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 virtual machine. Besides programs, on the internet, tools for this architecture can be found, including an assembler and a C compiler.

In this document I won't explain the syntax of the LC-3 assembly language or how to write programs for this architecture. Rather I will explain how I've implemented the virtual machine, how the instruction are encoded, and what each instruction does.

General Purpose 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`. The general purpose registers are used as operands for instructions or to store results.

Program Counter

Besides these general purpose registers, there's also a 16-bit program counter which is ofter referred to as `PC`. The program counter is basically a pointer that refers to the next instruction to be executed. When executing instructions the program the program counter automatically gets incremented but there are also instructions that explicitly set the program counter.

Conditional Codes

The LC-3 contains three conditional codes which are set by some instruction. The three conditional codes are `N` (negative), `Z` (zero), and `P` (positive). Most instructions that modify the contents of a register set these codes according to the final value of that register. If the two's complement value of that register is less than zero, the `N` flag will get set. If the value is equal to zero, the `Z` flag gets set. Finally, if the final value is greater than zero, the `P` flag gets set.

Instruction Encoding

The four most significant bits of an instruction 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

Term	Bits
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 order 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.

`lc3::cpu::perform` member function takes an opcode as template parameter and an encoded instruction as regular parameter. For each opcode, I've defined a template specialization.

`lc3::cpu::setcc` is a small member function used to set the conditional codes according to the state of its passed value.

```
namespace lc3 {
    void cpu::setcc(std::int16_t value) {
        m_condition.n = (value < 0);
        m_condition.z = (value == 0);
        m_condition.p = (value > 0);
    }
}
```

Whats left are the private data members. There is a program counter (`m_pc`), eight general purpose registers (`m_regs`), a boolean value that indicates whether the CPU is halted (`m_halted`), and the conditional codes (`m_condition`).

Operations

In this section I will explain how I've implemented 15 operations. I'm going to roughly explain what each operation does and how I've implemented it. Again, for a more detailed description, consult the ISA. The ISA contain pseudo code for the logic behind each instruction which was really helpful to me.

Addition

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]);
        }
    }
}
```

Bit-wise Logical AND

The bit-wise 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

This is the first operation that modifies control flow. It modifies the program counter explicitly based on the state of the conditional codes. The encoding consists of an offset and three bits which indicate what condition to check for. If the 11^{th}, 10^{th}, or 9^{th} bits are set, the offset will be added to the program counter if also the conditional code `n` (negative), `z` (zero), or `p` (positive) are set respecitvely.

```
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);
        }
    }
}
```

Jump

This unlike BR, this operation unconditionally sets the program counter to the value of a source register.

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::JMP>(word bin) {
        auto [idx] = decode<BaseR>(bin);
        m_pc = m_regs[idx];
    }
}
```

Jump to Subroutine

This operation is similar to JMP but works a bit differently. First, the current program counter gets copied into R7. Then if the 11^{th} bit of the encoded instruction is set, a jump operation will be performed, else an offset will be added to the program counter. The reason the program counter gets copied into R7 is so that after the procedure is done, it can return to the address it got called from.

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::JSR>(std::uint16_t bin) {
        m_regs[7] = m_pc;

        if (bit_at<11>(bin) == 0) {
            auto [idx] = decode<BaseR>(bin);
            m_pc = m_regs[idx];
        }
        else {
            auto [offset] = decode<PCoffset11>(bin);
            m_pc += sign_extend<PCoffset11>(offset);
        }
    }
}
```

Load

Operations such as `ADD` and `AND` only take registers or immediate values as source operands. Sometimes, however, we need to perform these operations on values stored in memory. The `LC-3` provides various operations for loading values from memory into registers. Once the value is into a register we can use it for the desired operation. `LD` is the first of these operation I'm going to cover. The encoding consists of a source register and a 9-bit offset that is relative to the program counter. The value stored at this offset in memory is what gets copied into the destination register.

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::LD>(std::uint16_t bin) {
        auto [idx, offset] = decode<DR, PCoffset9>(bin);
        m_regs[idx] = m_memory[m_pc + sign_extend<PCoffset9>(offset)];
        setcc(m_regs[idx]);
    }
}
```

Load Indirect

This operation (`LDI`) is similar to the `LD` operation. Its encoding also contains a `pcOffset9` but instead of just loading the value at this offset, it interprets the value at this offset as another address, the value at this address will get loaded into the desination register.

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::LDI>(std::uint16_t bin) {
        auto [idx, offset] = decode<DR, PCoffset9>(bin);
        m_regs[idx] = m_memory[m_memory[m_pc + sign_extend<PCoffset9>(offset)]];
        setcc(m_regs[idx]);
    }
}
```

Load Base+offset

Just like `LDI`, this operation is also similar to `LD`. The `LD` operation has a `pcOffset9` as operand, which is an offset relative to the program counter. With this operation however, a base register can be specified. Its second operand is a `offset6` which is the offset relative to the base register.

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::LDR>(std::uint16_t bin) {
        auto [a, b, offset] = decode<DR, BaseR, offset6>(bin);
        m_regs[a] = m_memory[m_regs[b] + sign_extend<offset6>(offset)];
        setcc(m_regs[a]);
    }
}
```

Load Effective Address

This operation is again similar to `LD` but doesn't dereference the result. It just stores the address you get when adding the program counter to the offset into the destination register.

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::LEA>(std::uint16_t bin) {
        auto [idx, offset] = decode<DR, PCoffset9>(bin);
        m_regs[idx] = m_pc + sign_extend<PCoffset9>(offset);
        setcc(m_regs[idx]);
    }
}
```

Bit-Wise Complement

This operation simply negates all bits of a general purpose register.

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::NOT>(std::uint16_t bin) {
        auto [a, b] = decode<DR, SR1>(bin);
        m_regs[a] = ~m_regs[b];
        setcc(m_regs[a]);
    }
}
```

Store

This operation copies the contents of a register into a memory location.

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::ST>(std::uint16_t bin) {
        auto [reg, offset] = decode<SR, PCoffset9>(bin);
        auto address = m_pc + sign_extend<PCoffset9>(offset);
        m_memory[address] = m_regs[reg];
    }
}
```

Store Indirect

This operation is similar to `LDI` but instead of loading the value an address points to, it copies the contents of a register to the memory an address points to.

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::STI>(std::uint16_t bin) {
        auto [reg, offset] = decode<SR, PCoffset9>(bin);
        auto address = m_memory[m_pc + sign_extend<PCoffset9>(offset)];
        m_memory[address] = m_regs[reg];
    }
}
```

Store Base+offset

The `ST` operation has a `pcOffset9` as operand, which is an offset relative to the program counter. With this operation however, a base register can be specified. Its second operand is a `offset6` which is the offset relative to the base register.

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::STR>(std::uint16_t bin) {
        auto [a, b, offset] = decode<SR, BaseR, offset6>(bin);
        m_memory[m_regs[b] + sign_extend<offset6>(offset)] = m_regs[a];
    }
}
```

System Call

System calls provide a way for user programs to interact with the operating system. The `TRAP` operation (system call) can be used to call Trap Service Routines. Normally these routines are located somewhere in memory but I've decided to hardcode them. The encoded instruction contains a trap vector (`trapvect8`) that indicates what type of system call needs to be performed.

```
namespace lc3 {
    template<>
    void cpu::perform<opcode::TRAP>(std::uint16_t bin) {
        enum class vectors {
            GETC  = 0x20,
            OUT   = 0x21,
            PUTS  = 0x22,
            IN    = 0x23,
            PUTSP = 0x24,
            HALT  = 0x25,
        };

        using enum vectors;

        auto [trapvect] = decode<trapvect8>(bin);
        auto vector = static_cast<vectors>(trapvect);

        // character input
        if (vector == GETC || vector == IN) {
            if (vector == IN) {
                std::cout << "Enter a character.\n";
            }
            m_regs[0] = std::cin.get();
        }

        // character output
        if (vector == OUT) {
            std::cout << static_cast<char>(m_regs[0]);
        }

        // string output
        if (vector == PUTS) {
            for (auto address = m_regs[0]; m_memory[address] != 0; ++address) {
                auto ch = static_cast<char>(m_memory[address]);
                std::cout << ch;
            }
        }

        // halting the CPU
        if (vector == HALT) {
            m_halted = true;
        }
    }
}
```

Testing the VM

From lc3tutor.org I've taken an example program, assembled it, and downloaded it as object file. The program reverses the contents of a hard coded string and prints it to the terminal.

The structure of the object file is very simple. The first word contains the address where the program should be loaded, the rest is just the assembled program. When reading the words, the order of the bytes should get swapped, because the object file uses little-endian.

```
#include <lc3/cpu.h>
#include <bit>
#include <fstream>
#include <vector>

int main() {
    lc3::cpu cpu{};

    std::ifstream ifs{"../reverse-string.o", std::ios::binary};

    auto read = [&ifs]() {
        lc3::word word{};
        ifs.read(reinterpret_cast<char*>(&word), sizeof word);
        return std::byteswap(word);
    };

    lc3::word origin = read();

    std::vector<lc3::word> program{};
    while (ifs) {
        program.push_back(read());
    }

    cpu.load(program, origin);
    cpu.run();
}
```

When executing this program the output is `fedcba` since the hard coded string was `abcdef`. lc3tutor.org offers some more example programs or you could write your own ones.