Assembly

Muchang Bahng

Fall 2024

Contents

One final note to mention, there are many assembly languages out there and various syntaxes.

Example 0.1 (Assembly Syntax)

The two most popular syntaxes are AT&T and Intel.

1. **Intel Syntax**: Specifies memory operands without any special prefixes. Square brackets [] are used to denote memory addresses. For example, mov eax, [ebx] means move the contents of the memory location pointed to by ebx into eax.
2. **AT&T Syntax**: Memory operands are denoted with parentheses () and include the % prefix for registers. An instruction moving data from a memory location into a register might look like movl (%ebx), %eax, with additional prefixes for immediate values and segment overrides.

Example 0.2 (Assembly Languages)

The various assembly languages are as follows:

1. **x86 Assembly** : The assembly language for Intel and AMD processors using the x86 architecture. Both AT&T and Intel syntax are available. Tools or environments often allow switching between the two, with AT&T being the default in GNU tools like GDB.
2. **ARM Assembly** : The assembly language for ARM processors. Has its own unique syntax, not categorized as AT&T or Intel. ARM syntax is closely tied to its instruction set architecture and is distinct from the x86 conventions.
3. **MIPS Assembly** : The assembly language for MIPS processors. MIPS uses its own assembly language syntax, which is neither AT&T nor Intel. MIPS syntax is designed around the MIPS instruction set architecture.
4. **PowerPC Assembly** : The assembly language for PowerPC processors. PowerPC has its own syntax style, tailored to its architecture and instruction set, distinct from the AT&T and Intel syntax models.
5. **6502 Assembly** : Used in many early microcomputers and gaming consoles. Utilizes a syntax unique to the 6502 processor, not following AT&T or Intel conventions.
6. **AVR Assembly** : The assembly language for Atmel's AVR microcontrollers. AVR assembly follows its own syntax style, designed specifically for AVR microcontrollers and not based on AT&T or Intel syntax.
7. **Z80 Assembly** : Associated with the Z80 microprocessor, used in numerous computing devices in the late 20th century. Z80 assembly language has its own syntax that does not adhere to AT&T or Intel syntax guidelines.

The most common one is the x86_64, which is the one that we will be focusing on, with the AT%T syntax.

Just like how memory addressing is different between 32 and 64 bit machines, CPUs also use these schemes. While 32-bit processors have 2^{32} possible addresses in their cache, it turns out that 64-bit processors have a 48-address space. This is because CPU manufacturers took a shortcut. They use an instruction set which allows a full 64-bit address space, but current CPUs just only use the last 48-bits. The alternative was wasting transistors on handling a bigger address space which wasn't going to be needed for many years (since 48-bits is about 256TB). Just a bit of history for you. Finally, just to briefly mention, the input/output device, as the name suggests, processes inputs and displays outputs, which is how you can see what the program does.

Example 0.3 (x86 Architecture)

The x86 architecture is a CISC architecture, which is the most common architecture for personal computers. Here are important properties:

1. It is a complex instruction set computer (CISC) architecture, which means that it has a large set of complex instructions[a].
2. Byte-addressing is enabled and words are stored in little-endian format.
3. In the x86_64 architecture, registers are 8 bytes long (and 4 bytes in x86_32) and there are 16

total general purpose registers, for a total of only 128 bytes (very small compared to many GB of memory). Other special purpose registers are also documented in the wikipedia page, but it is not fully documented.

Example 0.4 (ARM Archiecture)

Mainly in phones, tablets, laptops.

Example 0.5 (MIPS Architecture)

MIPS is a RISC architecture, which is used in embedded systems such as digital home and networking equipment.

Definition 0.1 (Input/Output Device)

The input device can read/load/write/store data from the outside world. The output device, which has **direct memory address**, can display data to the outside world.

1 Assembly in x86_64

Definition 1.1 (Instruction)

An instruction is a single line of assembly code. It consists of some instruction followed by its (one or more) operands. The instruction is a mnemonic for a machine language operation (e.g. `mov`, `add`, `sub`, `jmp`, etc.). The **size specifier** can be appended to this instruction mnemonic to specify the size of the operands.

1. **b** (byte) for 1 byte
2. **w** (word) for 2 bytes
3. **l** (long) for 4 bytes
4. **q** (quad word) for 8 bytes

Note that due to backwards compatibility, word means 2 bytes in instruction names. Furthermore, the maximum size is 8 bytes since that is the size of each register in x86_64. An operand can be of 3 types, determined by their **mode of access**:

1. **Immediate addressing** is denoted with a $ sign, e.g. a constant integer data `$1`.
2. **Register addressing** is denoted with a % sign with the following register name, e.g. `%rax`.
3. **Memory addressing** is denoted with the hexadecimal address in memory, e.g. `0x034AB`.

Like higher level programming languages, we can perform operations, do comparisons, and jump to different parts of the code. Instructions can be generally categorized into three types:

1. **Data Movement**: These instructions move data between memory and registers or between the registery and registery. Memory to memory transfer cannot be done with a single instruction.

```
%reg = Mem[address]     # load data from memory into register
Mem[address] = %reg     # store register data into memory
```

2. **Arithmetic Operation**: Perform arithmetic operation on register or memory data.

```
%reg = %reg + Mem[address]     # add memory data to register
%reg = %reg - Mem[address]     # subtract memory data from register
```

[a] https://en.wikipedia.org/wiki/X86_instruction_listings

```
%reg = %reg * Mem[address]     # multiply memory data to register
%reg = %reg / Mem[address]     # divide memory data from register
```

3. **Control Flow**: What instruction to execute next.

```
jmp label     # jump to label
je label      # jump to label if equal
jne label     # jump to label if not equal
jg label      # jump to label if greater
jl label      # jump to label if less
call label    # call a function
ret           # return from a function
```

Now unlike compiled languages, which are translated into machine code by a compiler, assembly code is translated into machine code through a two-step process. First, we **assemble** the assembly code into an **object file** by an **assembler**, and then we **link** the object file into an executable by a **linker**. Some common assemblers are **NASM** (Netwide Assembler) and **GAS/AS** (GNU Assembler), and common linkers are **ld** (GNU Linker) and **lld** (LLVM Linker), both installable with **sudo pacman -S nasm ld**.

Definition 1.2 (mov)

Let's talk about the `mov` instruction. A good diagram to see is the following:

Parantheses indicate that we are using a pointer dereference.

Definition 1.3 (int)

The `int` instruction is used to generate a software interrupt. It is often used to invoke a system call.

Definition 1.4 (ret)

The `ret` instruction is used to return from a function. It returns the value in the `%rax` register.

Example 1.1 (Swap Function)

In **gdb**, we may have a function that swaps two integers.

```
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

which is the assembly code for the following C code.

```
void swap(long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

2 Stuff

2.1 x86 Assembly Registers

The specific type of registers that are available to a CPU depends on the computer architecture, or more specifically, the ISA, but here is a list of common ones for the x86-64. We have `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, `%rsp`, `%r8`, `%r9`, `%r10`, `%r11`, `%r12`, `%r13`, `%r14`, `%r15`. Therefore, the x86-64 Intel CPU has a total of 16 registers for storing 64 bit data. However, it is important to know which registers are used for what.

Definition 2.1 (Parameter Registers)

Compilers typically store the first six parameters of a function in registers

$$\texttt{\%rdi}, \texttt{\%rsi}, \texttt{\%rdx}, \texttt{\%rcx}, \texttt{\%r8}, \texttt{\%r9}, \tag{1}$$

respectively.

Definition 2.2 (Return Register)

The return value of a function is stored in the

$$\texttt{\%rax} \tag{2}$$

register.

Definition 2.3 (Stack and Frame Pointers)

The `%rsp` register is the **stack pointer**, which points to the top of the stack. The `%rbp` register is the **frame pointer**, or **base pointer**, which points to the base of the current stack frame. In a typical function prologue, **%rbp** is set to the current stack pointer (**%rsp**) value, and then `%rsp` is adjusted to allocate space for the local variables of the function. This establishes a fixed point of reference (`%rbp`) for accessing those variables and parameters, even as the stack pointer (`%rbp`) moves.

Definition 2.4 (Instruction Pointer)

The `%rip` register is the **instruction pointer**, which points to the next instruction to be executed. Unlike all the registers that we have shown so far, programs cannot write directly to `%rip`.

Definition 2.5 (Notation for Accessing Lower Bytes of Registers)

Sometimes, we need a more fine grained control of these registers, and x86-64 provides a way to access the lower bits of the 64 bit registers. We can visualize them with the diagram below.

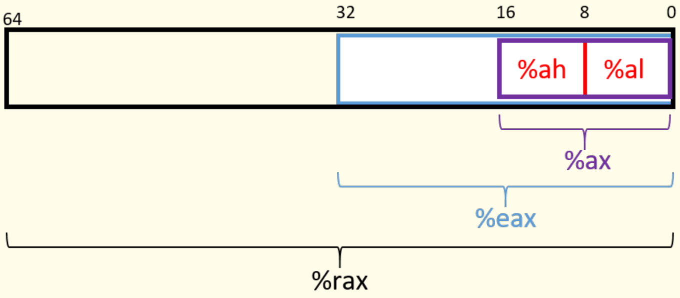


Figure 1: The names that refer to subsets of register `%rax`.

A complete list is shown below.

64-bit Register	32-bit Register	Lower 16 Bits	Lower 8 Bits
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rdi	%edi	%di	%dil
%rsi	%esi	%si	%sil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

Table 1: Register mapping in x86-64 architecture

2.2 x86 Addressing Modes

Example 2.1 (Immediate Addressing)

```
movq $0x4, %rax
```

Example 2.2 (Normal Addressing)

The following example shows the source operand being a memory address, with normal addressing, and the destination operand being a register.

```
movq (%rax), %rbx
```

Example 2.3 (Displacement Addressing)

The following example shows the source operand being a memory address and the destination operand being a register. They are both addressed normally.

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

Example 2.4 (Indexed Addressing)

The following shows the source operand being a memory address and the destination operand being a register. Say that `%rdx = 0xf000` and `%rcx = 0x0100`. Then

$$\texttt{0x80(,\%rdx,2) = Mem[2*0xF000 + 0x80] = Mem[0x1E080]} \tag{3}$$

We see that

```
movq 0x100(%rdi, %rsi, 8), %rdx
```

2.3 x86-64 Instructions

Let's talk about moving instructions first.

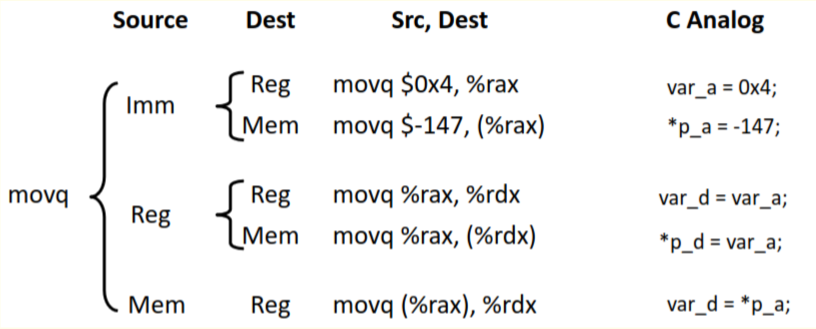
Definition 2.6 (mov)

Let's talk about the `mov` instruction which copies data from the source to the destination (the data in the source still remains!) and has the syntax

$$\texttt{mov\_ src, dest} \tag{4}$$

1. The source can be a register (`%rsi`), a value (`$0x4`), or a memory address (`0x4`).
2. The destination can be a register or a memory address.
3. The `_` is defined to be one of the size operands, which determine how big the data is. For example, we can call `movq` to move 8 bytes of data (which turns about to be the maximum size of a register).

A good diagram to see is the following:



	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	movq $0x4, %rax	var_a = 0x4;
		Mem	movq $-147, (%rax)	*p_a = -147;
	Reg	Reg	movq %rax, %rdx	var_d = var_a;
		Mem	movq %rax, (%rdx)	*p_d = var_a;
	Mem	Reg	movq (%rax), %rdx	var_d = *p_a;

Even with just the mov instruction, we can look at a practical implementation of a C program in Assembly.

Example 2.5 (Swap Function)

Let us take a look at a function that swaps two integers. Let's see what they do.

1. In C, we dereference both xp and yp (note that they are pointers to longs, so they store 8 bytes), and assign these two values to two temporary variables. Then, we assign the value of yp to xp and the value of xp to yp.
2. In Assembly, we first take the registers %rdi and %rsi, which are the 1st and 2nd arguments of the function, dereference them with the parantheses, and store them in the temporary registers %rax and %rdx. Then, we store the value of %rdx into the memory address of %rdi and the value of %rax into the memory address of %rsi. Note that the input values (the actual of)

```c
void swap(long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

Definition 2.7 (movz and movs)

The movz and movs instructions are used to move data from the source to the destination, but with zero and sign extension, respectively. It is used to copy from a smaller source value to a larger destination, with the syntax

```
movz__ src, dest
movs__ src, dest
```

where the first _ is the size of the source and the second _ is the size of the destination.

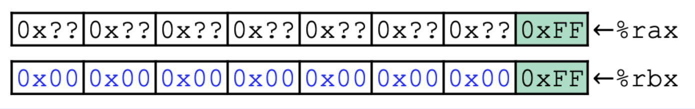
1. The source can be from a memory or register.
2. The destination must be a register.

Example 2.6 (Simple example with movz)

Take a look at the code below.

```
movzbq %al, %rbx
```

The %al represents the last byte of the %rax register. It is 1 byte long. The %rbx register is 8 bytes long, so we can fill in the rest of the 7 bytes with zeros.



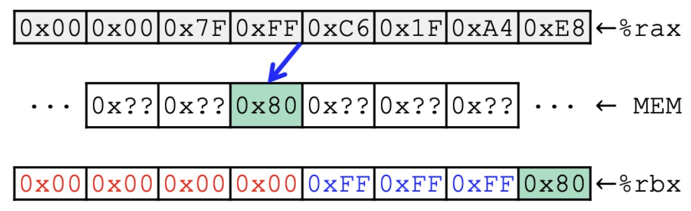
0x??	0x??	0x??	0x??	0x??	0x??	0x??	0xFF	←%rax
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0xFF	←%rbx

Example 2.7 (Harder example with movs)

Take a look at the code below.

```
movsbl (%rax), %ebx
```

You want to move the value at the memory address in %rax into %ebx. Since the source size is set to 1 byte, you take that byte, say it is 0x80, from the memory, and then sign extend it (by a size of 4 bytes!) into %ebx. Note that therefore, the first four bytes of %rbx will not be affected since it's not a part of %ebx. An exception to this is that in x86-64, any instruction that generates a 32-bit long word value for a register also sets the high-order 32 bits of the register to 0, so this ends up clearing the first 4 bytes to 0.



0x00 0x00 0x7F 0xFF 0xC6 0x1F 0xA4 0xE8 ←%rax

· · · 0x?? 0x?? 0x80 0x?? 0x?? 0x?? · · · ← MEM

0x00 0x00 0x00 0x00 0xFF 0xFF 0xFF 0x80 ←%rbx

Now we can talk about control transfer. Say that you have the following C and Assembly code.

```c
int add(int x) {
  return x + 2;
}

int main() {
  int a = 2;
  int b = add(a);
  return 0;
}
```

```
add:
  movq %rdi, %rax
  addq $2, %rax
  ret
main:
  movq $3, $rdi
  call add
  movq $0, %rax
  ret
```

Figure 2: A simple function.

If you go through the instructions, you see that in main, you first move $3 into the %rdi register. Then, you call the add function, and within it you also have the %rdi register. This is a conflict in the register, and we don't want to simply overwrite the value of %rdi in the main function. Simply putting it to another register isn't a great idea since we can't always guarantee that it will be free. Therefore, we must use the memory itself.

Recall the stack, which we can think of as a giant array in which data gets pushed and popped in a last-in-first-out manner. The stack is used to store data and return addresses, and is used to manage function calls. Visually, we want to think of the elements getting pushed in from the bottom (upside down) towards lower memory addresses.

Definition 2.8 (Stack Pointer)

Note that every time we want to push or pop something from the stack, we must know *where* to push or pop it. This is where the **stack pointer** comes in. It is a special register that always points to the top of the stack, and is used to keep track of the stack.

Definition 2.9 (Push and Pop)

The push and pop instructions are used to push and pop data onto and off the stack, respectively.

push_ src	rsp = rsp - 8; Mem[rsp] = src
pop_ dest	dest = Mem[rsp]; rsp = rsp + 8

1. When we push the source, we fetch the value at the source and store it at the memory address pointed to by the stack pointer %rsp. Then, we decrement %rsp by 8.
2. When we pop from the stack, we fetch the value at the memory address pointed to by the stack pointer %rsp and store it in the destination. Then, we increment %rsp by 8.

Note that no matter what the size of the operand, we always subtract 8 from the stack pointer. This is because the stack grows downwards, and we want to make sure that the next element is pushed into the next available space.

Note that the register `%rsp` is the stack pointer, which points to the top of the stack. The stack is used to store data and return addresses, and is used to manage function calls.

Definition 2.10 (Push and Pop)

The **push** and **pop** instructions are used to push and pop data onto and off the stack, respectively.

```
push_ src                 rsp = rsp - 8; Mem[rsp] = src
pop_ dest               dest = Mem[rsp]; rsp = rsp + 8
```

The _ is a size operand, which determines how big the data is.

Definition 2.11 (Call and Ret)

The `call` instruction pushes the return address onto the stack and jumps to the function. The `ret` instruction pops the return address from the stack and jumps to it.

We also talked about how there is instruction code that is even below the stack that is stored. This is where all the machine code/assembly is stored, and we want to find out where we are currently at in this code. This is done with the program counter.

Definition 2.12 (Program Counter, Instruction Pointer)

The **program counter**, or **instruction pointer**, is a special register **rip** that points to the current instruction in the program. It is used to keep track of the next instruction to be executed.

Let's go through one long example to see in detail how this is calculated.

Example 2.8 (Evaluating a Function)

Say that we have the following C code.

```c
int adder2(int a) {
  return a + 2;
}

int main() {
  int x = 40;
  x = adder2(x);
  printf("x is: %d\n", x);
  return 0;
}
```

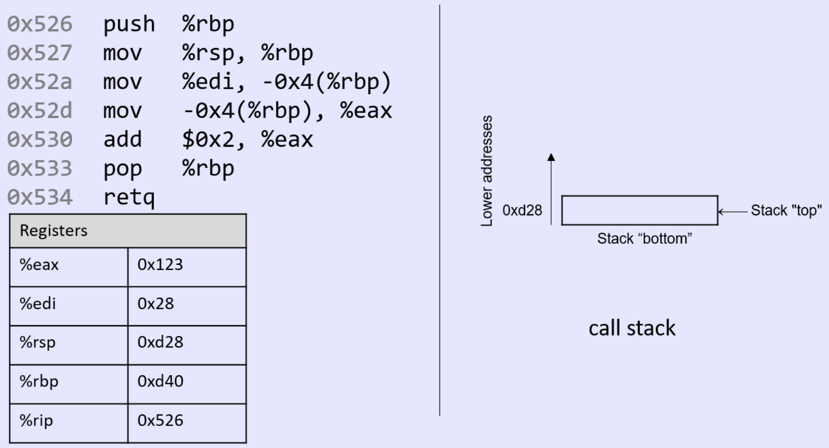
When we compile this program, we can view its full assembly code by calling `objdump -d a.out`. The output is quite long, so we will focus on the instruction for the `adder2` function.

```
0000000000400526 <adder2>:
400526:       55                          push   %rbp
400527:       48 89 e5                    mov    %rsp,%rbp
40052a:       89 7d fc                    mov    %edi,-0x4(%rbp)
40052d:       8b 45 fc                    mov    -0x4(%rbp),%eax
400530:       83 c0 02                    add    $0x2,%eax
400533:       5d                          pop    %rbp
400534:       c3                          retq
```

Figure 3: The output of objdump for the `adder2` function. The leftmost column represents the addresses (in hex) of where the actual instructions lie. The second column represents the machine code that is being executed. The third column represents the assembly code.

Note some things. Since `adder2` is taking in an integer input value, we want to load it into the lower 32 bits (4 bytes) of the `%rdi` register, which is the first parameter. So we use `%edi`. Likewise for the return value, we want to output an int so we use `%eax` rather than `%rax`. Let's go through some of the steps.

1. By the time we get into calling `adder2`, we can take a look at the relevant registers.

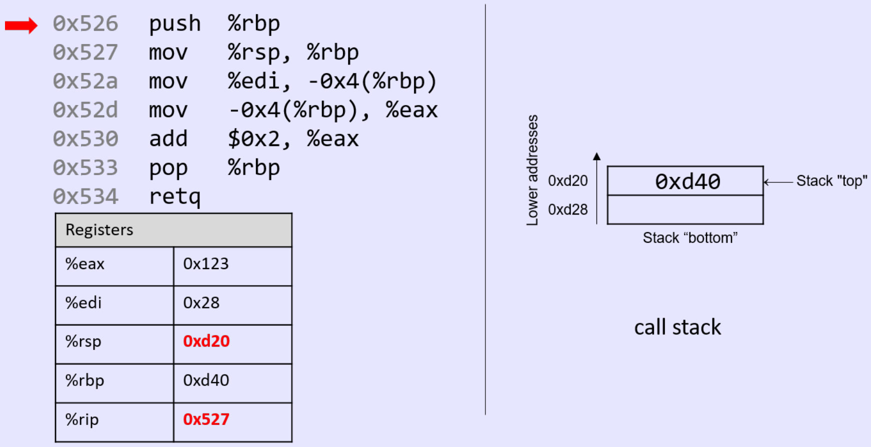


```
0x526  push  %rbp
0x527  mov   %rsp, %rbp
0x52a  mov   %edi, -0x4(%rbp)
0x52d  mov   -0x4(%rbp), %eax
0x530  add   $0x2, %eax
0x533  pop   %rbp
0x534  retq
```

Registers	
%eax	0x123
%edi	0x28
%rsp	0xd28
%rbp	0xd40
%rip	0x526

   (a) First, the `%eax` is filled with garbage, which are leftovers from previous programs that haven't been overwritten yet.
   (b) Second, the `%edi=0x28` since we have set `x=40` in `main`, before calling `adder2`, so it lingers on.
   (c) `%rsp=0xd28` since that is where the top of the stack is.
   (d) `%rbp=0xd40`
   (e) `%rip=0x526` since that is where we are currently at in our instruction (we are about to do it, but haven't done it yet).

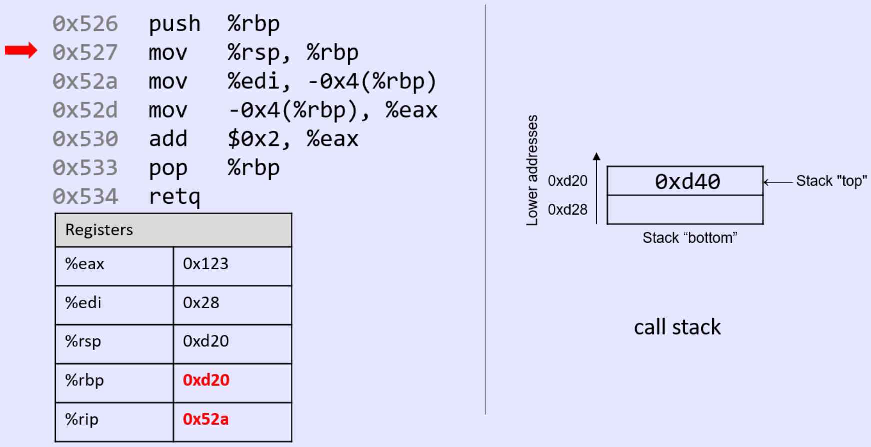
2. When we execute the first line of code, we simply push the value at `%rbp` into the stack. The top of the stack gets decremeneted by 8 and the value at `%rbp` is stored there. This means that the top of the stack is at `%rsp=0xd20` and the next instruction will be at `%rip=0x527`.



```
➡ 0x526  push  %rbp
  0x527  mov   %rsp, %rbp
  0x52a  mov   %edi, -0x4(%rbp)
  0x52d  mov   -0x4(%rbp), %eax
  0x530  add   $0x2, %eax
  0x533  pop   %rbp
  0x534  retq
```

Registers	
%eax	0x123
%edi	0x28
%rsp	**0xd20**
%rbp	0xd40
%rip	**0x527**

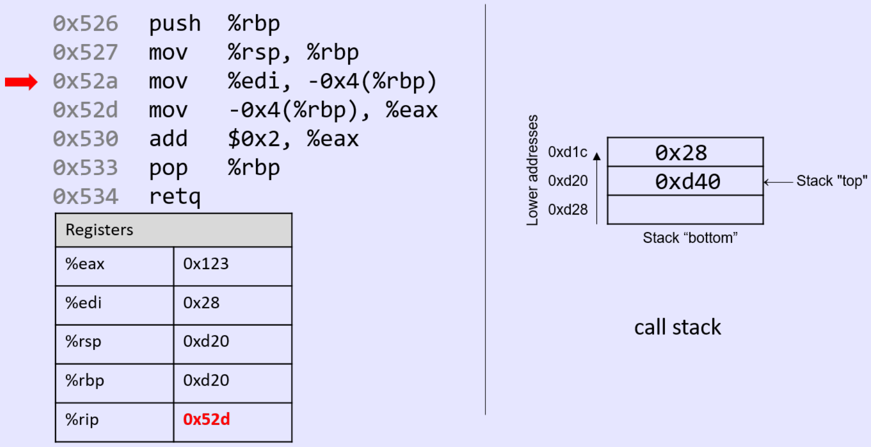
3. The reason we have pushed %rbp onto the stack is that we want to save it before it gets overwritten by this next execution. We basically move the value of %rsp into %rbp, and the %rip advances to the next instruction. %rip moves to the next instruction.



```
  0x526  push  %rbp
➡ 0x527  mov   %rsp, %rbp
  0x52a  mov   %edi, -0x4(%rbp)
  0x52d  mov   -0x4(%rbp), %eax
  0x530  add   $0x2, %eax
  0x533  pop   %rbp
  0x534  retq
```

Registers	
%eax	0x123
%edi	0x28
%rsp	0xd20
%rbp	**0xd20**
%rip	**0x52a**

4. Now we want to take our first argument %edi and store it in memory. Note that since this is 4 bytes, we can move this value into memory that is 4 bytes below the stack (-0x4(%rbp)). Note that the storing the value of %edi into memory doesn't affect the stack pointer %rsp. As far as the program is concerned, the top of this stack is still address 0xd20.



```
  0x526  push  %rbp
  0x527  mov   %rsp, %rbp
➡ 0x52a  mov   %edi, -0x4(%rbp)
  0x52d  mov   -0x4(%rbp), %eax
  0x530  add   $0x2, %eax
  0x533  pop   %rbp
  0x534  retq
```

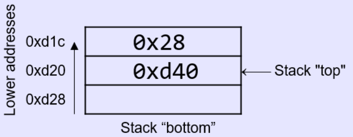
Registers	
%eax	0x123
%edi	0x28
%rsp	0xd20
%rbp	0xd20
%rip	**0x52d**

5. The next instruction simply goes into memory 4 bytes below the stack pointer, takes the value there, and stores it into %eax. This is the value of %edi that we just stored. This may seem redundant since we are making a round trip to memory and back to ultimately move the value

of `%edi` into `%eax`, but compilers are not smart and just follow these instructions.

```
   0x526  push  %rbp
   0x527  mov   %rsp, %rbp
   0x52a  mov   %edi, -0x4(%rbp)
➡ 0x52d  mov   -0x4(%rbp), %eax
   0x530  add   $0x2, %eax
   0x533  pop   %rbp
   0x534  retq
```

Registers	
%eax	0x28
%edi	0x28
%rsp	0xd20
%rbp	0xd20
%rip	0x530

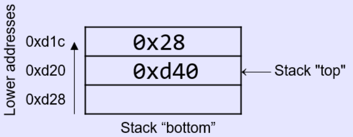


call stack

6. Finally, we add the value `$0x2` to `%eax` and store it back into `%eax`.

```
   0x526  push  %rbp
   0x527  mov   %rsp, %rbp
   0x52a  mov   %edi, -0x4(%rbp)
   0x52d  mov   -0x4(%rbp), %eax
➡ 0x530  add   $0x2, %eax
   0x533  pop   %rbp
   0x534  retq
```

Registers	
%eax	0x2A
%edi	0x28
%rsp	0xd20
%rbp	0xd20
%rip	0x533

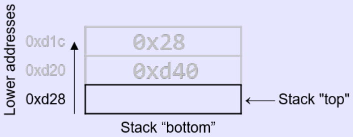


call stack

7. Finally, we pop the value at the top of the stack and store it into `%rbp`. Note that this is *not* the value `0x28`. It is simply the value that is stored at `%rsp=0xd20`, which is `(%rsp)=0xd40`.

```
   0x526  push  %rbp
   0x527  mov   %rsp, %rbp
   0x52a  mov   %edi, -0x4(%rbp)
   0x52d  mov   -0x4(%rbp), %eax
   0x530  add   $0x2, %eax
➡ 0x533  pop   %rbp
   0x534  retq
```

Registers	
%eax	0x2A
%edi	0x28
%rsp	0xd28
%rbp	0xd40
%rip	0x534



call stack

8. Finally, we return the value with `retq`.

Note that the final values in the registers `%rsp` and `%rip` are `0xd28` and `0x534`, respectively, which are the same values as when the function started executing! This is normal and expected behavior with the call stack, which just stores temporary variable sand data of each function as it executes a program. Once a function completes executing, the stack returns to the state it was in prior to the function call. Therefore, it is common to see the following two instructions at the beginning of a function:

```
push %rbp
mov %rsp, %rbp
```

and the following two at the end of a function

```
pop %rbp
retq
```

Now arithemtic operations are quite simple.

Definition 2.13 (Add, Subtract, Multiply)

The **add** and **sub** instructions are used to add and subtract data from the destination.

`add_ src, dest`	`dest = dest + src`
`sub_ src, dest`	`dest = dest - src`

The **imul** instruction is used to multiply data between the source and destination and store it in the destination.

`imul_ src, dest`	`dest = dest * src`

Again the _ is a size operand, which determines how big the data is.

Definition 2.14 (Increment, Decrement)

The **inc** and **dec** instructions are used to increment and decrement the value in the destination.

`inc_ dest`	`dest = dest + 1`
`dec_ dest`	`dest = dest - 1`

Definition 2.15 (Negative)

The **neg** instruction is used to negate the value in the destination.

`neg_ dest`	`dest = -dest`

Example 2.9 (Basic Arithmetic Function)

The following represents the same program in C and in assembly. Let's go through each one:

1. In C, we first initialize `a = 4`, then `b = 8`, add them together to get `c`, and then return `c`.
2. In Assembly, we move the value 4 to the `%rax` register, then move the value 8 to the `%rbx` register, add the two values together to store it into `%rax`, and then return the value in the `%rax` register.

```c
int main() {
  int a = 4, b = 8;
  int c = a + b;
  return c;
}
```

```
main:
  movq $4, %rax
  movq $8, %rbx
  addq %rbx, %rax
  ret
```

It is slightly different in Assembly since rather than storing 4 in some intermediate register, we immediately store it in the return register. In a way it is more optimized, and this is what the compiler does for you so that as few registers are used.

A shorthand way to do this is with `lea`, which stands for load effective address.

Definition 2.16 (Load Effective Address)

The **lea** instruction is used to load the effective address of the source into the destination. For now, we will focus on the arithmetic operations that it can do

```
              lea_ (src1, src2), dest                   dest = src1 + src2
       lea_ (src1, src2, scale), dest             dest = src1 + src2*scale
         lea_ const(src1, src2), dest           dest = src1 + src2 + const
  lea_ const(src1, src2, scale), dest     dest = src1 + src2*scale + const
```

This is useful for doing arithmetic operations on the address of a variable.

Definition 2.17 (Bitwise)

The **and**, **or**, **xor**, and **not** instructions are used to perform bitwise operations on the source and destination.

```
and src, dest                 dest = dest & src
 or src, dest                 dest = dest | src
xor src, dest                   dest = dest ŝrc
     neg dest                      dest = -dest
     not dest                      dest = ∼dest
```

Definition 2.18 (Arithmetic and Logical Bit Shift)

The `sal` arithmetic instruction is used to shift the bits of the destination to the left by the number of bits specified in the source. The `shr` instruction is used to shift the bits of the destination to the right by the number of bits specified in the source.

```
sal src, dest                 dest = dest « src
shr src, dest                 dest = dest » src
```

The `sar` instruction is used to shift the bits of the destination to the right by the number of bits specified in the source, and fill the leftmost bits with the sign bit. The `shl` instruction is used to shift the bits of the destination to the left by the number of bits specified in the source, and fill the rightmost bits with zeros.

```
sar src, dest                 dest = dest » src
shl src, dest                 dest = dest « src
```

Example 2.10 (Harder Arithmetic Example)

The following two codes are equivalent.

```c
long arith(long x, long y, long z) {
  long t1 = x + y;
  long t2 = z + t1;
  long t3 = x + 4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
.
.
.
.
.
```

```
arith:
  # rax/t1 = x + y
  leaq  (%rdi, %rsi), %rax
  # rax/t2 = z + t1
  addq  %rdx, %rax
  #rdx = 3 * y
  leaq  (%rsi, %rsi, 2), %rdx
  #rdx/t4 = (3*y) * 16
  salq  $4, %rdx
  #rcx/t5 = x + t4 + 4
  leaq  4(%rdi, %rdi), %rcx
  # rax/rval = t5 * t2
  imulq %rcx, %rax
  ret
```

The final thing in our list is condition codes.

Sometimes, we want to move (really copy) some value to another register if some condition is met. This is where we use conditional moves. These conditions are met by the flags register, which is a special register that stores the status of the last operation. It is the value of these flags that determine whether all future conditional statements are met in assembly.

Definition 2.19 (Condition Code Flags)

The flags register in the x86 CPU keeps 4 *condition code* flag bits internally. Think of these as status flags that are *implicitly* set by the most recent arithmetic operation (think of it as side effects). Note that condition codes are NOT set by `lea` or `mov` instructions!

1. **Zero Flag**: if the last operation resulted in a zero value.
2. **Sign Flag**: if the last operation resulted in a negative value (i.e. the most significant bit is 1).
3. **Overflow Flag**: if the last operation resulted in a signed overflow.
4. **Carry Flag**: if the last operation resulted in a carry out of the most significant bit, i.e. an unsigned overflow.

Every operation may or may not changes these flags to test for zero or nonzero, positive or negative, or overflow conditions, and combinations of these flags express the full range of conditions and cases, e.g. for signed and unsigned values.

Example 2.11 (Zero Flag)

If the code below was just run, then ZF would be set to 1.

```
movq $2, %rax
subq $2, %rax
```

Example 2.12 (Sign Flag)

If the code below was just run, then SF would be set to 1.

```
movq $2, %rax
subq $4, %rax
```

Example 2.13 (Overflow Flag)

If either code below was just run, then OF would be set to 1.

```
movq $0x7fffffffffffffff, %rax
addq $1, %rax
```

```
movq 0x8000000000000000, %rax
addq 0xffffffffffffffff, %rax
```

This is because in the left in signed arithmetic, we have a positive + positive = negative (result is `0x8000000000000000`), which is a signed overflow. Furthermore, in the right we have negative + negative = positive (result is `0x7fffffffffffffff`).

Example 2.14 (Carry Flag)

If the code below was just run, then CF would be set to 1.

```
movq $0xffffffffffffffff, %rax
addq $1, %rax
```

This is because the result is $0x0$, which is a carry out of the most significant bit and an unsigned overflow.

It would be tedious to always set these flags manually, so there are two methods that can be used to *explicitly* set these flags.

Definition 2.20 (Compare)

The **cmp** instruction is used to perform a subtraction between the source and destination, and set the flags accordingly, but it does not store the result.

`cmp_ src, dest`   `dest - src`

The following flags are set if the conditions are met:

1. **ZF** = **1** if `dest == src`
2. **SF** = **1** if `dest < src` (MSB is 1)
3. **OF** = **1** if signed overflow
4. **CF** = **1** if unsigned overflow

Definition 2.21 (Test)

The **test** instruction is used to perform a bitwise AND operation between the source and destination, and set the flags accordingly.

`test_ src, dest`   `dest & src`

The following flags are set if the conditions are met. Note that you can't have carry out (CF) or overflow (OF) if these flags are set.

1. **ZF** = **1** if `dest & src == 0`
2. **SF** = **1** if `dest & src < 0` (MSB is 1)

Example 2.15 (Compare)

Assuming that `%al = 0x80` and `%bl = 0x81`, which flags are set when we execute `cmpb %al, %bl`? Well we must first compute

$$\texttt{\%bl - \%al = 0x81 - 0x80 = 0x81 + } \sim \texttt{ 0x80 + 1 = 0x81 + 0x7F + 1 = 0x101 = 0x01} \tag{5}$$

1. CF=1 since the result is greater than 0xFF (i.e. larger than byte)
2. ZF=0 since the result is not 0
3. SF=0 since the MSB is 0, i.e. there is unsigned overflow
4. OF=0 since there is no signed overflow

For conditional moves and jumps later shown, it basically uses these explicit sets and always compares them to 0. We will see what this means later.

Finally, we can actually set a byte in a register to 1 or 0 based on the value of a flag.

Definition 2.22 (Set)

We can then talk about conditional moves and jumps.

Definition 2.23 (Equality with 0)

The `test` instruction is used to perform a bitwise AND operation between the source and destination, and set the flags accordingly.

```
test_ src, dest                    dest & src
```

The `sete` instruction is used to set the destination to 1 if the zero flag is set, and 0 otherwise.

```
sete_ dest                dest = (ZF == 1) ?  1 :  0
```

The `cmovne` instruction is used to move the source to the destination if the zero flag is not set.

```
cmovne_ src, dest          dest = (ZF == 0) ?  src :  dest
```

Definition 2.24 (Jump)

There are several jump instructions, but essentially they are used to jump to another part of the code. We can use the following mnemonic to jump to a label.

Letter	**Word**
j	jump
n	not
e	equal
s	signed
g	greater (signed interpretation)
l	less (signed interpretation)
a	above (unsigned interpretation)
b	below (unsigned interpretation)

Table 2: Letter to Word Mapping

Figure 4: Mnemonic for Jump Instructions

For completeness, we include all the jump instructions.

Signed Comparison	Unsigned Comparison	Description
je (jz)		jump if equal (==) or jump if zero
jne (jnz)		jump if not equal (!=)
js		jump if negative
jns		jump if non-negative
jg (jnle)	ja (jnbe)	jump if greater (>)
jge (jnl)	jae (jnb)	jump if greater than or equal (>=)
jl (jnge)	jb (jnae)	jump if less (<)
jle (jng)	jbe (jna)	jump if less than or equal (<=)

Table 3: Comparison Instructions in Assembly

Figure 5: All jump instructions

Definition 2.25 (int)

The `int` instruction is used to generate a software interrupt. It is often used to invoke a system call.

Definition 2.26 (ret)

The `ret` instruction is used to return from a function. It returns the value in the `%rax` register.

Now we can have a basic idea of how if statements can be used as a sequence of conditionals and jump operators. Let's first look at the **goto** version of C.

Definition 2.27 (Goto Syntax)

The goto version processes instructions sequentially as long as there is no jump. This is useful because compilers translating code into assembly designate a jump when a condition is true. Contrast this behavior with the structure of an if statement, where a "jump" (to the else) occurs when conditions are not true. The goto form captures this difference in logic.

```c
int getSmallest(int x, int y) {
  int smallest;
  if ( x > y ) { //if (conditional)
    smallest = y; //then statement
  }
  else {
    smallest = x; //else statement
  }
  return smallest;
}
.
.
.
.
.
```

```c
int getSmallest(int x, int y) {
  int smallest;

  if (x <= y ) { //if (!conditional)
    goto else_statement;
  }
  smallest = y; //then statement
  goto done;

else_statement:
  smallest = x; //else statement

done:
  return smallest;
}
```

Figure 6: C vs GoTo code of the same function. While GoTo code allows us to view C more like assmebly, it is generally not readable and is not considered best practice.

Now let's see how if statements are implemented by taking a look at this function straight up in assembly.

```c
int getSmallest(int x, int y) {
  int smallest;
  if ( x > y ) { //if (conditional)
    smallest = y; //then statement
  }
  else {
    smallest = x; //else statement
  }
  return smallest;
}
.
```

```
Dump of assembler code for function getSmallest:
0x40059a <+4>:   mov    %edi,-0x14(%rbp)
0x40059d <+7>:   mov    %esi,-0x18(%rbp)
0x4005a0 <+10>:  mov    -0x14(%rbp),%eax
0x4005a3 <+13>:  cmp    -0x18(%rbp),%eax
0x4005a6 <+16>:  jle    0x4005b0 <getSmallest+26>
0x4005a8 <+18>:  mov    -0x18(%rbp),%eax
0x4005ae <+24>:  jmp    0x4005b9 <getSmallest+35>
0x4005b0 <+26>:  mov    -0x14(%rbp),%eax
0x4005b9 <+35>:  pop    %rbp
0x4005ba <+36>:  retq
```

Figure 7: Assembly code of a simple if statement

Again, note that since we are working with int types, the respective parameter registers are `%edi` and `%esi`, the respective lower 32-bits of the registers `%rdi` and `%rsi`. Let's walk through this again.

1. The first mov instruction copies the value located in register %edi (the first parameter, x) and places it at memory location %rbp-0x14 on the call stack. The instruction pointer (%rip) is set to the address of the next instruction, or 0x40059d.
2. The second mov instruction copies the value located in register %esi (the second parameter, y) and places it at memory location %rbp-0x18 on the call stack. The instruction pointer (%rip) updates to point to the address of the next instruction, or 0x4005a0.
3. The third mov instruction copies x to register %eax. Register %rip updates to point to the address of the next instruction in sequence.
4. The cmp instruction compares the value at location %rbp-0x18 (the second parameter, y) to x and sets appropriate condition code flag registers. Register %rip advances to the address of the next instruction, or 0x4005a6.
5. The jle instruction at address 0x4005a6 indicates that if x is less than or equal to y, the next instruction that should execute should be at location <getSmallest+26> and that %rip should be set to address 0x4005b0. Otherwise, %rip is set to the next instruction in sequence, or 0x4005a8.

With the `cmov` instruction, this can be a lot shorter. With the gcc compiler with level 1 optimizations turned on, we can see that a lot of redundancies are turned off.

```
<getSmallest>:
0x400546 <+0>: cmp    %esi,%edi       #compare x and y
0x400548 <+2>: mov    %esi,%eax       #copy y to %eax
0x40054a <+4>: cmovle %edi,%eax       #if (x<=y) copy x to %eax
0x40054d <+7>: retq                   #return %eax
```

Figure 8: Compiled with `gcc -O1 -o getSmallest getSmallest.c`

Like if statements, loops in assembly can be implementing using jump functions that revisit some instruction address based on the result on an evaluated condition. Let's take a look at a basic loop function.

```c
int sumUp(int n) {
  int total = 0;
  int i = 1;

  while (i <= n) {
    total += i;
    i++;
  }
  return total;
}
.
.
.
.
.
.
```

```
Dump of assembler code for function sumUp:
0x400526 <+0>:   push   %rbp
0x400527 <+1>:   mov    %rsp,%rbp
0x40052a <+4>:   mov    %edi,-0x14(%rbp)
0x40052d <+7>:   mov    $0x0,-0x8(%rbp)
0x400534 <+14>:  mov    $0x1,-0x4(%rbp)
0x40053b <+21>:  jmp    0x400547 <sumUp+33>
0x40053d <+23>:  mov    -0x4(%rbp),%eax
0x400540 <+26>:  add    %eax,-0x8(%rbp)
0x400543 <+29>:  add    $0x1,-0x4(%rbp)
0x400547 <+33>:  mov    -0x4(%rbp),%eax
0x40054a <+36>:  cmp    -0x14(%rbp),%eax
0x40054d <+39>:  jle    0x40053d <sumUp+23>
0x40054f <+41>:  mov    -0x8(%rbp),%eax
0x400552 <+44>:  pop    %rbp
0x400553 <+45>:  retq
```

Figure 9: Simple loop function in C and assembly.

Finally, we want to let the reader know the convention of calle and caller saved registers. The compiler tries to pick these registers, and by convention in x86, we have the following.

%rax	Return value - Caller saved	%r8	Argument #5 - Caller saved
%rbx	Callee saved	%r9	Argument #6 - Caller saved
%rcx	Argument #4 - Caller saved	%r10	Caller saved
%rdx	Argument #3 - Caller saved	%r11	Caller Saved
%rsi	Argument #2 - Caller saved	%r12	Callee saved
%rdi	Argument #1 - Caller saved	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

Figure 10: Caller save and callee save registers.

So far, we've traced through simple functions in assembly. In this section, we discuss the interaction between multiple functions in assembly in the context of a larger program. We also introduce some new instructions involved with function management.

Definition 2.28 (Leave)

The **leave** instruction is used to deallocate the current stack frame. For example, the leaveq instruction is a shorthand that the compiler uses to restore the stack and frame pointers as it prepares to leave a function. When the callee function finishes execution, leaveq ensures that the frame pointer is restored to its previous value. It is equivalent to the following two instructions:

```
leaveq                    movq %rbp, %rsp
                               popq %rbp
```

Definition 2.29 (Call and Return)

The **call** instruction is used to call a function and the **ret** to return from a function. The callq and retq instructions play a prominent role in the process where one function calls another. Both instructions modify the instruction pointer (register %rip).

1. When the caller function executes the callq instruction, the current value of %rip is saved on the stack to represent the return address, or the program address at which the caller resumes executing once the callee function finishes. The callq instruction also replaces the value of %rip with the address of the callee function.

```
callq addr <fname>                 push %rip
                              mov addr, %rip
```

2. The retq instruction restores the value of %rip to the value saved on the stack, ensuring that the program resumes execution at the program address specified in the caller function. Any value returned by the callee is stored in %rax or one of its component registers (e.g., %eax). The retq instruction is usually the last instruction that executes in any function.

```
retq                    pop %rip
```

Let's work through an example to solidify our knowledge.

Example 2.16 (Calling Functions in Assembly)

Let's take the following code and trace through main.

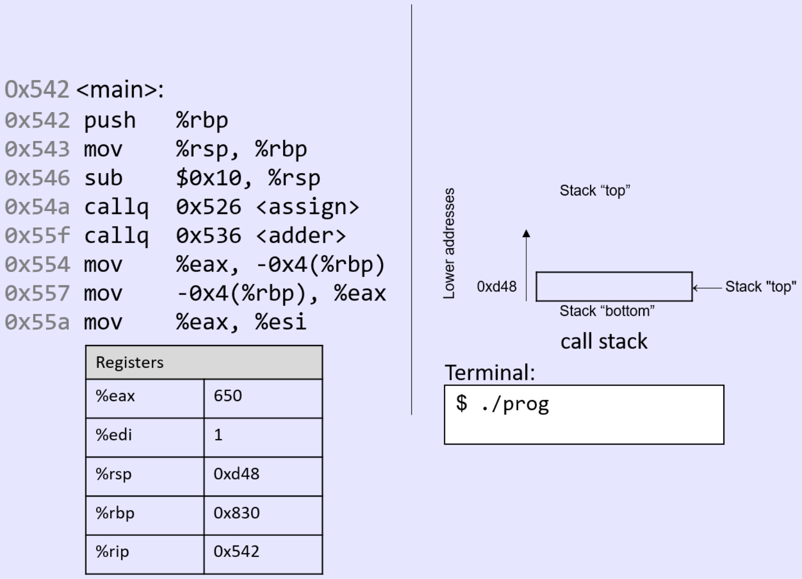
```c
#include <stdio.h>

int assign(void) {
    int y = 40;
    return y;
}

int adder(void) {
    int a;
    return a + 2;
}

int main(void) {
    int x;
    assign();
    x = adder();
    printf("x is: %d\n", x);
    return 0;
}
.
.
.
.
.
.
.
.
.
.
.
.
```

```
0000000000400526 <assign>:
  400526:       55                      push   %rbp
  400527:       48 89 e5                mov    %rsp,%rbp
  40052a:       c7 45 fc 28 00 00 00    movl   $0x28,-0x4(%rbp)
  400531:       8b 45 fc                mov    -0x4(%rbp),%eax
  400534:       5d                      pop    %rbp
  400535:       c3                      retq

0000000000400536 <adder>:
  400536:       55                      push   %rbp
  400537:       48 89 e5                mov    %rsp,%rbp
  40053a:       8b 45 fc                mov    -0x4(%rbp),%eax
  40053d:       83 c0 02                add    $0x2,%eax
  400540:       5d                      pop    %rbp
  400541:       c3                      retq

0000000000400542 <main>:
  400542:       55                      push   %rbp
  400543:       48 89 e5                mov    %rsp,%rbp
  400546:       48 83 ec 10             sub    $0x10,%rsp
  40054a:       e8 e3 ff ff ff          callq  400526 <assign>
  40054f:       e8 d2 ff ff ff          callq  400536 <adder>
  400554:       89 45 fc                mov    %eax,-0x4(%rbp)
  400557:       8b 45 fc                mov    -0x4(%rbp),%eax
  40055a:       89 c6                   mov    %eax,%esi
  40055c:       bf 04 06 40 00          mov    $0x400604,%edi
  400561:       b8 00 00 00 00          mov    $0x0,%eax
  400566:       e8 95 fe ff ff          callq  400400
    <printf@plt>
  40056b:       b8 00 00 00 00          mov    $0x0,%eax
  400570:       c9                      leaveq
  400571:       c3                      retq
```

Figure 11: C code and its assembly equivalent. Main function calls two other functions.

Let's trace through what happens here in detail. This will be long.

1. %rbp is the base pointer that is initialized to something. Before we even begin main, say that we have the following initializations, where %eax, %edi is garbage. %rsp denotes where on the stack we are right before calling to main, **%rbp** is the base pointer to the current program, and **%rip** should be the address of the first instruction in main. Again since we work with integers we use the lower 32-bits of the registers. %rip now points to the next instruction.



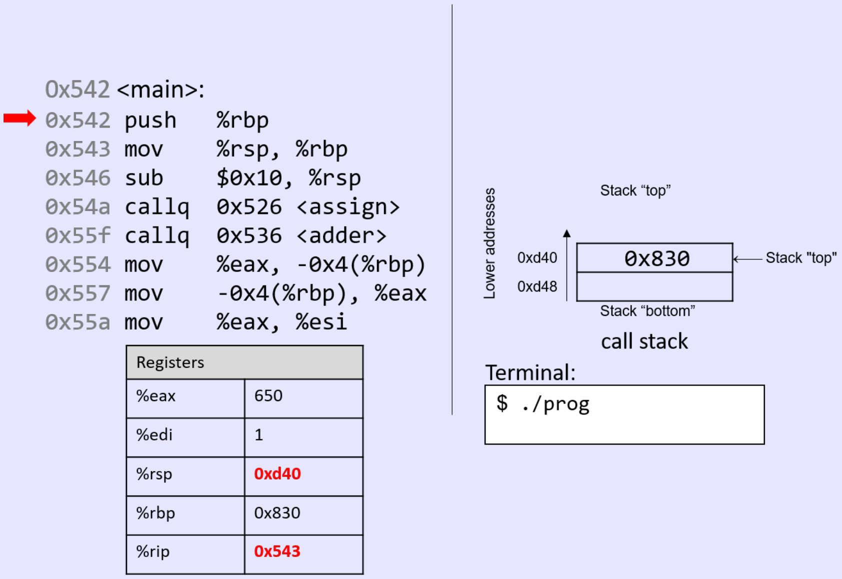
```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	650
%edi	1
%rsp	0xd48
%rbp	0x830
%rip	0x542

Terminal:

```
$ ./prog
```

2. Now we start the main function. By calling main, the base pointer `%rbp` of the stack outside of the main frame will be overwritten by the base of the main stack frame, so we must save it for when main is done. Therefore, we push it onto the stack where `%rsp` is pointing. `%rip` now points to the next instruction.



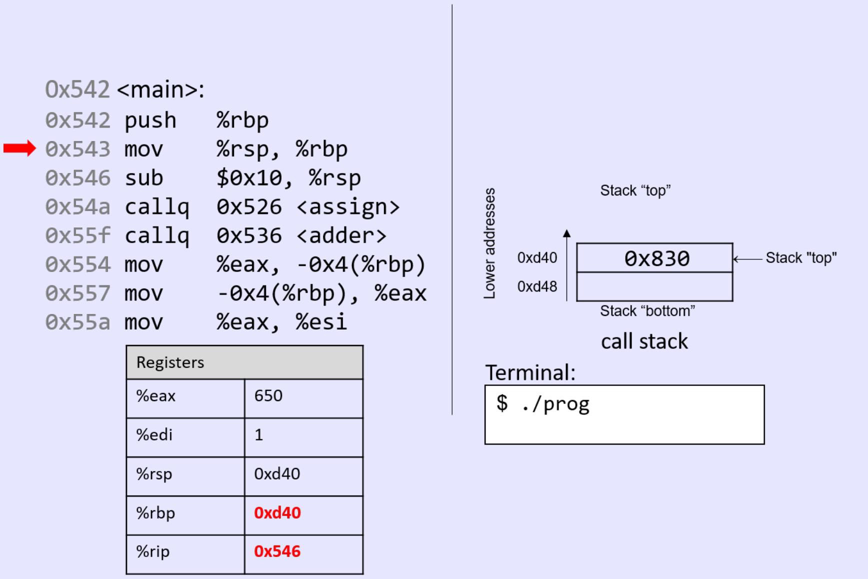
```
0x542 <main>:
➡ 0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	650
%edi	1
%rsp	**0xd40**
%rbp	0x830
%rip	**0x543**

Terminal:

```
$ ./prog
```

3. Then we actually change the location of the base pointer to the top of the stack, which now includes the first instruction in main.



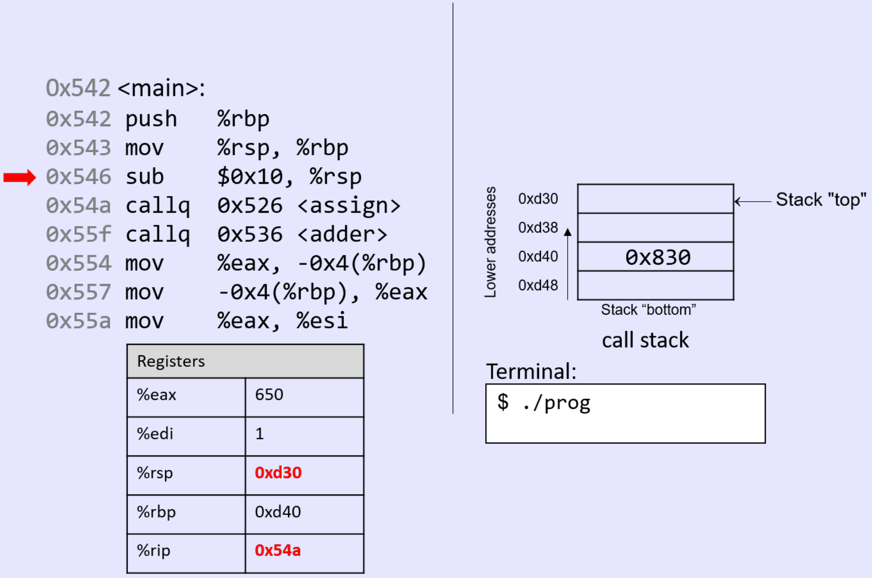
```
    0x542 <main>:
    0x542 push   %rbp
➡ 0x543 mov    %rsp, %rbp
    0x546 sub    $0x10, %rsp
    0x54a callq  0x526 <assign>
    0x55f callq  0x536 <adder>
    0x554 mov    %eax, -0x4(%rbp)
    0x557 mov    -0x4(%rbp), %eax
    0x55a mov    %eax, %esi
```

Registers	
%eax	650
%edi	1
%rsp	0xd40
%rbp	**0xd40**
%rip	**0x546**

Terminal:

```
$ ./prog
```

4. Now we manually change the stack pointer and have it grow by two bytes (`0x10`). Therefore, `%rsp` is decremented by `0x10` and `%rip` points to the next instruction at `0x54a`.



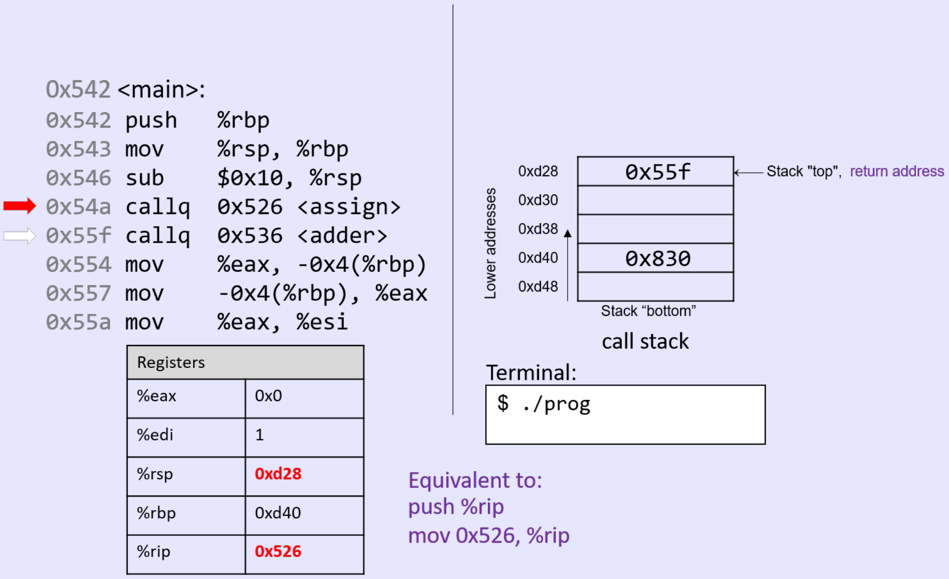
```
    0x542 <main>:
    0x542 push   %rbp
    0x543 mov    %rsp, %rbp
➡ 0x546 sub    $0x10, %rsp
    0x54a callq  0x526 <assign>
    0x55f callq  0x536 <adder>
    0x554 mov    %eax, -0x4(%rbp)
    0x557 mov    -0x4(%rbp), %eax
    0x55a mov    %eax, %esi
```

Registers	
%eax	650
%edi	1
%rsp	**0xd30**
%rbp	0xd40
%rip	**0x54a**

Terminal:

```
$ ./prog
```

5. Now the next instruction pointed at by `%rip` is the `callq` instruction, which tells to go to the address of the `assign` function. We by default first update `%rip` to point to the next instruction at `0x55f`. However, this should not be the actual next instruction that we execute since we are calling another function. Rather, we want to update `%rip` to address `0x526` where `assign` is located at, but after completion we also want to know that we want to execute the instruction after it at address `0x55f`. Therefore, we should *save* address `0x55f` onto the stack and then update `%rip` to point to `0x526`. This is what we refer to as a **return address**.



```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

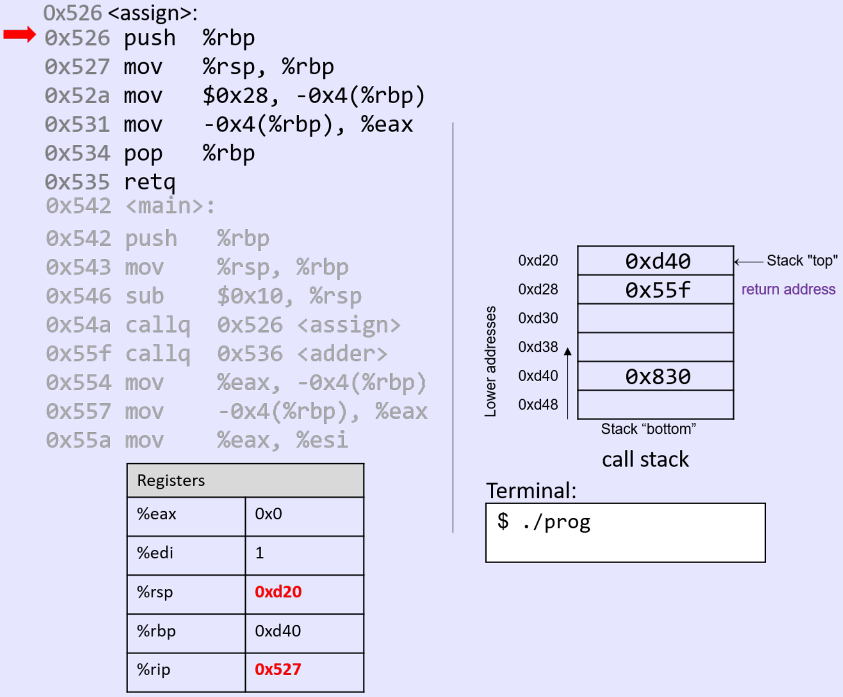
Registers	
%eax	0x0
%edi	1
%rsp	0xd28
%rbp	0xd40
%rip	0x526

Terminal:

```
$ ./prog
```

Equivalent to:
push %rip
mov 0x526, %rip

6. %rip is incremented to the next address. We step into the `assign` function, which is now a new stack frame, so the first thing we do is save the base pointer of the main stack frame onto the stack since we must immediately update it with the base pointer of the assign stack frame, which is where %rsp is pointing to.



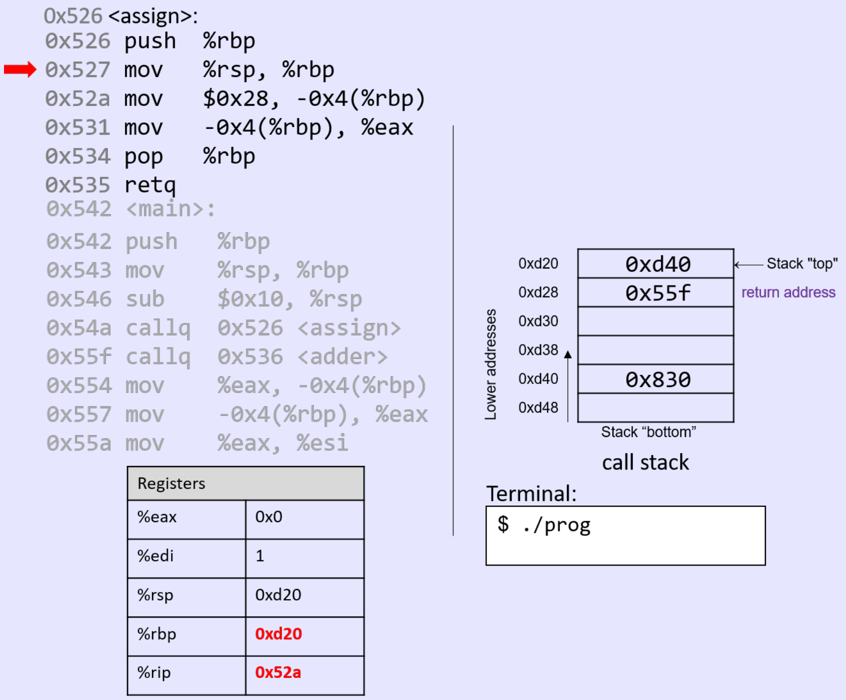
```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x0
%edi	1
%rsp	0xd20
%rbp	0xd40
%rip	0x527

Terminal:

```
$ ./prog
```

7. %rip is incremented to the next address. We then update the base pointer to the top of the stack.



```
    0x526 <assign>:
    0x526 push  %rbp
➡   0x527 mov   %rsp, %rbp
    0x52a mov   $0x28, -0x4(%rbp)
    0x531 mov   -0x4(%rbp), %eax
    0x534 pop   %rbp
    0x535 retq
    0x542 <main>:
    0x542 push   %rbp
    0x543 mov    %rsp, %rbp
    0x546 sub    $0x10, %rsp
    0x54a callq  0x526 <assign>
    0x55f callq  0x536 <adder>
    0x554 mov    %eax, -0x4(%rbp)
    0x557 mov    -0x4(%rbp), %eax
    0x55a mov    %eax, %esi
```

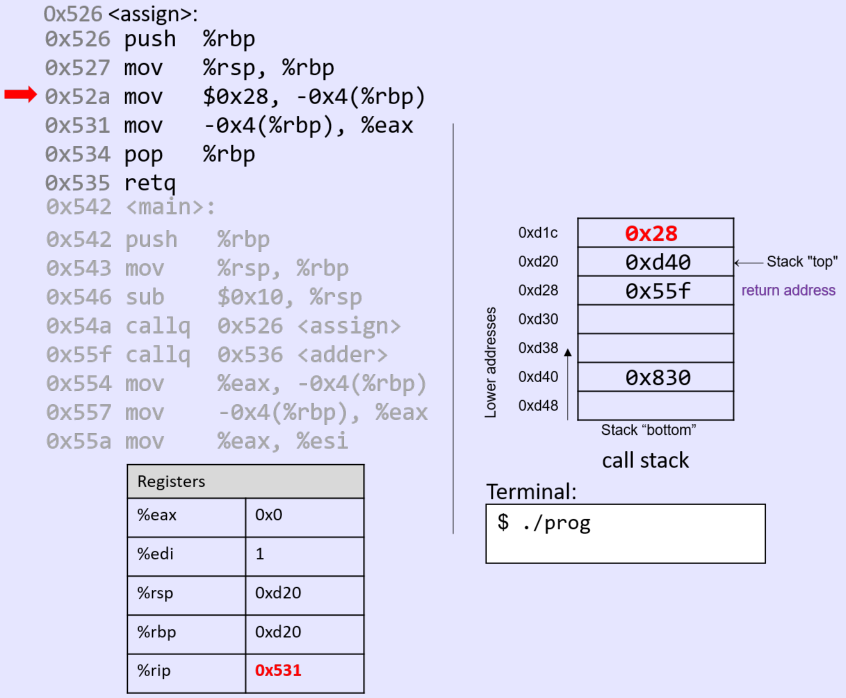
Registers	
%eax	0x0
%edi	1
%rsp	0xd20
%rbp	**0xd20**
%rip	**0x52a**

0xd20	0xd40	Stack "top"
0xd28	0x55f	return address
0xd30		
0xd38		
0xd40	0x830	
0xd48		

Terminal:

```
$ ./prog
```

8. Now we want to move the number `0x28` (40) into the memory location `-0x4(%rbp)` of the stack, which is 4 bytes above the frame pointer, which is also the stack pointer. It is common that the frame pointer is used to reference locations on the stack. Note that this does not update the stack pointer.



```
    0x526 <assign>:
    0x526 push  %rbp
    0x527 mov   %rsp, %rbp
➡   0x52a mov   $0x28, -0x4(%rbp)
    0x531 mov   -0x4(%rbp), %eax
    0x534 pop   %rbp
    0x535 retq
    0x542 <main>:
    0x542 push   %rbp
    0x543 mov    %rsp, %rbp
    0x546 sub    $0x10, %rsp
    0x54a callq  0x526 <assign>
    0x55f callq  0x536 <adder>
    0x554 mov    %eax, -0x4(%rbp)
    0x557 mov    -0x4(%rbp), %eax
    0x55a mov    %eax, %esi
```

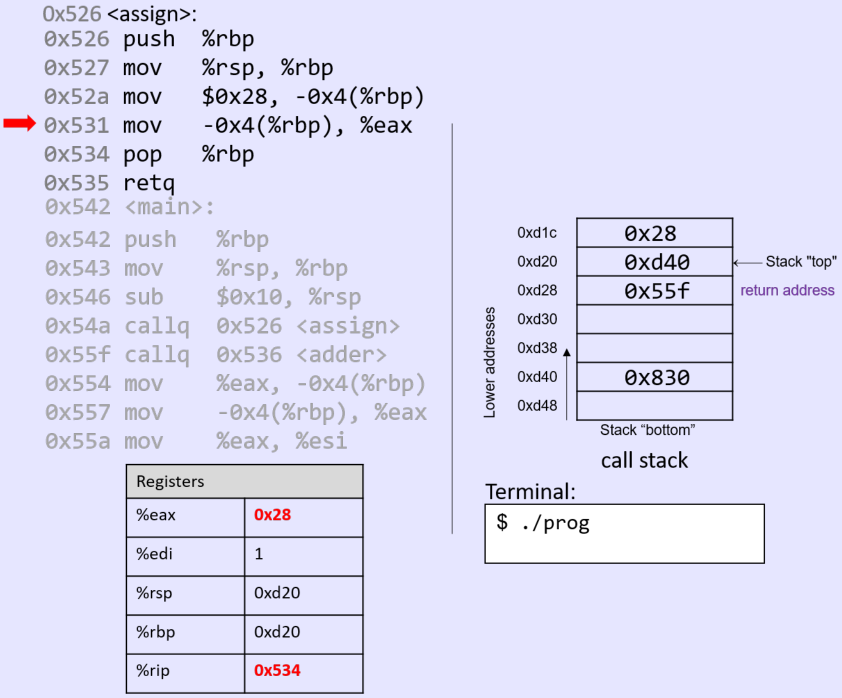
Registers	
%eax	0x0
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	**0x531**

0xd1c	**0x28**	
0xd20	0xd40	Stack "top"
0xd28	0x55f	return address
0xd30		
0xd38		
0xd40	0x830	
0xd48		

Terminal:

```
$ ./prog
```

9. Now we take the same address where we stored `0x28` to and move it into `%eax`, effectively loading 40 onto the return value.



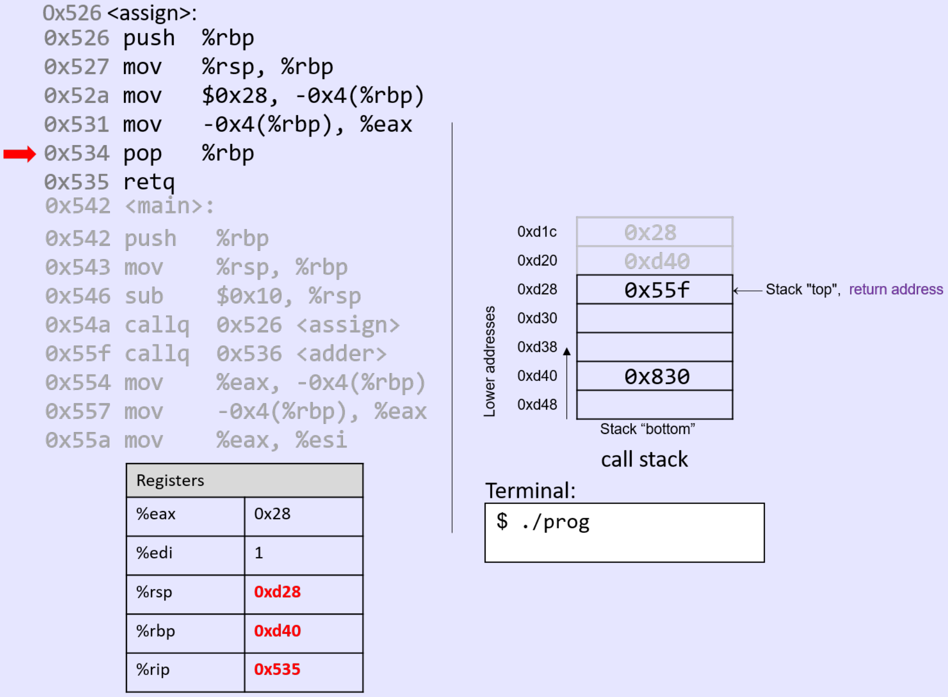
```
  0x526 <assign>:
  0x526 push  %rbp
  0x527 mov   %rsp, %rbp
  0x52a mov   $0x28, -0x4(%rbp)
➡ 0x531 mov   -0x4(%rbp), %eax
  0x534 pop   %rbp
  0x535 retq
  0x542 <main>:
  0x542 push   %rbp
  0x543 mov    %rsp, %rbp
  0x546 sub    $0x10, %rsp
  0x54a callq  0x526 <assign>
  0x55f callq  0x536 <adder>
  0x554 mov    %eax, -0x4(%rbp)
  0x557 mov    -0x4(%rbp), %eax
  0x55a mov    %eax, %esi
```

Registers	
%eax	**0x28**
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	**0x534**

Terminal:

```
$ ./prog
```

10. We see that we will return this value soon, but before we do, we want to make sure that when the assign stack frame gets deleted (not really, but overwritten), we want to restore the base pointer of the main stack frame. We have already saved this before at `%rsp`, which hasn't changed since we only worked with displacements from the base pointer. We retrieve the main stack pointer data and load it back into `%rbp`. Note that this increments `%rsp` by 8 bytes, shrinking the stack, and we are technically out of the assign stack frame.



```
  0x526 <assign>:
  0x526 push  %rbp
  0x527 mov   %rsp, %rbp
  0x52a mov   $0x28, -0x4(%rbp)
  0x531 mov   -0x4(%rbp), %eax
➡ 0x534 pop   %rbp
  0x535 retq
  0x542 <main>:
  0x542 push   %rbp
  0x543 mov    %rsp, %rbp
  0x546 sub    $0x10, %rsp
  0x54a callq  0x526 <assign>
  0x55f callq  0x536 <adder>
  0x554 mov    %eax, -0x4(%rbp)
  0x557 mov    -0x4(%rbp), %eax
  0x55a mov    %eax, %esi
```

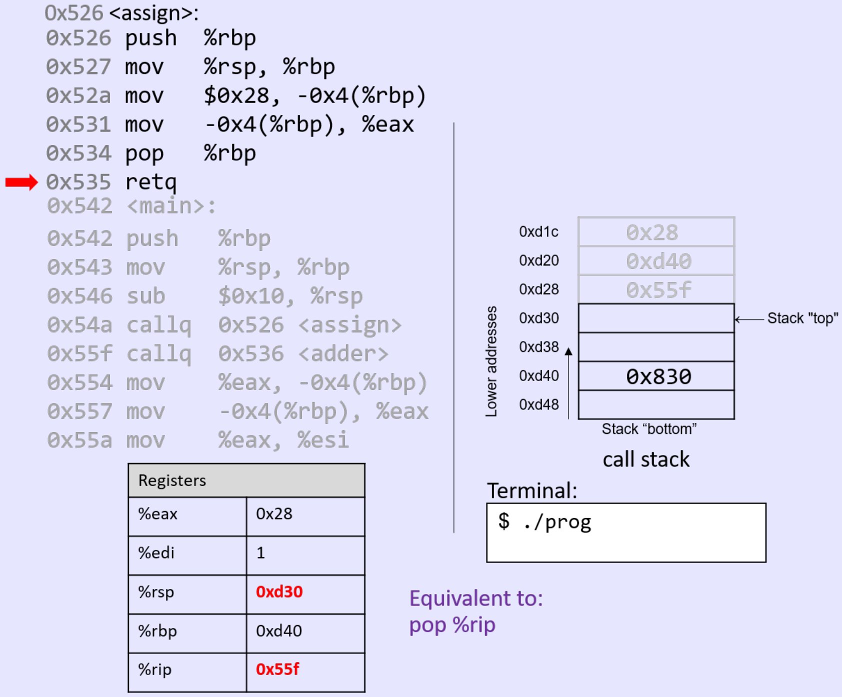
Registers	
%eax	0x28
%edi	1
%rsp	**0xd28**
%rbp	**0xd40**
%rip	**0x535**

Terminal:

```
$ ./prog
```

11. Note that at this point, since `%rbp` was popped off, the next value that is at the top of the stack

is the address `%rip` that we store earlier, which points to the next execution in main. When `retq` executes, this value at the top of the stack is popped into `%rip`, allowing main to continue executing within the main stack frame. Note that the return value is stored in `%eax`.



```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

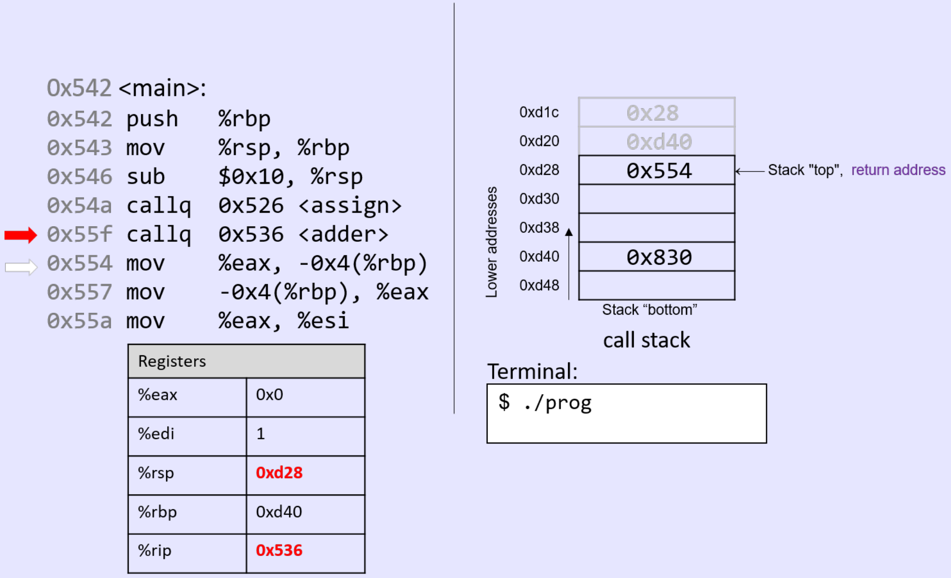
Registers	
%eax	0x28
%edi	1
%rsp	0xd30
%rbp	0xd40
%rip	0x55f

Terminal:

```
$ ./prog
```

Equivalent to:
pop %rip

12. Now we execute the next instruction in `%rip` which is a call to the `adder` function. `%rip` is automatically updated to the next address at `0x554`, but since this is a `callq` instruction, we first want to store this `%rip` into the stack so we can come back to it, and then update `%rip` to the first instruction in `adder`, which is address `0x536`.



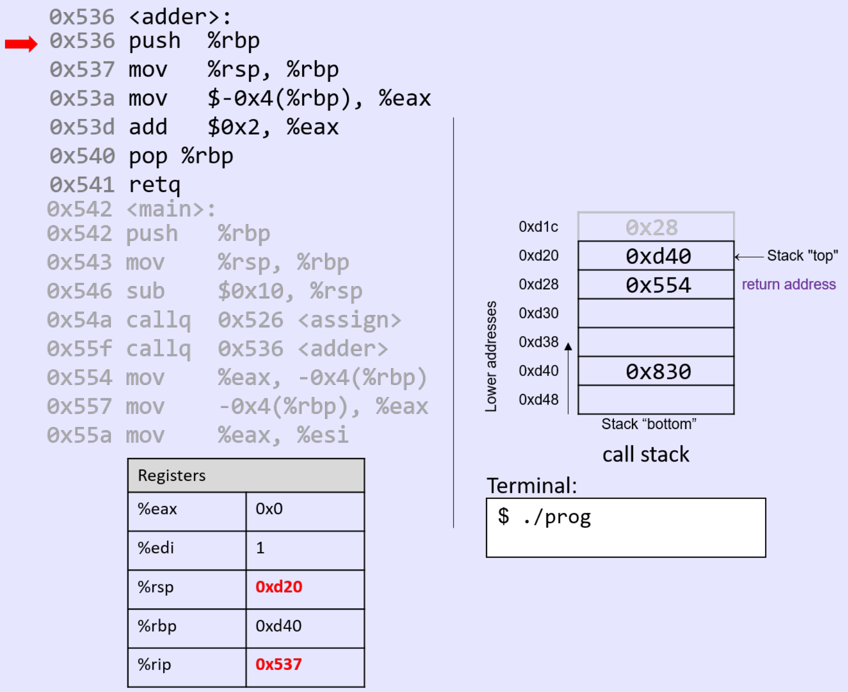
```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x0
%edi	1
%rsp	0xd28
%rbp	0xd40
%rip	0x536

Terminal:

```
$ ./prog
```

13. Since we are in the adder function, this creates a new stack frame and we must update `%rbp`. Again, we don't want to overwrite the base pointer of main, so we save it onto the stack by pushing `%rbp`.



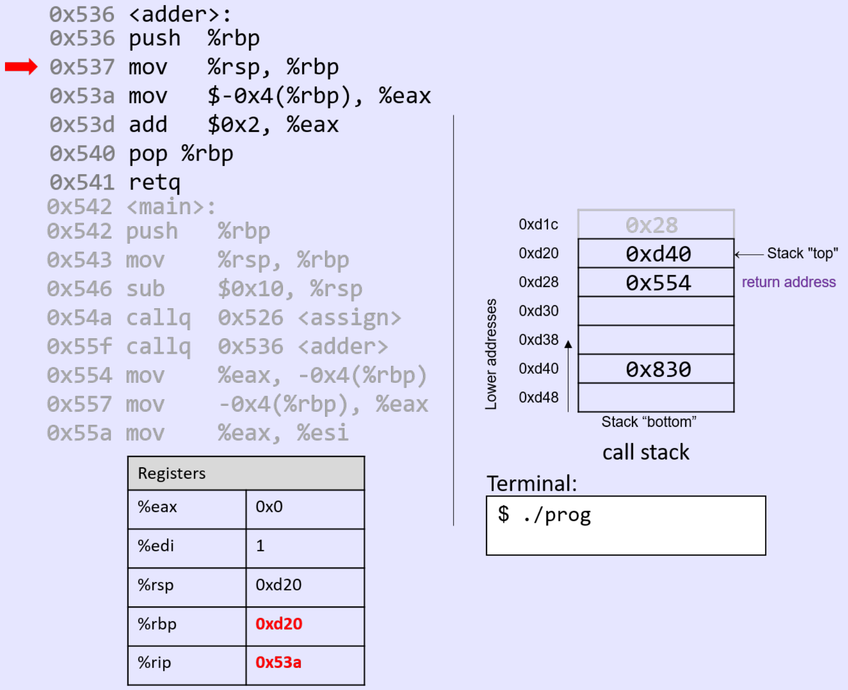
```
0x536 <adder>:
0x536 push  %rbp
0x537 mov   %rsp, %rbp
0x53a mov   $-0x4(%rbp), %eax
0x53d add   $0x2, %eax
0x540 pop %rbp
0x541 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x0
%edi	1
%rsp	0xd20
%rbp	0xd40
%rip	0x537

Terminal:

```
$ ./prog
```

14. Then we update %rbp to the current stack pointer.



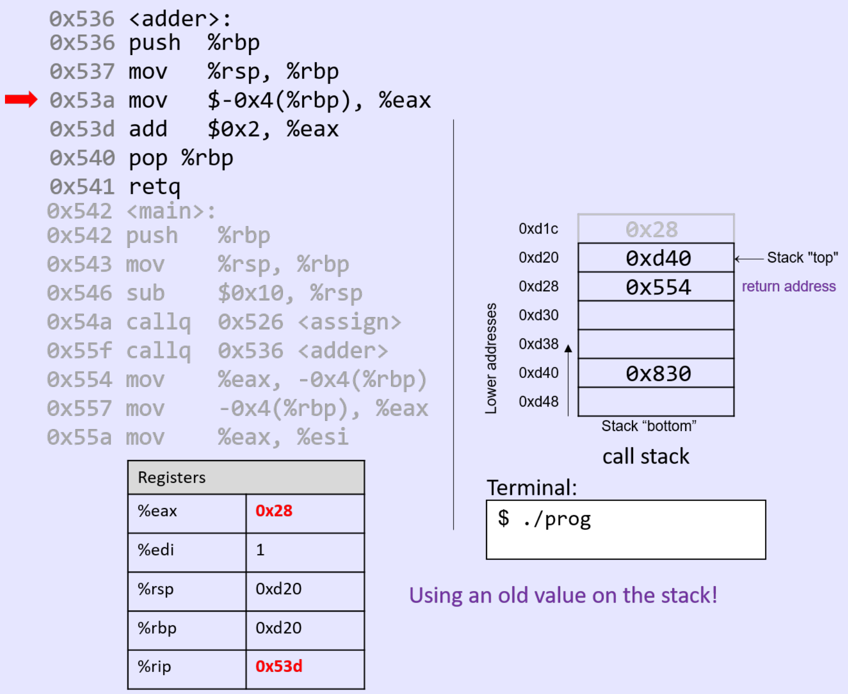
```
0x536 <adder>:
0x536 push  %rbp
0x537 mov   %rsp, %rbp
0x53a mov   $-0x4(%rbp), %eax
0x53d add   $0x2, %eax
0x540 pop %rbp
0x541 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

Registers	
%eax	0x0
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	0x53a

Terminal:

```
$ ./prog
```

15. This part is a bit tricky. Note that the value of 0x28 still lives at 0xd1c, which is conveniently at address -0x4(%rbp). Therefore, when we call int a; in that corresponding line in adder, we can actually add 2 to it, though it seems like there was no value assigned to it. This is just a trick though. So, we can take these remnant value and store it into %eax.



```
   0x536 <adder>:
   0x536 push  %rbp
   0x537 mov   %rsp, %rbp
➡  0x53a mov   $-0x4(%rbp), %eax
   0x53d add   $0x2, %eax
   0x540 pop %rbp
   0x541 retq
   0x542 <main>:
   0x542 push   %rbp
   0x543 mov    %rsp, %rbp
   0x546 sub    $0x10, %rsp
   0x54a callq  0x526 <assign>
   0x55f callq  0x536 <adder>
   0x554 mov    %eax, -0x4(%rbp)
   0x557 mov    -0x4(%rbp), %eax
   0x55a mov    %eax, %esi
```

Registers	
%eax	0x28
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	0x53d

0xd1c	0x28	
0xd20	0xd40	Stack "top"
0xd28	0x554	return address
0xd30		
0xd38		
0xd40	0x830	
0xd48		

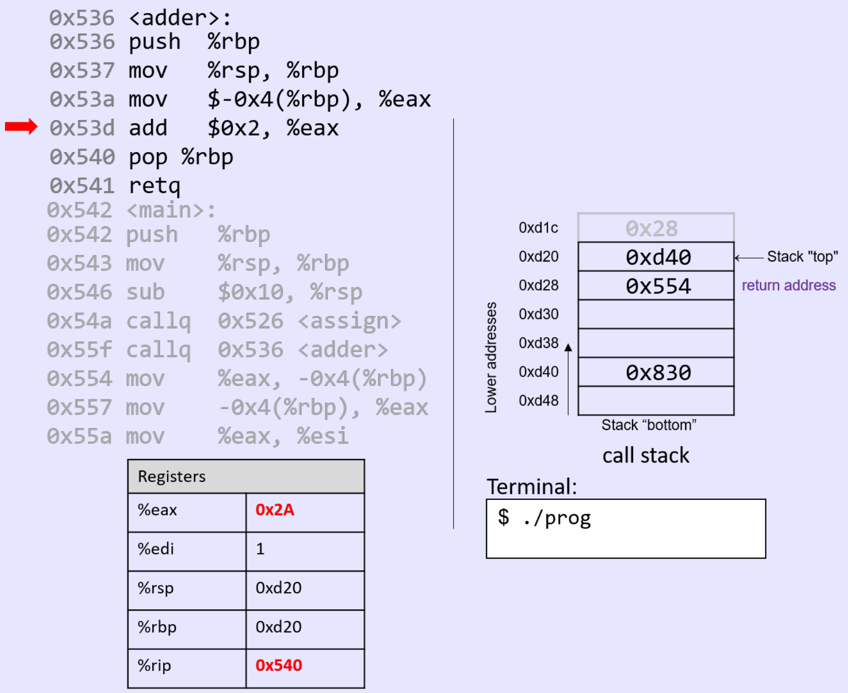
call stack

Terminal:

```
$ ./prog
```

Using an old value on the stack!

16. We then add 2 to it.



```
   0x536 <adder>:
   0x536 push  %rbp
   0x537 mov   %rsp, %rbp
   0x53a mov   $-0x4(%rbp), %eax
➡  0x53d add   $0x2, %eax
   0x540 pop %rbp
   0x541 retq
   0x542 <main>:
   0x542 push   %rbp
   0x543 mov    %rsp, %rbp
   0x546 sub    $0x10, %rsp
   0x54a callq  0x526 <assign>
   0x55f callq  0x536 <adder>
   0x554 mov    %eax, -0x4(%rbp)
   0x557 mov    -0x4(%rbp), %eax
   0x55a mov    %eax, %esi
```

Registers	
%eax	0x2A
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	0x540

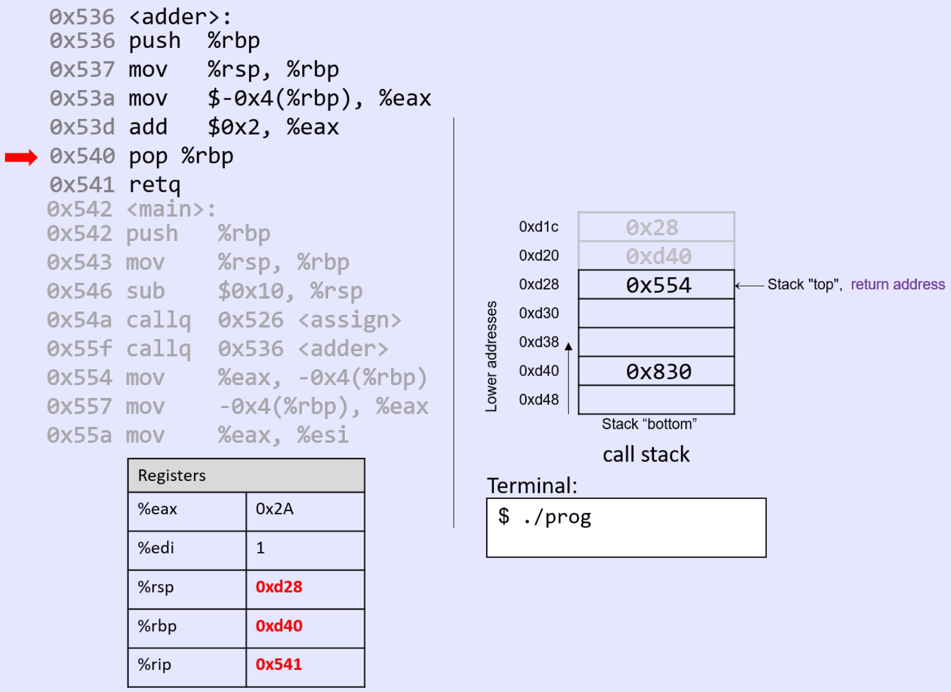
0xd1c	0x28	
0xd20	0xd40	Stack "top"
0xd28	0x554	return address
0xd30		
0xd38		
0xd40	0x830	
0xd48		

call stack

Terminal:

```
$ ./prog
```

17. Now we are almost done, so we pop the base pointer of the main stack frame, at `0xd40`, back into `%rbp`.



```
   0x536 <adder>:
   0x536 push  %rbp
   0x537 mov   %rsp, %rbp
   0x53a mov   $-0x4(%rbp), %eax
   0x53d add   $0x2, %eax
➡  0x540 pop %rbp
   0x541 retq
   0x542 <main>:
   0x542 push   %rbp
   0x543 mov    %rsp, %rbp
   0x546 sub    $0x10, %rsp
   0x54a callq  0x526 <assign>
   0x55f callq  0x536 <adder>
   0x554 mov    %eax, -0x4(%rbp)
   0x557 mov    -0x4(%rbp), %eax
   0x55a mov    %eax, %esi
```

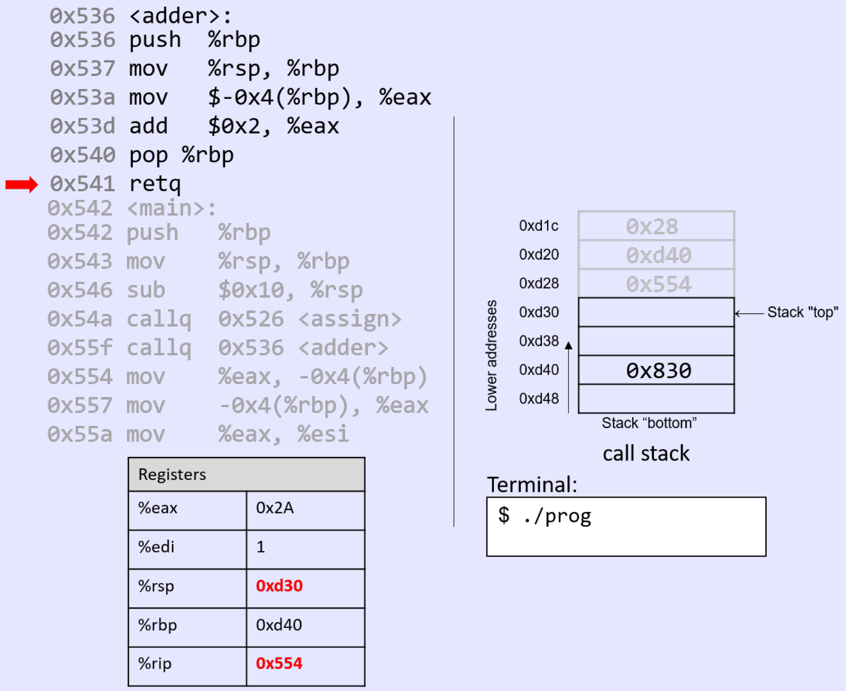
Registers	
%eax	0x2A
%edi	1
%rsp	0xd28
%rbp	0xd40
%rip	0x541

0xd1c	0x28
0xd20	0xd40
0xd28	0x554
0xd30	
0xd38	
0xd40	0x830
0xd48	

Terminal:

```
$ ./prog
```

18. We now return the value in `%eax` and pop the base pointer of the adder stack frame, which simply updates the instruction pointer `%rip` back to the next instruction in main. This is equivalent to `pop %rip`, which is equivalent to moving the stack pointer `%rsp` into `%rip` and then shrinking the stack by 8 bytes `subq $8, %rsp`.



```
   0x536 <adder>:
   0x536 push  %rbp
   0x537 mov   %rsp, %rbp
   0x53a mov   $-0x4(%rbp), %eax
   0x53d add   $0x2, %eax
   0x540 pop %rbp
➡  0x541 retq
   0x542 <main>:
   0x542 push   %rbp
   0x543 mov    %rsp, %rbp
   0x546 sub    $0x10, %rsp
   0x54a callq  0x526 <assign>
   0x55f callq  0x536 <adder>
   0x554 mov    %eax, -0x4(%rbp)
   0x557 mov    -0x4(%rbp), %eax
   0x55a mov    %eax, %esi
```

Registers	
%eax	0x2A
%edi	1
%rsp	0xd30
%rbp	0xd40
%rip	0x554

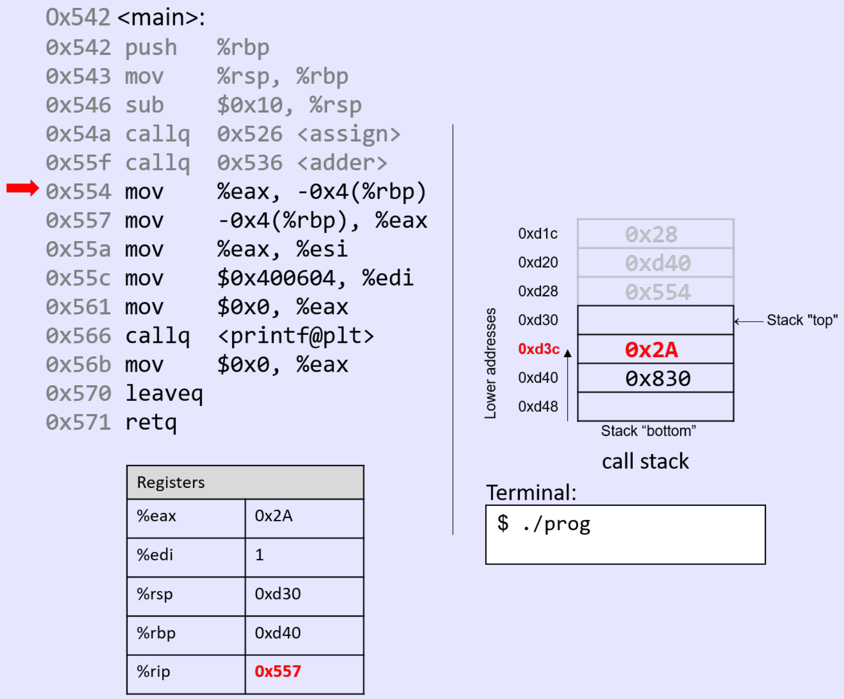
0xd1c	0x28
0xd20	0xd40
0xd28	0x554
0xd30	
0xd38	
0xd40	0x830
0xd48	

Terminal:

```
$ ./prog
```

19. Now it is relatively straightforward since we do the rest in main (except for the print statement). The current value in `%eax` represents the return value of adder. We want to put this in the variable `x`, which we have already allocated some memory for right above the base pointer in the main stack frame. We move it there. Note that right after, it places this right back into

%eax.



```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```

Lower addresses	
0xd1c	0x28
0xd20	0xd40
0xd28	0x554
0xd30	← Stack "top"
0xd3c	0x2A
0xd40	0x830
0xd48	

Stack "bottom"

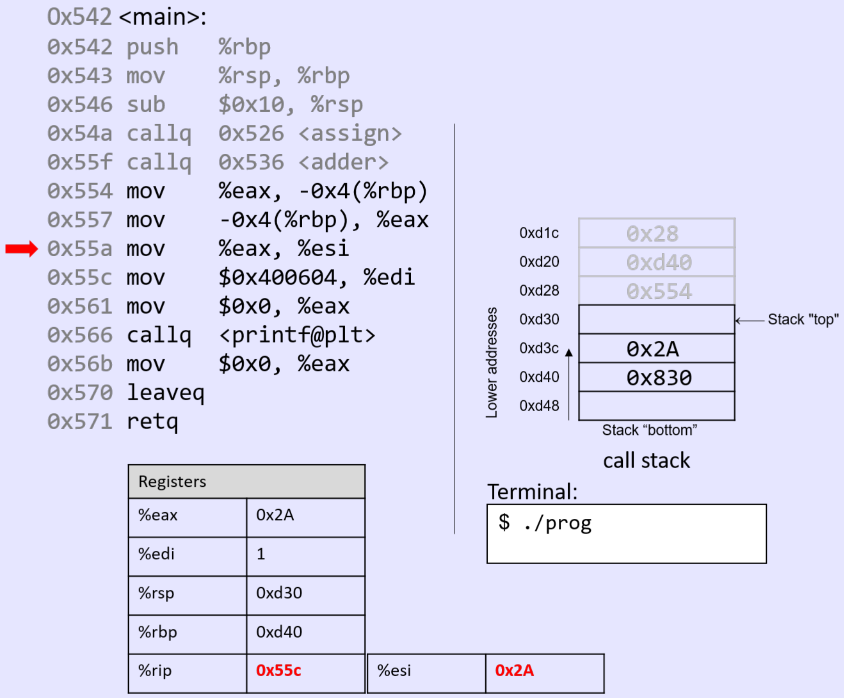
call stack

Registers	
%eax	0x2A
%edi	1
%rsp	0xd30
%rbp	0xd40
%rip	0x557

Terminal:

```
$ ./prog
```

20. the mov instruction at address 0x55a copies the value in `%eax` (or 0x2A) to register `%esi`, which is the 32-bit component register associated with `%rsi` and typically stores the second parameter to a function. We can see why since this will be put into a print statement, which is a function, and `x = %esi` is the second argument of `printf`.



```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```

Lower addresses	
0xd1c	0x28
0xd20	0xd40
0xd28	0x554
0xd30	← Stack "top"
0xd3c	0x2A
0xd40	0x830
0xd48	

Stack "bottom"

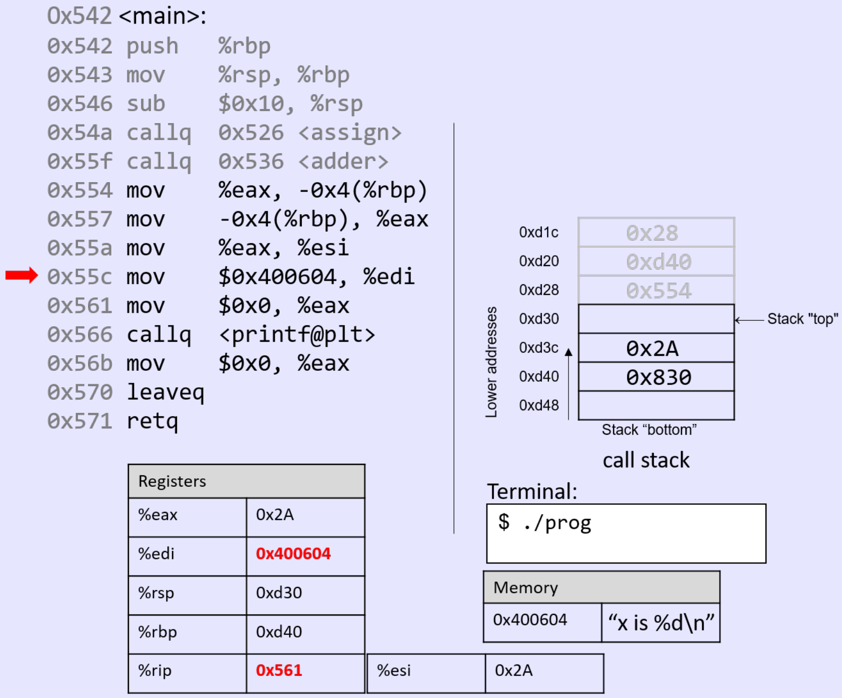
call stack

Registers			
%eax	0x2A		
%edi	1		
%rsp	0xd30		
%rbp	0xd40		
%rip	0x55c	%esi	0x2A

Terminal:

```
$ ./prog
```

21. Now we want to retrieve the first argument of the print function. The address at `$0x400604` is some address in the code segment memory that holds the string `"x is %d\n"`.



```
   0x542 <main>:
   0x542 push   %rbp
   0x543 mov    %rsp, %rbp
   0x546 sub    $0x10, %rsp
   0x54a callq  0x526 <assign>
   0x55f callq  0x536 <adder>
   0x554 mov    %eax, -0x4(%rbp)
   0x557 mov    -0x4(%rbp), %eax
   0x55a mov    %eax, %esi
➡ 0x55c mov    $0x400604, %edi
   0x561 mov    $0x0, %eax
   0x566 callq  <printf@plt>
   0x56b mov    $0x0, %eax
   0x570 leaveq
   0x571 retq
```

Registers	
%eax	0x2A
%edi	0x400604
%rsp	0xd30
%rbp	0xd40
%rip	0x561

%esi	0x2A

0xd1c	0x28
0xd20	0xd40
0xd28	0x554
0xd30	
0xd3c	0x2A
0xd40	0x830
0xd48	

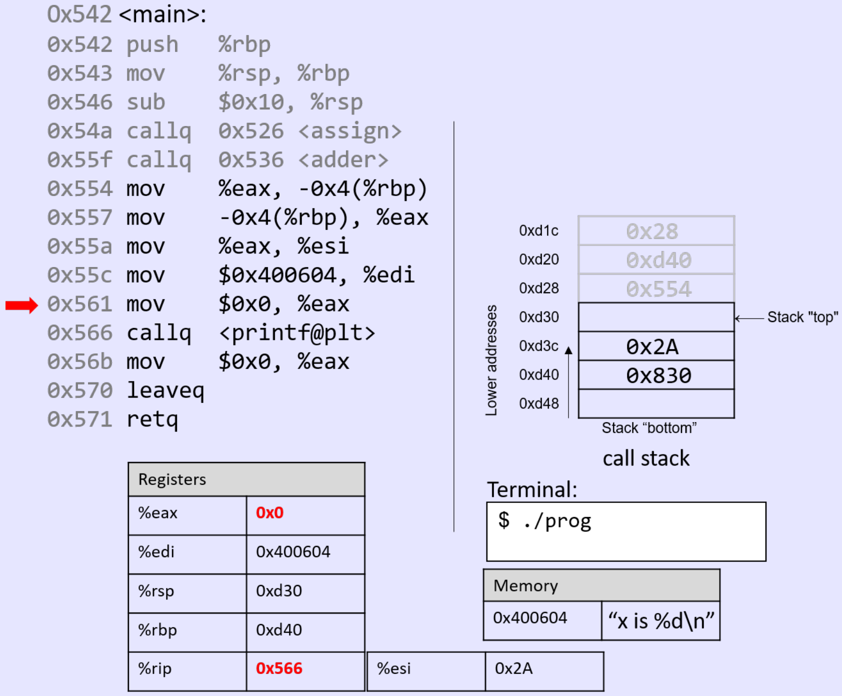
call stack

Terminal:

```
$ ./prog
```

Memory	
0x400604	"x is %d\n"

22. Then we move 0 into the `%eax` register to clear it.



```
   0x542 <main>:
   0x542 push   %rbp
   0x543 mov    %rsp, %rbp
   0x546 sub    $0x10, %rsp
   0x54a callq  0x526 <assign>
   0x55f callq  0x536 <adder>
   0x554 mov    %eax, -0x4(%rbp)
   0x557 mov    -0x4(%rbp), %eax
   0x55a mov    %eax, %esi
   0x55c mov    $0x400604, %edi
➡ 0x561 mov    $0x0, %eax
   0x566 callq  <printf@plt>
   0x56b mov    $0x0, %eax
   0x570 leaveq
   0x571 retq
```

Registers	
%eax	0x0
%edi	0x400604
%rsp	0xd30
%rbp	0xd40
%rip	0x566

%esi	0x2A

0xd1c	0x28
0xd20	0xd40
0xd28	0x554
0xd30	
0xd3c	0x2A
0xd40	0x830
0xd48	

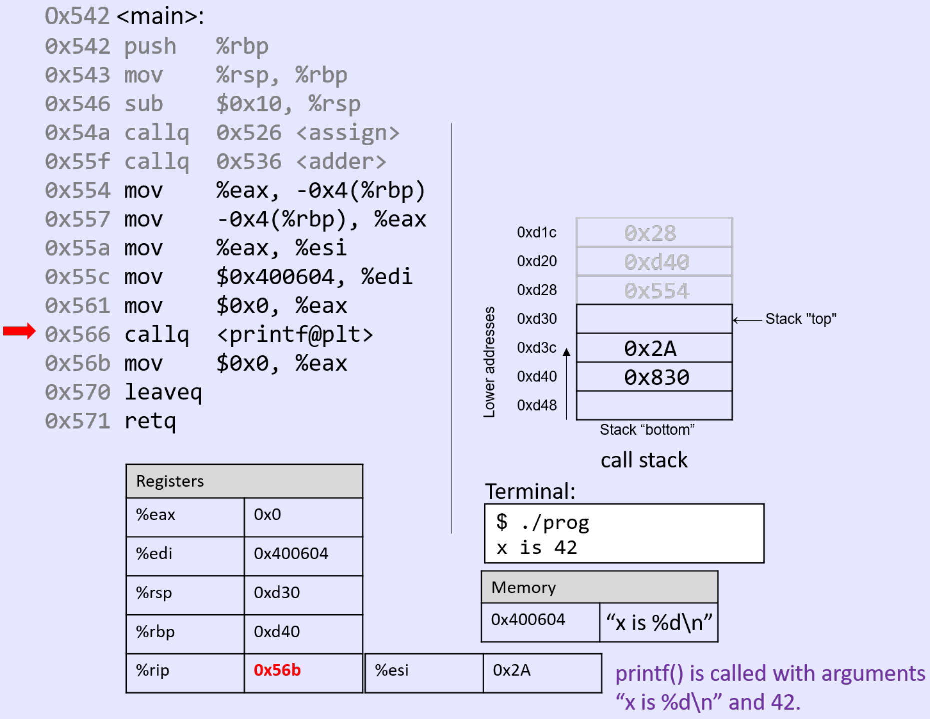
call stack

Terminal:

```
$ ./prog
```

Memory	
0x400604	"x is %d\n"

23. We then call the `printf` function, which we won't trace through but it outputs to stdout.



```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```

Registers	
%eax	0x0
%edi	0x400604
%rsp	0xd30
%rbp	0xd40
%rip	0x56b

%esi	0x2A

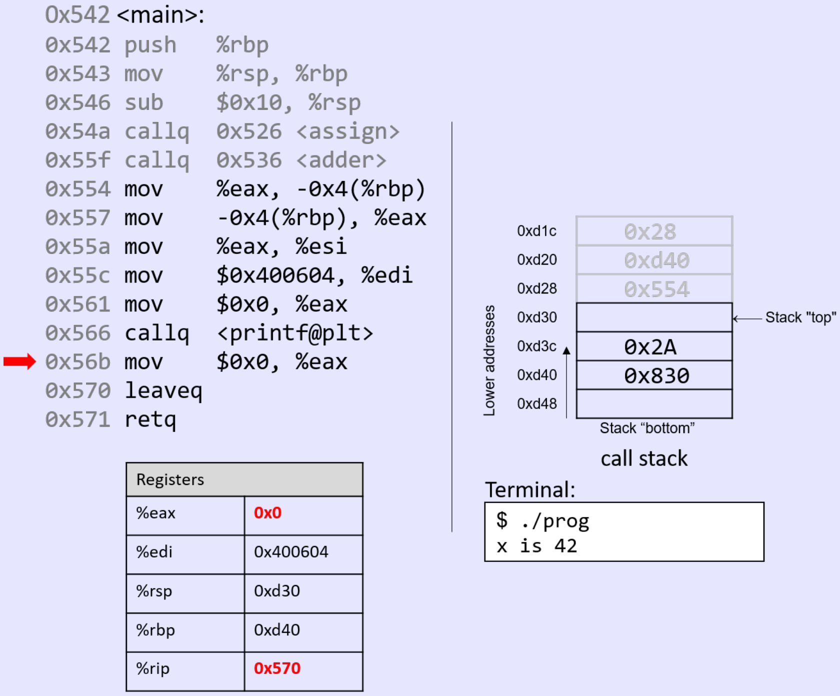
Terminal:

```
$ ./prog
x is 42
```

Memory	
0x400604	“x is %d\n”

printf() is called with arguments “x is %d\n” and 42.

24. The print function might have returned something, but we don't care. We want to main function to return 0, so we move 0 into `%eax`.



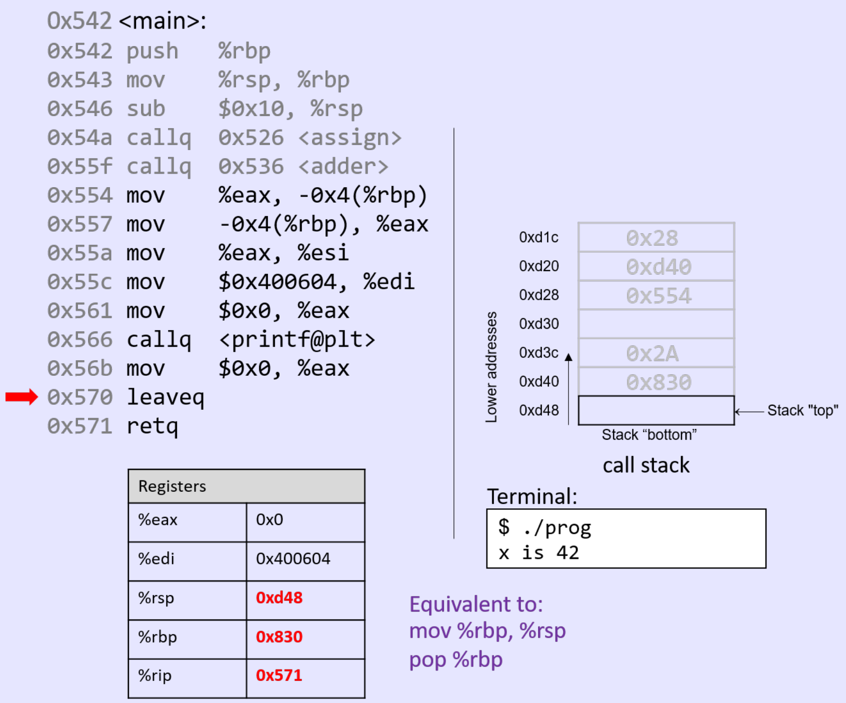
```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```

Registers	
%eax	0x0
%edi	0x400604
%rsp	0xd30
%rbp	0xd40
%rip	0x570

Terminal:

```
$ ./prog
x is 42
```

25. Finally we execute `leaveq`, which prepares the stack for returning from the function call. It essentially moves the base pointer back to the stack pointer and then pops the base pointer off the stack. The new `%rbp` is the original base pointer of whatever was outside the main function, `0x830`.



```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```

Registers	
%eax	0x0
%edi	0x400604
%rsp	0xd48
%rbp	0x830
%rip	0x571

0xd1c	0x28
0xd20	0xd40
0xd28	0x554
0xd30	
0xd3c	0x2A
0xd40	0x830
0xd48	

Terminal:

```
$ ./prog
x is 42
```

Equivalent to:
mov %rbp, %rsp
pop %rbp

26. Finally, we execute `retq`, which pops the return address off the stack and puts it into `%rip`.

We have omitted the details of caller and callee saved registers, but they do exist and are important for the general implementations.

For arrays, there's not anything new here. Let's go over some code and follow through it.

```c
int sumArray(int *array, int length) {
  int i, total = 0;
  for (i = 0; i < length; i++) {
    total += array[i];
  }
  return total;
}
```

This function takes the address of an array and the length of it and sums up all the elements in the array.

```
0x400686 <+0>: push %rbp                  # save %rbp
0x400687 <+1>: mov  %rsp,%rbp             # update %rbp (new stack frame)
0x40068a <+4>: mov  %rdi,-0x18(%rbp)      # copy array to %rbp-0x18
0x40068e <+8>: mov  %esi,-0x1c(%rbp)      # copy length to %rbp-0x1c
0x400691 <+11>:   movl $0x0,-0x4(%rbp)       # copy 0 to %rbp-0x4 (total)
0x400698 <+18>:   movl $0x0,-0x8(%rbp)       # copy 0 to %rbp-0x8 (i)
0x40069f <+25>:   jmp  0x4006be <sumArray+56> # goto <sumArray+56>
0x4006a1 <+27>:   mov  -0x8(%rbp),%eax       # copy i to %eax
0x4006a4 <+30>:   cltq                       # convert i to a 64-bit integer
0x4006a6 <+32>:   lea  0x0(,%rax,4),%rdx     # copy i*4 to %rdx
0x4006ae <+40>:   mov  -0x18(%rbp),%rax      # copy array to %rax
0x4006b2 <+44>:   add  %rdx,%rax             # compute array+i*4, store in %rax
0x4006b5 <+47>:   mov  (%rax),%eax           # copy array[i] to %eax
0x4006b7 <+49>:   add  %eax,-0x4(%rbp)       # add %eax to total
0x4006ba <+52>:   addl $0x1,-0x8(%rbp)       # add 1 to i (i+=1)
0x4006be <+56>:   mov  -0x8(%rbp),%eax       # copy i to %eax
```

```
0x4006c1 <+59>:   cmp  -0x1c(%rbp),%eax       # compare i to length
0x4006c4 <+62>:   jl   0x4006a1 <sumArray+27> # if i<length goto <sumArray+27>
0x4006c6 <+64>:   mov  -0x4(%rbp),%eax        # copy total to %eax
0x4006c9 <+67>:   pop  %rbp                   # prepare to leave the function
0x4006ca <+68>:   retq                        # return total
```