Practical Assembly

From the Ground Up

John Cunniff
OSIRIS Lab Hack Night

Overview - Rev 2

- Overview
  - Who is this for?
  - Whats assembly?
  - Why should we learn it?
  - Required Mindset
- Basic Assembly topics
  - Memory
  - Registers
  - Basic instructions
- Advanced Topics
  - Registers (again)
  - Memory layout
  - Stack Frame
- Helpful tools!
  - objdump
  - Binary Ninja
  - godbolt.org

Overview

Who is this for?

- People new to assembly

Who is this for?

- People new to assembly
- Anyone that wants a better understanding of how computers work!

Who is this for?

- People new to assembly
- Anyone that wants a better understanding of how computers work!
- Those that want to start playing reverse engineering / pwning challenges

Who is this for?

- People new to assembly
- Anyone that wants a better understanding of how computers work!
- Those that want to start playing reverse engineering / pwning challenges
    - We're going to start with very abstract concepts

Who is this for?

- People new to assembly
- Anyone that wants a better understanding of how computers work!
- Those that want to start playing reverse engineering / pwning challenges
  - We're going to start with very abstract concepts
  - Build up to what it actually looks like

What is assembly?

What is assembly?

- Have you ever wondered how a machine can run code?

What is assembly?

- Have you ever wondered how a machine can run code?
  - The answer is “machine code”

What is assembly?

- Have you ever wondered how a machine can run code?
    - The answer is “machine code”
    - Machine code is translated from Assembly

What is assembly?

- Have you ever wondered how a machine can run code?
  - The answer is “machine code”
  - Machine code is translated from Assembly
- Machine code is what your cpu is able to actually interpret and run

What is assembly?

- Have you ever wondered how a machine can run code?
  - The answer is “machine code”
  - Machine code is translated from Assembly
- Machine code is what your cpu is able to actually interpret and run
- The instructions can let you modify the cpu, memory and other physical devices *directly*

Why should we learn assembly?

Why should we care at all?

Why should we learn assembly?

- There are basically no rules!

Why should we learn assembly?

- There are basically no rules!
- Bugs at this low level lead to vulnerabilities

Why should we learn assembly?

- There are basically no rules!
- Bugs at this low level lead to vulnerabilities
- Better our understanding of how computers work!

Why should we learn assembly?

- There are basically no rules!
- Bugs at this low level lead to vulnerabilities
- Better our understanding of how computers work!
    - Everything that runs has to be in one way or another translated back to assembly

Why should we learn assembly?

- There are basically no rules!
- Bugs at this low level lead to vulnerabilities
- Better our understanding of how computers work!
    - Everything that runs has to be in one way or another translated back to assembly
    - There is no avoiding it!

Why should we learn assembly?

- There are basically no rules!
- Bugs at this low level lead to vulnerabilities
- Better our understanding of how computers work!
  - Everything that runs has to be in one way or another translated back to assembly
  - There is no avoiding it!
  - The computer you're viewing this lecture from, the stream your watching it on, the servers all that data is passing through, all assembly

Why should we learn assembly?

- Assembly can’t lie….

Why should we learn assembly?

- Assembly can’t lie….
- Have you ever paid for an application that you have downloaded?

Why should we learn assembly?

- Assembly can’t lie….
- Have you ever paid for an application that you have downloaded?
    - That application will be a compiled program of “machine code”

Why should we learn assembly?

- Assembly can’t lie….
- Have you ever paid for an application that you have downloaded?
  - That application will be a compiled program of “machine code”
  - Using special tools we can figure out how the application works!

Why should we learn assembly?

- Assembly can't lie….
- Have you ever paid for an application that you have downloaded?
    - That application will be a compiled program of "machine code"
    - Using special tools we can figure out how the application works!
    - This is called "reverse engineering"

Required Mindset

Going into assembly

Required Mindset

- I’m going to ask you to do something difficult during this lecture

Required Mindset

- I'm going to ask you to do something difficult during this lecture
  - Let go of whatever understanding you have of how programs work (in python, c++, java, go, etc.)

Required Mindset

- I’m going to ask you to do something difficult during this lecture
  - Let go of whatever understanding you have of how programs work (in python, c++, java, go, etc.)
  - Instead try to conceptualize this as a totally new way of thinking

Required Mindset

- I’m going to ask you to do something difficult during this lecture
    - Let go of whatever understanding you have of how programs work (in python, c++, java, go, etc.)
    - Instead try to conceptualize this as a totally new way of thinking
    - Don’t try to connect this new way of thinking to your current understanding (yet)

Required Mindset

- I’m going to ask you to do something difficult during this lecture
  - Let go of whatever understanding you have of how programs work (in python, c++, java, go, etc.)
  - Instead try to conceptualize this as a totally new way of thinking
  - Don’t try to connect this new way of thinking to your current understanding (yet)
  - Once you understand these systems we’ll connect it back to C specifically

Required Mindset

- I’m going to ask you to do something difficult during this lecture
    - Let go of whatever understanding you have of how programs work (in python, c++, java, go, etc.)
    - Instead try to conceptualize this as a totally new way of thinking
    - Don’t try to connect this new way of thinking to your current understanding (yet)
    - Once you understand these systems we’ll connect it back to C specifically
        - C is easily readable and can be basically directly connected back to assembly

Required Mindset

- We're going to be using Intel syntax, **NOT** AT&T syntax

Required Mindset

- We’re going to be using Intel syntax, **NOT** AT&T syntax
- This lecture will focus on 32bit x86

Required Mindset

- We’re going to be using Intel syntax, **NOT** AT&T syntax
- This lecture will focus on 32bit x86
  - Almost all modern computers are 64bit now

Required Mindset

- We're going to be using Intel syntax, **NOT** AT&T syntax
- This lecture will focus on 32bit x86
  - Almost all modern computers are 64bit now
  - Only small differences!

The basics

Memory

The basics - Memory

- Think of memory as one long array…

The basics - Memory

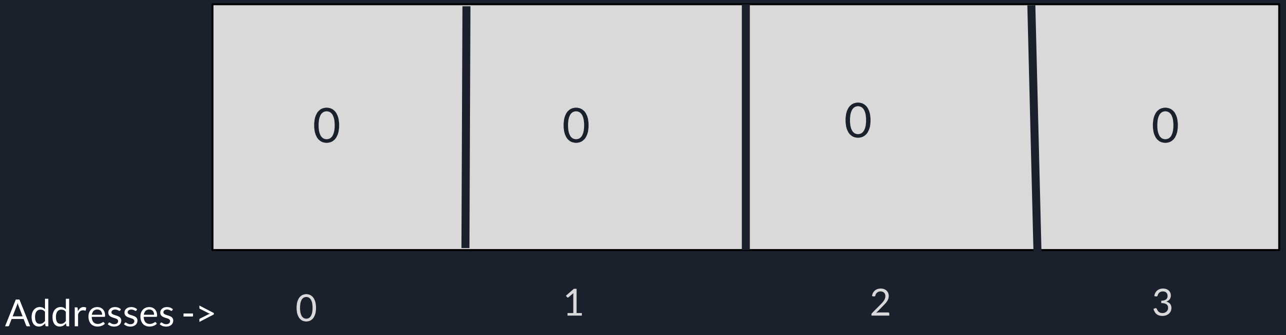
- Think of memory as one long array…
- This array has indices (called addresses)

The basics - Memory

- Think of memory as one long array…
- This array has indices (called addresses)
- Same as with any program, you can't read outside the array!

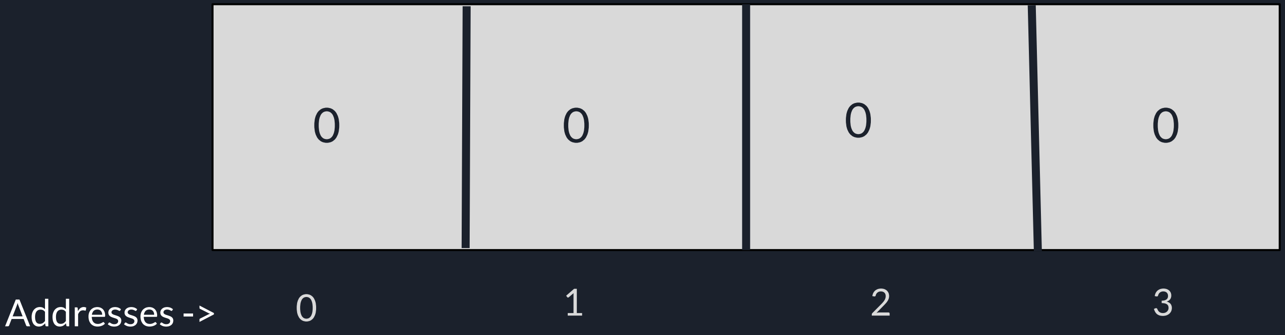
The basics - Memory

- Think of memory as one long array…
- This array has indices (called addresses)
- Same as with any program, you can't read outside the array!



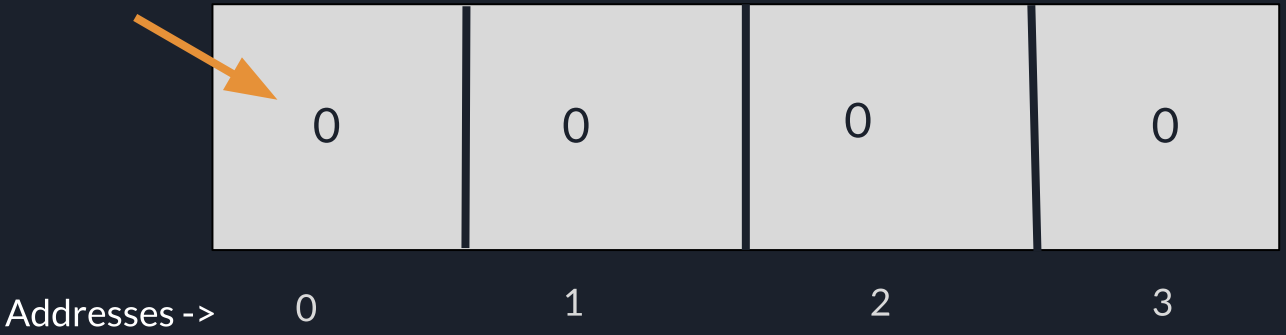
The basics - Memory

- In assembly we move values in and out of memory one at a time



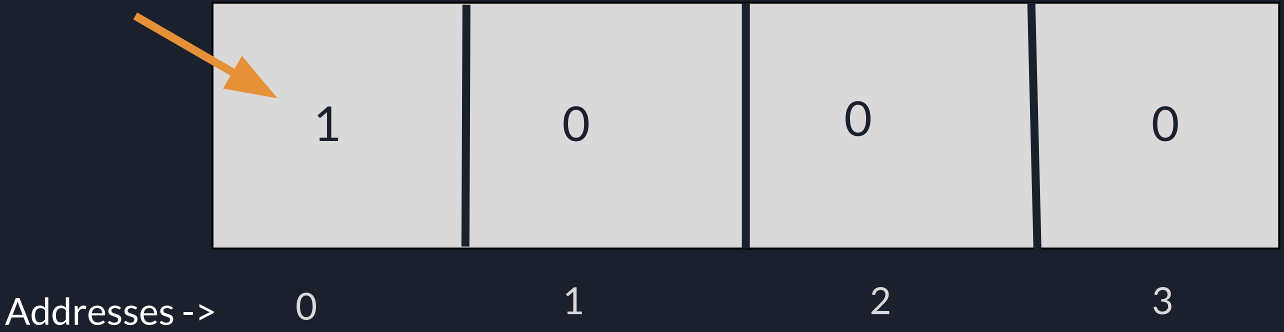
The basics - Memory

- In assembly we move values in and out of memory one at a time
- We can mov 1 into address 0



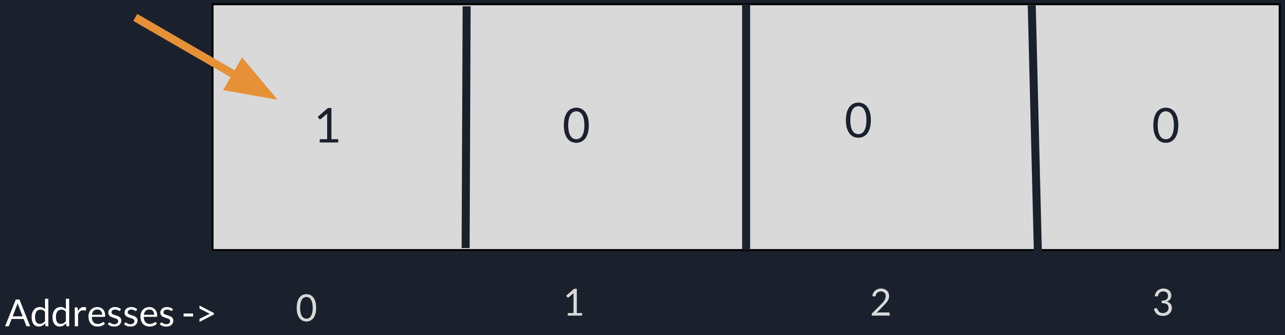
The basics - Memory

- In assembly we move values in and out of memory one at a time
- We can “mov” 1 into address 0



The basics - Memory

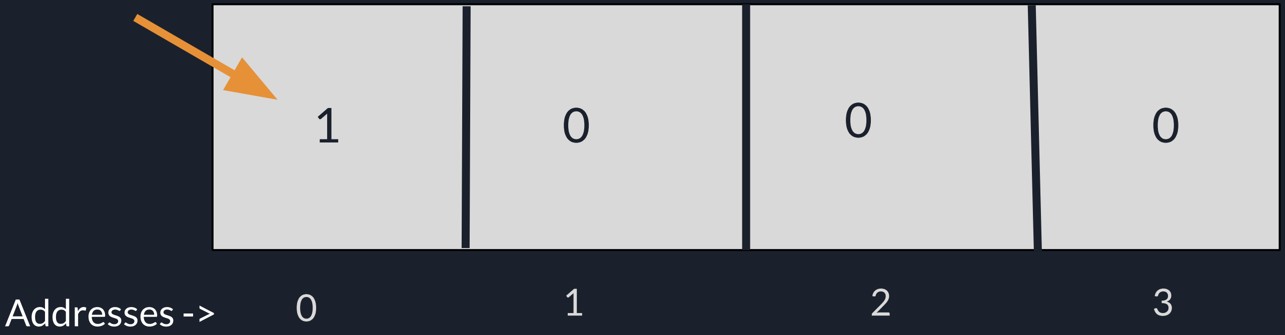
- In assembly we move values in and out of memory one at a time
- We can "mov" 1 into address 0
- In assembly this would be: mov [0], 1



The basics - Memory

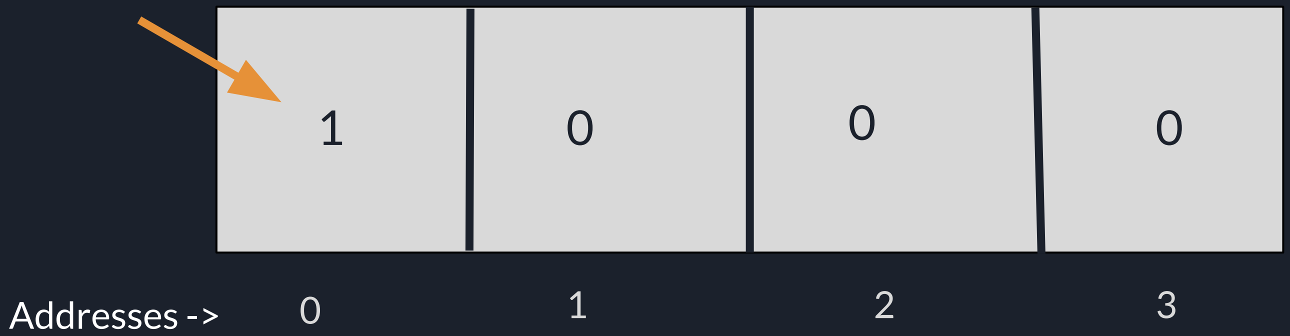
The brackets mean "dereference"

- mov [destination address], source



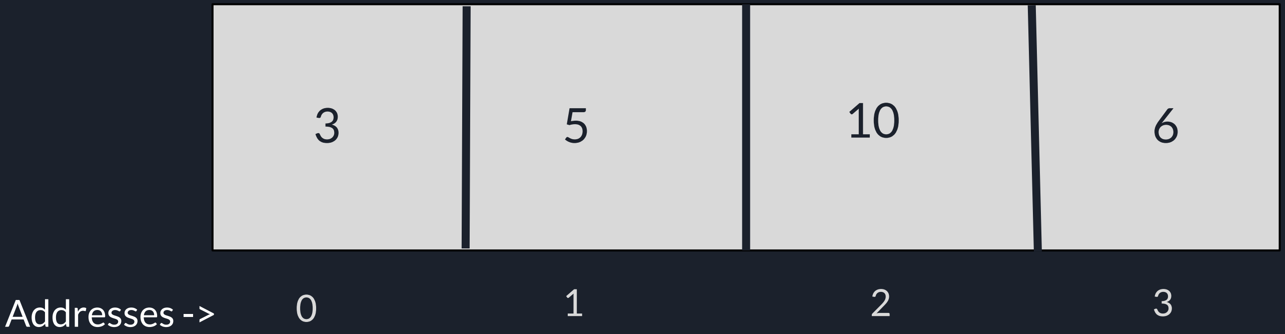
The basics - Memory

- mov [destination address], source
- Where destination address is the index (0) and source is the value of the new number (1)



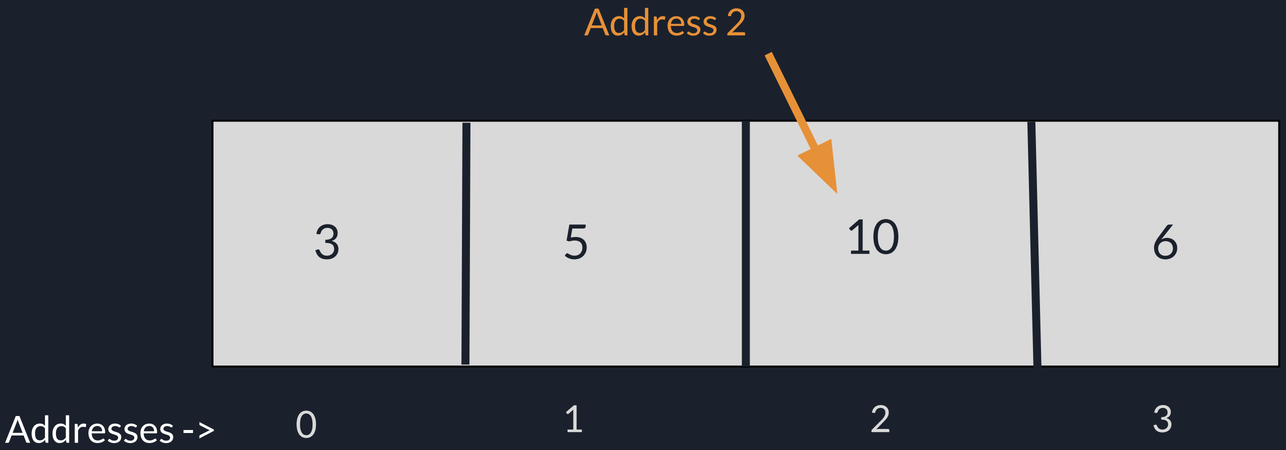
The basics - Memory

- So what will “mov [2], 3” do?



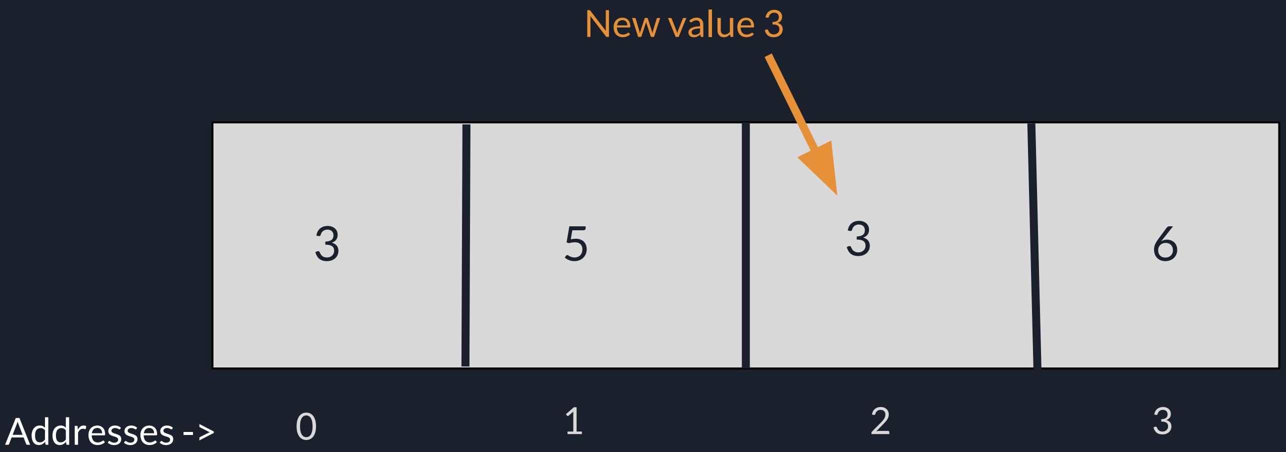
The basics - Memory

- So what will “mov [2], 3” do?



The basics - Memory

- So what will “mov [2], 3” do?



The basics

Registers

Registers

- The biggest mistake you can make is thinking of registers as variables…

Registers

- The biggest mistake you can make is thinking of registers as variables…
- Registers are the physical memory locations on the CPU

Registers

- The biggest mistake you can make is thinking of registers as variables…
- Registers are the physical memory locations on the CPU
- Hold small amounts of data

Registers

- The biggest mistake you can make is thinking of registers as variables…
- Registers are the physical memory locations on the CPU
- Hold small amounts of data
- Perform super fast operations

Registers

- The biggest mistake you can make is thinking of registers as variables…
- Registers are the physical memory locations on the CPU
- Hold small amounts of data
- Perform super fast operations
  - Much faster than memory

Registers

- Let's say we have a register “a”

Registers

- Let's say we have a register “a”
- This register a can hold 8 bits or 1 byte of data

Registers

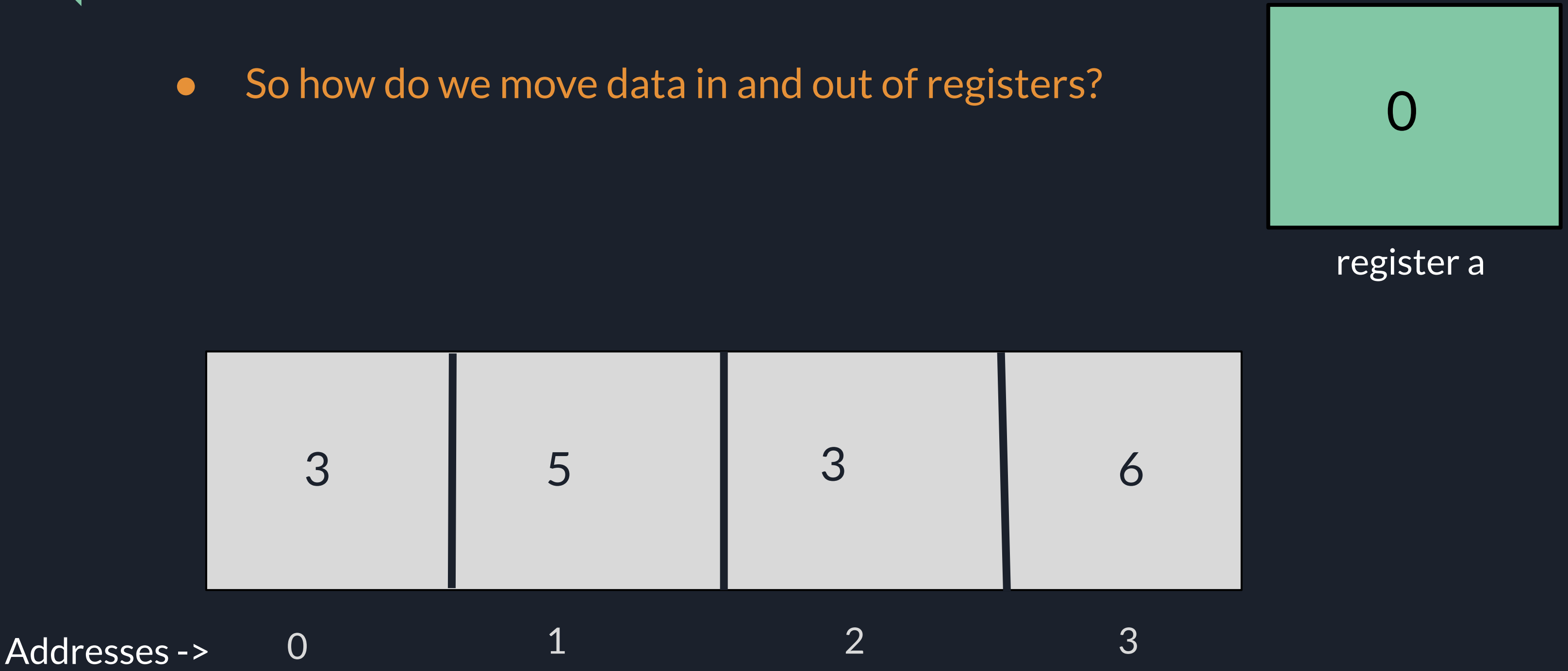
- Let's say we have a register “a”
- This register a can hold 8 bits or 1 byte of data
- That means we can have values between 0 and 255 in a

Registers

- Let's say we have a register “a”
- This register a can hold 8 bits or 1 byte of data
- That means we can have values between 0 and 255 in a
- So how do we move values in and out of registers?

The basics - Registers

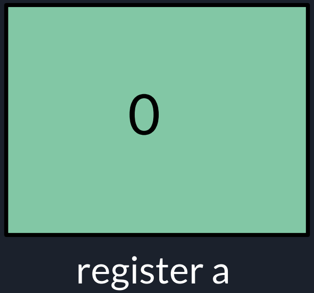
- So how do we move data in and out of registers?

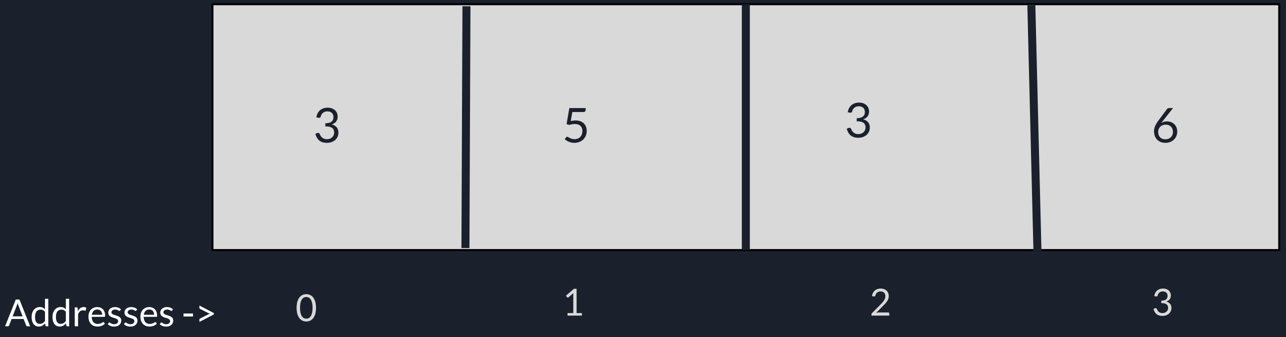


The basics - Registers

Remember its:
mov dst, src

- So how do we move data in and out of registers?
- mov a, 10

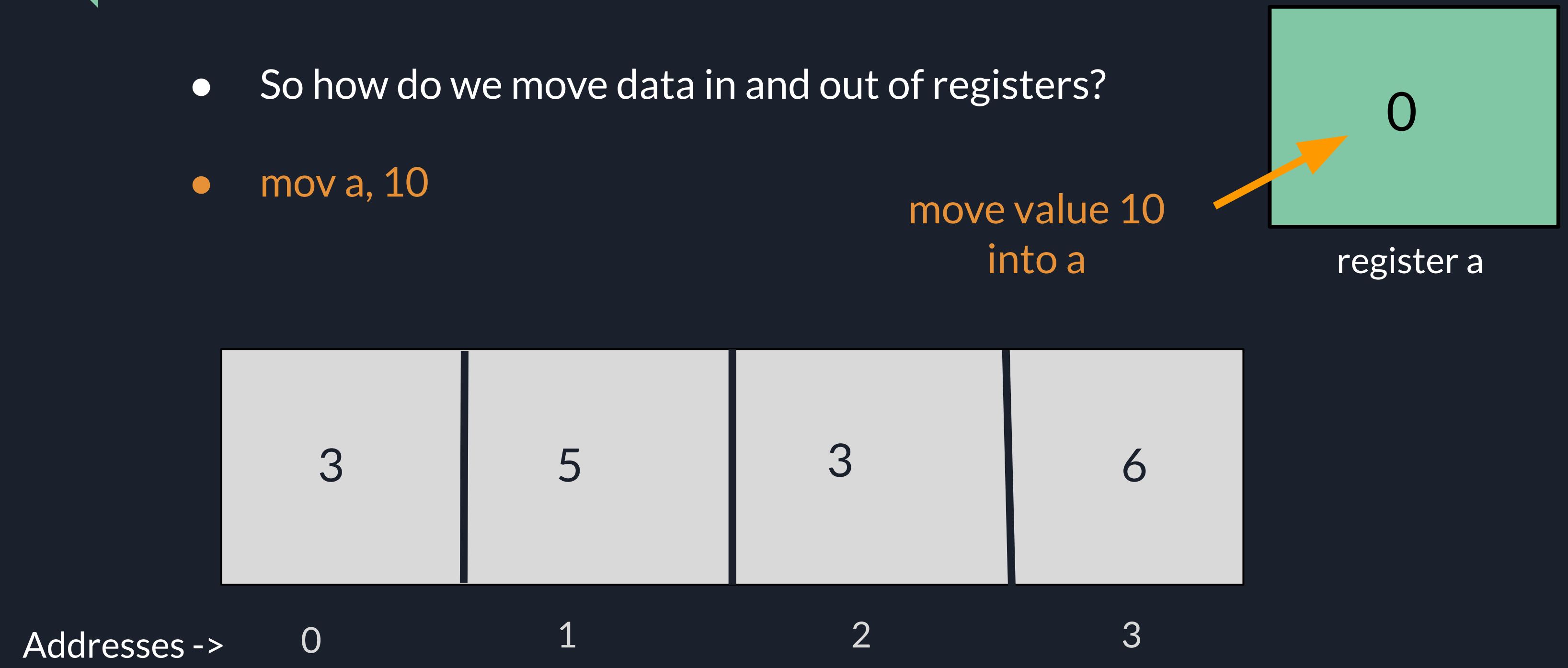




The basics - Registers

Remember its:
mov dst, src

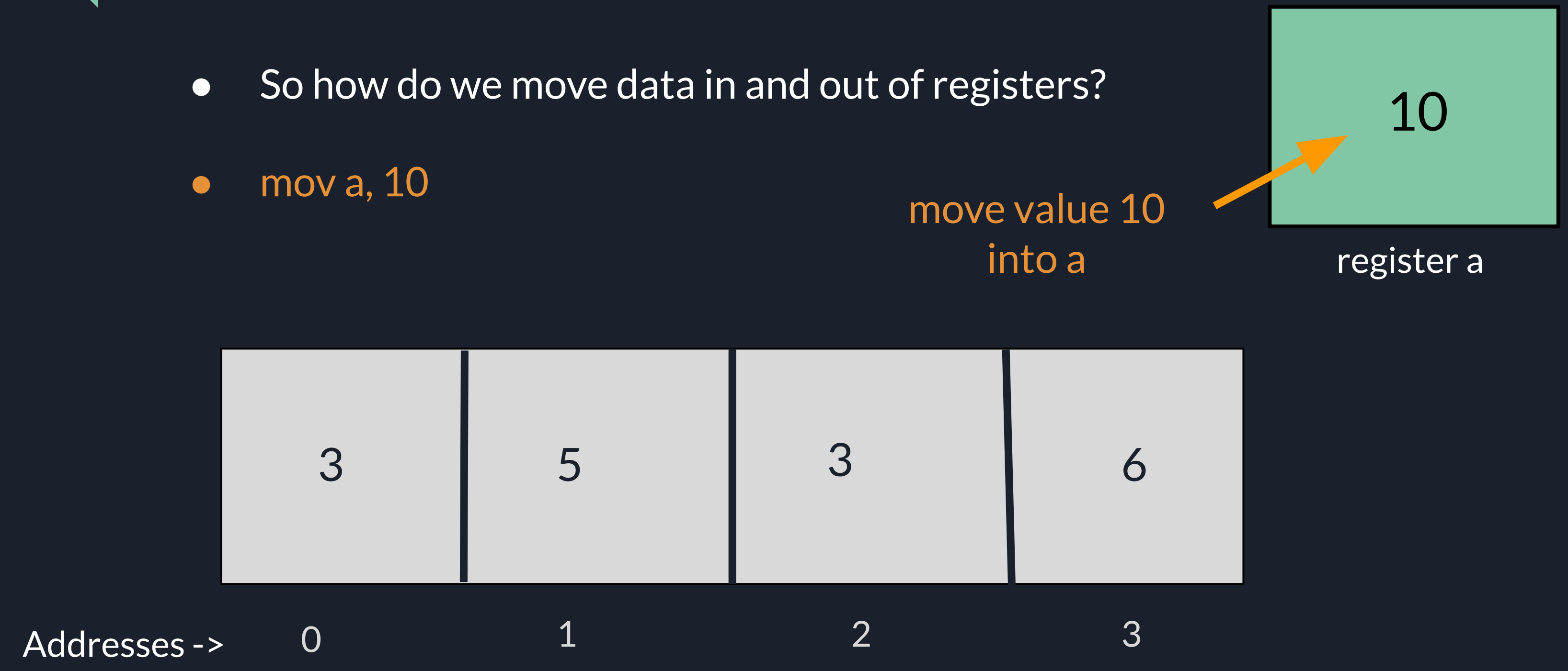
- So how do we move data in and out of registers?
- mov a, 10



The basics - Registers

Remember its:
mov dst, src

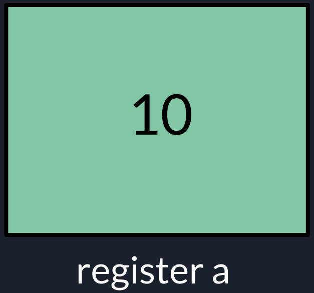
- So how do we move data in and out of registers?
- mov a, 10

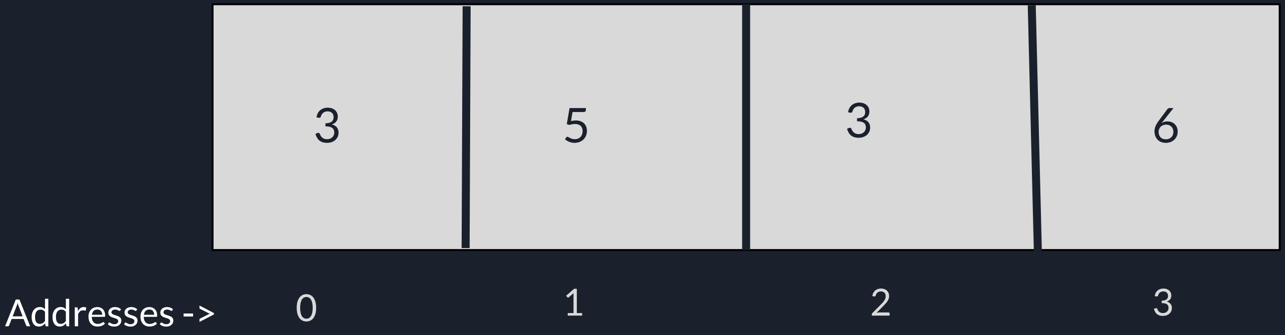


The basics - Registers

Remember its:
mov dst, src

- What about moving things between memory?

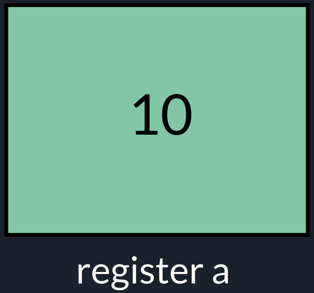


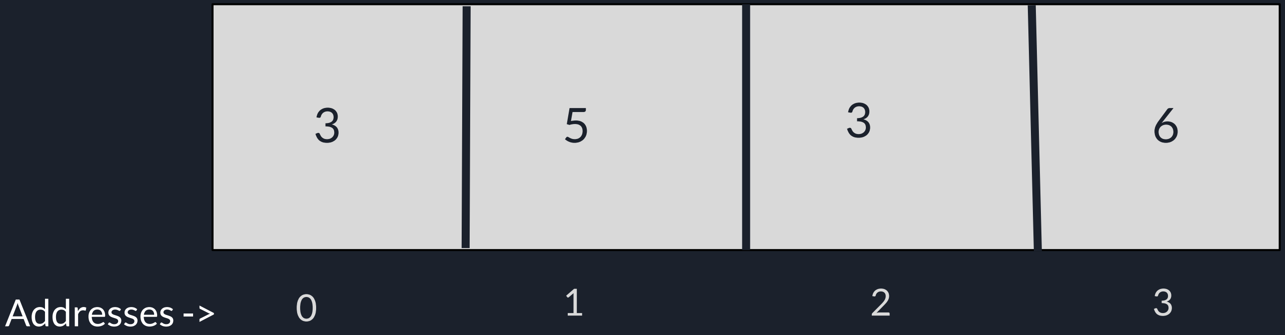


The basics - Registers

Remember its:
mov dst, src

- What about moving things between memory?
- mov a, [0]



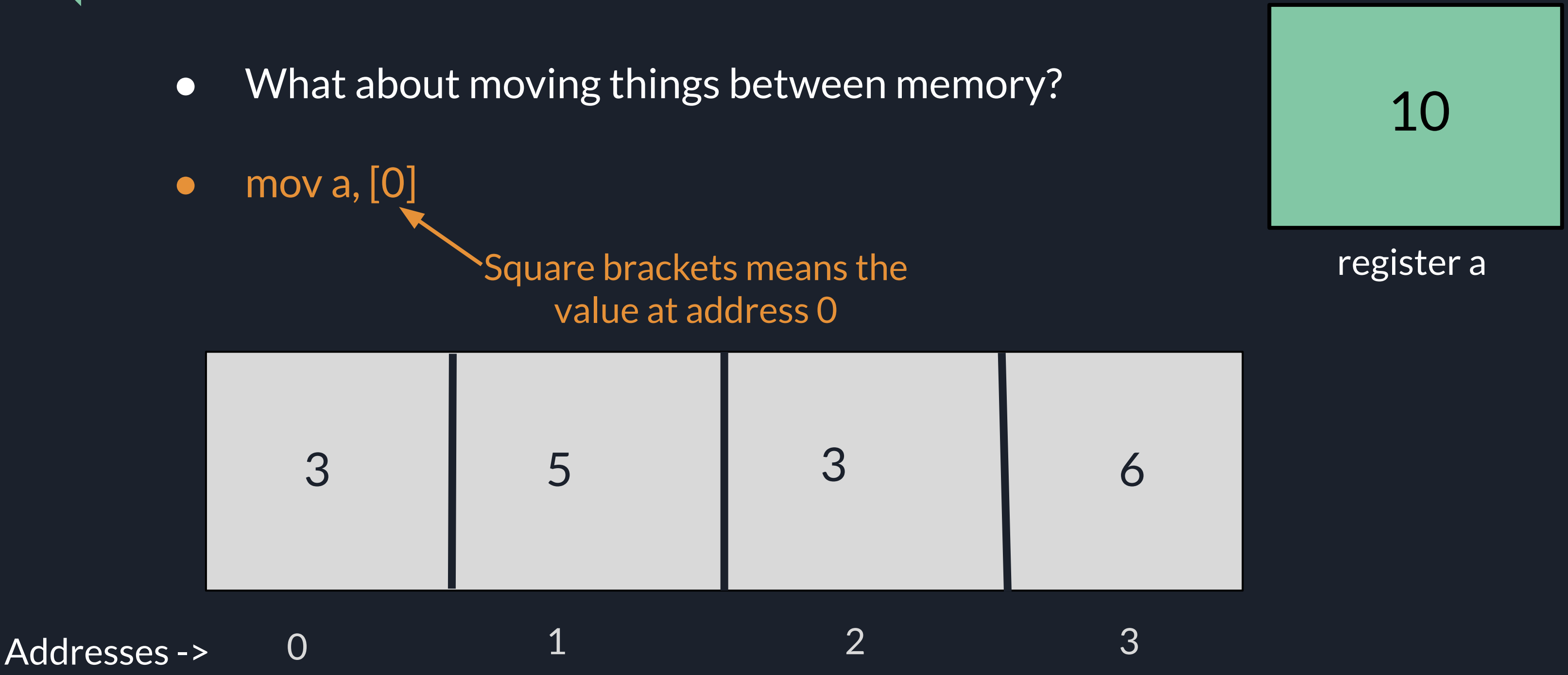


The basics - Registers

Remember its:
mov dst, src

- What about moving things between memory?
- mov a, [0]

Square brackets means the value at address 0

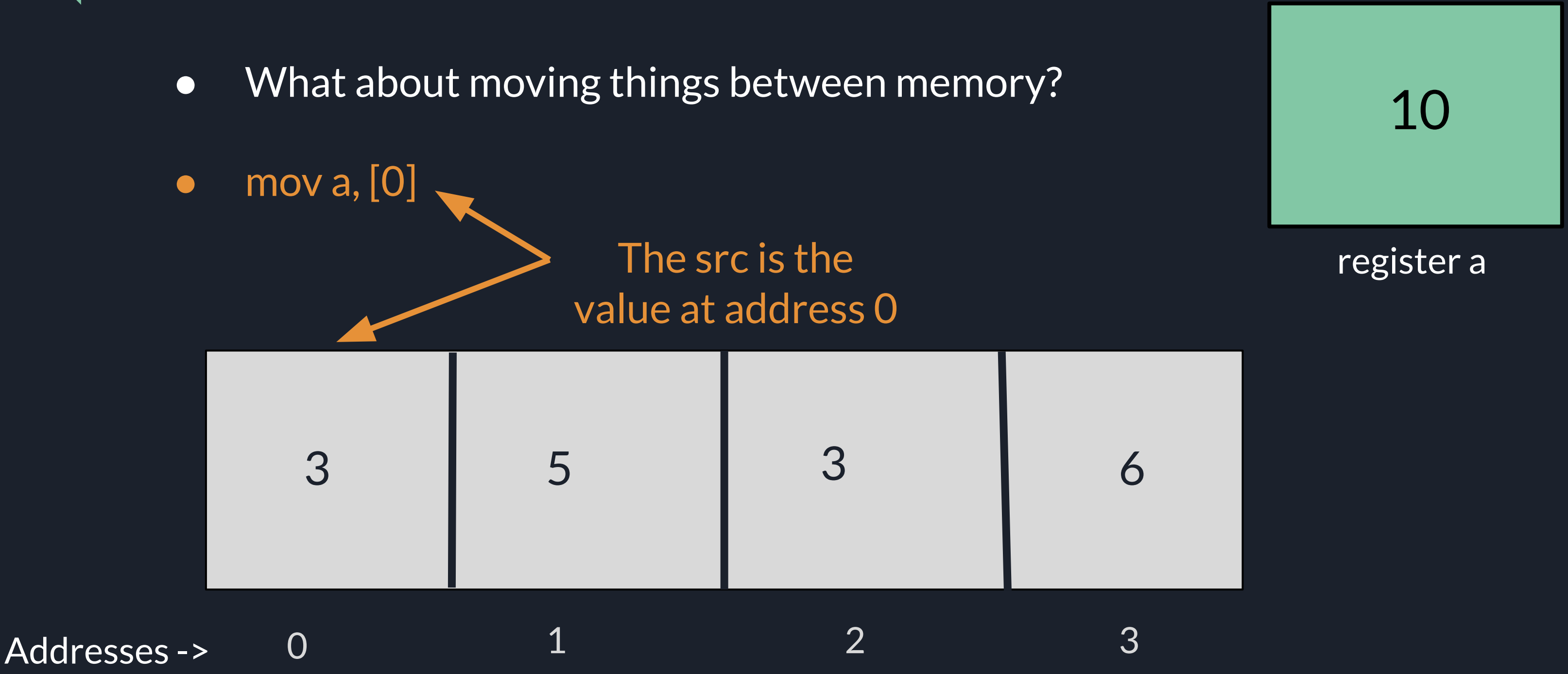


The basics - Registers

Remember its:
mov dst, src

- What about moving things between memory?
- mov a, [0]

The src is the value at address 0

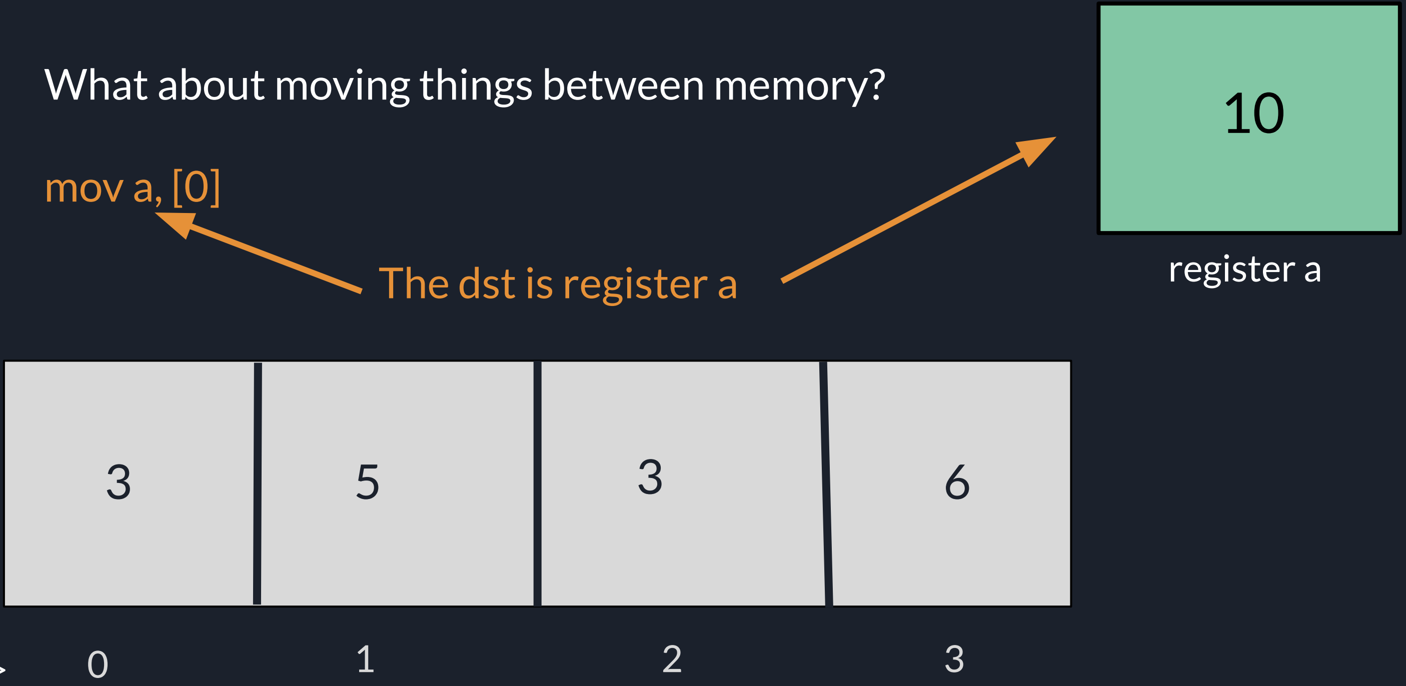


The basics - Registers

Remember its:
mov dst, src

- What about moving things between memory?
- mov a, [0]

The dst is register a

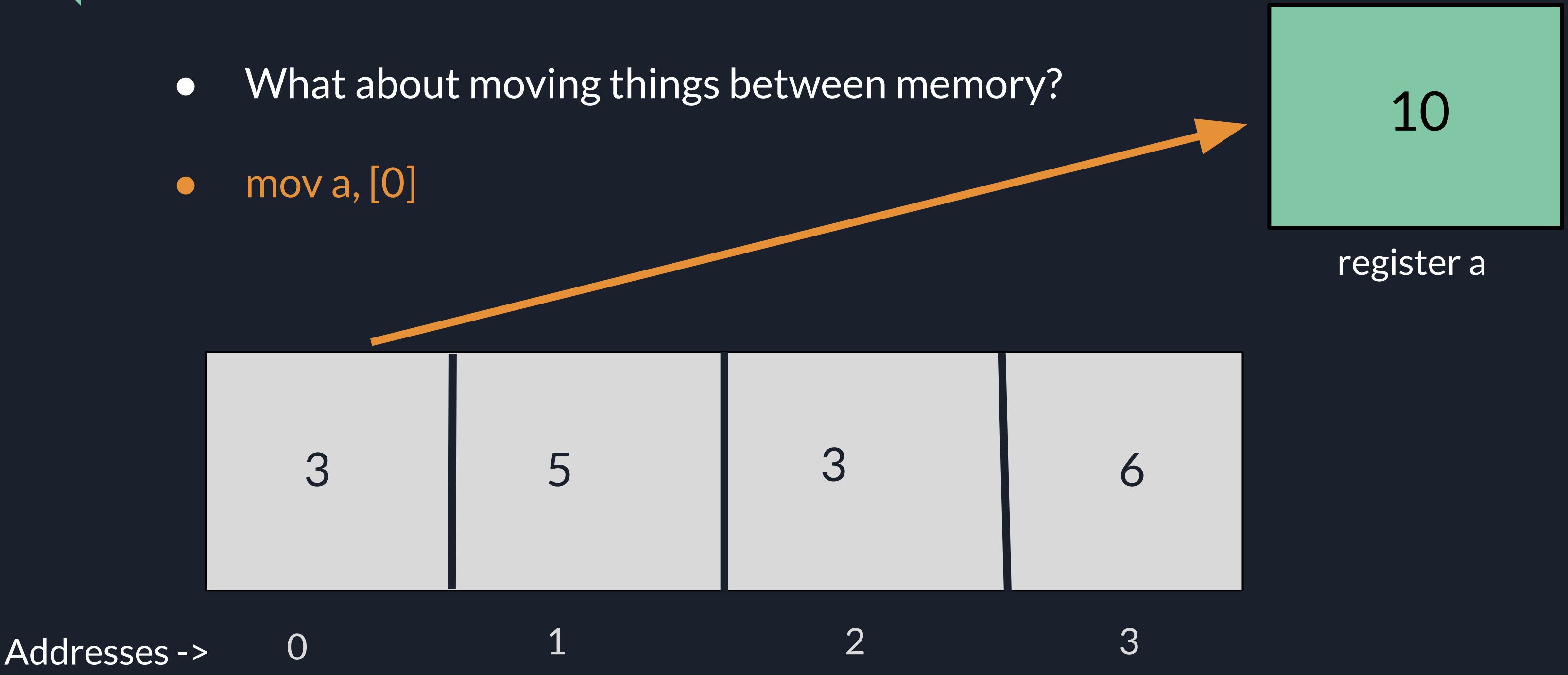


Addresses ->

The basics - Registers

Remember its:
mov dst, src

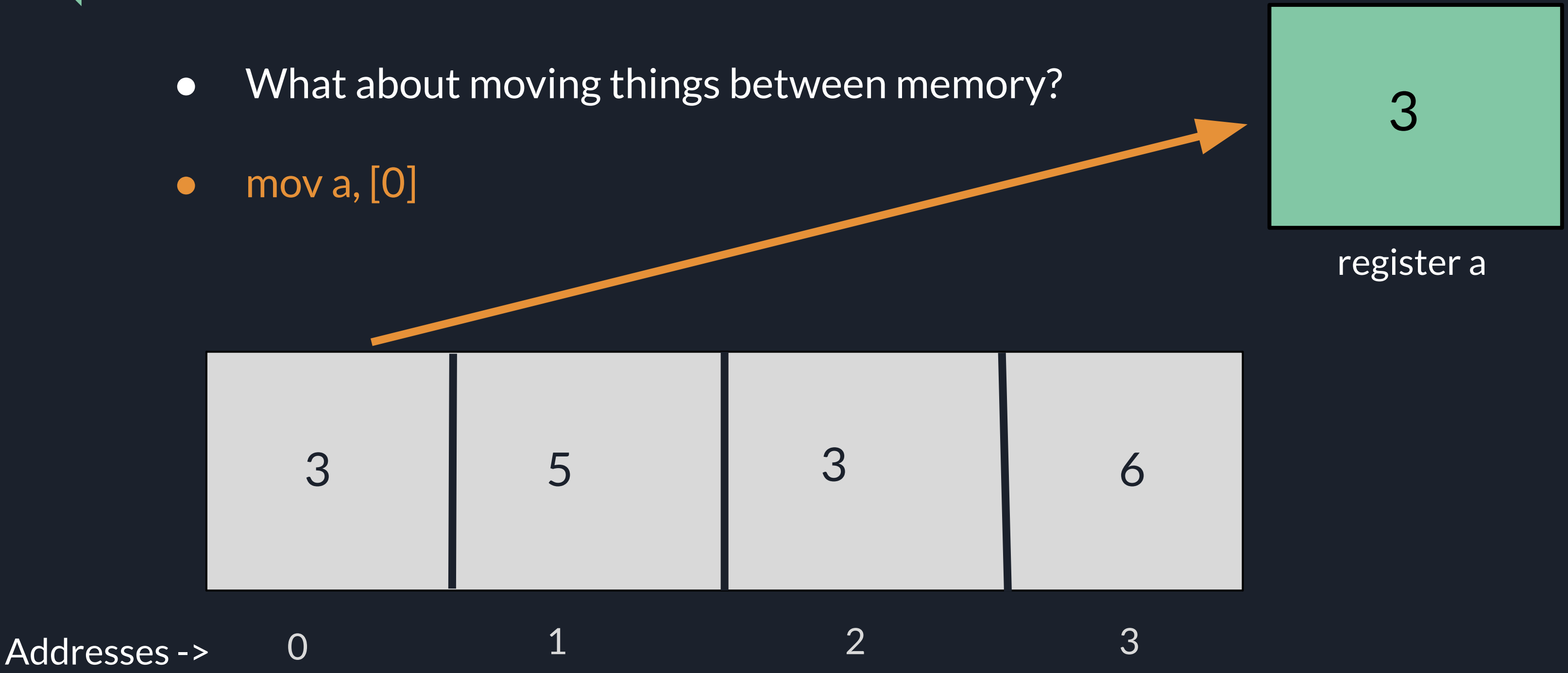
- What about moving things between memory?
- mov a, [0]



The basics - Registers

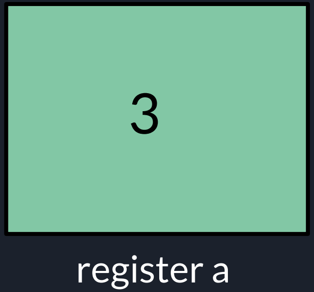
Remember its:
mov dst, src

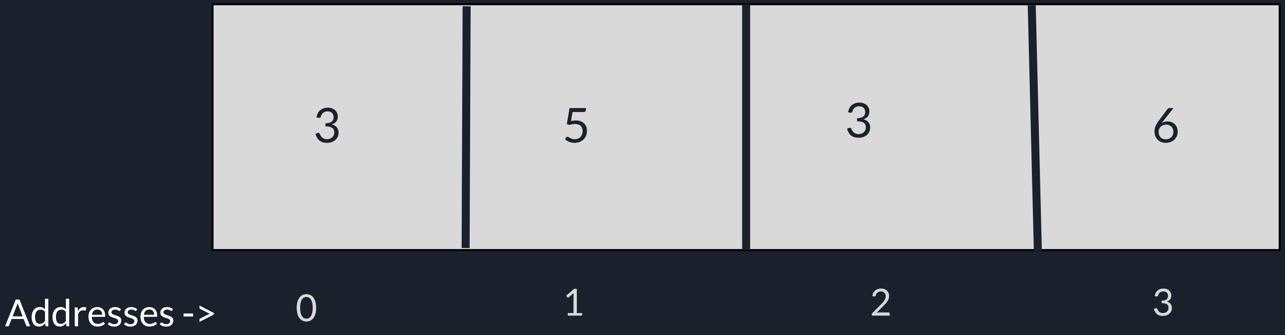
- What about moving things between memory?
- mov a, [0]



The basics - Registers

- Ok now let's change the value...

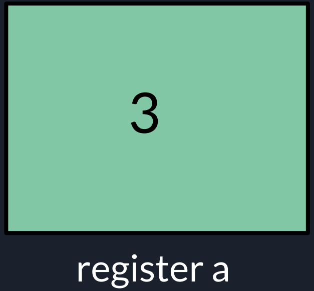


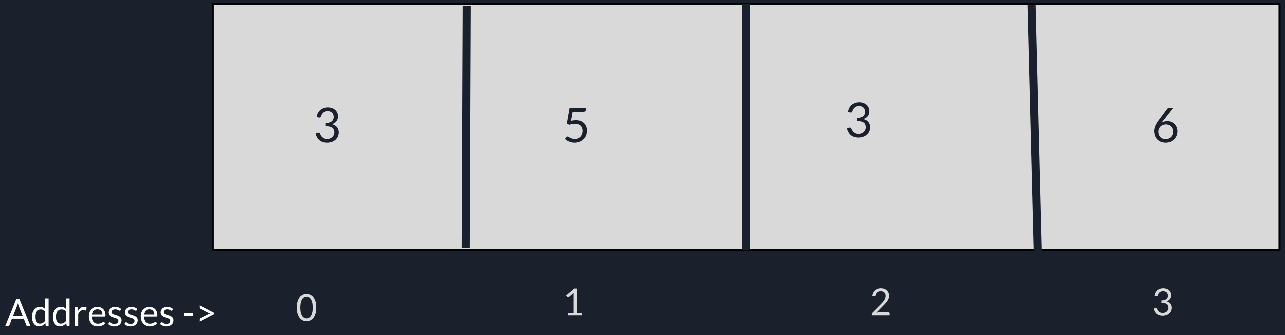


The basics - Registers

Remember its:
add dst, src

- Ok now let's change the value…
- add a, 10



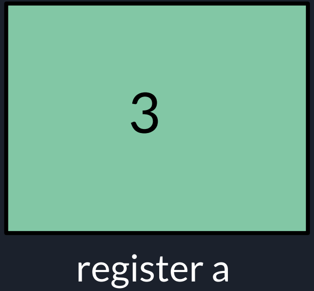


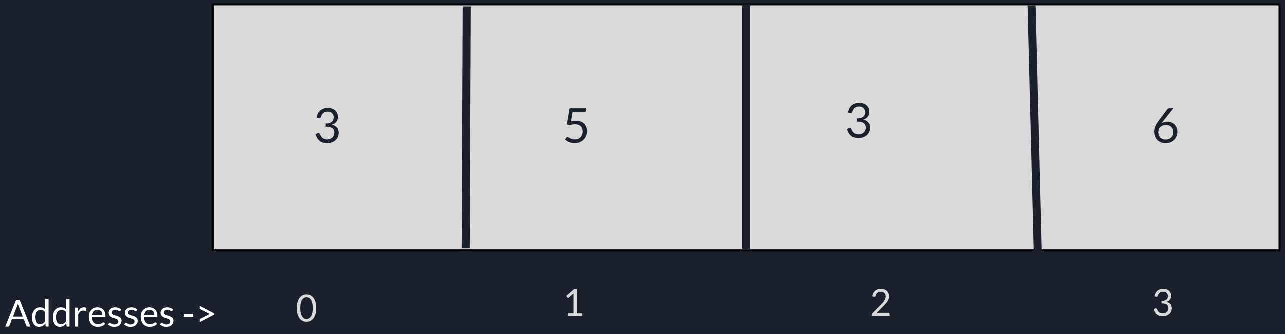
The basics - Registers

Remember its:
add dst, src

- Ok now let's change the value…
- add a, 10

Add 10 and the value in a

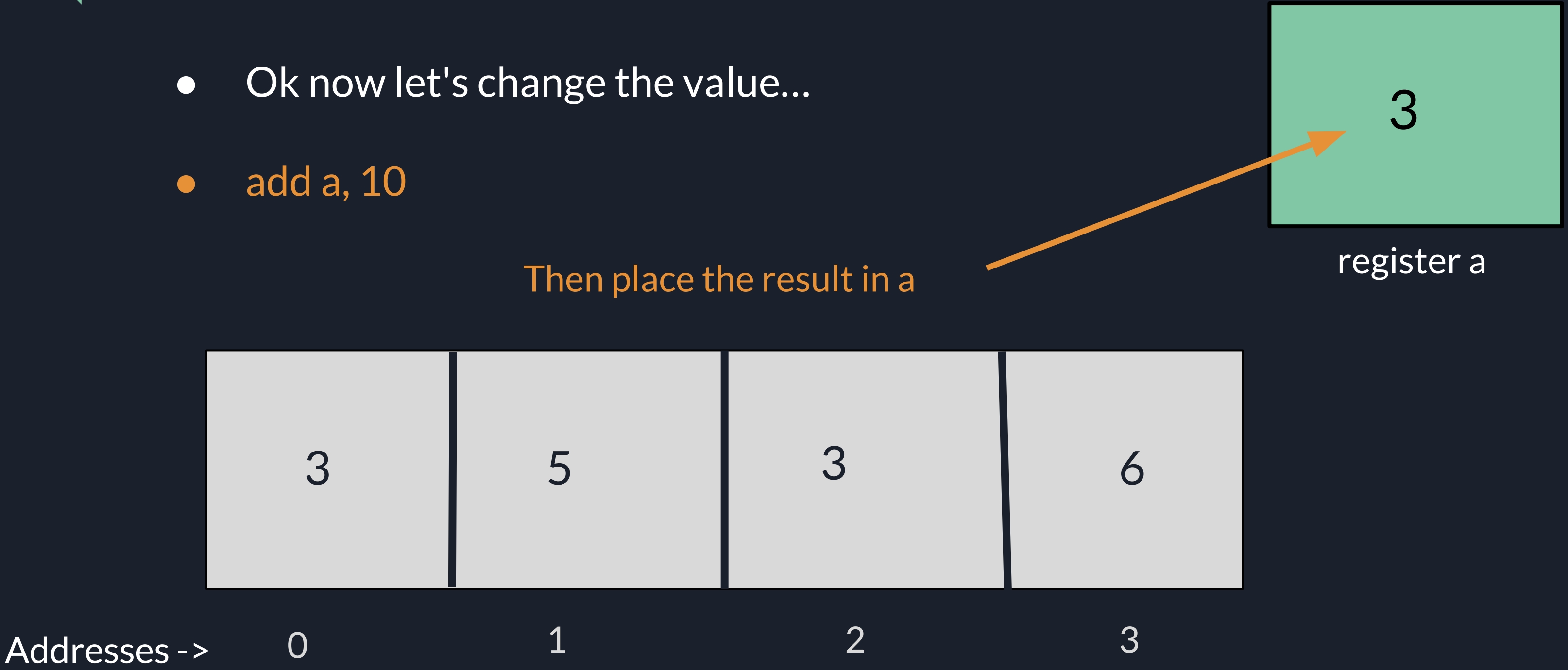




The basics - Registers

Remember its:
add dst, src

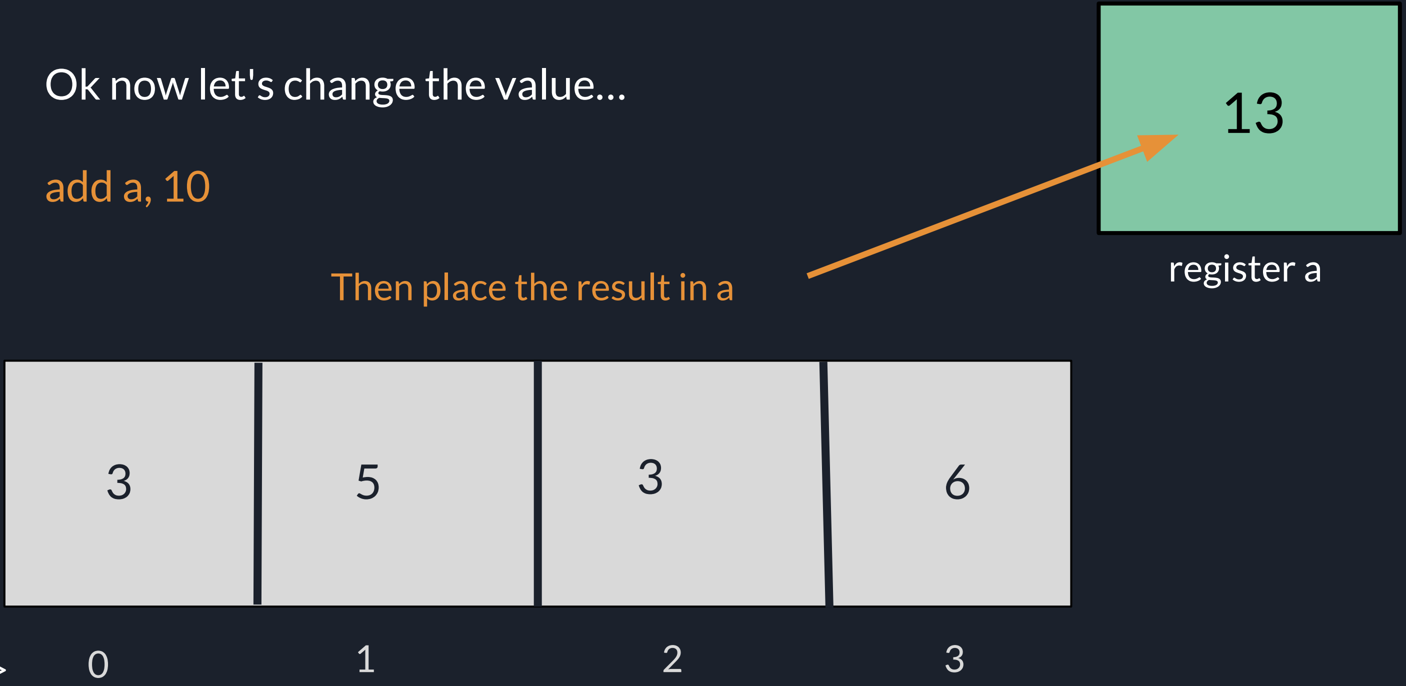
- Ok now let's change the value…
- add a, 10



The basics - Registers

Remember its:
add dst, src

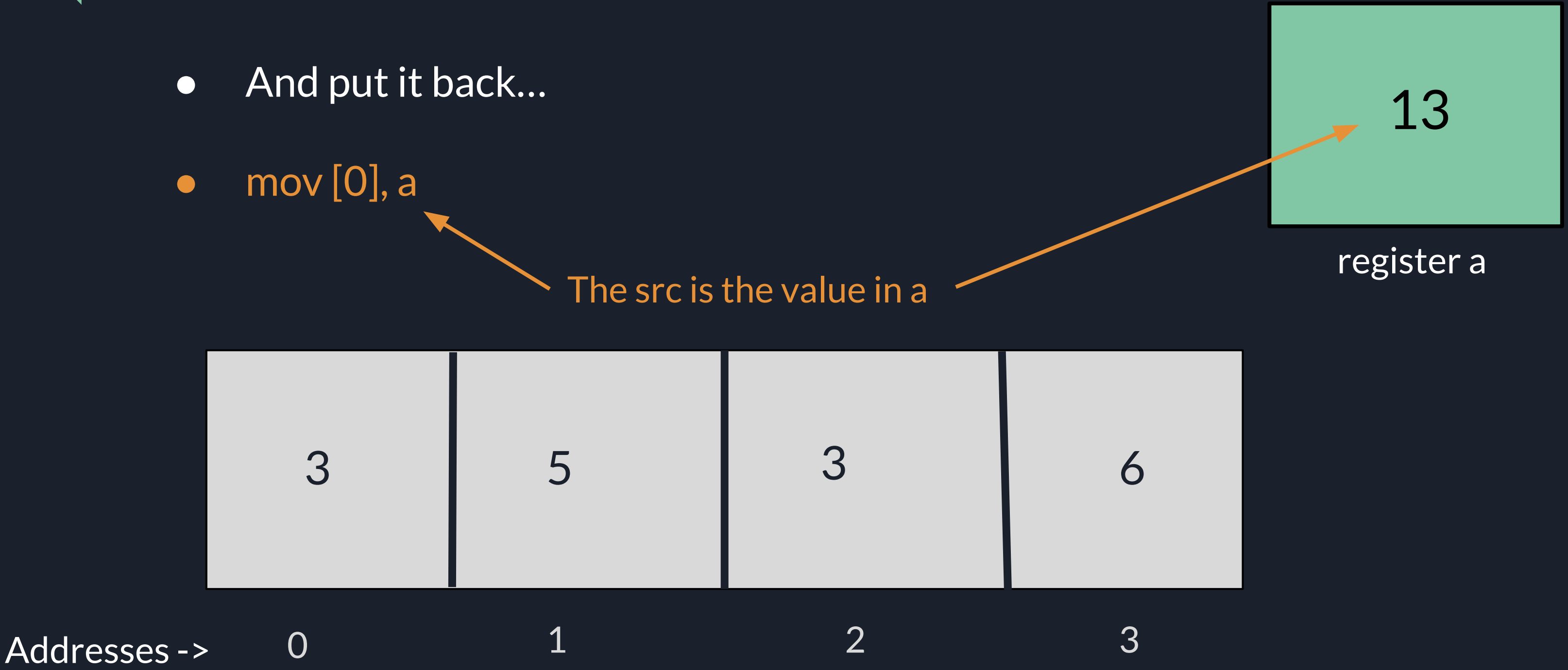
- Ok now let's change the value…
- add a, 10



The basics - Registers

Remember its:
mov dst, src

- And put it back…
- mov [0], a

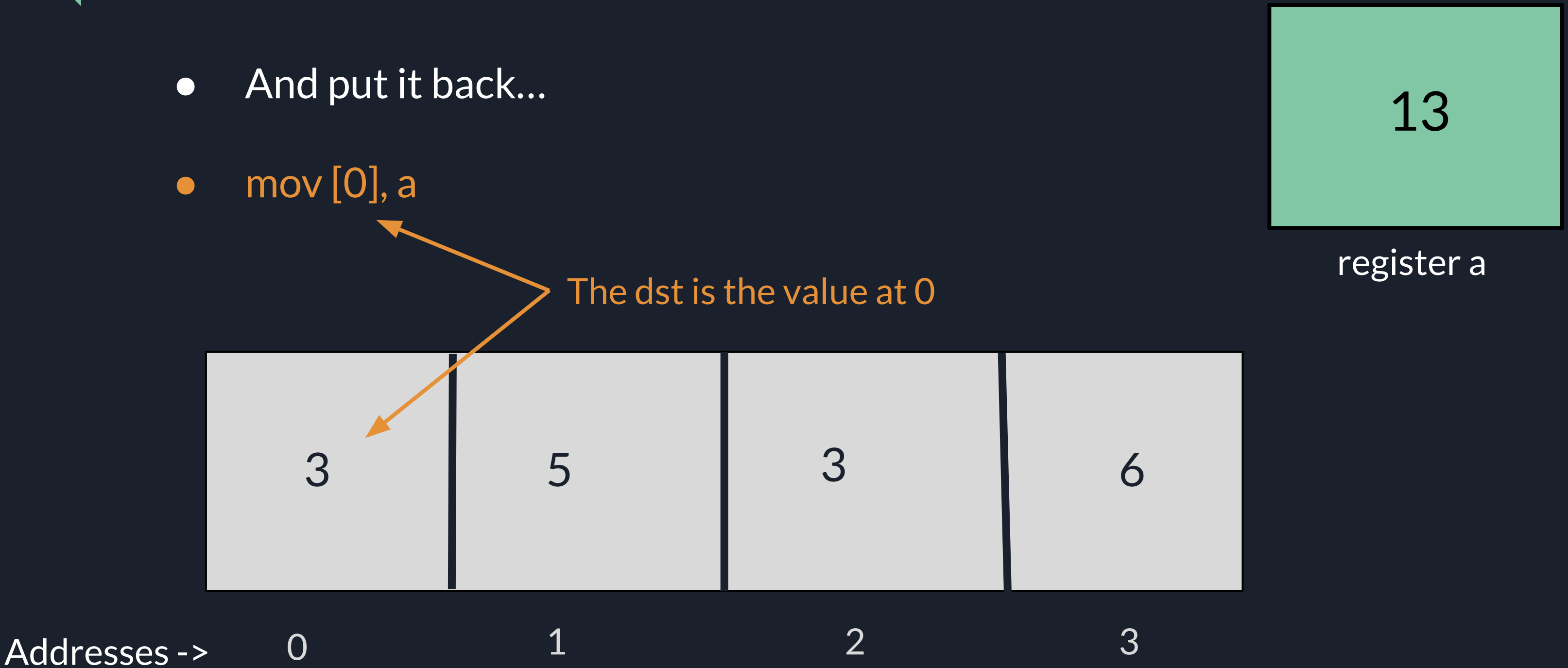


The basics - Registers

Remember its:
mov dst, src

- And put it back…
- mov [0], a

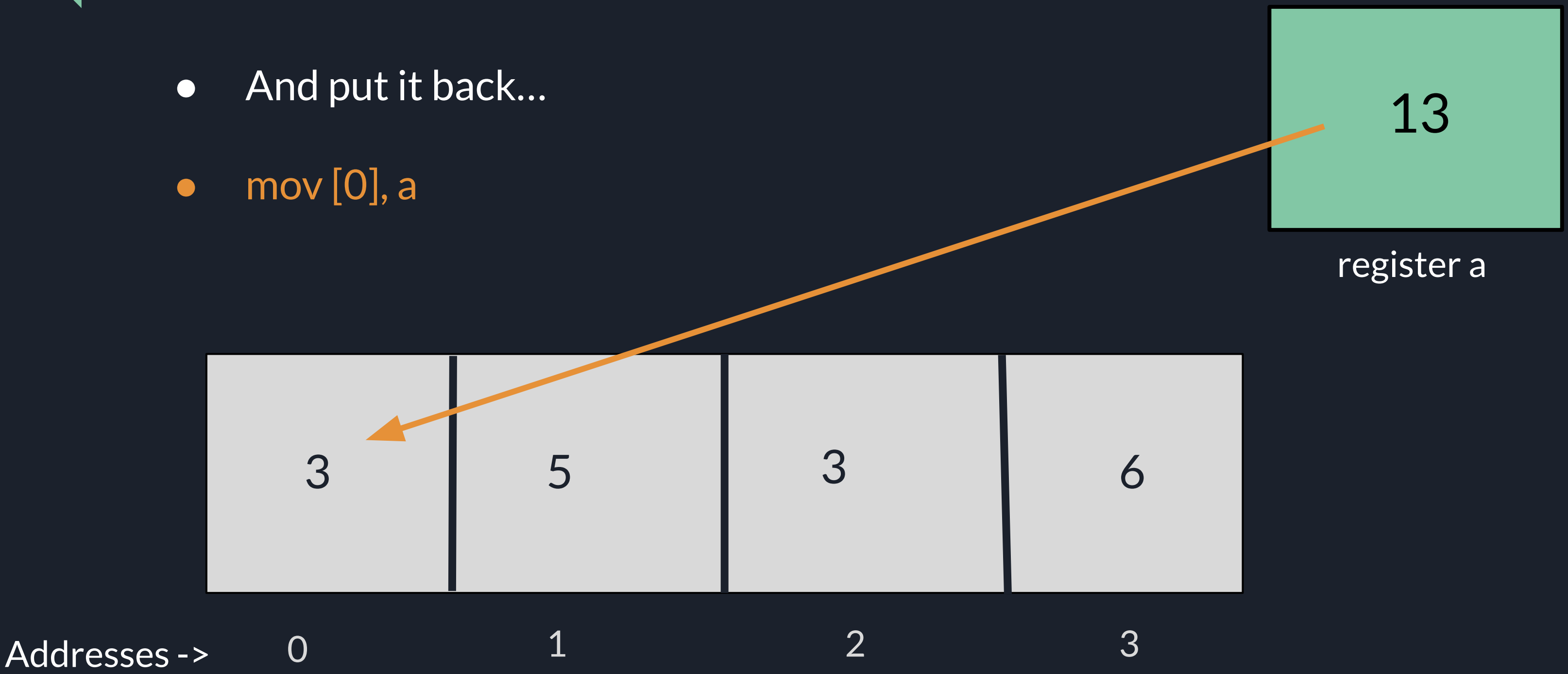
The dst is the value at 0



The basics - Registers

Remember its:
mov dst, src

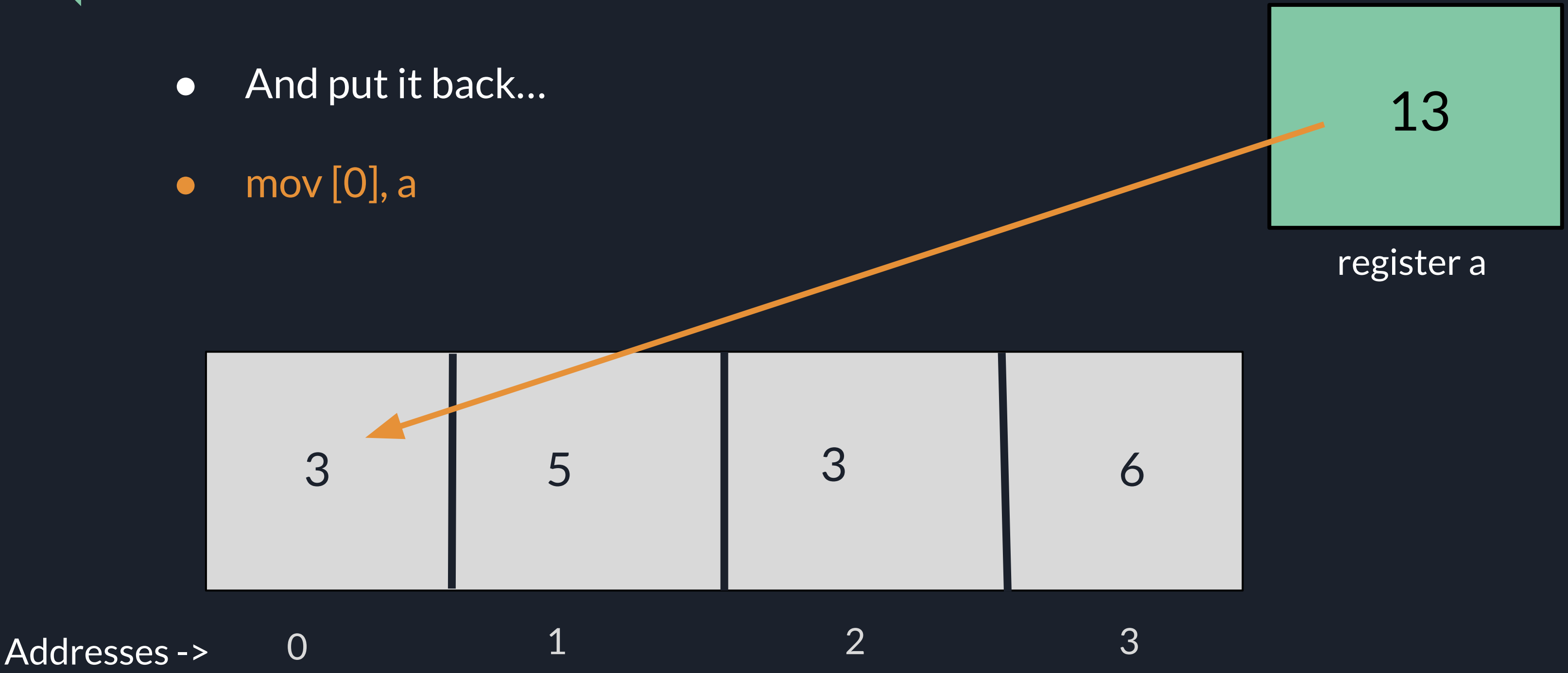
- And put it back…
- mov [0], a



The basics - Registers

Remember its:
mov dst, src

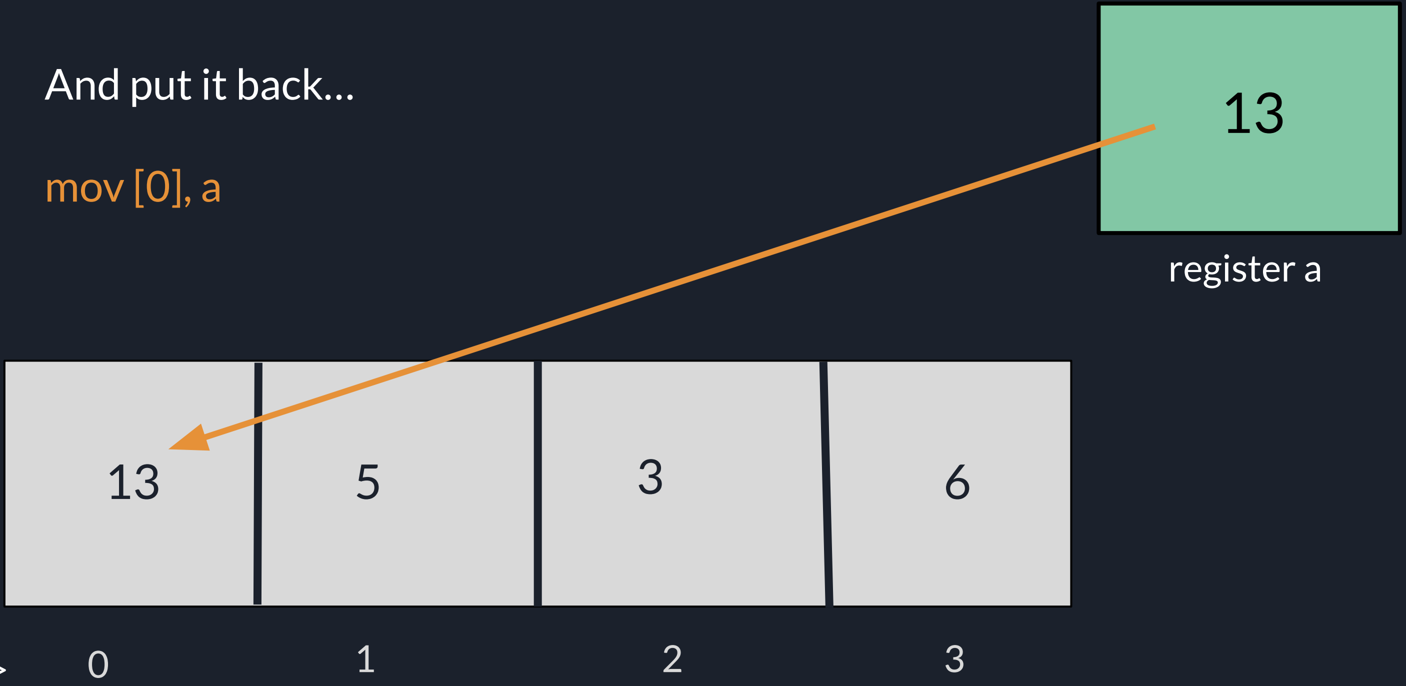
- And put it back…
- mov [0], a



The basics - Registers

Remember its:
mov dst, src

- And put it back…
- mov [0], a



That was a lot of steps!

I promise it gets easier…

The basics

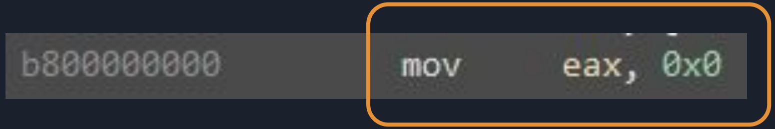
Instructions

Instructions

- Instructions are sometimes called “machine code” or “bytecode”
-

Instructions

- Instructions are sometimes called "machine code" or "bytecode"
- The assembly that we can read (like mov a, 10) gets translated to bytecode

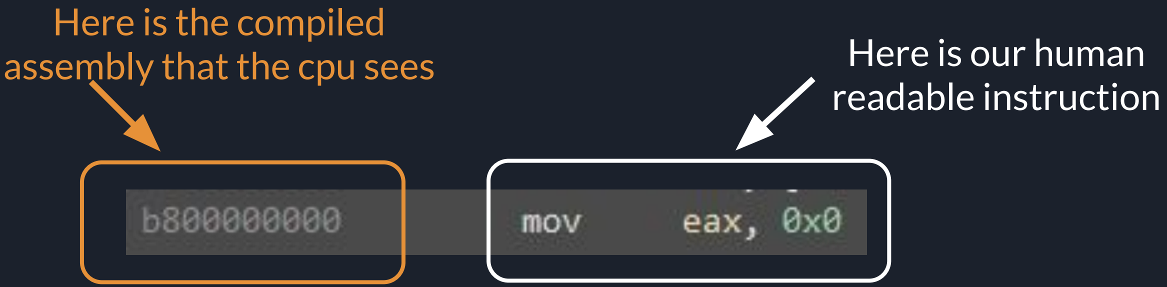


```
b800000000        mov     eax, 0x0
```

Here is our human readable instruction

Instructions

- Instructions are sometimes called “machine code” or “bytecode”
- The assembly that we can read (like mov a, 10) gets translated to bytecode


Here is the compiled assembly that the cpu sees
Here is our human readable instruction

```
b800000000        mov     eax, 0x0
```

Instructions - Some you know...

- mov dst, src - Moves memory from src to dst

Instructions - Some you know...

- mov dst, src - Moves memory from src to dst
- add dst, src - Adds dst and src, stores result in dst
- [address] - Dereference (says the value in brackets is an address)
- jmp address - Jumps from one location in the code to another

Instructions - Some you know...

- mov dst, src - Moves memory from src to dst
- add dst, src - Adds dst and src, stores result in dst
- jmp address - Jumps from one location in the code to another

Instructions - Some you know...

- mov dst, src - Moves memory from src to dst
- add dst, src - Adds dst and src, stores result in dst
- jmp address - Jumps from one location in the code to another
- call address - Fancy jump (more on this later)

Instructions - Some you know...

- mov dst, src - Moves memory from src to dst
- add dst, src - Adds dst and src, stores result in dst
- jmp address - Jumps from one location in the code to another
- call address - Fancy jump (more on this later)
- push value - Pushes the value to the top of the stack

Instructions - Some you know...

- mov dst, src - Moves memory from src to dst
- add dst, src - Adds dst and src, stores result in dst
- jmp address - Jumps from one location in the code to another
- call address - Fancy jump (more on this later)
- push value - Pushes the value to the top of the stack
- cmp value1, value2 - Sets flags in EFLAGS

Instructions - Some you know...

- mov dst, src - Moves memory from src to dst
- add dst, src - Adds dst and src, stores result in dst
- jmp address - Jumps from one location in the code to another
- call address - Fancy jump (more on this later)
- push value - Pushes the value to the top of the stack
- cmp value1, value2 - Sets flags in EFLAGS
- je, jne, jz, jnz - Conditional jumps (look them up there are a lot)

Instructions - Some you know...

- Any time you see an instruction you don’t know...

Instructions - Some you know...

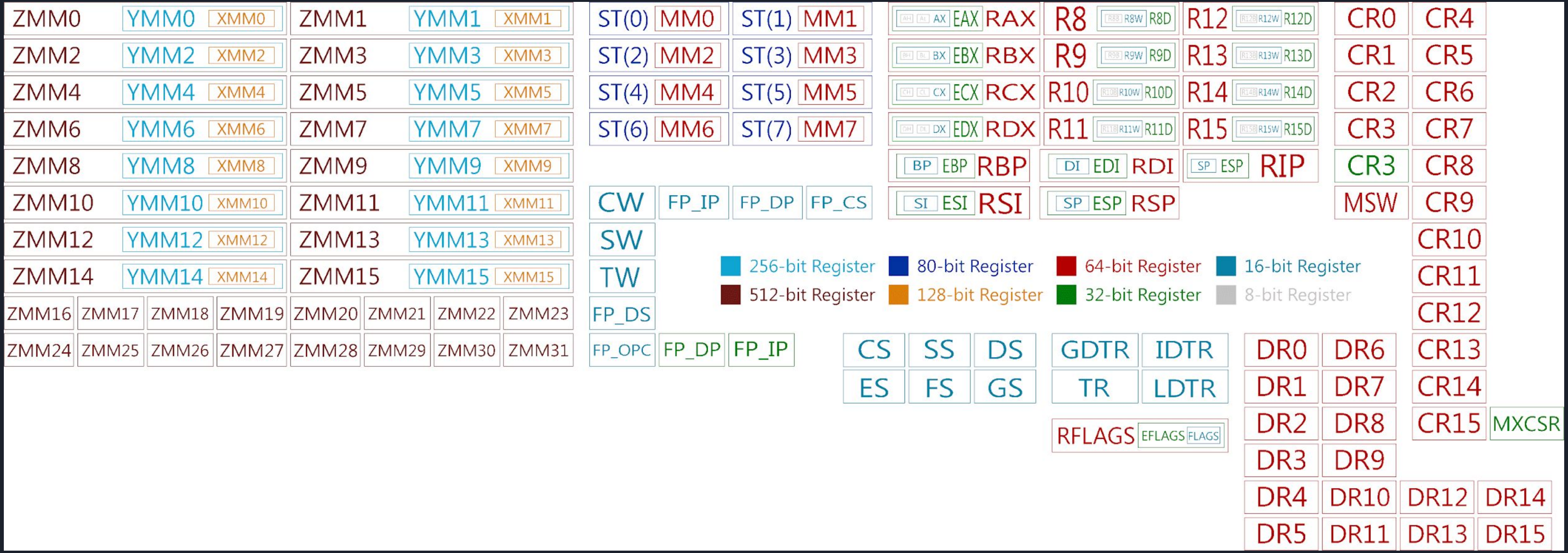
- Any time you see an instruction you don't know...
  - Look it up!

Instructions - Some you know...

- Any time you see an instruction you don't know...
  - Look it up!
  - There are decades of forum questions on any and all assembly problems

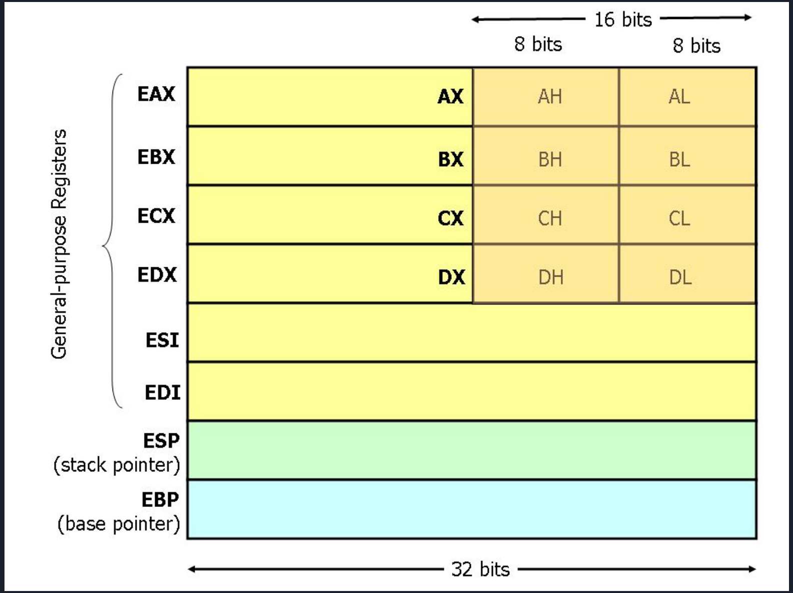
Again but advanced…

Registers (there's a lot of them)



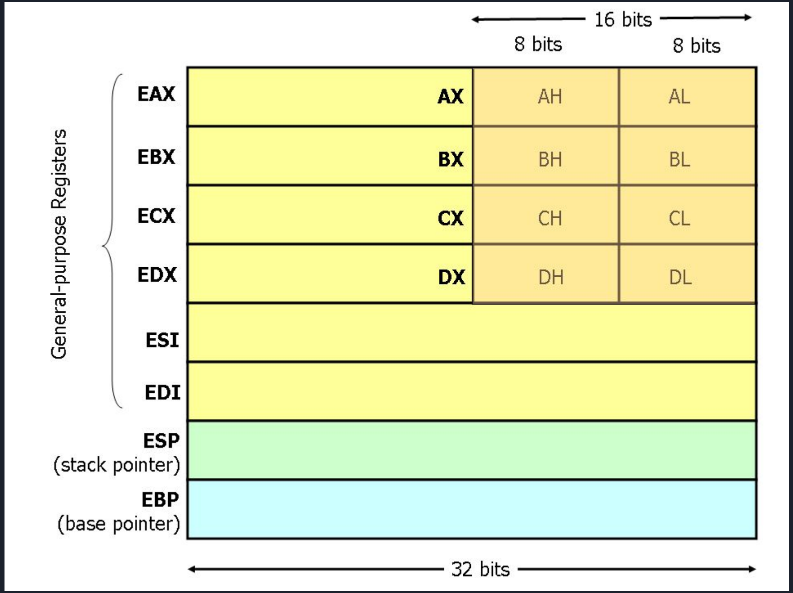
Registers

- Here are the ones we care about…



Registers

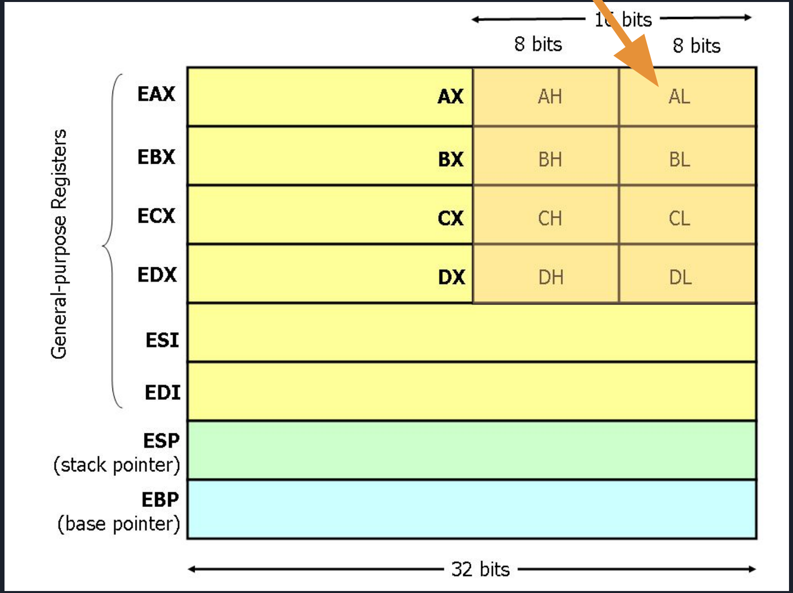
- Here are the ones we care about…
- You can refer to different sections of the same space...



Registers

- Here are the ones we care about…
- You can refer to different sections of the same space...

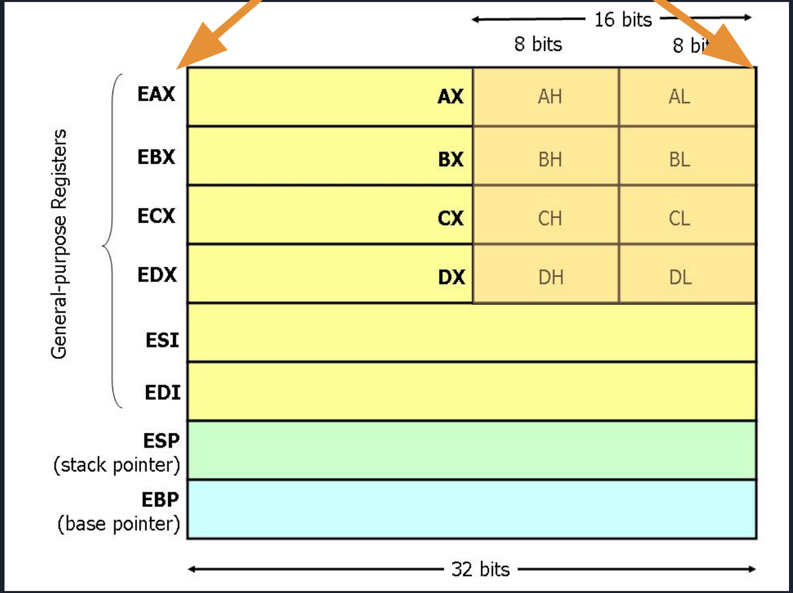
al refers to the lowest byte of the eax register



Registers

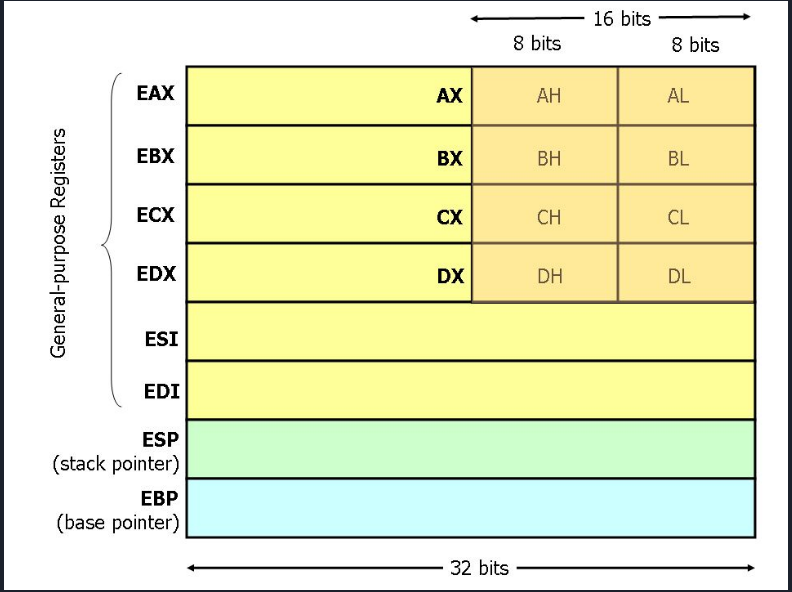
- Here are the ones we care about…
- You can refer to different sections of the same space...

eax refers to the full 4 bytes



Registers

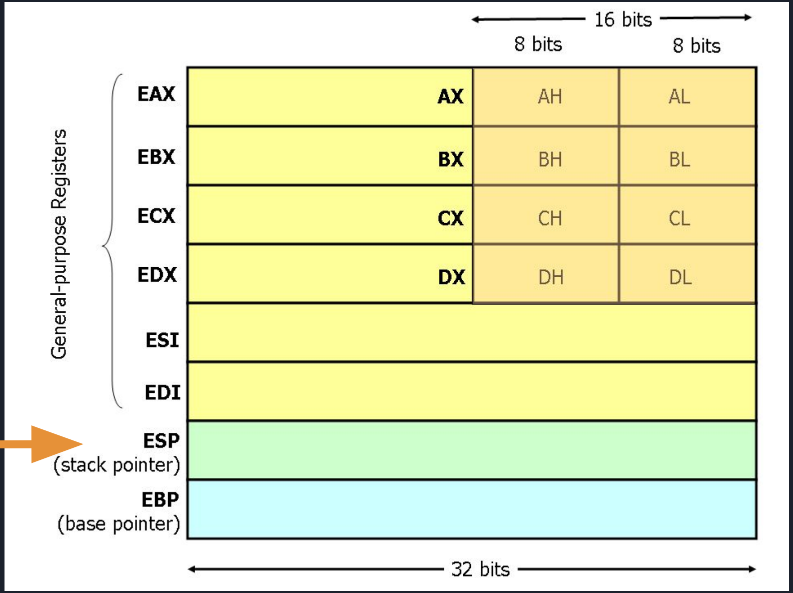
- Here are the ones we care about…
- You can refer to different sections of the same space…
- There are some registers we can use for anything, some are reserved



Registers

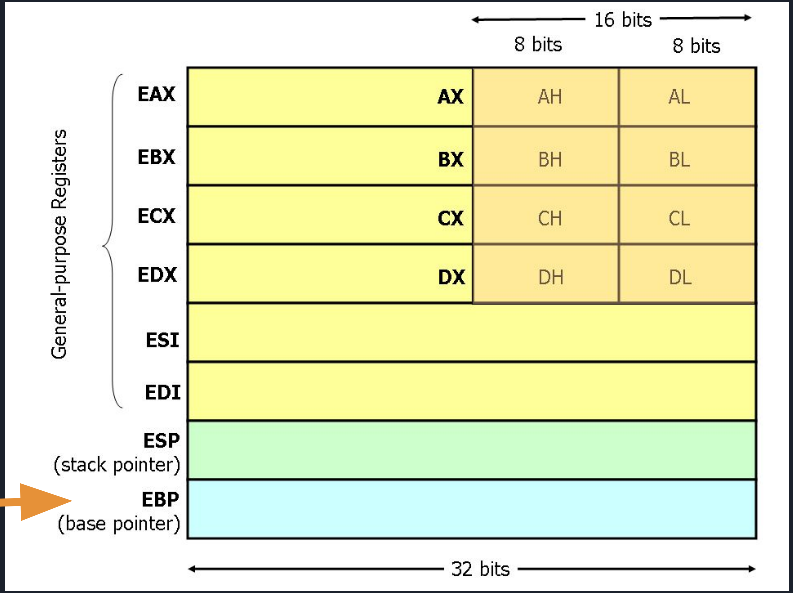
- Here are the ones we care about…
- You can refer to different sections of the same space…
- There are some registers we can use for anything, some are reserved

esp is reserved for the stack pointer



Registers

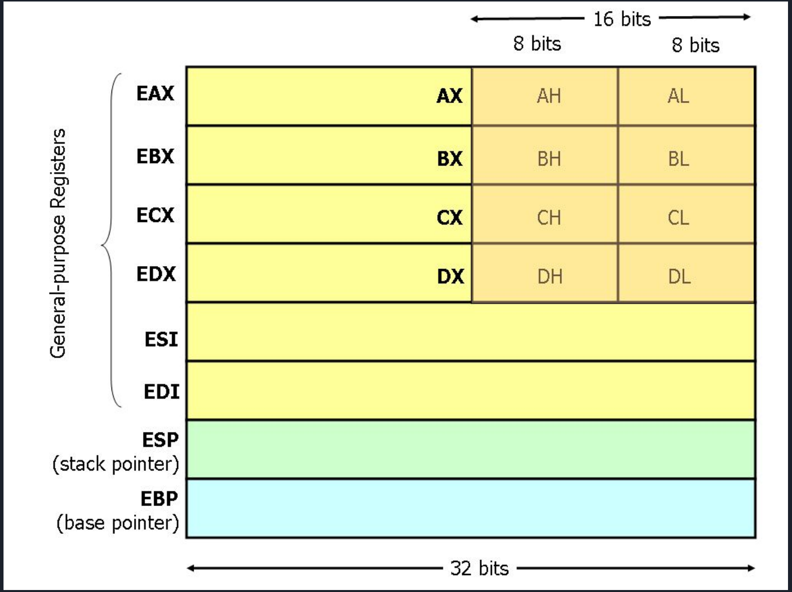
- Here are the ones we care about…
- You can refer to different sections of the same space…
- There are some registers we can use for anything, some are reserved



ebp is reserved for the stack base pointer

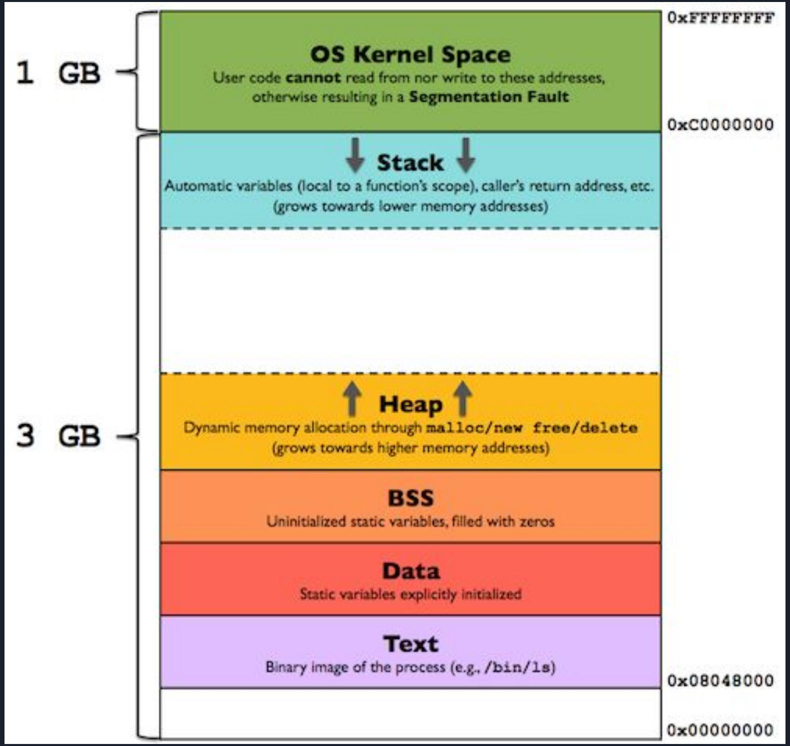
Registers

- Here are the ones we care about…
- You can refer to different sections of the same space…
- There are some registers we can use for anything, some are reserved
- More on that later...



Memory Layout

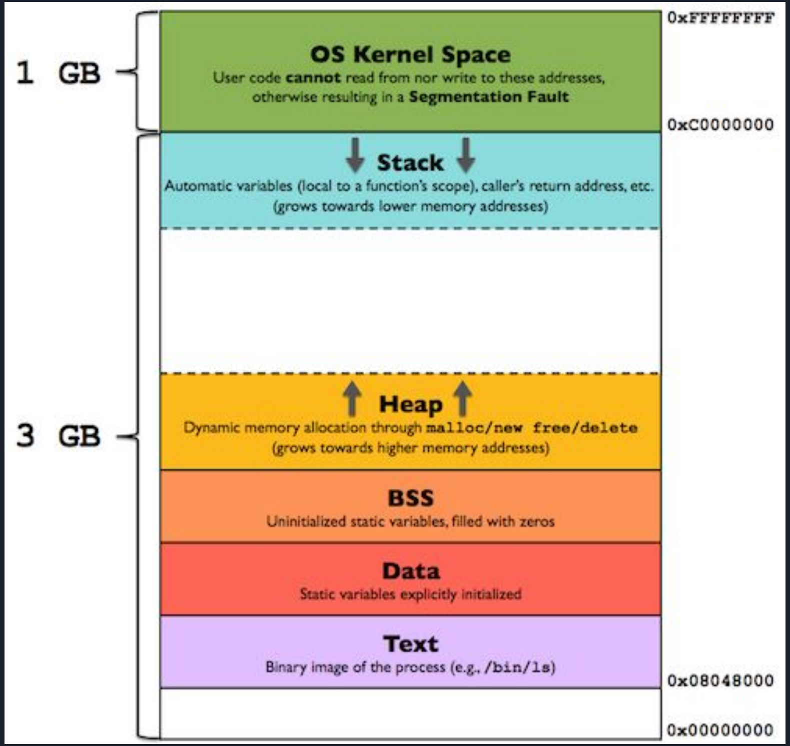
Some will draw the stack growing up, some growing down.


0xFFFFFFFF
1 GB
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

Memory Layout

Some will draw the stack growing up, some growing down.

It doesn't matter which model you adopt, they are exactly the same!



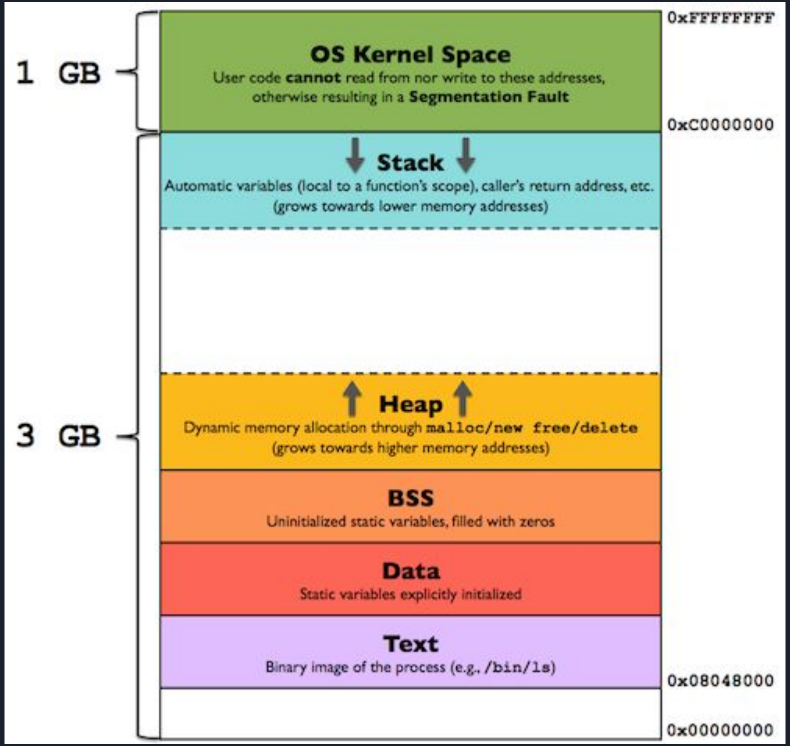
1 GB
3 GB
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)

Memory Layout

Some will draw the stack growing up, some growing down.

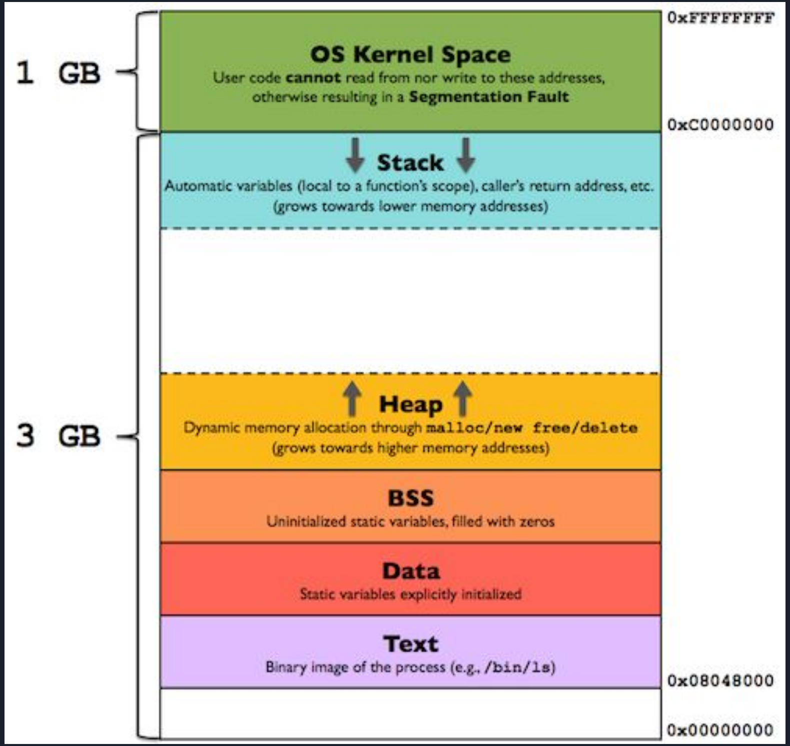
It doesn't matter which model you adopt, they are exactly the same!

Programs have "sections" of memory


0xFFFFFFFF
1 GB
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

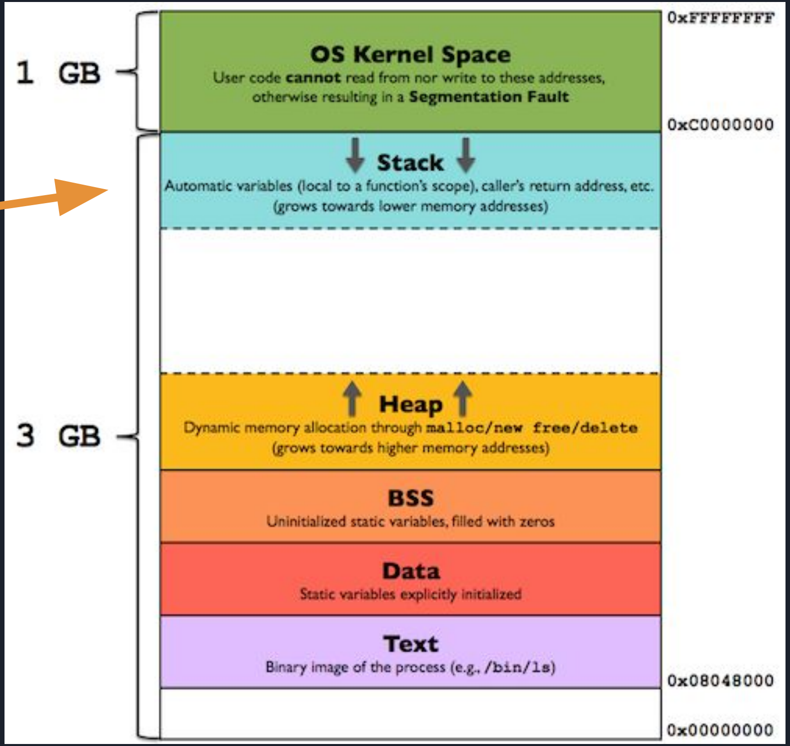
Memory Layout

Something important:
All of this is just data! We as programmers give purpose to this data in the way we treat it!


1 GB
3 GB
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)

Memory Layout

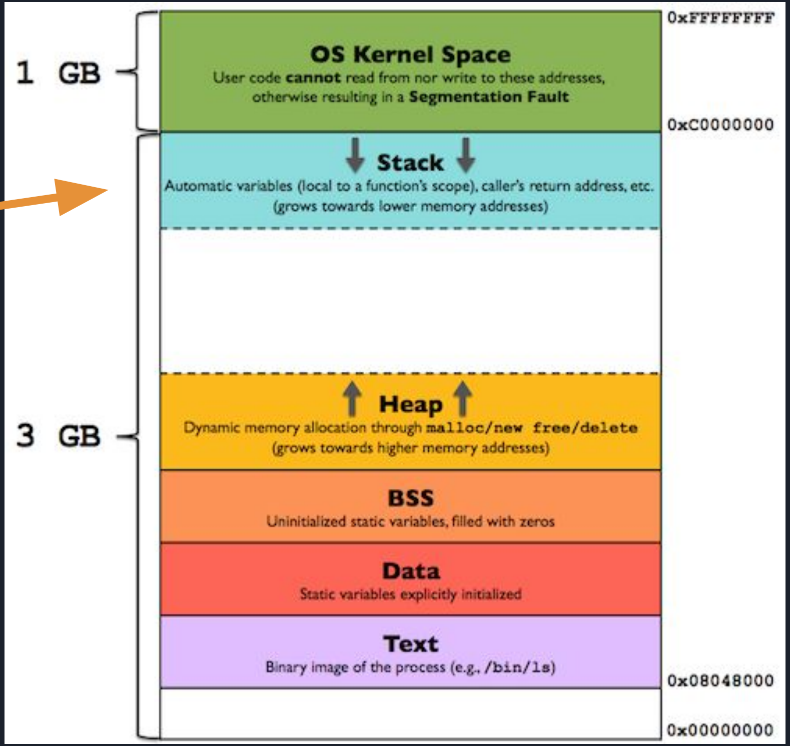
Stack is where the local variables for functions live



0xFFFFFFFF
1 GB
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

Memory Layout

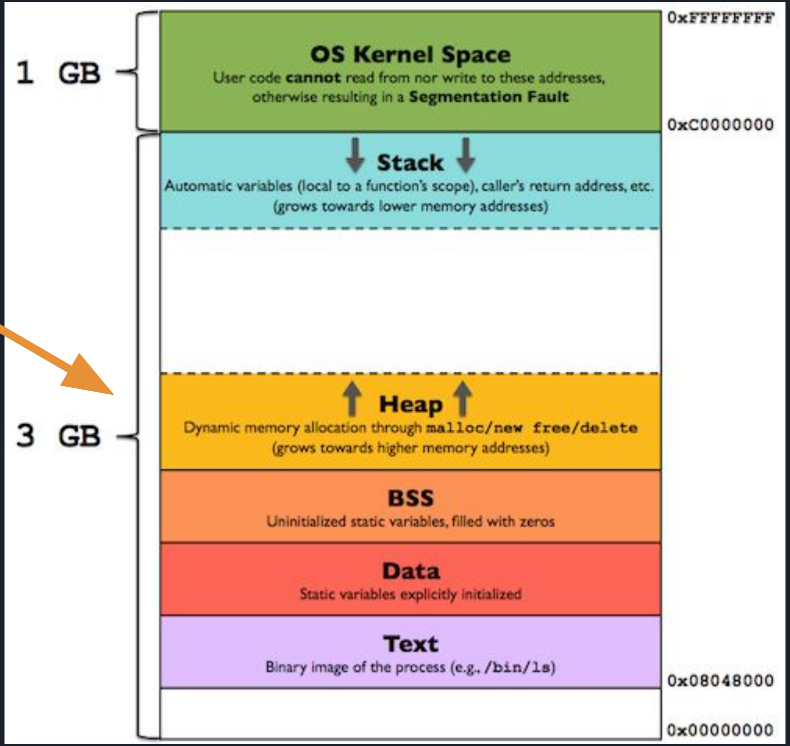
As you call functions, the stack grows down



0xFFFFFFFF
1 GB
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

Memory Layout

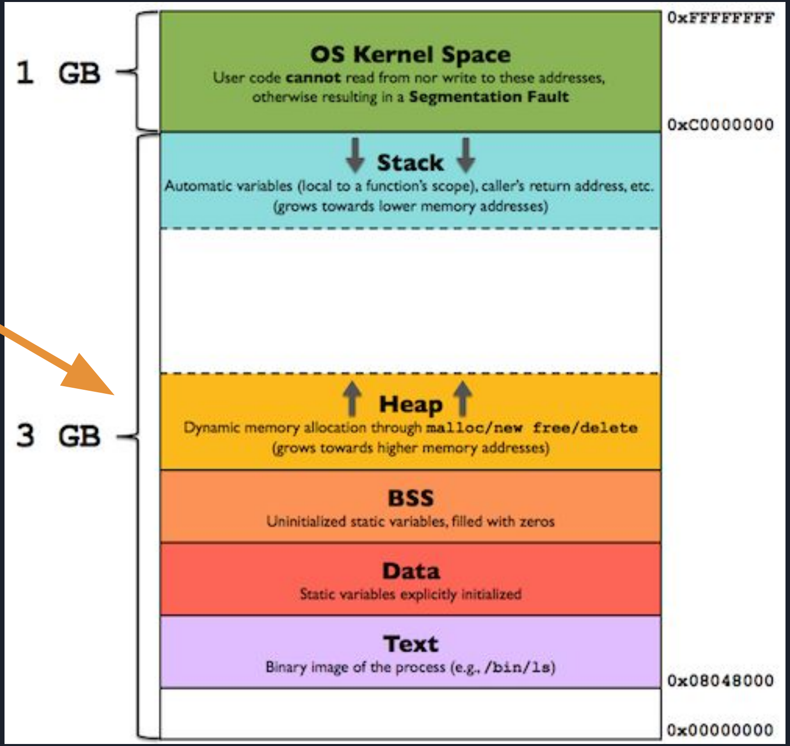
The heap is where dynamically allocated data lives



1 GB
3 GB
0xFFFFFFFF
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

Memory Layout

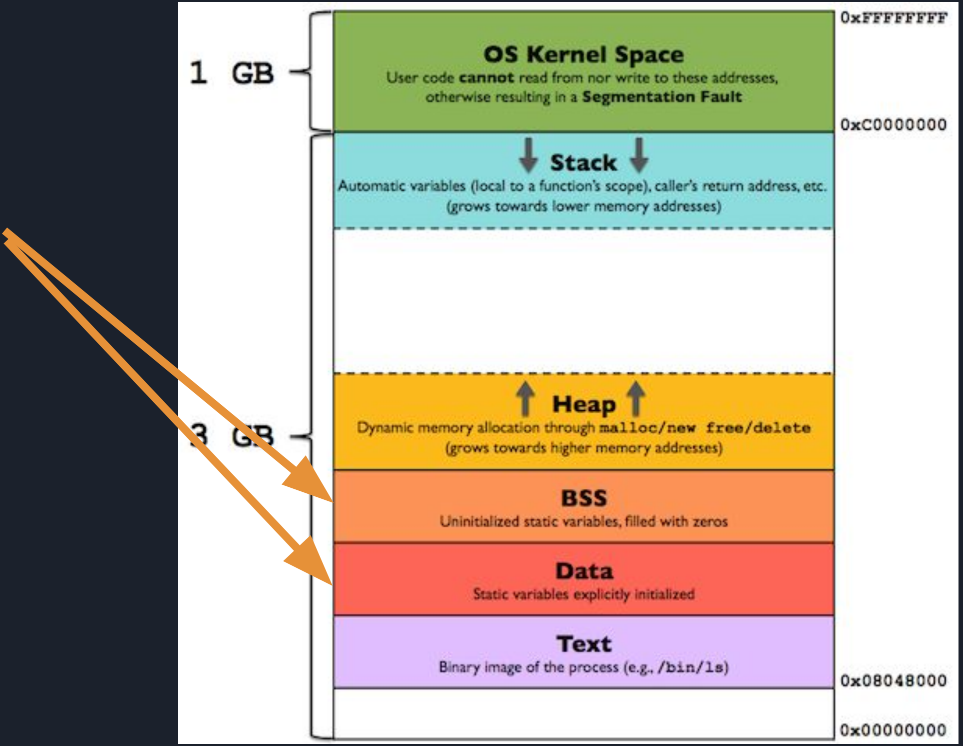
As you allocate more data, the heap grows up



1 GB
3 GB
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)

Memory Layout

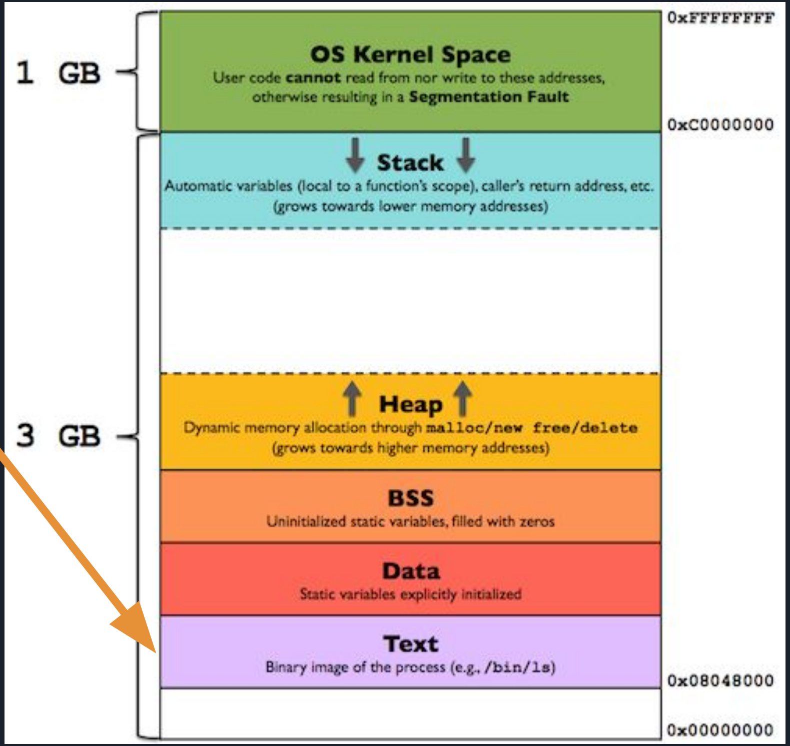
The bss and data sections are where global variables live



1 GB
3 GB
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)

Memory Layout

The “text” section holds the assembly code of the program



0xFFFFFFFF
1 GB
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

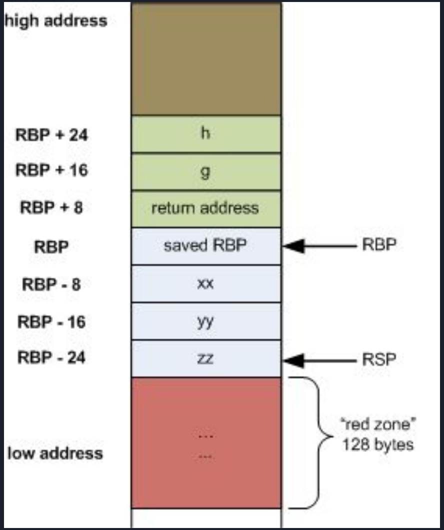
Ok now we have a problem...

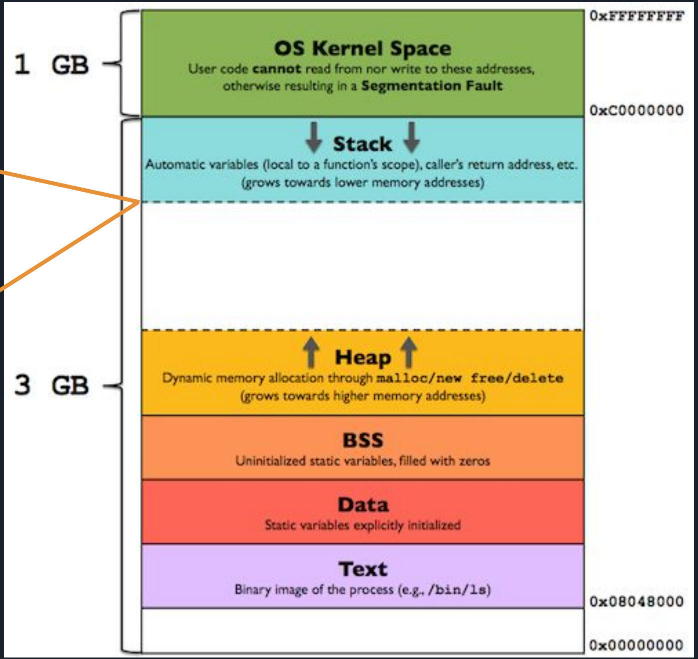
- How exactly do we organize memory?
- At this level there are basically no rules right?
- So how do we keep things straight?

The organization solution

The Stack Frame

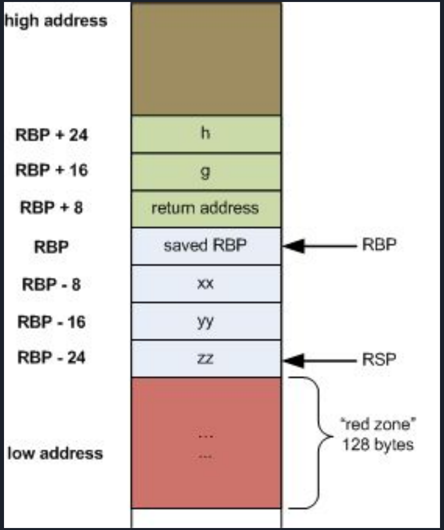
Stack Frame

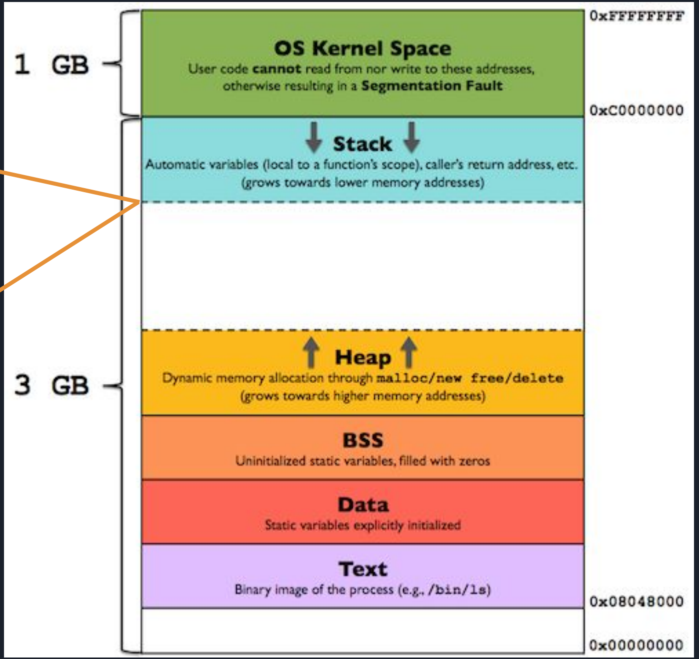

high address
RBP + 24
h
RBP + 16
g
RBP + 8
return address
RBP
saved RBP
RBP
RBP - 8
xx
RBP - 16
yy
RBP - 24
zz
RSP
low address
...
...
"red zone"
128 bytes


0xFFFFFFFF
1 GB
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc.
(grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete
(grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

Stack Frame

The stack is made up of many "frames" for each function call

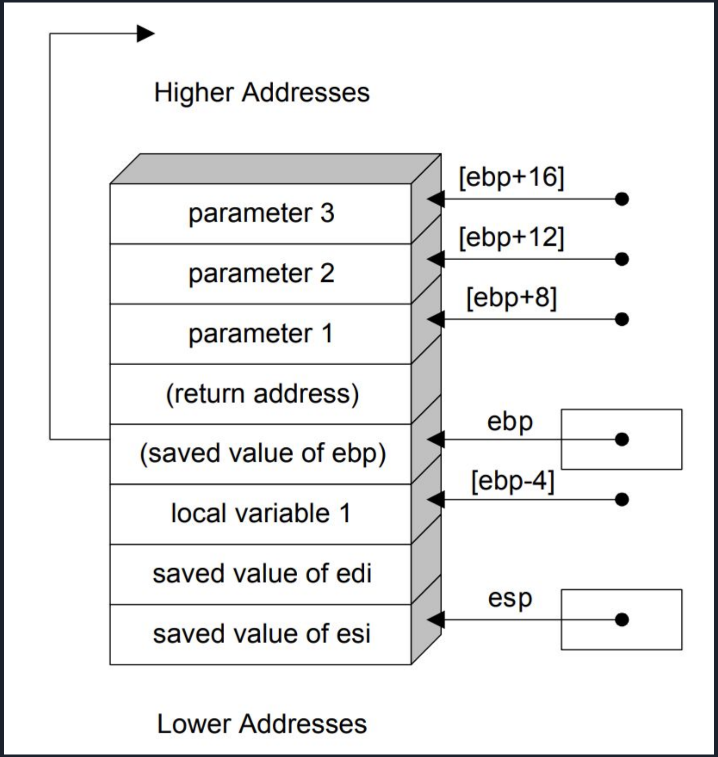




0xFFFFFFFF
1 GB
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

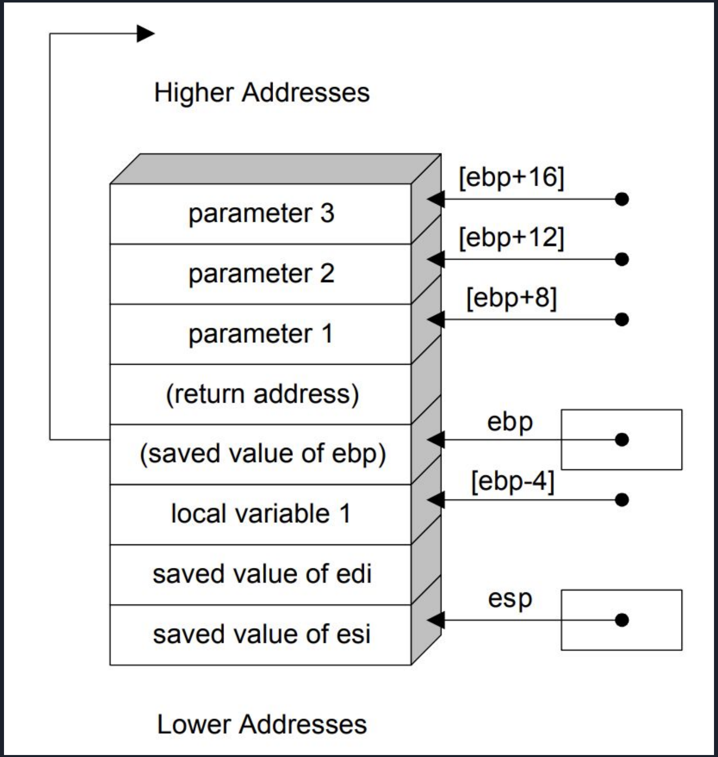
Stack Frame

- Let's talk instructions
  - call address - pushes the address of the next instruction then jmp



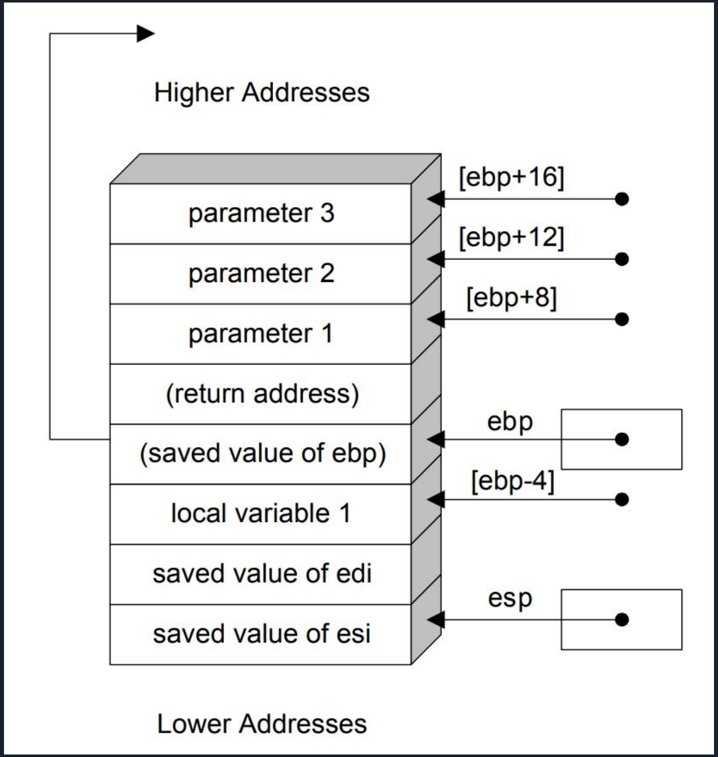
Stack Frame

- Let’s talk instructions
  - call address - pushes the address of the next instruction then jmp
  - ret - pop’s top value off the stack into eip



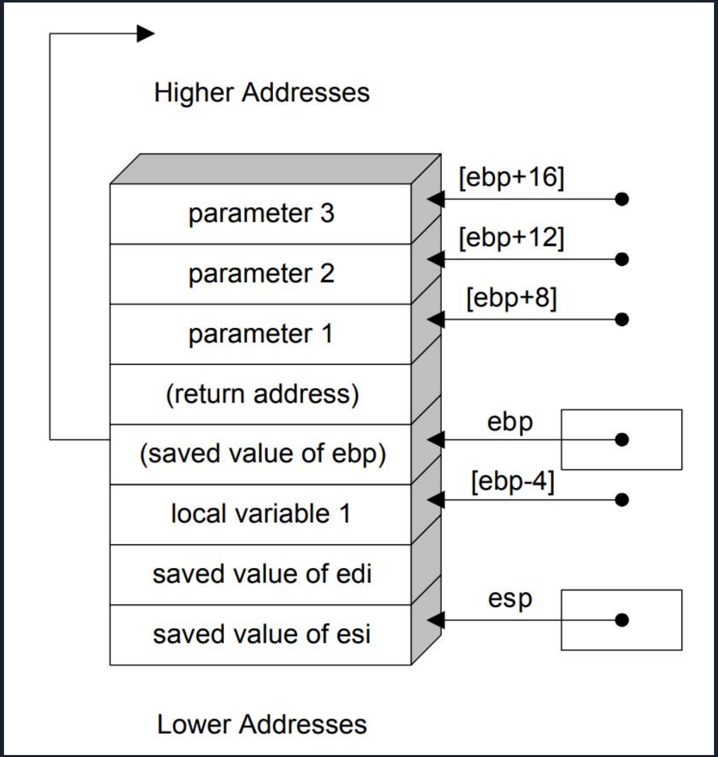
Stack Frame

- Let’s talk instructions
  - call address - pushes the address of the next instruction then jmp
  - ret - pop’s top value off the stack into eip
- When a new function is called, it will run a prolog and epilog
  - The prolog sets up the frame
  - The epilog cleans it up and ret’s



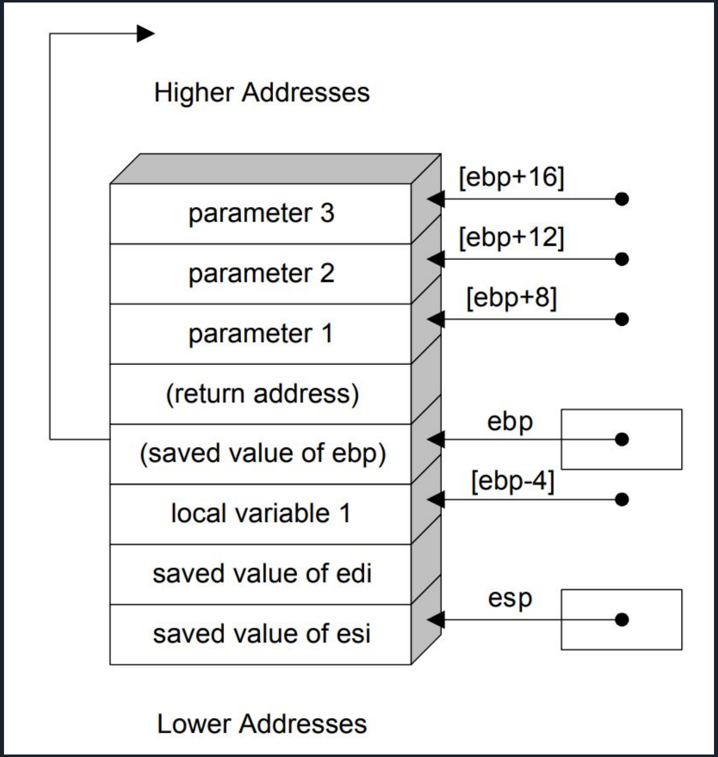
Stack Frame

- Let’s talk instructions
  - call address - pushes the address of the next instruction then jmp
  - ret - pop’s top value off the stack into eip
- When a new function is called, it will run a prolog and epilog
  - The prolog sets up the frame



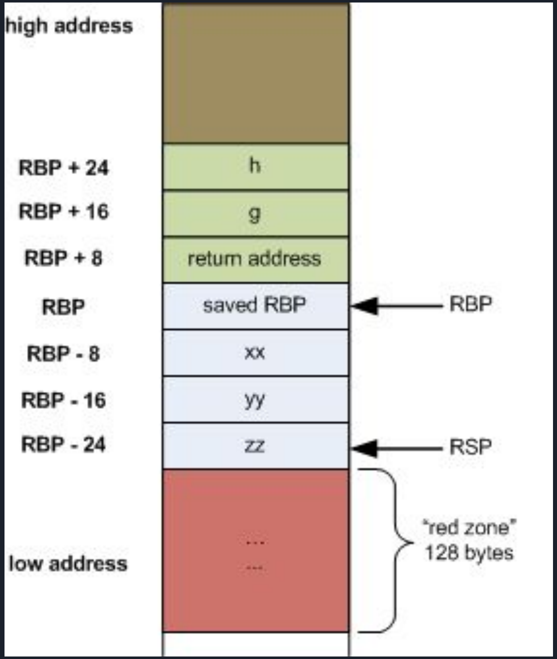
Stack Frame

- Let’s talk instructions
  - call address - pushes the address of the next instruction then jmp
  - ret - pop’s top value off the stack into eip
- When a new function is called, it will run a prolog and epilog
  - The prolog sets up the frame
  - The epilog cleans it up and ret’s



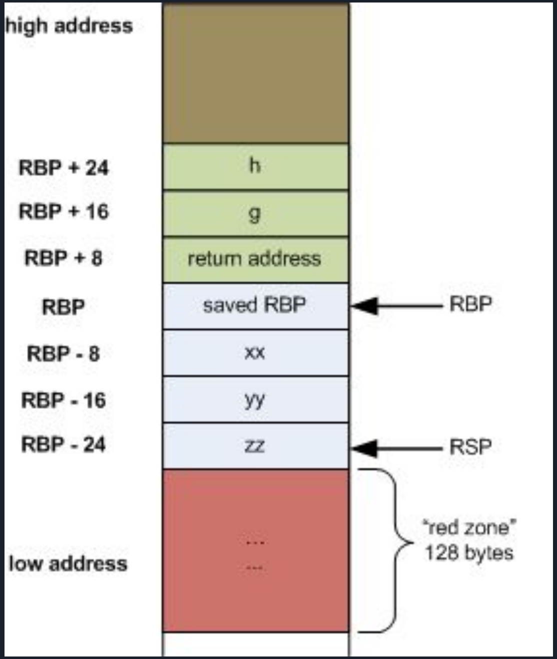
Stack Frame

- Every function will make a new stack frame, then clean it up before returning



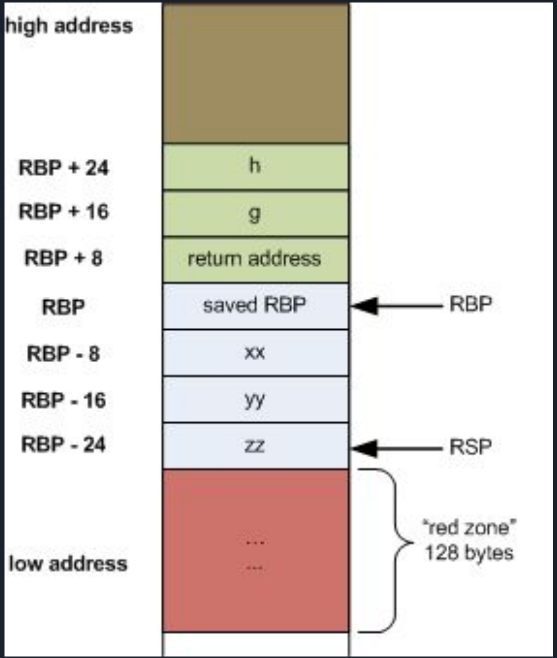
Stack Frame

- Every function will make a new stack frame, then clean it up before returning
- The stack frame is where local variables are stored



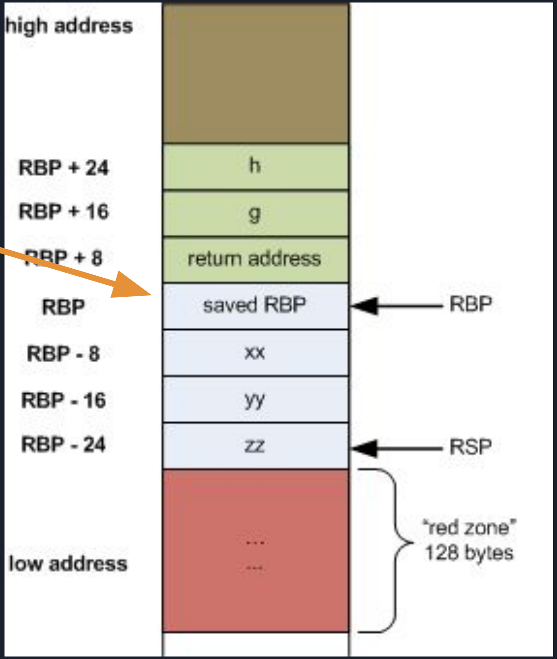
Stack Frame

Remember ebp and esp from before?



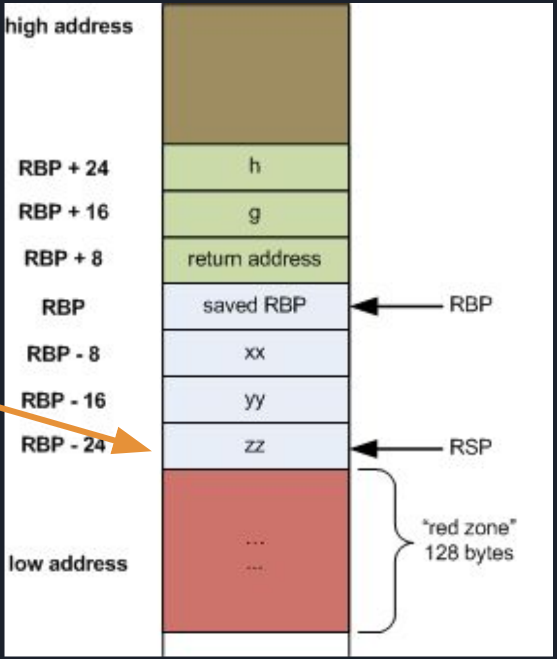
Stack Frame

ebp holds the address of the "base" of the stack frame



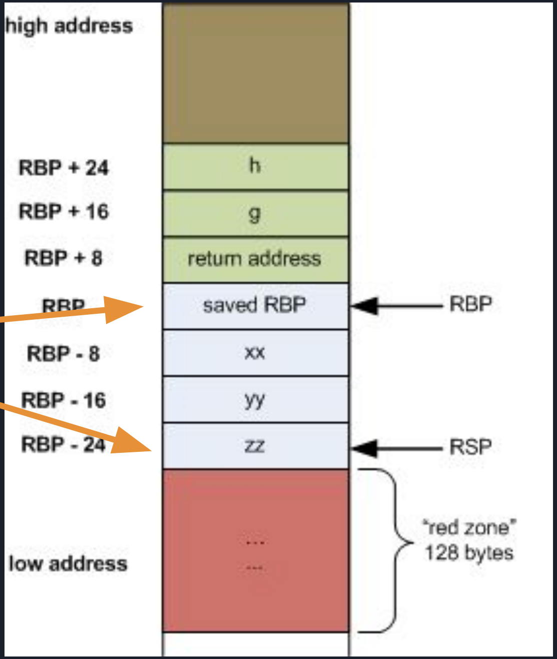
Stack Frame

esp holds the address of the bottom of the stack



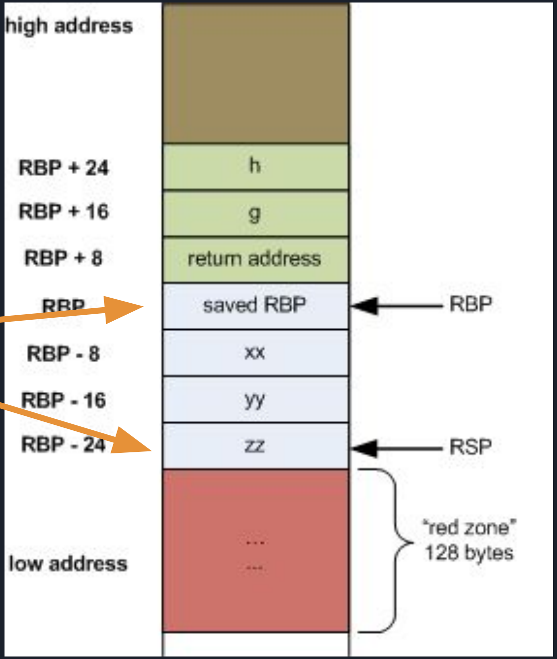
Stack Frame

Everything in between is space for the local variables!



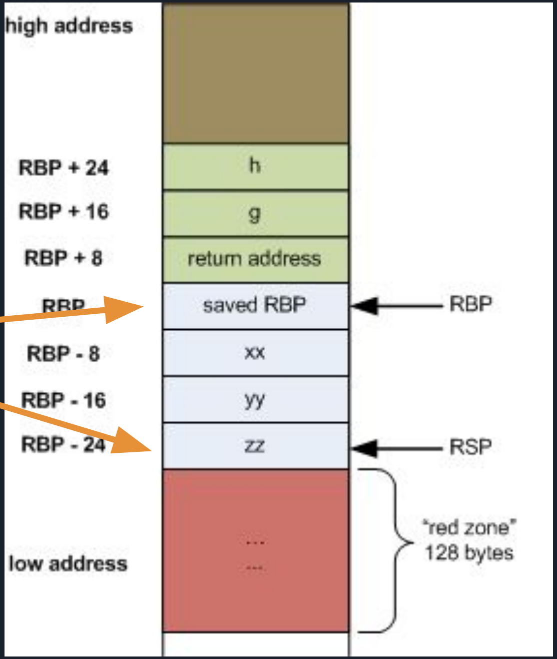
Stack Frame

Different functions will need different amounts of space for locals



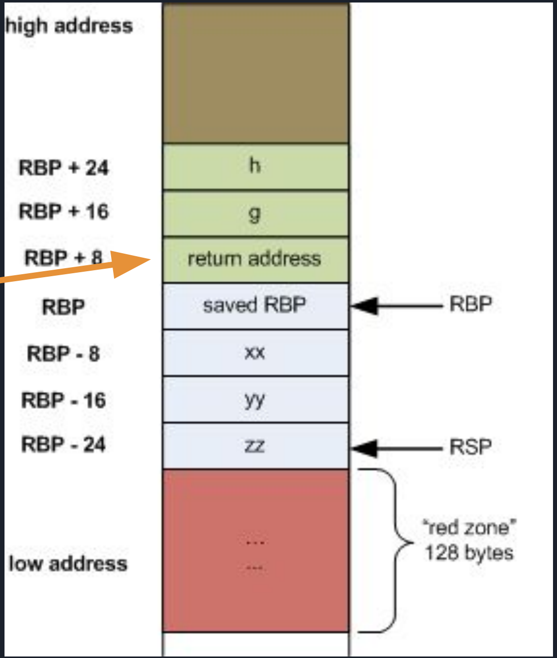
Stack Frame

It's up to each function to set up the space they need

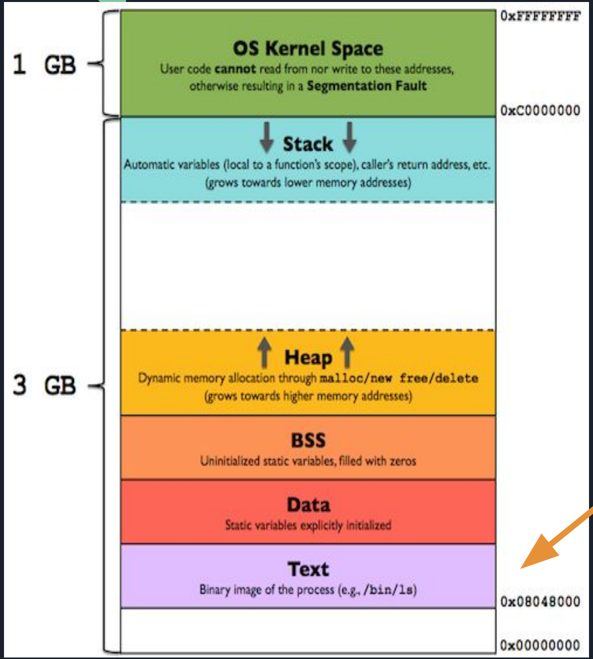


Stack Frame

To understand the return address, let's remember how the program layout looks

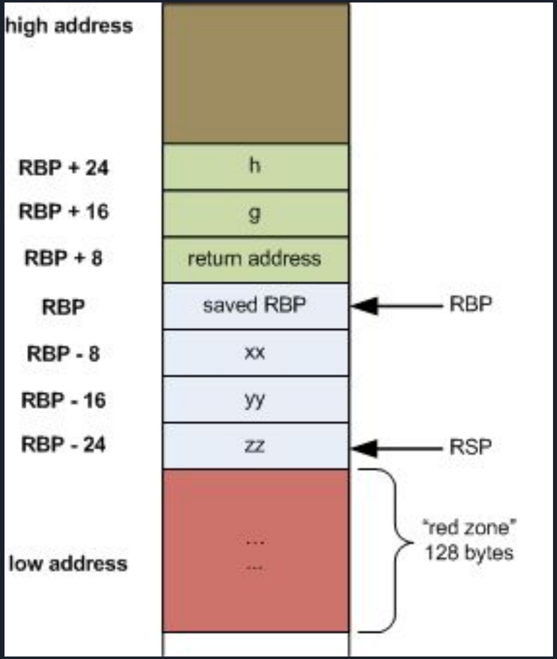


Stack Frame

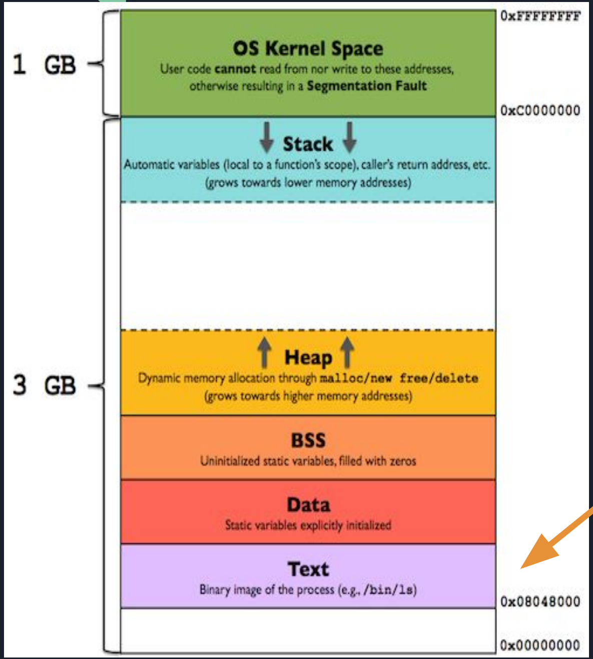


1 GB
3 GB
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000

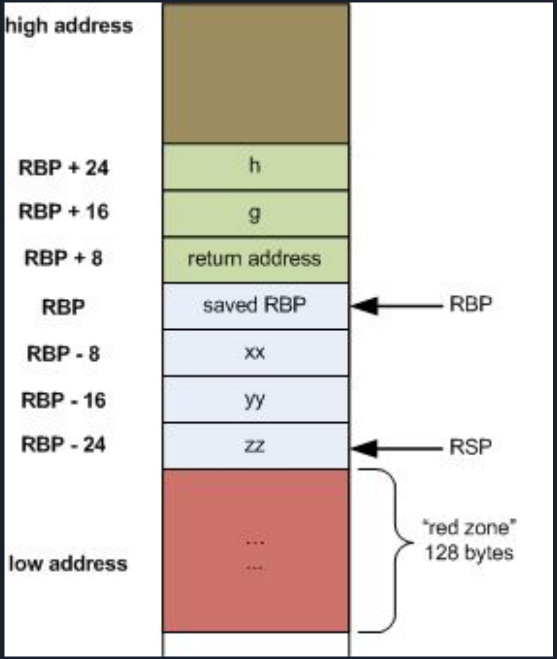
The compiled assembly lives in memory



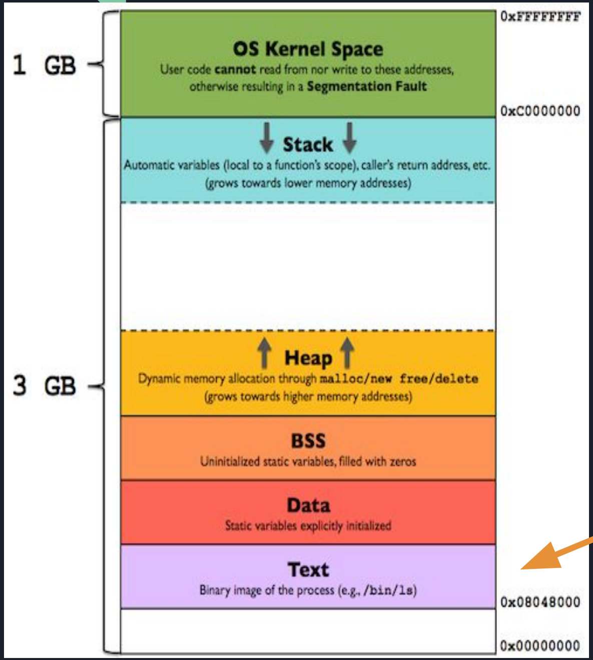
Stack Frame


1 GB
3 GB
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000

That means it has addresses!

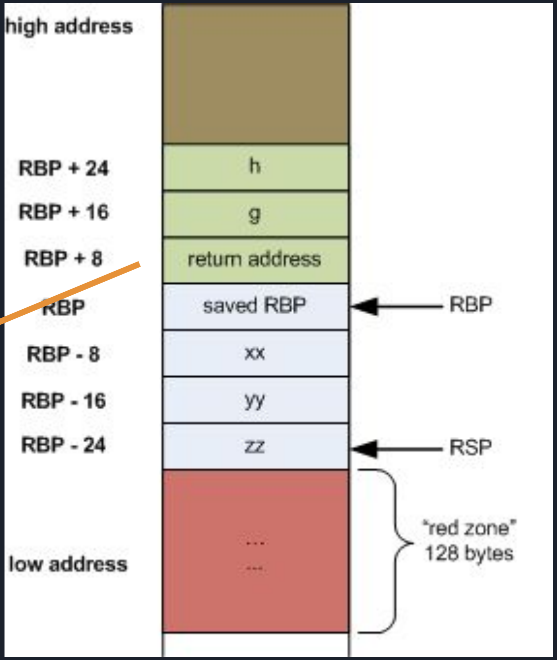


Stack Frame



1 GB
3 GB
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000

So the value in the return address is the address of where the function should return to



32 vs 64 bit

Calling conventions are the main difference.

- 32 bit uses stack
- 64 bit uses registers
    - Don’t memorize, just look them up!

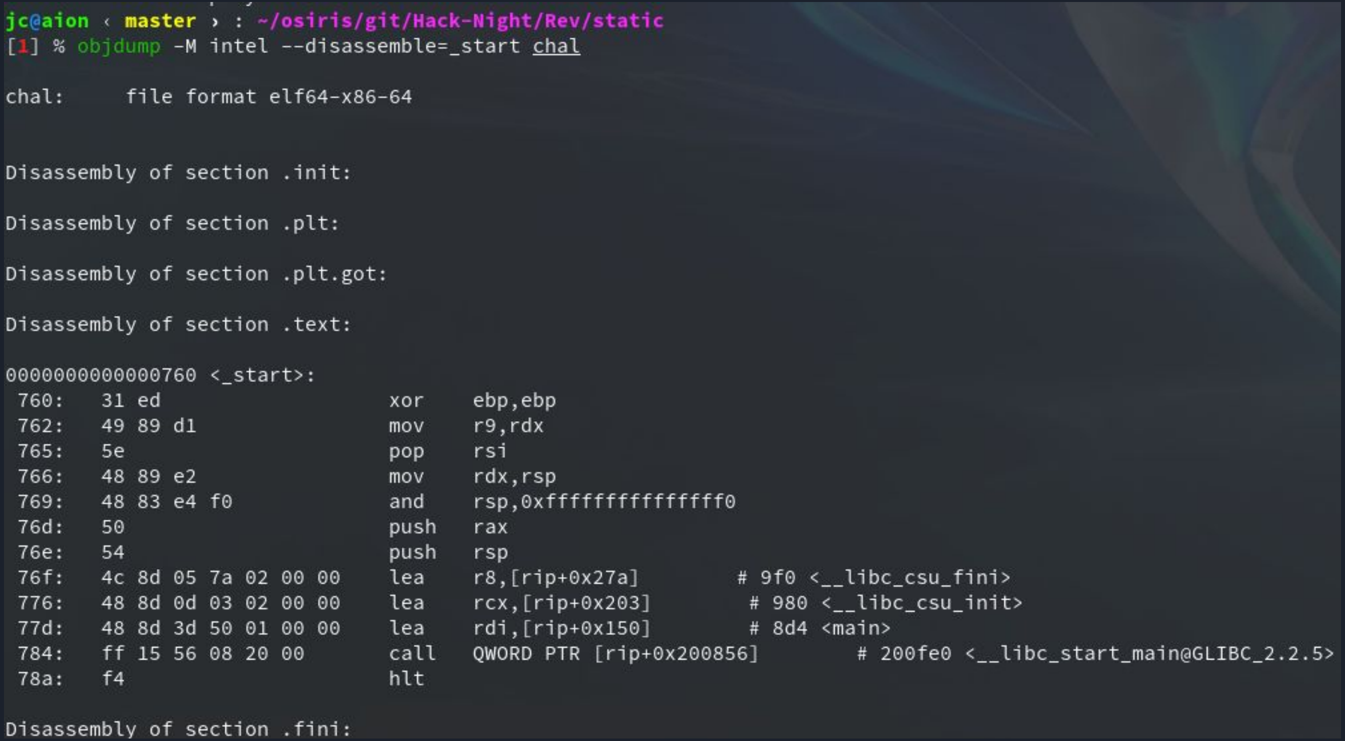
Tools

We'll take all the help we can get….

Objdump

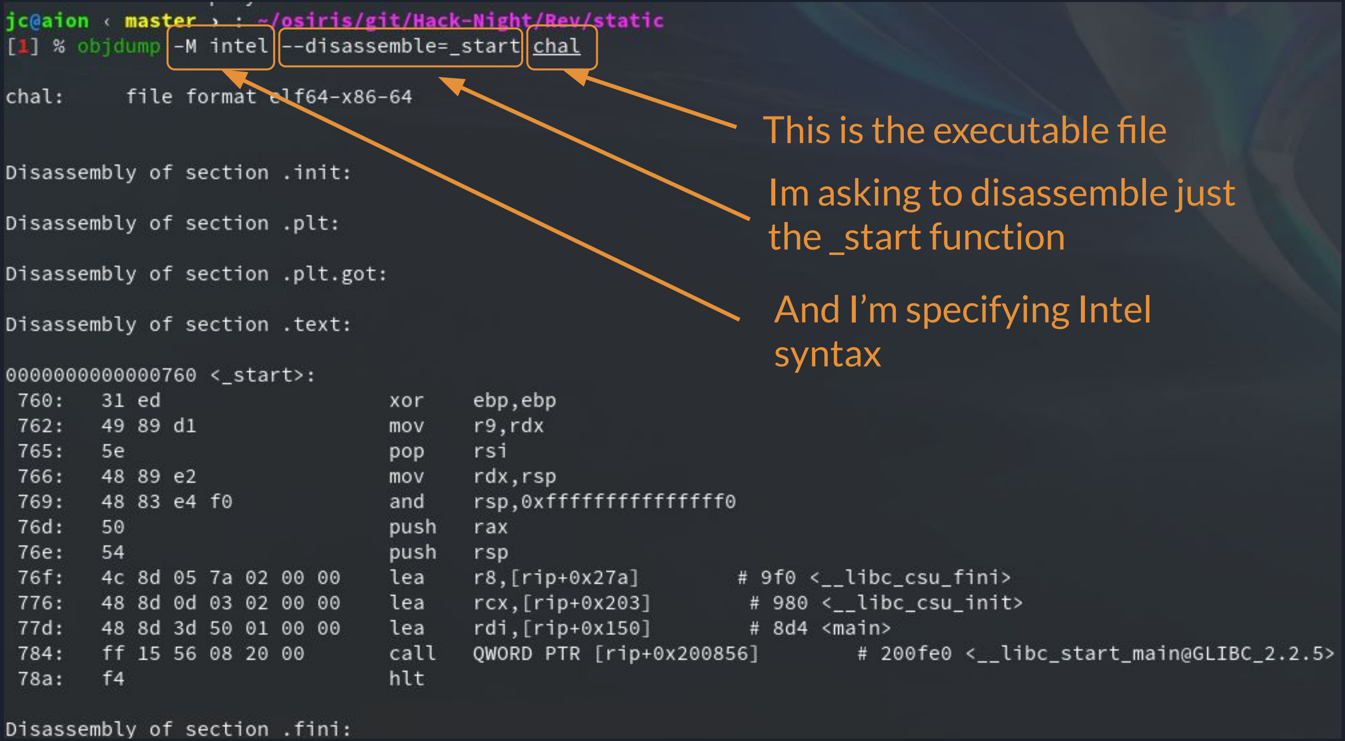
For when you have no other option…

Objdump - As basic as it gets



```
jc@aion ‹ master › : ~/osiris/git/Hack-Night/Rev/static
[1] % objdump -M intel --disassemble=_start chal

chal:     file format elf64-x86-64


Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .plt.got:

Disassembly of section .text:

0000000000000760 <_start>:
 760:   31 ed                   xor    ebp,ebp
 762:   49 89 d1                mov    r9,rdx
 765:   5e                      pop    rsi
 766:   48 89 e2                mov    rdx,rsp
 769:   48 83 e4 f0             and    rsp,0xfffffffffffffff0
 76d:   50                      push   rax
 76e:   54                      push   rsp
 76f:   4c 8d 05 7a 02 00 00    lea    r8,[rip+0x27a]        # 9f0 <__libc_csu_fini>
 776:   48 8d 0d 03 02 00 00    lea    rcx,[rip+0x203]        # 980 <__libc_csu_init>
 77d:   48 8d 3d 50 01 00 00    lea    rdi,[rip+0x150]        # 8d4 <main>
 784:   ff 15 56 08 20 00       call   QWORD PTR [rip+0x200856]        # 200fe0 <__libc_start_main@GLIBC_2.2.5>
 78a:   f4                      hlt

Disassembly of section .fini:
```

Objdump - As basic as it gets



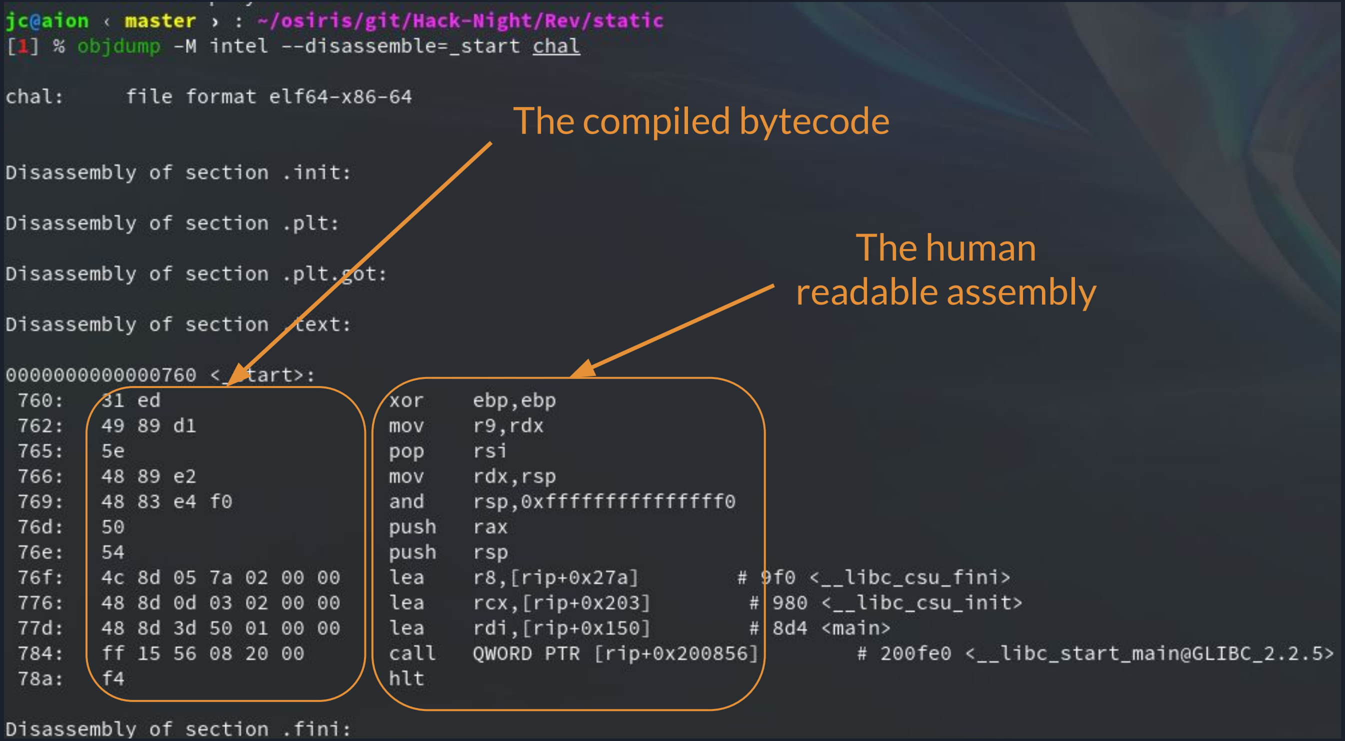
```
jc@aion ‹ master › : ~/osiris/git/Hack-Night/Rev/static
[1] % objdump -M intel --disassemble=_start chal

chal:     file format elf64-x86-64


Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .plt.got:

Disassembly of section .text:

0000000000000760 <_start>:
 760:   31 ed                   xor    ebp,ebp
 762:   49 89 d1                mov    r9,rdx
 765:   5e                      pop    rsi
 766:   48 89 e2                mov    rdx,rsp
 769:   48 83 e4 f0             and    rsp,0xfffffffffffffff0
 76d:   50                      push   rax
 76e:   54                      push   rsp
 76f:   4c 8d 05 7a 02 00 00    lea    r8,[rip+0x27a]        # 9f0 <__libc_csu_fini>
 776:   48 8d 0d 03 02 00 00    lea    rcx,[rip+0x203]        # 980 <__libc_csu_init>
 77d:   48 8d 3d 50 01 00 00    lea    rdi,[rip+0x150]        # 8d4 <main>
 784:   ff 15 56 08 20 00       call   QWORD PTR [rip+0x200856]        # 200fe0 <__libc_start_main@GLIBC_2.2.5>
 78a:   f4                      hlt

Disassembly of section .fini:
```

This is the executable file

Im asking to disassemble just the _start function

And I'm specifying Intel syntax

Objdump - As basic as it gets

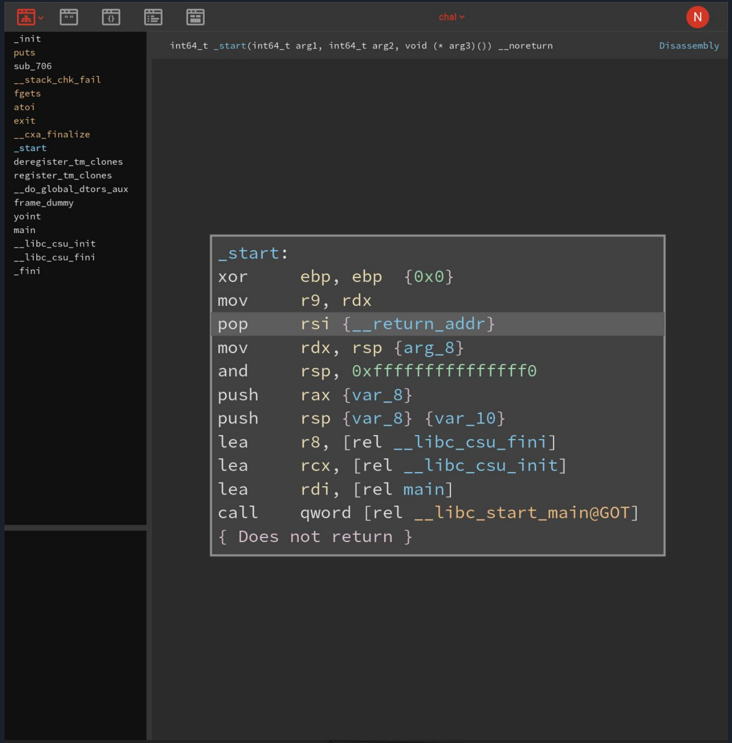


```
jc@aion ‹ master › : ~/osiris/git/Hack-Night/Rev/static
[1] % objdump -M intel --disassemble=_start chal

chal:     file format elf64-x86-64


Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .plt.got:

Disassembly of section .text:

0000000000000760 <_start>:
 760:	31 ed                	xor    ebp,ebp
 762:	49 89 d1             	mov    r9,rdx
 765:	5e                   	pop    rsi
 766:	48 89 e2             	mov    rdx,rsp
 769:	48 83 e4 f0          	and    rsp,0xfffffffffffffff0
 76d:	50                   	push   rax
 76e:	54                   	push   rsp
 76f:	4c 8d 05 7a 02 00 00 	lea    r8,[rip+0x27a]        # 9f0 <__libc_csu_fini>
 776:	48 8d 0d 03 02 00 00 	lea    rcx,[rip+0x203]        # 980 <__libc_csu_init>
 77d:	48 8d 3d 50 01 00 00 	lea    rdi,[rip+0x150]        # 8d4 <main>
 784:	ff 15 56 08 20 00    	call   QWORD PTR [rip+0x200856]        # 200fe0 <__libc_start_main@GLIBC_2.2.5>
 78a:	f4                   	hlt

Disassembly of section .fini:
```

Binary Ninja

cloud.binary.ninja

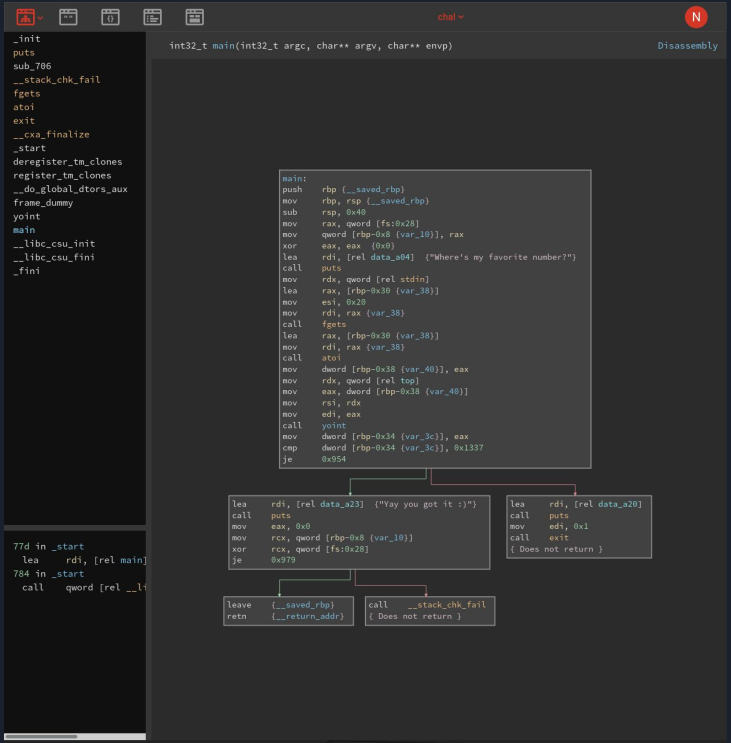
Binary Ninja - cloud



```
_init
puts
sub_706
__stack_chk_fail
fgets
atoi
exit
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
yoint
main
__libc_csu_init
__libc_csu_fini
_fini
```

```
int64_t _start(int64_t arg1, int64_t arg2, void (* arg3)()) __noreturn
```

```
_start:
xor     ebp, ebp  {0x0}
mov     r9, rdx
pop     rsi {__return_addr}
mov     rdx, rsp {arg_8}
and     rsp, 0xfffffffffffffff0
push    rax {var_8}
push    rsp {var_8} {var_10}
lea     r8, [rel __libc_csu_fini]
lea     rcx, [rel __libc_csu_init]
lea     rdi, [rel main]
call    qword [rel __libc_start_main@GOT]
{ Does not return }
```



```
_init
puts
sub_706
__stack_chk_fail
fgets
atoi
exit
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
yoint
main
__libc_csu_init
__libc_csu_fini
_fini
```

```
77d in _start
  lea     rdi, [rel main]
784 in _start
  call    qword [rel __li
```

```
int32_t main(int32_t argc, char** argv, char** envp)
```

```
main:
push    rbp {__saved_rbp}
mov     rbp, rsp {__saved_rbp}
sub     rsp, 0x40
mov     rax, qword [fs:0x28]
mov     qword [rbp-0x8 {var_10}], rax
xor     eax, eax  {0x0}
lea     rdi, [rel data_a04]  {"Where's my favorite number?"}
call    puts
mov     rdx, qword [rel stdin]
lea     rax, [rbp-0x30 {var_38}]
mov     esi, 0x20
mov     rdi, rax {var_38}
call    fgets
lea     rax, [rbp-0x30 {var_38}]
mov     rdi, rax {var_38}
call    atoi
mov     dword [rbp-0x38 {var_40}], eax
mov     rdx, qword [rel top]
mov     eax, dword [rbp-0x38 {var_40}]
mov     rsi, rdx
mov     edi, eax
call    yoint
mov     dword [rbp-0x34 {var_3c}], eax
cmp     dword [rbp-0x34 {var_3c}], 0x1337
je      0x954
```

```
lea     rdi, [rel data_a23]  {"Yay you got it :)"}
call    puts
mov     eax, 0x0
mov     rcx, qword [rbp-0x8 {var_10}]
xor     rcx, qword [fs:0x28]
je      0x979
```

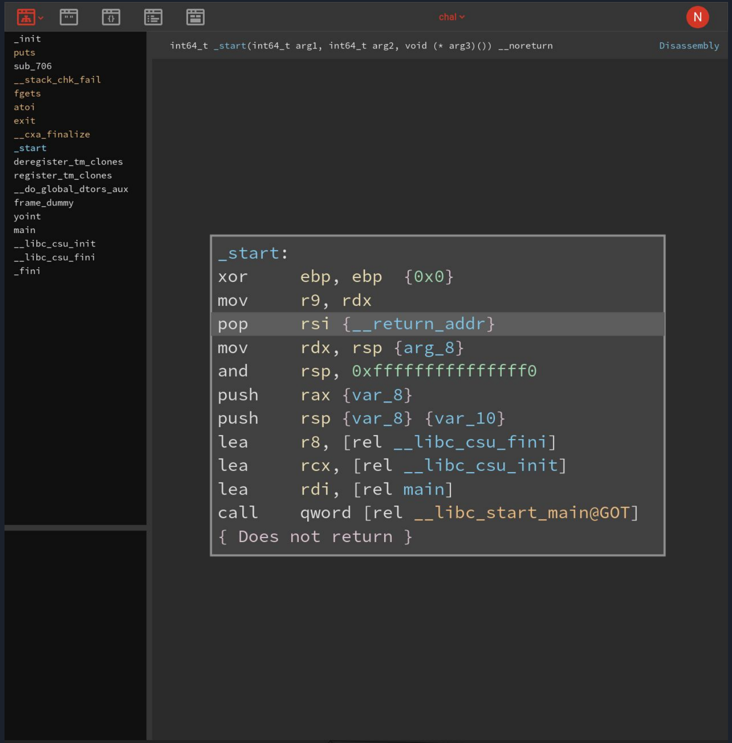
```
lea     rdi, [rel data_a20]
call    puts
mov     edi, 0x1
call    exit
{ Does not return }
```

```
leave   {__saved_rbp}
retn    {__return_addr}
```

```
call    __stack_chk_fail
{ Does not return }
```

Binary Ninja - cloud

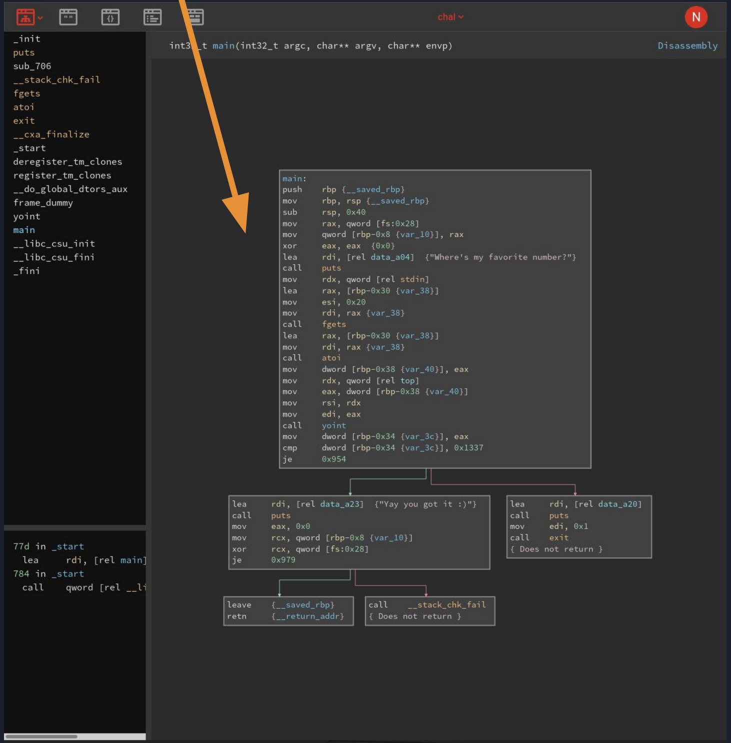
Graphs!



```
_init
puts
sub_706
__stack_chk_fail
fgets
atoi
exit
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
yoint
main
__libc_csu_init
__libc_csu_fini
_fini
```

```
int64_t _start(int64_t arg1, int64_t arg2, void (* arg3)()) __noreturn
```

```
_start:
xor     ebp, ebp  {0x0}
mov     r9, rdx
pop     rsi {__return_addr}
mov     rdx, rsp {arg_8}
and     rsp, 0xfffffffffffffff0
push    rax {var_8}
push    rsp {var_8} {var_10}
lea     r8, [rel __libc_csu_fini]
lea     rcx, [rel __libc_csu_init]
lea     rdi, [rel main]
call    qword [rel __libc_start_main@GOT]
{ Does not return }
```



```
_init
puts
sub_706
__stack_chk_fail
fgets
atoi
exit
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
yoint
main
__libc_csu_init
__libc_csu_fini
_fini
```

```
77d in _start
  lea     rdi, [rel main]
784 in _start
  call    qword [rel __li
```

```
int32_t main(int32_t argc, char** argv, char** envp)
```

```
main:
push    rbp {__saved_rbp}
mov     rbp, rsp {__saved_rbp}
sub     rsp, 0x40
mov     rax, qword [fs:0x28]
mov     qword [rbp-0x8 {var_10}], rax
xor     eax, eax  {0x0}
lea     rdi, [rel data_a04]  {"Where's my favorite number?"}
call    puts
mov     rdx, qword [rel stdin]
lea     rax, [rbp-0x30 {var_38}]
mov     esi, 0x20
mov     rdi, rax {var_38}
call    fgets
lea     rax, [rbp-0x30 {var_38}]
mov     rdi, rax {var_38}
call    atoi
mov     dword [rbp-0x38 {var_40}], eax
mov     rdx, qword [rel top]
mov     eax, dword [rbp-0x38 {var_40}]
mov     rsi, rdx
mov     edi, eax
call    yoint
mov     dword [rbp-0x34 {var_3c}], eax
cmp     dword [rbp-0x34 {var_3c}], 0x1337
je      0x954
```

```
lea     rdi, [rel data_a23]  {"Yay you got it :)"}
call    puts
mov     eax, 0x0
mov     rcx, qword [rbp-0x8 {var_10}]
xor     rcx, qword [fs:0x28]
je      0x979
```

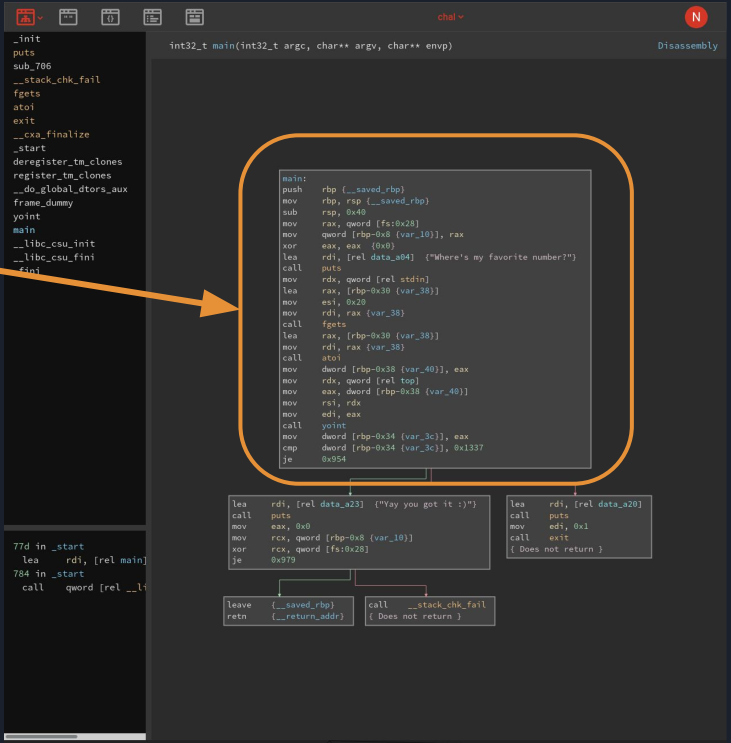
```
lea     rdi, [rel data_a20]
call    puts
mov     edi, 0x1
call    exit
{ Does not return }
```

```
leave   {__saved_rbp}
retn    {__return_addr}
```

```
call    __stack_chk_fail
{ Does not return }
```

Binary Ninja - cloud

We call these chunks of code in between jumps "Basic Blocks"



```
_init
puts
sub_706
__stack_chk_fail
fgets
atoi
exit
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
yoint
main
__libc_csu_init
__libc_csu_fini
_fini
```

```
77d in _start
  lea     rdi, [rel main]
784 in _start
  call    qword [rel __li
```

```
int32_t main(int32_t argc, char** argv, char** envp)
```

```
main:
push    rbp {__saved_rbp}
mov     rbp, rsp {__saved_rbp}
sub     rsp, 0x40
mov     rax, qword [fs:0x28]
mov     qword [rbp-0x8 {var_10}], rax
xor     eax, eax  {0x0}
lea     rdi, [rel data_a04]  {"Where's my favorite number?"}
call    puts
mov     rdx, qword [rel stdin]
lea     rax, [rbp-0x30 {var_38}]
mov     esi, 0x20
mov     rdi, rax {var_38}
call    fgets
lea     rax, [rbp-0x30 {var_38}]
mov     rdi, rax {var_38}
call    atoi
mov     dword [rbp-0x38 {var_40}], eax
mov     rdx, qword [rel top]
mov     eax, dword [rbp-0x38 {var_40}]
mov     rsi, rdx
mov     edi, eax
call    yoint
mov     dword [rbp-0x34 {var_3c}], eax
cmp     dword [rbp-0x34 {var_3c}], 0x1337
je      0x954
```

```
lea     rdi, [rel data_a23]  {"Yay you got it :)"}
call    puts
mov     eax, 0x0
mov     rcx, qword [rbp-0x8 {var_10}]
xor     rcx, qword [fs:0x28]
je      0x979
```

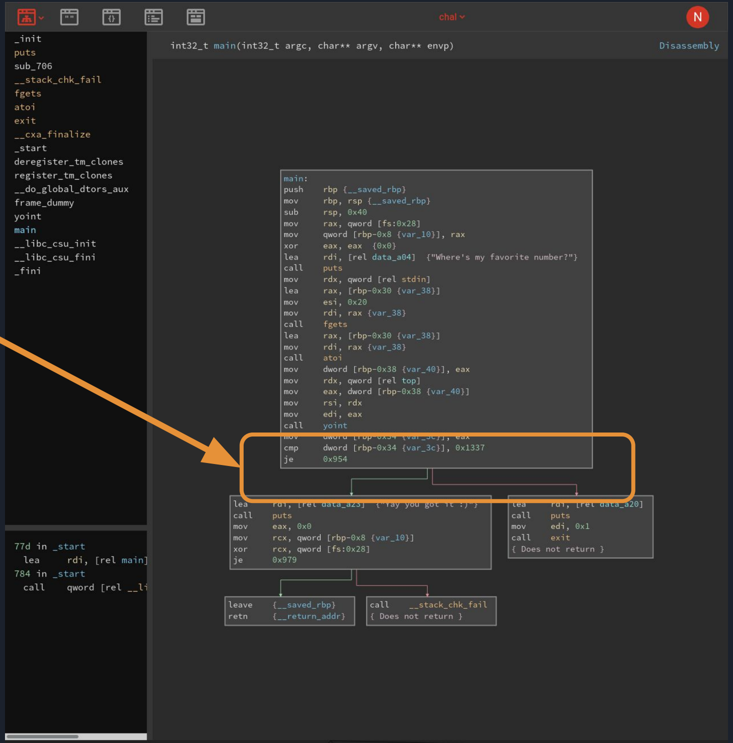
```
lea     rdi, [rel data_a20]
call    puts
mov     edi, 0x1
call    exit
{ Does not return }
```

```
leave   {__saved_rbp}
retn    {__return_addr}
```

```
call    __stack_chk_fail
{ Does not return }
```

Binary Ninja - cloud

Down at the bottom there is a cmp, then je



```
_init
puts
sub_706
__stack_chk_fail
fgets
atoi
exit
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
yoint
main
__libc_csu_init
__libc_csu_fini
_fini
```

```
77d in _start
  lea     rdi, [rel main]
784 in _start
  call    qword [rel __li
```

```
int32_t main(int32_t argc, char** argv, char** envp)
```

```
main:
push    rbp {__saved_rbp}
mov     rbp, rsp {__saved_rbp}
sub     rsp, 0x40
mov     rax, qword [fs:0x28]
mov     qword [rbp-0x8 {var_10}], rax
xor     eax, eax  {0x0}
lea     rdi, [rel data_a04]  {"Where's my favorite number?"}
call    puts
mov     rdx, qword [rel stdin]
lea     rax, [rbp-0x30 {var_38}]
mov     esi, 0x20
mov     rdi, rax {var_38}
call    fgets
lea     rax, [rbp-0x30 {var_38}]
mov     rdi, rax {var_38}
call    atoi
mov     dword [rbp-0x38 {var_40}], eax
mov     rdx, qword [rel top]
mov     eax, dword [rbp-0x38 {var_40}]
mov     rsi, rdx
mov     edi, eax
call    yoint
mov     dword [rbp-0x34 {var_3c}], eax
cmp     dword [rbp-0x34 {var_3c}], 0x1337
je      0x954
```

```
lea     rdi, [rel data_a23]  {"Yay you got it :)"}
call    puts
mov     eax, 0x0
mov     rcx, qword [rbp-0x8 {var_10}]
xor     rcx, qword [fs:0x28]
je      0x979
```

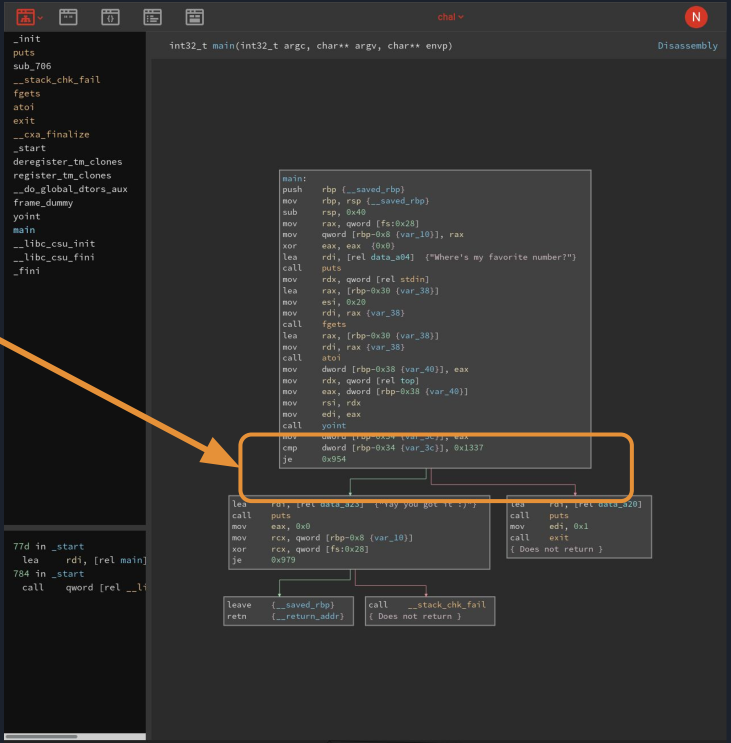
```
lea     rdi, [rel data_a20]
call    puts
mov     edi, 0x1
call    exit
{ Does not return }
```

```
leave   {__saved_rbp}
retn    {__return_addr}
```

```
call    __stack_chk_fail
{ Does not return }
```

Binary Ninja - cloud

This is a “conditional jump”



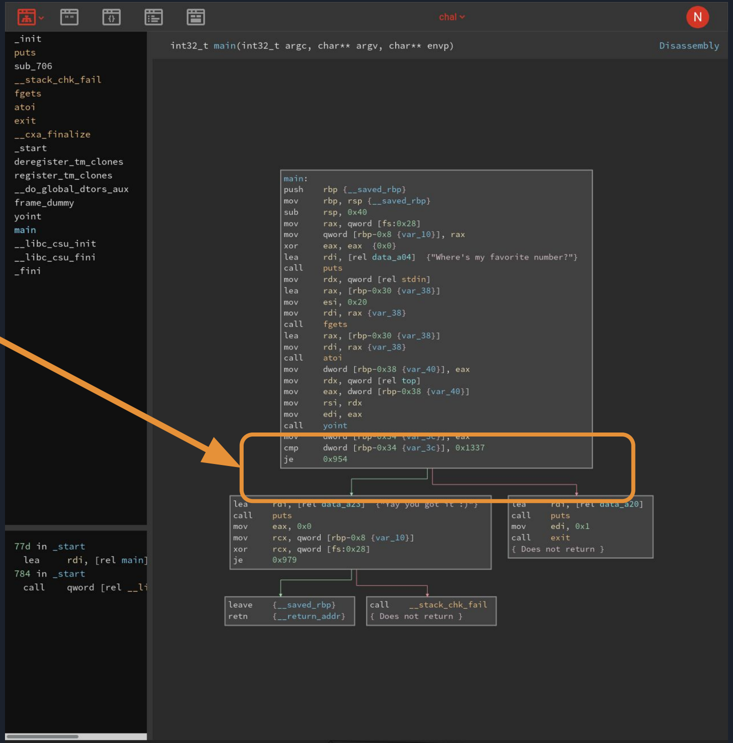
```
_init
puts
sub_706
__stack_chk_fail
fgets
atoi
exit
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
yoint
main
__libc_csu_init
__libc_csu_fini
_fini
```

```
77d in _start
  lea     rdi, [rel main]
784 in _start
  call    qword [rel __li
```

```
int32_t main(int32_t argc, char** argv, char** envp)

main:
push    rbp {__saved_rbp}
mov     rbp, rsp {__saved_rbp}
sub     rsp, 0x40
mov     rax, qword [fs:0x28]
mov     qword [rbp-0x8 {var_10}], rax
xor     eax, eax  {0x0}
lea     rdi, [rel data_a04]  {"Where's my favorite number?"}
call    puts
mov     rdx, qword [rel stdin]
lea     rax, [rbp-0x30 {var_38}]
mov     esi, 0x20
mov     rdi, rax {var_38}
call    fgets
lea     rax, [rbp-0x30 {var_38}]
mov     rdi, rax {var_38}
call    atoi
mov     dword [rbp-0x38 {var_40}], eax
mov     rdx, qword [rel top]
mov     eax, dword [rbp-0x38 {var_40}]
mov     rsi, rdx
mov     edi, eax
call    yoint
mov     dword [rbp-0x34 {var_3c}], eax
cmp     dword [rbp-0x34 {var_3c}], 0x1337
je      0x954

lea     rdi, [rel data_a23]  {"Yay you got it :)"}
call    puts
mov     eax, 0x0
mov     rcx, qword [rbp-0x8 {var_10}]
xor     rcx, qword [fs:0x28]
je      0x979

lea     rdi, [rel data_a20]
call    puts
mov     edi, 0x1
call    exit
{ Does not return }

leave   {__saved_rbp}
retn    {__return_addr}

call    __stack_chk_fail
{ Does not return }
```

Binary Ninja - cloud

Think of this as an if statement



```
_init
puts
sub_706
__stack_chk_fail
fgets
atoi
exit
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
yoint
main
__libc_csu_init
__libc_csu_fini
_fini
```

```
77d in _start
  lea     rdi, [rel main]
784 in _start
  call    qword [rel __li
```

```
int32_t main(int32_t argc, char** argv, char** envp)
```

```
main:
push    rbp {__saved_rbp}
mov     rbp, rsp {__saved_rbp}
sub     rsp, 0x40
mov     rax, qword [fs:0x28]
mov     qword [rbp-0x8 {var_10}], rax
xor     eax, eax  {0x0}
lea     rdi, [rel data_a04]  {"Where's my favorite number?"}
call    puts
mov     rdx, qword [rel stdin]
lea     rax, [rbp-0x30 {var_38}]
mov     esi, 0x20
mov     rdi, rax {var_38}
call    fgets
lea     rax, [rbp-0x30 {var_38}]
mov     rdi, rax {var_38}
call    atoi
mov     dword [rbp-0x38 {var_40}], eax
mov     rdx, qword [rel top]
mov     eax, dword [rbp-0x38 {var_40}]
mov     rsi, rdx
mov     edi, eax
call    yoint
mov     dword [rbp-0x34 {var_3c}], eax
cmp     dword [rbp-0x34 {var_3c}], 0x1337
je      0x954
```

```
lea     rdi, [rel data_a23]  {"Yay you got it :)"}
call    puts
mov     eax, 0x0
mov     rcx, qword [rbp-0x8 {var_10}]
xor     rcx, qword [fs:0x28]
je      0x979
```

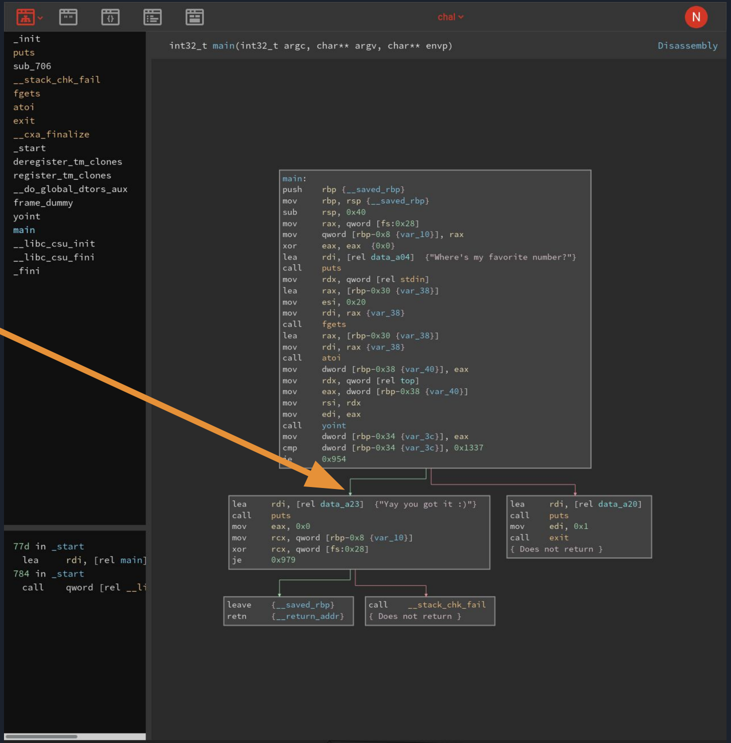
```
lea     rdi, [rel data_a20]
call    puts
mov     edi, 0x1
call    exit
{ Does not return }
```

```
leave   {__saved_rbp}
retn    {__return_addr}
```

```
call    __stack_chk_fail
{ Does not return }
```

Binary Ninja - cloud

If true, jump to this block



```
_init
puts
sub_706
__stack_chk_fail
fgets
atoi
exit
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
yoint
main
__libc_csu_init
__libc_csu_fini
_fini
```

```
77d in _start
  lea     rdi, [rel main]
784 in _start
  call    qword [rel __li
```

```
int32_t main(int32_t argc, char** argv, char** envp)
```

```
main:
push    rbp {__saved_rbp}
mov     rbp, rsp {__saved_rbp}
sub     rsp, 0x40
mov     rax, qword [fs:0x28]
mov     qword [rbp-0x8 {var_10}], rax
xor     eax, eax  {0x0}
lea     rdi, [rel data_a04]  {"Where's my favorite number?"}
call    puts
mov     rdx, qword [rel stdin]
lea     rax, [rbp-0x30 {var_38}]
mov     esi, 0x20
mov     rdi, rax {var_38}
call    fgets
lea     rax, [rbp-0x30 {var_38}]
mov     rdi, rax {var_38}
call    atoi
mov     dword [rbp-0x38 {var_40}], eax
mov     rdx, qword [rel top]
mov     eax, dword [rbp-0x38 {var_40}]
mov     rsi, rdx
mov     edi, eax
call    yoint
mov     dword [rbp-0x34 {var_3c}], eax
cmp     dword [rbp-0x34 {var_3c}], 0x1337
je      0x954
```

```
lea     rdi, [rel data_a23]  {"Yay you got it :)"}
call    puts
mov     eax, 0x0
mov     rcx, qword [rbp-0x8 {var_10}]
xor     rcx, qword [fs:0x28]
je      0x979
```

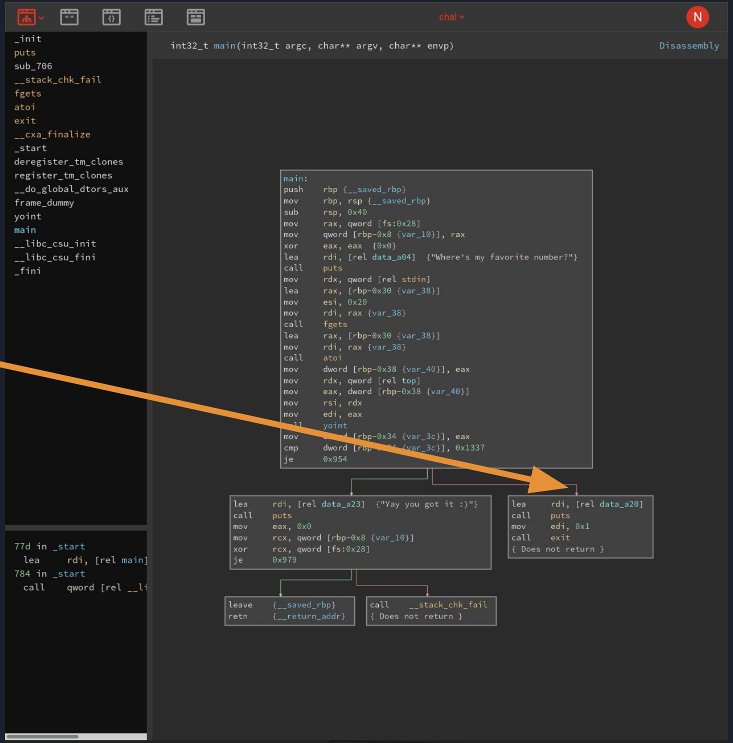
```
lea     rdi, [rel data_a20]
call    puts
mov     edi, 0x1
call    exit
{ Does not return }
```

```
leave   {__saved_rbp}
retn    {__return_addr}
```

```
call    __stack_chk_fail
{ Does not return }
```

Binary Ninja - cloud

Else, jump to this block



```
_init
puts
sub_706
__stack_chk_fail
fgets
atoi
exit
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
yoint
main
__libc_csu_init
__libc_csu_fini
_fini
```

```
77d in _start
  lea     rdi, [rel main]
784 in _start
  call    qword [rel __li
```

```
int32_t main(int32_t argc, char** argv, char** envp)
```

```
main:
push    rbp {__saved_rbp}
mov     rbp, rsp {__saved_rbp}
sub     rsp, 0x40
mov     rax, qword [fs:0x28]
mov     qword [rbp-0x8 {var_10}], rax
xor     eax, eax  {0x0}
lea     rdi, [rel data_a04]  {"Where's my favorite number?"}
call    puts
mov     rdx, qword [rel stdin]
lea     rax, [rbp-0x30 {var_38}]
mov     esi, 0x20
mov     rdi, rax {var_38}
call    fgets
lea     rax, [rbp-0x30 {var_38}]
mov     rdi, rax {var_38}
call    atoi
mov     dword [rbp-0x38 {var_40}], eax
mov     rdx, qword [rel top]
mov     eax, dword [rbp-0x38 {var_40}]
mov     rsi, rdx
mov     edi, eax
call    yoint
mov     dword [rbp-0x34 {var_3c}], eax
cmp     dword [rbp-0x34 {var_3c}], 0x1337
je      0x954
```

```
lea     rdi, [rel data_a23]  {"Yay you got it :)"}
call    puts
mov     eax, 0x0
mov     rcx, qword [rbp-0x8 {var_10}]
xor     rcx, qword [fs:0x28]
je      0x979
```

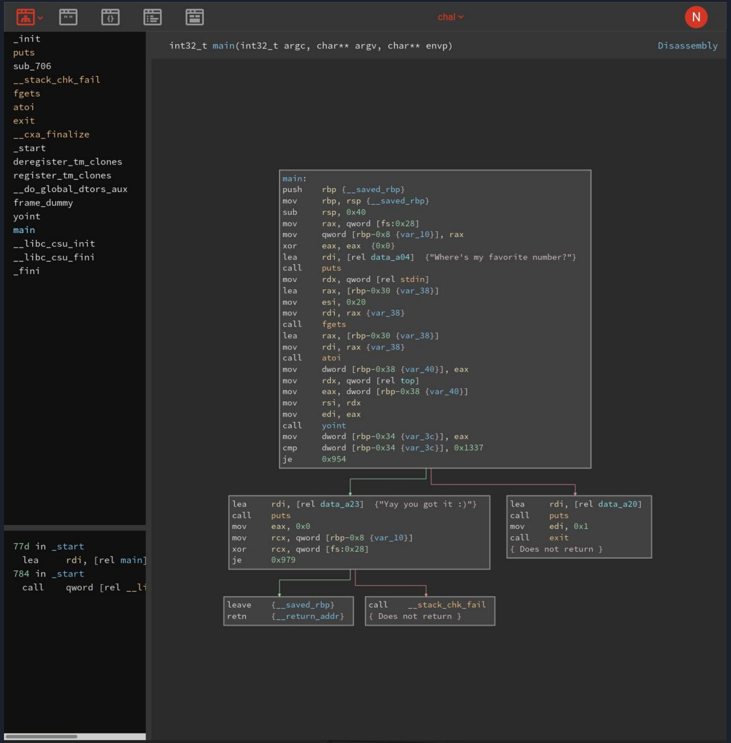
```
lea     rdi, [rel data_a20]
call    puts
mov     edi, 0x1
call    exit
{ Does not return }
```

```
leave   {__saved_rbp}
retn    {__return_addr}
```

```
call    __stack_chk_fail
{ Does not return }
```

Binary Ninja - cloud

This type of “graph” view is the most peoples prefered way to read assembly



```
_init
puts
sub_706
__stack_chk_fail
fgets
atoi
exit
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
yoint
main
__libc_csu_init
__libc_csu_fini
_fini
```

```
77d in _start
  lea     rdi, [rel main]
784 in _start
  call    qword [rel __li
```

```
int32_t main(int32_t argc, char** argv, char** envp)
```

```
main:
push    rbp {__saved_rbp}
mov     rbp, rsp {__saved_rbp}
sub     rsp, 0x40
mov     rax, qword [fs:0x28]
mov     qword [rbp-0x8 {var_10}], rax
xor     eax, eax  {0x0}
lea     rdi, [rel data_a04]  {"Where's my favorite number?"}
call    puts
mov     rdx, qword [rel stdin]
lea     rax, [rbp-0x30 {var_38}]
mov     esi, 0x20
mov     rdi, rax {var_38}
call    fgets
lea     rax, [rbp-0x30 {var_38}]
mov     rdi, rax {var_38}
call    atoi
mov     dword [rbp-0x38 {var_40}], eax
mov     rdx, qword [rel top]
mov     eax, dword [rbp-0x38 {var_40}]
mov     rsi, rdx
mov     edi, eax
call    yoint
mov     dword [rbp-0x34 {var_3c}], eax
cmp     dword [rbp-0x34 {var_3c}], 0x1337
je      0x954
```

```
lea     rdi, [rel data_a23]  {"Yay you got it :)"}
call    puts
mov     eax, 0x0
mov     rcx, qword [rbp-0x8 {var_10}]
xor     rcx, qword [fs:0x28]
je      0x979
```

```
lea     rdi, [rel data_a20]
call    puts
mov     edi, 0x1
call    exit
{ Does not return }
```

```
leave   {__saved_rbp}
retn    {__return_addr}
```

```
call    __stack_chk_fail
{ Does not return }
```

Godbolt.org

Connecting it back to C / C++

Questions?