Assembly

Muchang Bahng

Fall 2024

Contents

There are many assembly languages out there and various syntaxes. *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. In *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. In here, we will talk about the three most popular architectures.

> **Definition 0.1 (x86)**
>
> **x86 Assembly** is 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.
> 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.
>
> ──────────────
> [a]https://en.wikipedia.org/wiki/X86_instruction_listings

> **Definition 0.2 (ARM)**
>
> **ARM Assembly** is 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. It is mainly in phones, tablets, laptops.

> **Definition 0.3 (RISC-V)**
>

A debatable 4th mainstream one is the MIPs assembly, which is based off of the MIPS RISC archiecture used in embedded systems such as digital home and networking equipment. Historically through, there are many many more variants. *PowerPC assembly* is 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. *6502 Assembly* is used in many early microcomputers and gaming consoles. Utilizes a syntax unique to the 6502 processor, not following AT&T or Intel conventions. *Z80 Assembly* is 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.

We begin with ARM64 because first, I use it on my Macbook M3, and second, ARM is usually simpler than x86. 64-bit ARM is significantly different from 32-bit ARM since obviously the CPU registers are 64-bits wide and perform 64-bit integer arithmetic.

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.

1   ARM Data Movement

At this point (assuming you are going through my computer science notes in order), we have encountered our first lexical computer language. We aren't just describing things with psuedocode like we did with architecture, and we aren't relying on hardware-like systems like circuits or Conway's game of life here. This extra level of abstraction is nice to work with, but in order to fully appreciate it, we must know how to convert assembly into machine code. As we have seen, this is done in two steps.

1. *Assemblers* convert them into object files.

2. *Linkers* use a relocation table to convert them into executables, written in machine code.

This is essentially a translation from one language into another, and to do this, we might want to have some organization in our code. Therefore, we divide .s files into *sections*. Furthermore, we might want to include instructions that tell the assembler—not the CPU—how to process your code, analogous to preprocessing text or tuning parameters for translation.

Both sections and directives have a period . at the front of their name, so you must tell them apart by context.

> **Definition 1.1 (Section)**
>
> In order for assemblers and linkers to interpret your programs, we must organize them into **sections**. Each section—specified by the distinctive . at the front of its name—specifies the following non-exhaustive list of properties.
>   1. The read/write/executable permissions.
>   2. How data is initialized.

> **Example 1.1 (Must-Know Sections)**
>
> The main sections you should be familiar with are
>   1. `.text` (read+execute). This is where you write your code.
>   2. `.data` (read+write). This is where you store data and memory.
>   3. `.rodata` (read). Stores constant data that should not be modified during program execution.
>   4. `.bss` (read+write). Zero-initialized and stores uninitialized variables.

You can also create your own sections.

1.1   Loading

The first thing you should know about are registers. Here are the register conventions for ARM64.

> **Definition 1.2 (ARM64 Registers)**
>
> A 64-bit program on an ARM processor has access to 31-general purpose registers, a program counter, and a stack pointer (aka a combination zero register).
>   1. `X0 - X30`. These 31 registers are general purpose. You can use them for anything you want, though there are some standards.
>   2. `SP, XZR`. The link register. If you call a function, this register will be used to hold the return address.
>   3. `PC`. Program counter. The memory address of the currently executing instruction.
> All the X registers can be operated on as 32-bit registers by referring to them as W0–W30 and WZR. When we do this, the instruction will use the lower 32 bits of the register and set the upper 32 bits to zero. Using 32 bits saves memory, since you only use 4 bytes rather than 8 bytes for each quantity saved. Some Apple specific things:
>   1. Apple reserves X18 for its own use. Do not use this register.

2. The frame pointer register (FP, X29) must always address a valid frame record. This is for backtraces.

Definition 1.3 (Data Movement Operations)

```
1   # Basic move operations
2   mov x0, x1              // Move register to register
3   mov x0, #42             // Move immediate to register
4
5   # Move with zero/not/keep
6   movz x0, #0x1234        // Move immediate, zero other bits
7   movn x0, #0x1234        // Move NOT of immediate
8
9   # Conditional moves (covered in logical section)
10  csel x0, x1, x2, eq    // Conditional select
11  csinc x0, x1, x2, ne   // Conditional select and increment
12
13  # Register to register with operations
14  sxtb x0, w1            // Sign extend byte to 64-bit
15  sxth x0, w1            // Sign extend halfword to 64-bit
16  sxtw x0, w1            // Sign extend word to 64-bit
17  uxtb w0, w1            // Zero extend byte to 32-bit
18  uxth w0, w1            // Zero extend halfword to 32-bit
```

Definition 1.4 (Exit Codes)

Exit codes are values that represent the status of a program upon termination. It is usually the number residing in x0 and can take in value between 0 and 255, inclusive. Any other numbers will be truncated to its 8 LSBs.

Definition 1.5 (Basic Load Operations)

```
1   ldr x0, [x1]           // Load 64-bit from [x1]
2   ldr w0, [x1]           // Load 32-bit from [x1]
3   ldrh w0, [x1]          // Load 16-bit halfword (zero extend)
4   ldrb w0, [x1]          // Load 8-bit byte (zero extend)
5
6   # Sign-extending loads
7   ldrsw x0, [x1]         // Load 32-bit, sign extend to 64-bit
8   ldrsh x0, [x1]         // Load 16-bit, sign extend to 64-bit
9   ldrsh w0, [x1]         // Load 16-bit, sign extend to 32-bit
10  ldrsb x0, [x1]         // Load 8-bit, sign extend to 64-bit
11  ldrsb w0, [x1]         // Load 8-bit, sign extend to 32-bit
```

Definition 1.6 (Basic Store Operations)

```
1   str x0, [x1]           // Store 64-bit to [x1]
2   str w0, [x1]           // Store 32-bit to [x1]
3   strh w0, [x1]          // Store 16-bit halfword to [x1]
4   strb w0, [x1]          // Store 8-bit byte to [x1]
```

Definition 1.7 (Addressing Modes)

```
1   # Immediate offset
2   ldr x0, [x1, #8]       // Load from [x1 + 8]
3   str x0, [x1, #16]      // Store to [x1 + 16]
4
5   # Register offset
6   ldr x0, [x1, x2]       // Load from [x1 + x2]
7   ldr x0, [x1, x2, lsl #3]  // Load from [x1 + (x2 << 3)]
8
9   # Pre-indexed (update base register before)
10  ldr x0, [x1, #8]!      // x1 = x1 + 8, then load from [x1]
11  str x0, [x1, #-16]!    // x1 = x1 - 16, then store to [x1]
12
13  # Post-indexed (update base register after)
14  ldr x0, [x1], #8       // Load from [x1], then x1 = x1 + 8
15  str x0, [x1], #16      // Store to [x1], then x1 = x1 + 16
```

Definition 1.8 (Pair Load/Store Operations)

```
1   # Load/store register pairs
2   ldp x0, x1, [x2]       // Load pair: x0=[x2], x1=[x2+8]
3   stp x0, x1, [x2]       // Store pair: [x2]=x0, [x2+8]=x1
4
5   # With immediate offset
6   ldp x0, x1, [x2, #16]  // Load pair from [x2+16], [x2+24]
7   stp x0, x1, [x2, #32]  // Store pair to [x2+32], [x2+40]
8
9   # Pre/post-indexed pairs
10  ldp x0, x1, [x2, #16]!    // x2+=16, then load pair
11  stp x0, x1, [x2], #16     // Store pair, then x2+=16
12
13  # Mixed register sizes
14  ldp w0, w1, [x2]       // Load 32-bit pair
15  stp w0, w1, [x2]       // Store 32-bit pair
```

Definition 1.9 (Atomic and Exclusive Operations)

```
1   # Load/store exclusive
2   ldxr x0, [x1]          // Load exclusive 64-bit
3   stxr w2, x0, [x1]      // Store exclusive 64-bit (w2 = status)
4   ldxrh w0, [x1]         // Load exclusive 16-bit
5   stxrh w2, w0, [x1]     // Store exclusive 16-bit
6   ldxrb w0, [x1]         // Load exclusive 8-bit
7   stxrb w2, w0, [x1]     // Store exclusive 8-bit
8
9   # Clear exclusive monitor
10  clrex                  // Clear exclusive access monitor
11
12  # Load/store exclusive pairs
13  ldxp x0, x1, [x2]      // Load exclusive pair
14  stxp w3, x0, x1, [x2] // Store exclusive pair (w3 = status)
```

Definition 1.10 (PC-Relative Addressing)

```
1   # Address generation
2   adr x0, label          // Load address of label (PC + offset)
3   adrp x0, label         // Load page address of label
4
5   # PC-relative loads
6   ldr x0, =value         // Load literal (assembler places in literal pool)
7   ldr x0, label          // Load from label address
8
9   # Combined page + offset addressing
10  adrp x0, symbol@PAGE
11  add x0, x0, symbol@PAGEOFF
12  ldr x1, [x0]           // Load from symbol
```

Definition 1.11 (Advanced Load/Store)

```
1   # Load with acquire, store with release (memory ordering)
2   ldar x0, [x1]          # Load acquire
3   stlr x0, [x1]          # Store release
4   ldarb w0, [x1]         # Load acquire byte
5   stlrb w0, [x1]         # Store release byte
6   ldarh w0, [x1]         # Load acquire halfword
7   stlrh w0, [x1]         # Store release halfword
8
9   # Prefetch operations
10  prfm pldl1keep, [x0]   # Prefetch for load, L1 cache, keep
11  prfm pstl1strm, [x0, #64]  # Prefetch for store, L1, streaming
12
13  # Non-temporal loads/stores
14  ldnp x0, x1, [x2]      # Load pair non-temporal
15  stnp x0, x1, [x2]      # Store pair non-temporal
```

1.2   Arithmetic

Definition 1.12 (Addition)

```
1   add x0, x1, x2      # x0 = x1 + x2 (64-bit)
2   add w0, w1, w2      # w0 = w1 + w2 (32-bit)
3   add x0, x1, #42     # x0 = x1 + 42 (immediate)
4   adds x0, x1, x2     # Add and set flags
5   adc x0, x1, x2      # Add with carry
6   adcs x0, x1, x2     # Add with carry and set flags
```

Definition 1.13 (Subtraction)

```
1   sub x0, x1, x2      # x0 = x1 - x2
2   sub w0, w1, w2      # 32-bit subtract
3   sub x0, x1, #42     # x0 = x1 - 42
```

```
4  subs x0, x1, x2     # Subtract and set flags
5  sbc x0, x1, x2      # Subtract with carry
6  sbcs x0, x1, x2     # Subtract with carry and set flags
7  neg x0, x1          # x0 = -x1 (negate)
8  negs x0, x1         # Negate and set flags
```

Definition 1.14 (Multiplication)

```
1  mul x0, x1, x2      # x0 = x1 * x2 (low 64 bits)
2  smull x0, w1, w2    # Signed multiply 32 to 64 bit
3  umull x0, w1, w2    # Unsigned multiply 32 to 64 bit
4  smulh x0, x1, x2    # Signed multiply high (upper 64 bits)
5  umulh x0, x1, x2    # Unsigned multiply high
6  madd x0, x1, x2, x3 # x0 = x3 + (x1 * x2) (multiply-add)
7  msub x0, x1, x2, x3 # x0 = x3 - (x1 * x2) (multiply-subtract)
```

Definition 1.15 (Division)

```
1  sdiv x0, x1, x2     # x0 = x1 / x2 (signed)
2  udiv x0, x1, x2     # x0 = x1 / x2 (unsigned)
```

Definition 1.16 (Multiply-Accumulate)

```
1  madd x0, x1, x2, x3   # x0 = x3 + (x1 * x2)
2  msub x0, x1, x2, x3   # x0 = x3 - (x1 * x2)
3  smaddl x0, w1, w2, x3 # x0 = x3 + (w1 * w2) signed 32 to 64
4  smsubl x0, w1, w2, x3 # x0 = x3 - (w1 * w2) signed 32 to 64
5  umaddl x0, w1, w2, x3 # x0 = x3 + (w1 * w2) unsigned 32 to 64
6  umsubl x0, w1, w2, x3 # x0 = x3 - (w1 * w2) unsigned 32 to 64
```

Definition 1.17 (Bitwise Operations)

```
1  and x0, x1, x2      # Bitwise AND
2  orr x0, x1, x2      # Bitwise OR
3  eor x0, x1, x2      # Bitwise XOR (exclusive OR)
4  bic x0, x1, x2      # Bit clear (x0 = x1 & ~x2)
5  orn x0, x1, x2      # OR NOT (x0 = x1 | ~x2)
6  eon x0, x1, x2      # XOR NOT (x0 = x1 ^ ~x2)
7  mvn x0, x1          # Move NOT (x0 = ~x1)
```

Definition 1.18 (Shift Operations)

```
1  lsl x0, x1, #5      # Logical shift left by 5
2  lsr x0, x1, #3      # Logical shift right by 3
```

```
3   asr x0, x1, #2        # Arithmetic shift right by 2
4   ror x0, x1, #4        # Rotate right by 4
```

Definition 1.19 (Combined Operations)

```
1   # Add with shifted register
2   add x0, x1, x2, lsl #3    # x0 = x1 + (x2 << 3)
3   sub x0, x1, x2, asr #2    # x0 = x1 - (x2 >> 2)
4
5   # Bitwise with shifted register
6   and x0, x1, x2, ror #4    # x0 = x1 & (x2 rotated right 4)
7   orr x0, x1, x2, lsl #1    # x0 = x1 | (x2 << 1)
```

Definition 1.20 (Comparison Operations)

```
1   cmp x1, x2            # Compare (sets flags, x1 - x2)
2   cmn x1, x2            # Compare negative (sets flags, x1 + x2)
3   tst x1, x2            # Test (sets flags, x1 & x2)
```

Definition 1.21 (Conditional Operations)

```
1   csel x0, x1, x2, eq    # x0 = (condition) ? x1 : x2
2   csinc x0, x1, x2, ne   # x0 = (condition) ? x1 : x2+1
3   csinv x0, x1, x2, gt   # x0 = (condition) ? x1 : ~x2
4   csneg x0, x1, x2, lt   # x0 = (condition) ? x1 : -x2
```

Definition 1.22 (Absolute Value and Min/Max)

```
1   # Using conditional select for abs(x1)
2   cmp x1, #0
3   csneg x0, x1, x1, ge    # x0 = (x1 >= 0) ? x1 : -x1
4
5   # Min/max using conditional select
6   cmp x1, x2
7   csel x0, x1, x2, lt     # x0 = min(x1, x2)
8   csel x0, x1, x2, gt     # x0 = max(x1, x2)
```

Definition 1.23 (Increment/Decrement)

```
1   add x0, x0, #1       # Increment by 1
2   sub x0, x0, #1       # Decrement by 1
3   adds x0, x0, #1      # Increment and set flags
4   subs x0, x0, #1      # Decrement and set flags
```

Definition 1.24 (Modulo Operation)

```
1  # x0 = x1 % x2 (signed) - No direct instruction
2  sdiv x3, x1, x2     # x3 = x1 / x2
3  msub x0, x3, x2, x1 # x0 = x1 - (x3 * x2)
```

Definition 1.25 (Power of 2 Operations)

```
1  # Multiply by power of 2
2  lsl x0, x1, #3      # x0 = x1 * 8 (2^3)
3
4  # Divide by power of 2
5  lsr x0, x1, #2      # x0 = x1 / 4 (unsigned)
6  asr x0, x1, #2      # x0 = x1 / 4 (signed)
```

1.3   Logical Operations

Definition 1.26 (Bit Field Operations)

```
1  sbfx x0, x1, #5, #8   # Signed bit field extract
2  ubfx x0, x1, #5, #8   # Unsigned bit field extract
3  sbfiz x0, x1, #5, #8  # Signed bit field insert zeros
4  ubfiz x0, x1, #5, #8  # Unsigned bit field insert zeros
5  bfi x0, x1, #5, #8    # Bit field insert
6  bfxil x0, x1, #5, #8  # Bit field extract and insert low
```

Definition 1.27 (Bit Manipulation)

```
1  rbit x0, x1          # Reverse bits
2  rev x0, x1           # Reverse bytes (64-bit)
3  rev32 x0, x1         # Reverse bytes in 32-bit words
4  rev16 x0, x1         # Reverse bytes in 16-bit halfwords
5  clz x0, x1           # Count leading zeros
6  cls x0, x1           # Count leading sign bits
```

Definition 1.28 (Advanced Logical Operations)

```
1  # Bitwise operations with immediates
2  and x0, x1, #0xFF00   # AND with immediate mask
3  orr x0, x1, #0x0F0F   # OR with immediate mask
4  eor x0, x1, #0xAAAA   # XOR with immediate mask
5
6  # Test and branch on bit
7  tbz x1, #5, label     # Test bit zero and branch
8  tbnz x1, #5, label    # Test bit non-zero and branch
```

Definition 1.29 (Conditional Logic)

```
1   ccmp x1, x2, #0, eq   # Conditional compare
2   ccmn x1, x2, #0, ne   # Conditional compare negative
3   cset x0, eq           # Conditional set (x0 = condition ? 1 : 0)
4   csetm x0, ne          # Conditional set mask (x0 = condition ? -1 : 0)
5   cinc x0, x1, gt       # Conditional increment
6   cinv x0, x1, lt       # Conditional invert
7   cneg x0, x1, ge       # Conditional negate
```

Definition 1.30 (Logical Shift Operations)

```
1    # Standalone shift operations
2    lsl x0, x1, x2        # Logical shift left by register
3    lsr x0, x1, x2        # Logical shift right by register
4    asr x0, x1, x2        # Arithmetic shift right by register
5    ror x0, x1, x2        # Rotate right by register
6
7    # Shift with immediate
8    lsl x0, x1, #5        # Logical shift left by immediate
9    lsr x0, x1, #3        # Logical shift right by immediate
10   asr x0, x1, #2        # Arithmetic shift right by immediate
11   ror x0, x1, #4        # Rotate right by immediate
```

Definition 1.31 (Pattern Operations)

```
1    # Extract and duplicate patterns
2    extr x0, x1, x2, #8   # Extract from register pair
3    dup v0.8b, w1         # Duplicate scalar to vector
4
5    # Bit pattern generation
6    movz x0, #0x1234      # Move with zero (clear other bits)
7    movn x0, #0x1234      # Move with NOT (invert pattern)
8    movk x0, #0x5678, lsl #16  # Move and keep (insert pattern)
```

1.4   Assembling and Disassembling

Definition 1.32 (Instruction Syntax)

Every ARM instruction—regardless of whether we're in 32-bit or 64-bit ARM—can be fit into 32 bits of memory. The fixed-length variable of this is good for speed.

31	30	29	28-24	23-22	21	20-16	15-10	9-5	4-0
Bits	Opcode	Set Condition Code	Opcode	Shift	0	Rm	Imm	Rn	Rd

Figure 1: Instruction encoding format.

1. *Bits.* If this bit is 0, then any registers are interpreted as the 32-bit W version. If 1, then they are the full 64-bit version of the register.[a]

2. *Opcode.* which instruction are we performing, e.g. `ADD` or `MUL`.
3. *Shift.* These two bits specify shifting operations that could be applied to the data.
4. *Set Condition Code.* A single bit indicating if this instruction should update any condition flags. If we don't want the result of this instruction to affect following branch instructions, we set it to 0.
5. *Rm, Rn.* Operand registers to use as input.
6. *Rd.* Destination register, i.e. where to put the result of whatever this instruction does.
7. *Imm6.* An immediate operand which is usually a small bit of data that you can specify directly in the instruction. So, if you want to add 1 to a register, you could have this as 1, rather than putting 1 in another register and adding the two registers. These are usually the bits left over after everything else is specified.

----

[a]You cannot mix W and Z bits in the same instruction!

A **dump** refers to a representation of the contents and structure of an object file or memory at a specific point in time. Once a file is assembled it's almost impossible to read. Fortunately, there are some nice shell commands to help us.

Definition 1.33 (`objdump`)

Taken from the man pages, the **objdump** utility prints the contents of object files and final linked images named on the command line.
1. `-d`, `-disassemble`. Disassemble all executable sections found in the input files. On some architectures (AArch64, PowerPC, x86), all known instructions are disassembled by default.
2. `-t`, `-syms`. Display the symbol table.

Example 1.2 (`objdump -d`)

```
1   .text
2     .globl _main
3     _main:
4         mov x0, #0x1
5         b _exit
```

Figure 2

```
1  > objdump -d hello
2
3  hello:  file format mach-o arm64
4
5  Disassembly of section __TEXT,__text:
6
7  00000001000003c0 <_main>:
8  1000003c0: d2800020      mov     x0, #0x1                  ; =1
9  1000003c4: 14000001      b       0x1000003c8 <_exit+0x1000003c8>
10
11 Disassembly of section __TEXT,__stubs:
12
13 00000001000003c8 <__stubs>:
14 1000003c8: 90000030      adrp    x16, 0x100004000 <_exit+0x100004000>
15 1000003cc: f9400210      ldr     x16, [x16]
16 1000003d0: d61f0200      br      x16
```

Figure 3

The mov command has the hex command `d2800020`, which translates in binary to

```
1  1101 0010 1000 0000 0000 0000 0010 0000
```

1. The first bit is 1, meaning use the 64-bit version of the registers, in this case X0 rather than W0.
2. The third bit is 0, which means that this instruction doesn't set any flags that would affect conditional instructions.
3. The second bit combined with the fourth to ninth bits make up the opcode for this MOV instruction. This is move wide immediate, meaning it contains a 16-bit immediate value.
4. The next 2 bits of 0 indicate there is no shift operation involved.
5. The next 16 bits are the immediate value which is 1.
6. The last 5 bits are the register to load. These are 0 since we are loading register X0.

Definition 1.34 (xxd)

Taken from the man pages, the **xxd** utility makes hex dump of a given file or standard input. It can also convert a hex dump back to its original binary form.

1.5 Directive

Definition 1.35 (Directive)

A **directive** is an instruction to the assembler. It tells the assembler how to process your code, but doesn't generate machine instructions, making it like commands for the assembler and not the CPU.
   1.

Example 1.3 (Symbol Control)

```
1  .globl _main       // Make symbol globally visible
2  .local helper      // Keep symbol local to this file
3  .extern _printf    // Reference external symbol
```

```
4   .weak _optional    // Make symbol weakly defined
```

Example 1.4 (Data Creation)

```
1   .byte 0x42         // Create 1-byte value
2   .word 42           // Create 4-byte value
3   .quad 42           // Create 8-byte value
4   .asciz "hello"     // Create null-terminated string
5   .ascii "hello"     // Create string (no null terminator)
6   .space 64          // Reserve 64 bytes of space
7   .fill 10, 4, 0     // Fill 10 4-byte words with 0
```

Example 1.5 (Alignment)

```
1   .align 4           // Align to 4-byte boundary
2   .p2align 2         // Align to 2^2 = 4-byte boundary
3   .balign 16         // Align to 16-byte boundary
```

Example 1.6 (Section Control)

```
1   .text              // Switch to code section
2   .data              // Switch to data section
3   .bss               // Switch to uninitialized data section
4   .section my_custom // Create/switch to custom section
```

Example 1.7 (Conditional Assembly)

```
1   .ifdef DEBUG
2     .asciz "Debug build"
3   .else
4     .asciz "Release build"
5   .endif
```

Example 1.8 (Macros)

```
1   .macro SAVE_REGS
2     stp x29, x30, [sp, #-16]!
3     mov x29, sp
4   .endm
5
6   # Usage:
7   _my_function:
8     SAVE_REGS           // Expands to the macro content
9     // ... function body
```

```
10    ret
```

Example 1.9 ()

```
1
2
```

Example 1.10 ()

```
1    # DIRECTIVES (instructions to assembler):
2    .globl _main       # "Make this symbol global"
3    .align 4           # "Align the next thing to 4 bytes"
4    .asciz "hello"     # "Create a null-terminated string here"
5
6    # INSTRUCTIONS (actual CPU operations):
7    mov x0, #42        # CPU instruction: move 42 into x0
8    bl _printf         # CPU instruction: branch with link
9    ret                # CPU instruction: return
```

You open up your text editor on an M1 Mac, and every assembly program should start with this.

```
1    .globl _main
2    _main:
3      ...
4      b _exit
```

`.globl` is an assembler directive that makes the symbol `_main` globally visible to the linker. This allows other files/modules to reference this `_main` function. `b _exit` is a specific function that tell the program to shut down.

2 ARM Arithmetic Operations

3 ARM Control Flow

4   x86 Data Movement

Definition 4.1 (Data Types)

In x86,
   1. A **word** refers to

4.1   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 4.2 (Parameter Registers)

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

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

respectively.

Definition 4.3 (Return Register)

The return value of a function is stored in the

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

register.

Definition 4.4 (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 4.5 (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 4.6 (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.
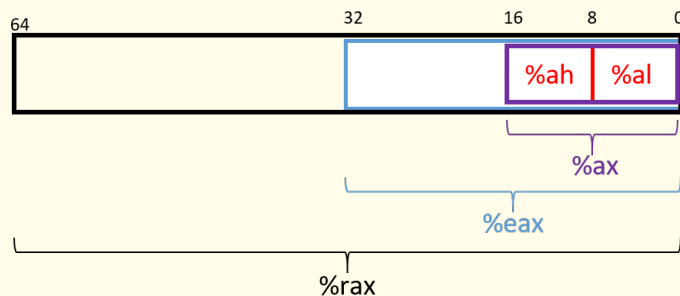
Figure 4: The names that refer to subsets of register `%rax`.

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

4.2 Addressing Modes

Example 4.1 (Immediate Addressing)

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

Example 4.2 (Normal Addressing)

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

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

Example 4.3 (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.

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

Example 4.4 (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

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

We see that

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

5   x86 Arithmetic Operations

> **Definition 5.1 (Size Specifier)**
>
> In x86 assembly, 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.

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.

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

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

```
1   %reg = %reg + Mem[address]      # add memory data to register
2   %reg = %reg - Mem[address]      # subtract memory data from register
3   %reg = %reg * Mem[address]      # multiply memory data to register
4   %reg = %reg / Mem[address]      # divide memory data from register
```

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

```
1   jmp label      # jump to label
2   je label       # jump to label if equal
3   jne label      # jump to label if not equal
4   jg label       # jump to label if greater
5   jl label       # jump to label if less
6   call label     # call a function
7   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 5.2 (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 5.3 (int)

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

Definition 5.4 (ret)

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

Example 5.1 (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;
}
```

Let's talk about moving instructions first.

Definition 5.5 (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

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

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 5.2 (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)

```
1   void swap(long *xp, long *yp) {
2     long t0 = *xp;
3     long t1 = *yp;
4     *xp = t1;
5     *yp = t0;
6   }
```

```
1   swap:
2     movq (%rdi), %rax
3     movq (%rsi), %rdx
4     movq %rdx, (%rdi)
5     movq %rax, (%rsi)
6     ret
```

Definition 5.6 (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 5.3 (Simple example with movz)

Take a look at the code below.

```
1   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.

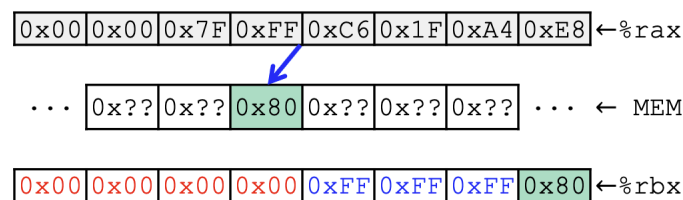| 0x?? | 0x?? | 0x?? | 0x?? | 0x?? | 0x?? | 0x?? | 0xFF | ←%rax |
| 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0xFF | ←%rbx |

Example 5.4 (Harder example with movs)

Take a look at the code below.

```
1   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
1   int add(int x) {
2     return x + 2;
3   }
4
5   int main() {
6     int a = 2;
7     int b = add(a);
8     return 0;
9   }
```

```
1   add:
2     movq %rdi, %rax
3     addq $2, %rax
4     ret
5   main:
6     movq $3, $rdi
7     call add
8     movq $0, %rax
9     ret
```

Figure 5: 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 5.7 (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 5.8 (Push and Pop)

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

$$\text{push\_ src} \qquad \text{rsp = rsp - 8; Mem[rsp] = src}$$
$$\text{pop\_ dest} \qquad \text{dest = Mem[rsp]; rsp = rsp + 8}$$

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 5.9 (Push and Pop)

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

$$\text{push\_ src} \qquad\qquad \text{rsp = rsp - 8; Mem[rsp] = src}$$

$$\text{pop\_ dest} \qquad\qquad \text{dest = Mem[rsp]; rsp = rsp + 8}$$

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

Definition 5.10 (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 5.11 (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 5.5 (Evaluating a Function)

Say that we have the following C code.

```
1   int adder2(int a) {
2     return a + 2;
3   }
4
5   int main() {
6     int x = 40;
7     x = adder2(x);
8     printf("x is: %d\n", x);
9     return 0;
10  }
```

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.

```
1  0000000000400526 <adder2>:
2   400526:        55                         push    %rbp
3   400527:        48 89 e5                   mov     %rsp,%rbp
4   40052a:        89 7d fc                   mov     %edi,-0x4(%rbp)
5   40052d:        8b 45 fc                   mov     -0x4(%rbp),%eax
6   400530:        83 c0 02                   add     $0x2,%eax
7   400533:        5d                         pop     %rbp
8   400534:        c3                         retq
```

Figure 6: 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.
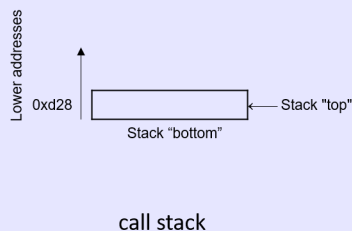
   (a) First, the `%eax` is filled with garbage, which are leftovers from previous programs that haven't been overwritten yet.
   (b) Second, the `%edi=0x28` since we have set `x=40` in `main`, before calling `adder2`, so it lingers on.
   (c) `%rsp=0xd28` since that is where the top of the stack is.
   (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**

Lower addresses

0xd20 | 0xd40 | ← Stack "top"
0xd28 |

Stack "bottom"

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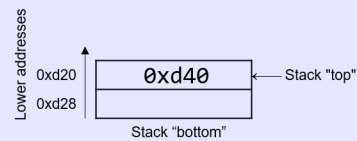
3. The reason we have pushed **%rbp** onto the stack is that we want to save it before it gets overwritten by this next execution. We basically move the value of **%rsp** into **%rbp**, and the **%rip** advances to the next instruction. **%rip** moves to the next instruction.

```
   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**

Lower addresses

0xd20 | 0xd40 | ← Stack "top"
0xd28 |

Stack "bottom"

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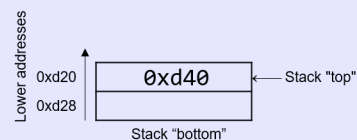
4. Now we want to take our first argument **%edi** and store it in memory. Note that since this is 4 bytes, we can move this value into memory that is 4 bytes below the stack (**-0x4(%rbp)**). Note that the storing the value of **%edi** into memory doesn't affect the stack pointer **%rsp**. As far as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

Lower addresses

0xd1c | 0x28 |
0xd20 | 0xd40 | ← Stack "top"
0xd28 |

Stack "bottom"

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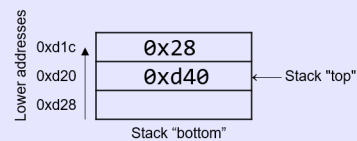
5. The next instruction simply goes into memory 4 bytes below the stack pointer, takes the value there, and stores it into **%eax**. This is the value of **%edi** that we just stored. This may seem redundant since we are making a round trip to memory and back to ultimately move the value

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**

Lower addresses

0xd1c	0x28
0xd20	0xd40	← Stack "top"
0xd28	

Stack "bottom"
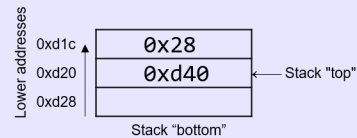
call stack

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**

Lower addresses

0xd1c	0x28
0xd20	0xd40	← Stack "top"
0xd28	

Stack "bottom"
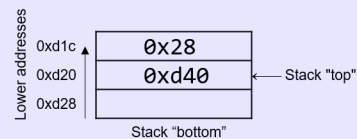
call stack

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**

Lower addresses

0xd1c	0x28
0xd20	0xd40
0xd28		← Stack "top"

Stack "bottom"
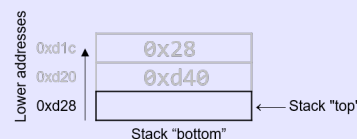
call stack

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:

```
1   push %rbp
2   mov %rsp, %rbp
```

and the following two at the end of a function

```
1   pop %rbp
2   retq
```

Now arithemtic operations are quite simple.

Definition 5.12 (Add, Subtract, Multiply)

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

$$\text{add\_ src, dest} \qquad\qquad \text{dest = dest + src}$$
$$\text{sub\_ src, dest} \qquad\qquad \text{dest = dest - src}$$

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

$$\text{imul\_ src, dest} \qquad\qquad \text{dest = dest * src}$$

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

Definition 5.13 (Increment, Decrement)

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

$$\text{inc\_ dest} \qquad\qquad \text{dest = dest + 1}$$
$$\text{dec\_ dest} \qquad\qquad \text{dest = dest - 1}$$

Definition 5.14 (Negative)

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

$$\text{neg\_ dest} \qquad\qquad \text{dest = -dest}$$

Example 5.6 (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.

```
1   int main() {
2       int a = 4, b = 8;
3       int c = a + b;
4       return c;
5   }
```

```
1   main:
2       movq $4, %rax
3       movq $8, %rbx
4       addq %rbx, %rax
5       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 5.15 (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

$$\text{lea\_ (src1, src2), dest} \qquad\qquad \text{dest = src1 + src2}$$
$$\text{lea\_ (src1, src2, scale), dest} \qquad\qquad \text{dest = src1 + src2*scale}$$
$$\text{lea\_ const(src1, src2), dest} \qquad\qquad \text{dest = src1 + src2 + const}$$
$$\text{lea\_ const(src1, src2, scale), dest} \qquad \text{dest = src1 + src2*scale + const}$$

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

Definition 5.16 (Bitwise)

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

$$\text{and src, dest} \qquad\qquad \text{dest = dest \& src}$$
$$\text{or src, dest} \qquad\qquad \text{dest = dest | src}$$
$$\text{xor src, dest} \qquad\qquad \text{dest = dest \^{} src}$$
$$\text{neg dest} \qquad\qquad \text{dest = -dest}$$
$$\text{not dest} \qquad\qquad \text{dest = \sim dest}$$

Definition 5.17 (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.

$$\text{sal src, dest} \qquad\qquad \text{dest = dest} \ll \text{src}$$
$$\text{shr src, dest} \qquad\qquad \text{dest = dest} \gg \text{src}$$

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.

$$\text{sar src, dest} \qquad\qquad \text{dest = dest} \gg \text{src}$$
$$\text{shl src, dest} \qquad\qquad \text{dest = dest} \ll \text{src}$$

Example 5.7 (Harder Arithmetic Example)

The following two codes are equivalent.

```
1  long arith(long x, long y, long z) {       1  arith:
2      long t1 = x + y;                        2      # rax/t1 = x + y
3      long t2 = z + t1;                       3      leaq  (%rdi, %rsi), %rax
4      long t3 = x + 4;                        4      # rax/t2 = z + t1
5      long t4 = y * 48;                       5      addq  %rdx, %rax
6      long t5 = t3 + t4;                      6      #rdx = 3 * y
7      long rval = t2 * t5;                    7      leaq  (%rsi, %rsi, 2), %rdx
8      return rval;                            8      #rdx/t4 = (3*y) * 16
9  }                                           9      salq  $4, %rdx
10  .                                         10      #rcx/t5 = x + t4 + 4
11  .                                         11      leaq  4(%rdi, %rdi), %rcx
12  .                                         12      # rax/rval = t5 * t2
13  .                                         13      imulq %rcx, %rax
14  .                                         14      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 5.18 (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 5.8 (Zero Flag)**
>
> If the code below was just run, then ZF would be set to 1.
>
> ```
> 1  movq $2, %rax
> 2  subq $2, %rax
> ```

> **Example 5.9 (Sign Flag)**
>
> If the code below was just run, then SF would be set to 1.
>
> ```
> 1  movq $2, %rax
> 2  subq $4, %rax
> ```

Example 5.10 (Overflow Flag)

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

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

```
1   movq 0x8000000000000000, %rax
2   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 5.11 (Carry Flag)

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

```
1   movq $0xffffffffffffffff, %rax
2   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 5.19 (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 5.20 (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 5.12 (Compare)

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

$$\%bl - \%al = 0x81 - 0x80 = 0x81 + \sim 0x80 + 1 = 0x81 + 0x7F + 1 = 0x101 = 0x01 \quad (5)$$

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 5.21 (Set)

We can then talk about conditional moves and jumps.

Definition 5.22 (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 5.23 (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 7: 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 8: All jump instructions

Definition 5.24 (int)

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

Definition 5.25 (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 5.26 (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.

```
1  int getSmallest(int x, int y) {
2    int smallest;
3    if ( x > y ) { //if (conditional)
4      smallest = y; //then statement
5    }
6    else {
7      smallest = x; //else statement
8    }
9    return smallest;
10 }
11 .
12 .
13 .
14 .
15 .
```

```
1  int getSmallest(int x, int y) {
2    int smallest;
3
4    if (x <= y ) { //if (!conditional)
5      goto else_statement;
6    }
7    smallest = y; //then statement
8    goto done;
9
10 else_statement:
11   smallest = x; //else statement
12
13 done:
14   return smallest;
15 }
```

Figure 9: 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.

```
1   int getSmallest(int x, int y) {
2     int smallest;
3     if ( x > y ) { //if (conditional)
4       smallest = y; //then statement
5     }
6     else {
7       smallest = x; //else statement
8     }
9     return smallest;
10  }
11  .
```

```
1   Dump of assembler code for function getSmallest:
2   0x40059a <+4>:    mov     %edi,-0x14(%rbp)
3   0x40059d <+7>:    mov     %esi,-0x18(%rbp)
4   0x4005a0 <+10>:   mov     -0x14(%rbp),%eax
5   0x4005a3 <+13>:   cmp     -0x18(%rbp),%eax
6   0x4005a6 <+16>:   jle     0x4005b0 <getSmallest+26>
7   0x4005a8 <+18>:   mov     -0x18(%rbp),%eax
8   0x4005ae <+24>:   jmp     0x4005b9 <getSmallest+35>
9   0x4005b0 <+26>:   mov     -0x14(%rbp),%eax
10  0x4005b9 <+35>:   pop     %rbp
11  0x4005ba <+36>:   retq
```

Figure 10: 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.

```
1   <getSmallest>:
2   0x400546 <+0>: cmp    %esi,%edi          #compare x and y
3   0x400548 <+2>: mov    %esi,%eax          #copy y to %eax
4   0x40054a <+4>: cmovle %edi,%eax          #if (x<=y) copy x to %eax
5   0x40054d <+7>: retq                      #return %eax
```

Figure 11: Compiled with `gcc -01 -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.

```
1   int sumUp(int n) {
2     int total = 0;
3     int i = 1;
4
5     while (i <= n) {
6       total += i;
7       i++;
8     }
9     return total;
10  }
11  .
12  .
13  .
14  .
15  .
16  .
```

```
1   Dump of assembler code for function sumUp:
2   0x400526 <+0>:    push    %rbp
3   0x400527 <+1>:    mov     %rsp,%rbp
4   0x40052a <+4>:    mov     %edi,-0x14(%rbp)
5   0x40052d <+7>:    mov     $0x0,-0x8(%rbp)
6   0x400534 <+14>:   mov     $0x1,-0x4(%rbp)
7   0x40053b <+21>:   jmp     0x400547 <sumUp+33>
8   0x40053d <+23>:   mov     -0x4(%rbp),%eax
9   0x400540 <+26>:   add     %eax,-0x8(%rbp)
10  0x400543 <+29>:   add     $0x1,-0x4(%rbp)
11  0x400547 <+33>:   mov     -0x4(%rbp),%eax
12  0x40054a <+36>:   cmp     -0x14(%rbp),%eax
13  0x40054d <+39>:   jle     0x40053d <sumUp+23>
14  0x40054f <+41>:   mov     -0x8(%rbp),%eax
15  0x400552 <+44>:   pop     %rbp
16  0x400553 <+45>:   retq
```

Figure 12: 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.

Register	Description		Register	Description
%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 13: 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 5.27 (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 5.28 (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 5.13 (Calling Functions in Assembly)

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

```c
1   #include <stdio.h>
2
3   int assign(void) {
4       int y = 40;
5       return y;
6   }
7
8   int adder(void) {
9       int a;
10      return a + 2;
11  }
12
13  int main(void) {
14      int x;
15      assign();
16      x = adder();
17      printf("x is:
        %d\n", x);
18      return 0;
19  }
20  .
21  .
22  .
23  .
24  .
25  .
26  .
27  .
28  .
29  .
30  .
31  .
```

```
1   0000000000400526 <assign>:
2     400526:      55                      push   %rbp
3     400527:      48 89 e5                mov    %rsp,%rbp
4     40052a:      c7 45 fc 28 00 00 00    movl   $0x28,-0x4(%rbp)
5     400531:      8b 45 fc                mov    -0x4(%rbp),%eax
6     400534:      5d                      pop    %rbp
7     400535:      c3                      retq
8
9   0000000000400536 <adder>:
10    400536:      55                      push   %rbp
11    400537:      48 89 e5                mov    %rsp,%rbp
12    40053a:      8b 45 fc                mov    -0x4(%rbp),%eax
13    40053d:      83 c0 02                add    $0x2,%eax
14    400540:      5d                      pop    %rbp
15    400541:      c3                      retq
16
17  0000000000400542 <main>:
18    400542:      55                      push   %rbp
19    400543:      48 89 e5                mov    %rsp,%rbp
20    400546:      48 83 ec 10             sub    $0x10,%rsp
21    40054a:      e8 e3 ff ff ff          callq  400526 <assign>
22    40054f:      e8 d2 ff ff ff          callq  400536 <adder>
23    400554:      89 45 fc                mov    %eax,-0x4(%rbp)
24    400557:      8b 45 fc                mov    -0x4(%rbp),%eax
25    40055a:      89 c6                   mov    %eax,%esi
26    40055c:      bf 04 06 40 00          mov    $0x400604,%edi
27    400561:      b8 00 00 00 00          mov    $0x0,%eax
28    400566:      e8 95 fe ff ff          callq  400400
                    <printf@plt>
29    40056b:      b8 00 00 00 00          mov    $0x0,%eax
30    400570:      c9                      leaveq
31    400571:      c3                      retq
```

Figure 14: 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

Stack "top"

Lower addresses

0xd48 ←— Stack "top"

Stack "bottom"

call stack
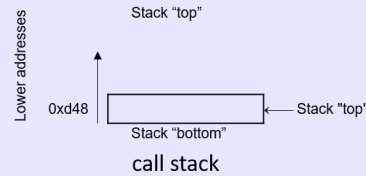
Terminal:
```
$ ./prog
```

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**

Stack "top"

Lower addresses

0xd40   0x830   ←— Stack "top"
0xd48

Stack "bottom"

call stack
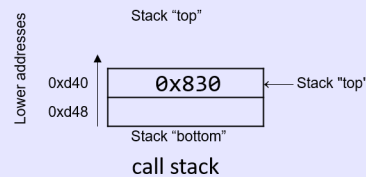
Terminal:
```
$ ./prog
```

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
```

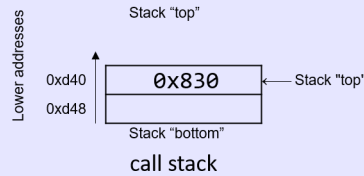| Registers	
%eax	650
%edi	1
%rsp	0xd40
%rbp	**0xd40**
%rip	**0x546**

Stack "top"

0xd40 | 0x830 | ⟵ Stack "top"
0xd48

Lower addresses

Stack "bottom"

call stack

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**

0xd30 | | ⟵ Stack "top"
0xd38
0xd40 | 0x830 |
0xd48

Lower addresses

Stack "bottom"

call stack
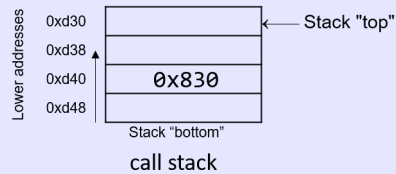
Terminal:
```
$ ./prog
```

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
```

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

Lower addresses

Stack "bottom"

call stack

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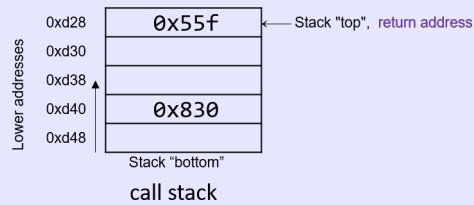
Terminal:

```
$ ./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
```

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

Lower addresses

Stack "bottom"

call stack

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
```

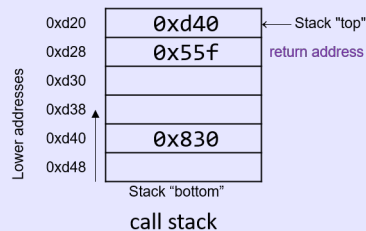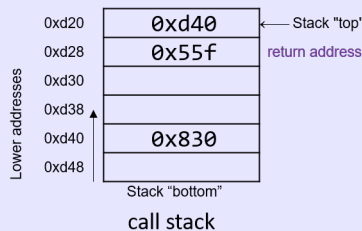| Registers	
%eax	0x0
%edi	1
%rsp	0xd20
%rbp	**0xd20**
%rip	**0x52a**

Stack (call stack):
0xd20 0xd40 ← Stack "top"
0xd28 0x55f return address
0xd30
0xd38
0xd40 0x830
0xd48
Stack "bottom"
Lower addresses

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**

Stack (call stack):
0xd1c **0x28**
0xd20 0xd40 ← Stack "top"
0xd28 0x55f return address
0xd30
0xd38
0xd40 0x830
0xd48
Stack "bottom"
Lower addresses
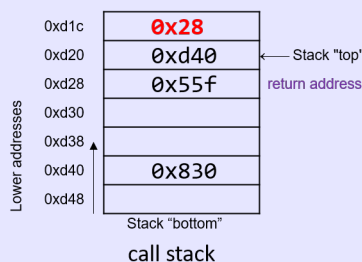
Terminal:
```
$ ./prog
```

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
```

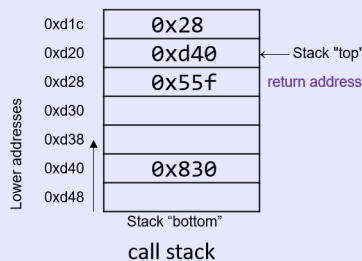| Registers	
%eax	**0x28**
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	**0x534**

Call stack (Lower addresses):

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

Stack "bottom"
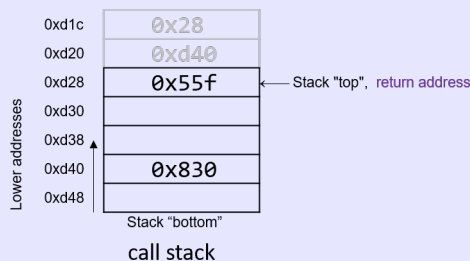
call stack

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**

Call stack (Lower addresses):

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

Stack "bottom"

call stack

Terminal:
```
$ ./prog
```

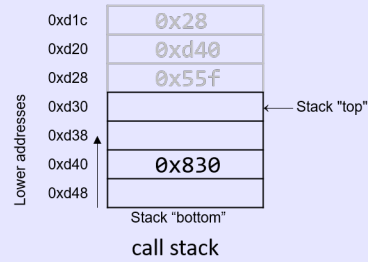11. Note that at this point, since `%rbp` was popped off, the next value that is at the top of the stack

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
```

0xd1c	0x28
0xd20	0xd40
0xd28	0x55f
0xd30	← Stack "top"
0xd38	
0xd40	0x830
0xd48	

Lower addresses

Stack "bottom"

call stack

Terminal:
```
$ ./prog
```

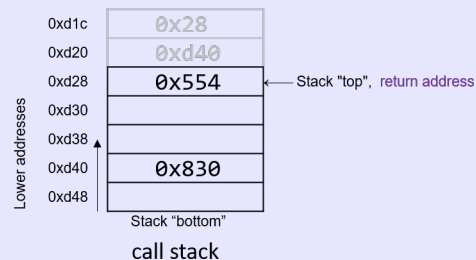| Registers	
%eax	0x28
%edi	1
%rsp	**0xd30**
%rbp	0xd40
%rip	**0x55f**

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
```

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

Lower addresses

Stack "bottom"

call stack

Terminal:
```
$ ./prog
```

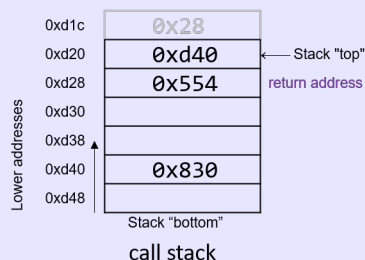| Registers	
%eax	0x0
%edi	1
%rsp	**0xd28**
%rbp	0xd40
%rip	**0x536**

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
```

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

Stack "bottom"

call stack

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

Terminal:

```
$ ./prog
```

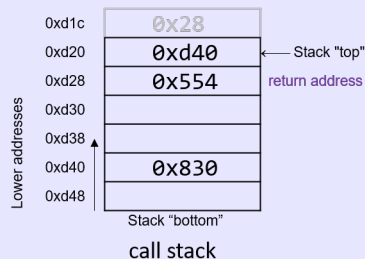14. Then we update **%rbp** to the current stack pointer.

```
          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
```

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

Stack "bottom"

call stack

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

Terminal:

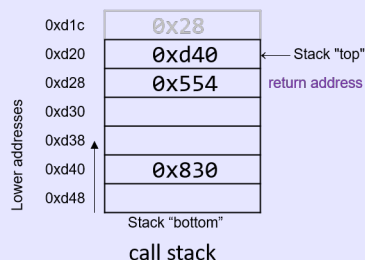```
$ ./prog
```

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**

Call stack diagram:
- 0xd1c : 0x28
- 0xd20 : 0xd40 ← Stack "top"
- 0xd28 : 0x554 return address
- 0xd30 :
- 0xd38 :
- 0xd40 : 0x830
- 0xd48 :
- Stack "bottom"
- call stack
- Lower addresses

Terminal:
```
$ ./prog
```
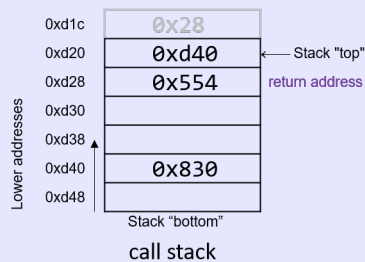
Using an old value on the stack!

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**

Call stack diagram:
- 0xd1c : 0x28
- 0xd20 : 0xd40 ← Stack "top"
- 0xd28 : 0x554 return address
- 0xd30 :
- 0xd38 :
- 0xd40 : 0x830
- 0xd48 :
- Stack "bottom"
- call stack
- Lower addresses

Terminal:
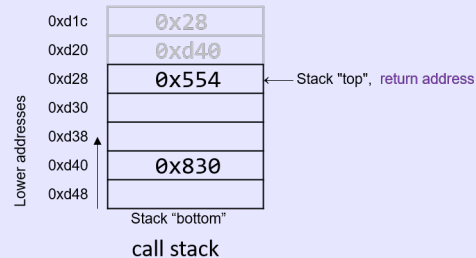```
$ ./prog
```

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
```

(arrow pointing at `0x540 pop %rbp`)

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

Call stack (addresses, lower addresses going up):

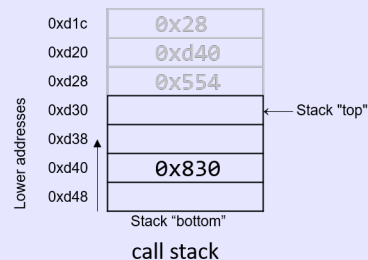| Address	Value	
0xd1c	0x28	
0xd20	0xd40	
0xd28	0x554	← Stack "top", return address
0xd30		
0xd38		
0xd40	0x830	
0xd48		Stack "bottom"

call stack

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
```

(arrow pointing at `0x541 retq`)

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

Call stack:

Address	Value	
0xd1c	0x28	
0xd20	0xd40	
0xd28	0x554	
0xd30		← Stack "top"
0xd38		
0xd40	0x830	
0xd48		Stack "bottom"

call stack

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
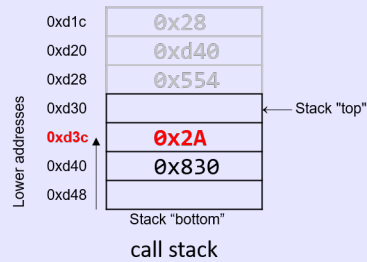
%eax.

```
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
```

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

Stack (call stack):

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

Lower addresses

Stack "bottom"

call stack

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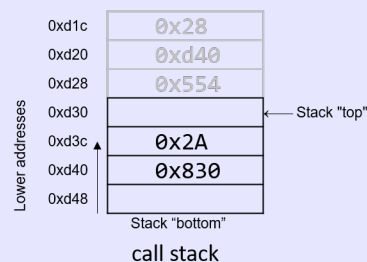
```
0x542 <main>:
0x542 push    %rbp
0x543 mov     %rsp, %rbp
0x546 sub     $0x10, %rsp
0x54a callq   0x526 <assign>
0x55f callq   0x536 <adder>
0x554 mov     %eax, -0x4(%rbp)
0x557 mov     -0x4(%rbp), %eax
0x55a mov     %eax, %esi
0x55c mov     $0x400604, %edi
0x561 mov     $0x0, %eax
0x566 callq  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

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

Lower addresses

Stack "bottom"

call stack

Terminal:
```
$ ./prog
```

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

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
```

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

Stack "bottom"

call stack

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

%esi	0x2A

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
```

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

Stack "bottom"

call stack

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

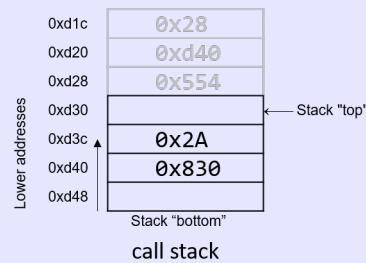| %esi	0x2A

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
```

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

Lower addresses

Stack "bottom"

call stack

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

%esi	0x2A

Terminal:
```
$ ./prog
x is 42
```
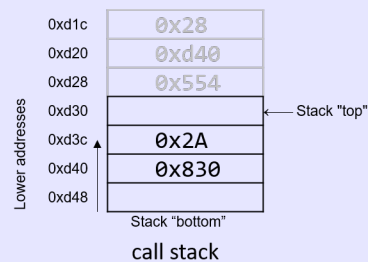
Memory	
0x400604	"x is %d\n"

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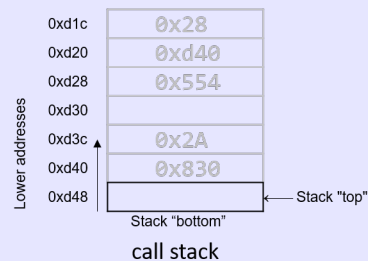
0xd1c	0x28
0xd20	0xd40
0xd28	0x554
0xd30		← Stack "top"
0xd3c	0x2A
0xd40	0x830
0xd48	

Lower addresses

Stack "bottom"

call stack

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
```

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

Lower addresses

Stack "top"

Stack "bottom"

call stack

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

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.

```
1  int sumArray(int *array, int length) {
2    int i, total = 0;
3    for (i = 0; i < length; i++) {
4      total += array[i];
5    }
6    return total;
7  }
```

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

```
1   0x400686 <+0>: push %rbp                  # save %rbp
2   0x400687 <+1>: mov  %rsp,%rbp             # update %rbp (new stack frame)
3   0x40068a <+4>: mov  %rdi,-0x18(%rbp)      # copy array to %rbp-0x18
4   0x40068e <+8>: mov  %esi,-0x1c(%rbp)      # copy length to %rbp-0x1c
5   0x400691 <+11>:   movl $0x0,-0x4(%rbp)       # copy 0 to %rbp-0x4 (total)
6   0x400698 <+18>:   movl $0x0,-0x8(%rbp)       # copy 0 to %rbp-0x8 (i)
7   0x40069f <+25>:   jmp  0x4006be <sumArray+56> # goto <sumArray+56>
8   0x4006a1 <+27>:   mov  -0x8(%rbp),%eax       # copy i to %eax
9   0x4006a4 <+30>:   cltq                       # convert i to a 64-bit integer
10  0x4006a6 <+32>:   lea  0x0(,%rax,4),%rdx     # copy i*4 to %rdx
11  0x4006ae <+40>:   mov  -0x18(%rbp),%rax      # copy array to %rax
12  0x4006b2 <+44>:   add  %rdx,%rax             # compute array+i*4, store in %rax
13  0x4006b5 <+47>:   mov  (%rax),%eax           # copy array[i] to %eax
14  0x4006b7 <+49>:   add  %eax,-0x4(%rbp)       # add %eax to total
15  0x4006ba <+52>:   addl $0x1,-0x8(%rbp)       # add 1 to i (i+=1)
16  0x4006be <+56>:   mov  -0x8(%rbp),%eax       # copy i to %eax
```

```
17   0x4006c1 <+59>:    cmp  -0x1c(%rbp),%eax        # compare i to length
18   0x4006c4 <+62>:    jl   0x4006a1 <sumArray+27> # if i<length goto <sumArray+27>
19   0x4006c6 <+64>:    mov  -0x4(%rbp),%eax         # copy total to %eax
20   0x4006c9 <+67>:    pop  %rbp                    # prepare to leave the function
21   0x4006ca <+68>:    retq                         # return total
```

6   x86 Control Flow

7 RISC-V Data Movement

8   RISC-V Arithmetic Operations

9   RISC-V Control Flow