MEF UNIVERSITY

YOUR FREEDOM IN LEARNING

COMP 206 Computer Architecture Project Report

Project: Dual-Core Computer Design

Şuayb Ş. ARSLAN

Ahmet Selim DİZER [041801087]
Abdullah Selim ÖZTEN [041801114]
Ömer Oğuz ÇELİKEL [041801090]

Introduction

This paper will give an idea about the dual core computer design. The project consists of three parts. In the first part, assembly code is taken and converted to machine code. This software is called assembly and it is of great importance in the execution of the program. In the second part, the microarchitecture design and application of the core is done with the given ISA. In the last part, simple CPU main components are applied.

Multi-Core Computer Overview

Multi-core architectures based on the replication of multiple processor cores on a single die. The core fits on a single processor submitted. Each core has its own ALU. They have a register file (data path), register ALU. They share a common memory.

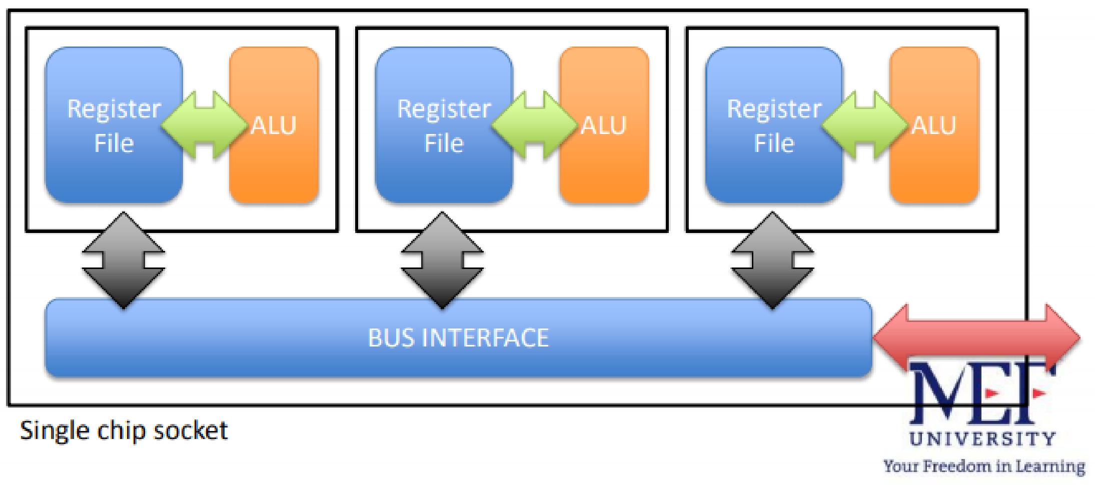


Figure 1: Single Chip Socket

Methodology

Hardware

The Dual-Core Computer required a significant amount of hardware design validation and implementation.

First, we created our Single Core and ALU components according to the given ISA specifications (Table 1 [1]).

Table 1: ISA for the cores.

opcode	menomic	name	operation
0000	BRZ x	Branch on Zero	if ACC = 0: PC $\leftarrow$ PC + signed(x)
0001	BRN x	Branch on Negative	if ACC < 0: PC $\leftarrow$ PC + signed(x)
0010	LDI x	Load Immediate	ACC $\leftarrow$ signed(x)
0011	LDM x	Load from Memory	ACC $\leftarrow$ RAM[x]
0100	STR x	Store	RAM[x] $\leftarrow$ ACC
0101	ADD x	Add	ACC $\leftarrow$ ACC + RAM[x]
0110	SUB x	Subtract	ACC $\leftarrow$ ACC - RAM[x]
0111	MUL x	Multiply	ACC $\leftarrow$ ACC * RAM[x]
1000	DIV x	Divide	ACC $\leftarrow$ ACC / RAM[x]
1001	NEG x	Negate	ACC $\leftarrow$ -ACC
1010	LSL x	Logical Shift Left	ACC $\leftarrow$ ACC (shift left by x times)
1011	LSR x	Logical Shift Right	ACC $\leftarrow$ ACC (shift right by x times)
1100	XOR x	Bitwise XOR	ACC $\leftarrow$ ACC ˆ RAM[x]
1101	NOT x	Bitwise NOT	ACC $\leftarrow$!ACC
1110	AND x	Bitwise AND	ACC $\leftarrow$ ACC && RAM[x]
1111	ORR x	Bitwise OR	ACC $\leftarrow$ ACC || RAM[x]

ALU was required for specific opcodes. In these cases, the PC register was added to signed(x) by controlling the ACC. Then we used other necessary circuit elements to create a single core. These elements (sign extend, MUX, adder, register, DeMUX, logic gates). The required Control Logic Unit was created for a single core. After creating the truth table, the commands required for ALU Select, ACC MUX, ACC Selection and Branch Selection were seen. Required connections in Single Core have been made.

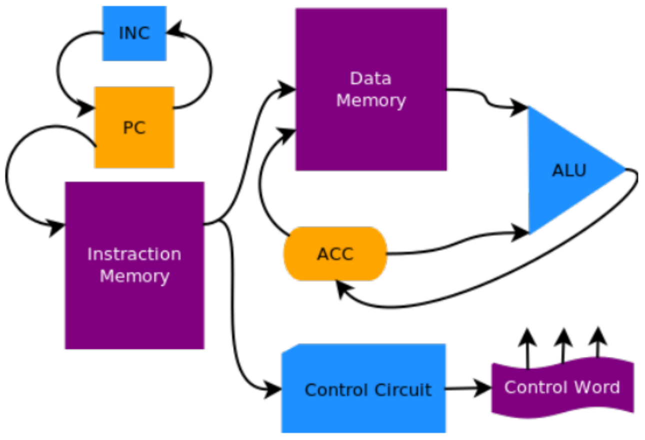


Figure 2: Block diagram of single-address machine.

Finally, the CMU was built to make it dual core. The appropriate instructions for the kernels are given by the CMU. CMU controls the instruction flow between installation ROMs. After the CMU_Control was designed, it was connected to the ROMs of the 2 Cores and Cores with the correct connections (Figure 3) . Also you can see our final design in Figure 3.1.

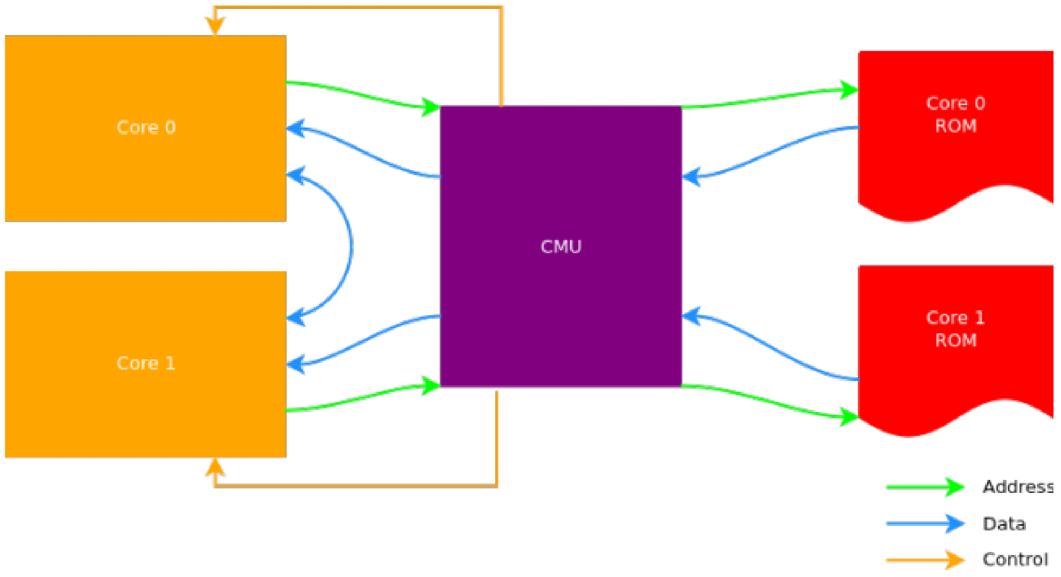


Figure 3: Connection of the Multicore CPU components.

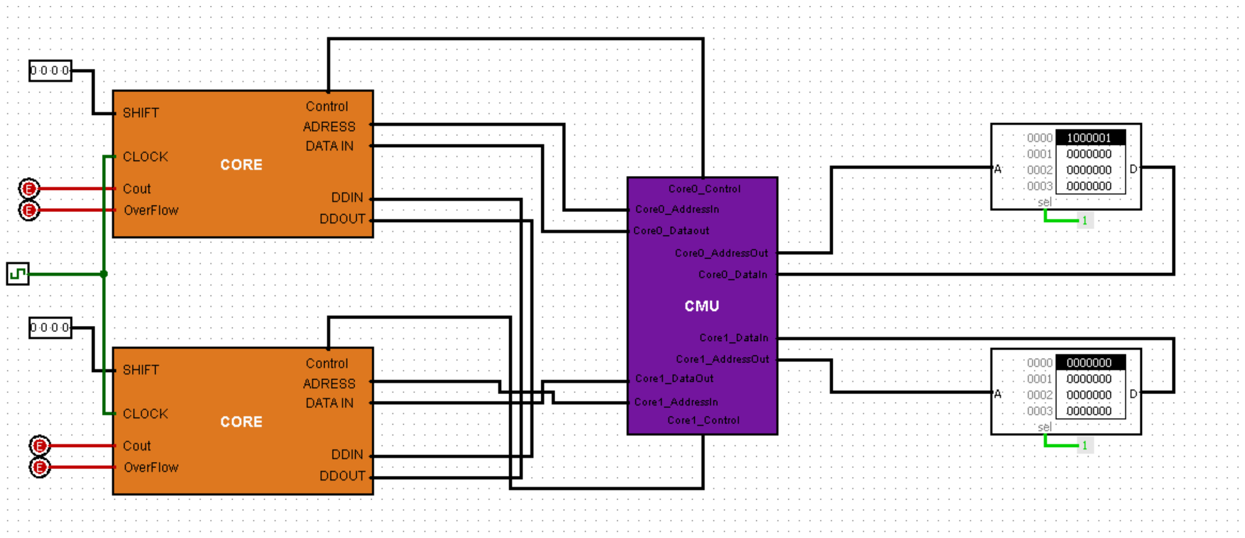


Figure 3.1 Connection of the Multicore CPU components. (our design)

Software

The hex format required for the ROM was written in Java code. In addition to the Java file, the assembly.txt file was created and the assembly codes were thrown into that file. When the code ran, it took the information from the txt file and saved our file on the desktop.

Implementation

ALU

We added the operations in Table1 to the ALU and connected the outputs of all operations to MUX. MUX Select is also called ALU Select. This select consists of 4 bits and works according to OPcodes.

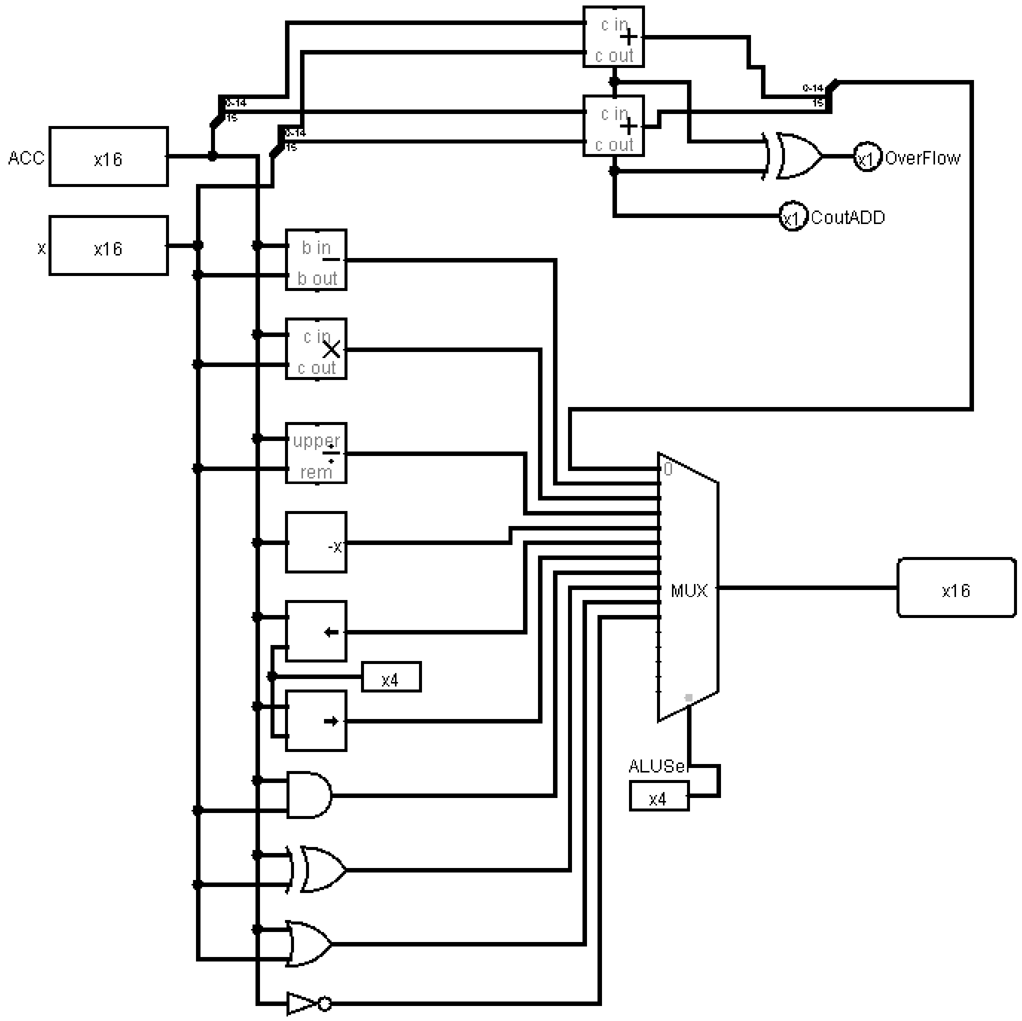


Figure 4: ALU design in Logisim.

CORE

Our 16-bit core consists of 5 inputs and 4 outputs. Our Core design includes the **ALU** that we created previously, a 4kb ram, two 16-bit multiplexers, a demux, our control logic unit (**CLU**), a program counter (**PC**), **ACC** register, 12bit to 16 bit extender, and a group of logic gates. Gets the **DATAIN** 16 bit input to split as *xxxxyyyyyyyyyyyy* first 4 being the op code and the last 12 being the data that we will use. **OP** code that we extracted enters the **CLU** and gives the outputs in respect to our truth table. Our demux decides whether our **ACC** enters **ALU** or directly goes to the ram. We have specific inputs and outputs for MOV operation **DDOUT**(Direct Data Out) and **DDIN**(Direct Data In) that only works when requested. There is a 3-bit **Control** input from **CMU** that we added for **MOV** operation.

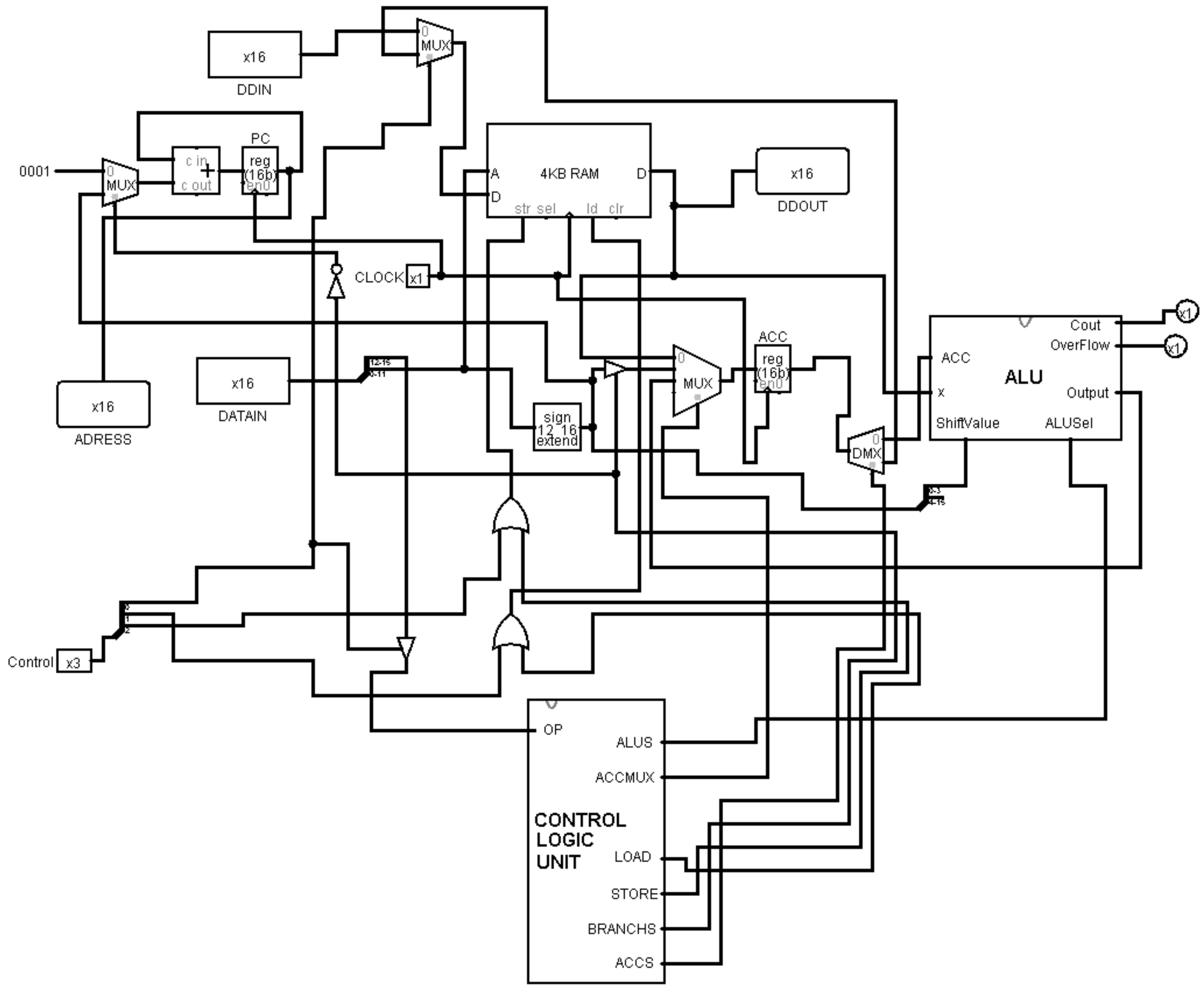
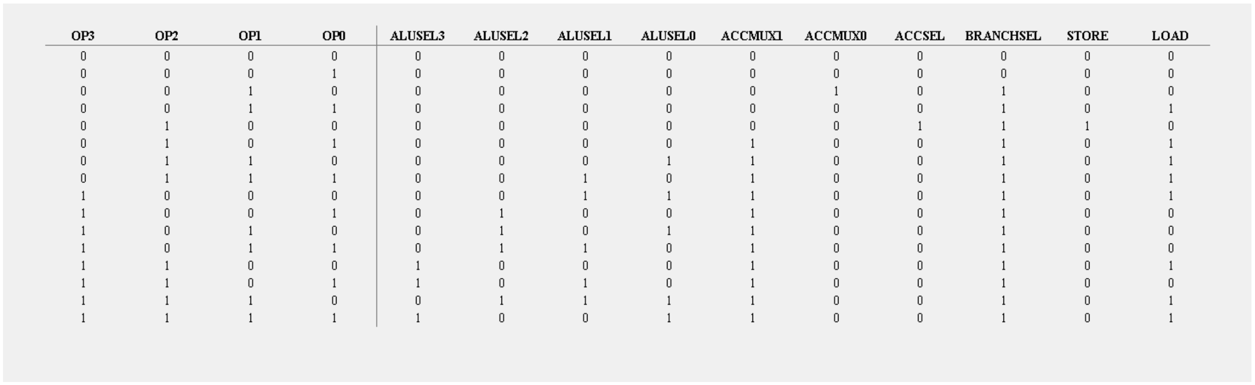


Figure 5: Single Core design in Logisim.

CLU(Control Logic Unit)

Clu Control changes the signal from Core according to instructions in Table 1. This changes according to Table 2.



OP3	OP2	OP1	OP0	ALUSEL3	ALUSEL2	ALUSEL1	ALUSEL0	ACCMUX1	ACCMUX0	ACCSEL	BRANCHSEL	STORE	LOAD
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	1	0	1	0	0
0	0	1	1	0	0	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	0	0	1	1	1	0
0	1	0	1	0	0	0	0	1	0	0	1	0	1
0	1	1	0	0	0	0	1	1	0	0	1	0	1
0	1	1	1	0	0	1	0	1	0	0	1	0	1
1	0	0	0	0	0	1	1	1	0	0	1	0	1
1	0	0	1	0	1	0	0	1	0	0	1	0	0
1	0	1	0	0	1	0	1	1	0	0	1	0	0
1	0	1	1	0	1	1	0	1	0	0	1	0	0
1	1	0	0	1	0	0	0	1	0	0	1	0	1
1	1	0	1	1	0	1	0	1	0	0	1	0	0
1	1	1	0	0	1	1	1	1	0	0	1	0	1
1	1	1	1	1	0	0	1	1	0	0	1	0	1

Table 2: CLU's Truth Table

ALU Select: This signal is the select signal that ALU determines the operation.

ACCMUX: This signal is a select of MUX with 3 inputs. For the first input Load from Memory operation status. For the second input Load Immediate operation status. For the last input all ALU operations status.According to the output of these inputs, it puts the result in the ACC.

ACCSelect: This signal works in collaboration with the store signal. In case of Store instruction, move the value ACC to RAM. If the instruction is not Store, it remains 0 in all other instructions.

BranchSelect: This signal is for Branch instructions. If this signal is 1, The Load immediate instruction does not work and adds the sign (x) value to the PC instead of 1.

Store: This signal works in collaboration with the ACCSelect signal. If the signal is 1, it opens RAM's Store. If the instruction is not Store, it remains 0 in all other instructions.

Load: The load signal works for all instructions that we receive data from RAM. If the signal is 1, it opens RAM's Load.

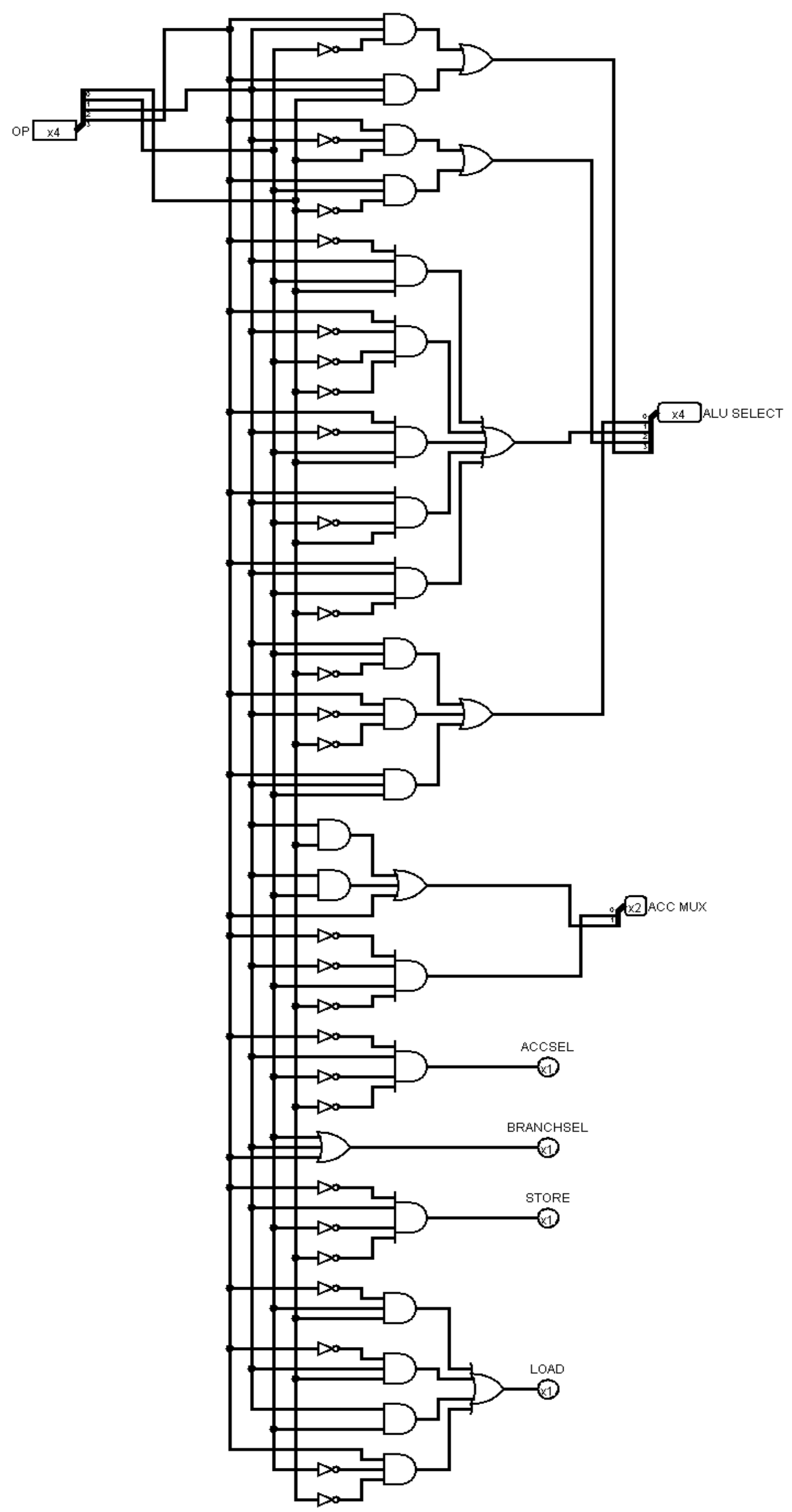


Figure 6: CLU(Control Logic Unit) design in Logisim.

CMU (Core Management Unit)

As its name suggests CMU Manages the cores and ensures our cores are working together properly it has 4 inputs two of them being address pass-through and the other two being the 25-bit data inputs from rom's and splits these 25-bit inputs according to the data itself. When the move operation is requested the data can be split as XYYYYYYYYYYYYZZZZZZZZZZZZ X being the enabler for this operation and the following bits are the addresses. When move is not requested data splits as (XXXXXXXXX)YYYYYYYYYYYYYYYY and the first 9 bits are not used, the last 16 bits directly goes to core to engage.

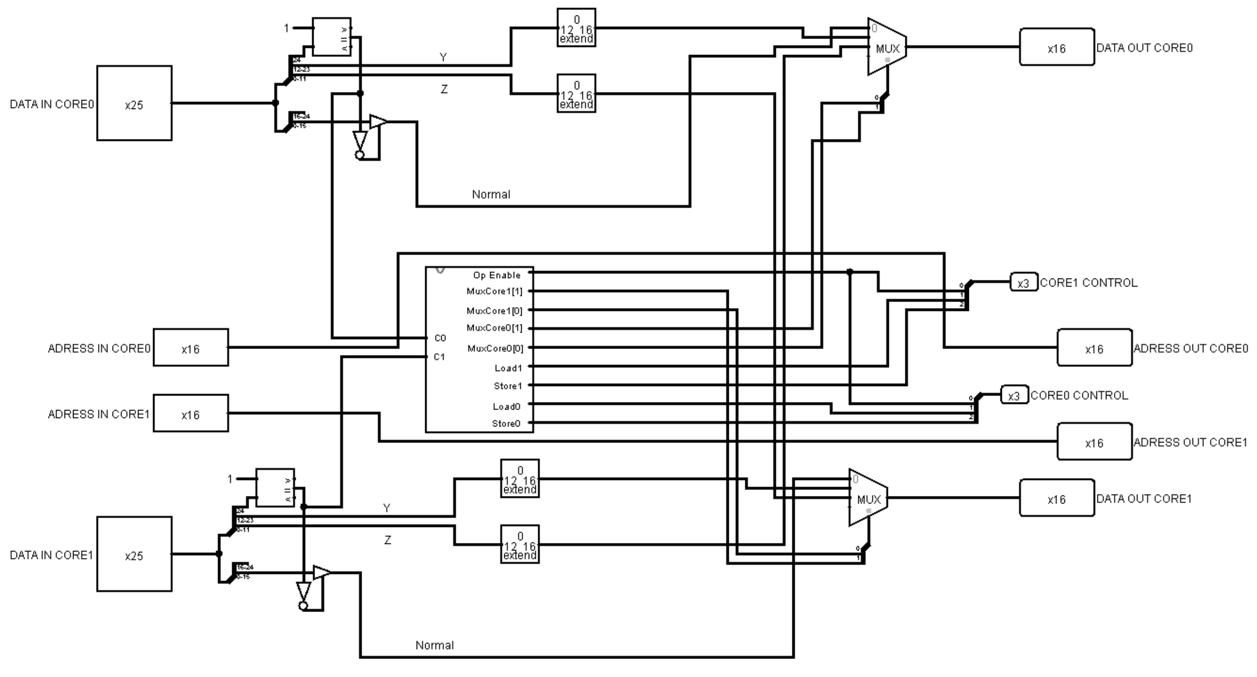
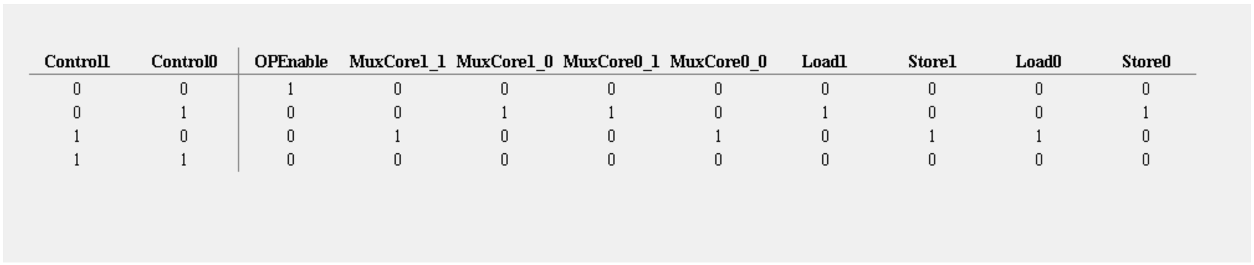


Figure 7: CMU design in Logisim.

CMU Control

In order to create Table 2, we examined the Control inputs in 3 cases. The first case is the situation where the control inputs are 00, that is no cores moves. The second case is when control signals are 01, that is only Core0 moves .The third case is when control signals are 10, that is only Core1 moves.



Control1	Control0	OPEnable	MuxCore1_1	MuxCore1_0	MuxCore0_1	MuxCore0_0	Load1	Store1	Load0	Store0
0	0	1	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	0	0	1
1	0	0	1	0	0	1	0	1	1	0
1	1	0	0	0	0	0	0	0	0	0

Table 3: CMU Control's Truth Table

Control: This input signal comes from whether the last bit of the Data from the ROM is 1 or not. If it is 1, it sends a 1 signal, otherwise it sends a 0 signal. This event takes place in both cores.

OP Enable: This signal is for the CLU (Control Logic Unit) circuit to be disconnected in MOV situations. It also determines where the Address to enter the RAM should come from.

MUX_Core: These signals select which Data should be sent. It has functions similar to control signals. It has 3 states as in control signals.Each Core has its own MUX_Core signal.

Load: The load signal is different for each Core. In the MOV instruction, Core from which Data will be taken becomes 1, while Core from which Data will be moved is 0.

Store: The Store signal is different for each Core. In the MOV instruction, Core from which Data will be taken becomes 0, while Core from which Data will be moved is 1.

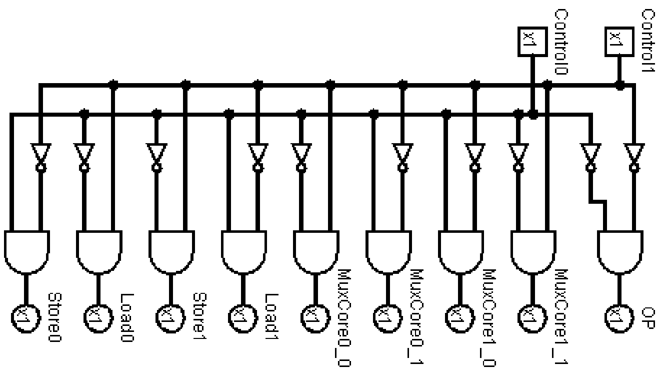


Figure 8: CMU Control design in Logisim.

Software Development

The Initial Sample Code file was not enough and we developed it. The algorithm was created for two cases: single-core and dual-core. We have read line by line from txt. We then split the information in the row. The first part was the instruction part. In the second part, it was done by converting the numbers into single core in Initial Sample Code ready. Dual-core was a little different. Because the second column gave which core to write to. We kept this information in the arrayList and performed the printing to the required cores by checking if. There was an extra column for MOV. That's why we made it according to the length property.

The source code is shown below.

```java
import java.awt.*;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Scanner;

public class Main {
    public static ArrayList<String> cores = new ArrayList<String>();


    public static void main(String[] args) throws IOException {

        if (args.length < 1) {
            System.out.println("There is no Assembly file to compile!");
            System.exit(1);
        }
        File inputFile = null;
        Scanner input = null;

        try {
            inputFile = new File(args[0]);
            input = new Scanner(inputFile);

        } catch (FileNotFoundException e) {
            System.err.println("No such File!");
            System.exit(1);
        }
        // Initializing files for write operation
        String homeDir = System.getProperty("user.home");
        FileOutputStream single = new FileOutputStream(homeDir + "/Desktop/single-core");
        FileOutputStream core0 = new FileOutputStream(homeDir + "/Desktop/core-0");
        FileOutputStream core1 = new FileOutputStream(homeDir + "/Desktop/core-1");
        single.write("v2.0 raw\n".getBytes());
        core0.write("v2.0 raw\n".getBytes());
        core1.write("v2.0 raw\n".getBytes());


        // Iterate through file
        System.out.println(cores.toString());
        int counter = 0; // to count every step of cores
        while (input.hasNext()) {
            String inputText = input.nextLine();
            String controlWord = getCW(inputText) + " ";

            // 0 for core 0
            if(cores.get(counter).equals("0")){
                core0.write(controlWord.getBytes());
            }

            // 1 for core 1
            else if(cores.get(counter).equals("1")){
                core1.write(controlWord.getBytes());
            }
            else{ // X durumu
                core0.write(controlWord.getBytes());
                core1.write(controlWord.getBytes());
            }

            counter+=1;

        } //while end

        // Opening directory of core files.
        Desktop desktop = Desktop.getDesktop();
        desktop.open(new File(homeDir + "/Desktop/single-core"));
        desktop.open(new File(homeDir + "/Desktop/core-0"));
        desktop.open(new File(homeDir + "/Desktop/core-1"));

        // Closing File Streams of files.
        single.close();
        core0.close();
        core1.close();
    }

    /**
     * This helper method gets a menomic and returns the corresponding binary string
     * @param menomic
     * @return binary string
     */

    private static String getOpCode(String menomic) {
        menomic = menomic.toUpperCase();
        switch (menomic) {
            case "BRZ":
                return "0000";
            case "BRN":
                return "0001";
            case "LDI":
                return "0010";
            case "LDM":
                return "0011";
            case "STR":
                return "0100";
            case "ADD":
                return "0101";
            case "SUB":
                return "0110";
            case "MUL":
                return "0111";
            case "DIV":
                return "1000";
            case "NEG":
                return "1001";
            case "LSL":
                return "1010";
            case "LSR":
                return "1011";
            case "XOR":
                return "1100";
            case "NOT":
                return "1101";
            case "AND":
                return "1110";
            case "ORR":
                return "1111";
            case "MOV":
                return "1";
        }
        return null;
    }
    /**
     * This helper method gets a number and returns translates it to a binary string
     * @param number address or an immediate value
     * @return binary version of the given number
     */
    private static String getBinaryString(String number) {
        int num = Integer.parseInt(number);

        String binaryString = Integer.toBinaryString(num);

        if (number.contains("-"))
            binaryString = binaryString.substring(27, 32);

        if (binaryString.length() < 12) {
            int length = binaryString.length();
            StringBuilder stringBuilder = new StringBuilder();
            for (int i = 0; i < 12 - length; i++)
                stringBuilder.append("0");
            stringBuilder.append(Integer.toBinaryString(num));
            return stringBuilder.toString();
        } else if (binaryString.length() > 12)
            System.out.println("Number can't be represented with 12 bits!");

        return binaryString;
    }

    /**
     * This helper method gets the input string and returns the control word in hexadecimal form
     * @param inputText a line from the assembly file
     * @return control word in hexadecimal form
     */
    private static String getCW(String inputText) {
        String[] array = inputText.split("\\s");
        String menomic = array[0];
        StringBuilder output = new StringBuilder();
        output.append(getOpCode(menomic));
        // cores list shows which core is
        cores.add(array[1]);

        if (array.length == 2) {
            String address = array[1];
            output.append(getBinaryString(address));
        } else {
            String address = array[2];
            output.append(getBinaryString(address));
            // for MOV method we have one more length(2).
            if (array.length == 4) {
                String address2 = array[3];
                output.append(getBinaryString(address2));
            }

        }
        int temp = Integer.parseInt(output.toString(), 2);
        return Integer.toHexString(temp);
    }
}
```

Conclusions

First, we implemented Core, and then we implemented Control Logic, creating the truth table that appears in Table 2 to select the instructions.Then we had to create a CMU to do MultiCore, so we thought about how to create a CMU circuit and designed a CMU circuit. Then we created CMU Control to select the instructions as we applied for Core.(Table 3) Finally, we wrote an assembler using Java to convert the assembly code given to us to a hexadecimal number.

Achievement

It was a good example to teach how a Core works in this project. Not only Core, also Cache, CMU, Assembler and Ram, we understood how it works and implemented it. From Single Core to multi core, we learned how to do it. Using the instructions given, we learned how to implement the hardware.

Acknowledgements and Distribution of Credit

We would like to thank Ayşenaz Ezgi Ergin and Alp Gökçek for their help with our questions and Project Sessions, which were of great help in the development of the project.

<u>The distribution of credit for each part of the project:</u>

- **Hardware Simulation and Verification**
  - 35% : Ahmet Selim Dizer
  - 35% : Abdullah Selim Özten
  - 30% : Ömer Oğuz Çelikel
- **Hardware Implementation**
  - 35% : Ahmet Selim Dizer
  - 35% : Abdullah Selim Özten
  - 30% : Ömer Oğuz Çelikel

- **Software Development**
  - 30% : Ahmet Selim Dizer
  - 30% : Abdullah Selim Özten
  - 40% : Ömer Oğuz Çelikel

Regardless of the percentages stated, this project was well coordinated and completed as a result of good teamwork.

References

- COMP206 Computer Architecture - (c) MEF University, Spring 2021 - PROJECT
- Logic & Computer Design Fundamentals, 5/E, M. Morris R. Mano, Charles R. Kime, Tom Martin
- Computer Organization and Design, The Hardware/Software Interface, 5th Edition, David Patterson and John Hennessy.

Signatures

[illegible] [illegible] [illegible]

Ahmet Selim Dizer Ömer Oğuz Çelikel Abdullah Selim Özten