Computer Systems

Muchang Bahng

Spring 2024

Contents

Before we do any coding, we must learn the theory behind how computer systems work, which all starts from memory management and CPU architecture. We will use C with the gcc compiler, along with MIPS and NASM assembler. It is imperative to learn these two since given that you know a high level language pretty well (Python in my case), you want to learn C to appreciate the things Python does for you, and you want to learn Assembly to appreciate the things C does for you.[1].

To start off, we want a big overall picture of high a computer works. We introduce this with the simplest model of the computer, the Von Nuemann architecture. It consists of a **central processing unit** (CPU), **memory**, and an **input/output** (I/O) system. We show a diagram of this first for conciseness in Figure 1.

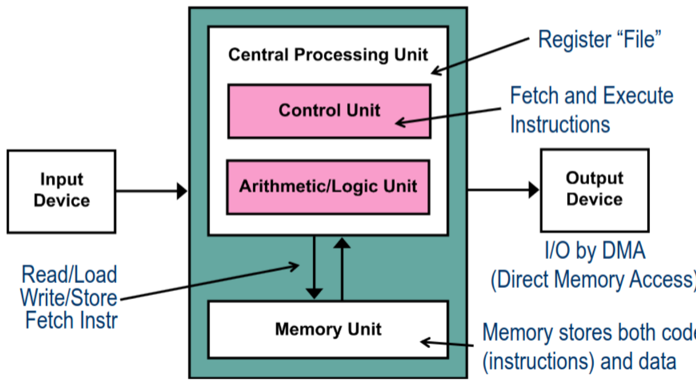


Figure 1: von Neumann Architecture

We will go through these one by one, touching on C and Assembly along the way, but the implementation of these things can differ by the **computer architecture**, so let's list some of the basic ones.

Definition 0.1 (Computer Architecture)

The **computer architecture** is the design of the computer, which includes the CPU, memory, and I/O system. There are many different architectures, but we will focus on the most common ones.

We first go over some basic theoretical properties of basic data types, focusing on C, and then we cover all the stuff about memory and then all the stuff about the CPU. This is a natural progression since to work with data, you must first know where to store the data and how it is stored (the memory), and then you want to know how data is manipulated (the CPU).

1 Encoding Schemes

In order to get into memory, it is helpful to know the theory behind how primitive types are stored in memory.

Definition 1.1 (Collections of Bits)

There are many words that are used to talk about values of different data types:

1. A **bit** (b) is either 0 or 1.
2. A **Hex** (x) is a collection of 4 bits, with a total of $2^4 = 16$ possible values, and this is used since it is easy to read for humans.
3. A **Byte** (B) is a collection of 8 bits or 2 hex, with a total of $2^8 = 256$ possible values, and most computers will work with Bytes as the smallest unit of memory.

[1] https://www.youtube.com/watch?v=XlvfHOrF26M

Definition 1.2 (Collections of Bytes)

Sometimes, we want to talk about slightly larger collections, so we group them by how many bytes they have. However, note that these may not always be the stated size, depending on what architecture or language you are using. This is more of a general term, and they may have different names in different languages. If there is a difference, we will state it explicitly.

1. A **word** (w) is 2 Bytes.
2. A **long** (l) is 4 Bytes.
3. A **quad** (q) is 8 Bytes.

Try to know which letter corresponds to which structure, since that will be useful in both C and Assembly.

1.1 Booleans and Characters

Definition 1.3 (Booleans in C)

The most basic type is the boolean, which is simply a bit. In C, it is represented as `bool`, and it is either `true` (1) or `false` (0).

We can manually check the size of the boolean type in C with the following code.

```c
#include<stdio.h>
#include<stdbool.h>

int main() {
  printf("%lu\n", sizeof(bool));
  return 0;
}
```

```
1
.
.
.
.
.
.
```

Figure 2: We can verify the size of various primitive data types in C with the `sizeof` operator.

1.2 Integer Family

The most primitive things that we can store are integers. Let us talk about how we represent some of the simplest primitive types in C: unsigned short, unsigned int, unsigned long, unsigned long long.

Definition 1.4 (Unsigned Integer Types in C)

In C, there are several integer types. We use this hierarchical method to give flexibility to the programmer on the size of the integer and whether it is signed or not.

1. An **unsigned short** is 2 bytes long and can be represented as a 4-digit hex or 16 bits, with values in $[0 : 65,535]$. Therefore, say that we have
2. An **unsigned int** is 4 bytes long and can be represented as an 8-digit hex or 32 bits, with values in $[0 : 4,294,967,295]$.
3. An **unsigned long** is 8 bytes and can be represented as an 16-digit hex or 64 bits, but they are only guaranteed to be stored in 32 bits in other systems.
4. An **unsigned long long** is 8 bytes and can be represented as an 16-digit hex or 64 bits, and they are guaranteed to be stored in 64 bits in other systems.

Theorem 1.1 (Bit Representation of Unsigned Integers in C)

To encode a signed integer in bits, we simply take the binary expansion of it.

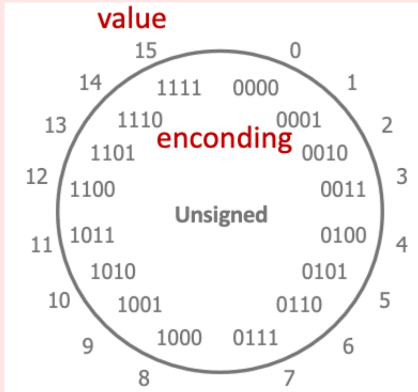


Figure 3: Unsigned encoding of 4-bit integers in C.

Example 1.1 (Bit Representation of Unsigned Integers in C)

We can see for ourselves how these numbers are represented in bits. Printing the values out in binary requires to make new functions, but we can easily convert from hex to binary.

```c
int main() {

  unsigned short x = 13;
  unsigned int y = 256;

  printf("%x\n", x);
  printf("%x\n", y);

  return 0;
}
```

```
d
100
.
.
.
.
.
.
.
.
```

So far, the process of converting unsigned numbers to bits seemed simple. Now let's introduce signed integers.

Definition 1.5 (Signed Integer Types in C)

In C, there are several signed integer types. We use this hierarchical method to give flexibility to the programmer on the size of the integer and whether it is signed or not.

1. A **signed short** is 2 bytes long and can be represented as a 4-digit hex or 16 bits, with values in $[-32,768 : 32,767]$.
2. A **signed int** is 4 bytes long and can be represented as an 8-digit hex or 32 bits, with values in $[-2,147,483,648 : 2,147,483,647]$.
3. A **signed long** is 8 bytes and can be represented as an 16-digit hex or 64 bits, but they are only guaranteed to be stored in 32 bits in other systems.
4. A **signed long long** is 8 bytes and can be represented as an 16-digit hex or 64 bits, and they are guaranteed to be stored in 64 bits in other systems.

To store signed integers, it is intuitive to simply take the first (left-most) bit and have that be the sign. Therefore, we lose one significant figure but gain information about the sign. However, this has some problems: first, there are two representations of zeros: -0 and $+0$. Second, the continuity from -1 to 0 is not natural. It is best explained through an example, which doesn't lose much insight into the general case.

Example 1.2 (Problems with the Signed Magnitude)

Say that you want to develop the signed magnitude representation for 4-bit integers in C. Then, you can imagine the following diagram to represent the numbers.

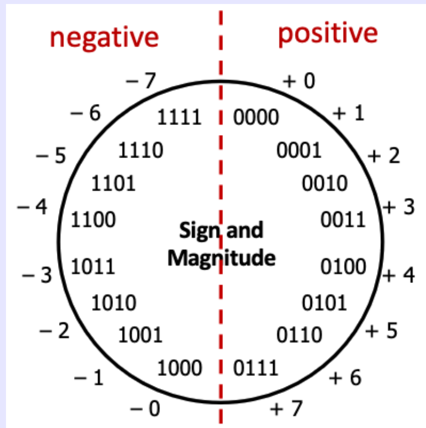


Figure 4: Signed magnitude encoding of 4-bit integers in C.

You can see that there are some problems:
1. There are two representations for 0, which is 0000 and 1000.
2. -1 (1001) plus 1 becomes -2 (1010).
3. The lowest number -7 (1111) plus 1 goes to 0 (0000) when it should go to -6 (1100).
4. The highest number 7 (0111) plus 1 goes to 0 (1000).

An alternative way is to use the two's complement representation, which solves both problems and makes it more natural.

Theorem 1.2 (Bit Representation of Signed Integers in C)

The **two's complement** representation is a way to represent signed integers in binary. It is defined as follows. Given that you want to store a decimal number p in n bits,
1. If p is positive, then take the binary expansion of that number, which should be at most $n-1$ bits (no overflow), pad it with 0s on the left.
2. If p is negative, then you can do two things: First, take the binary expansion of the positive number, flip all the bits, and add 1. Or second, represent $p = q - 2^n$, take the binary representation of q in $n-1$ bits, and add a 1 to the left.

If you have a binary number $b = b_n b_{n-1} \cdots b_1$ then to convert it to a decimal number, you simply calculate

$$q = -b_n 2^{n-1} + b_{n-1} 2^{n-2} + \cdots + b_1 \tag{1}$$

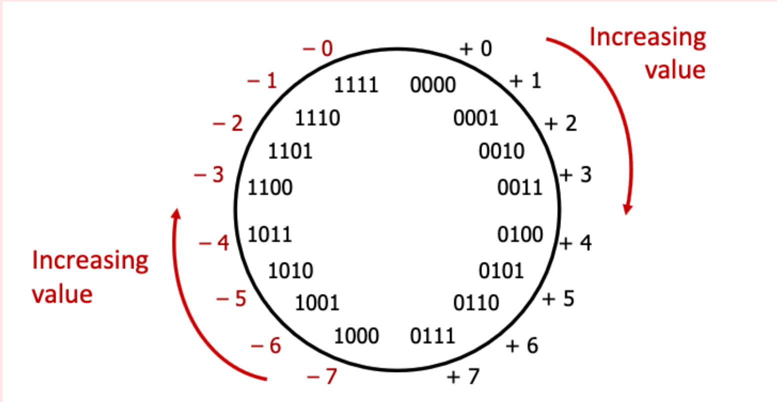


Figure 5: Two's complement encoding of 4-bit integers in C.

Example 1.3 (Bit Representation of Signed Integers in C)

We can see for ourselves how these numbers are represented in bits.

```c
int main() {

  short short_pos = 13;
  short short_neg = -25;
  int int_pos = 256;
  int int_neg = -512;

  printf("%x\n", short_pos);
  printf("%x\n", short_neg);
  printf("%x\n", int_pos);
  printf("%x\n", int_neg);

  return 0;
}
```

```
d
ffe7
100
ffffffe00
.
.
.
.
.
.
.
.
.
.
```

```c
#include<stdio.h>
#include<stdbool.h>

int main() {
  printf("%lu\n", sizeof(bool));
  printf("%lu\n", sizeof(short));
  printf("%lu\n", sizeof(int));
  printf("%lu\n", sizeof(long));
  printf("%lu\n", sizeof(long long));
  return 0;
}
```

```
1
2
4
8
8
.
.
.
.
.
.
```

Figure 6: Size of various integer types in C with the `sizeof`.

1.2.1 Arithmetic Operations on Binary Numbers

Theorem 1.3 (Inversion of Binary Numbers)

Given a binary number p, to compute $-p$, simply invert the bits and add 1.

Theorem 1.4 (Addition and Subtraction of Binary Numbers)

Given two binary numbers p and q.

1. To compute $p + q$, simply add the numbers together as you would in base 10, but carry over when the sum is greater than 1.
2. To compute $p - q$, you can invert q to $-q$ and compute $p + (-q)$.

1.3 Float Family

Definition 1.6 (Floating Point Types in C)

In C, there are several floating point types. We use this hierarchical method to give flexibility to the programmer on the size of the integer and whether it is signed or not.

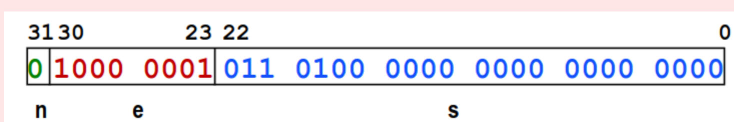
1. A **float** is 4 bytes long and can be represented as an 8-digit hex or 32 bits, with values in $[1.2 \times 10^{-38} : 3.4 \times 10^{38}]$.
2. A **double** is 8 bytes long and can be represented as an 16-digit hex or 64 bits, with values in $[2.3 \times 10^{-308} : 1.7 \times 10^{308}]$.
3. A **long double** is 8 bytes and can be represented as an 16-digit hex or 64 bits, but they are only guaranteed to be stored in 80 bits in other systems.

Theorem 1.5 (Bit Representation of Floating Point Types in C)

Floats are actually like signed magnitude. We have

$$(-1)^n \times 2^{e-127} \times 1.s \tag{2}$$

where



Doubles encode 64 bits, so not we have exponent having 11 bits (so bias is not 1023) and 52 bits for mantissa.

2 Memory

Definition 2.1 (Memory)

The **memory** is where the computer stores data and instructions, which can be though of as a giant array of memory addresses, with each containing a byte. This data consists of graphical things or even instructions to manipulate other data. It can be visualized as a long array of boxes that each have an **address** (where it is located) and **contents** (what is stored in it).
Memory simply works as a bunch of bits in your computer with each bit having some memory address, which is also a bit. For example, the memory address `0b0010` (2) may have the bit value of `0b1` (1) stored in it.

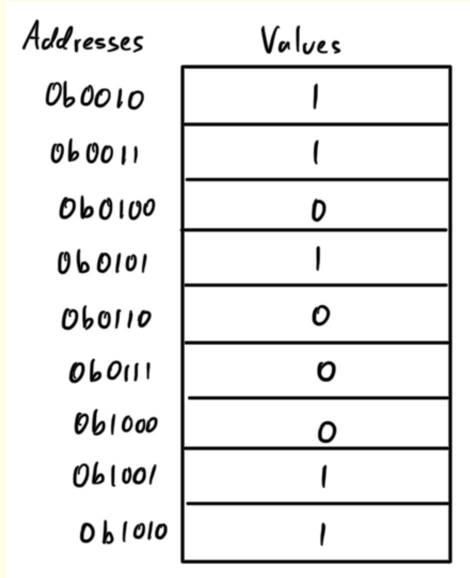


Figure 7: Visualization of memory as a long array of boxes of bits.

However, computers do not need this fine grained level of control on the memory, and they really work at the Byte level rather than the bit level. Therefore, we can visualize the memory as a long array of boxes indexed by *Bytes*, with each value being a byte as well. In short, the memory is **byte-addressable**. In certain arthitectures, some systems are **word-addressable**, meaning that the memory is addressed by words, which are 4 bytes.[a]

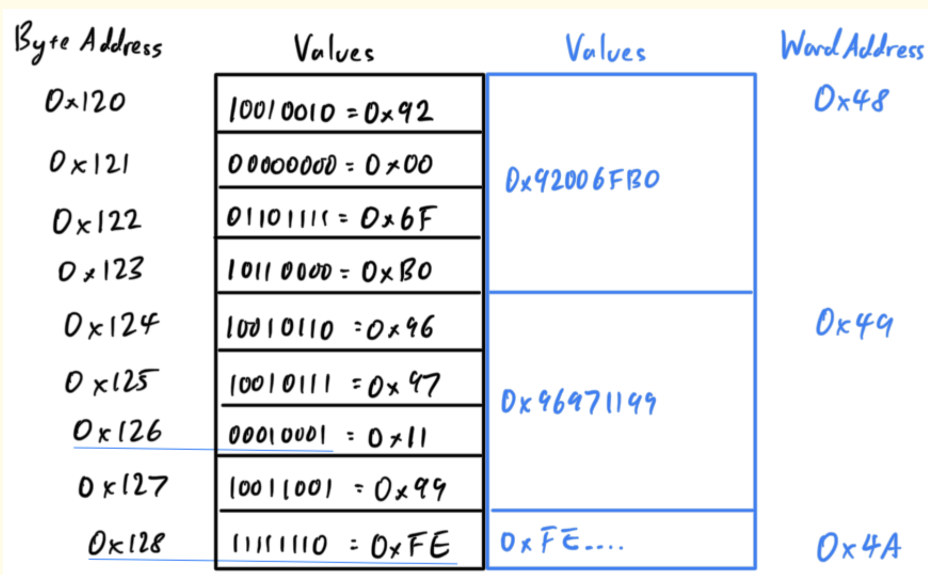


Figure 8: Visualization of memory as a long array of boxes of bytes. Every address is a byte and its corresponding value at that address is also a byte, though we represent it as a 2-digit hex.

[a]Note that in here the size of a word is 2 bytes rather than 4 as stated above. This is just how it is defined in some `x86` architectures.

In the examples above, I listed the memory addresses as a 3 hex character (1.5 bytes) for brevity. In reality, the number of bytes that a memory address takes is much longer.

Definition 2.2 (32 and 64 Bit Machines)

There are two types of machines that tend to format these boxes very differently: 32-bit and 64-bit machines.

1. 32 bit machines store addresses in 32 bits, so they can have 2^{32} addresses, which is about 4 GB of memory.
2. 64 bit machines store addresses in 64 bits, so they can have 2^{64} addresses, which is about 16 EB of memory. This does not mean that the actual RAM is 16 EB, but it means that the machine can *handle* that much memory.

```
...
0x00007FFF7FBFF860 --> 0b000000000000000000000000011111111111
                       111101111111101111111111100001100000
0x00007FFF7FBFF861 --> 0b000000000000000000000000011111111111
                       111101111111101111111111100001100001
0x00007FFF7FBFF862 --> 0b000000000000000000000000011111111111
                       111101111111101111111111100001100010
0x00007FFF7FBFF863 --> 0b000000000000000000000000011111111111
                       111101111111101111111111100001100011
0x00007FFF7FBFF864 --> 0b000000000000000000000000011111111111
                       111101111111101111111111100001100100
...
```

The numbers typically mean the size of the type that the machine works best with, so all memory addresses will be 32 or 64 bits wide. Most machines are 64-bits, and so everything in this notes will assume that we are working with a 64 bit machine. As we will later see, this is why pointers are 8 bytes long, i.e. 64 bits. This is because the memory addresses are 64 bits long, though all of them are not used.

With this structure in mind and knowing the size of some primitive types, we can now focus on how declaring them works in the backend.

Definition 2.3 (Declaration, Initialization)

Assigning a value to a variable is a two step process, which is often not distinguished in high level languages like Python.

1. You must first **initialize** the variable by setting aside the correct number of bytes in memory.
2. You must then **assign** that variable to be some actual value.

The two step process is often called declaration.

This is the reason why C is statically, or strongly, typed. In order to set aside some memory for a variable, you must know how big that variable will be, which you know by its type. This makes sense. We can first demonstrate how to both initialize and declare a variable.

```c
int main() {
  // declaring
  int x = 4;
  printf("%p\n", &x);

  // initializing and assigning
  int y;
  printf("%p\n", &y);
  y = 3;
  printf("%p\n", &y);

  return 0;
}
```

```
0x16d37ee68
0x16d37ee64
0x16d37ee64
.
.
.
.
.
.
.
.
.
.
```

Figure 9: How to declare variables in C. As you can see, by initializing `y`, the memory address is already assigned and it doesn't change when you assign it. The address is only shown to be 9 hex digits long, but it is actually 16 hex digits long and simply 0 padded on the left.

One question that may come to mind is, what is the value of the variable if you just initialize it? After all the value at that address that is initialized must be either 0s or 1s. Let's find out.

```c
int main() {
  int y;
  printf("%d\n", y);
  y = 3;
  printf("%d\n", y);

  return 0;
}
```

```
6298576
3
.
.
.
.
.
.
```

Figure 10: The value of an uninitialized variable is some random number.

It may be interesting to see how this random unititialized value is generated. It is simply the value that was stored in that memory address before, and it is not cleared when you initialize it, so you should not use this as a uniform random number generator.

2.1 Debugging and Object Dumping

Talk about gdb, lldb, objdump, etc. These are debugging tools that allow you to parse your code line by line. However, to actually see the C code, you must compile it with the debugging flag. This adds a little bit of overhead memory to the binary, but not a lot.

2.2 Endian Architecture

It is intuitive to think that given some multi-byte object like an `int` (4 bytes), the beginning of the int would be the lowest address and the end of the int would be the highest address, like how consecutive integers are stored in an array. However, this is not always the case (almost always not the case since most computers are little-endian).

Definition 2.4 (Endian Architecture)

Depending on the machine architecture, computers may store these types slightly differently in their *byte* order. Say that we have an integer of value `0xA1B2C3D4` (4 bytes). Then,

1. A **big-endian architecture** (e.g. SPARC, z/Architecture) will store it so that the least significant byte has the highest address.
2. A **little-endian architecture** (e.g. x86, x86-64, RISC-V) will store it so that the least significant byte has the lowest address.
3. A **bi-endian architecture** (e.g. ARM, PowerPC) can specify the endianness as big or little.

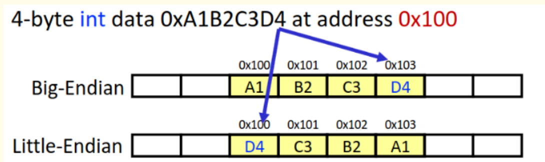


Figure 11: The big vs little endian architectures.

We can simply print out the hex values of primitive types to see how they are stored in memory, but it does not provide the level of details that we want on which bytes are stored where. At this point, we must use certain **debuggers** to directly look at the memory. For x86 architectures, we can use `gdb` and for ARM architectures, we can use `lldb`. At this point, we need to understand assembly to look through debuggers, so we will provide the example here.

Example 2.1 (Endianness of C Int in x86-64)

To do.

Example 2.2 (Endianness of C Int in ARM64)

To do.

2.3 Type Casting

2.4 Pointers

We have learned how to declare/initialize a variable, which frees up some space in the memory and possibly assigns a value to it. One great trait of C is that we can also store the memory address of a variable in another variable called a pointer. You access both the memory and the value at that memory with this pointer variable.

Definition 2.5 (Pointer Variable)

A **pointer** variable/type is a variable that stores the memory address of another variable.

1. You can declare a pointer in the same way that you declare a variable, but you must add a asterisk in front of the variable name.
2. The size of this variable is the size of the memory address, which is 8 bytes in a 64-bit architecture.
3. To get the value of the variable that the pointer points to, called **dereferencing**, you simply put a asterisk in front of the pointer. This is similar to how you put a ampersand in front of a variable to get its memory address.

```c
int main() {
  // declare an integer
  int x = 4;
  printf("x = %d\n", x);
  printf("&x = %p\n", &x);

  // declare pointer
  int *p = &x;
  printf("p = %p\n", p);
  printf("*p = %d\n", *p);

  // initialize pointer
  int *q;
  q = &x;
  printf("q = %p\n", q);
  printf("*q = %d\n", *q);
  return 0;
}
```

```
x = 4
&x = 0x16d49ae68
p = 0x16d49ae68
*p = 4
q = 0x16d49ae68
*q = 4
.
.
.
.
.
.
.
.
.
.
.
.
```

Figure 12

Since the size of addresses are predetermined by the architecture, it may not seem like we need to know the underlying data type of what it points to, so why do we need to write strongly type the underlying data type? Remember that to do pointer arithmetic, you need to know how large the underlying data type is so that you can know how many bytes to move when traversing down an array.

One of the reasons why pointers are so valuable is that they allow you to pass by reference, which is a way to change the value of a variable in a function.

2.4.1 Call by Value vs Call by Reference

Definition 2.6 (Call by Value)

Definition 2.7 (Call by Reference)

2.4.2 Pointer Errors

Just like for regular variables, you may be curious on the value of an unassigned pointer. Let's take a look.

Example 2.3 (Uninitialized Pointers)

```c
int main() {
  int x = 4;
  int *p;
  printf("p = %p\n", p);
  printf("*p = %x\n", *p);

  return 0;
}
```

```
p = 0x10249ff20
*p = d100c3ff
.
.
.
.
.
.
```

Figure 13: The value of an uninitialized pointer is some random address and at a random address it would be some random byte.

This is clearly not good, especially since the program compiles correctly and runs without any errors. This kind of pointer that hasn't been initialized is called a wild pointer.

Definition 2.8 (Wild Pointer)

A **wild pointer** is a pointer that has not been initialized to a known value.

To fix this, we must always initialize a pointer to a known value. This may come at a disadvantage, since now we can't reap the benefits of initializing first and assigning later. A nice compromise is to initialize the pointer to a null pointer.

Definition 2.9 (Null Pointer)

A **null pointer** is a pointer that has been initialized to a known value, which is the address 0x0. You can set the type of the pointer and then initialize it to `NULL`.

```c
int main() {
  int *p = NULL;
  printf("p = %p\n", p);

  // the code below gives seg fault
  /* printf("*p = %d\n", *p); */

  int x = 4;
  p = &x;
  printf("p = %p\n", p);
  printf("*p = %d\n", *p);
  return 0;
}
```

```
p = 0x0
p = 0x16da72e5c
*p = 4
.
.
.
.
.
.
.
.
.
.
```

Figure 14: Initializing a null pointer. It is a good practice to initialize a pointer to a null value.

Therefore, the null pointer allows you to set the type of the underlying data type, but the actual address will be 0x0. You cannot dereference a null pointer, and doing so will give you a segmentation fault. There may be times when you do not even know the data type of the pointer, and for this you can use the void pointer, which now doesn't know the type of the variable that it points to but it does allocate address.

Definition 2.10 (Void Pointer)

A **void pointer** is a pointer that does not know the type of the variable that it points to. We can initialize it by simply setting the underlying type to be void. This initializes the address, which should always be 8 bytes, but trying to access the value of the variable is not possible.

```c
int main() {
  void *p;
  printf("p = %p\n", p);
  int x = 4;
  p = &x;
  printf("%d", *((int*)p));
  return 0;
}
```

```
p = 0x102553f54
4
.
.
.
.
.
.
```

Figure 15: Initialize a void pointer and then use typecasting to access the value of the variable that it points to.

2.5 Pointer Arithmetic

With pointers out of the way, we can talk about how arrays are stored in memory.

Definition 2.11 (Array)

A C array is a collection of elements of the same type, which are stored in contiguous memory locations. You can initialize and declare arrays in many ways, and access their elements with the index, e.g. `arr[i]`.

1. You declare an array of some constant number of elements n with the elements themselves.

```c
int arr[5] = {1, 2, 3, 4, 5};
```

2. You declare an array with out its size n and simply assign them. Then n is automatically determined.

```c
int arr[] = {1, 2, 3, 4, 5};
```

3. You initialize an array of some constant size c, and then you assign each element of the array.

```c
int arr[5];
for (int i = 0; i < 5; i++) {
  arr[i] = i + 1;
}
```

Unfortunately, C does not provide a built-in way to get the size of the array (like `len` in Python), so we must keep track of the size of the array ourselves. Furthermore, the address of the array is the address of where it begins at, i.e. the address of the first element.

You can literally see that the elements of the array are contiguous in memory by iterating through each element and printing out its address.

```c
int main(void) {
  // initialize array
  int arr[5];
  for (int val = 1; val < 6; val++) {
    arr[val-1] = val * val;
  }

  int* p = &arr[0];
  for (int i = 0; i < 5; i++) {
    printf("Value at position %d : %d\n", i,
     arr[i]);
    printf("Address at position %d : %p\n",
     i, p + i);
  }

  return 0;
}
```

```
Value at position 0 : 1
Address at position 0 : 0x7ffd8636b0d0
Value at position 1 : 4
Address at position 1 : 0x7ffd8636b0d4
Value at position 2 : 9
Address at position 2 : 0x7ffd8636b0d8
Value at position 3 : 16
Address at position 3 : 0x7ffd8636b0dc
Value at position 4 : 25
Address at position 4 : 0x7ffd8636b0e0
.
.
.
.
.
.
.
```

Figure 16: Ints are 4 bytes long, so the address of the next element is 4 bytes away from the previous element, making this a contiguous array.

The most familiar implementation of an array is a string in C.

Definition 2.12 (String)

A string is an array of characters, which is terminated by a null character `\0`. You can initialize them in two ways:

1. You can declare a string with the characters themselves, which you must make sure to end with the null character.

```c
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

2. You can declare them with double quotes, which automatically adds the null character.

```c
char str[] = "Hello";
```

Note that for whatever string we initialize, the size of the array is the number of characters plus 1.

To access elements of an array, you simply use the index of the element, e.g. `arr[i]`, but in the backend, this is implemented with *pointer arithmetic*.

Definition 2.13 (Pointer Arithmetic)

Pointer arithmetic is the arithmetic of pointers, which is done by adding or subtracting an integer to a pointer.

1. If you add an integer n to a pointer p, e.g. `p + n`, then the new pointer will point to the nth element after the current element, with the next element being `sizeof(type)` bytes away from the pervious element.
2. If you subtract an integer n from a pointer, then the pointer will point to the nth element before the current element.

This is why you can access the elements of an array with the index, since the index is simply the number of elements away from the first element.

Example 2.4 (Pointer Arithmetic with Arrays of Ints and Chars)

Ints have a size of 4 bytes and chars 1 byte. You can see that using pointer arithmetic, the addresses of the elements of ints increment by 4 and those of the char array increment by 1.

```c
int main() {
  int integers[3] = {1, 2, 3};
  char characters[3] = {'a', 'b', 'c'};
  int *p = &integers[0];
  char *q = &characters[0];

  printf("Array of Integers\n");
  for (int i = 0; i < 3; i++) {
    printf("%p\n", integers+i); }

  printf("Array of Characters\n");
  for (int i = 0; i < 3; i++) {
    printf("%p\n", characters+i); }
  return 0;
}
```

```
Array of Integers
0x16d39ee58
0x16d39ee5c
0x16d39ee60
.
Array of Characters
0x16d39ee50
0x16d39ee51
0x16d39ee52
.
.
.
.
.
.
```

Therefore, we can think of accessing the elements of an array as simply pointer arithmetic.

Theorem 2.1 (Bracket Notation is Pointer Arithmetic)

The bracket notation is simply pointer arithmetic in the backend.

```c
int main() {
  int arr[3] = {1, 2, 3};
  int *p = &arr[0];

  for (int i = 0; i < 3; i++) {
    printf("%d\n", arr[i]);
    printf("%d\n", *(p+i));
  }
  return 0;
}
```

```
1
1
2
2
3
3
.
.
.
.
```

Figure 17: Accessing the elements of the list using both ways is indeed the same.

2.6 Global, Stack, and Heap Memory

Everything in a program is stored in memory, variables, functions, and even the code itself. However, we will find out that they are stored in different parts of the memory. When a program runs, its application memory consists of four parts, as visualized in the Figure 18.

1. The **code** is where the code text is stored.
2. The **global memory** is where all the global variables are stored.
3. The **stack** is where all of the functions and local variables are stored.
4. The **heap** is variable and can expand to as much as the RAM on the current system. We can specifically store whatever variables we want in the heap.

We provide a visual of these four parts first, and we will go into them later.

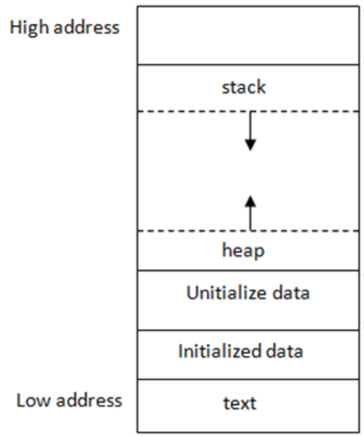


Figure 18: The four parts of memory in a C program.

Definition 2.14 (Code Memory)

This is where the code text is stored. It is read-only and is not modifiable.

In high level languages, we always talk about local and global scope. That is, variables defined within functions have a local scope in the sense that anything we modify in the local scope does not affect the global scope. We can now understand what this actually means by examining the backend. The global scope variables are stored in the global memory, and all local variables (and functions) are stored in the stack.

Definition 2.15 (Global Memory)

This is where all the global variables are stored.

Definition 2.16 (Stack Memory)

This is where all of the functions and local variables are stored. As we will see later, the compiler will always run the main function, which must exist in your file. By the main function is a function itself, and therefore it has its own local scope.
Then, when you initialize any functions or local variables within those functions (which will be the majority of your code), all these will be stored in the stack, which is an literally an implementation of the stack data structure. It is LIFO, and the first thing that goes in is the `main` function and its local variables, which is referred to as the **stack frame**. You can't free memory in the stack unless its in the top of the stack.

To see what happens in the stack, we can go through an example.

Example 2.5 (Going through the Stack)

Say that you have the following code:

```
int total;
int Square(int x) {
  return x*x;
}
int SquareOfSum(int x, int y) {
  int z = Square(x + y);
  return z;
```

```
}
int main() {
  int a = 4, b = 8;
  total = SquareOfSum(a, b);
  printf("output = %d", total);
  return 0;
}
```

The memory allocation of this program will run as such:

1. The `total` variable is initialized and is put into global memory.
2. `main` is called. It is put into the stack.
3. The local variables `a=4` and `b=8` are initialized and are put into the stack.
4. The `SquareOfSum` function is called and put into the stack.
5. The input local variables `x=4`, `y=8`, `z` are initialized and put into the stack.
6. `x + y=12` is computed and put into the stack.
7. The `Square` function is called and put into the stack.
8. The `x=12` local variable of `Square` is initialized and put into the stack.
9. The CPU computes `x*x=144` and returns the output. The `Square` function is removed from the stack.
10. We assign `z=144` and `SquareOfSum` returns it. Now `SquareOfSum` is removed from the stack.
11. `total=144` is assigned in the global memory still.
12. The `printf` function is called and put into the stack.
13. The `printf` function prints the output and is removed from the stack.
14. The `main` function returns `0` and is removed from the stack, ending our application.

One limitation of the stack is that its total available memory is fixed from the start, ranging from 1MB to 8MB, and so you can't initialize arrays of billions of integers in the stack. It will cause a memory overflow. In fact, the memory of the stack, along with the global and text memory, are assigned at compile time, making it a **static memory**.

Since the stack is really just a very small portion of available memory, the heap comes into rescue, which is the pool of memory available to you in RAM.

Definition 2.17 (Heap Memory)

The **heap memory** (nothing to do with the heap data structure) is a variable length (meaning it can grow at runtime) and **dynamically allocated** (meaning that we can assign memory addresses during runtime) memory that is limited to your computer's hardware. Unlike simply initializing variables to allocate memory as in the stack, we must use the **malloc** and **free** functions in C, and **new** and **delete** operations in C++.

Definition 2.18 (malloc)

Definition 2.19 (free)

The stack can store pointer variables that point to the memory address in the heap. So the only way to access variables in the heap is through pointer reference, and the stack provides you that window to access that big pool of heap memory.

One warning: if you allocate another address, the previous address does not get deallocated off the memory.

Definition 2.20 (Memory Leak)

On the other hand, if you free an address but have a pointer still pointing to that address, this is also a problem called the dangling pointer.

Definition 2.21 (Dangling Pointer)

At this point, we might be wondering why we need both a stack and a heap. Well the benefits of heaps are clearer since you can dynamically allocate memory, and you don't have the LIFO paradigm that is blocking you from deallocating memory that has been allocated in the beginning of your program. A problem with just having heap is that stacks can be orders of magnitude times faster when allocating/deallocating from it than the heap, and the sequence of function calls is naturally represented as a stack.

2.7 Dynamic Memory Allocation

Let's talk about how `malloc` and `free` are implemented in C. If you make a for loop and simply print all the addresses that you allocate to. You will find that they can be quite random. After a program makes some calls to malloc and free, the heap memory can becomes fragmented, meaning that there are chunks of free heap space interspersed with chunks of allocated heap space. The heap memory manager typically keeps lists of different ranges of sizes of heap space to enable fast searching for a free extent of a particular size. In addition, it implements one or more policies for choosing among multiple free extents that could be used to satisfy a request.

The free function may seem odd in that it only expects to receive the address of the heap space to free without needing the size of the heap space to free at that address. That's because malloc not only allocates the requested memory bytes, but it also allocates a few additional bytes right before the allocated chunk to store a header structure. The header stores metadata about the allocated chunk of heap space, such as the size. As a result, a call to free only needs to pass the address of heap memory to free. The implementation of free can get the size of the memory to free from the header information that is in memory right before the address passed to free.

3 Implementations of Memory Structures in C

3.1 Arrays

3.2 Strings

3.3 Structs

3.4 Functions

3.5 Classes (for C++)

3.6 Input Output

We have standard in, standard out, and standard error.

4 Central Processing Unit

Now let's talk about how functions work on a deeper level. When we write a command, like `int x = 4`, we are manually looking for an address (in the stack, global, or heap) and rewriting the bits that are at that address. Functions are just an automated way to do this, and all these modifications and computations are done by the CPU.

Definition 4.1 (Central Processing Unit)

The CPU is responsible for taking instructions (data) from memory and executing them.
1. The CPU is composed of **registers** (different from the cache), which are small, fast storage locations. These registers can either be **general purpose** (can be used with most instructions) or **special purpose** (can be accessed through special instructions, or have special meanings/uses, or are simply faster when used in a specific way).
2. The CPU also has an **arithmetic unit** and **logic unit**, which is responsible for performing arithmetic and logical operations.
3. The CPU also has a **control unit**, which is responsible for fetching instructions from memory through the **databus**, which is literally a wire connecting the CPU and RAM, and executing them.

It executes instructions from memory one at a time and executes them, known as the **fetch-execute cycle**. It consists of 4 main operations.
1. **Fetch**: The **program counter**, which holds the memory address of the next instruction to be executed, tells the control unit to fetch the instruction from memory through the databus.
2. **Decode**: The fetched data is passed to the **instruction decoder**, which figures out what the instruction is and what it does and stores them in the registers.
3. **Execute**: The arithmetic and logic unit then carries out these operations.
4. **Store**: Then it puts the results back on the databus, and stores them back into memory.

The CPU's **clock cycle** is the time it takes for the CPU to execute one instruction. More specifically, the clock cycle refers to a single oscillation of the clock signal that synchronizes the operations of the processor and the memory (e.g. fetch, decode, execute, store), and decent computers have clock cycles of at least 2.60GHz (2.6 billion clock cycles per second).

Therefore, in order to actually do computations on the data stored in the memory, the CPU must first fetch the data, perform the computations, and then store the results back into memory. This can be done in two ways.

1. Load and Store Operations: CPUs use load instructions to move data from memory to registers (where operations can be performed more quickly) and store instructions to move the modified data back into memory.

2. If the data is too big to fit into the registers, the CPU will use the **cache** to store the data, and in worse cases, the actual memory itself. Compilers optimize code by maximizing the use of registers for operations to minimize slow memory access. This is why you often see assembly code doing a lot in registers.

To clarify, let us compare registers and memory. Memory is addressed by an unsigned integer while registers have names like `%rsi`. Memory is much bigger at several GB, while the total register space is much smaller at around 128 bytes (may differ depending on the architecture). The memory is much slower than registers, which is usually on a sub-nanosecond timescale. The memory is dynamic and can grow as needed while the registers are static and cannot grow.

The specific structure/architecture of the CPU is determined by the instruction set architecture (ISA), which can be thought of as a subset of the general computer architecture.

Definition 4.2 (Instruction Set Architecture)

The **ISA** or just **architecture** of a CPU is a high level description of what it can do. Some differences are listed here:
1. What instructions it can execute.
2. The instruction length and decoding, along with its complexity.
3. The performance vs power efficiency.

ISAs can be classified into two types.

1. The **complex instruction set computer** (CISC) is characterized by a large set of complex instructions, which can execute a variety of low-level operations. This approach aims to reduce the number of instructions per program, attempting to achieve higher efficiency by performing more operations with fewer instructions.
2. The **reduced instruction set computer** (RISC) emphasizes simplicity and efficiency with a smaller number of instructions that are generally simpler and more uniform in size and format. This approach facilitates faster instruction execution and easier pipelining, with the philosophy that simpler instructions can provide greater performance when optimized.

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

Example 4.1 (x86 Architecture)

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

1. It is a complex instruction set computer (CISC) architecture, which means that it has a large set of complex instructions[a].
2. Byte-addressing is enabled and words are stored in little-endian format.
3. In the x86_64 architecture, registers are 8 bytes long (and 4 bytes in x86_32) and there are 16 total general purpose registers, for a total of only 128 bytes (very small compared to many GB of memory). Other special purpose registers are also documented in the wikipedia page, but it is not fully documented.

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

Example 4.2 (ARM Archiecture)

Mainly in phones, tablets, laptops.

Example 4.3 (MIPS Architecture)

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

Definition 4.3 (Input/Output Device)

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

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

Example 4.4 (Assembly Syntax)

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

1. **Intel Syntax**: Specifies memory operands without any special prefixes. Square brackets [] are used to denote memory addresses. For example, mov eax, [ebx] means move the contents of the

memory location pointed to by ebx into eax.
2. **AT&T Syntax**: Memory operands are denoted with parentheses () and include the % prefix for registers. An instruction moving data from a memory location into a register might look like movl (%ebx), %eax, with additional prefixes for immediate values and segment overrides.

Example 4.5 (Assembly Languages)

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

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

4.1 Circuits

Let's go over some common logic gates since this is at the basis of how to construct arithmetic operations.

Definition 4.4 (AND, NOT, OR)

Definition 4.5 (XOR, NAND, NOR)

Definition 4.6 (NAND)

Talk about how to construct arithmetic operations with these gates such as adding two integers or multiplying them, and not just that, but other operations that we may need in a programming language.

Theorem 4.1 (Implementation of Moving Data in Circuits)

Theorem 4.2 (Implementation of Addition, Subtraction in Circuits)

Theorem 4.3 (Implementation of Multiplication in Circuits)

Theorem 4.4 (Implementation of Bitwise Operations in Circuits)

Theorem 4.5 (Implementation of Bitshift Operations)

We also want some sort of conditionals. This then can be used to implement loops by checking some conditional.

Theorem 4.6 (Implementation of Conditionals in Circuits)

As a bonus, we talk about the difference between volatile and non-volatile memory. We already learned that RAM is volatile, and this is simple to implement in a circuit since we can manually set all the bits to 0 or just deplete all power. If this is the case, then how does non-volatile memory like SSDs maintain their state?

Theorem 4.7 (Implementation of Volatile Memory)

Theorem 4.8 (Implementation of Non-Volatile Memory)

4.2 Registers

To understand anything that the CPU does, we must understand assembly language. In here, everything is done within registers, and we can see how the CPU fetches, decodes, and executes instructions. So what exactly are these registers?

Definition 4.7 (Register)

A register is a small, fast storage location within the CPU. It is used to store data that is being used immediately, and is the only place where the CPU can perform operations, which is why it must move data from memory to registers before it can perform operations on it. Everything in a register is in binary, at most 8 bytes, or 64 bits.
There are very specific types of registers that you should know. All of these registers are implemented for all assembly languages and are integral to the workflow of the CPU.

1. **parameter registers** which store the parameters of a function.
2. **Return registers** which store return values of functions.
3. **stack pointers** which point to the top of the stack (at the top of the current stack frame).
4. **frame pointers** which point to the base of the current stack frame.
5. **instruction pointers** which point to the next instruction to be executed.

4.2.1 x86 Assembly Registers

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

Definition 4.8 (Parameter Registers)

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

$$\texttt{\%rdi}, \texttt{\%rsi}, \texttt{\%rdx}, \texttt{\%rcx}, \texttt{\%r8}, \texttt{\%r9}, \tag{3}$$

respectively.

Definition 4.9 (Return Register)

The return value of a function is stored in the

$$\texttt{\%rax} \tag{4}$$

register.

Definition 4.10 (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 4.11 (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 4.12 (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.

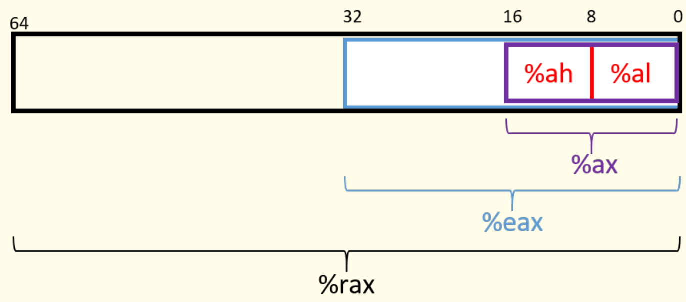


Figure 19: The names that refer to subsets of register `%rax`.

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

4.2.2 ARM Assembly Registers

4.3 Addressing Modes

Registers being 8 bytes mean that we can store memory addresses, and if we can store memory addresses, we can access memory, i.e. the values at those memory addresses. There are 4 ways to do this, called **addressing modes**: immediate, normal, displacement, and indexed. When we parse an instruction, its operands are either

1. Constant (literal) values
2. Registers
3. Memory forms

Definition 4.13 (Immediate Addressing)

Immediate addressing is when the operand is a constant value, used with a $ sign.

$$\texttt{\$val} \tag{5}$$

Definition 4.14 (Normal Addressing)

Normal addressing is when the operand is a register, used with a % sign and the following syntax. The parentheses are used to dereference the memory address like dereferencing a pointer in C.

$$\texttt{(R) = Mem[Reg[R]]} \tag{6}$$

where `R` is the register name, `Reg[R]` is the value in the register, and `Mem[Reg[R]]` is the value in the memory address pointed to by the register.

Definition 4.15 (Displacement Addressing)

When we have a memory address stored in a register, we can add an offset to it to access a different memory address.

$$\texttt{D(R) = Mem[Reg[R] + D]} \tag{7}$$

where `R` is the register name and `D` is a constant displacement that specifies offset.

Definition 4.16 (Indexed Addressing)

Indexed addressing gives us more flexibility, allowing us to multiply the value in the register by a constant and add it to the value in another register. The general formula is shown as the top, but there are special cases:

$$\begin{aligned}
\texttt{D(Rb, Ri, S)} &= \texttt{Mem[Reg[Rb] + S*Reg[Ri] + D]} \\
\texttt{D(Rb, Ri)} &= \texttt{Mem[Reg[Rb] + Reg[Ri] + D]} \\
\texttt{(Rb, Ri, S)} &= \texttt{Mem[Reg[Rb] + S*Reg[Ri]]} \\
\texttt{(Rb, Ri)} &= \texttt{Mem[Reg[Rb] + Reg[Ri]]} \\
\texttt{(, Ri, S)} &= \texttt{Mem[S*Reg[Ri]]}
\end{aligned}$$

where `D` is a constant displacement of 1, 2, or 4 bytes, `Rb` is the base register (can be any of 8 integer registers), `Ri` is the index register (can be any register except `rsp`), and `S` is the scale factor (1, 2, 4, or 8).

4.3.1 x86 Assembly Addressing Modes

Example 4.6 (Immediate Addressing)

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

Example 4.7 (Normal Addressing)

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

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

Example 4.8 (Displacement Addressing)

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

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

Example 4.9 (Indexed Addressing)

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

$$\texttt{0x80(,\%rdx,2) = Mem[2*0xF000 + 0x80] = Mem[0x1E080]} \tag{8}$$

We see that

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

4.3.2 ARM Assembly Addressing Modes

4.4 Instructions

Now that we've gotten a sense of what these registers are and some commonalities between them, let's do some operations on them with instructions.

Definition 4.17 (Instruction)

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

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

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

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

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

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

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

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

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

3. **Control Flow**: What instruction to execute next both unconditional and conditional (if statements) ones. With if statements, loops can then be defined.

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

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

4.4.1 Moving and Arithmetic

Again, it is more important to have a general feel of what instructions every assembly language should and get the ideas down rather than the syntax. We list them here, beginning with simply moving.

Definition 4.18 (Moving)

Next we want to have some sort of arithmetic to do calculations and to compare values.

Definition 4.19 (Arithmetic Operations)

4.4.2 Conditionals

Definition 4.20 (Conditionals)

4.4.3 Control Transfer on Stack

These are really the three basic functions needed to do anything in assembly, but let's talk about an important implementation called the **control transfer**. Say that you want to compute a function.

1. Then we must retrieve the data from the memory.

2. We must load it into our registers in the CPU and perform some computation.
3. Then we must store the data back into memory.

Let's begin with a refresher on how the call stack is managed. Recall that `%rsp` is the stack pointer and always points to the top of the stack. The register `%rbp` represents the base pointer (also known as the frame pointer) and points to the base of the current stack frame. The stack frame (also known as the activation frame or the activation record) refers to the portion of the stack allocated to a single function call. The currently executing function is always at the top of the stack, and its stack frame is referred to as the active frame. The active frame is bounded by the stack pointer (at the top of stack) and the frame pointer (at the bottom of the frame). The activation record typically holds local variables for a function.

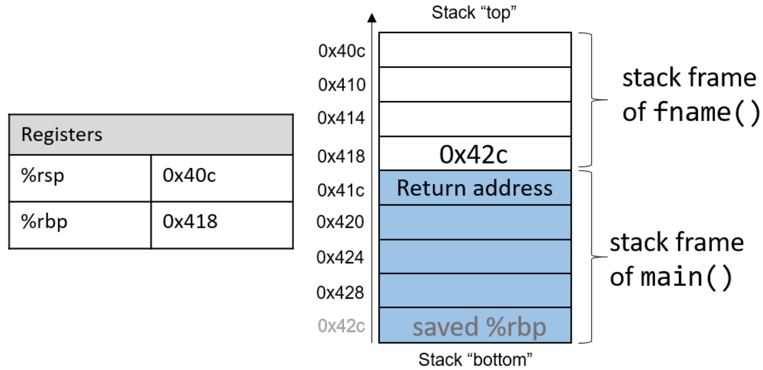


Figure 20: The current active frame belongs to the callee function (fname). The memory between the stack pointer and the frame pointer is used for local variables. The stack pointer moves as local values are pushed and popped from the stack. In contrast, the frame pointer remains relatively constant, pointing to the beginning (the bottom) of the current stack frame. As a result, compilers like GCC commonly reference values on the stack relative to the frame pointer. In Figure 1, the active frame is bounded below by the base pointer of fname, which is stack address 0x418. The value stored at address 0x418 is the "saved" `%rbp` value (0x42c), which itself is an address that indicates the bottom of the activation frame for the main function. The top of the activation frame of main is bounded by the return address, which indicates where in the main function program execution resumes once the callee function fname finishes executing.

Once we have done this we are really done. Formally, this is called Turing complete (?).

Definition 4.21 (Control Transfers)

We list some.
1. Push
2. Pop
3. Call to call a function
4. Return to return from a function
5. Continue
6. Get out of stack with leave.

Example 4.10 (Control Transfer Example)

We show this with a minimal example with psuedocode.

4.4.4 Multiple Functions

Now what happens if there are multiple functions calling each other? Take a look at the following example with two functions.

Example 4.11 (Multiple Functions Example)

There is a bit of a concern here from the previous example. The main function had two functions that returned two values. As the subfunction stack frame is removed from the stack, the return value is stored in the `%rax` register. If another function is called right after, then the return value of the second function will overwrite that of the previous one. This was not a problem in the previous example since the return value of the `assign` function was not used. However, if it was, then the return value of the `adder` function would have overwritten it. This is known as register saving.

1. For **caller-saved registers**, the caller function is responsible for saving the value of the register before calling a function and restoring it after the function returns. The caller should save values in its stack frame before calling the callee function, e.g. by pushing all the return values of each callee in the caller stack frame. Then it will restore values after the call.

   Therefore, if we have a set of registers {`%reg`}, the caller must take everything and push them in the caller stack frame. Then it will restore them after the call.

2. For **callee-saved registers**, it is the callee's responsibility to save any data in these registers before using the registers.

   Therefore, if we have a set of registers {`%reg`}, then inside the callee stack frame, the callee must take everything and push them in the callee stack frame. Once it computes the final return value, then it will restore all the saved register values from the callee stack frame back into the registers for the caller to use.

Ideally, we want *one* calling convention to simply separate implementation details between caller and callee. In general, however, neither is best. If the caller isn't using a register, then caller-save is better, and if callee doesn't need a register, then callee-save is better. If we do need to save, then callee save generally makes smaller programs, so we compromise and use a combination of both caller-save and callee-save.

4.4.5 x86-64 Instructions

Let's talk about moving instructions first.

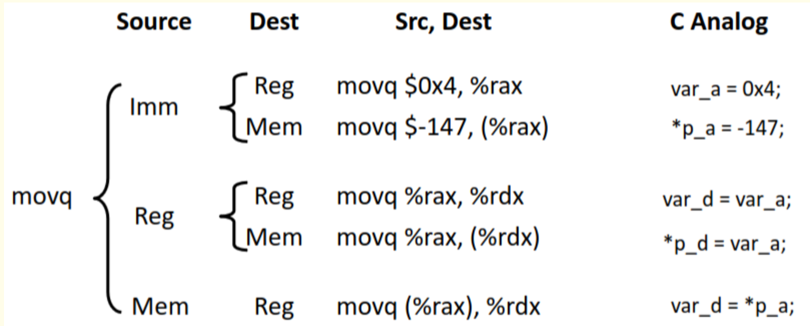
Definition 4.22 (mov)

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

```
mov_ src, dest
```
(9)

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:



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

Example 4.12 (Swap Function)

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

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

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

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

Definition 4.23 (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.

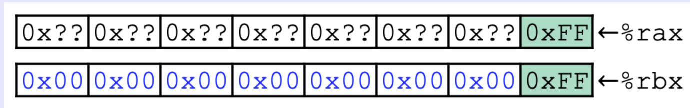
1. The source can be from a memory or register.
2. The destination must be a register.

Example 4.13 (Simple example with movz)

Take a look at the code below.

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

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

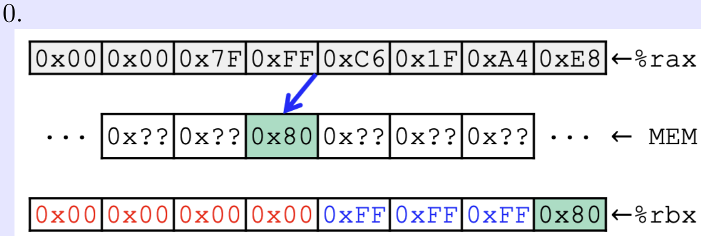


Example 4.14 (Harder example with movs)

Take a look at the code below.

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

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



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

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

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

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

Figure 21: 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 4.24 (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 4.25 (Push and Pop)

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

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

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

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

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

Definition 4.26 (Push and Pop)

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

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

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

Definition 4.27 (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 4.28 (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 4.15 (Evaluating a Function)

Say that we have the following C code.

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

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

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

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

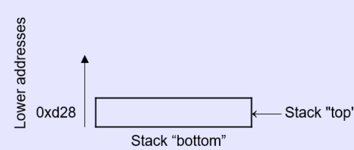
Figure 22: The output of objdump for the `adder2` function. The leftmost column represents the addresses (in hex) of where the actual instructions lie. The second column represents the machine code that is being executed. The third column represents the assembly code.

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

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

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

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



call stack

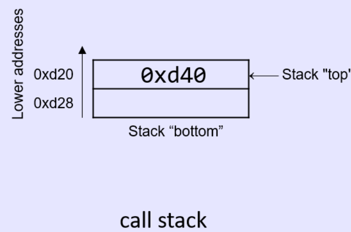
   (a) First, the `%eax` is filled with garbage, which are leftovers from previous programs that haven't been overwritten yet.
   (b) Second, the `%edi=0x28` since we have set `x=40` in `main`, before calling `adder2`, so it lingers on.
   (c) `%rsp=0xd28` since that is where the top of the stack is.
 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

it, but haven't done it yet).

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

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

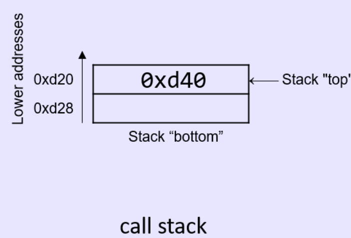
Registers	
%eax	0x123
%edi	0x28
%rsp	**0xd20**
%rbp	0xd40
%rip	**0x527**



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

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

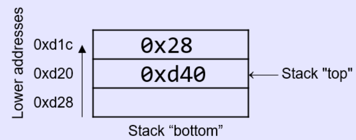
Registers	
%eax	0x123
%edi	0x28
%rsp	0xd20
%rbp	**0xd20**
%rip	**0x52a**



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

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

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

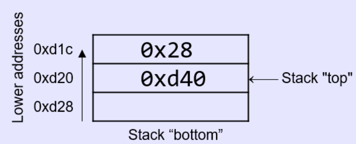


call stack

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**

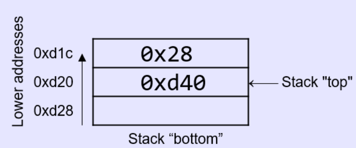


call stack

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

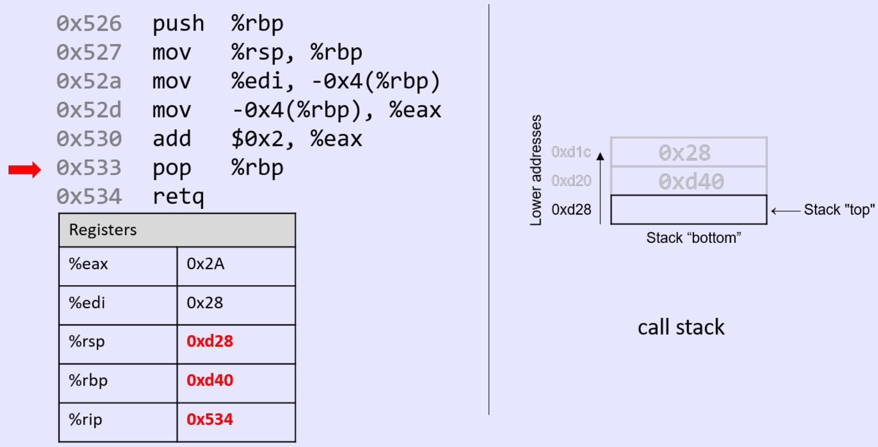
```
0x526  push  %rbp
0x527  mov   %rsp, %rbp
0x52a  mov   %edi, -0x4(%rbp)
0x52d  mov   -0x4(%rbp), %eax
0x530  add   $0x2, %eax
0x533  pop   %rbp
0x534  retq
```

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



call stack

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



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

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

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

and the following two at the end of a function

```
pop %rbp
retq
```

Now arithemtic operations are quite simple.

Definition 4.29 (Add, Subtract, Multiply)

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

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

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

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

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

Definition 4.30 (Increment, Decrement)

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

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

Definition 4.31 (Negative)

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

`neg_ dest`	`dest = -dest`

Example 4.16 (Basic Arithmetic Function)

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

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

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

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

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

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

Definition 4.32 (Load Effective Address)

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

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

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

Definition 4.33 (Bitwise)

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

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

Definition 4.34 (Arithmetic and Logical Bit Shift)

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

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

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

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

Example 4.17 (Harder Arithmetic Example)

The following two codes are equivalent.

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

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

The final thing in our list is condition codes.

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

Definition 4.35 (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 4.18 (Zero Flag)

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

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

Example 4.19 (Sign Flag)

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

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

Example 4.20 (Overflow Flag)

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

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

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

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

Example 4.21 (Carry Flag)

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

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

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

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

Definition 4.36 (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 4.37 (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 4.22 (Compare)

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

$$\texttt{\%bl - \%al = 0x81 - 0x80 = 0x81 + } \sim \texttt{ 0x80 + 1 = 0x81 + 0x7F + 1 = 0x101 = 0x01} \quad (10)$$

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 4.38 (Set)

We can then talk about conditional moves and jumps.

Definition 4.39 (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 4.40 (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 23: 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 24: All jump instructions

Definition 4.41 (int)

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

Definition 4.42 (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 4.43 (Goto Syntax)

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

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

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

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

else_statement:
  smallest = x; //else statement

done:
  return smallest;
}
```

Figure 25: C vs GoTo code of the same function. While GoTo code allows us to view C more like assmebly, it is generally not readable and is not considered best practice.

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

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

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

Figure 26: Assembly code of a simple if statement

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

1. The first mov instruction copies the value located in register %edi (the first parameter, x) and places it at memory location %rbp-0x14 on the call stack. The instruction pointer (%rip) is set to the address of the next instruction, or 0x40059d.
2. The second mov instruction copies the value located in register %esi (the second parameter, y) and places it at memory location %rbp-0x18 on the call stack. The instruction pointer (%rip) updates to point to the address of the next instruction, or 0x4005a0.

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

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

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

Figure 27: Compiled with `gcc -O1 -o getSmallest getSmallest.c`

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

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

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

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

Figure 28: Simple loop function in C and assembly.

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

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

Figure 29: 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 4.44 (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 4.45 (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 4.23 (Calling Functions in Assembly)

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

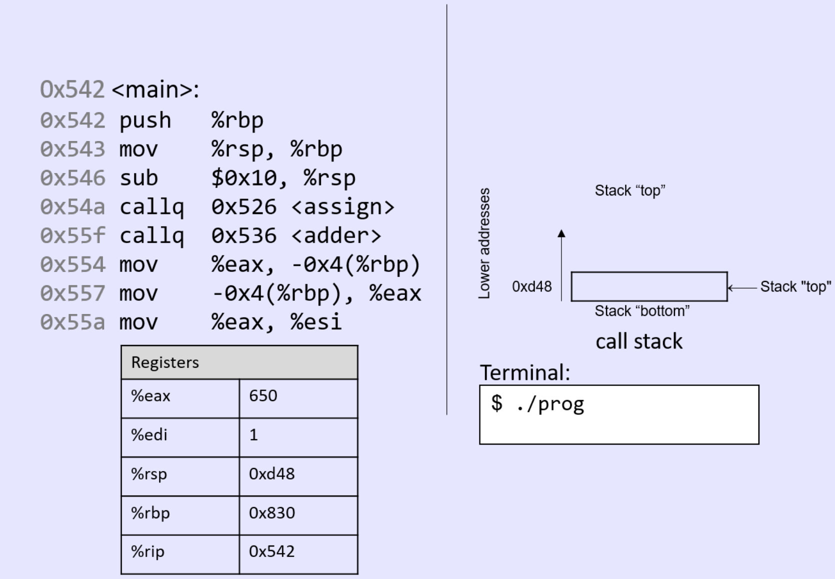
```c
#include <stdio.h>

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

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

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

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

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

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

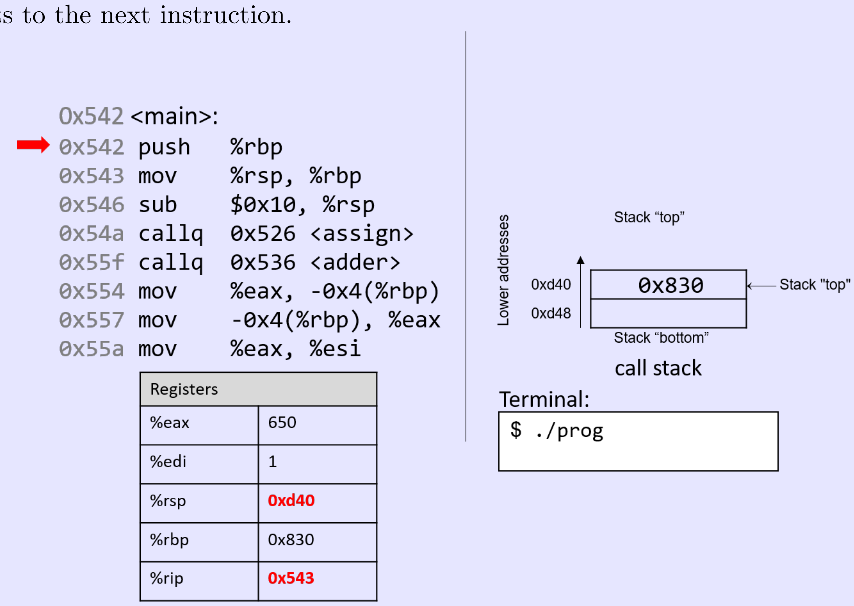
Figure 30: C code and its assembly equivalent. Main function calls two other functions.

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.



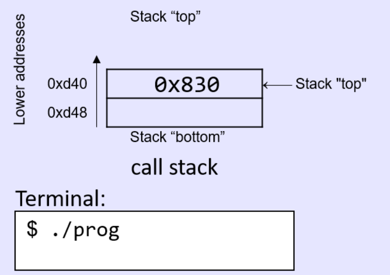
2. Now we start the main function. By calling main, the base pointer `%rbp` of the stack outside of the main frame will be overwritten by the base of the main stack frame, so we must save it for when main is done. Therefore, we push it onto the stack where `%rsp` is pointing. `%rip` now points to the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



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
```

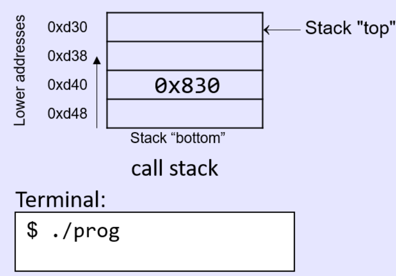
Registers	
%eax	650
%edi	1
%rsp	0xd40
%rbp	**0xd40**
%rip	**0x546**



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
```

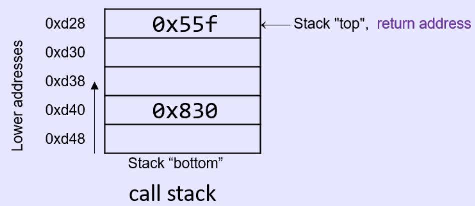
Registers	
%eax	650
%edi	1
%rsp	**0xd30**
%rbp	0xd40
%rip	**0x54a**



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

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

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



Terminal:

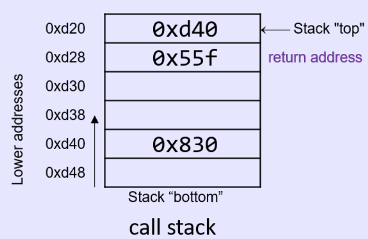
```
$ ./prog
```

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

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

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

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



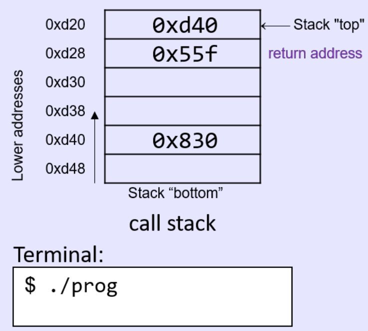
Terminal:

```
$ ./prog
```

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

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

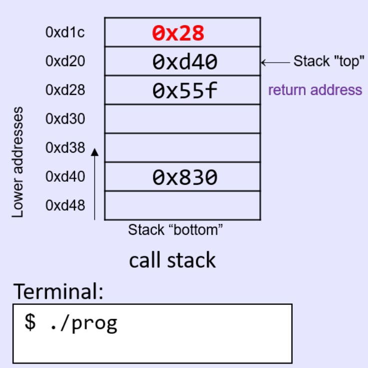
Registers	
%eax	0x0
%edi	1
%rsp	0xd20
%rbp	**0xd20**
%rip	**0x52a**



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
```

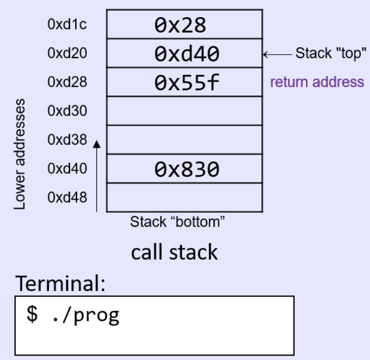
Registers	
%eax	0x0
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	**0x531**



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
```

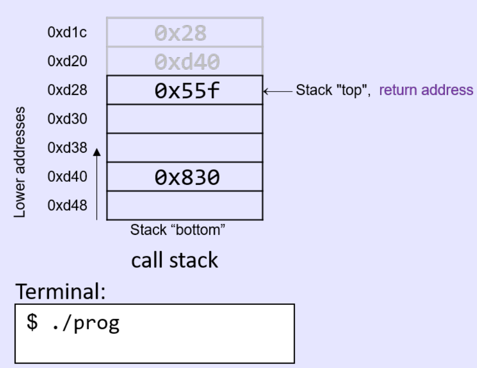
Registers	
%eax	0x28
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	0x534



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
```

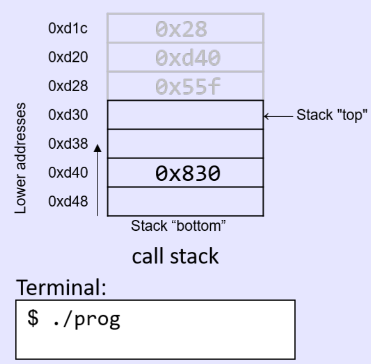
Registers	
%eax	0x28
%edi	1
%rsp	0xd28
%rbp	0xd40
%rip	0x535



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

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

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

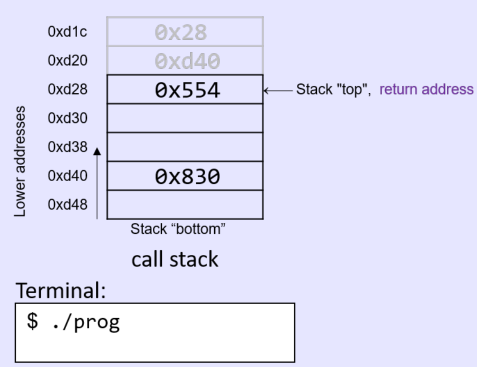


Registers	
%eax	0x28
%edi	1
%rsp	0xd30
%rbp	0xd40
%rip	0x55f

Equivalent to:
pop %rip

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

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

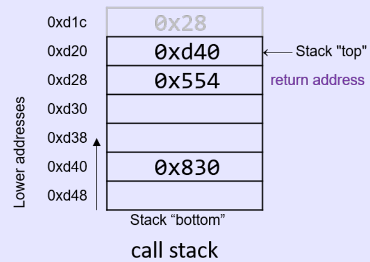


Registers	
%eax	0x0
%edi	1
%rsp	0xd28
%rbp	0xd40
%rip	0x536

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

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

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



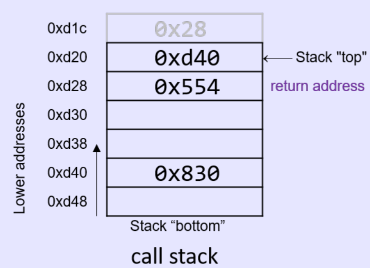
Terminal:

```
$ ./prog
```

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

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

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

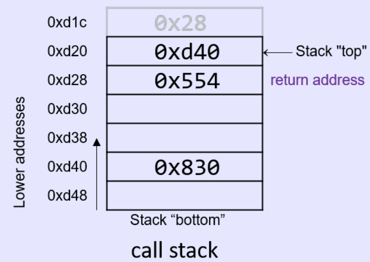


Terminal:

```
$ ./prog
```

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

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



Terminal:

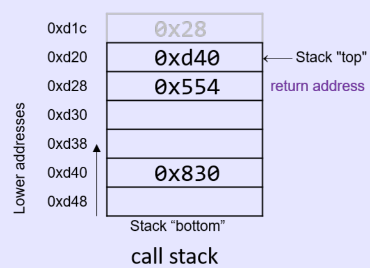
```
$ ./prog
```

Using an old value on the stack!

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

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
```



Terminal:

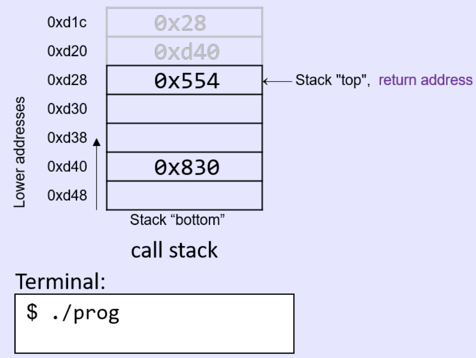
```
$ ./prog
```

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

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

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

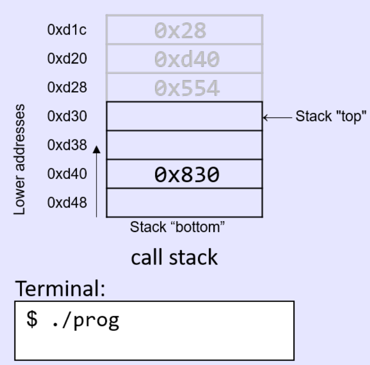
Registers	
%eax	0x2A
%edi	1
%rsp	0xd28
%rbp	0xd40
%rip	0x541



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

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

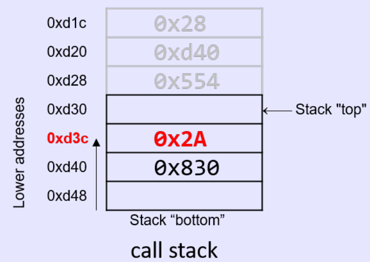
Registers	
%eax	0x2A
%edi	1
%rsp	0xd30
%rbp	0xd40
%rip	0x554



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
```



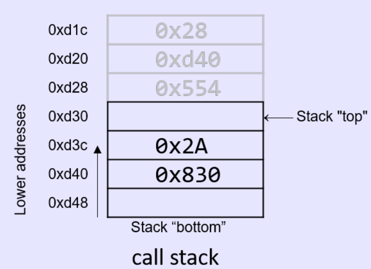
Terminal:

```
$ ./prog
```

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

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
```

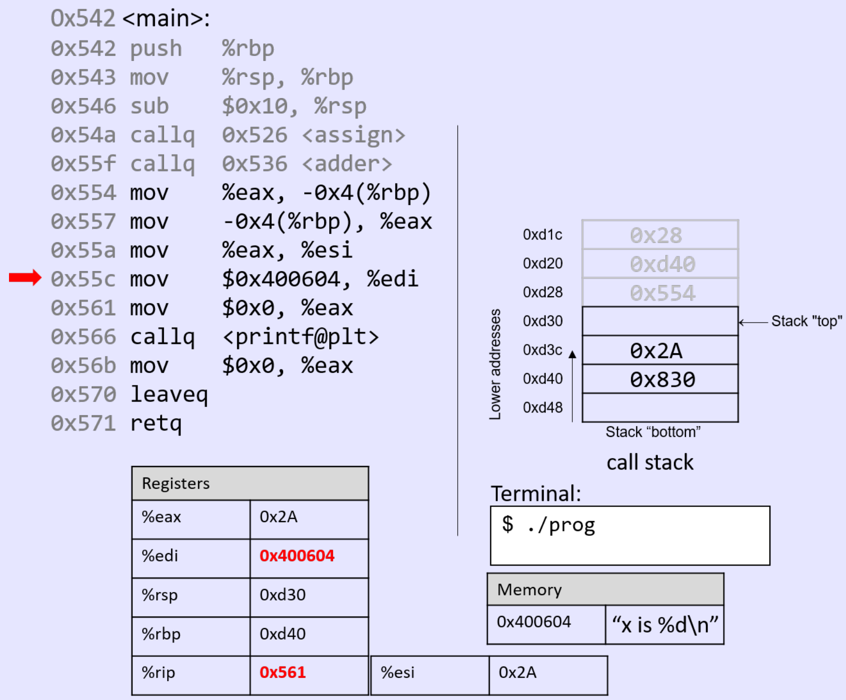


Terminal:

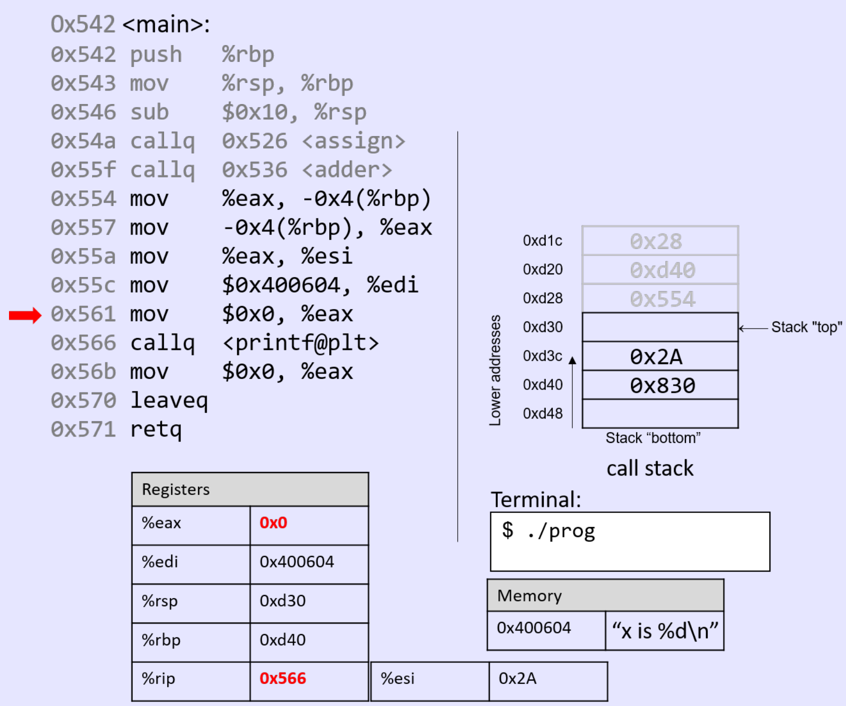
```
$ ./prog
```

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

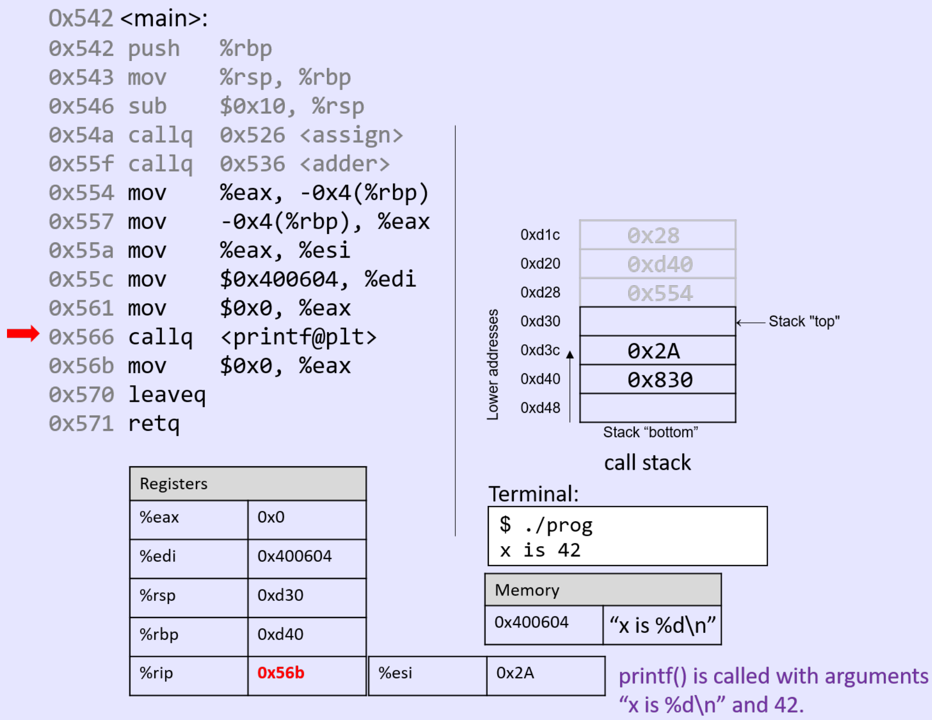
21. Now we want to retrieve the first argument of the print function. The address at `$0x400604` is some address in the code segment memory that holds the string `"x is %d\n"`.



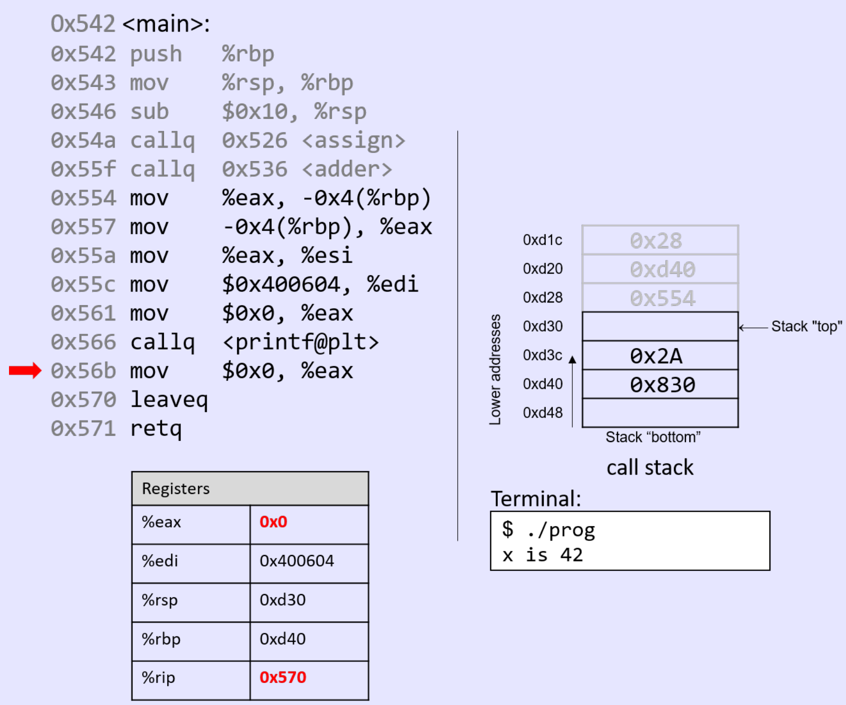
22. Then we move 0 into the `%eax` register to clear it.



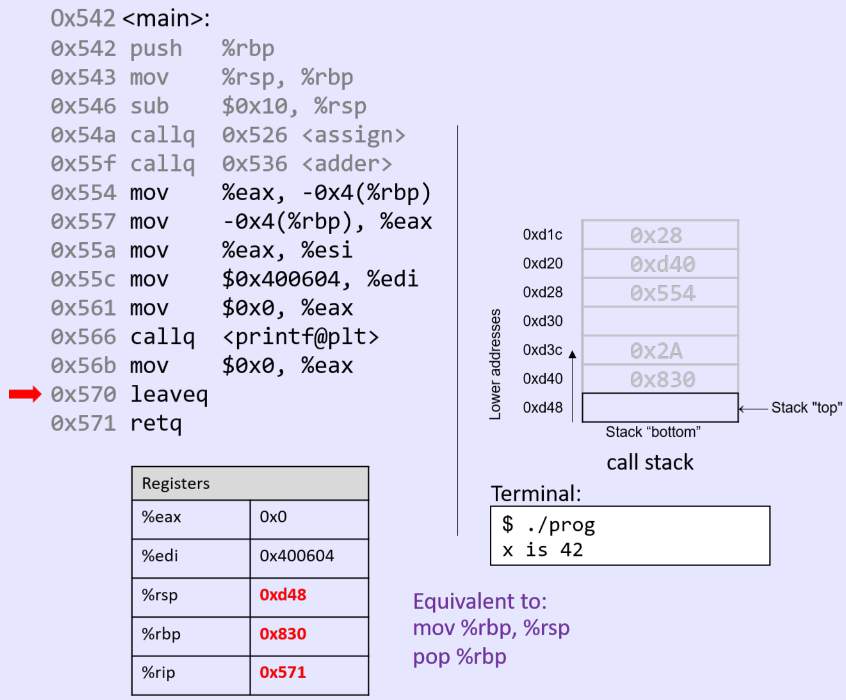
23. We then call the `printf` function, which we won't trace through but it outputs to stdout.



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`.



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`.



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

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

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

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

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

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

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

4.4.6 ARM Instructions

4.4.7 Buffer Overflows

5 Storage Hierarchy

There are different types of memory, with three key components that we should think about:

1. The **capacity**, i.e. amount of data, it can store (how large the water tank is).
2. The **latency**, i.e. amount of time it takes for a device to respond with data after it has been instructed to perform a data retrieval operation (how fast the data flows).
3. The **transfer rate**, i.e. amount of data that can be moved between the device and main memory (how wide the pipe is).

We must provide a good balance of these three qualities, and also note that there are some physical limitations (i.e. latency cannot be faster than speed of light). The highest level categorization of memory is between primary and secondary storage devices, which simply distinguishes the memory that is directly accessible by the CPU and memory that is not.

Definition 5.1 (Primary Storage)

Primary storage devices are directly accessible by the CPU and are used to store data that is currently being processed. This includes CPU registers, cache memory, and RAM. There are two primary ways:

1. **Static RAM (SRAM)** stores data in small electrical circuits (e.g. latches) and is typically the fastest type of memory. However, it is more expensive to build, consumers more power, and occupies more space, limiting the SRAM storage.
2. **Dynamic RAM (DRAM)** stores data using electrical components (e.g. capacitors) that hold an electrical charge. It is called *dynamic* because a DRAM system must frequently refresh the charge of its capacitors to maintain a stored value..

Device	**Capacity**	**Approx. latency**	**RAM type**
Register	4 - 8 bytes	$<$ 1 ns	SRAM
CPU cache	1 - 32 megabytes	5 ns	SRAM
Main memory	4 - 64 gigabytes	100 ns	DRAM

Table 4: Memory hierarchy characteristics

Definition 5.2 (Secondary Storage)

Secondary storage devices are not directly accessible by the CPU and are used to store data that is not currently being processed. This includes hard drives, SSDs, and magnetic tapes.

The figure overviews the different types of memory.

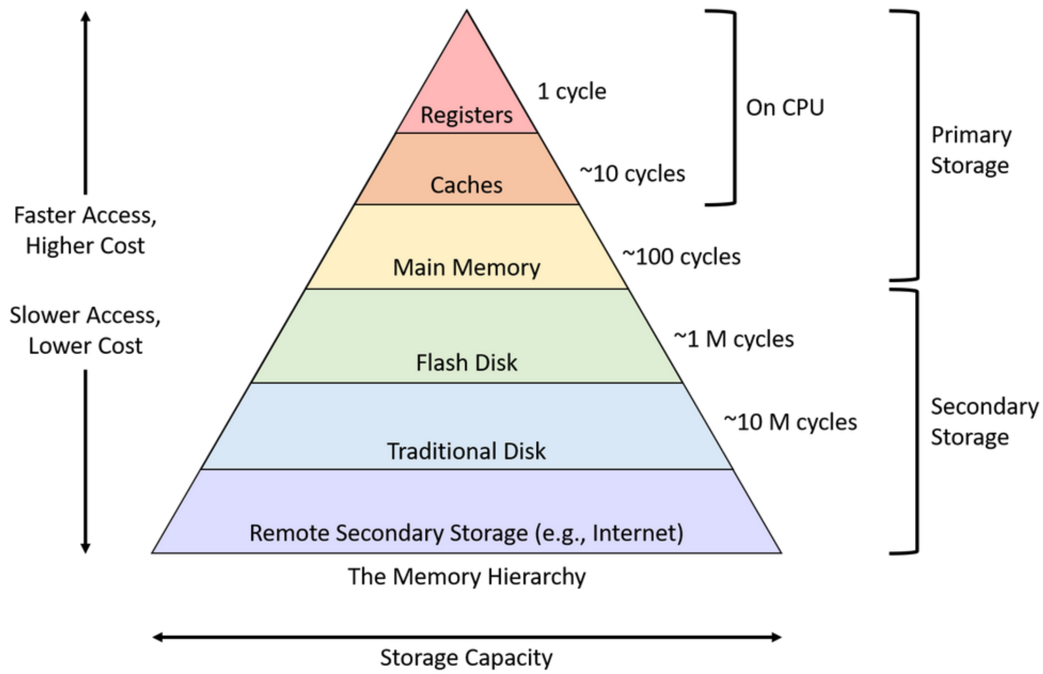


Figure 31: Memory hierarchy.

5.1 Locality

So far, we have abstracted away most of these memory types as a single entity with nearly instantaneous access, but in practice this is not the case. The most simple way is to simply have RAM and our CPU registers, but by introducing more intermediate memory types, we can achieve greater efficiency.

Definition 5.3 (Locality)

Locality is a principle that generally states that a program that accesses a memory location n at time t is likely to access memory location $n + \epsilon$ at time $t + \epsilon$. This principle motivates the design of efficient caches.

1. **Temporal locality** is the idea that if you access a memory location, you are likely to access it again soon.
2. **Spatial locality** is the idea that if you access a memory location, you are likely to access nearby memory locations soon.

This generally means that if you access some sort of memory, the values around that address is also likely to be accessed and therefore it is wise to store it closer to your CPU.

Example 5.1 (Locality)

Consider the following code.

```c
int sum_array(int *array, int len) {
  int i;
  int sum = 0;

  for (i = 0; i < len; i++) {
    sum += array[i];
  }
```

```

  return sum;
}
```

1. The repetitive nature of the for loop exploits temporal locality. More specifically, the CPU accesses the same memory (stored in variables `i`, `len`, `sum`, `array`) within each iteration and therefore at similar times.
2. The spatial locality is exploited when the CPU accesses memory locations from each element of the array, which are contiguous in memory. Even though the program accesses each array element only once, a modern system loads more than one `int` at a time from memory to the CPU cache. That is, accessing the first array index fills the cache with not only the first integer but also the next few integers after it too. Exactly how many additional integers get moved depends on the cache's **block size**. For example, a cache with a 16 byte block size will store `array[i]` and the elements in `i+1`, `i+2`, `i+3`.

We can see the differences in spatial locality in the following example.

Example 5.2 ()

One may find that simply changing the order of loops can cause a significant speed up in your program. Consider the following code.

```c
float averageMat_v1(int **mat, int n) {
  int i, j, total = 0;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      // Note indexing: [i][j]
      total += mat[i][j];
    }
  }
  return (float) total / (n * n);
}
```

```c
float averageMat_v2(int **mat, int n) {
  int i, j, total = 0;

  for (j = 0; j < n; j++) {
    for (i = 0; i < n; i++) {
      total += mat[i][j];
    }
  }
  return (float) total / (n * n);
}
```

Figure 32: Two implementations of taking the total sum of all elements in a matrix.

It turns out that the left hand side of the code executes about 5 times faster than the second version. Consider why. When we iterate through the `i` first and then the `j`, we access the values `array[i][j]` and then by spatial locality, the next few values in the array, which are `array[i][j+1]`, ... are stored in the cache.

1. In the left hand side of the code, these next stored values are exactly what is being accessed, and the CPU can access them in the cache rather than having to go into memory.
2. In the right hand side of the code, these next values are *not* being accessed since we want to access `array[i+1][j]`, ... Unfortunately, this is not stored in the cache and so for every n^2 loops we have to go back to the memory to retrieve it.

5.2 RAM

5.3 Caches

Valgrind's cachegrind mode.

5.4 SSD

5.5 HDD

6 Compiling and Linking

Now let's talk about how this compiling actually happens. *Compiling* is actually an umbrella term that is misused. Turning at C file into an executable file consists of multiple intermediate steps, one of which is actually compiling, but the whole series is sometimes referred to as compiling. A more accurate term would be *building*. Before we get onto it, there are two types of compilers.

Definition 6.1 (GCC, CLang)

The two mainstream compilers used is GCC (with the gdb debugger) and Clang (with lldb). For now, the difference is that

1. gcc is more established.
2. clang is newer and has more features.

A useful flag to know is that we can always specify the name of the (final or intermediary) output file with the `-o` flag.

Definition 6.2 (Complete Build Process)

To actually turn a C file into an executable file, we need to go through a series of steps. We start off with the C code, which are the `.c`, `.cpp`, or `.h` files.

1. **Preprocessing**: The precompiler step expands the *preprocessor directives* (all the `#include` and `#define` statements) and removes comments. This results in a `.i` file. The preprocessor will replace these macros with the actual code. This results in a `.i` file.

```
clang/gcc -E main.c -o main.i
```

2. **Compiling**: We take these and generate assembly code. This results in a `.asm` or `.s` file.

```
clang/gcc -S main.c -o main.s
```

3. **Assembler**: We take the assembly code and generate machine code in the form of relocatable binary object code (this is machine code, not assembly). This results in a `.o` or `.obj` file.

```
clang/gcc -c main.c -o main.o
```

4. **Linking**: We take these object files and link them together to form an executable file. This results in a `.exe` or `.out` file.

The GCC or CLang compiler automates this process for us. For example, `gcc -c hello.c` generates an object file, taking care of the preprocessing, compiling, and assembling code. Then, `gcc hello.o` links the object file to generate an executable file.

There are a lot of questions to be asked here, and we will go through them step by step.

6.1 Precompiling Stage

Just like how Python package managers like conda have specific directories that they find package in, the C library also has a certain directory.

Definition 6.3 (Standard Library Directory)

In Linux systems, there are two main directories you look at:

1. `/usr/include` contains the standard C library headers.
2. `/usr/local/include` contains the headers for libraries that you install yourself.

In Mac Silicon, these directories are a little bit more involved. You must first install the xcode command line developer tools, which will then create these directories.

1. The standard C library headers are in

   `/Library/Developer/CommandLineTools/SDKs/MacOSX*.sdk/usr/include`.

In here, we can find all the relevant import files like `stdio.h` and such. When we precompile, the output `.i` file represents a precompiled C file. It still has C code, but it has been optimized to

1. Remove comments.
2. Replace all the `#include` statements with the actual code.
3. Replace all the global variables declared in `#define` with the actual value.

Between x86 and ARM, there are no significant differences in how C files are precompiled.

Example 6.1 ()

Take a look at the following minimal example.

```c
#include "second.h"
#define a 3

int add(int x, int y) {
  return x + y;
}

int main() {
  // test comment
  int b = 5;
  int c = add(a, b);
  int d = subtract(a, b);
  return 0;
}
```

```c
int subtract(int a, int b) {
  return a - b;
}
.
.
.
.
.
.
.
.
.
.
.
```

Figure 33: I have included a `main.c` file that imports statements from a `second.h` file.

Now, I run `gcc -E main.c -o main.i` to generate the precompiled file, which gives me the following.

```
# 1 "main.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 418 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "main.c" 2
# 1 "./second.h" 1
int subtract(int a, int b) {
  return a - b;
}
# 2 "main.c" 2


int add(int x, int y) {
  return x + y;
}

int main() {

  int b = 5;
  int c = add(3, b);
  int d = subtract(3, b);
  return 0;
}
```

Figure 34: The precompiled file.

Notice a few things:

1. The header file `second.h` has been replaced with the actual code.
2. The comments have indeed been removed.
3. The global variable `a` has been replaced with the actual value 3.

This leaves us with the question of what all the rest of the lines that start with a `#` are for. They are called *preprocessor directives*.

Definition 6.4 (Preprocessor Directives)

Preprocessor directives are commands that are executed before the actual compilation begins. These directives allow additional actions to be taken on the C source code before it is compiled into object code. Directives are not part of the C language itself, and they are always prefixed with a `#` symbol.

1. `#include` is used to include the contents of a file into the source file. It selects portions of the file to include based on the file name.
2. `#define` is used to define a macro, which is a way to give a name to a constant value or a piece of code.
3. `#ifdef`, `#ifndef`, `#else`, and `#endif` are used for conditional compilation.
4. `#error` is used to generate a compilation error.
5. `#pragma` is used to give the compiler specific instructions.

6.2 Compiling Stage

Once we have precompiled, we can compile the code into assembly code. For the following two examples, we will parse through the general syntax of assembly code. It is quite different between x86 and ARM, so we will use the minimal C code

```
int add(int x, int y) {
  return x + y;
}

int main() {
  int a = 3;
  int b = 5;
  int c = add(a, b);
  return 0;
}
```

for both examples.

Example 6.2 (x86 Compiled Assembly Language)

The assmebly code is shown.

```
.
  .file  "main.c"
  .text
  .globl add
  .type  add, @function
add:
.LFB0:
  .cfi_startproc
  endbr64
  pushq  %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq   %rsp, %rbp
  .cfi_def_cfa_register 6
  movl   %edi, -4(%rbp)
  movl   %esi, -8(%rbp)
  movl   -4(%rbp), %edx
  movl   -8(%rbp), %eax
  addl   %edx, %eax
  popq   %rbp
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
.LFE0:
  .size  add, .-add
  .globl main
  .type  main, @function
main:
.LFB1:
  .cfi_startproc
  endbr64
  pushq  %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
```

```
  movq   %rsp, %rbp
  .cfi_def_cfa_register 6
  subq   $16, %rsp
  movl   $3, -12(%rbp)
  movl   $5, -8(%rbp)
  movl   -8(%rbp), %edx
  movl   -12(%rbp), %eax
  movl   %edx, %esi
  movl   %eax, %edi
  call   add
  movl   %eax, -4(%rbp)
  movl   $0, %eax
  leave
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
.LFE1:
  .size  main, .-main
  .ident "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0"
  .section  .note.GNU-stack,"",@progbits
  .section  .note.gnu.property,"a"
  .align 8
  .long   1f - 0f
  .long   4f - 1f
  .long   5
0:
  .string    "GNU"
1:
  .align 8
  .long   0xc0000002
  .long   3f - 2f
2:
  .long   0x3
3:
  .align 8
4:
```

Example 6.3 (ARM Compiled Assembly Language)

The assembly code is shown.

```
.
  .section  __TEXT,__text,regular,pure_instructions
  .build_version macos, 14, 0 sdk_version 14, 4
  .globl _add                          ; -- Begin function add
  .p2align  2
_add:                                 ; @add
  .cfi_startproc
; %bb.0:
  sub sp, sp, #16
  .cfi_def_cfa_offset 16
  str w0, [sp, #12]
  str w1, [sp, #8]
  ldr w8, [sp, #12]
  ldr w9, [sp, #8]
```

```
  add w0, w8, w9
  add sp, sp, #16
  ret
  .cfi_endproc
                                        ; -- End function
  .globl _main                          ; -- Begin function main
  .p2align  2
_main:                                  ; @main
  .cfi_startproc
; %bb.0:
  sub sp, sp, #48
  .cfi_def_cfa_offset 48
  stp x29, x30, [sp, #32]             ; 16-byte Folded Spill
  add x29, sp, #32
  .cfi_def_cfa w29, 16
  .cfi_offset w30, -8
  .cfi_offset w29, -16
  mov w8, #0
  str w8, [sp, #12]                   ; 4-byte Folded Spill
  stur   wzr, [x29, #-4]
  mov w8, #3
  stur   w8, [x29, #-8]
  mov w8, #5
  stur   w8, [x29, #-12]
  ldur   w0, [x29, #-8]
  ldur   w1, [x29, #-12]
  bl  _add
  mov x8, x0
  ldr w0, [sp, #12]                   ; 4-byte Folded Reload
  str w8, [sp, #16]
  ldp x29, x30, [sp, #32]             ; 16-byte Folded Reload
  add sp, sp, #48
  ret
  .cfi_endproc
                                        ; -- End function
.subsections_via_symbols
```

We can see that in both examples, there are generally two types of codes.

1. The regular CPU operations with registers and memory.
2. Some code starts off with some code that starts with a `.`. Every line that starts with a `.` are called *assembler directives*.

Let's elaborate more on what these directives are.

Definition 6.5 (Assembler Directives)

An **assembler directives** are instructions in assembly language programming that that give commands to the assembler (which then converts this to an object file) about various aspects of the assembly process, but they do not represent actual CPU instructions that execute in the program. Unlike typical assembly language instructions that directly manipulate registers and execute arithmetic or logical operations, directives are used to organize, control, and provide necessary information for the assembly and linking of binary programs. They can manage memory allocation, define symbols, control compilation settings, and much more.
There are general types of directives that are common in both x86 and ARM that we should be aware

about:
1. Section directives.
2. Data allocation directives.
3. Symbol definition directives.
4. Macro and Include directives.
5. Debugging and error handling directives.

Example 6.4 (x86 Assembly Directives)

Let us elaborate on the specific directives in the x86 assembly code, some of which are in the example above.
1. `.file "main.c"` is a directive that tells the assembler that the following code is from the file `main.c`. It is a form of metadata.
2. `.text` is a directive that tells the assembler that the following code is the text section (the text/code portion of memory) of the program. This is where the actual code is stored.
3. `.globl add` is a directive that tells the assembler that the following code is a global function called `add`.
4. `.type add, @function` is a directive that tells the assembler that the following code is a function.

Example 6.5 (ARM Assembly Directives)

You also see that there are symbols that represent memory addresses. Let's elaborate on what symbols mean.

Definition 6.6 (Symbol)

A **symbol** is a name that is used to refer to a memory location. It can be a function name, a global variable, or a local variable.
1. Global symbols are symbols that can be referenced by other object files, e.g. non-static functions and global variables.
2. Local symbols are symbols that are only visible within the object file, e.g. static functions and local variables. The linker won't know about these types.
3. External symbols are referenced by this object file but defined in another object file.

6.3 Objdump

Since we will be using the `objdump` package quite a lot, it is worth mentioning the different commands you will use and store them here as a reference. For first readers, don't expect to know what each of them do, but rather look back at this for a reference.

6.3.1 ELF and Mach-O Formats

Objdump is a command line utility that is used to display information about object files, which are often outputted in a specific format. The two main output file types are called ELF (Executable and Linkable Format) and Mach-O (Mach Object).

Definition 6.7 (ELF)

The **Executable and Linkable Format** (ELF) is a common standard file format for executables, object code, shared libraries, and core dumps. It is analogous to a book, with the following parts:

1. **Header**, which is like the cover of the book. It contains metadata about the file, such as the architecture, the entry point, and the sections.
2. **Sections**, which are like chapters. Each section contains the content for some given purpose or use wthin the program. e.g. `.binary` is just a block of bytes, `.text` contains the machine code, `.data` contains initialized data, and `.bss` contains uninitialized data.
3. **Symbol Table**, is like a detailed table of contents of all defined symbols such as functions, external (global) variables, local maps, etc.
4. **Relocation records**, which is like the index of the book that lists references to symbols.

The format is generally as such when you run `objdump -d -r hello.o` (d represents disassembly and r represents relocation entries).

```
ELF header         # file type

.text section
  - code goes here

.rodata section
  - read only data

.data section
  - initialized global variables

.bss section
  - uninitialized global variables

.symtab section
  - symbol table (symbol name, type, address)

.rel.text section
  - relocation entries for .text section
  - addresses of instructions that will need to be modified in the executable.

.rel.data section
  - relocation info for .data section
  - addresses of pointer data that will need to be modified in the merged executable.

.debug section
  - info for symbolic debugging (gcc -g)
```

Definition 6.8 (Mach-O)

6.3.2 Objdump Commands

Theorem 6.1 (File Headers with Objdump)

Given that you have an object file, the first thing you might want to do is see the file header. You do with this `objdump -f main.o`.

```
main.o:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000
```

Theorem 6.2 (Section with Objdump)

To look at the section headers to get a closer overview, you use `objdump -h main.o`.

```
main.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         0000004b  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  0000008b  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  0000008b  2**0
                  ALLOC
  3 .comment      0000002c  0000000000000000  0000000000000000  0000008b  2**0
                  CONTENTS, READONLY
  4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000b7  2**0
                  CONTENTS, READONLY
  5 .note.gnu.property 00000020  0000000000000000  0000000000000000  000000b8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .eh_frame     00000058  0000000000000000  0000000000000000  000000d8  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

Theorem 6.3 (Disassembly with Objdump)

Now you might actually want to look at the disassembly of the code, which is what we often use it for. To do this, you use `objdump -D main.o` to get the entire output.

1. The leftmost column represents the address of the instruction.
2. The next column represents the machine code of the instruction.
3. The next column represents the assembly code of the instruction.

```
main.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <add>:
   0: f3 0f 1e fa             endbr64
   ...
  17: c3                      retq

0000000000000018 <main>:
  18: f3 0f 1e fa             endbr64
  ...
  4a: c3                      retq

Disassembly of section .comment:

0000000000000000 <.comment>:
   0: 00 47 43                add    %al,0x43(%rdi)
   ...
  2a: 30 00                   xor    %al,(%rax)

Disassembly of section .note.gnu.property:

0000000000000000 <.note.gnu.property>:
```

```
   0: 04 00                   add    $0x0,%al
  ...

Disassembly of section .eh_frame:

0000000000000000 <.eh_frame>:
   0: 14 00                   adc    $0x0,%al
  ...
```

If you just want to look at the contents of the executable sections, then you can use `objdump -d main.o`.

```
main.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <add>:
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 89 7d fc                mov    %edi,-0x4(%rbp)
   b: 89 75 f8                mov    %esi,-0x8(%rbp)
   e: 8b 55 fc                mov    -0x4(%rbp),%edx
  11: 8b 45 f8                mov    -0x8(%rbp),%eax
  14: 01 d0                   add    %edx,%eax
  16: 5d                      pop    %rbp
  17: c3                      retq

0000000000000018 <main>:
  18: f3 0f 1e fa             endbr64
  1c: 55                      push   %rbp
  1d: 48 89 e5                mov    %rsp,%rbp
  20: 48 83 ec 10             sub    $0x10,%rsp
  24: c7 45 f4 03 00 00 00    movl   $0x3,-0xc(%rbp)
  2b: c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
  32: 8b 55 f8                mov    -0x8(%rbp),%edx
  35: 8b 45 f4                mov    -0xc(%rbp),%eax
  38: 89 d6                   mov    %edx,%esi
  3a: 89 c7                   mov    %eax,%edi
  3c: e8 00 00 00 00          callq  41 <main+0x29>
  41: 89 45 fc                mov    %eax,-0x4(%rbp)
  44: b8 00 00 00 00          mov    $0x0,%eax
  49: c9                      leaveq
  4a: c3                      retq
```

If you want to see the source code intermixed with disassembly, then you can use the `-S` flag, but make sure that the object file is a generated with debugging information, i.e. use `gcc -c -g main.c -o main.o`.

```
main.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <add>:
int add(int x, int y) {
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 89 7d fc                mov    %edi,-0x4(%rbp)
   b: 89 75 f8                mov    %esi,-0x8(%rbp)
  return x + y;
   e: 8b 55 fc                mov    -0x4(%rbp),%edx
  11: 8b 45 f8                mov    -0x8(%rbp),%eax
  14: 01 d0                   add    %edx,%eax
}
  16: 5d                      pop    %rbp
  17: c3                      retq

0000000000000018 <main>:

int main() {
  18: f3 0f 1e fa             endbr64
  1c: 55                      push   %rbp
  1d: 48 89 e5                mov    %rsp,%rbp
  20: 48 83 ec 10             sub    $0x10,%rsp
  int a = 3;
  24: c7 45 f4 03 00 00 00    movl   $0x3,-0xc(%rbp)
  int b = 5;
  2b: c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
  int c = add(a, b);
  32: 8b 55 f8                mov    -0x8(%rbp),%edx
  35: 8b 45 f4                mov    -0xc(%rbp),%eax
  38: 89 d6                   mov    %edx,%esi
  3a: 89 c7                   mov    %eax,%edi
  3c: e8 00 00 00 00          callq  41 <main+0x29>
  41: 89 45 fc                mov    %eax,-0x4(%rbp)
  return 0;
  44: b8 00 00 00 00          mov    $0x0,%eax
}
  49: c9                      leaveq
  4a: c3                      retq
```

Figure 35: Disassembly of the object file back into assembly using `objdump -d -S main.o`.

Note that you can always see this disassembly with debuggers like gdb or lldb, but objdump generally works for all architectures.

Theorem 6.4 (Symbol Table)

If you want to look at all the symbols existing within the object file, you use `objdump -t main.o` (t for table of symbols).

1. The leftmost column represents the address of the symbol.

2. The next column represents the type of the symbol. The `g` and `l` represent global and local symbols, respectively. The `O` and `F` represent object and function symbols, while the `UND` and `ABS` represent undefined and absolute symbols.
3. The next column represents the section that the symbol is in.
4. The next column represents the size of the symbol.
5. The last column represents the name of the symbol.

```
main.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*   0000000000000000 main.c
0000000000000000 l    d  .text   0000000000000000 .text
0000000000000000 l    d  .data   0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack  0000000000000000 .note.GNU-stack
0000000000000000 l    d  .note.gnu.property  0000000000000000 .note.gnu.property
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment   0000000000000000 .comment
0000000000000000 g     F .text   0000000000000018 add
0000000000000018 g     F .text   0000000000000033 main
```

Theorem 6.5 (Relocation Table)

If you want to look then at the relocation table, then you use `objdump -r main.o`.

1. The leftmost column represents the offset of the relocation (i.e. the location within the section where this relocation needs to be applied).
2. The second column represents the type of relocation.
3. The third column represents the symbol that this relocation references.

```
main.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000003d R_X86_64_PLT32    add-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
0000000000000040 R_X86_64_PC32     .text+0x0000000000000018
```

6.4 Assembling Stage and Object Files

Now, once you have gotten the object file, you cannot simply open it up in a text edit as it is in machine code. To actually interpret anything from it, you must **disassmble** it, meaning that you convert the machine code back into assembly code. The main software that you use to do this is `objdump`. Let's take a look again at the object file.

```
Disassembly of section .text:

0000000000000000 <add>:
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 89 7d fc                mov    %edi,-0x4(%rbp)
   b: 89 75 f8                mov    %esi,-0x8(%rbp)
   e: 8b 55 fc                mov    -0x4(%rbp),%edx
  11: 8b 45 f8                mov    -0x8(%rbp),%eax
  14: 01 d0                   add    %edx,%eax
  16: 5d                      pop    %rbp
  17: c3                      retq

0000000000000018 <main>:
  18: f3 0f 1e fa             endbr64
  1c: 55                      push   %rbp
  1d: 48 89 e5                mov    %rsp,%rbp
  20: 48 83 ec 10             sub    $0x10,%rsp
  24: c7 45 f4 03 00 00 00    movl   $0x3,-0xc(%rbp)
  2b: c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
  32: 8b 55 f8                mov    -0x8(%rbp),%edx
  35: 8b 45 f4                mov    -0xc(%rbp),%eax
  38: 89 d6                   mov    %edx,%esi
  3a: 89 c7                   mov    %eax,%edi
  3c: e8 00 00 00 00          callq  41 <main+0x29>
  41: 89 45 fc                mov    %eax,-0x4(%rbp)
  44: b8 00 00 00 00          mov    $0x0,%eax
  49: c9                      leaveq
  4a: c3                      retq
```

Figure 36: Disassembly of the object file back into assembly using `objdump -d main.o`.

Let's note a couple things.

1. The functions are organized by their starting address followed by their name, e.g.

   ```
   0000000000000000 <add>:
   ```

   Within each function, each line of assembly code is shown. To find the total memory the function takes up, you can just take the address of the last line and subtract it from the address of the first line. Or you can literally count the number of bytes in each line (remember 2 hex is 1 byte).

2. The line that calls the `add` function is `0x0` (`00 00 00 00`), with is the *relative target address* intended to be filled in by the linker. The actual assembly line just says that the function continues on to the next line at address `0x41`. This is because the object file is not aware of where it will be loaded into memory, and all lines with this opcode `e8 00 00 00 00` is intended to be filled in by the linker.

3. Look at address `0x3c`. It is calling another function, but the values starting from address `0x3d` is `00 00 00 00`, which is not the actual address of the function but also a dummy address. This is because the object file is not aware of where the function is located in memory.

6.5 Linking Stage and Relocation

6.5.1 Relocation

If the object file is already in machine code, then why do we need a separate linking stage that converts `main.o` into `main` the binary? The reason is stated in the previous section: because the object files uses relative memory addressing and does not know about which memory is accessed in other object files, we need to **relocate** the symbols in the object file to their proper addresses. So how does the linker actually know how to relocate these symbols into their proper addresses? It uses the *relocation table*, which contains information about the addresses that need to be modified in the object file.

```
main.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000003d R_X86_64_PLT32    add-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
0000000000000040 R_X86_64_PC32     .text+0x0000000000000018
```

Figure 37: Relocation table for `main.o` object file.

Let's talk about how to actually read this table. We can look at the first entry, which shows an offset of `0x3d`. This represents the offset from the beginning of the `.text` section where the relocation needs to be applied. Looking back at the disassembly file, this address `0x3d` is precisely where there was a dummy address `00 00 00 00`. We want to replace this with the actual address defined in the `VALUE` column, which is `add` (with a slight offset of `0x4`, which is typically used to compensate for the PC-relative addressing mode where the CPU might be adding the length of the instruction to the program counter (PC) before the relocation value is applied). The type of relocation won't be covered in our scope. Let's go through each relocation entry:

1. The first entry is for the `add` function. If we look at the disassembly, within the `main` function, the address `0x3d` is where the `add` function is called. The linker will replace the dummy address with the actual address of the `add` function.

```
Disassembly of section .text:

0000000000000000 <add>:
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 89 7d fc                mov    %edi,-0x4(%rbp)
   b: 89 75 f8                mov    %esi,-0x8(%rbp)
   e: 8b 55 fc                mov    -0x4(%rbp),%edx
  11: 8b 45 f8                mov    -0x8(%rbp),%eax
  14: 01 d0                   add    %edx,%eax
  16: 5d                      pop    %rbp
  17: c3                      retq

0000000000000018 <main>:
  18: f3 0f 1e fa             endbr64
  1c: 55                      push   %rbp
  1d: 48 89 e5                mov    %rsp,%rbp
  20: 48 83 ec 10             sub    $0x10,%rsp
```

```
  24: c7 45 f4 03 00 00 00    movl   $0x3,-0xc(%rbp)
  2b: c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
  32: 8b 55 f8                mov    -0x8(%rbp),%edx
  35: 8b 45 f4                mov    -0xc(%rbp),%eax
  38: 89 d6                   mov    %edx,%esi
  3a: 89 c7                   mov    %eax,%edi
  3c: e8 00 00 00 00          callq  41 <main+0x29>     <-- here
  41: 89 45 fc                mov    %eax,-0x4(%rbp)
  44: b8 00 00 00 00          mov    $0x0,%eax
  49: c9                      leaveq
  4a: c3                      retq
```

2. The second and third entries are for the `.eh_frame` section. We can see that the offset of `0x20` and `0x40` represents the following lines below. They also have dummy addresses that need to be replaced. They are replaced by the address `.text`, which represents the first address in the `.text` section, i.e. the address of the `add` function, and the address `.text+0x18`, which represents the address of the `main` function.

```
Disassembly of section .eh_frame:

0000000000000000 <.eh_frame>:
   0: 14 00                   adc    $0x0,%al
   2: 00 00                   add    %al,(%rax)
   4: 00 00                   add    %al,(%rax)
   6: 00 00                   add    %al,(%rax)
   8: 01 7a 52                add    %edi,0x52(%rdx)
   b: 00 01                   add    %al,(%rcx)
   d: 78 10                   js     1f <.eh_frame+0x1f>
   f: 01 1b                   add    %ebx,(%rbx)
  11: 0c 07                   or     $0x7,%al
  13: 08 90 01 00 00 1c       or     %dl,0x1c000001(%rax)
  19: 00 00                   add    %al,(%rax)
  1b: 00 1c 00                add    %bl,(%rax,%rax,1)
  1e: 00 00                   add    %al,(%rax)
  20: 00 00                   add    %al,(%rax)     <-- here for 2nd entry
  22: 00 00                   add    %al,(%rax)
  24: 18 00                   sbb    %al,(%rax)
  26: 00 00                   add    %al,(%rax)
  28: 00 45 0e                add    %al,0xe(%rbp)
  2b: 10 86 02 43 0d 06       adc    %al,0x60d4302(%rsi)
  31: 4f 0c 07                rex.WRXB or $0x7,%al
  34: 08 00                   or     %al,(%rax)
  36: 00 00                   add    %al,(%rax)
  38: 1c 00                   sbb    $0x0,%al
  3a: 00 00                   add    %al,(%rax)
  3c: 3c 00                   cmp    $0x0,%al
  3e: 00 00                   add    %al,(%rax)
  40: 00 00                   add    %al,(%rax)     <-- here for 3rd entry
  42: 00 00                   add    %al,(%rax)
  44: 33 00                   xor    (%rax),%eax
```

Therefore, we can see that the object file generates a "skeleton" code that contains all the instructions, with some dummy addresses that need to be replaced. The relocation table T tells us exactly where these dummy addresses are in the code and what they need to be replaced with. Therefore, if we want to call a function `printf` that is in the text section at address `0x30`, then we can actually look at the value at `T[30]` to see where the actual address is. At this point, note that we still do not know the actual memory address of `add`. This is determined by the linker.

6.5.2 Linking with One Object File

Now let's see what happens once we link the object file `main.o` into the final executable `main`. If we disassemble it, then we can see a few things:

1. The addresses of all the functions have been changed. `add` starts on address `0x1129` rather than `0x0` and `main` starts on address `0x1141` rather than `0x18`.
2. The dummy address `0x0` of the call to function `add` in `main` have been replaced with the actual addresses `0x1129`.

```
0000000000001129 <add>:
  1129:  f3 0f 1e fa             endbr64
  112d:  55                      push   %rbp
  112e:  48 89 e5                mov    %rsp,%rbp
  1131:  89 7d fc                mov    %edi,-0x4(%rbp)
  1134:  89 75 f8                mov    %esi,-0x8(%rbp)
  1137:  8b 55 fc                mov    -0x4(%rbp),%edx
  113a:  8b 45 f8                mov    -0x8(%rbp),%eax
  113d:  01 d0                   add    %edx,%eax
  113f:  5d                      pop    %rbp
  1140:  c3                      retq

0000000000001141 <main>:
  1141:  f3 0f 1e fa             endbr64
  1145:  55                      push   %rbp
  1146:  48 89 e5                mov    %rsp,%rbp
  1149:  48 83 ec 10             sub    $0x10,%rsp
  114d:  c7 45 f4 03 00 00 00    movl   $0x3,-0xc(%rbp)
  1154:  c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
  115b:  8b 55 f8                mov    -0x8(%rbp),%edx
  115e:  8b 45 f4                mov    -0xc(%rbp),%eax
  1161:  89 d6                   mov    %edx,%esi
  1163:  89 c7                   mov    %eax,%edi
  1165:  e8 bf ff ff ff          callq  1129 <add>     <-- replaced with actual address
  116a:  89 45 fc                mov    %eax,-0x4(%rbp)
  116d:  b8 00 00 00 00          mov    $0x0,%eax
  1172:  c9                      leaveq
  1173:  c3                      retq
  1174:  66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  117b:  00 00 00
  117e:  66 90                   xchg   %ax,%ax
```

6.5.3 Global vs External Symbols

So far, we have talked about using the `#include` as a precompiling command that says "put all the text from this other file right here." Take the following code for instance.

```c
// file1.c
#include "sum.h"

int array[2] = {1, 2};

int main() {
  int val = sum(array, 2);
  return val;
}
```

```c
// sum.h
int sum(int *a, int n) {
  int i, s = 0;
  for (i = 0; i < n; i++) {
    s += a[i];
  }
  return s;
}
.
```

Figure 38: Including a header file in `file1.c` to import functions and variables.

However, there is another way to do this. We can use *external symbols* to access. Rather than simply copying and pasting the code into the file, the `extern` keyword marks that the variable or function exists externally to this source file and does not allocate storage for it.

```c
// main.c
extern int sum(int *array, int n);

int array[2] = {1, 2};

int main(void) {
  int val = sum(array, 2);
  return val;
}
```

```c
// sum.c
int sum(int *array, int n) {
  int i, s = 0 ;
  for (int i = 0; i < n; i++) {
      s += array[i];
    }
  return s;
}
.
```

Figure 39: Using external symbols to access functions and variables.

One is not a replacement for the other, so what advantage does this have? Well, as we will see, if we have multiple object (source) files, say `A.c`, `B.c`, and `C.c`, that need to reference the same function or variable `var` in `ext.c`, then how would we do this? If we simply put `#include "ext.h"` in all the files, then we would have multiple copies of the same code. This means that for each source there would be its own copy of `var` created and the linker would be unable to resolve this symbol. However, if we put `extern int var;` at the top of each source file, then only one copy of `var` would be created (in `ext.c`), which creates a single instance of `var` for the linker to resolve. [2]

Therefore, there are three types of symbols (variables, functions, etc.) that we need to consider:

1. **Global symbols** that are defined in the global scope of a C file.
2. **Local symbols** that are defined in the local scope of a C file, e.g. within functions, loops, etc.
3. **External symbols** that are defined in another C file referenced by the `extern` keyword.

Linkers will only know about global and external symbols, and will have no idea that any local symbols exist. With the information of these two types of symbols and the relocation tables of each object file, the linker can then resolve the addresses of all the symbols in the final binary.

The two types of symbols that the linker will know about are the global and external symbols. We can see that external symbols can be problematic if the object files don't know about each other.

[2] https://stackoverflow.com/questions/1330114/whats-the-difference-between-using-extern-and-including-header-files

Example 6.6 (Global and Local Symbols)

Consider the following code where the left file includes the right file.

```c
// main.c
#include "sum.h"

int array[2] = {1, 2};

int main() {
  int val = sum(array, 2);
  return val;
}
```

```c
// sum.h
int sum(int *a, int n) {
  int i, s = 0;
  for (i = 0; i < n; i++) {
    s += a[i];
  }
  return s;
}
.
```

In the left file,

1. We define the global symbol `main()`.
2. Inside main, `val` is a local symbol so the linker knows nothing about it.
3. The `sum` function is an external symbol, and it references a global symbol that's defined in `sum` the right file.
4. The `array` is a global symbol that is defined in the right file.

In the right file, the linker knows nothing of the local symbols `i` or `s`.

6.5.4 Linking with Multiple Object Files

We have seen the case of linking when we simply have one object file. The relocation was simple since the `.text` section is contiguous and so we needed simple translations of addresses to relocate `add` and `main`, along with whatever other sections and files. Now let's consider the case where we have multiple object files.

```c
// main.c
extern int sum(int *array, int n);

int array[2] = {1, 2};

int main(void) {
  int val = sum(array, 2);
  return val;
}
```

```c
// sum.c
int sum(int *array, int n) {
  int i, s = 0 ;
  for (int i = 0; i < n; i++) {
      s += array[i];
    }
  return s;
}
.
```

Now they have their own object files shown below, where I also put the source code lines to make it easier to parse. Note that again, in `main.o` the call to function `sum` is a dummy address that needs to be replaced. Furthermore, in both `main.o` and `sum.o`, the `.text` section is at address `0x0`, where the addresses of the function `main` and `sum` are, respectively. This causes an overload in the address space.

To demonstrate what happens, we look at how the disassembly, symbol tables, and relocation tables are updated before (with the object files) and after (in the binary) linking.

Example 6.7 (Disassembly of Object Files)

In here, note that both the `array` and `sum` are not initialized and are therefore set to dummy addresses.

```
main.o:     file format elf64-x86-64
Disassembly of section .text:

0000000000000000 <main>:
extern int sum(int *array, int n);

```

```
int array[2] = {1, 2};

int main(void) {
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 48 83 ec 10             sub    $0x10,%rsp
  int val = sum(array, 2);
   c: be 02 00 00 00          mov    $0x2,%esi
  11: 48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 18 <main+0x18>  <-- dummy
    address
  18: e8 00 00 00 00          callq  1d <main+0x1d>                          <-- dummy
    address
  1d: 89 45 fc                mov    %eax,-0x4(%rbp)
  return val;
  20: 8b 45 fc                mov    -0x4(%rbp),%eax
}
  23: c9                      leaveq
  24: c3                      retq
```

```
sum.o:     file format elf64-x86-64
Disassembly of section .text:

0000000000000000 <sum>:
int sum(int *array, int n) {
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 48 89 7d e8             mov    %rdi,-0x18(%rbp)
   c: 89 75 e4                mov    %esi,-0x1c(%rbp)
  int i, s = 0;
   f: c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
  for (int i = 0; i < n; i++) {
  16: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  1d: eb 1d                   jmp    3c <sum+0x3c>
    s += array[i];
  1f: 8b 45 fc                mov    -0x4(%rbp),%eax
  22: 48 98                   cltq
  24: 48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  2b: 00
  2c: 48 8b 45 e8             mov    -0x18(%rbp),%rax
  30: 48 01 d0                add    %rdx,%rax
  33: 8b 00                   mov    (%rax),%eax
  35: 01 45 f8                add    %eax,-0x8(%rbp)
  for (int i = 0; i < n; i++) {
  38: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
  3c: 8b 45 fc                mov    -0x4(%rbp),%eax
  3f: 3b 45 e4                cmp    -0x1c(%rbp),%eax
  42: 7c db                   jl     1f <sum+0x1f>
  }
  return s;
  44: 8b 45 f8                mov    -0x8(%rbp),%eax
}
  47: 5d                      pop    %rbp
  48: c3                      retq
```

1. In `main.o` at address `0x0`, we have the `main` function and this is because everything is stored relatively to the start of main. Once we have linked, `main` shows the absolute addresses of all the instructions.
2. In instruction 11 in `main.o` we can see that `48 8d 3d` is the `lea` instruction, which is the same as that in `main`. However, the address that is was acting on is `0x0` since the array has not been initialized yet. We can see in `main` that the address is now `0x00002ecf`.
3. The comment in `main` indicates that the final relocated address used to access the `array` is `0x4010`. To see relocated addresses in general, just look for the comments and shift them accordingly.

```
main:     file format elf64-x86-64

0000000000001129 <main>:
    1129:   f3 0f 1e fa             endbr64
    112d:   55                      push   %rbp
    112e:   48 89 e5                mov    %rsp,%rbp
    1131:   48 83 ec 10             sub    $0x10,%rsp
    1135:   be 02 00 00 00          mov    $0x2,%esi
    113a:   48 8d 3d cf 2e 00 00    lea    0x2ecf(%rip),%rdi        # 4010 <array>
    1141:   e8 08 00 00 00          callq  114e <sum>
    1146:   89 45 fc                mov    %eax,-0x4(%rbp)
    1149:   8b 45 fc                mov    -0x4(%rbp),%eax
    114c:   c9                      leaveq
    114d:   c3                      retq

000000000000114e <sum>:
    114e:   f3 0f 1e fa             endbr64
    1152:   55                      push   %rbp
    1153:   48 89 e5                mov    %rsp,%rbp
    1156:   48 89 7d e8             mov    %rdi,-0x18(%rbp)
    115a:   89 75 e4                mov    %esi,-0x1c(%rbp)
    ...
```

Example 6.8 (Symbol Tables of Object Files)

Let's take a look at the symbol table of each file as well. Again, all of the addresses of each symbol are 0s since they are using relative addressing. The `array` and `main` are global symbols since they reside in the global scope, while the `sum` function is an external and undefined symbol.

```
main.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*   0000000000000000 main.c
0000000000000000 l    d  .text   0000000000000000 .text
0000000000000000 l    d  .data   0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack  0000000000000000 .note.GNU-stack
0000000000000000 l    d  .note.gnu.property  0000000000000000 .note.gnu.property
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment   0000000000000000 .comment
0000000000000000 g     O .data   0000000000000008 array
0000000000000000 g     F .text   0000000000000025 main
0000000000000000         *UND*   0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000         *UND*   0000000000000000 sum
```

```
sum.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*   0000000000000000 sum.c
0000000000000000 l    d  .text   0000000000000000 .text
0000000000000000 l    d  .data   0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack  0000000000000000 .note.GNU-stack
0000000000000000 l    d  .note.gnu.property  0000000000000000 .note.gnu.property
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment   0000000000000000 .comment
0000000000000000 g     F .text   0000000000000049 sum
```

When we have the linked binary, note a few things.

1. In `main.o`, the numbers on the left represents the address of the symbol (all 0s since we haven't linked yet and their final addresses aren't known), while the addresses in `a.out` are all known.
2. In `main.o`, the `sum` function is an external symbol and is undefined. The linker will need to know where this is. In `main`, note that the `sum` function is now a global symbol and is defined, along with the size. We can now see that all the final addresses of each symbol is known, along with their sizes, and the `UND` marker is now gone as well.
3. Only the size of the global variable is known in `main.o` since we have defined it within the code. However, in `main`, the linker has now assigned an address to it.
4. To see the size in bytes of the array, you can look at the address and how much size it takes up.

```
main:     file format elf64-x86-64

SYMBOL TABLE:
...
0000000000004008 g     O .data     0000000000000000              .hidden __dso_handle
000000000000114e g     F .text     0000000000000049              sum
0000000000002000 g     O .rodata   0000000000000004              _IO_stdin_used
00000000000011a0 g     F .text     0000000000000065              __libc_csu_init
0000000000004020 g       .bss      0000000000000000              _end
0000000000001040 g     F .text     000000000000002f              _start
0000000000004018 g       .bss      0000000000000000              __bss_start
0000000000001129 g     F .text     0000000000000025              main
0000000000004018 g     O .data     0000000000000000              .hidden __TMC_END__
...
```

Example 6.9 (Relocation Tables)

Ignoring the `.eh_frame`, in `main.o` the relocation table contains entries for `array` and `sum` that must be relocated.

```
main.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000014 R_X86_64_PC32     array-0x0000000000000004
0000000000000019 R_X86_64_PLT32    sum-0x0000000000000004

RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
```

```
sum.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
```

We can see a couple things. Namely, there is nothing to be relocated in `a.out` since everything has been relocated already by the linker. So let's focus on the relocation for `main.o`. In here, we can see that in the `.text` section, there are two things being relocated:

1. The reference to the global variable `array` is being relocated. In this object file, we look at the offset `0x14` from the beginning of the `.text` section, which contains the instruction that needs to access `array`. This relocation record tells the linker to calculate the 32-bit offset from the instruction (at offset `0x14`) to the start of `array`, then adjust it by subtracting 4 bytes.
2. The reference to the `sum` function is being relocated. In this object file, we look at the offset `0x19` from the beginning of the `.text` section, which contains the instruction that needs to access `sum`. This relocation record tells the linker to calculate the 32-bit offset from the instruction (at offset `0x19`) to the start of the `.plt` section, then adjust it by subtracting 4 bytes.

```
main:     file format elf64-x86-64
```

6.6 Compiler Optimization

We have learned the complete process of compilers, but compilers can be a little smarter than just translating code line by line. They also come with flags that can optimize the code.

Definition 6.9 (gcc Optimization)

The gcc compiler can optimize the code with the `-O` flag. To run level 1 optimization, we can write

```
gcc -O1 -o main main.c
```

The level of optimizations are listed:

1. Level 1 perform basic optimizations to reduce code size and execution time while attempting to keep compile time to a minimum.
2. Level 2 optimizations include most of GCC's implemented optimizations that do not involve a space-performance trade-off.
3. Level 3 performs additional optimizations (such as function inlining) and may cause the program to take significantly longer to compile.

Let's see what common implementation are.

Definition 6.10 (Constant Folding)

Constants in the code are evaluated at compile time to reduce the number of resulting instructions. For example, in the code snippet that follows, macro expansion replaces the statement `int debug = N-5` with `int debug = 5-5`. Constant folding then updates this statement to `int debug = 0`.

```
#define N 5
int debug = N - 5; //constant folding changes this statement to debug = 0;
```

Definition 6.11 (Constant Propagation)

Constant propagation replaces variables with a constant value if such a value is known at compile time. Consider the following code segment, where the `if (debug)` statement is replaced with `if (0)`.

```c
int debug = 0;

int doubleSum(int *array, int length){
    int i, total = 0;
    for (i = 0; i < length; i++){
        total += array[i];
        if (debug) {
            printf("array[%d] is: %d\n", i, array[i]);
        }
    }
    return 2 * total;
}
```

Definition 6.12 (Dead Code Elimination)

Dead code elimination removes code that is never executed. For example, in the code snippet that follows, the `if (debug)` statement and its body is removed since the value of `debug` is known to be 0.

```c
int debug = 0;

int doubleSum(int *array, int length){
    int i, total = 0;
    for (i = 0; i < length; i++){
        total += array[i];
        if (debug) {                                    // remove
            printf("array[%d] is: %d\n", i, array[i]);    // remove
        }                                               // remove
    }
    return 2 * total;
}
```

Definition 6.13 (Simplifying Expressions)

Some instructions are more expensive than others, so things like

1. `2 * total` may be replaced with `total + total` because addition instruction is less expensive than multiplication.
2. `total * 8` may be replaced with `total « 3`
3. `total % 8` may be replaced with `total & 7`

Note that these optimization techniques are in no way a guarantee that the code will run faster since there are many factors and always edge cases (for example, maybe some localities are lost). Furthermore, compiler optimization will never be able to improve runtime complexity (e.g. by replacing bubble sort with quicksort).

6.7 Virtual Memory Addresses

Memory addresses are actual virtual. There exists a hashmap from a virtual address to a physical address for each process. If we used physical addresses only to begin with, another application that uses the same physical address would overwrite the data, so there needs to be some sort of isolation. Note that every process has its own mapping, that's completely different! It's not one mapping for all processes. These