Looking at the Assembly

Lecture 5

Christian A. Pagot

Universidade Federal da Paraíba
Centro de Informática

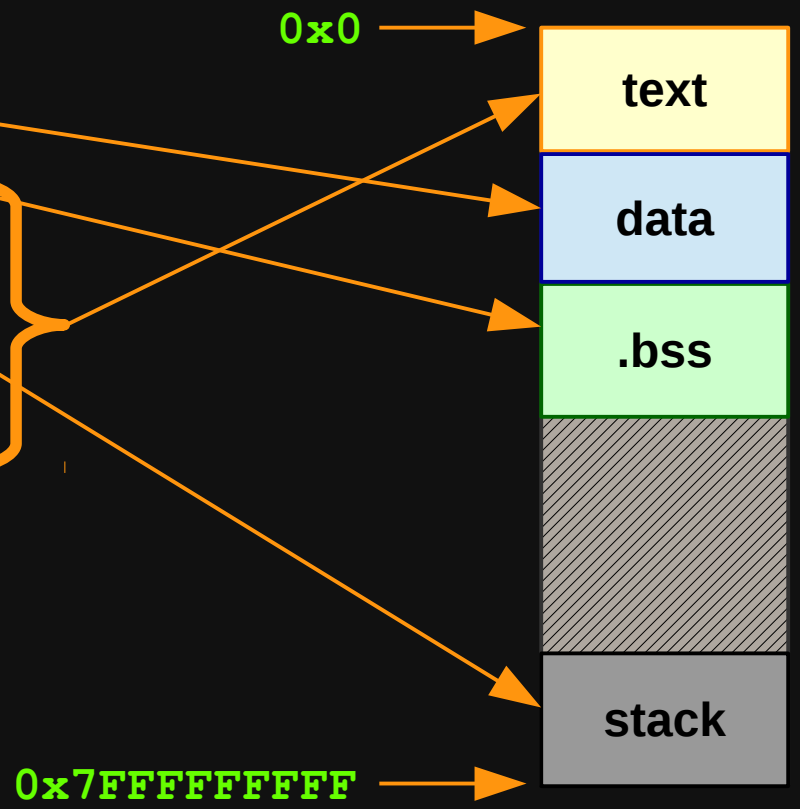
Data and Code Segmentation

How the **elements** of the program bellow are actually **segmented** within the **virtual memory**?

assembly1.c

```
int x = 10;
int y;

main() {
    int w;
    y = 20;
    w = x + y;
    return 0;
}
```



Check the **addresses** of the **variables** and the **main fuction**!

For a less abstract view, let's see what is happening at the **assembly** level...

Assembly

"*An **assembly language** (or assembler language) is a **low-level** programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) **correspondence** between the **language** and the architecture's **machine code instructions**.*"

Assembly language, Wikipedia

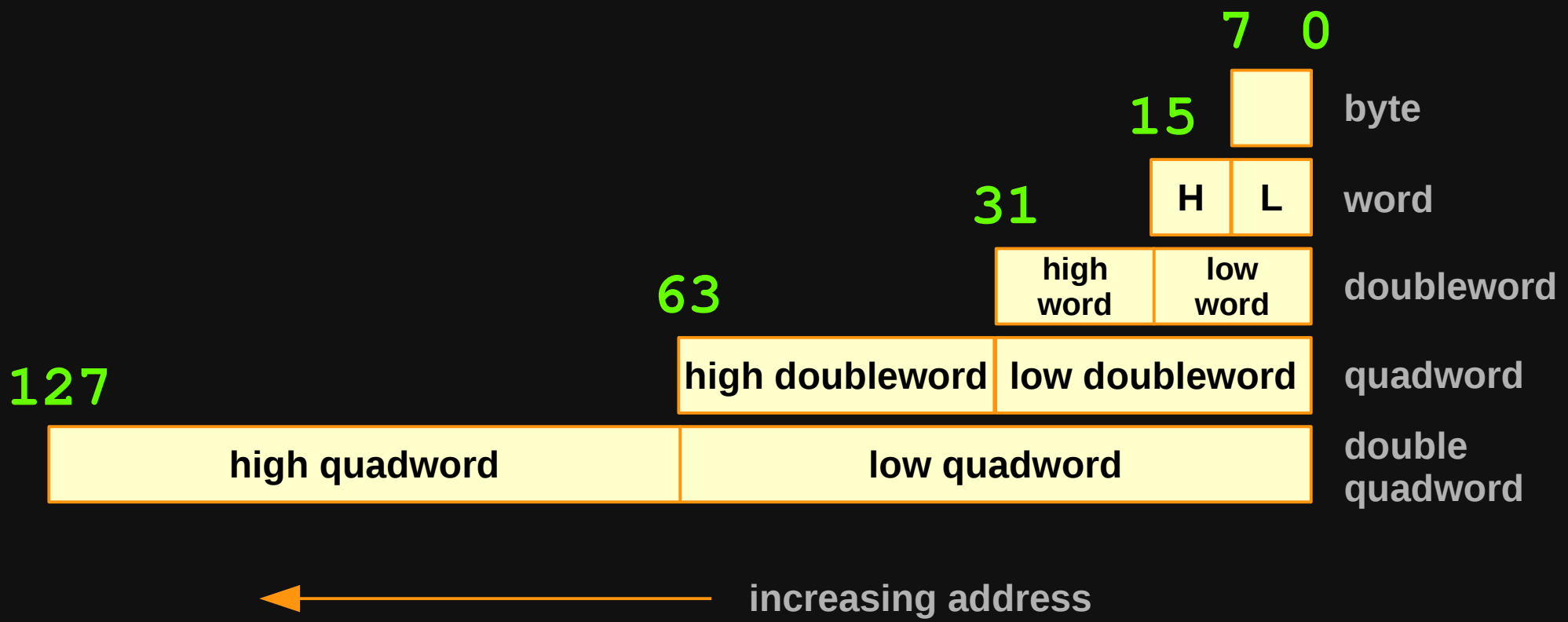
x86 and x86-64 (AMD64)

- x86
  - Intel's and AMD's 32 bit instruction set.
- x86-64 or AMD64
  - Intel's and AMD's 64 bit instruction set.

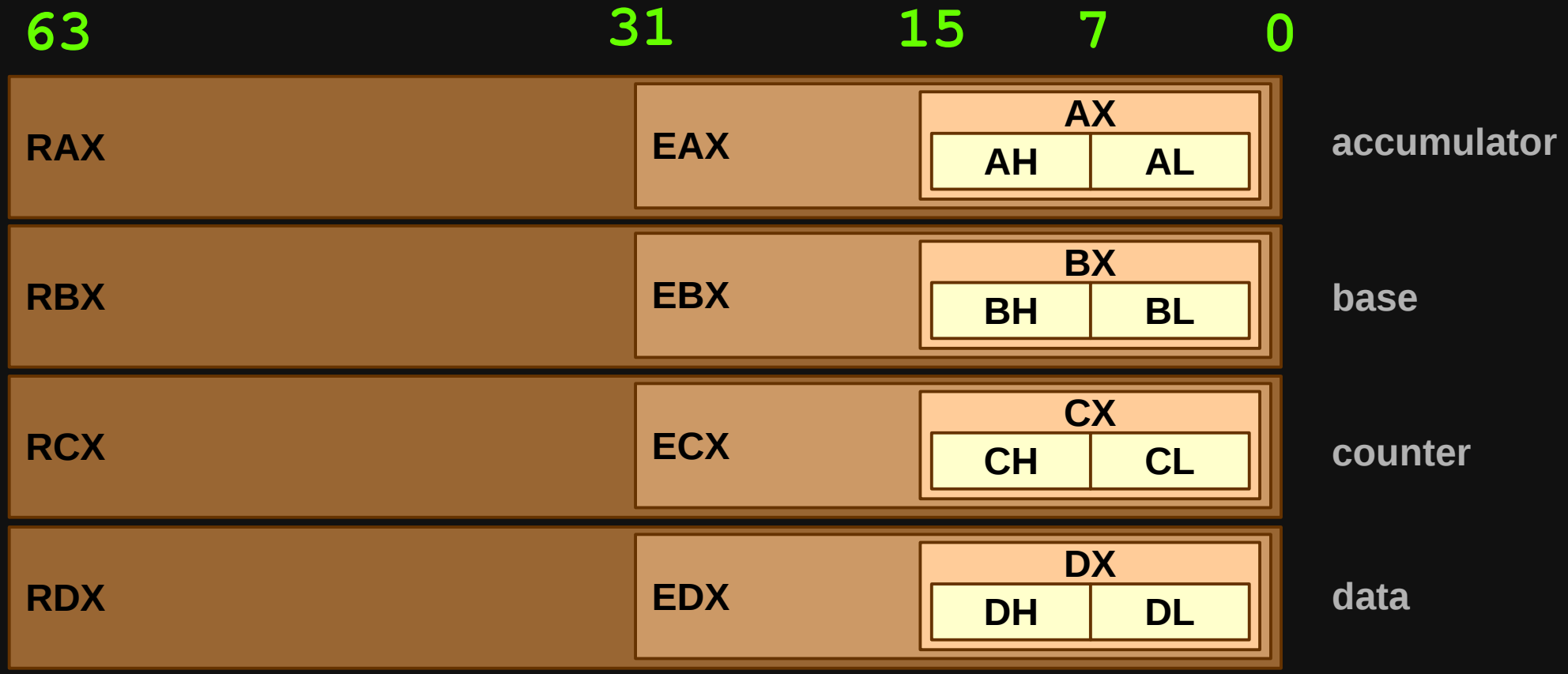
We will keep with the **x86-64 instruction set**!

x86-64 Register Set

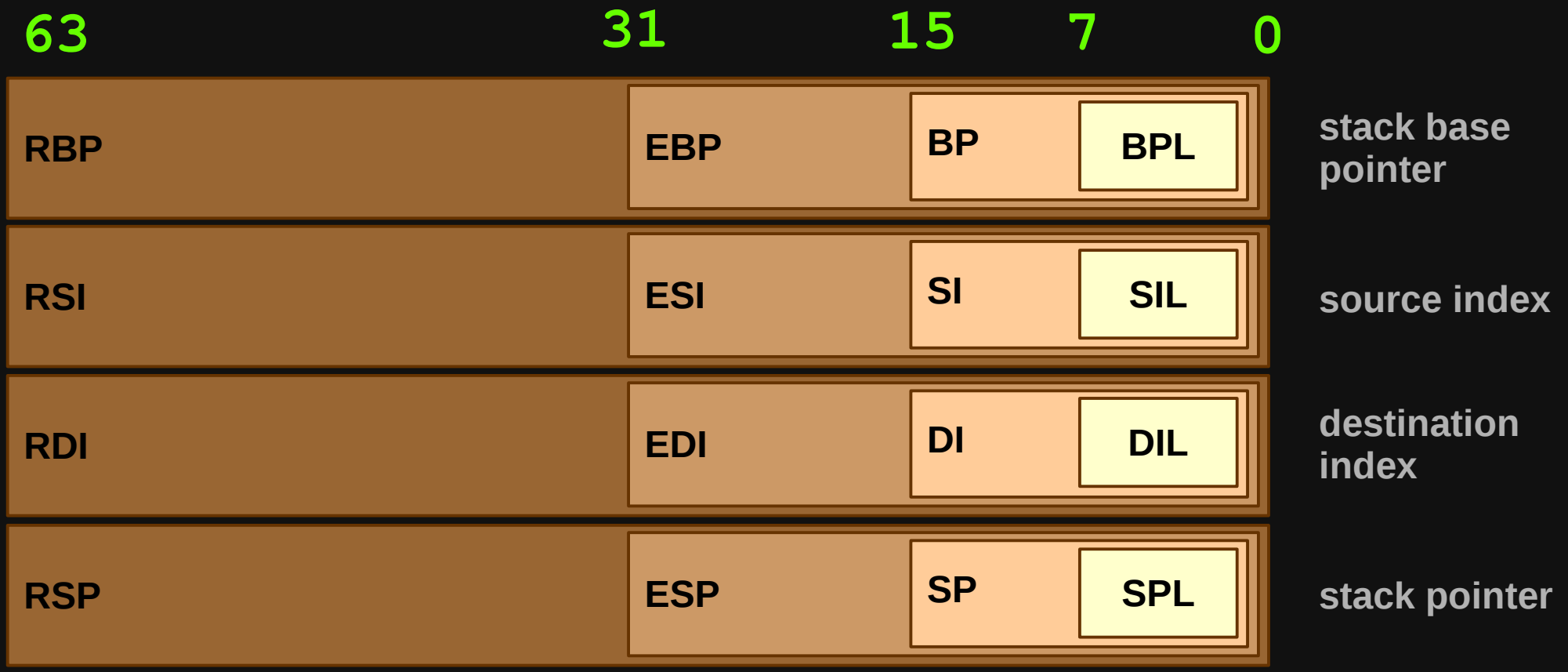
Register widths:



16 General Purpose Register Set



16 General Purpose Register Set



16 General Purpose Register Set

Other Registers

- **RIP** : 64 bit instruction pointer.
  - Points to the next instruction to be executed.
- **RFLAGS** : Flags.
  - Stores flags generated by computation results and for controlling the processor.
- **FPR0**, …, **FPR7** : Floating Point Unit (FPU) registers.
  - Floating point registers.
- **MMX** and **XMM**
  - A set of registers to be used by the MMX and SSE extensions.

Assembly Syntax

- There are two assembly syntaxes that are very **popular**:
  - Intel.
  - AT&T.

We will keep with the **AT&T** syntax!

AT&T Assembly Syntax

- General format of instructions
  - Mnemonic source, destination
- Operation suffixes:

`b`	**byte** (8 bit)
`s`	**short** (16 bit integer) or **single** (32-bit floating point)
`w`	**word** (16 bit)
`l`	**long** (32 bit integer or 64 bit floating point)
`q`	**quad** (64 bit)
`t`	**ten bytes** (80-bit floating point)

- Prefixes
  - Registers are prefixed with `%`.
- Example
  - `movb $0x05, %al`

AT&T Assembly Syntax

- Address operand syntax

```
segment:displacement(base register, offset register, scalar multiplier)
```

- Examples
  - `movl -4(%ebp, %edx, 4), %eax`
    - Full example: load *(ebp – 4 + (edx * 4)) into **eax**.
  - `movl -4(%ebp), %eax`
    - Typical example: load a stack variable into **eax**.
  - `movl (%ecx), %edx`
    - No offset: copy the target of a pointer into a register.

Obtaining the Assembly of C Code

- **How to** obtain the **assembly** code generated from **C code**?
  - Typically, two methods are used:
    - Getting the output assembly code **generated** by the **compiler**.
    - **Disassembling** the machine code with a **debugger** during the **runtime**.

Assembly Code Generated by GCC

```
~$ gcc -S -fno-asynchronous-unwind-tables assembly1.c
```

assembly1.c

```
int x = 10;
int y;

main() {
    int w;
    y = 20;
    w = x + y;
    return 0;
}
```

DIY!

assembly1.s

```
        .file   "disassembly1.c"
        .globl  x
        .data
        .align  4
        .type   x, @object                    .data
        .size   x, 4
x:
        .long   10
        .comm   y,4,4                         .bss
        .text
        .globl  main
        .type   main, @function
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $20, y(%rip)                  .text
        movl    x(%rip), %edx
        movl    y(%rip), %eax
        addl    %edx, %eax
        movl    %eax, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        ret
        .size   main, .-main
        .ident  "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4"
        .section        .note.GNU-stack,"",@progbits
```

GDB Disassembler

DIY!

```
(gdb) disassemble
```

assembly1.s

```
        .text
        .globl  main
        .type   main,@function
main:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $20, y(%rip)
    movl    x(%rip), %edx
    movl    y(%rip), %eax
    addl    %edx, %eax
    movl    %eax, -4(%rbp)
    movl    $0, %eax
    popq    %rbp
    ret
```

Only the `.text` segment.

GDB output.

```
Dump of assembler code for function main:
   0X00000000004004ed <+0>: push   %rbp
   0X00000000004004ee <+1>: mov    %rsp,%rbp
=> 0x00000000004004f1 <+4>: movl   $0x14,0x200b45(%rip) # 0x601040 <y>
   0X00000000004004fb <+14>: mov    0x200b37(%rip),%edx # 0x601038 <x>
   0X0000000000400501 <+20>: mov    0x200b39(%rip),%eax # 0x601040 <y>
   0X0000000000400507 <+26>: add    %edx,%eax
   0X0000000000400509 <+28>: mov    %eax,-0x4(%rbp)
   0X000000000040050c <+31>: mov    $0x0,%eax
   0X0000000000400511 <+36>: pop    %rbp
   0X0000000000400512 <+37>: retq
End of assembler dump.
```

Only the `.text` segment (actually, the `main()` function).

Symbols have been substituted by actual **virtual memory addresses**!

Disassembling with `/m`


```
Dump of assembler code for function main:
4   main() {
   0X00000000004004ed <+0>: push   %rbp
   0X00000000004004ee <+1>: mov    %rsp,%rbp

5       int w;
6       y = 20;
=> 0x00000000004004f1 <+4>: movl   $0x14,0x200b45(%rip) # 0x601040 <y>

7       w = x + y;
   0X00000000004004fb <+14>: mov    0x200b37(%rip),%edx # 0x601038 <x>
   0X0000000000400501 <+20>: mov    0x200b39(%rip),%eax # 0x601040 <y>
   0X0000000000400507 <+26>: add    %edx,%eax
   0X0000000000400509 <+28>: mov    %eax,-0x4(%rbp)

8       return 0;
   0X000000000040050c <+31>: mov    $0x0,%eax

9   }
   0X0000000000400511 <+36>: pop    %rbp
   0X0000000000400512 <+37>: retq

End of assembler dump.
```

Inspecting the Assembly Code

```
4   main() {
   0X0000000004004ed <+0>: push   %rbp
   0X0000000004004ee <+1>: mov    %rsp,%rbp
```

Prologue or preamble.

Inspecting the Assembly Code

```
4    main() {
   0X0000000004004ed <+0>: push   %rbp
   0X0000000004004ee <+1>: mov    %rsp,%rbp
```

```
5         int w;
6         y = 20;
=> 0x0000000004004f1 <+4>:  movl   $0x14,0x200b45(%rip)  # 0x601040 <y>

7         w = x + y;
   0X0000000004004fb <+14>: mov    0x200b37(%rip),%edx # 0x601038 <x>
   0X0000000000400501 <+20>: mov    0x200b39(%rip),%eax # 0x601040 <y>
   0X0000000000400507 <+26>: add    %edx,%eax
   0X0000000000400509 <+28>: mov    %eax,-0x4(%rbp)
```

move...

the value 20...

to the address generated by this expression. (which is the address of `y` in the data segment!)

Store the resulting value at the computed address (which is the address of `w`, on the stack!)

Before we sum two values, we have to move them to registers...

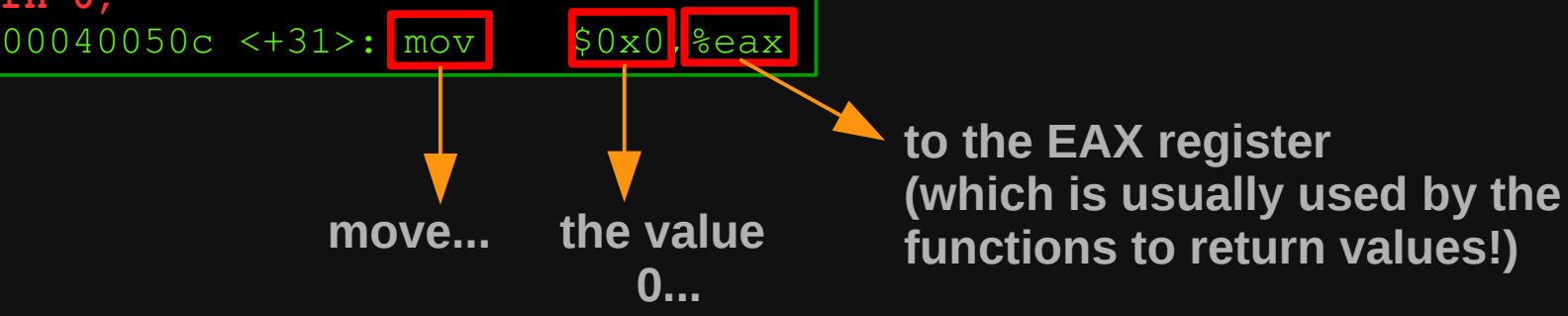
Inspecting the Assembly Code

```
4    main() {
   0X0000000004004ed <+0>: push   %rbp
   0X0000000004004ee <+1>: mov    %rsp,%rbp
```

```
5         int w;
6         y = 20;
=> 0x0000000004004f1 <+4>: movl   $0x14,0x200b45(%rip) # 0x601040 <y>

7         w = x + y;
   0X0000000004004fb <+14>: mov    0x200b37(%rip),%edx # 0x601038 <x>
   0X0000000000400501 <+20>: mov    0x200b39(%rip),%eax # 0x601040 <y>
   0X0000000000400507 <+26>: add    %edx,%eax
   0X0000000000400509 <+28>: mov    %eax,-0x4(%rbp)
```

```
8         return 0;
   0X000000000040050c <+31>: mov    $0x0,%eax
```



Inspecting the Assembly Code

```
4   main() {
   0X0000000004004ed <+0>: push   %rbp
   0X0000000004004ee <+1>: mov    %rsp,%rbp
```

```
5       int w;
6       y = 20;
=> 0x0000000004004f1 <+4>: movl   $0x14,0x200b45(%rip) # 0x601040 <y>

7       w = x + y;
   0X0000000004004fb <+14>: mov    0x200b37(%rip),%edx # 0x601038 <x>
   0X0000000000400501 <+20>: mov    0x200b39(%rip),%eax # 0x601040 <y>
   0X0000000000400507 <+26>: add    %edx,%eax
   0X0000000000400509 <+28>: mov    %eax,-0x4(%rbp)
```

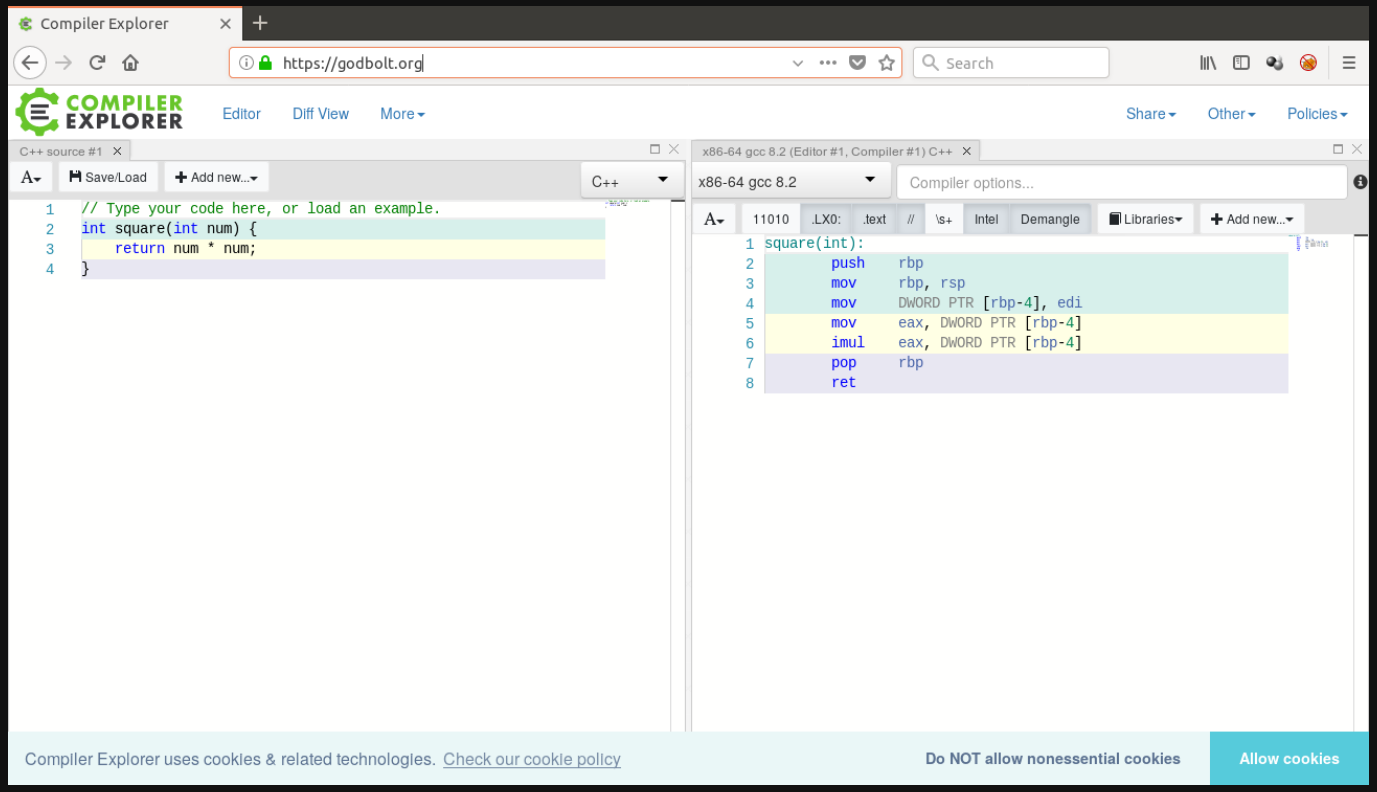
```
8       return 0;
   0X000000000040050c <+31>: mov    $0x0,%eax
```

```
9   }
   0X0000000000400511 <+36>: pop    %rbp
   0X0000000000400512 <+37>: retq
```

Epilogue.

Inspecting the Assembly Code

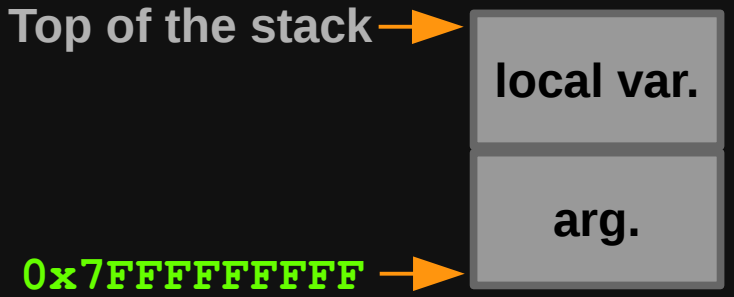
- Cool **C/D/C++/CUDA... to Assembly** online service!
  - https://godbolt.org



Understanding the Stack frames

OS pushes command line arguments and calls `main()`.

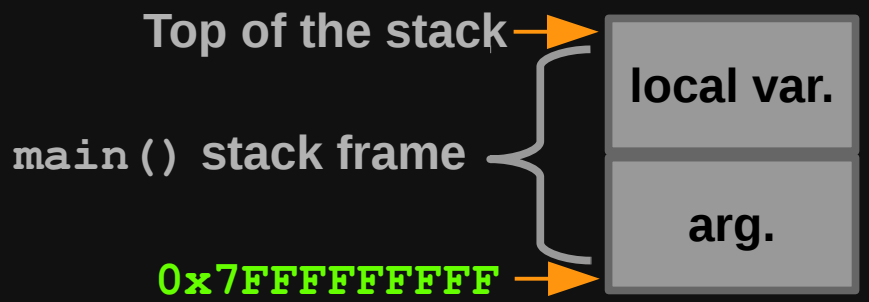
Understanding the Stack frames



`main()` pushes local variables onto the stack.

OS pushes command line arguments and calls `main()`.

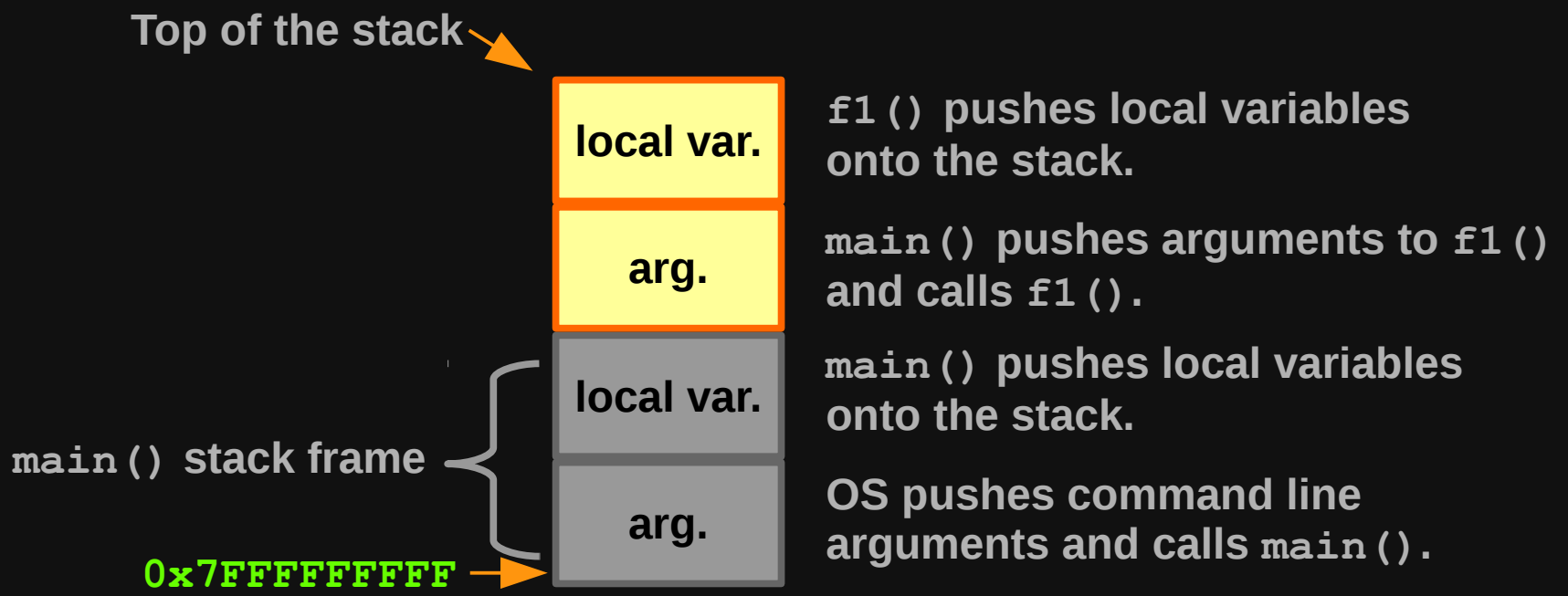
Understanding the Stack frames



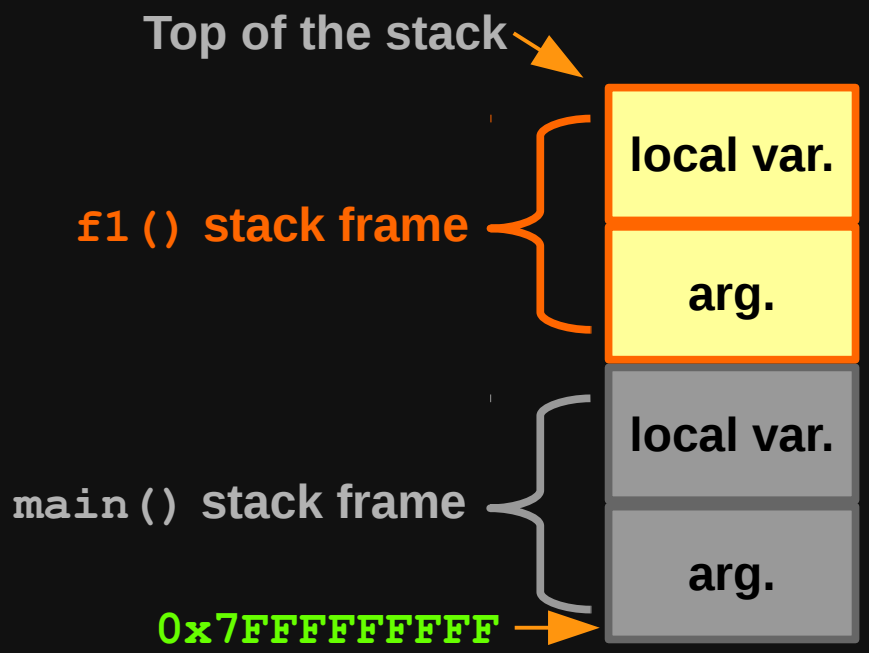
`main()` pushes local variables onto the stack.

OS pushes command line arguments and calls `main()`.

Understanding the Stack frames



Understanding the Stack frames



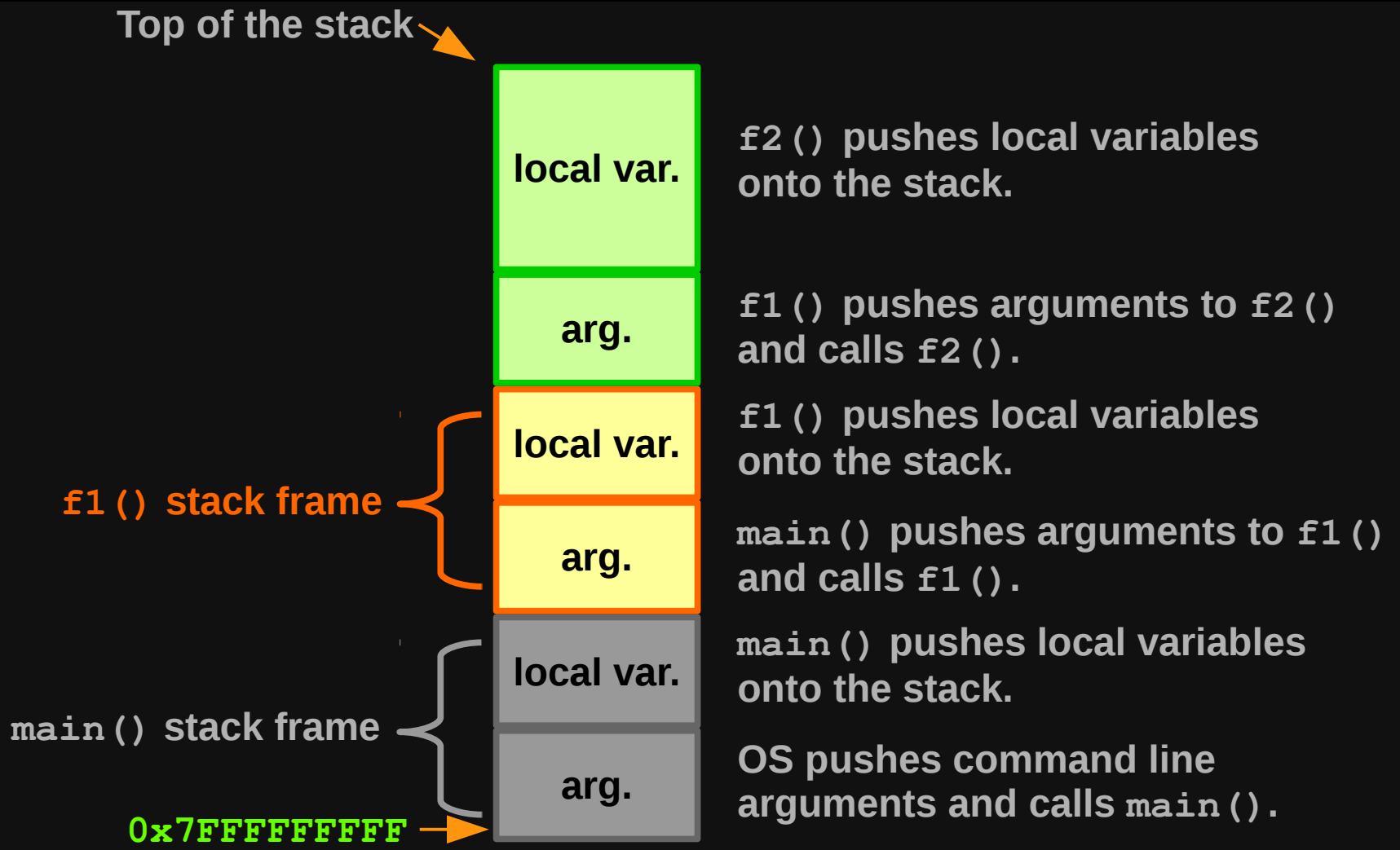
`f1()` pushes local variables onto the stack.

`main()` pushes arguments to `f1()` and calls `f1()`.

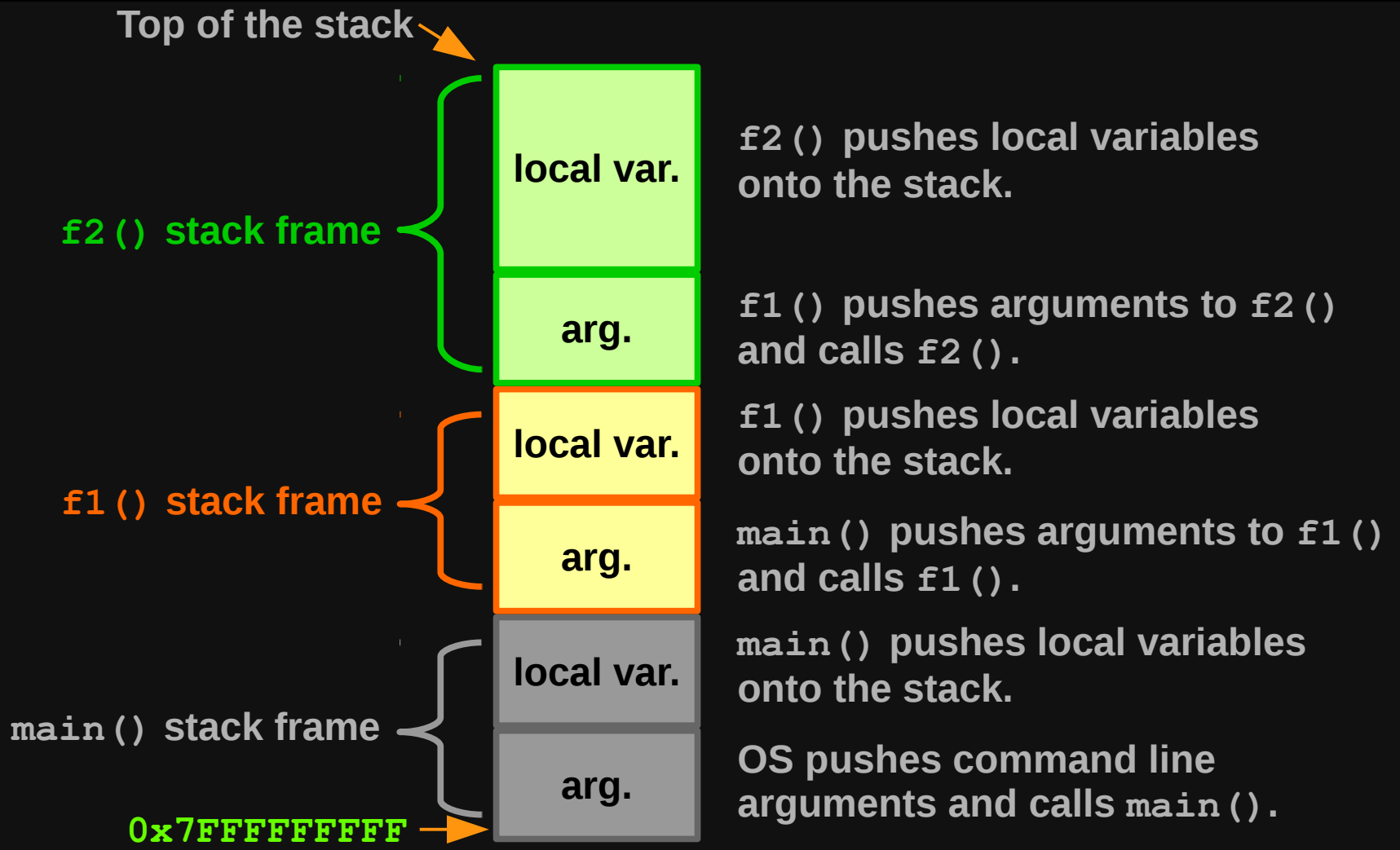
`main()` pushes local variables onto the stack.

OS pushes command line arguments and calls `main()`.

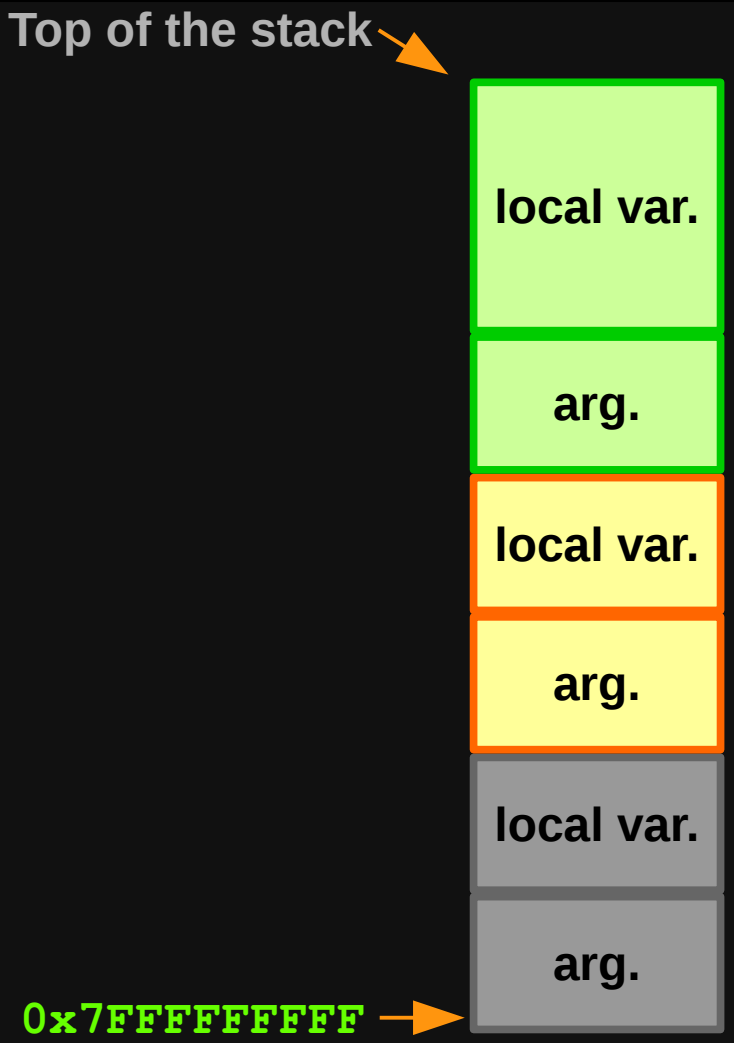
Understanding the Stack frames



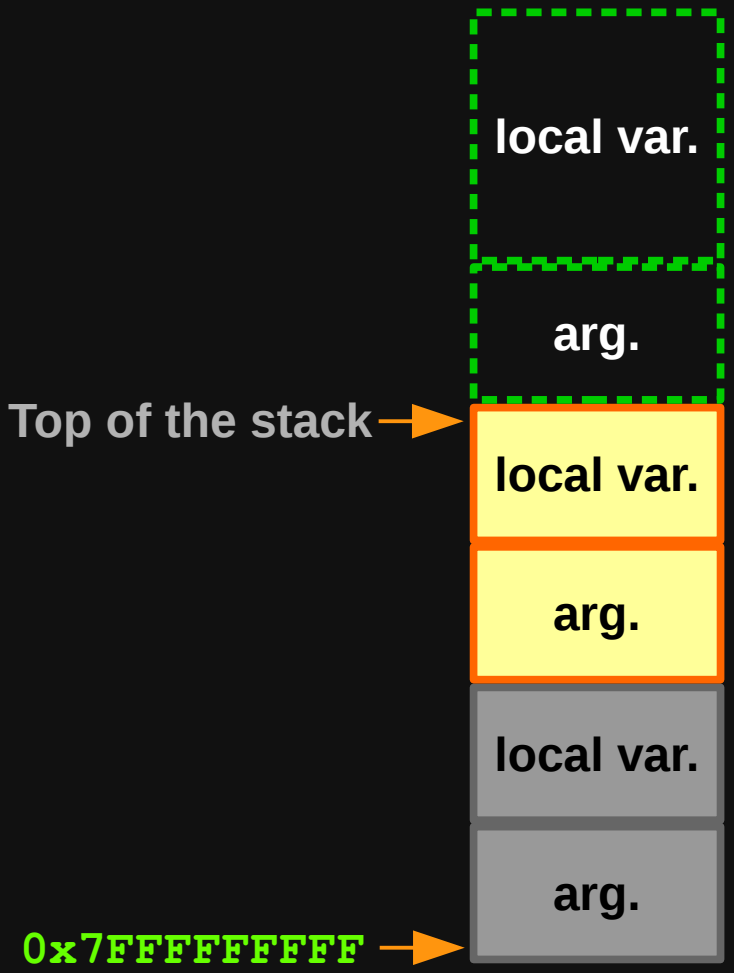
Understanding the Stack frames



Understanding the Stack frames

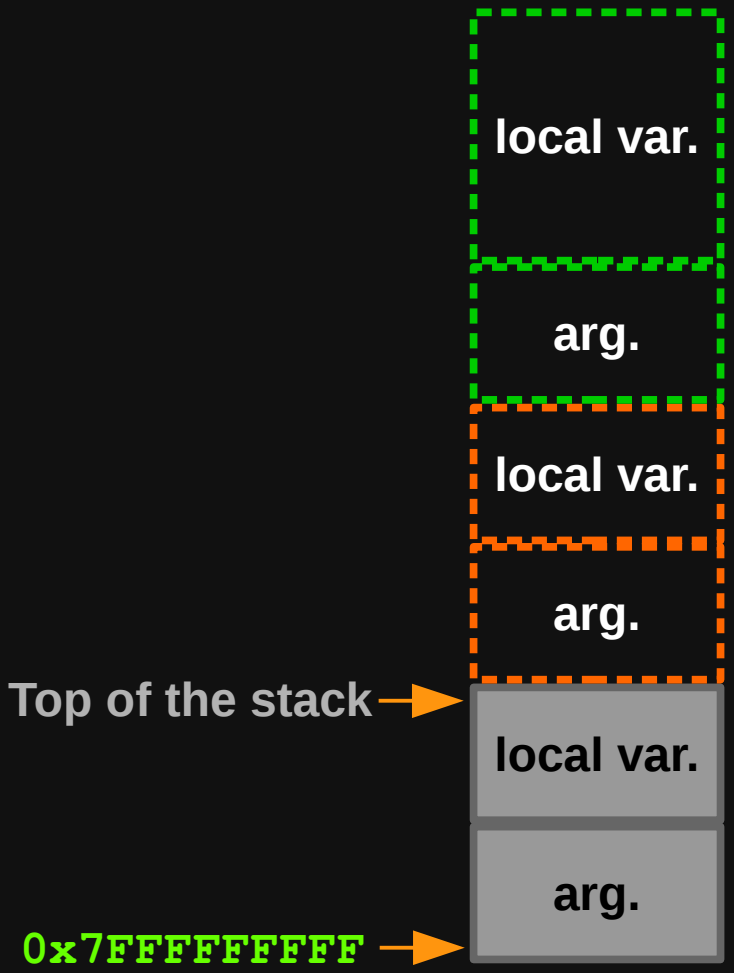


Understanding the Stack frames



`f2()` returns to `f1()`, removing its arguments.

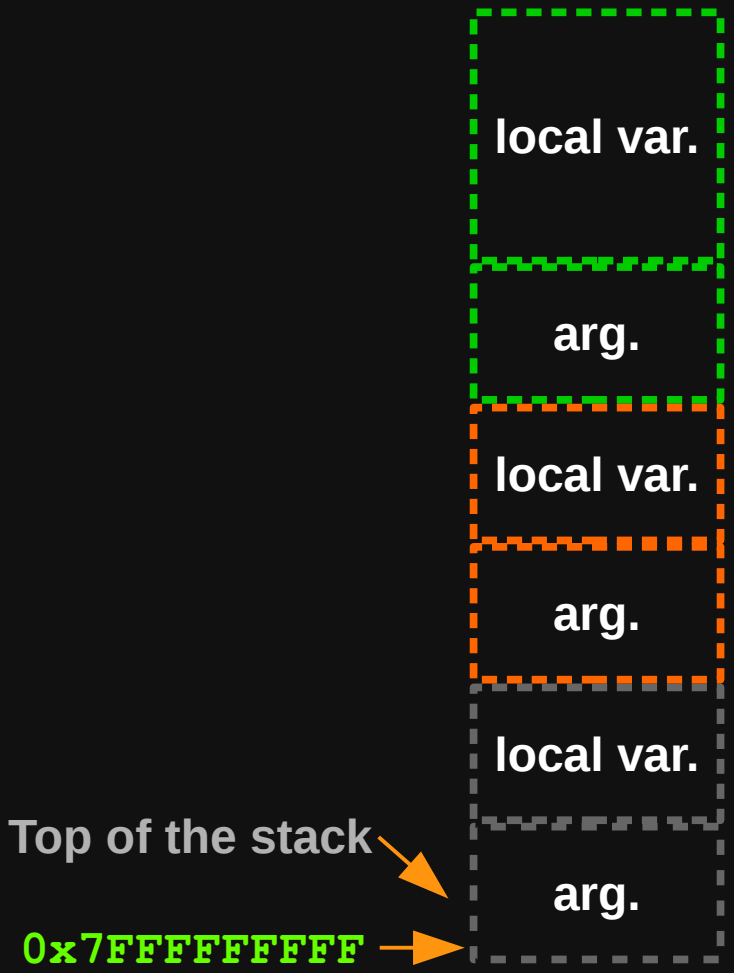
Understanding the Stack frames



`f2()` returns to `f1()`, removing its input arguments.

`f1()` returns to `main()`, removing its input arguments.

Understanding the Stack frames



`f2()` returns to `f1()`, removing its input arguments.

`f1()` returns to `main()`, removing its input arguments.

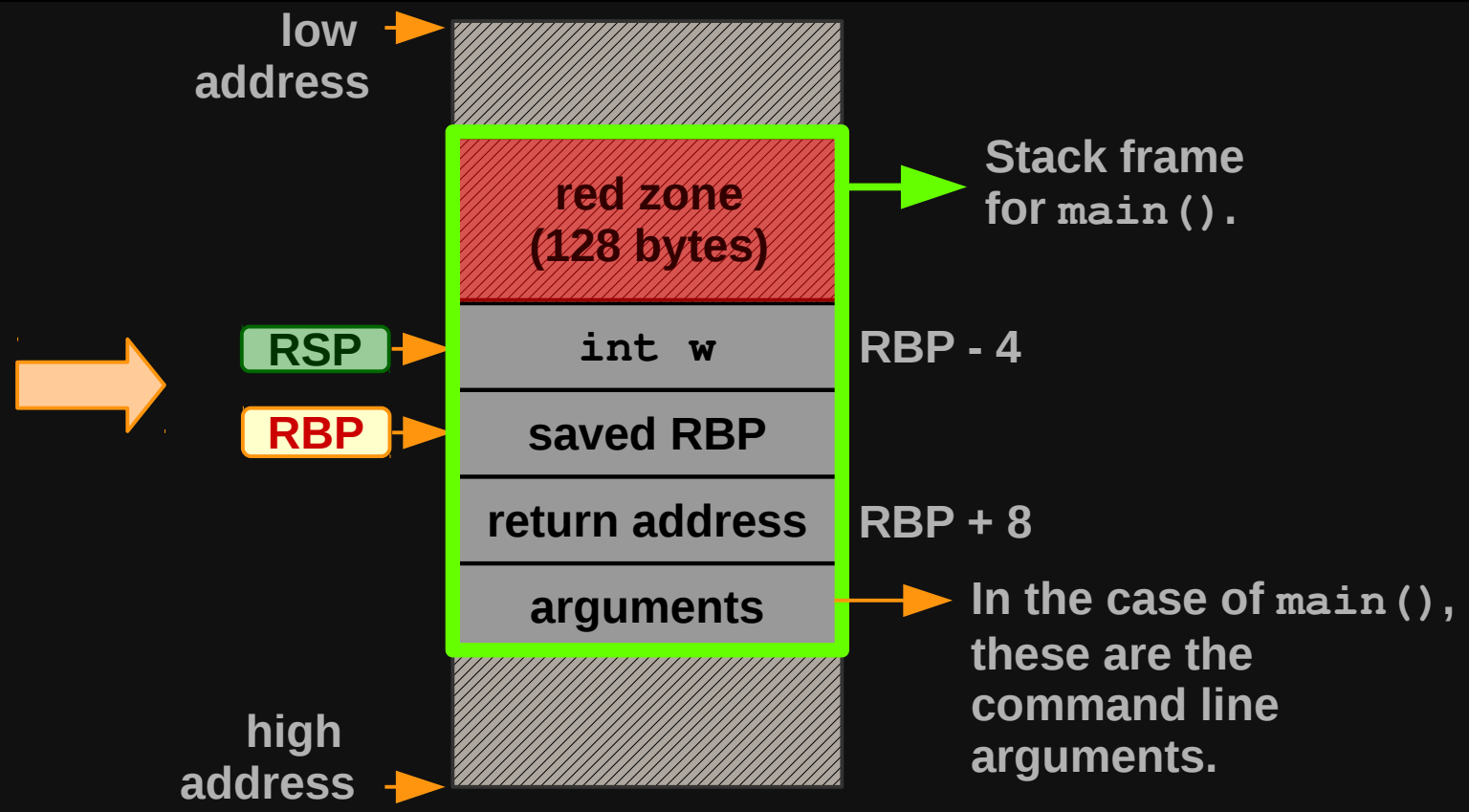
`main()` returns to OS.

A More Realistic `main()` Stack Frame

assembly1.c

```
int x = 10;
int y;

main() {
    int w;
    y = 20;
    w = x + y;
    return 0;
}
```



For those that want to understand all the nitty-gritty about **stacks**, **parameter passing** and **etc.** on the **x86-64**, check the **System V ABI** spec!

Application Binary Interface

- An ABI specifies:
  - Calling convention.
  - Object file formats.
  - Executable file formats.
  - Dynamic linking semantics.
  - Etc.
- Linux uses the **System V ABI**
  - The Executable and Linkable Format (ELF), which is used by Linux, is part of the System V ABI.

System V ABI for x86-64

- The stack grows towards **lower addresses**.
- Parameters to functions are passed in the **registers**:
  - **`rdi`**, **`rsi`**, **`rdx`**, **`rcx`**, **`r8`**, **`r9`** (further values are passed on the stack in reverse order).
- **`call`** instruction pushes the **return address** to the **stack**.
- **`ret`** instruction pops the **return address** from the **stack** and **jump** to it.
- **Return value** is stored in **`rax`** register.
- Etc.

Inspecting Stack Frames with GDB

- GDB has some commands for the **inspection** of the **current** stack frame:
  - `info frame`
  - `info args`
  - `info locals`
- To **change** the stack frame:
  - `frame <number>`
- To print the **current** value of the **stack pointer**:
  - `p/x $esp`

Exercise: Inspecting the Stack

assembly2.c

```c
int x = 10;

int f2( int c ) {
    int d = c + 100;
    return d;
}

int f1( int a ) {
    int b = a + f2( a
);
    return b;
}

int main() {
    int w = f1( x );
    return 0;
}
```

DIY!

Inspect the stack with the following **commands**:

- **`info locals`**
- **`info args`**
- **`info frame`**

Draw a **diagram** that depicts the **distribution** of data **within** the stack

Experiment the above procedures with **different function parameters**!

- More than 6 parameters.
- Parameters by value and reference
- Structs
- Arrays....