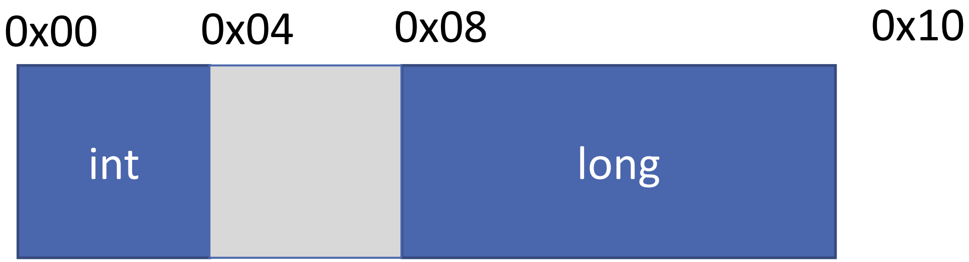
```
…
CALL $F1_OFF
…
```

```
F1:

subq $48,%rsp

...
ret
```



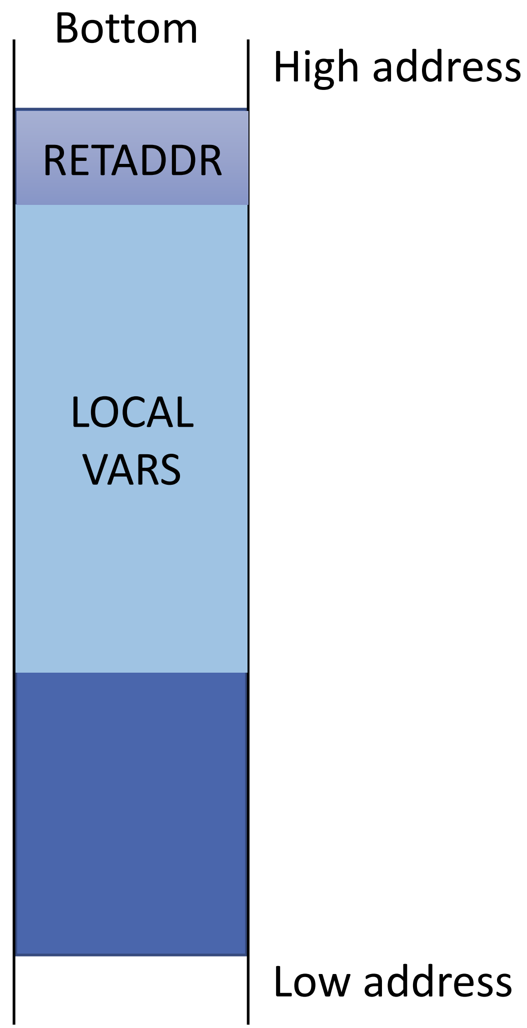
Alignment

- CPUs like aligned data
  - Better performance
- Compilers try to align data



The Stack Is Used…

- …to store the return address of caller functions
  - Code pointers!
- …to store local variables
  - Aka stack variables
- …to pass function arguments



Calling Conventions

- Defines the standard for passing arguments
- Caller and callee need to agree on the standard
- Enforced by compiler
- Important when using 3rd party libraries
  - Hence, also referred to as the Application Binary Interface (ABI)
- Different styles ↔ different advantages

```
...
F1(arg1, arg2, arg3);
...
```

??

```
void F1(short a1, long a2, char *a3)
{
  int a;
  char buf[16];
  unsigned long l;
  ...
}
```

Popular conventions:

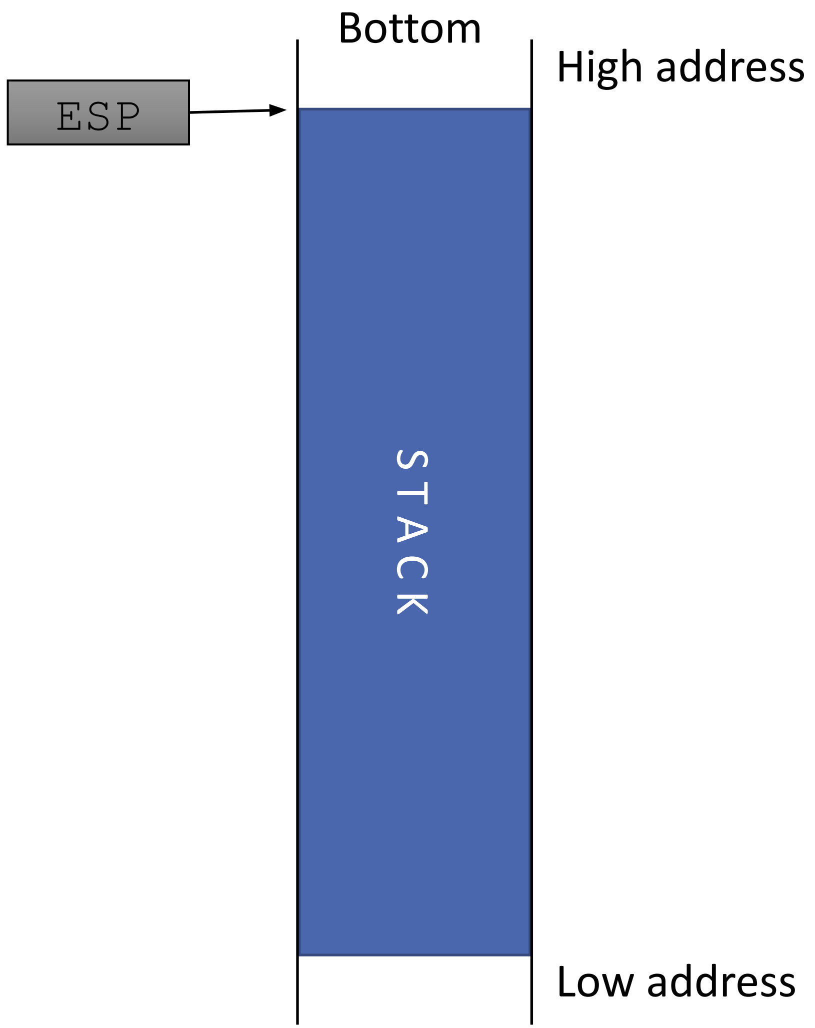
- cdecl (32-bit)
- System V AMD64 ABI

cdecl

- Arguments are passed on the stack
  - Pushed right to left

```
…
F1(0xff, UINT_MAX, argv[0]);
…
```

```
...
pushl   (%eax)
pushl   $-1
pushl   $255
call    F1
...
```

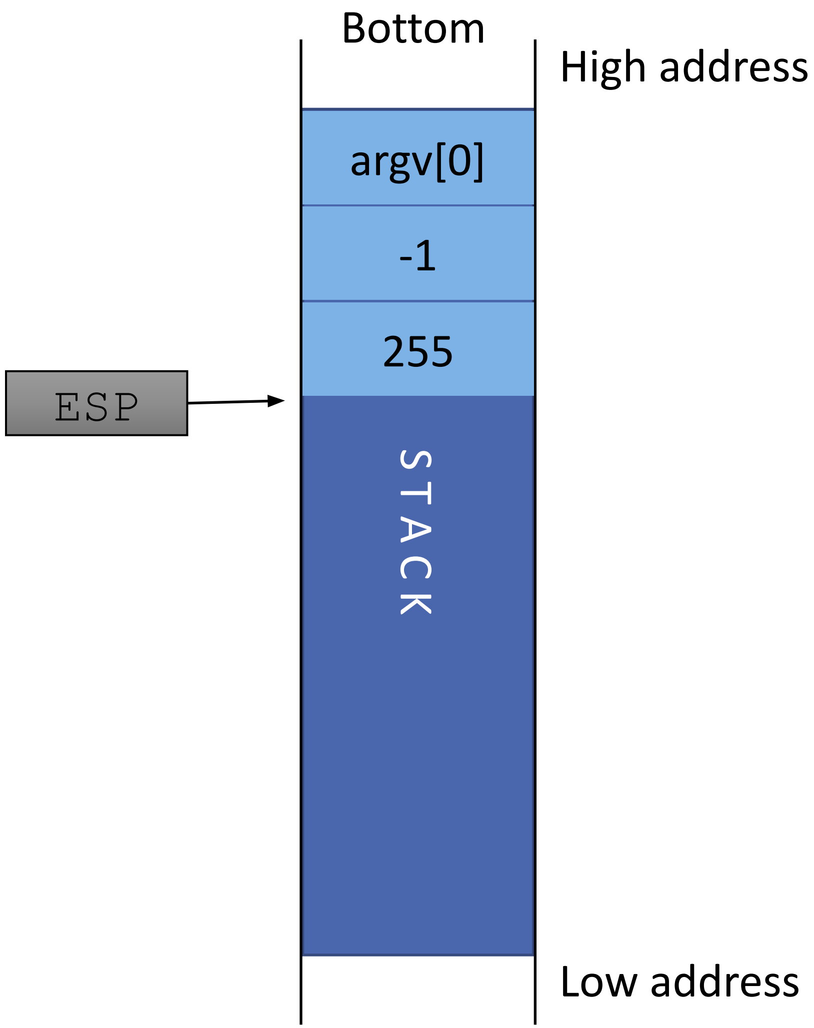


cdecl

- Arguments are passed on the stack
  - Pushed right to left

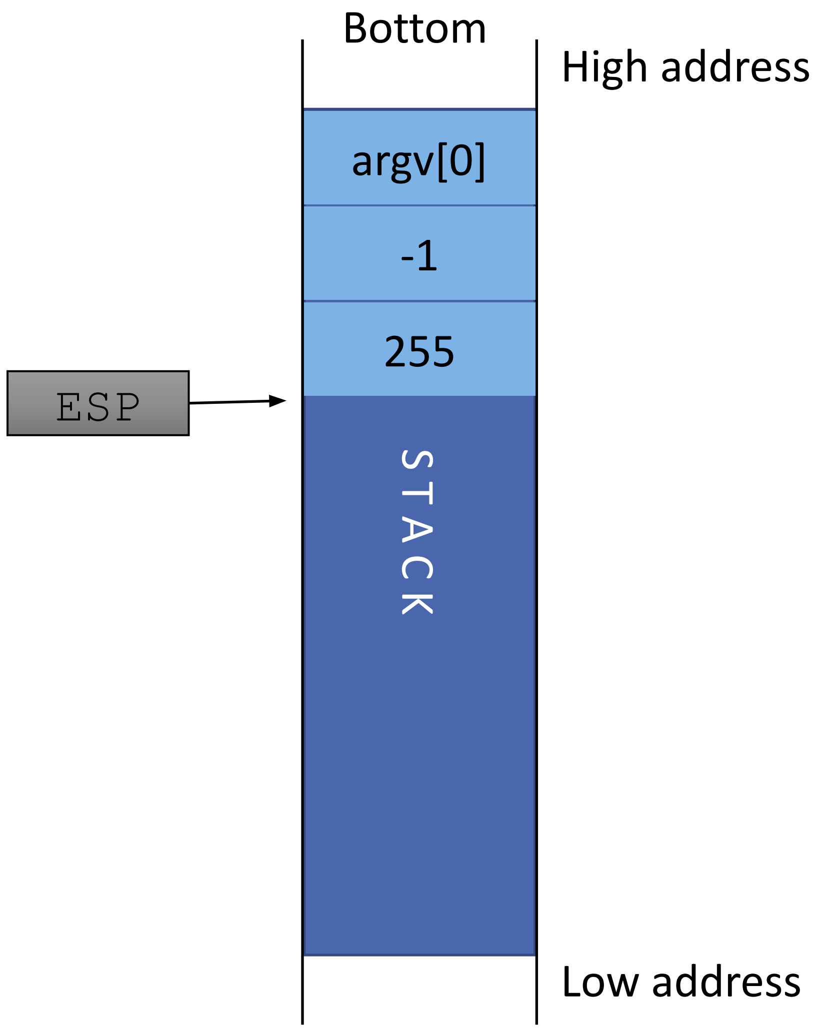
```
…
F1(0xff, UINT_MAX, argv[0]);
…
```

```
...
pushl   (%eax)
pushl   $-1
pushl   $255
call    F1
...
```



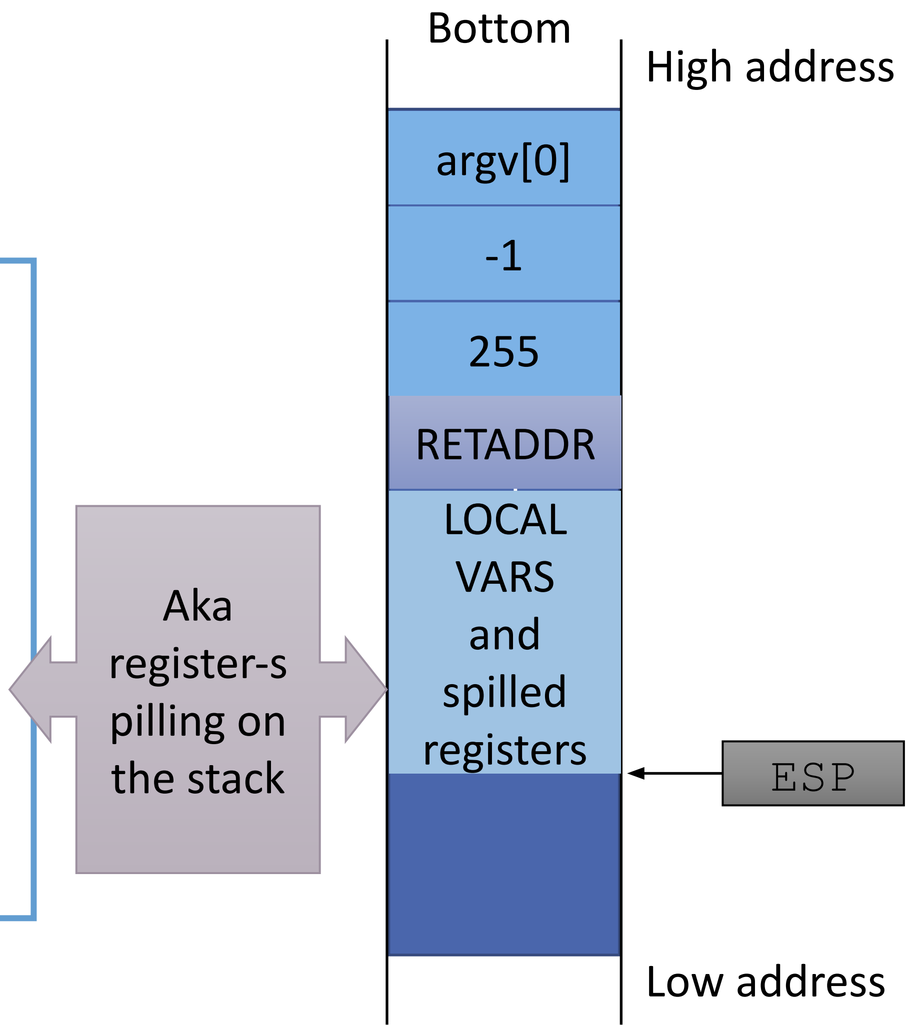
cdecl

- 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



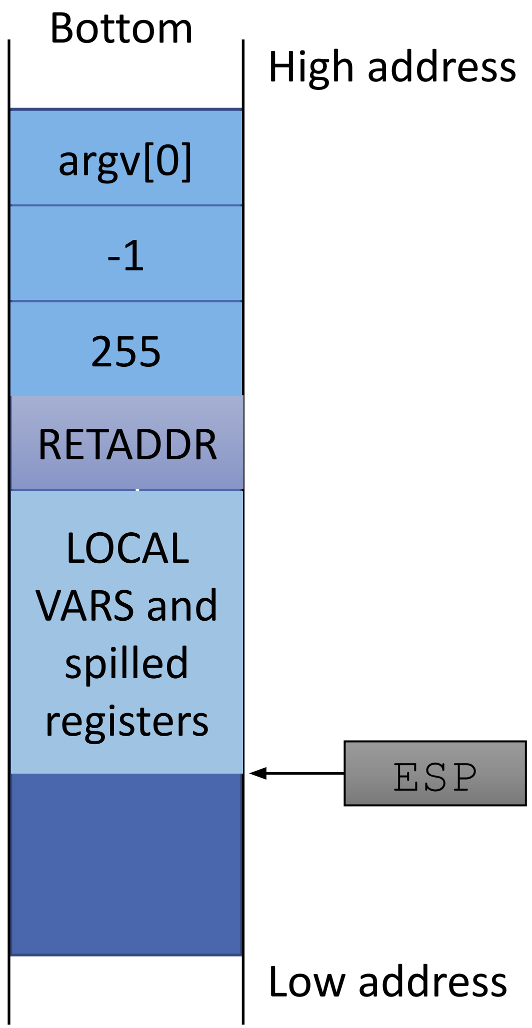
cdecl

- 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



cdecl

- 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

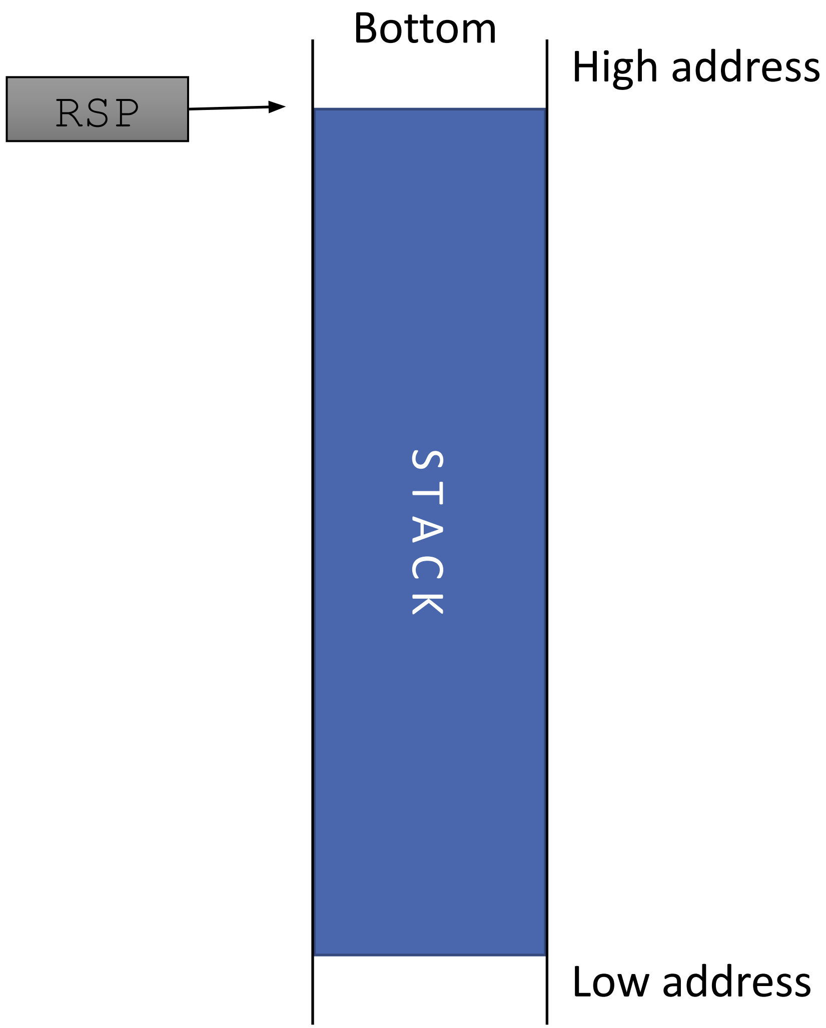


System V AMD64 ABI

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

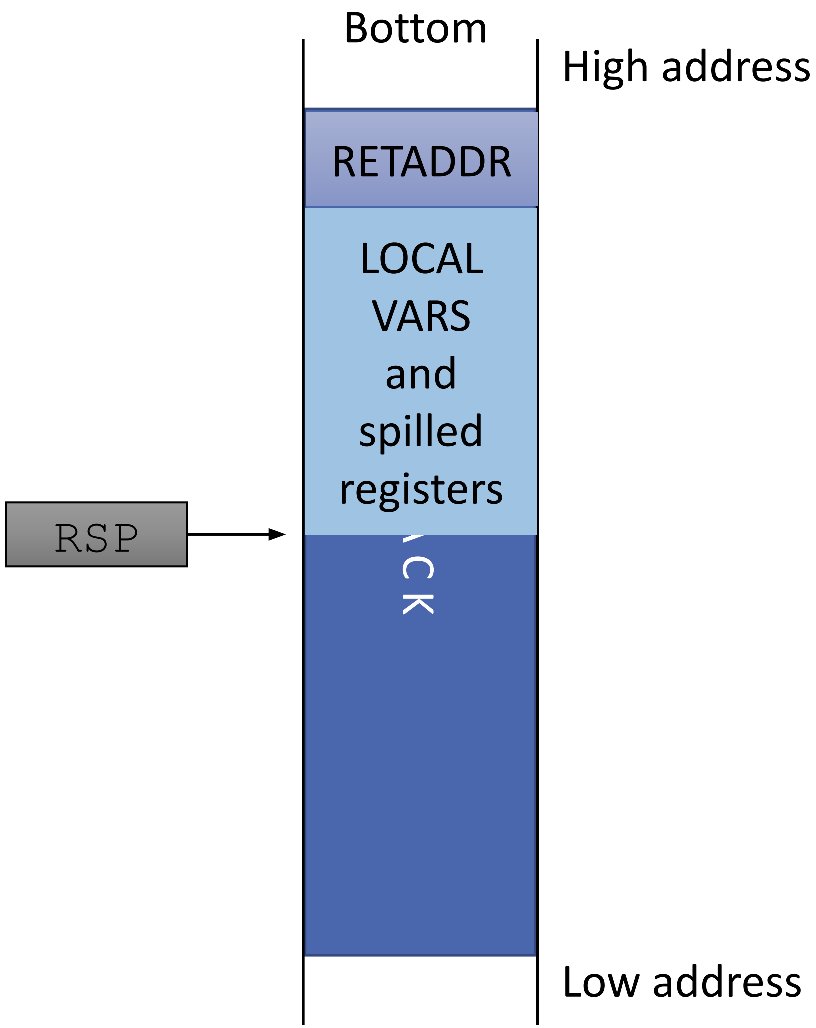
```
...
F1(0xff, UINT_MAX, argv[0]);
...
```

```
...
movq    (%rsi), %rdx
movl    $4294967295, %esi
movl    $255, %edi
call     F1
...
```



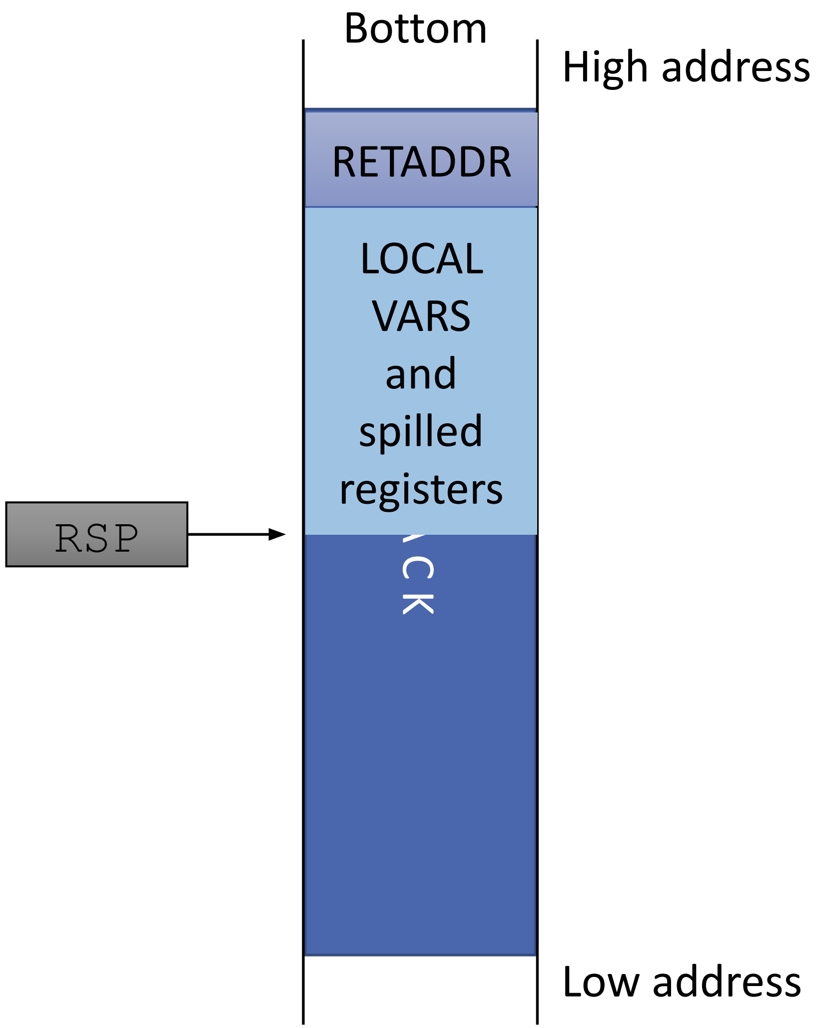
System V AMD64 ABI

- 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



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



Popular Conventions Summary

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

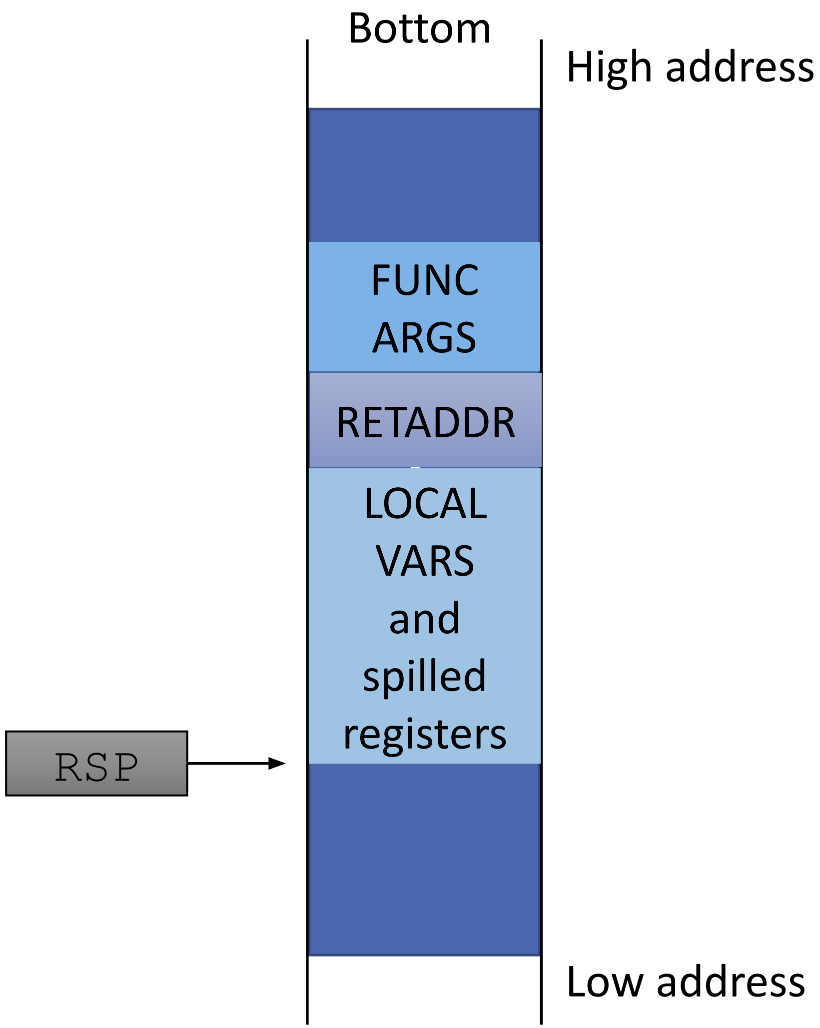
- Arguments are passed using registers
  - First 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9
- RBP, RBX, and R12–R15 are callee saved
- RAX used for function return

Conventions include additional information, consult reading material for thorough description

- Example: handling of floating point regs

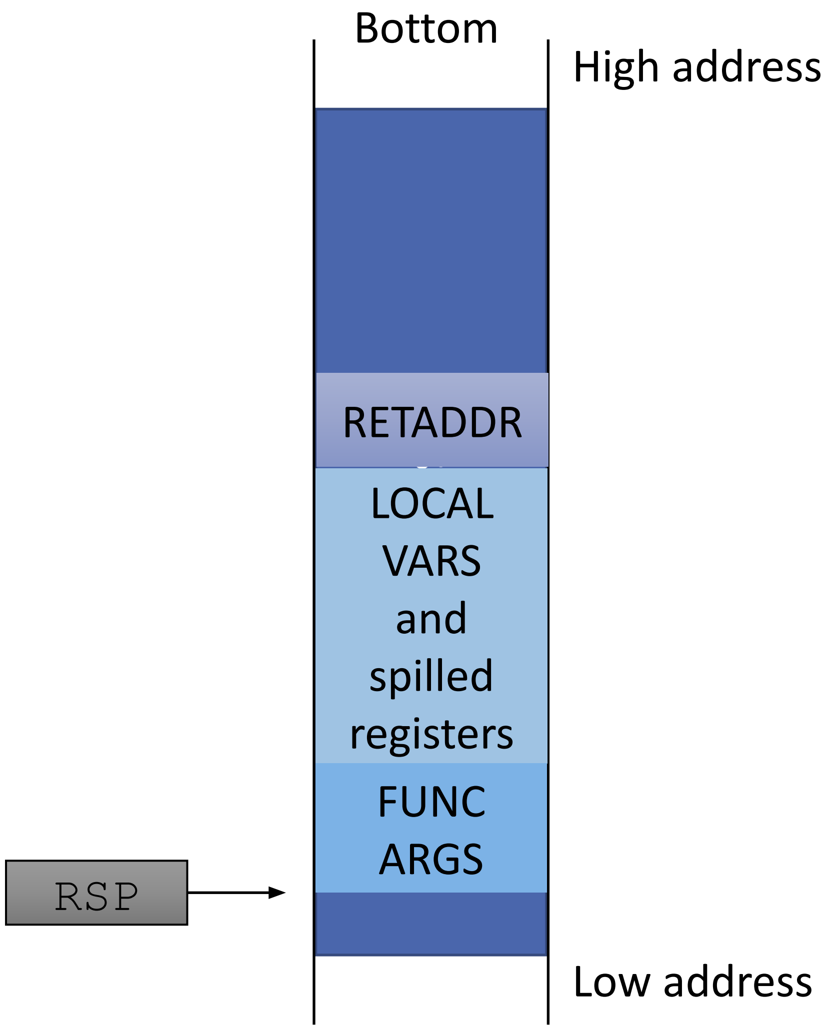
The Stack Is Used…

- …to store the return address of caller functions
  - Code pointers!
- …to store local variables
  - Aka stack variables
- …to pass function arguments
- …to temporarily store register values
- …to store the frame pointer



Stack Frame

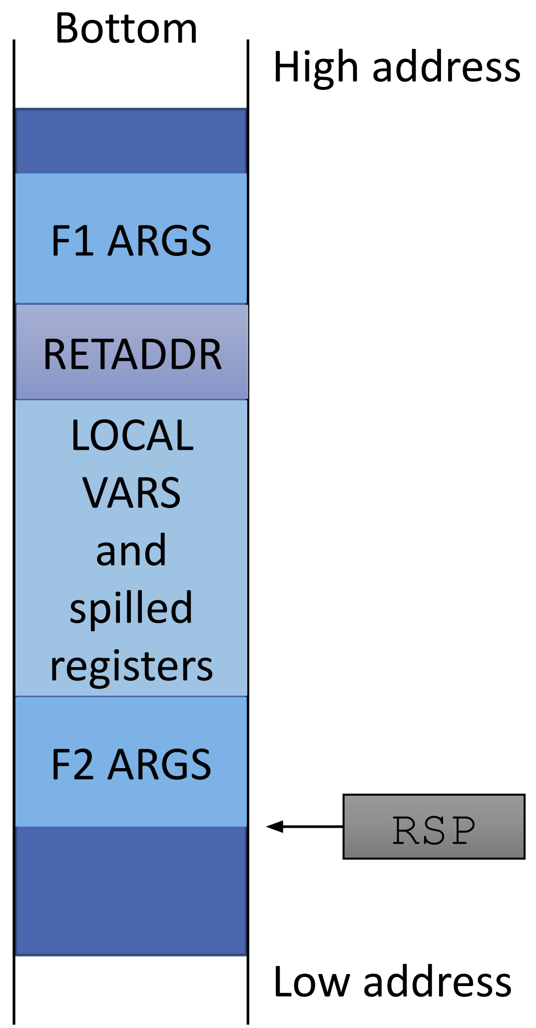
- A stack frame includes all function-local data
  - Local variables
  - Spilled registers
  - Function arguments pushed to the stack to make calls
- More of a logical entity
- Can grow as function executes



Stack Frame Boundaries

- Start below return address
- Stop at stack pointer

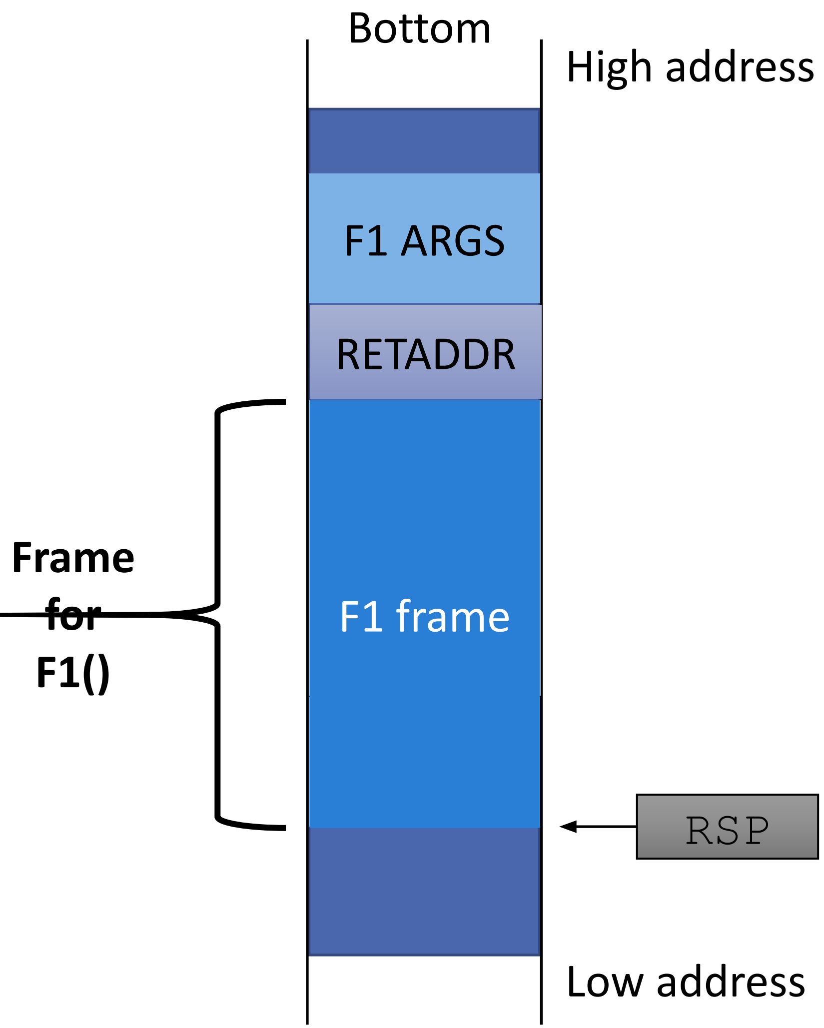
void F1(short a1, long a2, char *a3)
```
{
  int a;
  char buf[16];
  unsigned long l;
  ...
  long l2 = F2(a);
  ...
}
```



Stack Frame Boundaries

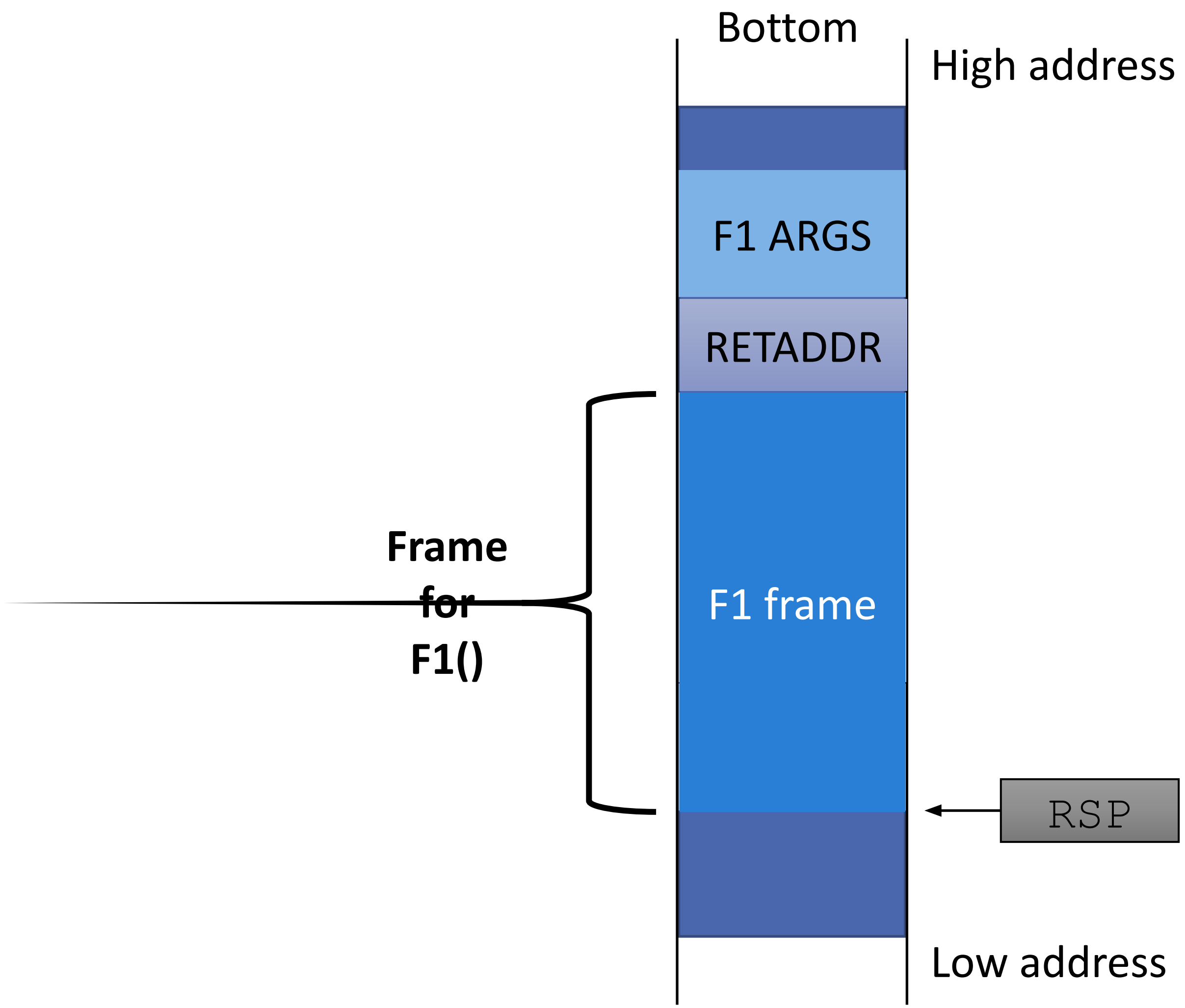
- Start below return address
- Stop at stack pointer

```
void F1(short a1, long a2, char *a3)
{
  int a;
  char buf[16];
  unsigned long l;
  ...
  long l2 = F2(a);
  ...
}
```



RETADDR and Stack Frames

- The return address may also be considered part of the frame
- We will not for simplicity

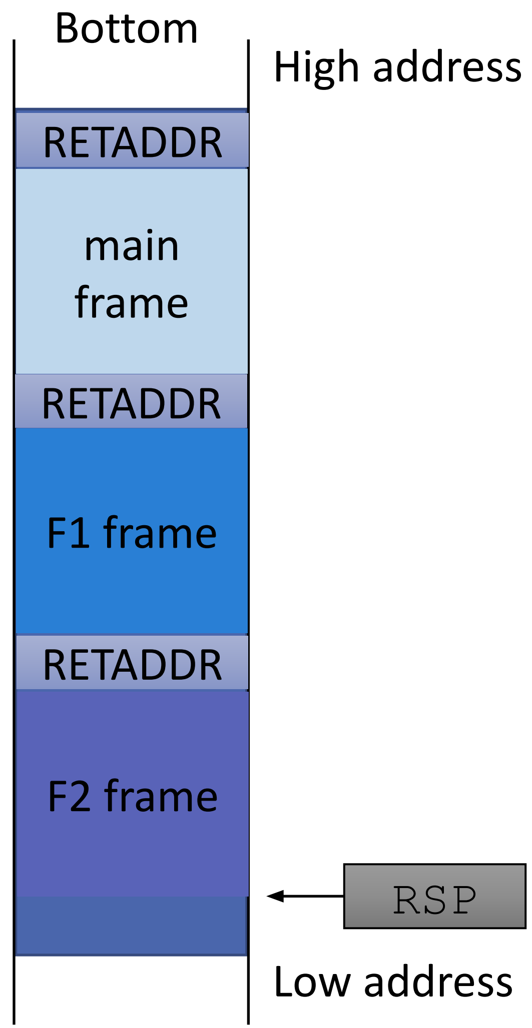


Stack Frames Example

```
int main(int argc, char **argv)
{
  ...
  F1(0xFF, UINT_MAX, argv[0]);
  ...
}
```

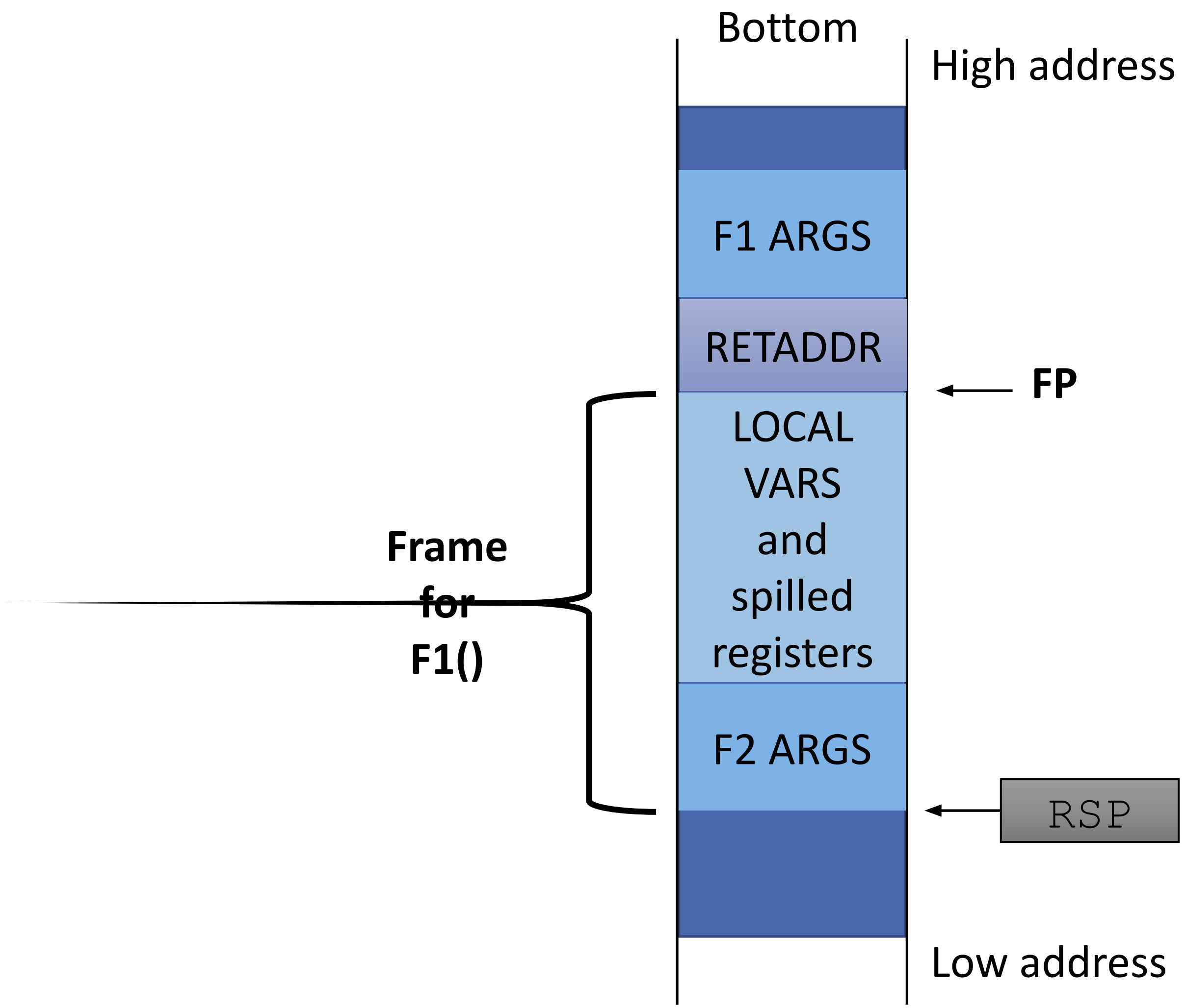
```
void F1(short a1, long a2, char *a3)
{
   int a;
   char buf[16];
   unsigned long l;
   ...
   long l2 = F2(a);
   ...
}
```

```
long F2(long a1)
{
  ...
}
```



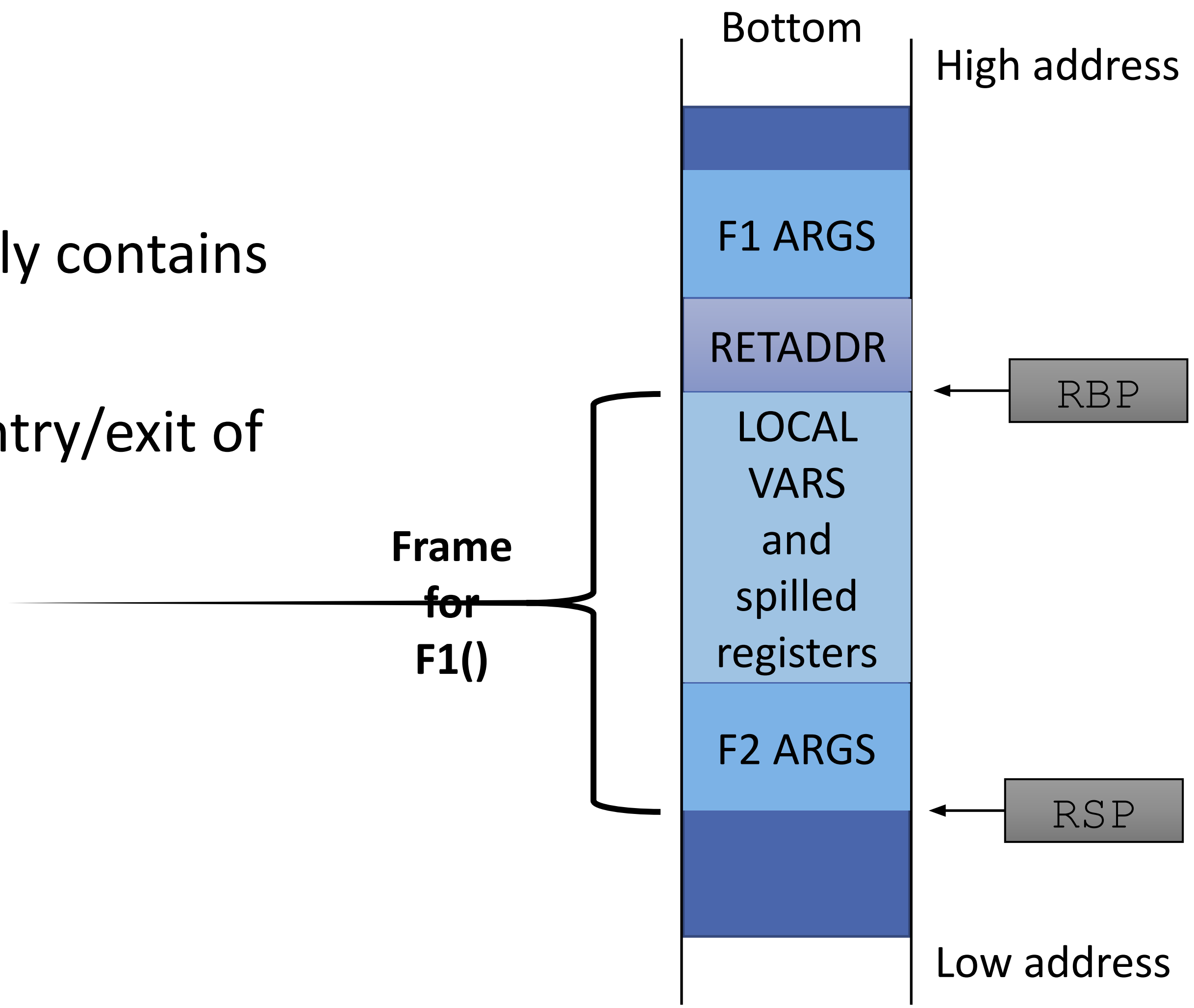
Frame Pointer (FP)

- Marks the highest address in the frame
  - Bottom of the frame
- Aka Base Pointer



Frame Pointer (FP)

- Marks the highest address in the frame
  - Bottom of the frame
- Aka Base Pointer
- The RBP/EBP register commonly contains the FP
- RBP needs to updated upon entry/exit of function



Maintaining the FP

C

```
…
F1();
…
```

```
void F1()
{
   int a;
   char buf[16];
   unsigne
   ...
}
```

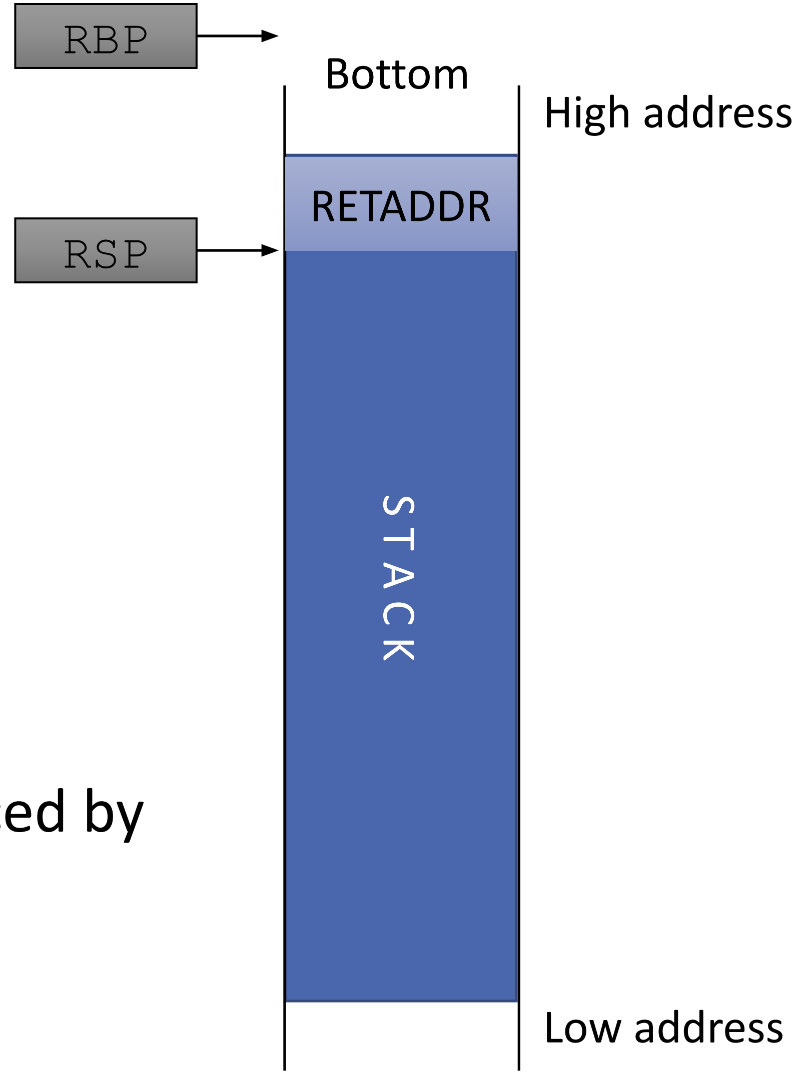
Prologue

Epilogue

ASM

```
…
CALL $F1_OFF
…
```

```
F1:
pushq %rbp
movq  %rsp, %rbp
subq  $48,%rsp

movq  %rbp, %rsp
pop   %rbp
ret
```



- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP

C

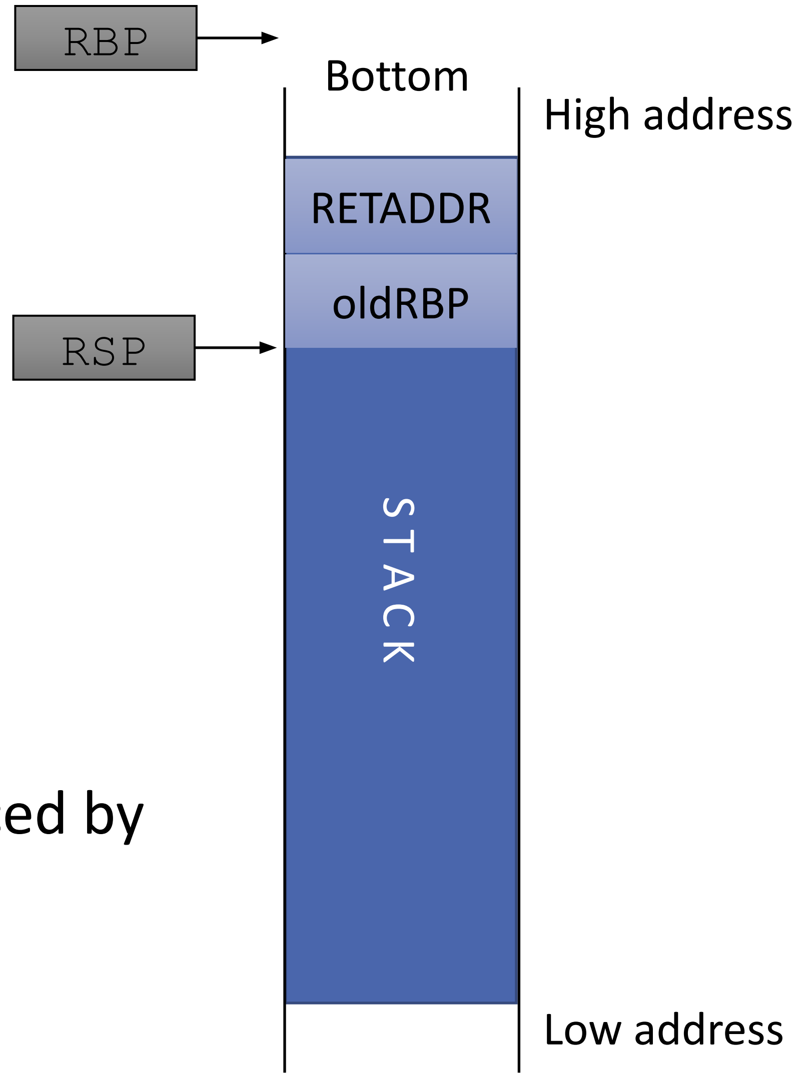
```
…
F1();
…
```

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

ASM

```
…
CALL $F1_OFF
…
```

```
F1:
pushq %rbp
movq  %rsp, %rbp
subq  $48,%rsp
...
movq  %rbp, %rsp
pop   %rbp
ret
```



- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP

C

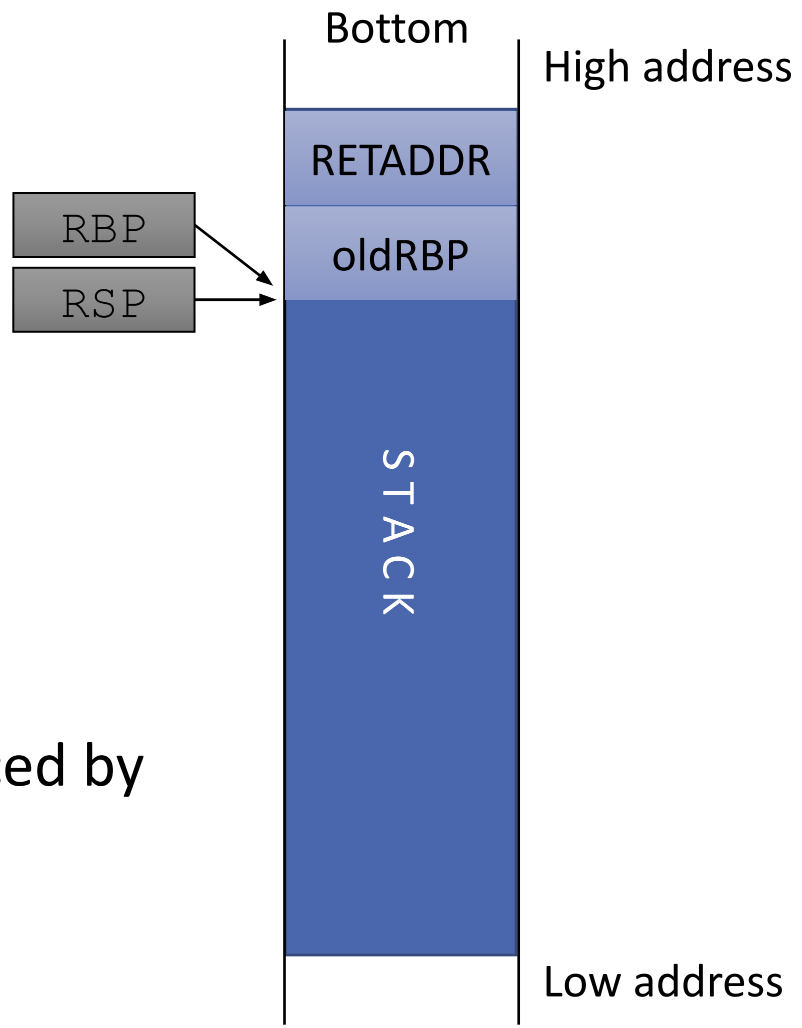
```
…
F1();
…
```

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

ASM

```
…
CALL $F1_OFF
…
```

```
F1:
pushq %rbp
movq  %rsp, %rbp
subq  $48,%rsp
...
movq  %rbp, %rsp
pop   %rbp
ret
```



- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP

C

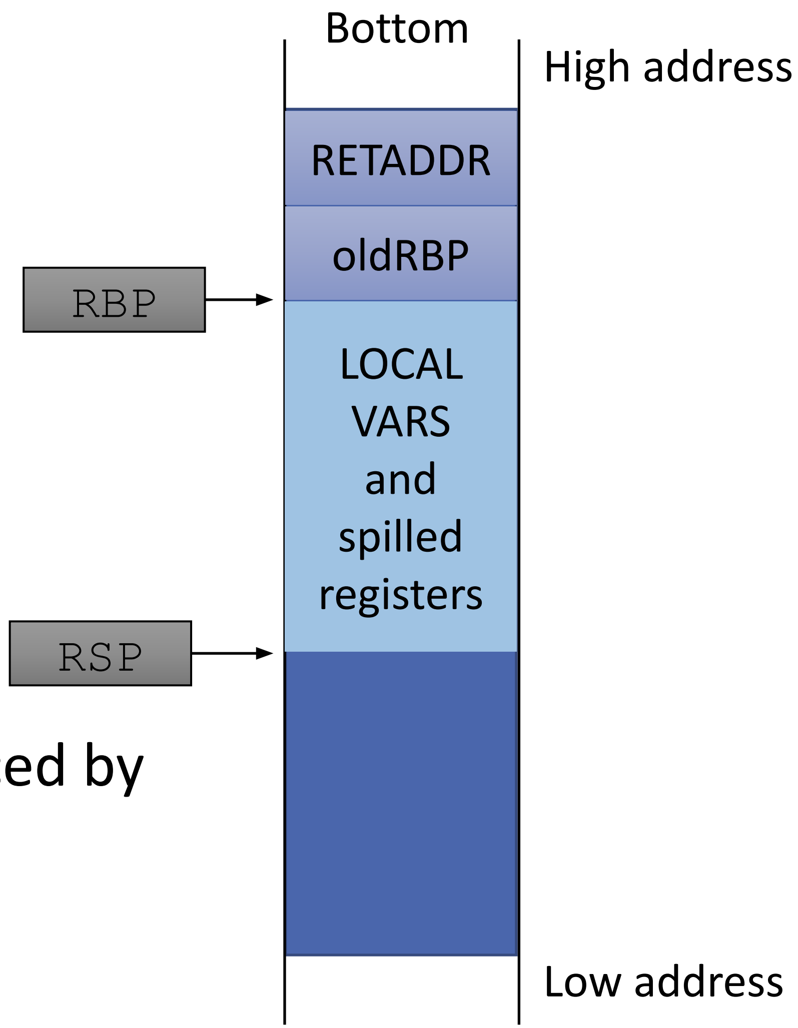
```
…
F1();
…
```

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

ASM

```
…
CALL $F1_OFF
…
```

```
F1:
pushq %rbp
movq  %rsp, %rbp
subq  $48,%rsp
...
movq  %rbp, %rsp
pop   %rbp
ret
```



- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP

C

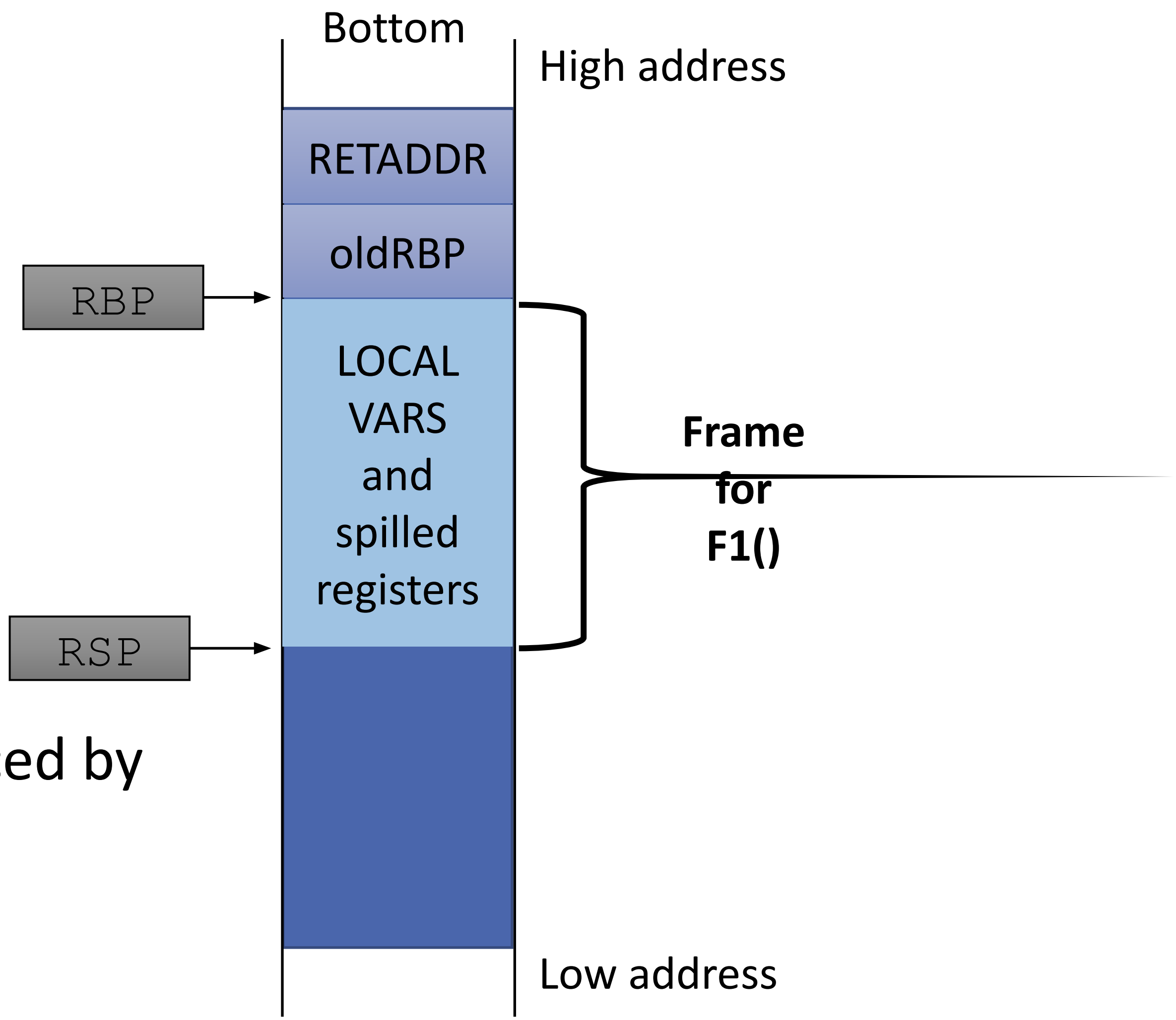
```
…
F1();
…
```

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

ASM

```
…
CALL $F1_OFF
…
```

```
F1:
pushq %rbp
movq  %rsp, %rbp
subq  $48,%rsp
...
movq  %rbp, %rsp
pop   %rbp
ret
```



- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP

C

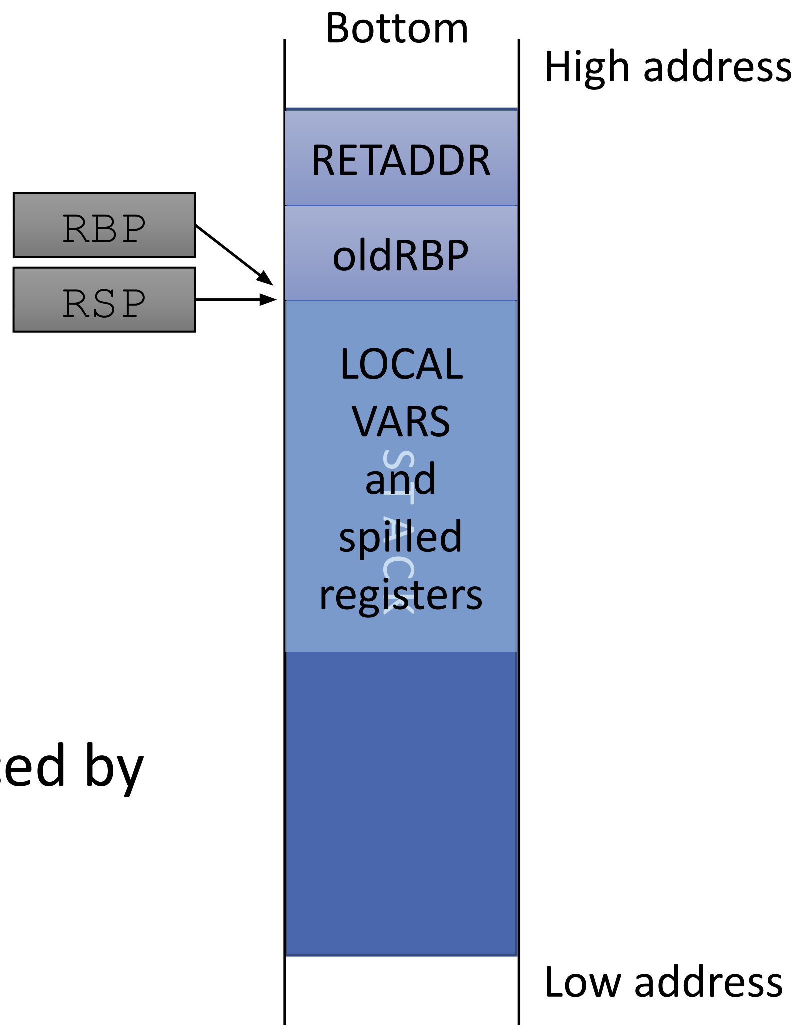
```
…
F1();
…
```

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

ASM

```
…
CALL $F1_OFF
…
```

```
F1:
pushq %rbp
movq  %rsp, %rbp
subq  $48,%rsp
...
movq  %rbp, %rsp
pop   %rbp
ret
```



- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP

C

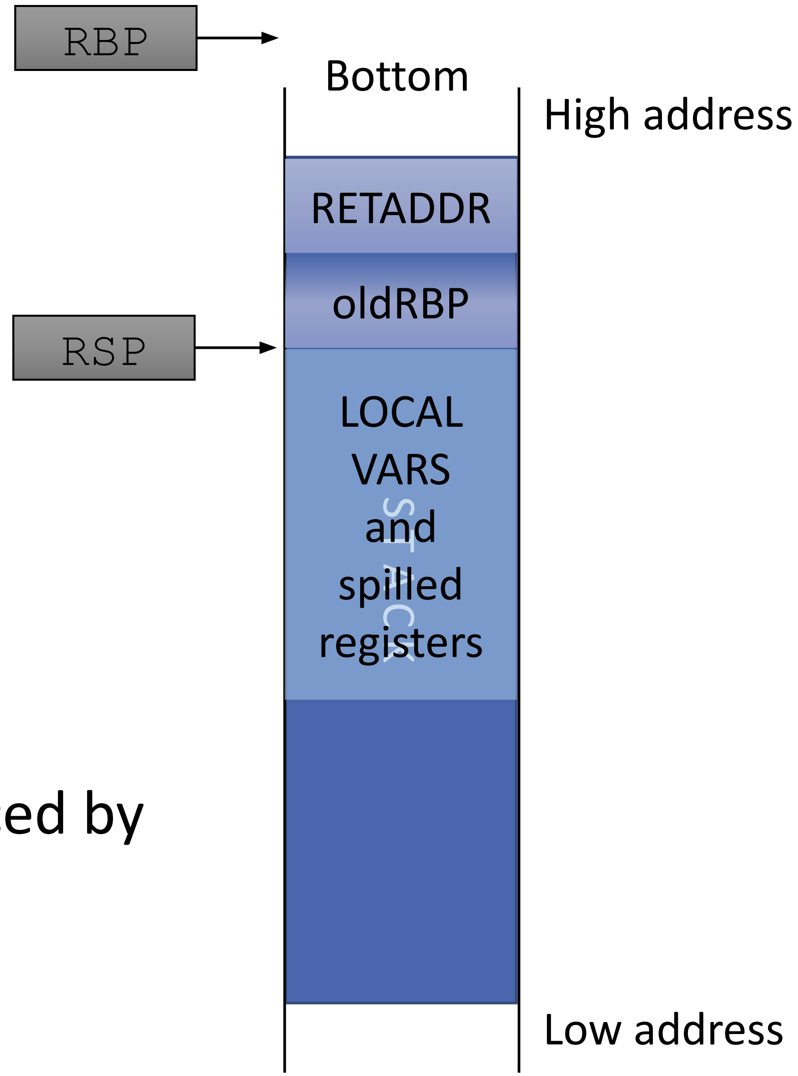
```
…
F1();
…
```

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

ASM

```
…
CALL $F1_OFF
…
```

```
F1:
pushq %rbp
movq  %rsp, %rbp
subq  $48,%rsp
...
movq  %rbp, %rsp
pop   %rbp
ret
```



- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP

C

```
…
F1();
…
```

ASM

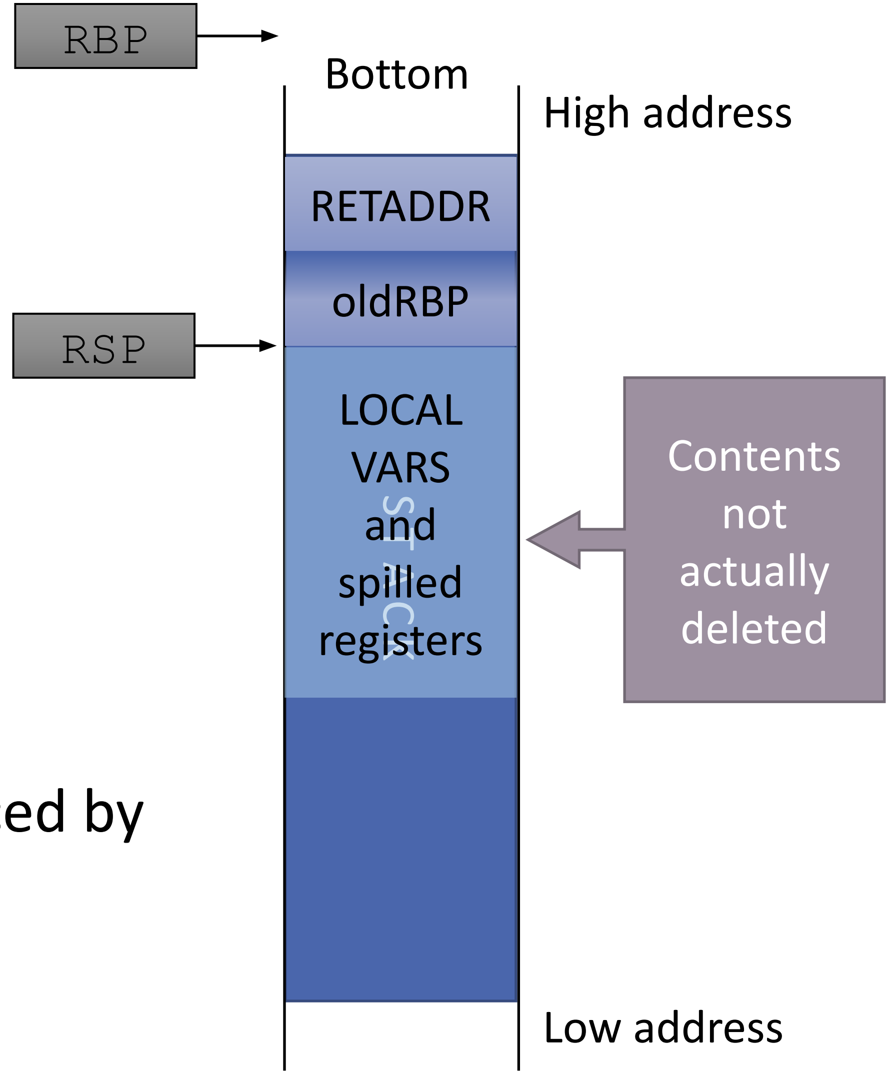
```
…
CALL $F1_OFF
…
```

```
void F1()
{
  int a;
  char bu
  unsigne
  …
}
```

Equivalent Instruction: `enter`

Equivalent Instruction: `leave`

```
F1:
pushq %rbp
movq  %rsp, %rbp
subq  $48,%rsp

movq  %rbp, %rsp
pop   %rbp
ret
```



- Instruction sequences are introduced by the compiler to maintain the FP

Putting It All Together

Function Call

- Prepare function call arguments
- Make the call
- Function prologue
  - Save RBP/EBP
  - Setup new RBP/EBP
- Callee saves registers that need to be preserved
- Callee allocates stack space

Function Return

- Function epilogue
  - Release stack space
  - Restore BP
- Return

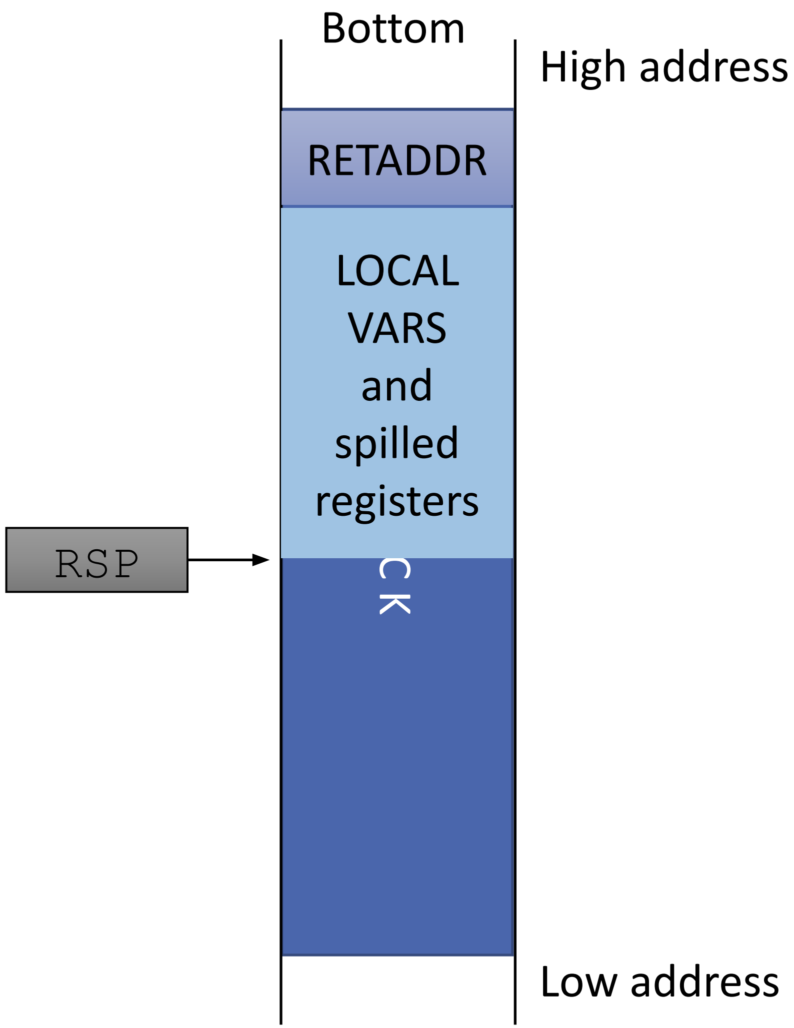
Accessing Stack Variables (no FP)

C

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

ASM

```
F1:

...
movl   76(%rsp), %eax
cltq
movq   %rax, %rdi
call   F2
movq   %rax, 40(%rsp)
...
ret
```



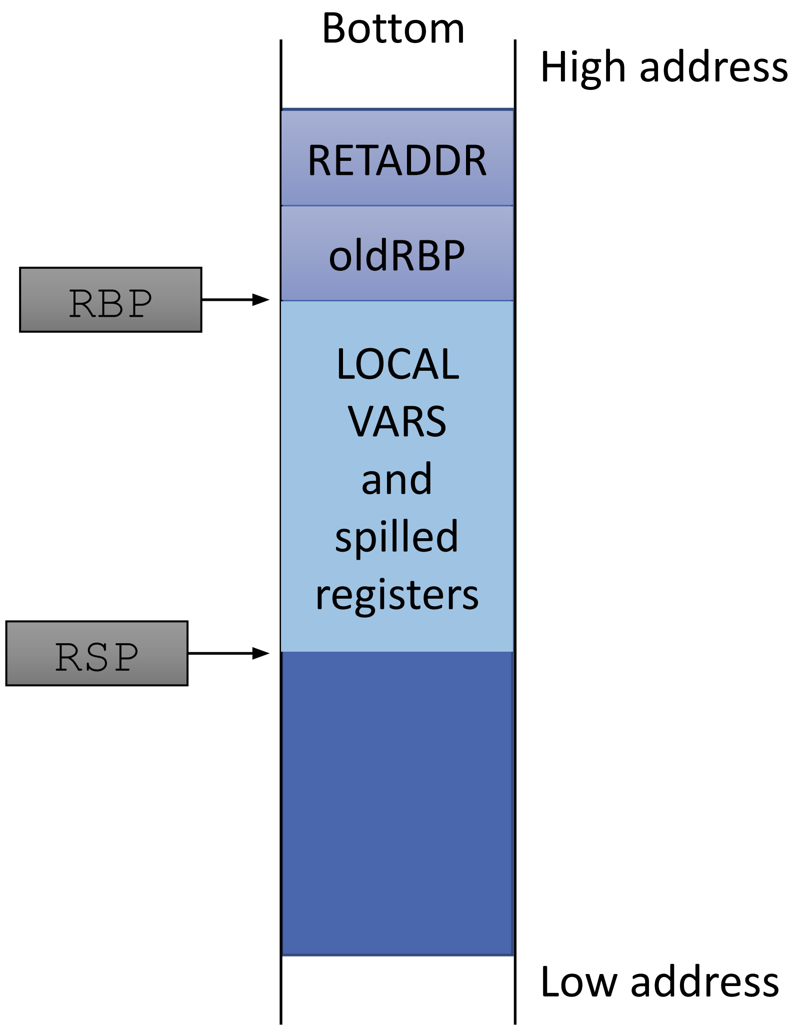
Accessing Stack Variables (with FP)

C

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

ASM

```
F1:

...
movl   -4(%rbp), %eax
cltq
movq   %rax, %rdi
call   F2
movq   %rax, -40(%rbp)
...
ret
```



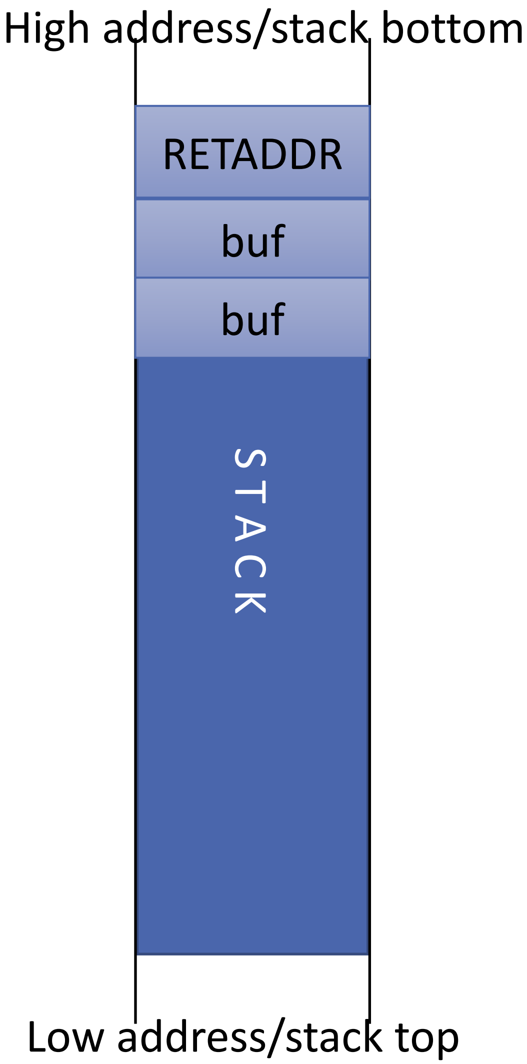
Stack Smashing Attacks

Stack Overflow Example

```
void copy(const char *str)
{
        char buf[16];

        strcpy(buf, str);

        puts(buf);
}
```

Stack Overflow Example

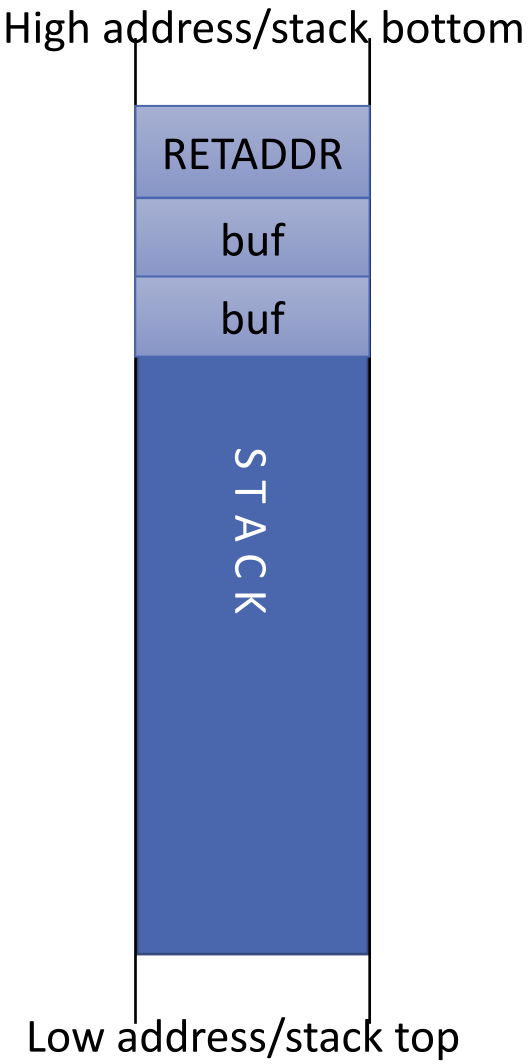
```
void copy(const char *str)
{
        char buf[16];

        strcpy(buf, str);

        puts(buf);
}
```



Stack Overflow Example

```
void copy(const char *str)
{
        char buf[16];

        strcpy(buf, str);

        puts(buf);
}
```

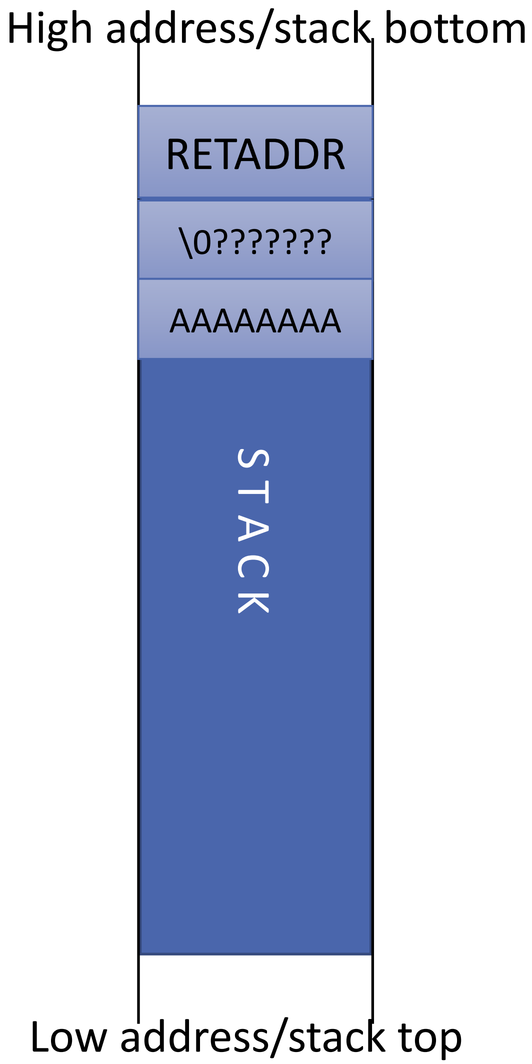
./copy AAAAAAAA



Stack Overflow Example

```
void copy(const char *str)
{
        char buf[16];

        strcpy(buf, str);

        puts(buf);
}
```

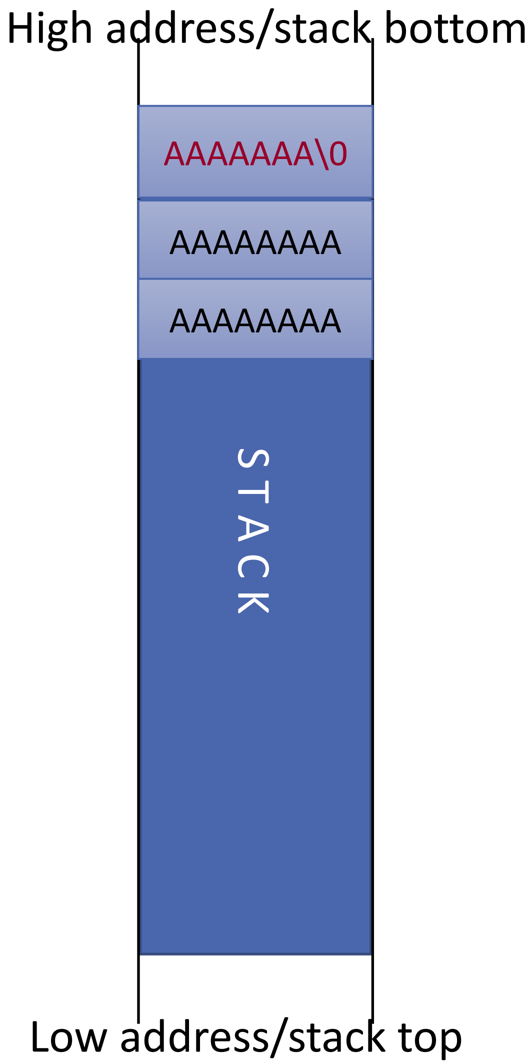
./copy AAAAAAA



Stack Overflow Example

```
void copy(const char *str)
{
        char buf[16];

        strcpy(buf, str);

        puts(buf);
}
```

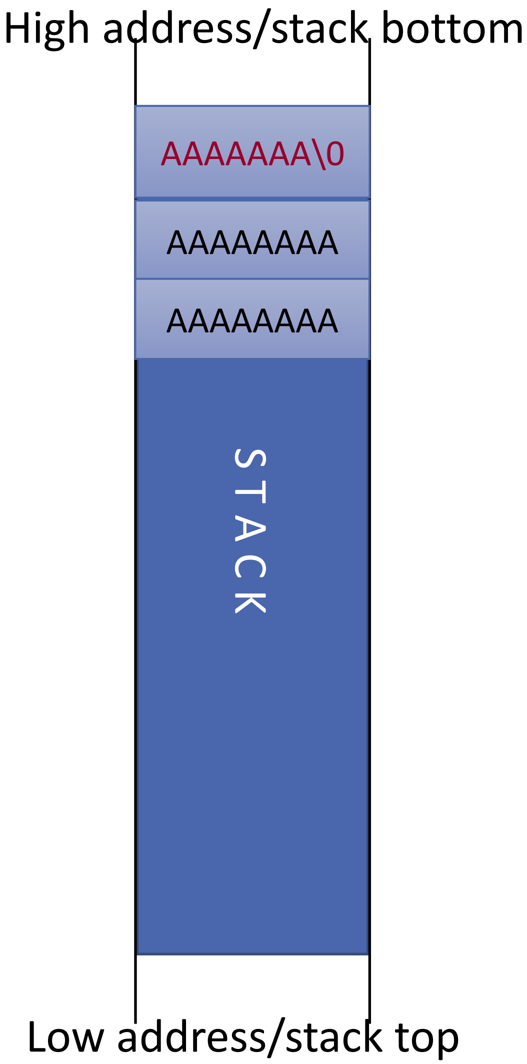
```
./copy AAAAAAAAAAAAAAAAAAAAAAAAAA
```



Stack Overflow Example

```
void copy(const char *str)
{
        char buf[16];

        strcpy(buf, str);

        puts(buf);
}
```

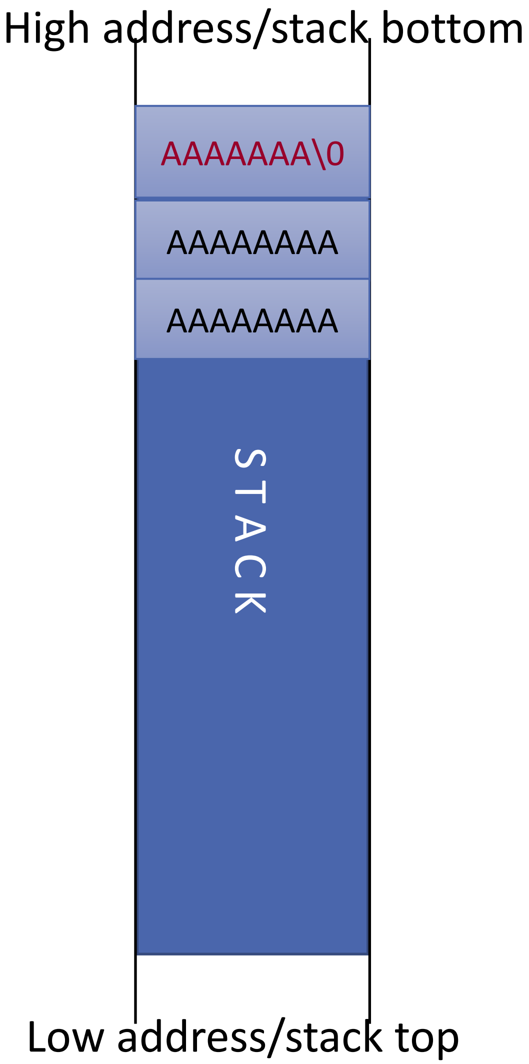
```
subq    $40, %rsp
movq    %rdi, 8(%rsp)
movq    8(%rsp), %rdx
leaq    16(%rsp), %rax
movq    %rdx, %rsi
movq    %rax, %rdi
call    strcpy@PLT
leaq    16(%rsp), %rax
movq    %rax, %rdi
call    puts@PLT
nop
addq    $40, %rsp
ret
```



Stack Overflow Example

- This stack overflow allows a to control the return address stored in the stack
- When ret executes, the control-flow of the program will be redirected to an arbitrary address control-flow hijacking

```
subq    $40, %rsp
movq    %rdi, 8(%rsp)
movq    8(%rsp), %rdx
leaq    16(%rsp), %rax
movq    %rdx, %rsi
movq    %rax, %rdi
call    strcpy@PLT
leaq    16(%rsp), %rax
movq    %rax, %rdi
call    puts@PLT
nop
addq    $40, %rsp
ret
```



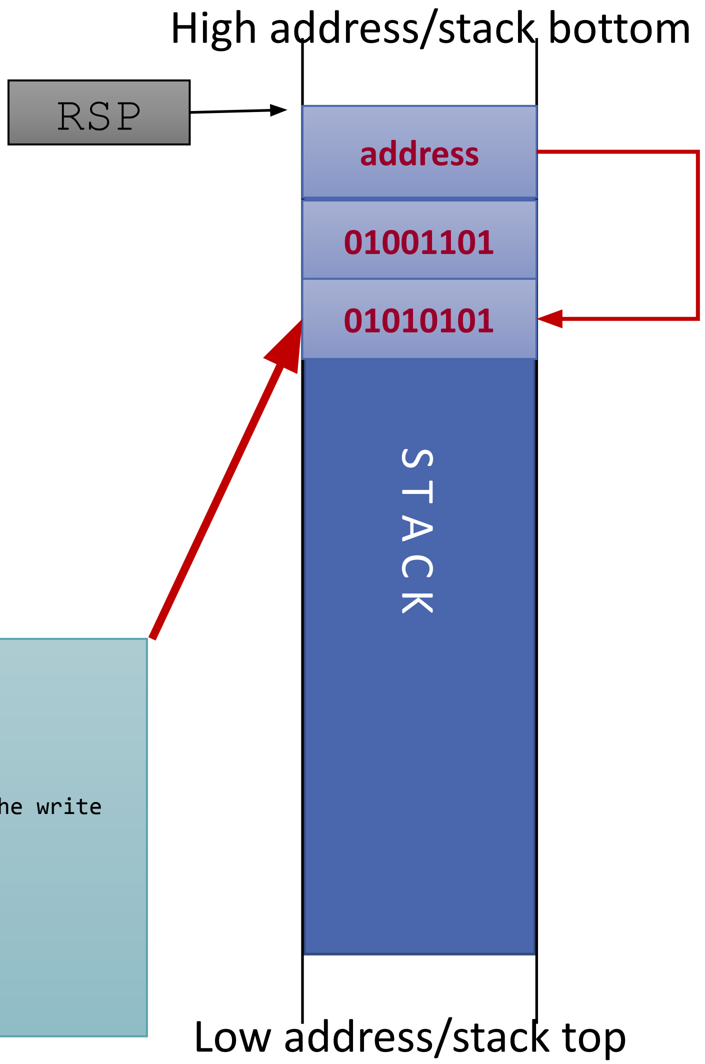
Control-Flow Hijacking Attacks

- Untrusted inputs that lead to corruption of a code pointer, which will be later dereferenced, lead to **control-flow hijacking attacks**

Original Stack Smashing Attack

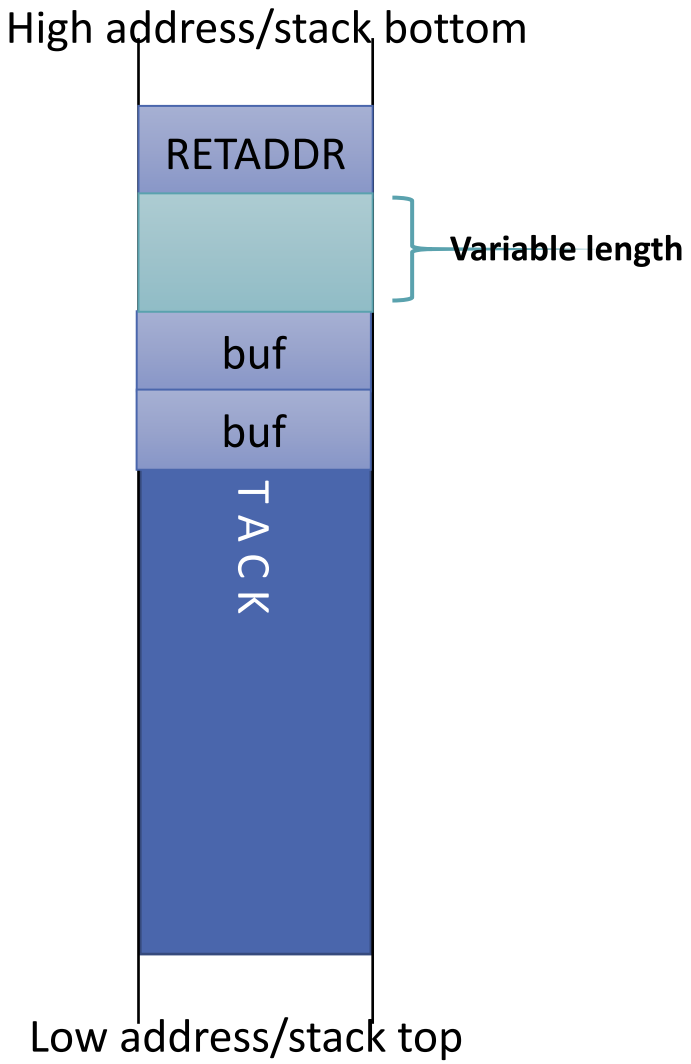
- Appeared at Phrack magazine
  http://phrack.org/issues/49/14.html#article
- Exploits the fact that stack used to be executable
  - Stores binary code in the controlled buffer
    - **Any** executable, controlled buffer will do!
  - Redirect program to inject code
- Performs arbitrary **code injection**!

```
        # write(1, message, 13)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi
        mov     $13, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        mov     $60, %rax
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello, world\n"
```



Making Exploits More Robust

- Observation: Different compiler may use different alignment, spill different register, etc.
- Problems:
  - Exact distance of return address may be different between binaries
  - Exact address of buffer may be different



Making Exploits More Robust

- Observation: Different compiler may use different alignment, spill different register, etc.
- Problems:
  - Exact distance of return address may be different between binaries
  - Exact address of buffer may be different
- Solutions:
  - Use multiple copies of the target address
  - Prepend a **NOP sled** to shellcode
    - NOPs No operations are special one byte instructions to do nothing
  - Aim for target address pointing into NOP sled
    - Execution will slide into shellcode