Assembly

Neil Kingdom

December 4, 2023

Contents

1 Preface

This document was very difficult for me to write because there are so many variants of assembly and so little information out on the internet. Assembly is not as commonly used nowadays, and those who do use it are using it on specific boards with specific manuals to do specific things. While this is probably how you ought to write assembly, I wanted to cover the core philosophies behind assembly and give a bigger picture of what to reasonably expect when studying it. The most useful application of this document will be for debugging purposes since it will mostly focus on modern processors and modern syntax, but the idea behind assembly remains static across all architectures, so you should hopefully be able to walk away more confident in your ability to read and write assembly code as well.

2 Introduction

Assembly language is largely considered to be the first real programming "language". Its creation is credited to Kathleen Booth around the late 1940s. Assembly uses mnemonics which are human-readable labels that correspond to instructions built-in to the actual microprocessor or CPU. These instructions are called operation codes, or op-codes for short. These are accessed with a lookup address in the instruction table (a grid of op-codes that are available). Mnemonics are expanded during the assembly phase into the addresses that correspond to the appropriate op-code. The address of the op-code, as well as which op-codes are available, is entirely incumbent upon the architecture of the processor. This means that most processors do not share the same instruction set, and therefore, are not cross compatible. CPU manufacturers try to create families of processors that share the same instruction set, or are at least backwards compatible with older instruction sets. Instruction sets that have spanned multiple generations of processors are known as Instruction Set Architectures (ISA). ISAs were designed to make things easier, so we wouldn't need to create a new instruction set for every new CPU or processor. Unfortunately, things are still confusing, as we have multiple ISAs to worry about still. For example, common ISAs include:

- **x86 / IA-32 / i386:** Intel 32-bit (based off of the 8086 series of processor)
- **x86_64 / x64 / AMD64 / EM64T / IA-32e:** Intel 64-bit (backwards compatible with x86)
- **MIPS:** Microprocessor without Interlocked Pipelined Stages (RISC-based)
- **PowerPC (PPC):** Performance Optimization With Enhanced RISC (IBM's ISA)
- **ARM:** Advanced RISC Machines. Arm is a company that develops architectures and licenses them to other companies. Many phone processors are based off of arm.
- **AARCH64:** ARM 64-bit
- **RISC V:** Free and open source ISA created by the University of Berkeley

3 RISC and CISC

I covered RISC and CISC in my computer hardware docs, but it's more relevant here so I don't mind repeating myself. In the old days, processor manufacturers wanted to compete to pack as many instructions into their processors as possible. This was seen as having the competitive edge. This type of design was known as Complex Instruction Set Computing (CISC). Over time, electrical engineers discovered that CISC introduced many problems, such as making the design too complex to keep up with, and actually hindering efficiency in most cases. Rethinking CISC, we sort of went the reverse way, and began designing processors that had as few instructions as possible. The idea was that fewer instructions that were versatile and reusable was better than many instructions with little purpose. The Reduced Instruction Set Computing (RISC) design was introduced, and is what most processors aim to achieve nowadays. Note that RISC is a design principle, not an ISA. This may be confusing to some, as RISC-V (pronounced RISC-Five), which was designed by the

University of Berkeley as a royaltee-free ISA, was designed in accordance with the RISC design principle, however the two are not interchangeable terms.

4 Assembly Flavors

As I've stated, assembly is its own language (like C, or Java, or PHP, etc.). However, unlike those languages, assembly language can come in different ”flavors” for the same target ISA. The most common example is on x86 architectures there are 2 competing styles/flavors of assembly: Intel, and AT&T. It should be noted that Intel largely dominates in almost all areas, except for Unix-based systems, where AT&T is most commonly used (since Unix was invented at AT&T Bell Labs).

5 Assemblers

If you're familiar with a compiler, then you should have a pretty easy time understanding what an assembler is. An assembler literally just takes assembly code, and translates it to machine code. In fact, the assembler is one intermediary step in the compilation pipeline. Unlike C, where we only really have gcc, LLVM/Clang, or perhaps MSVC, there are many assemblers; namely: MASM (old), NASM, FASM, TASM, YASM, and GAS. All of those, with the exception of GAS use Intel syntax (GAS originally using AT&T syntax, and then moving to support both).

6 Endianness

I'm sure that you're aware of what little endian and big endian are, at least conceptually, but I'd like to review with ISAs use which endiannesses. As an important reminder – endianness does not have anything to do with bit order. Hex 9F will have the exact same binary layout (1001 1111) in both big and little endian format. The endianness determines the direction that the computer reads each byte. Think of words in a book; we typically read a sentence from left to right. Now imaging reading the sentence backwards. We aren't reversing the letters of each word in the sentence, we are just reading the words right to left. The same is true for bytes on a computer. Big endian essentially reads left to right, and little endian reads right to left. Big endian is the dominant order in network protocols, and is referred to as network order, often requiring additional code on both the client and server to swap the byte order before sending and after receiving the data. Most PC's, on the other hand, use little endian, which is backwards to how we are accustomed to reading.

7 Registers

Registers can be confusing if you're not familiar with low-level programming, but I assure you, they are not complex. When we use the term “register”, what we are in fact usually referring to are the General Purpose Registers (GPRs). What a register really is, is just a very small amount of storage where we can store temporary values. Think of it as extremely fast-access RAM, but it can only contain up to 64 bits (on a 64-bit architecture, that is). Registers don't necessarily need to be located in the CPU either. Other peripherals and micro controllers may have their own registers which are used for R/W operations. Sometimes (depending on the design of the interface), reading or writing to/from a register might cause something to occur in the hardware. For example, writing to a register may set the interrupt enable pin high. This is why it is important to be familiar with the documentation of whatever device you're interfacing with.

8 Intel x86 Registers

If you are using a modern Intel or AMD processor, then the chances are you will have very similar registers to another person using an Intel or AMD processor. We will be focusing on Intel x86 registers for the duration of the document to keep things simple. I will go over each type of register in the following sections.

9 General Purpose Registers

The most important registers that you will be dealing with as a programmer are the GPRs. Despite the name "general purpose", each GPR has a specific use-case, however, these are not heavily enforced, meaning that technically we are allowed to use each for whatever we want. That being said, here is a diagram of each GPR:

The general purpose registers consist of EAX, EBX, ECX, EDX, ESI, and EDI. The 'E' stands for extended i.e., extended from the 16-bit Intel architecture. You'll note that all registers beginning with an 'E' are 32-bits long, however EAX, EBX, ECX, and EDX are unique in that they have a lower 16-bit region which contains two 8-bit registers. The 'H' stands for "Higher 8 bits" and the 'L' stands for "Lower 8 bits". Therefore, AH is the higher 8 bits of the 16-bit segment of EAX, and AL is the lower 8 bits of the 16-bit segment of EAX. Things get even more confusing with x86_64 where there are 'R' registers eg. RAX, RBX, RCX, RDX, RSI, RDI, RSP, and RBP. These are 64 bits long and extend the 'E' registers, in the same manner that the 'E' registers extend the lower 16-bit registers.

Let's cover what each of these registers actually does. The 'A' register is the "accumulator". This GPR is most commonly used for arithmetic. The 'B' register is the "base" GPR. This usually stores the address of some data. The 'C' register is the counter. This is used for, well, counting e.g., incrementing i in a for loop. The 'D' register is the data register. It stores variable values or numeric constants. The SI register is the source index register and marks the beginning of a string or array. The DI register is the destination index register and marks a location for which the SI register contents should get copied to. The SP register is the stack pointer register and keeps track of the top address of the stack. BP is the base pointer register and keeps track of the base address of the stack.

10 Pointer Registers

There are 3 pointer registers – each one and the same, but varying in length. RIP, EIP, and IP are the 64-bit, 32-bit, and 16-bit registers for the instruction pointer register, respectively. The instruction pointer points to the instruction that needs to be executed next. The IP register always contains the address of the next instruction that will get executed, not the instruction which is currently being executed.

11 Segment Registers

The segment registers wont make too much sense unless we discuss how memory is segmented. It is the responsibility of the assembler to segment memory, but there are specific convensions as to how memory is segmented. Each segment contains a specific kind of data. For example, the .data segment contains global and static variables, the .text segment contains our actual code, .rodata contains read-only data and the .bss (block starting symbol) segment contains local variables.

12 Executable Linker Format (ELF)

The Executable Linker Format (ELF, formerly known as the Extensible Linker Format), is a common file format for executable files, object code, shared libraries, and core dumps. It is the standard binary file format for Unix and Unix-like systems, which is why it arises frequently when debugging code. ELF files are made up of a header, known as the ELF header, followed by the file's data. The data might include the program header table, which defines zero or more memory segments; a section header table, describing zero or more logical sections; and finally, the actual data, referred to by entries in the program header or section header table. An ELF header is either 52 or 64 bytes long depending upon whether the binary targets a 32-bit or 64-bit architecture, respectively. In either case, the first 4 bytes are always 0x7F 0x45 0x4c 0x46. 0X45 0x4c 0x46 are the letters E, L, F in ASCII, spelling out ELF. These 4 bytes are known as the magic number. I recommend checking out the Wikipedia page for ELF to see some of the other bit fields and their purposes. For instance, the byte at offset 0x07 specifies the target OS's Application Binary Interface (ABI), which can be one of the following: System V, HP-UX, NetBSD, Linux, GNU Hurd, Solaris, AIX, IRIX, FreeBSD, Tru64, Novell Modesto, OpenBSD, OpenVMS, NonStop Kernel, AROS, FenixOS, Nuxi CloudABI, or Stratus Technologies OpenVOS. We will discuss ABI soon enough. What I described was only the header for the actual ELF file. As mentioned, the data may consist of a program header table which contains entries for zero or more memory segments. Each entry specifies the underlying memory layout for its corresponding section. For example, the section's offset into the file image, the virtual address of the segment, the size in bytes of the segment, etc. The section header table, on the other hand, contains entries for zero or more logical sections. This contains fields that specify things such as the attributes of the section (writable/executable/-contains null-terminated strings, etc.), the type of contents belonging to the section (program data, symbol table, string table, dynamic linking information, etc.), the byte alignment of the section, etc.

All of the following file extensions indicate that you are dealing with an ELF file: .axf, .bin, .elf, .o, .out, .prx, .puff, .ko, .mod, .so.

13 GDTR, LDTR, IDTR Registers

14 Writing a Basic Program in 64-bit Assembly

15 x86 Ports vs ARM IO

16 Micro-Ops

17 Assembly Extensions

18 Single Instruction, Mutliple Destination (SIMD)