8 Bits CPU Simulation Documentation

Compter Organization and Architecture

Software Engineering Program,

Department of Computer Engineering,

School of Engineering, KMITL

67011093 Chavit Sarutdeechaikul
67011352 Theepakorn Phayonrat

Preface

Hello, World!
Line 1
Line 2
Line 3
Line 4
Line 5

Abstract

This project, titled QtGroove, presents the design and implementation of a music player application developed in the C++ programming language with Qt GUI framework. As part of the Object Oriented Programming course in Software Engineering at KMITL, QtGroove was created to develop a user-friendly multi-platform music player in C++ programming language that provide users with typical features found in general music player.

Contents

Chapter 1

Introduction

1.1 Project Overview

QtGroove is a graphic-based music player written in C++ using the Qt framework. The project aims to be a lightweight music player with a friendly user interface.

QtGroove will have the functions of a typical music player like a file browser, the ability to make playlists, showing music file info, and having a bit of extra functions like speed up playback or player customization.

1.2 Background

We wanted to created our own multi-platform GUI music player, which is efficience to navigate through the UI with low learning curve.

1.3 Objective

This project aims to create a lightweight and multi-platform music player as an alternative to other music players. The app can great for listening to local music files. The making of this app also serves as an experience for us to learn C++ and work with the qt framework.

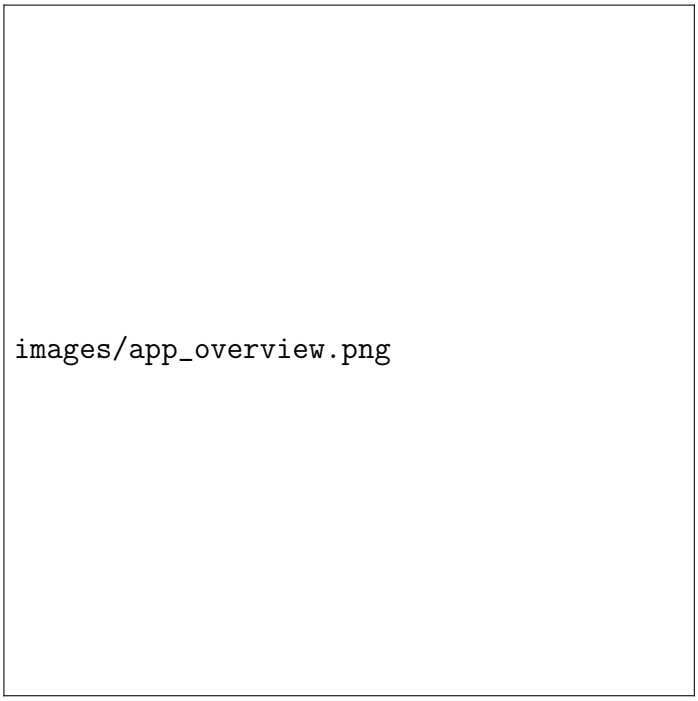
Since this is a duo project, it is a great opportunity to learn teamwork and strive to make the best products.

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 command ------------------------
    if 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/QtGroove`