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 is great for testing the VM. Besides programs, on the internet, 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 similar patterns. For example, instruction that use a desination register, it always encode that register index at bits 11 till 9. When looking at any other types, the same pattern can be found. All the types are listed below together with the bits they occupy in encodings.

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 every type can be represented using a width and an index (where `0` refers the the LSB). Below can be seen how these types can be represented in C++. The `lc3::word` and `lc3::sword` aliases defined in this header but will only be used later by other parts of the program.

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
```

Now that we've defined which bits each type occupies, we can make functions to extract the bits of these types. `lc3::extract` extracts a single type from 16-bit data. `lc3::decode` extracts a variable amount of type from 16-bit data. `lc3::bit_at` extracts a single bit from an integer.

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 compared to the other code listings, but I will explain everything in detail.

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
```

The Public Interface

The public interface of the `lc3::cpu` class consists of three member functions: `lc3::cpu::load`, `lc3::cpu::execute`, and `lc3::cpu::run`. The `lc3::cpu::load` function is for loading programs into memory. It accepts any sequence of 16-bit data that conforms to the `std::ranges::input_range` concept. By default the program gets loaded at address `0x0000` but an optional parameter can be specified in to load it at a different offset. The `lc3::cpu::execute` function accepts a single machine code instruction and executes it. The `lc3::cpu::run` function keeps executing machine code instructions starting at the current program counter and keeps doing this till the CPU gets halted (which will later be explained how to).

The Implementation Details

On second thought it does not make much sense to keep the memory inside the CPU since they're different units. It would make more sense if they both were part of a class that represents the machine as a whole (alternatively rename `lc3::cpu` to `lc3::machine`). Once I've noticed this, I was already working on this document and did not bother to change it anymore. However, it is something I would take in consideration if writing a virtual machine ever again.

Operations

In this section I'll explain how I've implemented all 15 operations. I will do by

Again, for a more detailed description, consult the ISA.

Addition

```
if (bit[5] == 0)
    DR=SR1+SR2;
else
    DR=SR1+SEXT(imm5);
setcc();
```

The first operation that I've implemented is addition. This operation has two variants, one where both source operands are registers, and another one where the first source operand is a register while the second is a 5-bit signed integer value. The 5^{th} bit of the encoded instruction indicates which of these variants should be executed. If this bit is cleared, addition should be performed between two registers; otherwise, addition between a register and an immediate value must be performed.

Performing Addition

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::ADD>(std::uint16_t bin) {
        if (bit_at<5>(bin) == 0) {
            auto [a, b, c] = decode<DR, SR1, SR2>(bin);
            m_regs[a] = m_regs[b] + m_regs[c];
            setcc(m_regs[a]);
        }
        else {
            auto [a, b, c] = decode<DR, SR1, imm5>(bin);
            m_regs[a] = m_regs[b] + sign_extend<imm5>(c);
            setcc(m_regs[a]);
        }
    }
}
```

Bitwise AND

The bitwise AND operation is almost identical to addition operation except for that instead of the `+` operator, the `&` operator should be used. For this reason I won't show the listing of the bitwise AND operation.

Conditional Branch

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::BR>(std::uint16_t bin) {
        auto [offset] = decode<PCoffset9>(bin);

        if (bit_at<11>(bin) && m_condition.n) {
            m_pc += sign_extend<PCoffset9>(offset);
        }

        if (bit_at<10>(bin) && m_condition.z) {
            m_pc += sign_extend<PCoffset9>(offset);
        }

        if (bit_at<9>(bin) && m_condition.p) {
            m_pc += sign_extend<PCoffset9>(offset);
        }
    }
}
```