Dive Into Systems Search book...

8.1. Diving into Assembly: Basics

For a first look at assembly, we modify the `adder` function from the beginning of the chapter to simplify its behavior. Here's the modified function (`adder2`):

```c
#include <stdio.h>

//adds two to an integer and returns the result
int adder2(int a) {
    return a + 2;
}

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

To compile this code, use the following command:

```
$ gcc -m32 -o modified modified.c
```

The `-m32` flag tells GCC to compile the code to a 32-bit executable. Forgetting to include this flag may result in assembly that is wildly different from the examples shown in this chapter; by default, GCC compiles to x86-64 assembly, the 64-bit variant of x86. However, virtually all 64-bit architectures have a 32-bit operating mode for backward compatibility. This chapter covers IA32; other chapters cover x86-64 and ARM. Despite its age, IA32 is still extremely useful for understanding how programs work and how to optimize code.

Next, let's view the corresponding assembly of this code by typing the following command:

```
$ objdump -d modified > output
$ less output
```

Search for the code snippet associated with `adder2` by typing `/adder2` while examining the file `output` using `less` . The section associated with `adder2` should look similar to the following:

Assembly output for the `adder2` function

```
0804840b <adder2>:
 804840b:       55                      push   %ebp
 804840c:       89 e5                   mov    %esp,%ebp
 804840e:       8b 45 08                mov    0x8(%ebp),%eax
 8048411:       83 c0 02                add    $0x2,%eax
 8048414:       5d                      pop    %ebp
 8048415:       c3                      ret
```

Don't worry if you don't understand what's going on just yet. We will cover assembly in greater detail in later sections. For now, we will study the structure of these individual instructions.

Each line in the preceding example contains an instruction's address in program memory, the bytes corresponding to the instruction, and the plaintext representation of the instruction itself. For example, `55` is the machine code representation of the instruction `push %ebp`, and the instruction occurs at address `0x804840b` in program memory.

It is important to note that a single line of C code often translates to multiple instructions in assembly. The operation `a + 2` is represented by the two instructions `mov 0x8(%ebp), %eax` and `add $0x2, %eax`.

WARNING

Your assembly may look different!

If you are compiling your code along with us, you may notice that some of your assembly examples look different from what is shown in this book. The precise assembly instructions that are output by any compiler depend on that compiler's version and the underlying operating system. Most of the assembly examples in this book were generated on systems running Ubuntu or Red Hat Enterprise Linux (RHEL).

In the examples that follow, we do not use any optimization flags. For example, we compile any example file (`example.c`) using the command `gcc -m32 -o example example.c`. Consequently, there are many seemingly redundant instructions in the examples that follow. Remember that the compiler is not "smart" — it simply follows a series of rules to translate human-readable code into machine language. During this translation process, it is not uncommon for some redundancy to occur. Optimizing compilers remove many of these redundancies during optimization, which is covered in a later chapter.

8.1.1. Registers

Recall that a **register** is a word-sized storage unit located directly on the CPU. There may be separate registers for data, instructions, and addresses. For example, the Intel CPU has a total of eight registers for storing 32-bit data:

`%eax`, `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`, `%esp`, and `%ebp`.

Programs can read from or write to all eight of these registers. The first six registers all hold general-purpose data, whereas the last two are typically reserved by the compiler to hold address data. While a program may interpret a general-purpose register's contents as integers or as addresses, the register itself makes no distinction. The last two registers (`%esp` and `%ebp`) are known as the **stack pointer** and the **frame pointer**, respectively. The compiler reserves these registers for operations that maintain the layout of the program stack. Typically, `%esp` points to the top of the program stack, whereas `%ebp` points to the base of the current stack frame. We discuss stack frames and these two registers in greater detail in our discussion on functions.

The last register worth mentioning is `%eip` or the **instruction pointer**, sometimes called the **program counter** (PC). It points to the next instruction to be executed by the CPU. Unlike the eight registers mentioned previously, programs cannot write directly to register `%eip`.

8.1.2. Advanced Register Notation

For the first six registers mentioned, the ISA provides a mechanism to access the lower 16 bits of each register. The ISA also provides a separate mechanism to access the 8-bit components of the lower 16 bits of the first four of these registers. Table 1 lists each of the six registers and the ISA mechanisms (if available) to access their component bytes.

Table 1. x86 Registers and Mechanisms for Accessing Lower Bytes.

32-bit register (bits 31-0)	Lower 16 bits (bits 15-0)	Bits 15-8	Bits 7-0
`%eax`	`%ax`	`%ah`	`%al`
`%ebx`	`%bx`	`%bh`	`%bl`
`%ecx`	`%cx`	`%ch`	`%cl`
`%edx`	`%dx`	`%dh`	`%dl`
`%edi`	`%di`		
`%esi`	`%si`		

The lower 16 bits for any of the aforementioned registers can be accessed by referencing the last two letters in the register's name. For example, use `%ax` to access the lower 16 bits of `%eax`.

The *higher* and *lower* bytes within the lower 16 bits of the first four listed registers can be accessed by taking the last two letters of the register name and replacing the last letter with either an `h` (for *higher*) or an `l` (for *lower*) depending on which byte is desired. For example, `%al` references the lower eight

bits of register `%ax`, whereas `%ah` references the higher eight bits of register `%ax`. These eight bit registers are commonly used by the compiler for storing single-byte values for certain operations, such as bitwise shifts (a 32-bit register cannot be shifted more than 32 places and the number 32 requires only a single byte of storage). In general, the compiler will use the smallest component register needed to complete an operation.

8.1.3. Instruction Structure

Each instruction consists of an operation code (or **opcode**) that specifies what it does, and one or more **operands** that tell the instruction how to do it. For example, the instruction `add $0x2, %eax` has the opcode `add` and the operands `$0x2` and `%eax`.

Each operand corresponds to a source or destination location for a specific operation. There are multiple types of operands:

- **Constant (literal)** values are preceded by the `$` sign. For example in the instruction `add $0x2, %eax`, `$0x2` is a literal value that corresponds to the hexadecimal value 0x2.
- **Register** forms refer to individual registers. The instruction `add $0x2, %eax` specifies register `%eax` as the destination location where the result %of the `add` operation will be stored.
- **Memory** forms correspond to some value inside main memory (RAM) and are commonly used for address lookups. Memory address forms can contain a combination of registers and constant values. For example, in the instruction `mov 0x8(%ebp),%eax`, the operand `0x8(%ebp)` is an example of a memory form. It loosely translates to "add 0x8 to the value in register `%ebp`, and then perform a memory lookup." If this sounds like a pointer dereference, that's because it is!

8.1.4. An Example with Operands

The best way to explain operands in detail is to present a quick example. Suppose that memory contains the following values:

Address	Value
0x804	0xCA
0x808	0xFD
0x80c	0x12
0x810	0x1E

Let's also assume that the following registers contain values:

Register	Value
%eax	0x804
%ebx	0x10
%ecx	0x4
%edx	0x1

Then the operands in Table 2 evaluate to the values shown. Each row of the table matches an operand with its form (e.g. constant, register, memory), how it is translated, and its value. Note that the notation M[x] in this context denotes the value at the memory location specified by address x.

Table 2. Example operands

Operand	Form	Translation	Value
%ecx	Register	%ecx	0x4
(%eax)	Memory	M[%eax] or M[0x804]	0xCA
$0x808	Constant	0x808	0x808
0x808	Memory	M[0x808]	0xFD
0x8(%eax)	Memory	M[%eax + 8] or M[0x80c]	0x12
(%eax, %ecx)	Memory	M[%eax + %ecx] or M[0x808]	0xFD
0x4(%eax, %ecx)	Memory	M[%eax + %ecx + 4] or M[0x80c]	0x12
0x800(,%edx,4)	Memory	M[0x800 + %edx*4] or M[0x804]	0xCA
(%eax, %edx, 8)	Memory	M[%eax + %edx*8] or M[0x80c]	0x12

In Table 2, the notation `%ecx` indicates the value stored in register `%ecx`. In contrast, M[`%eax`] indicates that the value inside `%eax` should be treated as an address, and to dereference (look up) the value at that address. Therefore, the operand `(%eax)` corresponds to M[0x804], which corresponds to the value 0xCA.

A few important notes before continuing. While Table 2 shows many valid operand forms, not all forms can be used interchangeably in all circumstances.

Specifically:

- Constant forms cannot serve as destination operands.
- Memory forms cannot serve *both* as the source and destination operand in a single instruction.
- In cases of scaling operations (see the last two operands in Table 2), the scaling factor must be one of 1, 2, 4, or 8.

Table 2 is provided as a reference; however, understanding key operand forms will help improve the reader's speed in parsing assembly language.

8.1.5. Instruction Suffixes

In several cases in upcoming examples, common and arithmetic instructions have a suffix that indicates the *size* (associated with the *type*) of the data being operated on at the code level. The compiler automatically translates code to instructions with the appropriate suffix. Table 3 shows the common suffixes for x86 instructions.

Table 3. Example Instruction Suffixes

Suffix	C Type	Size (bytes)
b	`char`	1
w	`short`	2
l	`int`, `long`, `unsigned`	4

Note that instructions involved with conditional execution have different suffixes based on the evaluated condition. We cover instructions associated with conditional instructions in a later section.

Contents