Project Report

8-Bit Fixed-Length Instruction Set

Dhruv Agrawal (22114030)
Mudit Mangal (22114057)

10th November, 2024

Introduction

This project explores the design and implementation of an 8-bit fixed-length instruction set architecture (ISA) for a simulated limited-functionality machine. Such architectures are crucial for devices with constrained resources, as they allow for efficient instruction processing with minimal memory and power usage. By implementing a simplified ISA, this project demonstrates how a compact, highly controlled instruction set can manage basic computation and data manipulation tasks on a small scale.

The primary aim of this project is to define a minimal yet functional set of instructions, construct a parser to interpret assembly language, and build an instruction decoder to simulate machine operations. The resulting machine can perform basic arithmetic, logical, and control operations, providing a foundation for future work in embedded systems and computer architecture. Through this project, key insights into instruction encoding, control flow, and data management are gained, offering a practical approach to ISA design and its role in computer architecture.

Project Overview

This project involves creating a functional, 8-bit fixed-length ISA for a virtual machine. An instruction set is the interface between software and hardware, providing the machine language that enables the CPU to interpret and execute commands. For this project, each instruction is encoded in 8 bits, with a limited set of opcodes representing fundamental operations. This design choice allows for a simplified control unit, fitting the intended application of this machine as a prototype for embedded or low-power devices.

The machine's architecture includes:

- Four general-purpose registers, each capable of storing an 8-bit value.
- A program counter (PC) that tracks the address of the next instruction.
- A zero flag used for conditional branching based on specific outcomes of operations.
- A fixed memory space for program and data storage, simulating a constrained environment typical of embedded systems.

To interpret and execute instructions, the project features an **Assembly Parser** to convert human-readable assembly code into machine code, and an **Instruction Decoder** that executes these machine instructions. Each component works in tandem to simulate the flow of operations in a basic CPU, ensuring that the machine can load, decode, and execute instructions correctly.

Instruction Set Architecture (ISA) Design

The ISA design defines the set of instructions the machine can understand and execute. For this project, the ISA is simplified to meet the constraints of an 8-bit architecture while providing core functionality. The instruction set is designed to include:

3.1 Instruction Format

Each instruction is fixed at 8 bits, divided into fields for the opcode and operands:

- **Opcode (4 bits)**: The first four bits specify the operation (e.g., ADD, SUB, MOV).
- **Destination (2 bits)**: Specifies the destination register.
- **Source (2 bits)**: Specifies the source register or immediate value.

The fixed-length format simplifies decoding, as each instruction is uniformly 8 bits, reducing the complexity in fetching and interpreting instructions.

3.2 Instruction Categories

The ISA consists of three main categories of instructions:

1. Data Transfer Instructions

- **MOV**: Moves data between registers or from an immediate value to a register.
- **LOAD**: Loads data from memory into a register.
- **STORE**: Stores data from a register into memory.

2. Arithmetic and Logical Instructions

- **ADD**: Adds the values of two registers and stores the result in the destination register.
- **SUB**: Subtracts the source register's value from the destination register's value.
- **AND**: Performs a bitwise AND operation between two registers.
- **OR**: Performs a bitwise OR operation.

3. Control Flow Instructions

- **JMP**: Unconditional jump to a specific memory address.
- **JZ**: Jumps to a specified memory address if the zero flag is set, allowing for conditional branching.

- **NOP**: No operation, allowing for delays or alignment.

3.3 Addressing Modes

The instruction set supports the following addressing modes:

- **Register Addressing**: Operands are specified by registers directly.
- **Immediate Addressing**: Operands include a direct constant value.
- **Memory Addressing**: For LOAD and STORE operations, data is accessed from specified memory locations.

3.4 Opcode Table

The following table outlines the opcodes and their corresponding operations:

Instruction	Opcode	Description
ADD	0x0	Addition of two registers
SUB	0x1	Subtraction of one register from another
MOV	0x2	Move data from one register to another
LOAD	0x3	Load data from memory to a register
STORE	0x4	Store data from a register to memory
LOADI	0x5	Load an immediate value into a register
JMP	0x6	Unconditional jump
JZ	0x7	Jump if the zero flag is set
INC	0x8	Increment the value in a register
DEC	0x9	Decrement the value in a register
AND	0xA	Logical AND between two registers
OR	0xB	Logical OR between two registers
NOP	0xC	No operation

Design and Implementation of Components

In this project, three core components—`AssemblyParser`, `InstructionDecoder`, and `Machine`—form the core of an 8-bit limited functionality CPU emulator. Below is a breakdown of each component, how it was implemented, and its function.

1. **AssemblyParser**:
    a. **Purpose**: This component parses assembly language instructions into 8-bit machine code, mapping mnemonics to opcodes and operands to encoded values.
    b. **Implementation**:
        i. The parser converts assembly instructions to binary code based on predefined opcodes.
        ii. It uses an `opcodeMap` to associate mnemonics (e.g., `ADD`, `SUB`) with 4-bit binary opcodes (e.g., `0x0` for `ADD`).
        iii. Each instruction is assembled into an 8-bit value, with the 4 most significant bits for the opcode and the remaining bits encoding operand data.
    c. **Key Functionality**: `AssemblyParser::parse` reads the assembly file, processes each instruction line, and outputs machine code bytes for execution.
2. **InstructionDecoder**:
    a. **Purpose**: Decodes each 8-bit machine code instruction and executes the corresponding operation on the `Machine` instance.
    b. **Implementation**:
        i. Extracts the opcode, destination, and source operands from each instruction using bitwise operations.
        ii. Implements a `decodeAndExecute` function to execute different instructions based on the opcode, including arithmetic (`ADD`, `SUB`), data movement (`MOV`, `LOAD`, `STORE`), and control flow operations (`JMP`, `JZ`).
    c. **Key Functionality**: A `switch` statement in `InstructionDecoder::decodeAndExecute` uses opcode values to direct control to specific operations (e.g., increment/decrement registers, update memory).
3. **Machine**:
    a. **Purpose**: Emulates the hardware, providing registers, a program counter, flags, and memory for the CPU.
    b. **Implementation**:

    i. Contains four 8-bit general-purpose registers, an 8-bit program counter, and a `Z` (Zero) flag.
    ii. `Machine::reset` initializes or clears the CPU state before each run.
c. **Key Functionality**: The `Machine` class holds the state modified by the `InstructionDecoder` during execution, updating registers, flags, memory, and program counter.

Data Flow and Control Flow

Data and control flow are crucial for understanding how instructions are processed and executed within the 8-bit machine. Each instruction cycle consists of three main phases: fetching, decoding, and executing. In these phases, data and control signals move between components according to a predetermined sequence to achieve the desired operations.

5.1 Data Flow

Data flow refers to the movement of data between registers, memory, and the ALU during instruction execution. The 8-bit machine follows a structured data flow for each instruction type, ensuring that data reaches its intended destination through efficient paths.

- **Data Transfer Instructions (e.g., MOV, LOADI, STORE)**:
    - **MOV**: Transfers data between two registers or from an immediate value to a register. Data flows directly from the source register or immediate value to the destination register.
    - **LOAD/STORE**: For LOAD, data moves from a specified memory address to a register, and for STORE, data in a register is written to a memory address.
- **Arithmetic and Logical Instructions (e.g., ADD, SUB, AND, OR)**:
    - The source registers provide operands to the ALU, which performs the calculation and then outputs the result to the destination register.
    - If the operation results in zero, the Zero Flag (ZF) is set, allowing for conditional instructions that may depend on this outcome.
- **Control Flow Instructions (e.g., JMP, JZ)**:
    - **JMP**: An unconditional jump to a new address, modifying the program counter directly.
    - **JZ**: A conditional jump that changes the PC only if the Zero Flag is set, allowing for branching based on previous operations' results.

5.2 Control Flow

Control flow manages the sequence of operations based on the program counter (PC) and instruction decoder outputs. It includes handling both sequential instructions and jumps, ensuring that each instruction in the program is executed in the correct order.

- **Fetch Phase**:
    - The program counter (PC) sends the address of the next instruction to memory.

    - The instruction at this address is fetched and stored temporarily for decoding.
- **Decode Phase**:
    - The instruction decoder reads the opcode and operands from the fetched instruction.
    - The decoder activates the necessary control signals to direct data flow for the specified operation, such as reading data from registers, sending it to the ALU, or loading a memory address.
- **Execute Phase**:
    - Based on the decoded instruction, the control unit directs data flow to the ALU, memory, or registers as needed.
    - After execution, the program counter either increments to the next sequential instruction or updates to a new address if a jump operation is encountered.

The control unit and program counter manage the instruction cycle, ensuring that each phase (fetch, decode, execute) occurs in the proper order. This structured control flow allows the machine to execute complex sequences of operations using a limited instruction set.

Code Explanation

Here, we'll examine some of the essential code sections that implement the core components, including the assembler, instruction decoder, and execution logic.

Opcode Mapping and Parsing

In `AssemblyParser`, the following snippet defines opcodes for instructions:

```
AssemblyParser::AssemblyParser()
{
    // Updated opcode map to align with the decoder's switch cases
    opcodeMap = {
        {"ADD", 0x0},   // Addition
        {"SUB", 0x1},   // Subtraction
        {"MOV", 0x2},   // Move (Register to Register)
        {"LOAD", 0x3},  // Load (Register from Memory)
        {"STORE", 0x4}, // Store (Register to Memory)
        {"LOADI", 0x5}, // Load Immediate (Immediate value to Register)
        {"JMP", 0x6},   // Unconditional Jump
        {"JZ", 0x7},    // Jump if Zero Flag is Set
        {"INC", 0x8},   // Increment Register
        {"DEC", 0x9},   // Decrement Register
        {"AND", 0xA},   // Logical AND
        {"OR", 0xB},    // Logical OR
        {"NOP", 0xC}    // No Operation
    };
}
```

This table associates mnemonics with their 4-bit opcodes, facilitating efficient translation from assembly to machine code.

Assembly Parsing Logic

The main parsing logic in `AssemblyParser::parse` converts assembly code into 8-bit machine instructions:

```
uint8_t opcode = static_cast<uint8_t>(getOpcode(mnemonic));
if (opcode == 0xFF)
{
    cerr << "Error: Unknown mnemonic '" << mnemonic << "'" << endl;
    continue;
}

uint8_t destValue = getOperand(dest);
uint8_t srcValue = src.empty() ? 5 : getOperand(src);
uint8_t instruction = 0;

if (srcValue == 5)
{
    instruction = (opcode << 4) | (destValue & 0x0F);
}
else
{
    instruction = (opcode << 4) | ((destValue & 0x03) << 2) | (srcValue & 0x03);
}
```

Here, each line is parsed to isolate the mnemonic, destination, and source. The opcode is shifted and combined with operands to create the final instruction.

Instruction Decoding and Execution

In `InstructionDecoder`, `decodeAndExecute` interprets each 8-bit instruction:

```
switch (opcode)
{
case 0x0: // ADD
    machine.registers[dest] += machine.registers[src];
    break;
case 0x1: // SUB
    machine.registers[dest] -= machine.registers[src];
    break;
case 0x2: // MOV (Register to Register)
    machine.registers[dest] = machine.registers[src];
    break;
case 0x3: // LOAD (Register from Memory)
    machine.registers[dest] = machine.memory[src];
    break;
```

The `switch` structure directs control to the appropriate operation, modifying `Machine` states like registers and flags based on each opcode.

Machine State Reset

The `Machine::reset` function prepares the machine for a fresh execution cycle:

```
void Machine::reset() {
    registers.fill(0);
    flags["Z"] = false;
    flags["PC"] = 0;
    memory.fill(0);
}
```

This function clears registers, resets the program counter, and initializes memory and flags, ensuring consistent starting conditions.

Main Execution Loop

In `main.cpp`, the loop executes the program, fetching instructions and displaying register states:

```
while (machine.programCounter < machineCode.size() && tc--)
{
    uint8_t instruction = machineCode[machine.programCounter];
    cout << endl
         << "PC: " << static_cast<int>(machine.programCounter) << endl;

    decoder.decodeAndExecute(instruction, machine);

    // Display register states after each instruction
    cout << "R0: " << static_cast<int>(machine.registers[0]) << "\t";
    cout << "R1: " << static_cast<int>(machine.registers[1]) << endl;
}
```

This main loop handles each instruction sequentially, updating the program counter and outputting the results, providing a simple visualization of CPU state evolution.

Example Program

We are given a program with the following instructions:

```
C/C++

LOADI R0, 1     // Load immediate value 1 into R0

LOADI R1, 3     // Load immediate value 3 into R1

ADD R0, R1      // Add R1 to R0 (R0 = R0 + R1)

STORE R0, 0x01  // Store R0 at memory address 0x01

DEC R1          // Decrement R1 by 1 (R1 = R1 - 1)

JZ 0x07         // Jump to address 0x07 if Zero flag is set

JMP 0x02        // Jump unconditionally to address 0x02

NOP              // No operation (does nothing)
```

We are also given the expected output of the program, including the final values of the program counter (PC) and the registers (R0, R1) at each step, along with the memory dump.

Program Execution Summary:

1. **PC = 0:**
   `R0 = 1, R1 = 0` (Initial state)
2. **PC = 1:**
   `LOADI R0, 1` → `R0 = 1, R1 = 0`
3. **PC = 2:**
   `LOADI R1, 3` → `R0 = 1, R1 = 3`
4. **PC = 3:**
   `ADD R0, R1` → `R0 = 4, R1 = 3`
5. **PC = 4:**
   `STORE R0, 0x01` → `Memory[0x01] = 4, R0 = 4, R1 = 3`

6. **PC = 5:**
   `DEC R1` → `R1 = 2`, `R0 = 4`
7. **PC = 6:**
   `JZ 0x07` → Not taken (`R0 != 0`), continue
8. **PC = 7:**
   `JMP 0x02` → Jump to `PC = 2`
9. **PC = 2:**
   `ADD R0, R1` → `R0 = 6`, `R1 = 2`
10. **PC = 3:**
    `STORE R0, 0x01` → `Memory[0x01] = 6`, `R0 = 6`, `R1 = 2`
11. **PC = 4:**
    `DEC R1` → `R1 = 1`, `R0 = 6`
12. **PC = 5:**
    `DEC R1` → `R1 = 0`, `R0 = 6`
13. **PC = 6:**
    `JMP 0x02` → Jump to `PC = 2`
14. **PC = 2:**
    `ADD R0, R1` → `R0 = 7`, `R1 = 0`
15. **PC = 3:**
    `STORE R0, 0x01` → `Memory[0x01] = 7`, `R0 = 7`, `R1 = 0`
16. **PC = 4:**
    `DEC R1` → No change (`R1 = 0`)
17. **PC = 5:**
    `JZ 0x07` → Jump to `PC = 7` (loop ends)

Memory Dump:

```
Address  |  Value

------------------

0x01     |  0x07
```

Output:

Final `Memory[0x01] = 7`, and the Program Counter ends at `PC = 7`.

```
(base) dhruvagrawal@Dhruvs-MacBook-Air-3 8_bit_instruction_set_project % ./machine
81 87 1 65 145 119 98 192

PC: 0
R0: 1   R1: 0

PC: 1
R0: 1   R1: 3

PC: 2
R0: 4   R1: 3

PC: 3
R0: 4   R1: 3

PC: 4
R0: 4   R1: 2

PC: 5
R0: 4   R1: 2

PC: 6
R0: 4   R1: 2

PC: 2
R0: 6   R1: 2

PC: 3
R0: 6   R1: 2

PC: 4
R0: 6   R1: 1

PC: 5
R0: 6   R1: 1

PC: 6
R0: 6   R1: 1

PC: 2
R0: 7   R1: 1

PC: 3
R0: 7   R1: 1

PC: 4
R0: 7   R1: 0

PC: 5
R0: 7   R1: 0

PC: 7
R0: 7   R1: 0

Memory Dump:
Address  |  Value
------------------
0x00     |  0x00
0x01     |  0x07
0x02     |  0x00
0x03     |  0x00
0x04     |  0x00
0x05     |  0x00
0x06     |  0x00
0x07     |  0x00
0x08     |  0x00
0x09     |  0x00
0x0A     |  0x00
0x0B     |  0x00
0x0C     |  0x00
0x0D     |  0x00
0x0E     |  0x00
0x0F     |  0x00
```

Challenges and Solutions

Limited Operand Size (0-3 Range):

- **Challenge:** The 8-bit instruction set allows only operands in the range 0-3 due to the 2-bit operand field.
- **Solution:** We designed operations to work within this limit, using memory or immediate values for larger numbers when needed.

Encoding Complexity:

- **Challenge**: Designing a compact 8-bit encoding for each instruction within a limited opcode space.
- **Solution**: A fixed binary format for each instruction was adopted, with operands fitting within the remaining bits while maintaining flexibility for future extensions.

Limited Resources (Registers and Memory):

- **Challenge**: Limited registers and small memory space constrained operations.
- **Solution**: Memory was managed efficiently with a small array, using registers effectively for sequential operations.

Typecasting for Non-Standard Data Types:

- **Challenge**: C++ typically prints only standard data types, but the project used `u_int8` for registers. This required typecasting to standard data types for correct printing.
- **Solution**: The `u_int8` values were typecast to standard data types (e.g., `int`) during print operations, ensuring compatibility with C++'s print functions.

References

- Computer Organization and Architecture by William Stallings
- Computer Architecture: A Quantitative Approach, Fifth Edition by David A Patterson and John L. Hennessy
- Lecture Slides