Source: https://xkcd.com/138/

Reverse Engineering

Week 1: Introduction to Disassembly, Layout and a Little Linking

Instructor: Sergey Bratus
Contributions and guest lectures: John Berry, Travis Goodspeed, Ryan Speers, more TBA

Dartmouth College -- Winter 2022

What is Reverse Engineering?

Goals of the course

- Rapid immersion: real systems, real tools, real specs, lots of surprising complications
  - You'll need to explore, search (a lot), and prioritize, not just follow the examples
- RE requires some creativity and intuition that is only developed through practice, not just listening to lectures
- You won't be taught everything you need to know in lectures. To be good at RE you have to learn to find answers on your own. Google is your friend.
- By the end of the course you should be able to take a blob of code and figure out what it does. (within reason)
- You will be exposed to malicious software (malware) so you will gain an understanding of what you have seen in the news about ransomware, etc

What is Reverse Engineering?

Homework

- There will be homework after every class that is due prior to the next one
  - Tuesday homework will be easier and are designed to solidify what you learned that day
  - Thursday homework will be a little more complex
- Many small assignments are better than fewer large ones. Getting better at RE is like learning a language. A little every day is better than a lot crammed just before it is due.

What is Reverse Engineering?

Mid-term & Final

- The course is project-based & all exams will be take-home
  - Subject to the Dartmouth Honor Code
- You will be given some software that you will RE and write up what it does
  - Probably some other small tasks
- If you want to go above and beyond the coursework, individual research projects and Senior Honors Theses will be encouraged
  - Improving state-of-the-art RE tools such as Ghidra and Binary Ninja is strongly encouraged

What is Reverse Engineering?

And why do we do it?

- Engineers work from a Design to an Artifact
- Reverse Engineers work backward, from an Artifact back to a Design
- In computing, this is often working from an Executable back to Source Code
- This is useful for many reasons:
  - We can preserve and sustain old software by emulating it
  - We can find security bugs without source code
  - We can copy software, or determine whether one program copies another

What is Reverse Engineering?

And is it legal?

We didn't talk about this in class, but we will.

- This isn't a course in law, nor are any of us law experts. Seek your own legal advice.
- There are many legal uses for Reverse Engineering, but also there are potential violations of law or contracts.
- The Electronic Frontier Foundation (EFF) has a helpful guide for reference at **https://www.eff.org/issues/coders/reverse-engineering-faq**
- "Five areas of United States law are particularly relevant for computer scientists engaging in reverse engineering:
  - Copyright law and fair use, codified at 17 U.S.C. 107;
  - Trade secret law;
  - The anti-circumvention provisions of the Digital Millennium Copyright Act (DMCA), codified at 17 U.S.C. section 1201;
  - Contract law, if use of the software is subject to an End User License Agreement (EULA), Terms of Service notice (TOS), Terms of Use notice (TOU), Non-Disclosure Agreement (NDA), developer agreement or API agreement; and
  - The Electronic Communications Privacy Act, codified at 18 U.S.C. 2510 et. seq." (-EFF)

What is Reverse Engineering?

And how is it done?

- Source Code is compiled and linked into Machine Code.
- Machine Code looks like this:
  - E8 F9 CA AD DE
- Machine Code translates directly to Assembly Code, like this:
  - CALL 0xDEADCAFE
- At a low level, we're just reading disassembly and annotating it to be legible
- At a high level, we're also trying to understand the program design

What is Reverse Engineering?

And how is it done?

- You can learn to read Disassembly, but there are complications:
  - It is very verbose, much more so than C
  - It often lacks variable and function names
- Tools can help!
  - Decompiling the Disassembly into C, or something like C
  - Accepting new variable and function names
  - Transferring symbol names between different programs

What is Reverse Engineering?

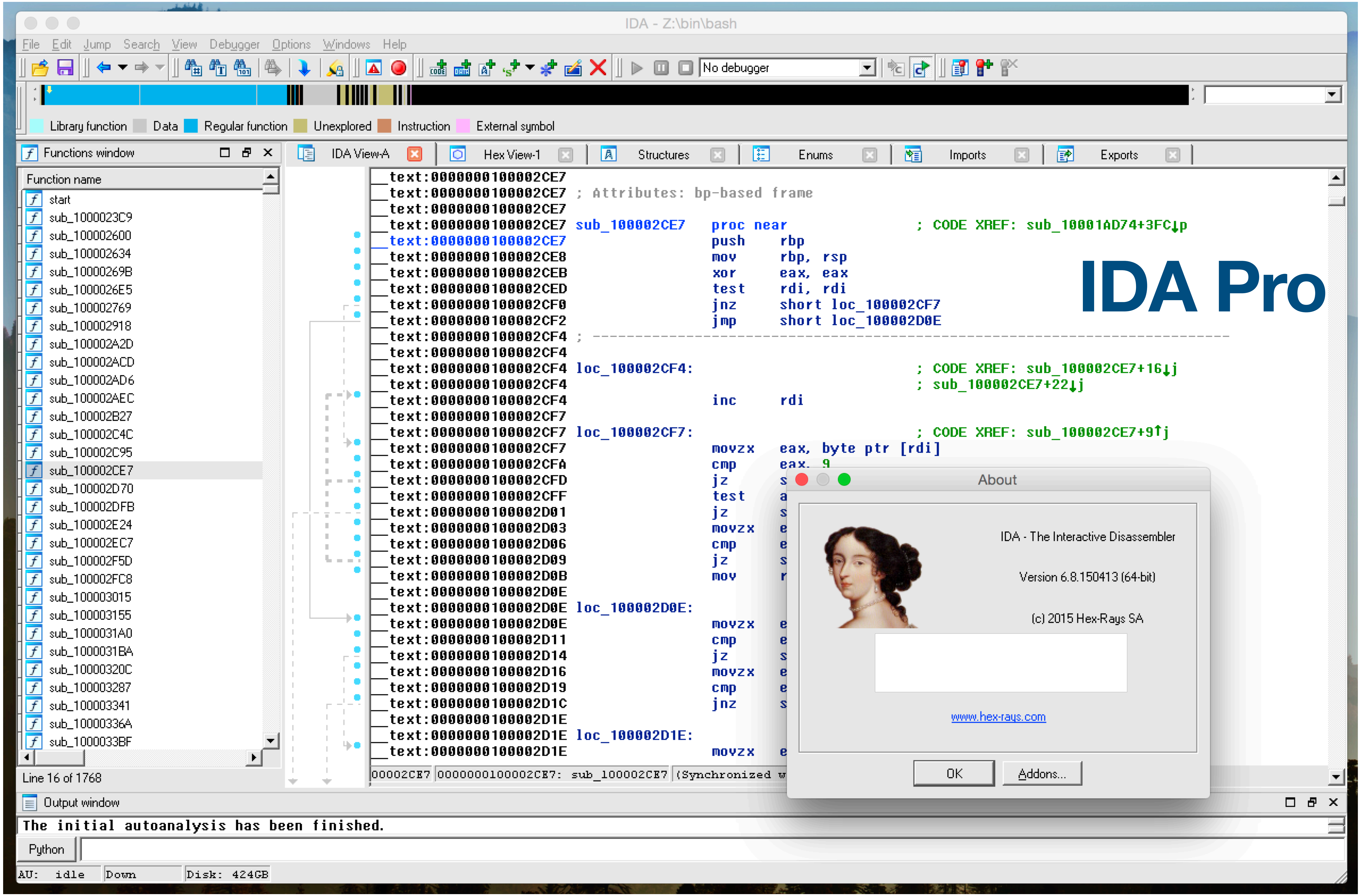
And what tools make it easier?

- IDA Pro
  - First popular Interactive Disassembler
  - First commercially useful Decompiler
- **Ghidra**
  - NSA's internal tool for reverse engineering, now with a declassified & free version
- Binary Ninja
  - Commercial disassembler with clean scripting

We will use Ghidra extensively.

Your final project or Honors Thesis could be a Ghidra plugin or another Ghidra improvement

IDA Pro



IDA - Z:\bin\bash
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Data Regular function Unexplored Instruction External symbol
Functions window
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
Function name
start
sub_1000023C9
sub_100002600
sub_100002634
sub_10000269B
sub_1000026E5
sub_100002769
sub_100002918
sub_100002A2D
sub_100002ACD
sub_100002AD6
sub_100002AEC
sub_100002B27
sub_100002C4C
sub_100002C95
sub_100002CE7
sub_100002D70
sub_100002DFB
sub_100002E24
sub_100002EC7
sub_100002F5D
sub_100002FC8
sub_100003015
sub_100003155
sub_1000031A0
sub_1000031BA
sub_10000320C
sub_100003287
sub_100003341
sub_10000336A
sub_1000033BF
Line 16 of 1768
__text:0000000100002CE7 ; Attributes: bp-based frame
__text:0000000100002CE7 sub_100002CE7 proc near ; CODE XREF: sub_10001AD74+3FC↓p
__text:0000000100002CE7 push rbp
__text:0000000100002CE8 mov rbp, rsp
__text:0000000100002CEB xor eax, eax
__text:0000000100002CED test rdi, rdi
__text:0000000100002CF0 jnz short loc_100002CF7
__text:0000000100002CF2 jmp short loc_100002D0E
__text:0000000100002CF4 loc_100002CF4: ; CODE XREF: sub_100002CE7+16↓j
; sub_100002CE7+22↓j
__text:0000000100002CF4 inc rdi
__text:0000000100002CF7 loc_100002CF7: ; CODE XREF: sub_100002CE7+9↑j
__text:0000000100002CF7 movzx eax, byte ptr [rdi]
__text:0000000100002D0E loc_100002D0E:
__text:0000000100002D1E loc_100002D1E:
00002CE7 0000000100002CE7: sub_100002CE7 (Synchronized w
About
IDA - The Interactive Disassembler
Version 6.8.150413 (64-bit)
(c) 2015 Hex-Rays SA
www.hex-rays.com
OK
Addons...
Output window
The initial autoanalysis has been finished.
Python
AU: idle
Down
Disk: 424GB

The Basics of Reverse Engineering

What it's like inside a C program.

```
air% cat pointers.c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

int main(int argc, char **argv){
  void *heapthing=malloc(512);
  printf("Main is at 0x%08llx.\n", (uint64_t) main);
  printf("The call stack is near 0x%08llx.\n",
         (uint64_t) &argc);
  printf("The heap is near 0x%08llx.\n",
         (uint64_t) heapthing);

}
air% ./pointers
Main is at 0x100003e84.
The call stack is near 0x16fdff76c.
The heap is near 0x100304310.
air%
```

Run this program several times.
If you see different addresses between runs, you are seeing the effects of ASLR.

Note which part of each address doesn't change: that's because ASLR is at page granularity (typically 4Kbytes)

The Basics of Reverse Engineering

What does Disassembly look like?

```
00101169 - main
undefined main()
        undefined         AL:1           <RETURN>
        undefined8        Stack[-0x10]:8 local_10
        undefined4        Stack[-0x1c]:4 local_1c
        undefined8        Stack[-0x28]:8 local_28
            main
...1169 END...
...116d PUSH  RBP
...116e MOV   RBP,RSP
...1171 SUB   RSP,0x20
...1175 MOV   dword ptr [RBP + local_1c]...
...1178 MOV   qword ptr [RBP + local_28]...
...117c MOV   EDI,0x200
...1181 CALL  <EXTERNAL>::malloc
...1186 MOV   qword ptr [RBP + local_10]...
...118a LEA   RAX,[main]
...1191 MOV   RSI=>main,RAX
...1194 LEA   RDI,[s_Main_is_at_0x%08llx...
...119b MOV   EAX,0x0
...11a0 CALL  <EXTERNAL>::printf
...11a5 LEA   RAX=>local_1c,[RBP + -0x14]
...11a9 MOV   RSI,RAX
...11ac LEA   RDI,[s_The_call_stack_is_n...
...11b3 MOV   EAX,0x0
...11b8 CALL  <EXTERNAL>::printf
...11bd MOV   RAX,qword ptr [RBP + local...
...11c1 MOV   RSI,RAX
...11c4 LEA   RDI,[s_The_heap_is_near_0x...
...11cb MOV   EAX,0x0
...11d0 CALL  <EXTERNAL>::printf
...11d5 MOV   EAX,0x0
...11da LEAVE
...11db RET
```

Note Intel syntax of this disassembler

The Basics of Reverse Engineering

What does Decompiled C look like?

- Denser than Assembly
- Some reasons it is difficult to read:
  - Missing variables names.
- Potential inaccuracies:
  - Missing arguments.

Decompile: main - (pointers)

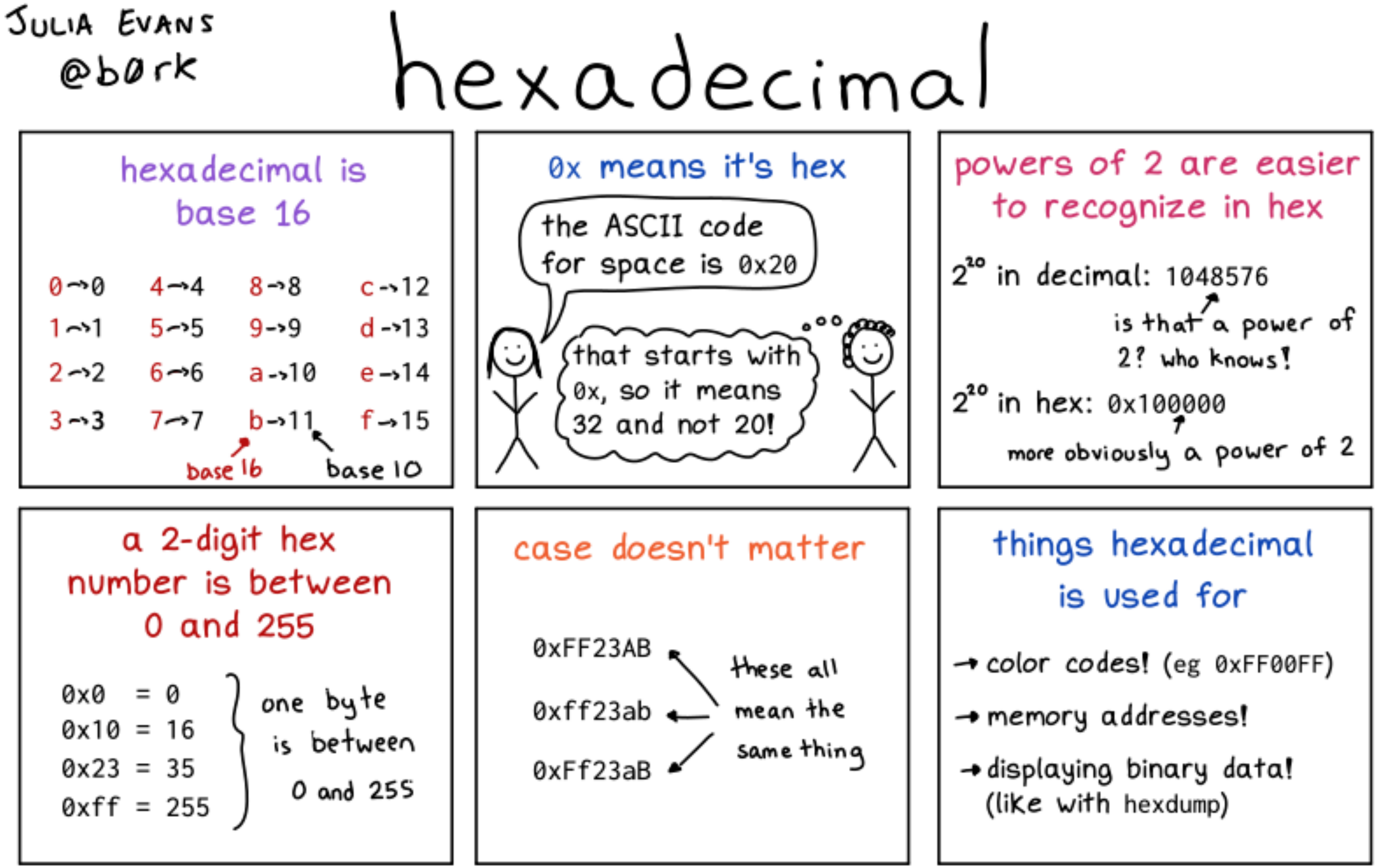
```c

undefined8 main(undefined4 param_1)

{
  undefined4 local_1c [3];
  void *local_10;

  local_1c[0] = param_1;
  local_10 = malloc(0x200);
  printf("Main is at 0x%08llx.\n",main);
  printf("The call stack is near 0x%08llx.\n",local_1c);
  printf("The heap is near 0x%08llx.\n",local_10);
  return 0;
}

```

A Quick Intro to Assembly Languages

- There are many of these languages, and they are different.
- This course focuses on x86 and will dabble in ARM.
  - ARM has three major dialects: ARM32, Thumb2, and ARM64
  - x86 has two major dialects: x86 and x86_64/amd64.
- You will be writing a little assembly, but reading a <u>lot</u> of it.

A Quick Intro to Assembly Languages

Hexadecimal



A Quick Intro to Assembly Languages

Registers

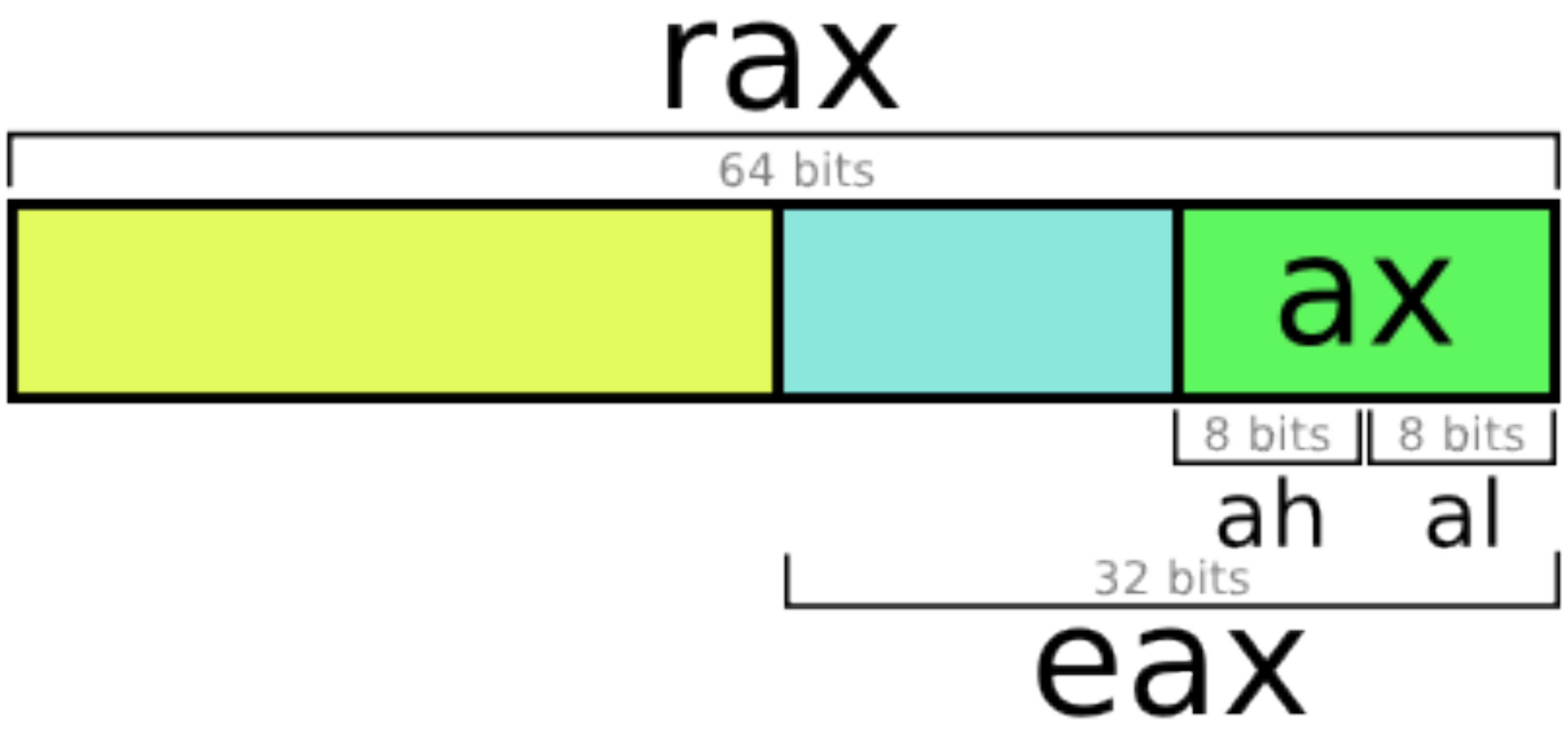
- Registers are like small variables that exist in hardware
- On x86_64 there are a lot of them but here are the most common
  - RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8-R15, and RIP
  - These are all 64 bits in length.
  - RIP (Instruction Pointer) is a special register that points to the next instruction to be executed. It is also commonly referred to as the Program Counter (PC)
- Floating point registers (3.14169)
- Flag Register (Zero, Signed, Carry, etc)
- Segment Registers (for memory stuff)
- AVX/SSE - xmm/ymm/zmm - 128/256/512 bits

See suggested reading list, Item [1]

A Quick Intro to Assembly Languages

Registers

- You can access subsets of the bits for many of them:
- RAX (all 64 bits) -> EAX (lower 32 bits) -> AX (lower 16 bits) -> AH (upper 8 bits of AX) -> AL (lower 8 bits of AX)



A Quick Intro to Assembly Languages

ADD RAX, RCX

- Each line is one Instruction.
- Each line begins with an Operation. In English grammar, a Verb.
- Parameters are typically called operands
  - 48 01 c8 is the machine code. ADD is often called the instruction mnemonic
- The first parameter is the Destination, storing the result.
- Instructions are grouped into Functions.
  - A function begins with the parameters on the Stack or in Registers.
  - A function ends with a standard instruction. (RET or BX LR.)

A Quick Intro to Assembly Languages

ADD RAX, RCX

- This instruction is x86_64:
  - This specific syntax view of the instruction is called (Intel). There are other ways to represent it, such as Gas/AT&T syntax used by GNU tools (GCC)
  - Destination register comes first (with AT&T, it’s the opposite! Why, oh why?)
  - The second register is one of the inputs.
  - The operation is ADDition.
- So what does this do?

A Quick Intro to Assembly Languages

Common Operations

- Operations are unique to each assembly language, but some are common.
    - MOV, ADD, SUB, MUL -- Arithmetic
    - CALL, BL -- Function Calls
    - RET, BLR -- Function Returns
    - PUSH, POP -- Grow or shrink the Stack.
- A table can be handy for each new assembly language.
- Learn the common instructions, look up the rest.

A Quick Intro to Assembly Languages

Stack

- You are probably familiar with the stack data type: Last in First Out (LIFO)
  - As opposed to Queue: First in First out (FIFO)
- The stack "grows down" from higher addresses to lower addresses
- Used to store local variables that were "statically allocated" at compile time
  - We say statically allocated because the size doesn't change when the program runs
- On x86_64 the stack is pointed to by RSP. It is an implicit operand in many instructions.

A Quick Intro to Assembly Languages

PUSH

- Used to store data on the stack
- PUSH RAX
- Effectively
  - SUB RSP, 8
  - MOV [RSP], RAX
    - (* Note that “[ ]” denotes a dereference. Like var[8] = ## in C *)

A Quick Intro to Assembly Languages

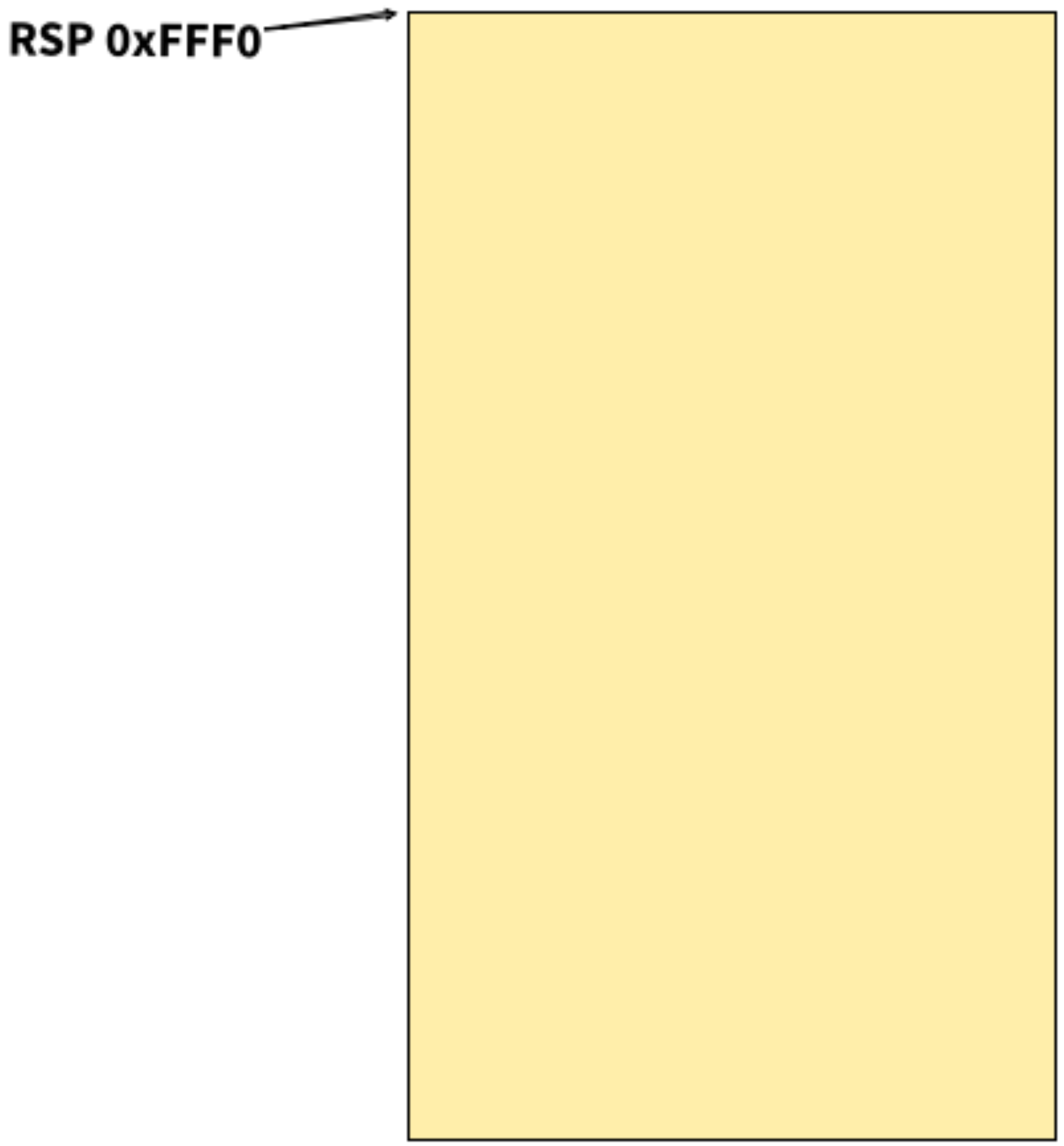
POP

- Used to take data from the stack and store it
- POP RAX
- Effectively
  - MOV RAX, [RSP]
  - ADD RSP, 8

A Quick Intro to Assembly Languages

PUSH/POP - Example

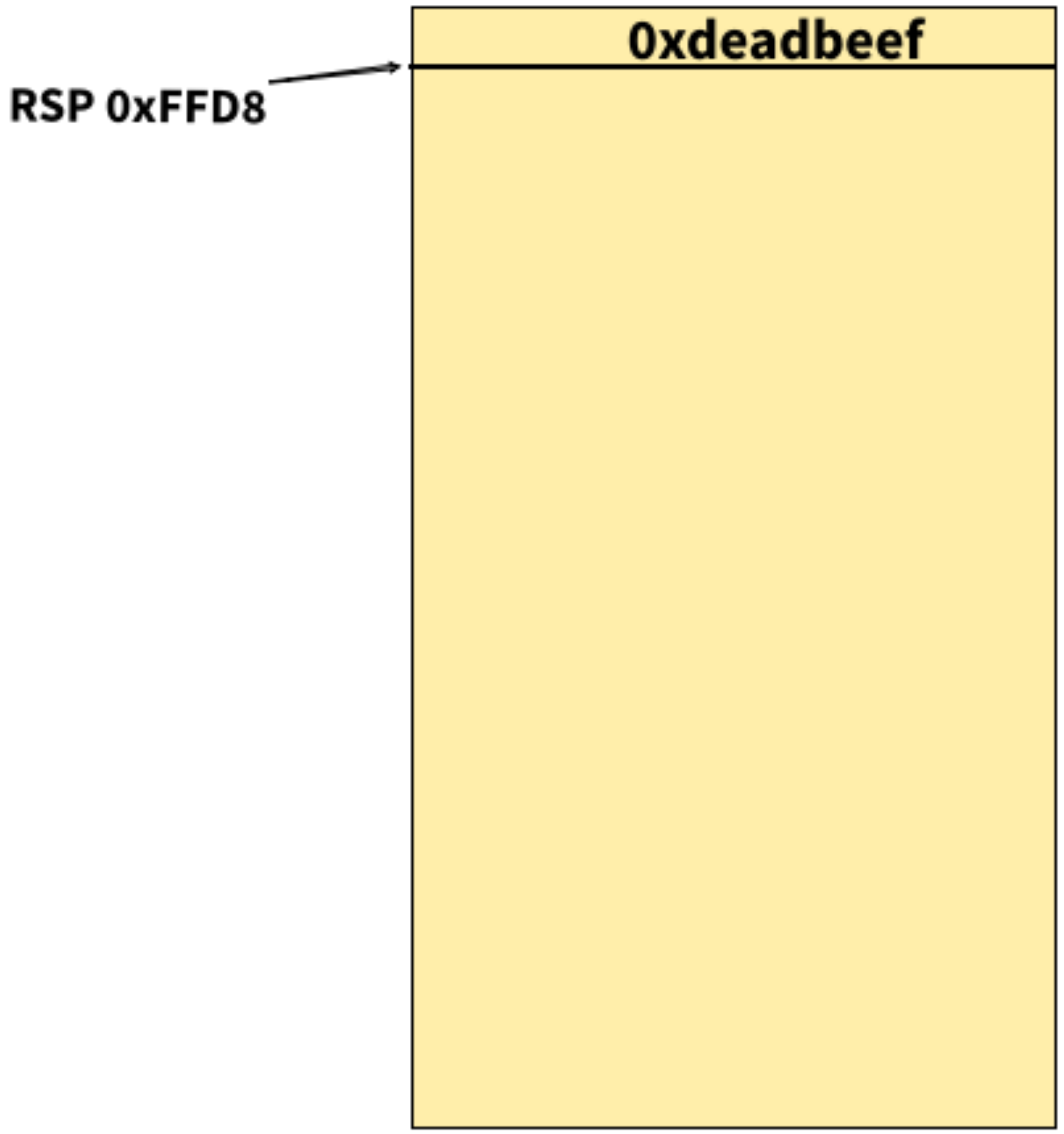
- RSP := 0xFFF0
- RAX := 0xdeadbeef
- RCX := 0xd00dd00d



A Quick Intro to Assembly Languages

PUSH/POP - Example

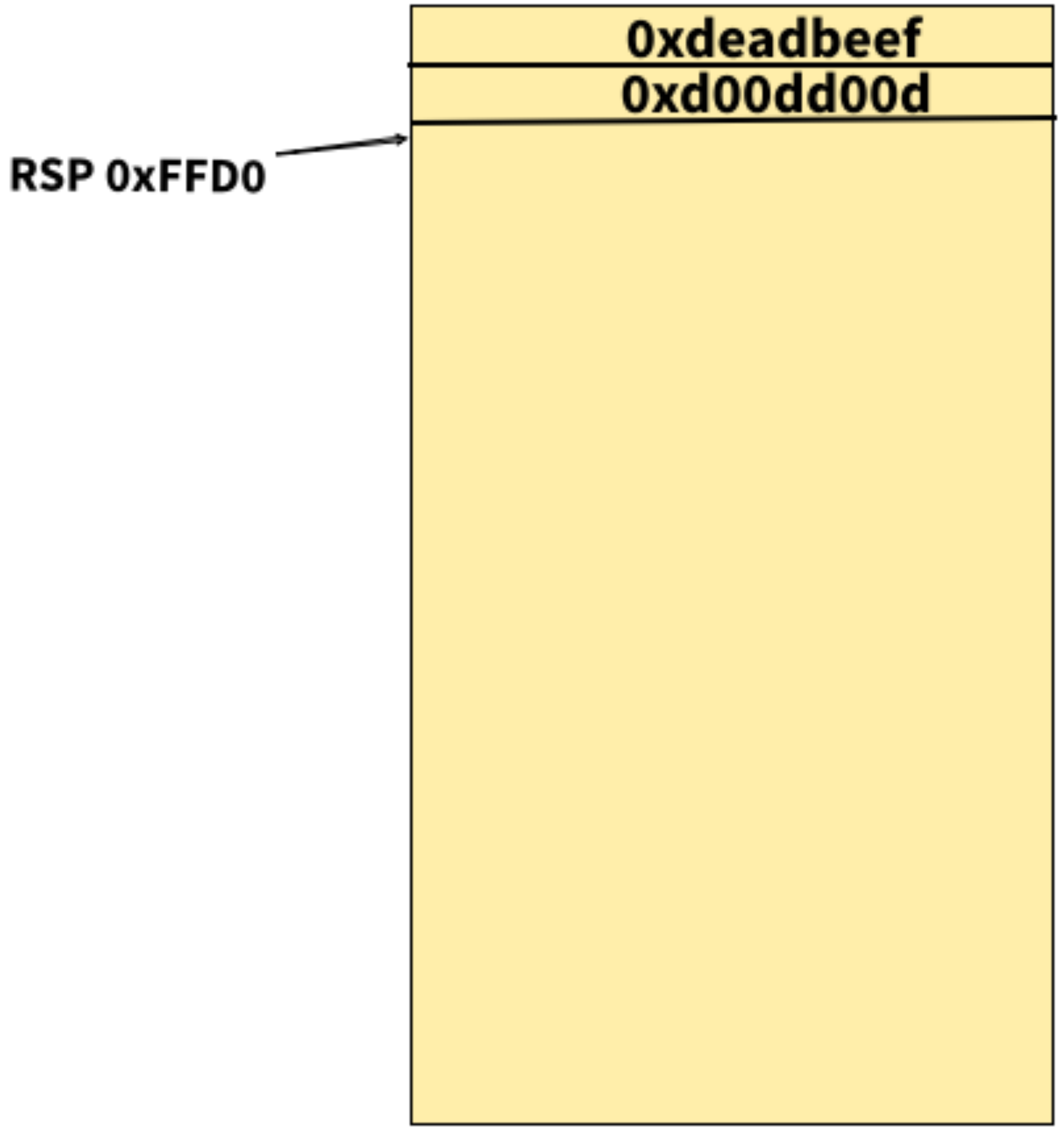
- PUSH RAX



A Quick Intro to Assembly Languages

PUSH/POP - Example

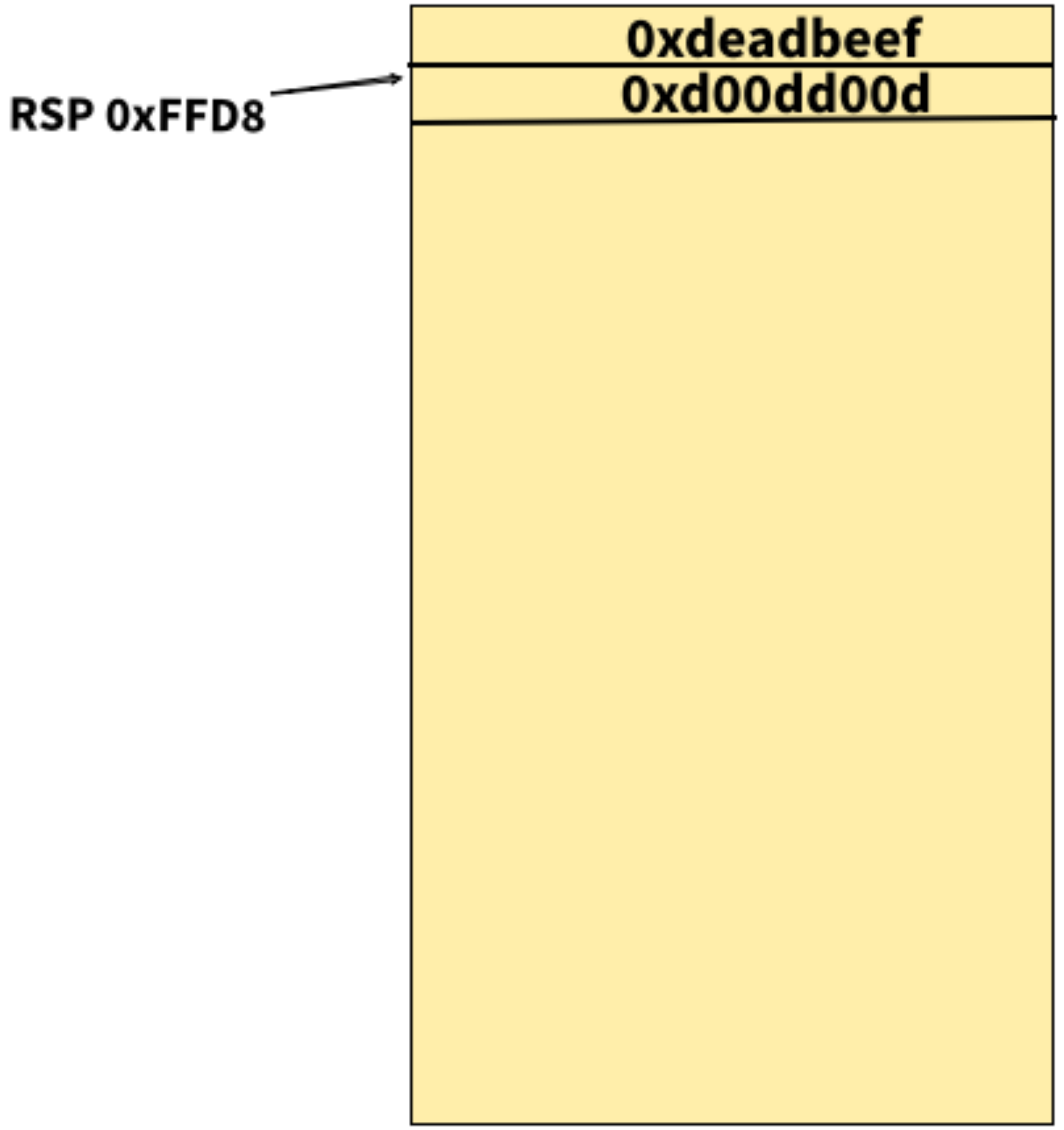
- PUSH RCX



A Quick Intro to Assembly Languages

PUSH/POP - Example

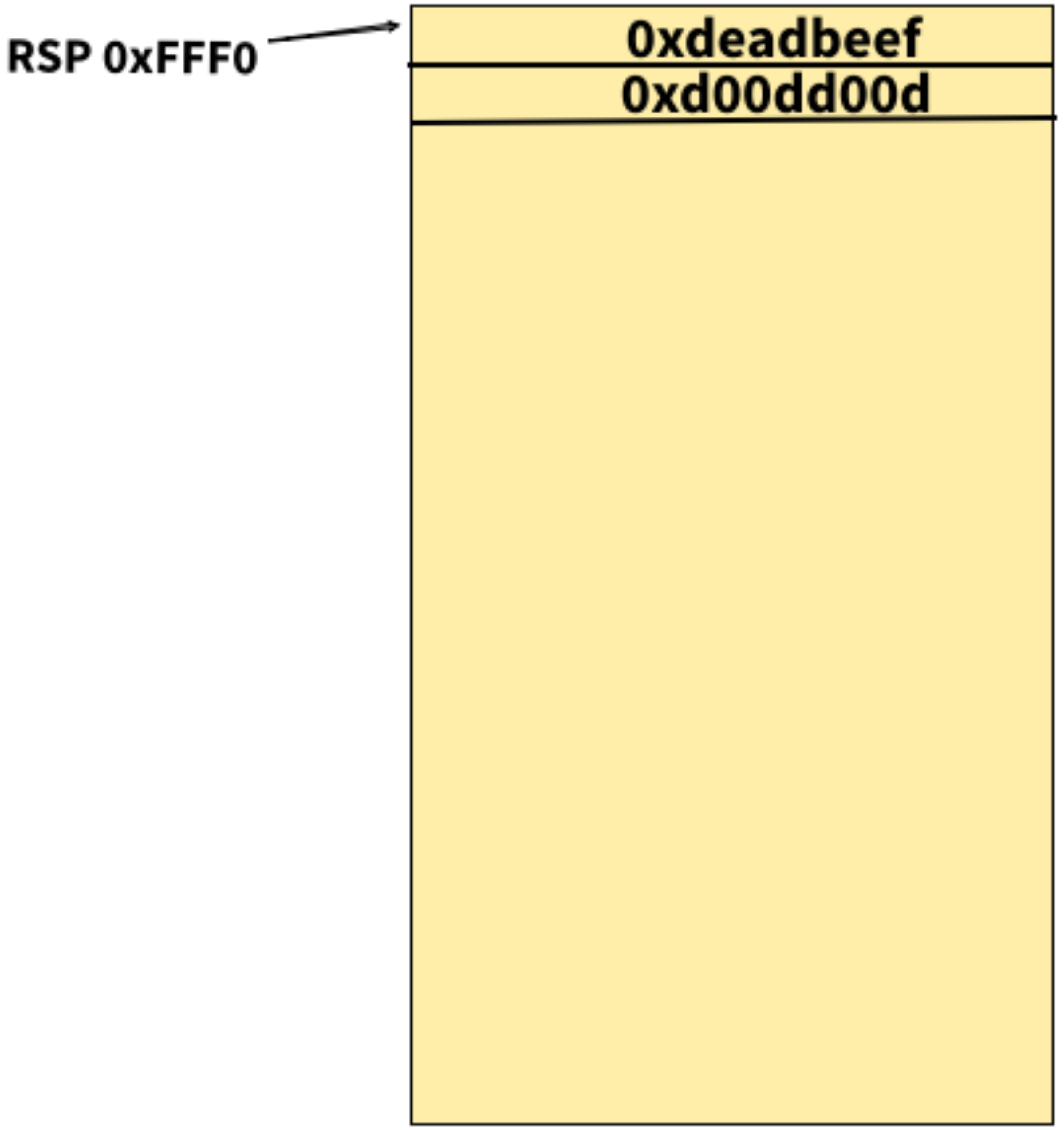
- POP RAX



A Quick Intro to Assembly Languages

PUSH/POP - Example

- POP RCX



A Quick Intro to Assembly Languages

PUSH/POP - Example

- RSP := 0x???
- RAX := 0x???
- RCX := 0x???

A Quick Intro to Assembly Languages

PUSH/POP - Example

- RSP := 0xFFF0
- RAX := 0xd00dd00d
- RCX := 0xdeadbeef

A Quick Intro to Assembly Languages

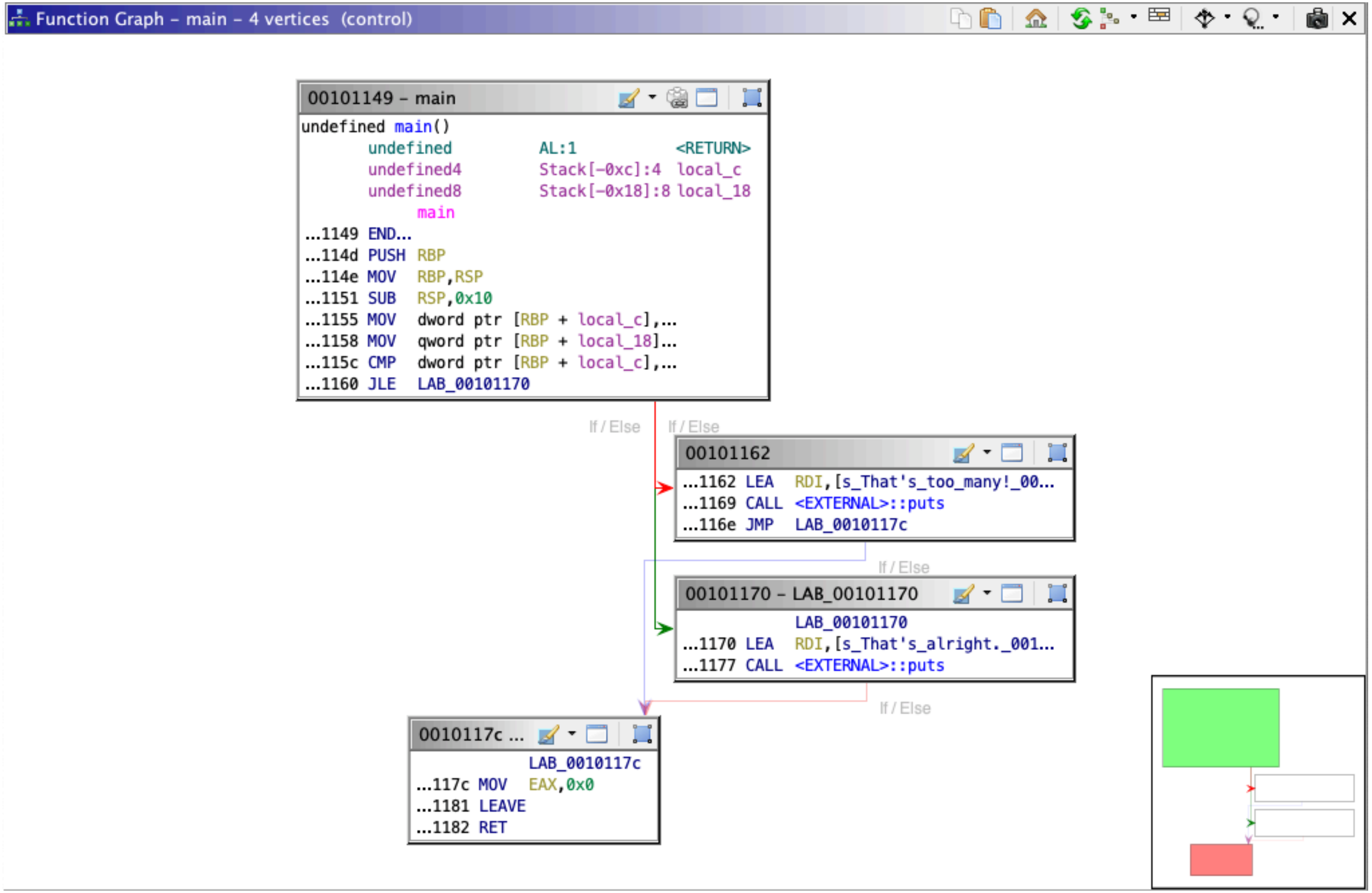
Control Flow

```c
#include <stdio.h>

int main(int argc, char **argv){
  if(argc>5)
    printf("That's too many!\n");
  else
    printf("That's alright.\n");
}
```

- A little C becomes a lot of Assembly.
- Sometimes it helps to think in a Graph.

A Quick Intro to Assembly Languages

Think of Control Flow as a Graph



A Quick Intro to Assembly Languages

CMP RAX, RCX

- You can think of CMP as a signed subtraction
  - Signed uses the most significant bit (MSB) to indicate the sign. 1 == negative; 0 == positive
  - Uses twos complement
  - 8-bit char: 0xFF = -1; 0x01 = 1
- Effectively
  - temp = RAX - RCX
- The value of temp is used to set the fields of the flag register. For example
- If temp == 0 then ZF = 1 (TRUE) else ZF = 0 (FALSE)
- If temp < 0 then SF = 1 else SF = 0
- There are other flags that may or may not be set depending on the instruction

A Quick Intro to Assembly Languages

JMP & JXX

- JMP is an unconditional branching statement. It jumps where you tell it without checking any of the condition flags.
- JXX - This is a family of instructions that jump if something is true. For example,
  - JZ 0x1234 will jump to 0x1234 if the Zero Flag is set to 1. This would happen if the operands of the last comparison were equal.
  - JNZ 0x1234 is the opposite, it will jump if not zero (ZF is set to 0)
- Different jump instructions check different flags but there are some that are equivalent.
  - JE is the same as JZ. If two operands are equal i.e. RAX and RCX then RAX - RCX == 0.
- Lots of different types
  - JNE, JE, JG, JGE, JL, JLE, etc

A Quick Intro to Assembly Languages

JMP & JXX

- Mentioned already but jumps are branching statements. i.e. they cause control flow to be non-linear

- For conditional branches there is a TRUE and a FALSE branch.

- Technically the FALSE branch just “falls through” (executes the next instruction

- The TRUE branch is taken if the check evaluates to TRUE

  - i.e. JZ evaluates the expression (ZF == 0). If ZF is equal to 0 then (ZF == 0) == 1

A Quick Intro to Assembly Languages

JXX example. Branch taken?

```
        mov     rax, 0x5
        mov     rcx, 0x10
        cmp     rax, rcx
        jne     _end

_end
```

A Quick Intro to Assembly Languages

JXX example. Branch taken?

```
        mov     rax, 0x5
        mov     rcx, 0x10
        cmp     rax, rcx
        je      _end

_end
```

A Quick Intro to Assembly Languages

JXX example. Branch taken?

```
        mov     rax, 0x5
        mov     rcx, 0x10
        cmp     rax, rcx
        jl      _end

_end
```

A Quick Intro to Assembly Languages

CALL & RET

- What about functions?
- Just like in C programming we want to organise code so that it can be reused
- How to we get there and back again though?
- CALL 0x1234 (Typically a 5 byte instruction on x86_64)
- Effectively:
  - PUSH RIP + 5
  - JMP 0x1234

A Quick Intro to Assembly Languages

CALL & RET

- Functions often take arguments
- x86 used the stack but x86_64 uses registers (mostly)
- Different **calling conventions** use different registers but for now we will focus on System V AMD64 **ABI** (used by the Linux-based Operating Systems)
    - RDI, RSI, RDX, RCX, R8, R10 (more than 6 uses the stack)
    - See: https://en.wikipedia.org/wiki/X86_calling_conventions
- The return value is stored in RAX

See suggested reading list, Item [2] re ABIs

A Quick Intro to Assembly Languages

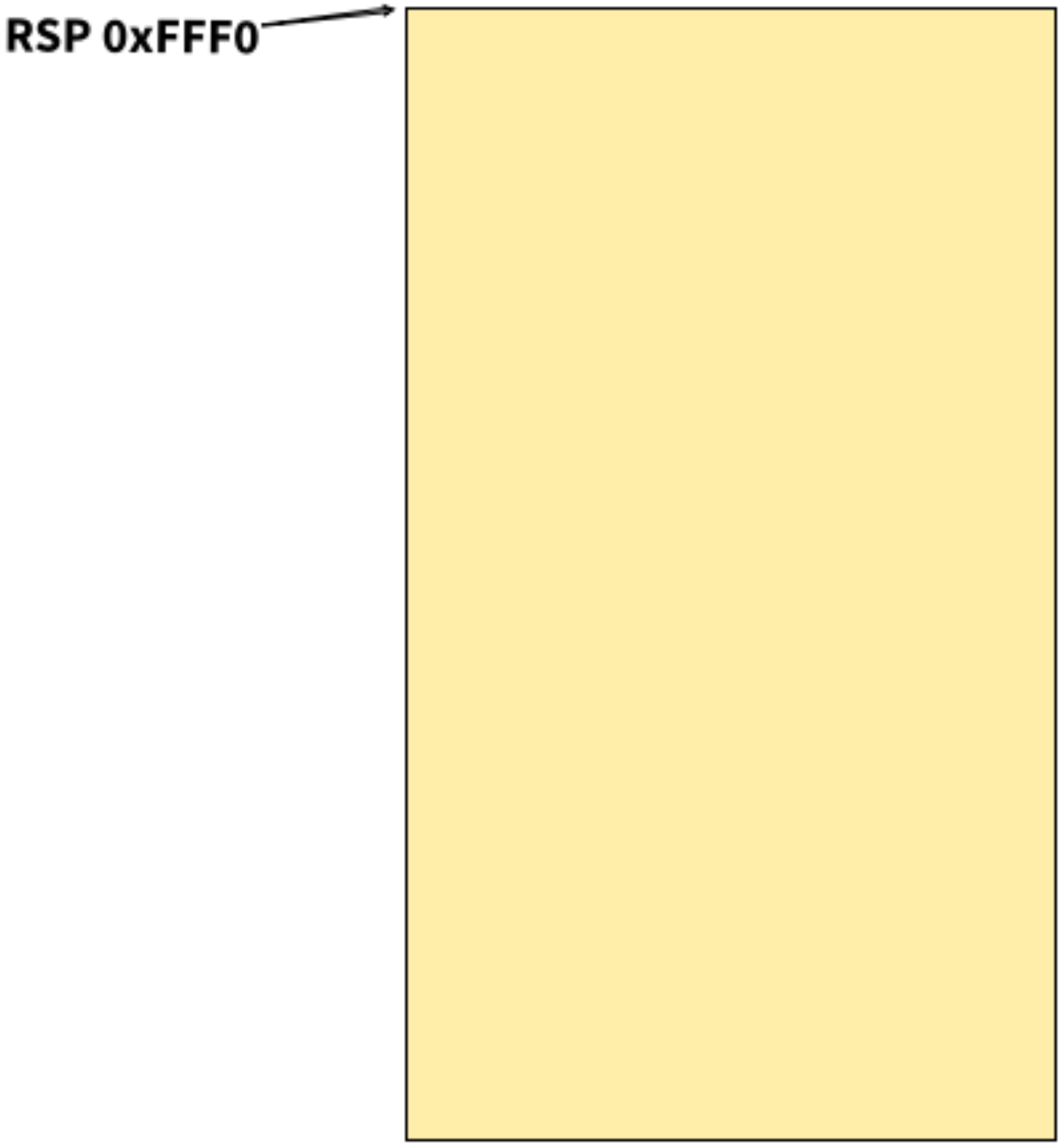
CALL & RET

- How do we get back
- RET (Return)
- Effectively:
    - POP RIP

A Quick Intro to Assembly Languages

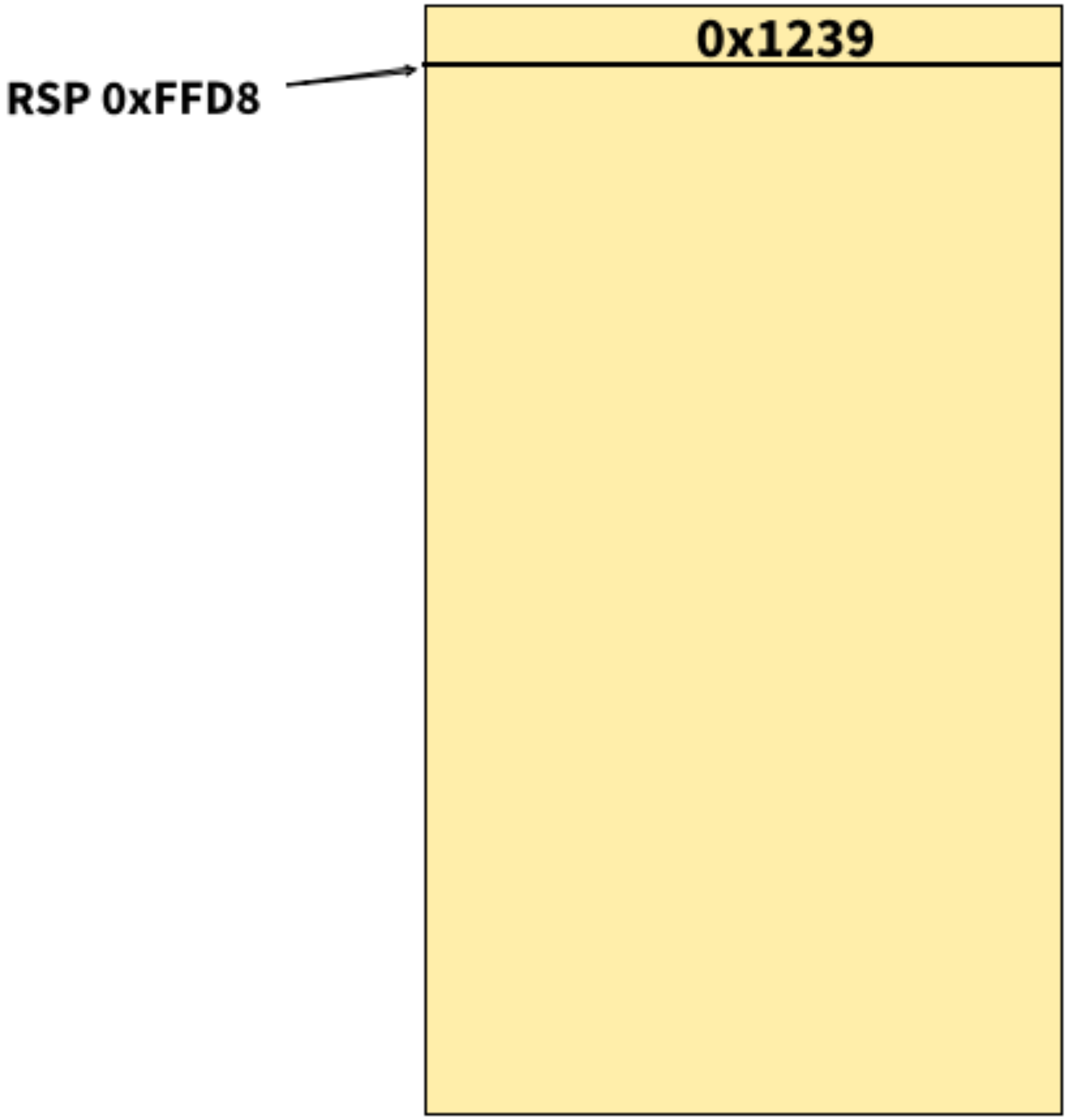
CALL / RET Example

- RIP := 0x1234
- RSP := 0xFFF0



A Quick Intro to Assembly Languages

CALL / RET Example

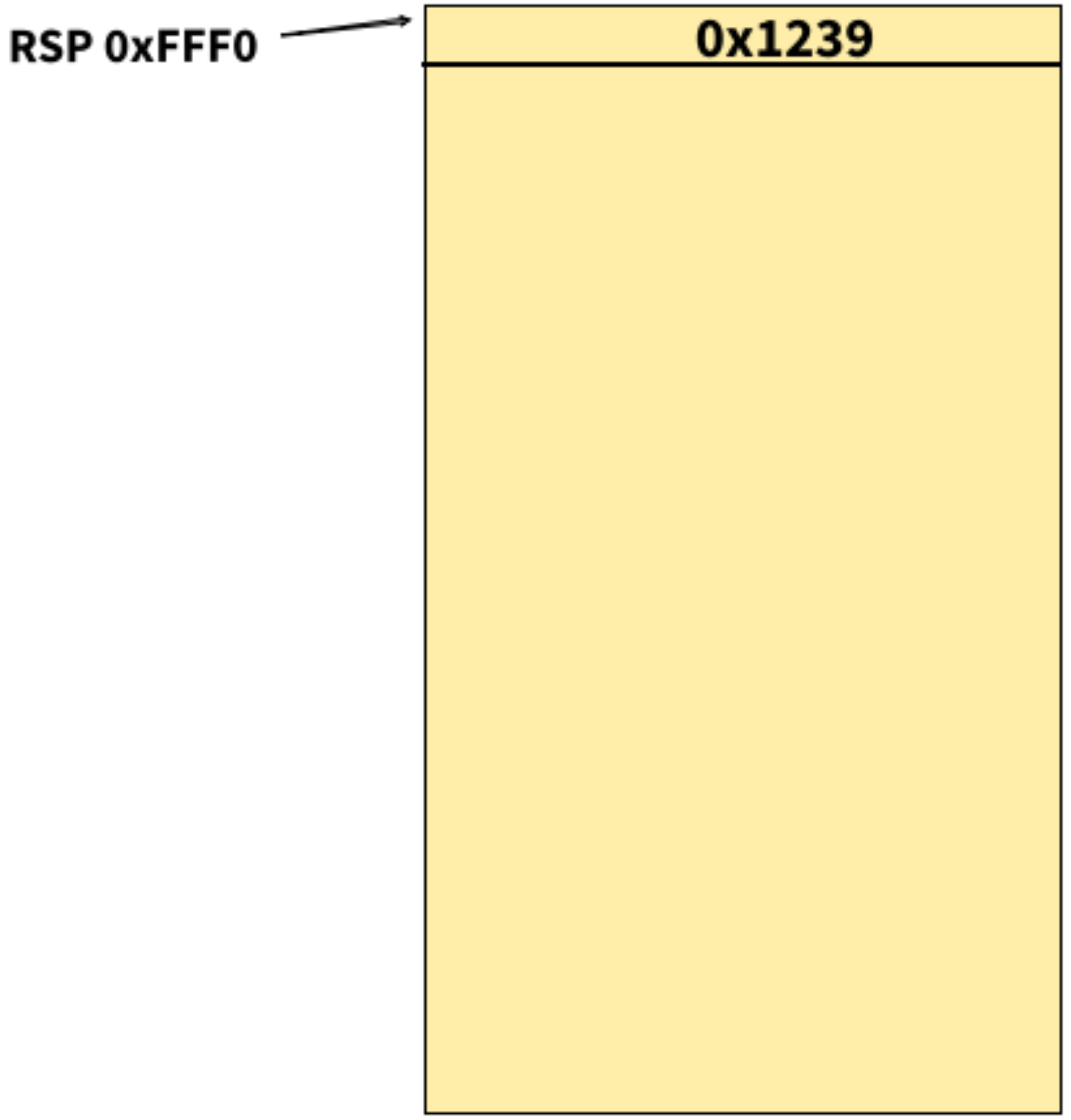


- CALL 0x4320
  - RSP := 0xFFD8
  - RIP := 0x4320

A Quick Intro to Assembly Languages

CALL / RET Example

- RET
  - RSP := 0xFFF0
  - RIP := 0x1239



A Quick Intro to Assembly Languages

Final

- This was a firehose of information
- This isn’t an assembly programming course
- There are a lot of instructions that you will have to look up on your own
- The INTEL instruction manual is your friend: 2A-2D
    - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Reversing a Simple Function

- Let's work out a few examples.
- First we'll see the assembly, and then we'll work backward to C.

Do this homework!

Reversing a Simple Function

An Example

```
imul    rdi, rdi
mov     rax, rdi
ret
```

Reversing a Simple Function

An Example

```
imul    rdi, rdi
mov     rax, rdi
ret
```

- Register rdi is the first parameter
- "imul rdi, rdi" MULtiplies RDI by itself and stores the result in RDI.
- "mov rax, rdi" MOVes the value of RDI into rax.
- "ret" is the standard return function on x86_64.

Reversing a Simple Function

An Example

```
imul    rdi, rdi
mov     rax, rdi
ret
```

↓

```
int square(int num) {
    return num * num;
}
```

Reversing a Simple Function

Another Example

```
0x0000          mov     rax, rdi
0x0003          cmp     rax, rsi
0x0006          jle     0x000b
0x0008          mov     rax, rsi
0x000b          ret
```

Reversing a Simple Function

Another Example

```
0x0000          mov     rax, rdi
0x0003          cmp     rax, rsi
0x0006          jle     0x000b
0x0008          mov     rax, rsi
0x000b          ret
```

- Register rdi is the first parameter, rsi is the second
- “mov, rax, rdi” sets rax equal to rdi. This is an optimisation to save a branch.
- "cmp rax, rsi” Compares rax (which was rdi) to rsi
- “jle” Jumps if rax is less than or equal to rsi
- “mov, rax, rsi” sets rax equal to rsi.
- "ret" is the standard return function on x86_64.

Reversing a Simple Function

Another Example

```
0x0000          mov     rax, rdi
0x0003          cmp     rax, rsi
0x0006          jle     0x000b
0x0008          mov     rax, rsi
0x000b          ret
```

↓

```
int min(int a, int b) {
        if ( a <= b ) {
          return a;
        } else {
          return b;
        }
}
```

Useful Tools for This Course

- GNU Objdump -- Command-line disassembler for many architectures.
- ghidra-sre.org -- GHIDRA, the NSA's reverse engineering tool.
- radare.org -- A free, command-line reverse engineering toolkit.
- godbolt.org -- Compiler Explorer, a tool to view assembly from snippets of C.
- GDB, LLDB -- Debuggers are very handy for exploring samples.
- Pen and Paper! -- Reverse engineering is puzzle solving.

Disassembling a Binary with Objdump

```
dell% objdump -d first

first:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
    1000: 48 83 ec 08          sub    $0x8,%rsp
    1004: 48 8b 05 dd 2f 00 00 mov    0x2fdd(%rip),%rax        # 3fe8 <__gmon_start__@Base>
    100b: 48 85 c0             test   %rax,%rax
    100e: 74 02                je     1012 <_init+0x12>
    1010: ff d0                call   *%rax
    1012: 48 83 c4 08          add    $0x8,%rsp
    1016: c3                   ret
```

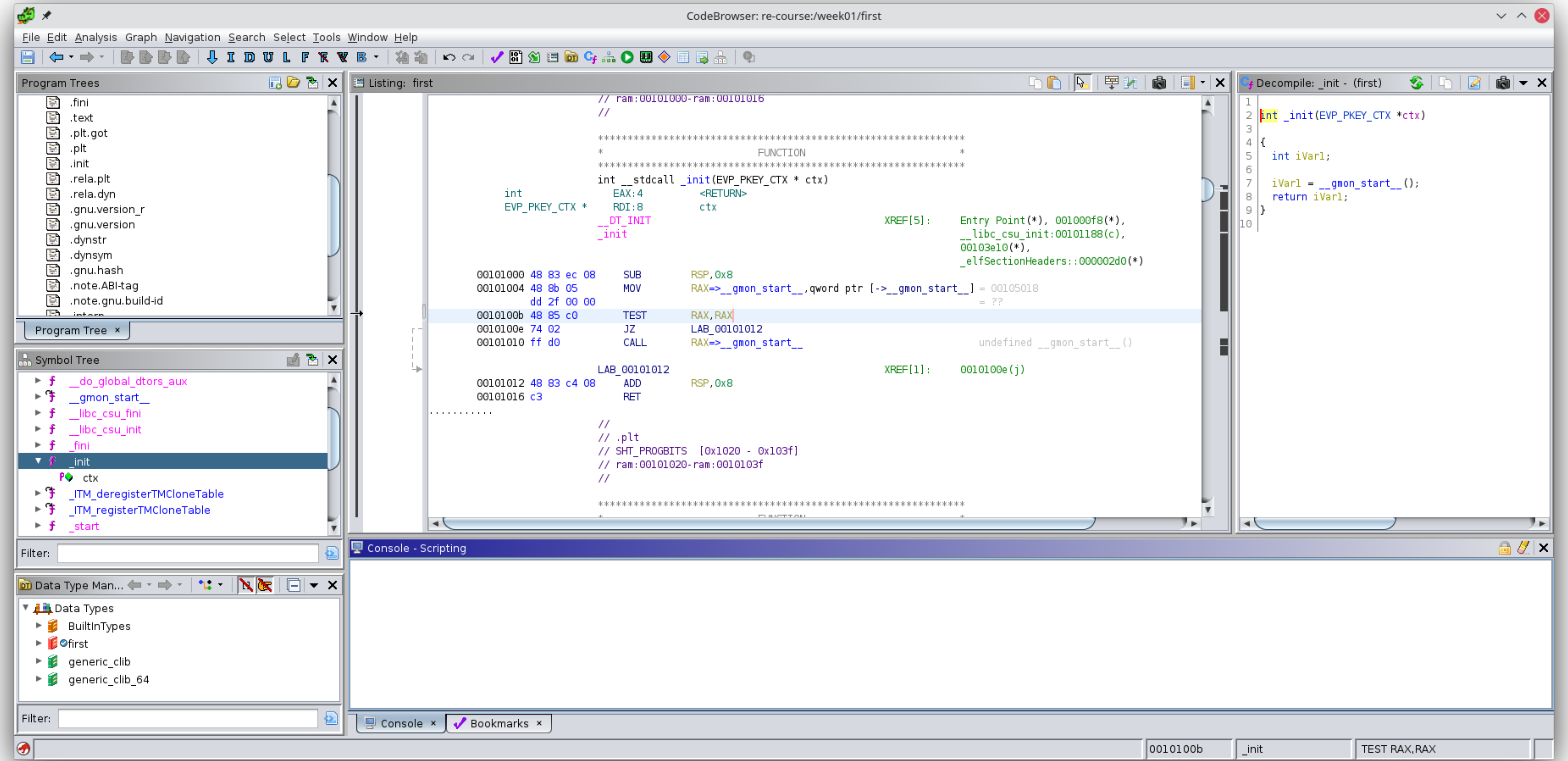
Disassembling with GDB

```
dell% gdb first
Reading symbols from first...
(No debugging symbols found in first)
(gdb) disassemble _init
Dump of assembler code for function _init:
   0x0000000000001000 <+0>:sub    $0x8,%rsp
   0x0000000000001004 <+4>:mov    0x2fdd(%rip),%rax        # 0x3fe8
   0x000000000000100b <+11>:test   %rax,%rax
   0x000000000000100e <+14>:je     0x1012 <_init+18>
   0x0000000000001010 <+16>:call   *%rax
   0x0000000000001012 <+18>:add    $0x8,%rsp
   0x0000000000001016 <+22>:ret
End of assembler dump.
(gdb)
```

Disassembling with Radare2

```
dell% r2 first
[0x00001050]> pd 7 @0x1000
            ;-- section..init:
            ;-- segment.LOAD1:
            ;-- _init:
            0x00001000      4883ec08       sub rsp, 8
            0x00001004      488b05dd2f00.  mov rax, qword [reloc.__gmon_start] ; [0x3fe8:8]=0
            0x0000100b      4885c0         test rax, rax
        ┌─< 0x0000100e      7402           je 0x1012
        │   0x00001010      ffd0           call rax
        └─> 0x00001012      4883c408       add rsp, 8
            0x00001016      c3             ret
[0x00001050]>
```

Decompiling with GHIDRA



Day 1 Homework

- From disassembly to pseudocode:
  - Take the listing you are given and write the C-like pseudocode
  - What will the function return given a specific value?
- For next class, have **Ghidra** installed and make sure that you can ssh into the Babylon servers from your laptop
  - https://kb.thayer.dartmouth.edu/article/361-linux-services
  - For login with Kerberos tokens rather than endlessly retyping your password: https://hackmd.io/e5Ft3DXCRze6NGnOudCt4Q
  - Also, enable GSSAPI in your .ssh/config for passwordless SCP to work: https://services.dartmouth.edu/TDClient/1806/Portal/KB/ArticleDet?ID=89203