Virtual Machines

Jochem Arends

Introduction

Virtual Machine (VM) are programs that emulates computer hardware. Virtual machines make it possible to run programs that aren't written for your device. In this document I'm going to explain how I wrote a virtual machine using C++.

Little Computer 3

Little Computer 3 (LC-3) is an educational computer architecuture developed at the University of Texas at Austin. The LC-3 has a simple yet versatile instruction set. Many programs for this architecture can be found online, which makes the VM easy to test. On the internet, some tools for this architecture can be found, including an assembler and a C compiler.

Registers

Registers are a type of computer memory that typically is build into the CPU. This type of memory is usually faster to access for a CPU than RAM. The LC-3 has eight 16-bit general purpose registers, these are called `R0-R7`. These registers can be used as operands for instructions. Besides these general purpose registers, there's also a 16-bit program counter which is ofter referred to as `PC` and three conditional codes (which I will cover later).

Instruction Encoding

The four most significant bits of an intruction form the opcode. The LC-3 architecture knows 15 distinct operations, which are listed below in the form of an `enum class`, but can also be found in the ISA at page number 525.

opcodes.h

```
#ifndef OPCODES_H
#define OPCODES_H

namespace lc3 {
    enum class opcode {
        ADD  = 0b0001,
        AND  = 0b0101,
        BR   = 0b0000,
        JMP  = 0b1100,
        JSR  = 0b0100,
        LD   = 0b0010,
        LDI  = 0b1010,
        LDR  = 0b0110,
        LEA  = 0b1110,
        NOT  = 0b1001,
        RTI  = 0b1000,
        ST   = 0b0011,
        STI  = 0b1011,
        STR  = 0b0111,
        TRAP = 0b1111,
    };
}

#endif
```

Instruction Decoding

Instructions consist of named bit sequences that occupy fixed sequence of bits. When looking at the instruction encodings (ISA page number 525) I found that many of them follow silimar patterns. For example, instruction uses a desination register, it will always encode that register at bits 11 till 9. When looking at any other these types, the same pattern can be found. All the types are listed below together with the bits they occupy in an encoding.

Term	Bits
Destination Register (`DR`)	11..9
Source Register (`SR`)	11..9
Source Register 1 (`SR1`)	8..6
Source Register 2 (`SR2`)	2..0
Base Register (`BaseR`)	8..6
5-bit Immediate Value (`imm5`)	4..0
6-bit Offset Value (`offset6`)	5..0
9-bit PC Offset Value (`PCoffset9`)	8..0
11-bit PC Offset Value (`PCoffset11`)	10..0
8-bit Trap Vector Value (`trapvect8`)	7..0

A nice thing of this pattern is that These types can be as a sequence of bits. They can be represented using a width and an index (where `0` refers the the LSB).

types.h

```
#ifndef TYPES_H
#define TYPES_H

#include <cstdint>

namespace lc3 {
    using word = std::uint16_t;
    using sword = std::int16_t;

    template<typename T>
    concept type = requires {
        T::width;
        T::index;
    };

    struct DR {
        static constexpr word width{3};
        static constexpr word index{9};
    };

    struct SR {
        static constexpr word width{3};
        static constexpr word index{9};
    };

    /* etc.. */
}

#endif
```

encoding.h

```
#ifndef ENCODING_H
#define ENCODING_H

#include <lc3/types.h>
#include <array>
#include <concepts>

namespace lc3 {
    template<std::size_t N>
    word bit_at(std::integral auto bin) {
        return (bin >> N) & 1;
    }

    template<type T>
    word extract(word bin) {
        auto a = static_cast<word>(bin >> T::index);
        return zero_extend<T>(a);
    }

    template<type... Ts>
    auto decode(word bin) {
        return std::array<word, sizeof... (Ts)>{(extract<Ts>(bin))...};
    }
}

#endif
```

Memory

The LC-3 architecture uses 16-bit wide addresses, each of these addresses refers to a 16-bit location in memory. As for now, a simple `std::array` can be used for memory. I made it a type alias for allowing us to easily change its defenition in the future. This can come in handy when we want to implement memory mapped IO.

memory.h

```
#ifndef MEMORY_H
#define MEMORY_H

#include <lc3/types.h>
#include <array>
#include <limits>

namespace lc3 {
    using memory = std::array<sword, 0x10000>;
}

#endif
```

The CPU

Below I've pasted the declaration of a structure that represents a LC-3 CPU. It's quite a lot, but I will explain every member.

cpu.h

```
#ifndef CPU_H
#define CPU_H

#include <lc3/memory.h>
#include <lc3/opcodes.h>
#include <lc3/types.h>
#include <algorithm>
#include <ranges>

namespace lc3 {
    class cpu {
    public:
        void load(const std::ranges::input_range auto& bin, word offset = 0x0000) {
            std::ranges::copy(bin, std::next(m_memory.begin(), offset));
        }

        void execute(word bin);

        void run();
    private:
        template<opcode Opcode>
        void perform(word bin);

        void setcc(sword value);

        word m_pc{0x3000};

        sword m_regs[8]{};

        memory m_memory{};

        bool m_halted{};

        struct {
            unsigned int n : 1;
            unsigned int z : 1;
            unsigned int p : 1;
        } m_condition{0, 1, 0};
    };
}

#endif
```