8 Bits CPU Simulation Documentation

Compter Architecture and Organization

Software Engineering Program,

Department of Computer Engineering,

School of Engineering, KMITL

67011093 Chavit Sarutdeechaikul
67011352 Theepakorn Phayonrat

Preface

This project, Design and Implementation of a Minimal 8-Bit CPU, was undertaken as part of the Computer Architecture and Organization course in the second year of Bachelor of Software Engineering at KMITL. It represents a practical exploration of fundamental computer architecture concepts, from logic design to instruction execution. The project allowed us to apply theoretical knowledge of CPU structure, instruction sets, and pipelining into a fully simulated and functioning processor. Using the Digital simulator by Hneemann, we developed and verified a modular CPU that embodies the essence of real processor operation. This report documents the design process, implementation details, and lessons learned throughout the development cycle.

Abstract

This mini project focuses on the design and simulation of a minimal 8-bit pipelined CPU using the Digital circuit simulation software. The CPU incorporates a five-stage pipeline—Fetch, Decode, Execute, Memory, and Write Back—and supports a custom instruction set architecture (ISA) with arithmetic, logical, branching, and I/O operations. The architecture features separate ROM and RAM modules, basic input and output ports, and a status register for flag management. Through simulation and test programs, the system demonstrates instruction execution, branching control, and interrupt handling. The project enhances understanding of CPU internals, digital logic integration, and the challenges of pipelined design, offering a hands-on approach to fundamental architectural concepts.

Contents

Chapter 1

Introduction

1.1 Project Overview

The 8-Bit CPU Simulation project aims to implement a simplified, educational model of a pipelined processor that captures the essential operations of modern CPUs while remaining manageable for simulation. The design follows a five-stage pipeline (Instruction Fetch, Instruction Decode, Execute, Memory, Write Back), allowing concurrent instruction processing for improved efficiency.

The CPU operates with 8-bit data paths and addresses, includes general-purpose and a special register (flag status register), and separates program memory (ROM) from data memory (RAM). A custom instruction set defines all operations, including data transfer (LD, ST, MOV), arithmetic and logic (ADD, SUB, MUL, AND, OR, XOR, NOT), branching (BZ, BNZ, BC, B), and I/O handling (RD, WR). Interrupt support and optional features such as basic caching or branch prediction may extend functionality.

Simulation and verification are performed in Digital, emphasizing modularity, waveform analysis, and clear documentation.

1.2 Background

Central Processing Units (CPUs) are the computational core of digital systems, responsible for executing instructions and managing data flow between memory and peripherals. Understanding how a CPU functions—from fetching an instruction to writing back results—is crucial in computer engineering education.

Traditional classroom learning often focuses on theoretical aspects such as microarchitecture, ISA design, and pipelining, but lacks direct visualization of hardware behavior. This project bridges that gap by using the Digital simulator to implement a CPU from basic logic components. By designing each pipeline stage, students explore how control signals, registers, buses, and memory interact to form a functioning processor.

The project serves as a foundational experience in CPU design, illustrating key topics such as instruction decoding, data hazards, memory access timing, and modular circuit organization.

1.3 Objective

1. **To design and implement** an 8-bit pipelined CPU with a custom instruction set architecture (ISA) using the *Digital* simulation environment.

2. **To apply theoretical concepts** of CPU architecture, including pipeline stages, ALU operations, branching, and flag management, in a practical design.

3. **To develop modular circuit components** (e.g., ALU, control unit, registers, and memory interfaces) that integrate seamlessly within the pipeline.

4. **To simulate and verify** the CPU's operation through test programs demonstrating arithmetic, logic, branching, stack, and interrupt handling.

5. **To enhance understanding** of digital logic design, data path organization, and hardware–software interaction in CPU execution.

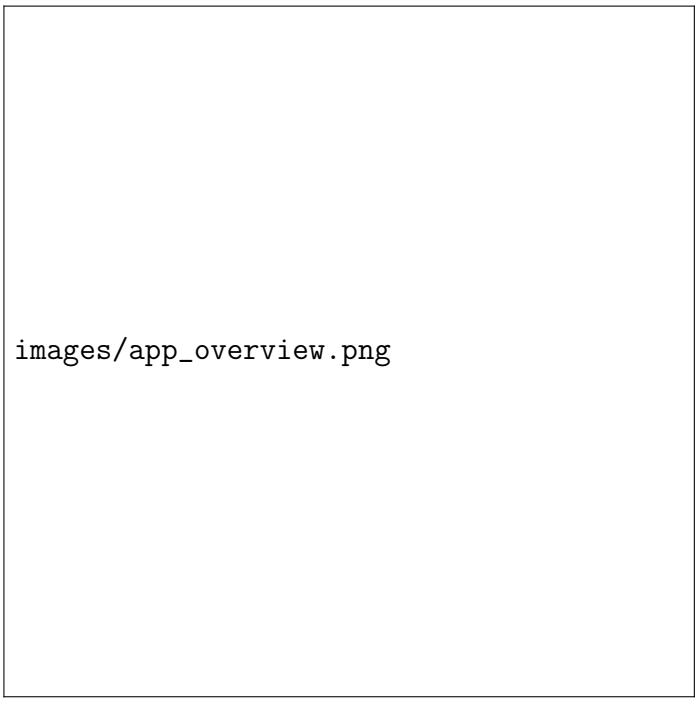
6. **To document** the architecture, instruction formats, and verification results clearly and comprehensively for academic evaluation.

Chapter 2

Project Overview

2.1 Hardware Design

2.1.1 Structure



2.1.2 Pipeline States

2.1.3 Pipeline Hazards Management

2.2 Software Design

2.2.1 Instructions

Instruction	Opcode + Fields	Description
LD	$0000 \vert IMMflag[18] \vert R_dest[13:11] \vert R_src1[2:0]$	Load immediate into register
MOV	$0001 \vert IMMflag[18] \vert R_dest[13:11] \vert R_src1[2:0] or IMM[7:0]$	Move data or immediate
ST	$0010 \vert IMMflag[18] \vert R_dest[13:11] \vert R_src1[2:0]$	Store register to memory
RD	$0011 \vert IMMflag[18] \vert R_dest[13:11] \vert R_src1[2:0]$	Read memory to register
ADD	$0100 \vert IMMflag[18] \vert R_dest[13:11] \vert R_src1[10:8] \vert R_src2[2:0] or IMM[7:0]$	Add two registers
SUB	$0101 \vert IMMflag[18] \vert R_dest[13:11] \vert R_src1[10:8] \vert R_src2[2:0] or IMM[7:0]$	Subtract two registers
MUL	$0110 \vert IMMflag[18] \vert R_dest[13:11] \vert R_src1[10:8] \vert R_src2[2:0] or IMM[7:0]$	Multiply two registers
WR	$0111 \vert IMMflag[18] \vert R_src1[2:0] or IMM[7:0]$	Write register to output
AND	$1000 \vert IMMflag[18] \vert R_dest[13:11] \vert R_src1[10:8] \vert R_src2[2:0] or IMM[7:0]$	Bitwise AND
OR	$1001 \vert IMMflag[18] \vert R_dest[13:11] \vert R_src1[10:8] \vert R_src2[2:0] or IMM[7:0]$	Bitwise OR
XOR	$1010 \vert IMMflag[18] \vert R_dest[13:11] \vert R_src1[10:8] \vert R_src2[2:0] or IMM[7:0]$	Bitwise XOR
NOT	$1011 \vert IMMflag[18] \vert R_dest[13:11] \vert R_src1[10:8] \vert R_src2[2:0] or IMM[7:0]$	Bitwise NOT
BC	$1100 \vert IMMflag[18] \vert R_check[13:11] \vert IMM[7:0]$	Branch if carry
BZ	$1101 \vert IMMflag[18] \vert R_check[13:11] \vert IMM[7:0]$	Branch if zero
BNZ	$1110 \vert IMMflag[18] \vert R_check[13:11] \vert IMM[7:0]$	Branch if not zero
B	$1111 \vert IMMflag[18] \vert R_check[13:11] \vert IMM[7:0]$	Unconditional branch

2.2.2 Assembler

We had written an assembler in Python.

```python
# ----------------- Assembly to 19-bit Hex Converter -----------------
# converter.py

import sys

OPCODES = {
    "LD":  "0000",
    "MOV": "0001",
    "ST":  "0010",
    "RD":  "0011",
    "ADD": "0100",
    "SUB": "0101",
    "MUL": "0110",
    "WR":  "0111",
    "AND": "1000",
    "OR":  "1001",
    "XOR": "1010",
```

```
    "NOT": "1011",
    "BC":  "1100",
    "BZ":  "1101",
    "BNZ": "1110",
    "B":   "1111"
}

def reg_3b(reg: str) -> str:
    """Convert rX to 3-bit binary (r0{r7)."""
    return format(int(reg.replace("r", "")), "03b")

def imm_8b(value: int) -> str:
    """Convert immediate number to 8-bit binary."""
    return format(value & 0xFF, "08b")

def assemble_binary(line: str) -> str:
    """Convert assembly line to 19-bit binary string."""

    # -- Remove comments while compiling -------------------------
    if ";" in line:
        line = line.split(';')[0]
    # ------------------------------------------------------------

    # -- Decode Opcode to Binary ---------------------------------
    line = line.strip()
    if not line or line.startswith("#"):
        return ""
    parts = line.replace(",", "").split()
    instr = parts[0].upper()
    opcode = OPCODES.get(instr, "????")
    # ------------------------------------------------------------

    # -- Set default binary --------------------------------------
    imm_flag = "0"
    r_dest = "000"
    r_src1 = "000"
    imm_or_r_src2 = "00000000"
    operands = parts[1:]
    # ------------------------------------------------------------

    def get_operand_bits(opnd) -> str:
        """Get operands binary bits"""
        nonlocal imm_flag
        # -- Check if second operand is immediate ------------
        if opnd.startswith("#"):
            # True: Set IsImm flag to 1 -----------------
            imm_flag = "1"
            return imm_8b(int(opnd[1:]))
        else:
            # False: Set IsImm flag to 0 ----------------
            return "00000" + reg_3b(opnd)
        # ------------------------------------------------

    # Assemble binary for each commnand -------------------------
    if instr == "BC" or instr == "BZ" or instr == "BNZ" or instr == "B":
        imm_flag = "1"
        r_dest = "000"
        r_src1 = "000"
        temp = 0
        if operands[0].startswith('#'):
            temp = int(operands[0][1:])
        imm_or_r_src2 = imm_8b(temp)
```

```
    elif instr == "NOT":
        # -- We found NOT -----------------------------------
        r_dest = reg_3b(operands[0])
        r_src1 = "000"
        imm_or_r_src2 = "00000" + reg_3b(operands[1])
    elif len(operands) == 2:
        # -- We found instruction which need 2 operands ------
        r_dest = reg_3b(operands[0])
        op2 = operands[1]
        if op2.startswith("#"):
            imm_flag = "1"
            imm_or_r_src2 = imm_8b(int(op2[1:]))
        else:
            imm_flag = "0"
            imm_or_r_src2 = "00000" + reg_3b(op2)
    elif len(operands) == 3:
        # -- We found instruction which need 3 operands ------
        r_dest = reg_3b(operands[0])
        r_src1 = reg_3b(operands[1])
        imm_or_r_src2 = get_operand_bits(operands[2])
    else:
        # -- We found empty line ----------------------------
        return ""
    # ------------------------------------------------------------

    return f"{imm_flag}{opcode}{r_dest}{r_src1}{imm_or_r_src2}"

def binary_to_hex(bin_str: str) -> str:
    """Convert binary string to hex (uppercase, no prefix)."""
    # -- Check if we need to convert to Hex or not ---------------
    if not bin_str:
        # -- If it is a empty line --------------------------
        return ""
    # ------------------------------------------------------------
    val = int(bin_str, 2)
    hex_str = format(val, "05X")  # 19 bits fit in 5 hex digits
                                  # (max 0x7FFFF)
    return hex_str

def cli_args_collect() -> list[str]:
    cli_args = sys.argv
    if len(cli_args) == 1:
        print("Please at least insert a file to convert")
        sys.exit(1)
    elif 2 <= len(cli_args) <= 3:
        input_file = cli_args[1]
        try:
            _ = input_file.split('.')
        except IndexError:
            output_file = f"{input_file}.hex"
        output_file = f"{input_file.split('.')[0]}.hex"

        if len(cli_args) == 3:
            output_file = cli_args[2]
            try:
                _ = output_file.split('.')[1]
            except IndexError:
                output_file = f"{output_file}.hex"
                print("Do not forget to add file extension .hex")
            if output_file.split('.')[1] != "hex":
                output_file = f"{output_file.split('.')[0]}.hex"
                print("Do not forget to change file extension to .hex")
```

```
    else:
        print("Too many arguments")
        sys.exit(1)
    return [input_file, output_file]


def main():
    [input_file, output_file] = cli_args_collect()

    with open(input_file, "r") as fin, open(output_file, "w") as fout:
        fout.write("v2.0 raw\n")
        for line in fin:
            binary = assemble_binary(line)
            if binary:
                hex_val = binary_to_hex(binary)
                fout.write(hex_val + "\n")

    print(f"Conversion complete. Hex output saved to '{output_file}'.")

if __name__ == "__main__":
    main()
```

Chapter 3

Installation and Execution Guide

3.1 Git Clone from the Remote Repository

```
git clone https://github.com/Pottarr/QtGroove.git
```

After that open project in Qt Creator, and run the program.

3.2 Alternative way for Windows users

You can download pre-release version (v0.1) from GitHub too. (Link in Appendix)

Chapter 4

Summary

4.1 Learning Outcomes

- We have learnt fundamental of concepts of creating good UX and UI.
- We have learnt how to develop multi-platform application using C++ Qt.
- We have learnt the workflow of project developing.
- We have learnt how to use Version Control to help developing application.

4.2 Accomplishment

We have created a user friendly multi-platform music player application.

Chapter 5

References

- Qt Group. (2025). *Qt Documentation*. Retrieved from `https://doc.qt.io/`

Chapter 6

Appendix

6.1 Github Repository

https://github.com/Pottarr/8Bit-CPU
https://github.com/hneemann/Digital