



COMP206 – COMPUTER ARCHITECTURE

DUAL CORE 16 BIT CPU PROJECT

INSTRUCTOR: ŞEFİK ŞUAYB ARSLAN

SELÇUK COŞKUN

041801079

TOLGA KAAAN TÜTÜN

041601015

Due Date: 31st of May 2020

## Table of Contents

<b>Objective .....</b>	<b>2</b>
<b>The First Control Logic.....</b>	<b>3</b>
<b>The Second Control Logic.....</b>	<b>5</b>
<b>The ALU (Arithmetic Logic Unit) .....</b>	<b>7</b>
<b>The CPU Core .....</b>	<b>8</b>
<b>Why Two Control Logics? .....</b>	<b>9</b>
<b>The CMU (Core Management Unit) .....</b>	<b>10</b>
<b>Dual Core CPU – 1.....</b>	<b>11</b>
<b>Dual Core CPU – 2.....</b>	<b>12</b>
<b>The Assembler .....</b>	<b>13</b>
<b>Work Acknowledgement.....</b>	<b>17</b>

## Objective

The aim of this project was to familiarize ourselves with how a CPU operates and how to build one in Logisim. We built a dual core CPU that employs the table of instructions available below. We also wrote up an assembler that implements these instructions.

Table 1: ISA for the cores.

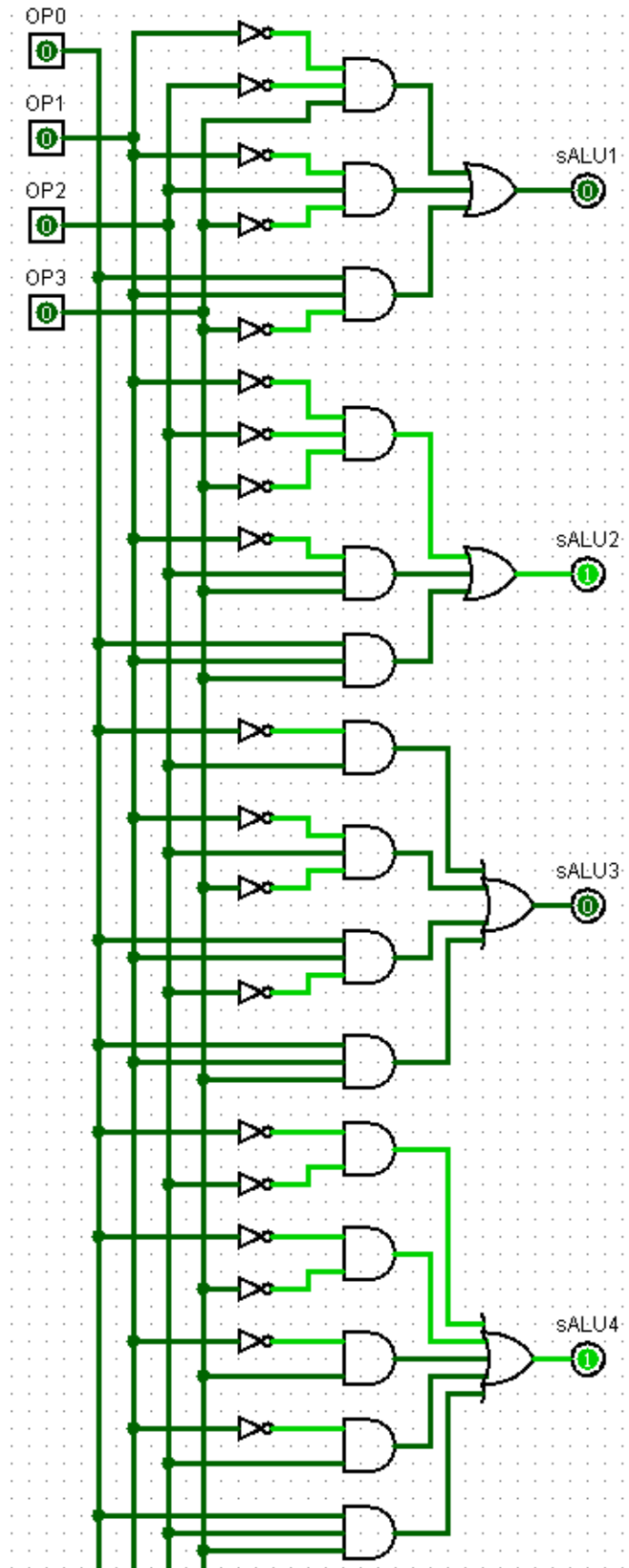
opcode	menomic	name	operation
0000	BRZ $x$	Branch on Zero	if $ACC = 0$ : $PC \leftarrow PC + \text{signed}(x)$
0001	BRN $x$	Branch on Negative	if $ACC < 0$ : $PC \leftarrow PC + \text{signed}(x)$
0010	LDI $x$	Load Immediate	$ACC \leftarrow \text{signed}(x)$
0011	LDM $x$	Load from Memory	$ACC \leftarrow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

Once these instructions were implemented, we obtained **two control logics**. Why there is two of them will be explained as we reach the dual core architecture part.

## The First Control Logic

We called this one ff7controledition for random naming purposes.

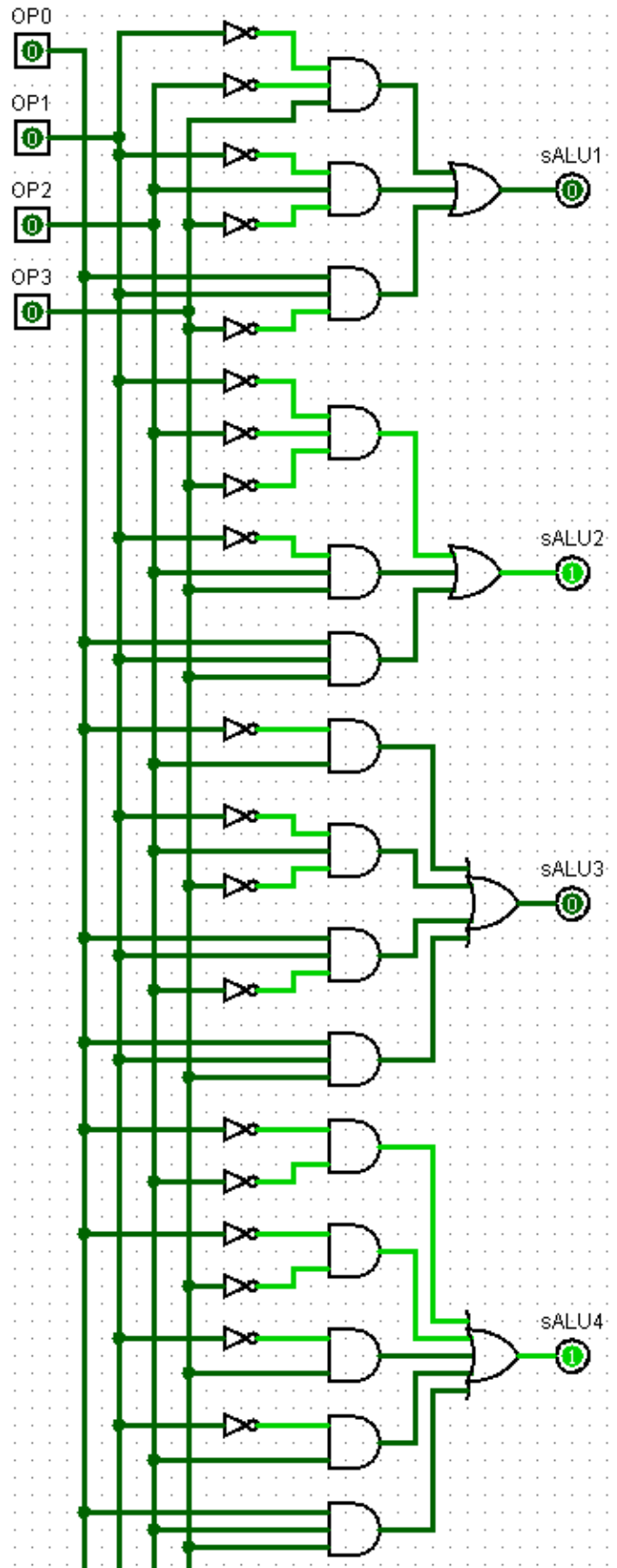
This control logic is able to execute all the instructions aforementioned on the graph in the overview but lack the ability to operate two cores at once.





## The Second Control Logic

This one is aptly named control and unfortunately is the final draft of our control logic. This control logic cannot write to memory. As in “cannot write” it makes the memory enter “ffff” in wrong locations. However, this control logic is able to operate both cores at once and seemed to only fail in the STR (store) operation in our testing.



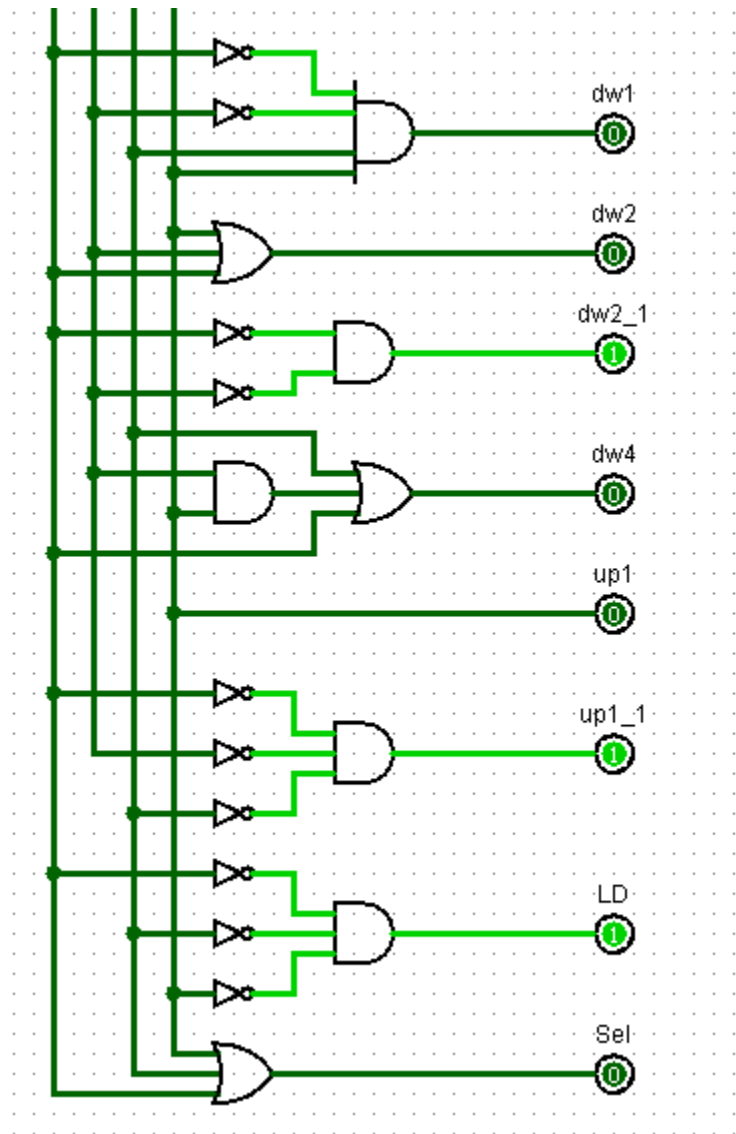


Figure 2: Control Logic (control)

## The ALU (Arithmetic Logic Unit)

The arithmetic logic unit implements:

- Logical Shift Left → sALU signal: 1000
- Logical Shift Right → sALU signal: 0111
- Addition → sALU signal: 1001
- Subtraction → sALU signal: 1011
- OR Operation → sALU signal: 0100
- NOT Operation → sALU signal: 0011
- AND Operation → sALU signal: 0010
- Multiplication → sALU signal: 0101
- Division → sALU signal: 1010
- Negation → sALU signal: 0110

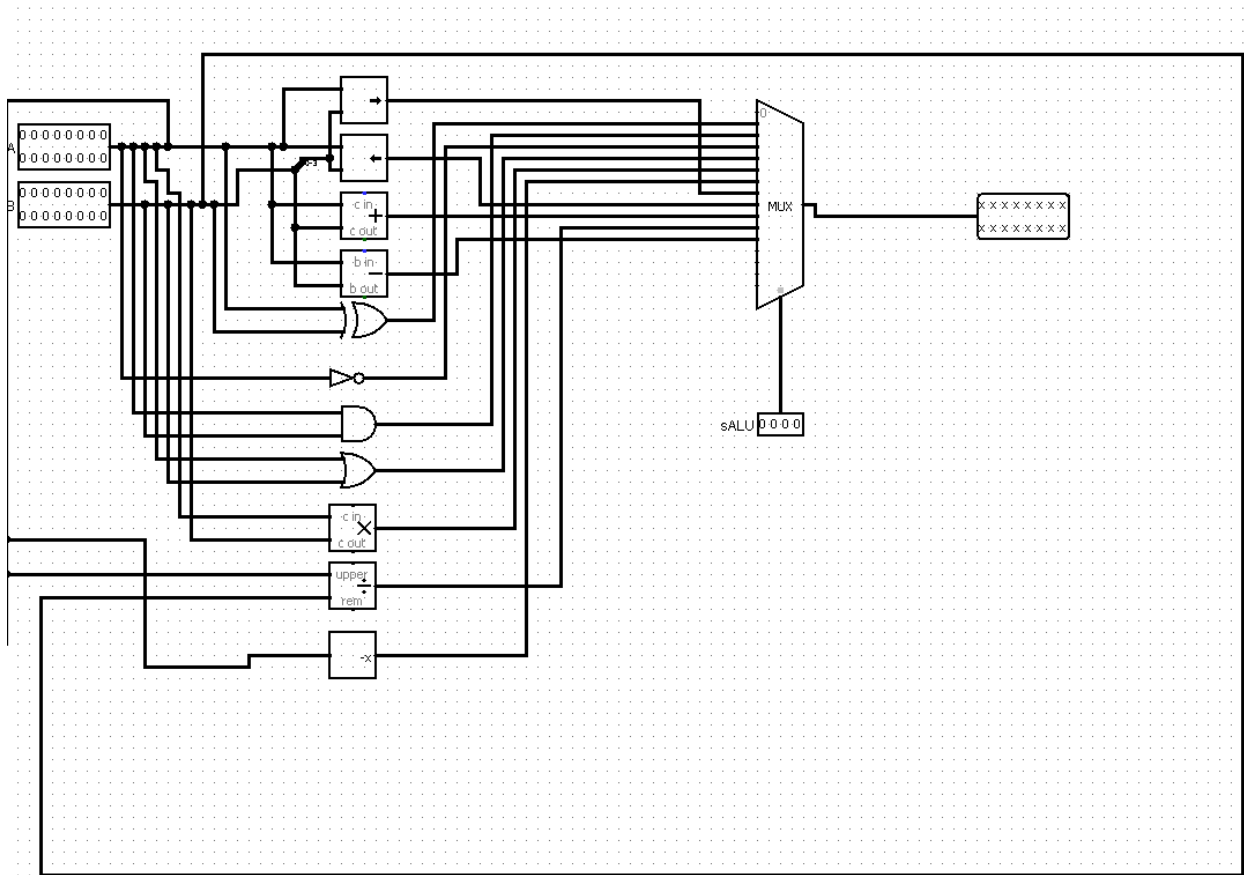


Figure 3: The ALU (Arithmetic Logic Unit)



## The CPU Core

The CPU core is 16-bits and takes 25-bit data and 16-bit addresses. The move addresses are 12-bits.

This is a single cycle single core model CPU that we have prepared and transformed to fit into a dual core architecture to utilize a CMU (CPU Memory Unit). This CPU utilizes the ff7controledition Control Logic to operate. There are 7 signals that connect to the control logic.

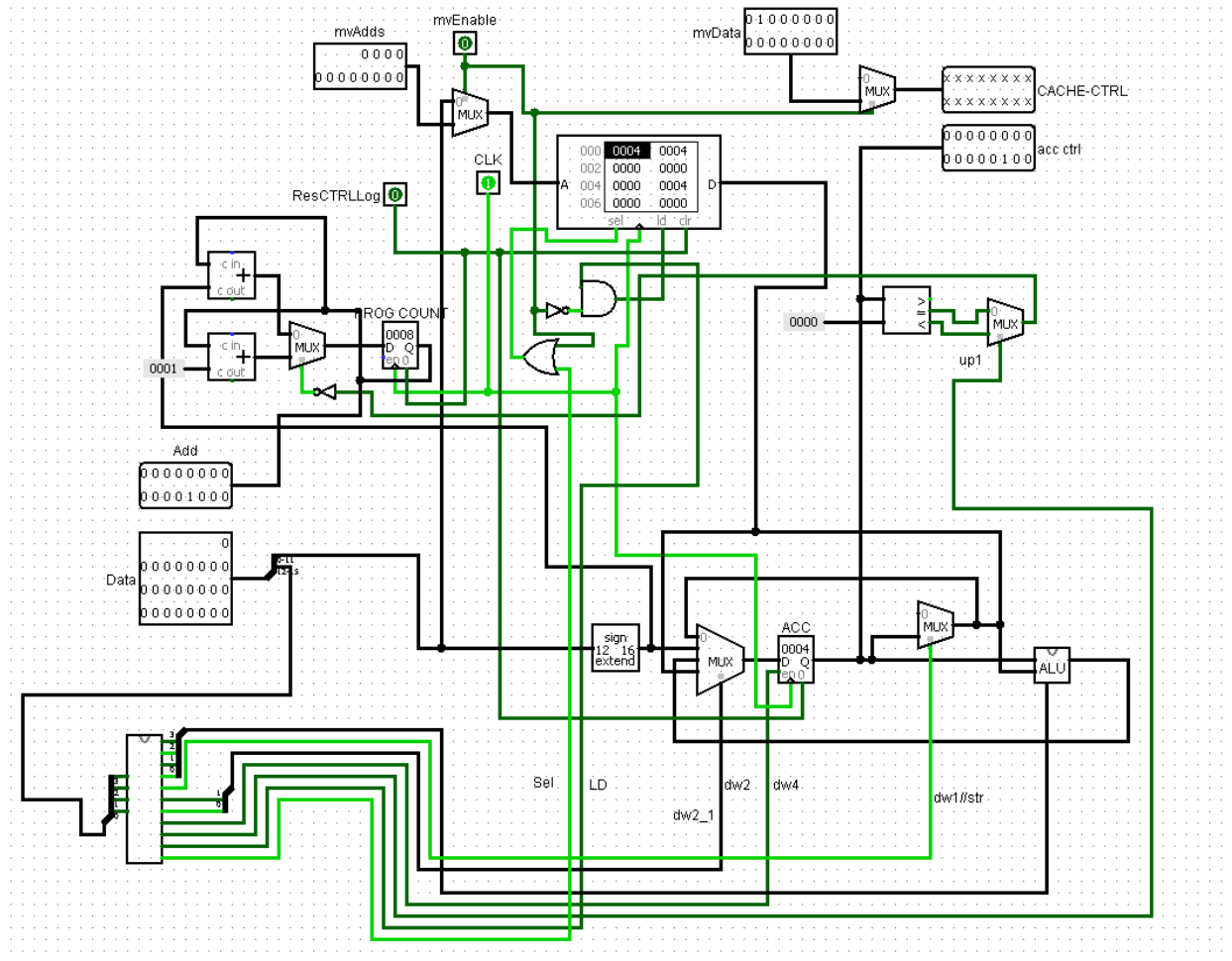


Figure 4: CPU Core utilizing “ff7controledition” with dual core error

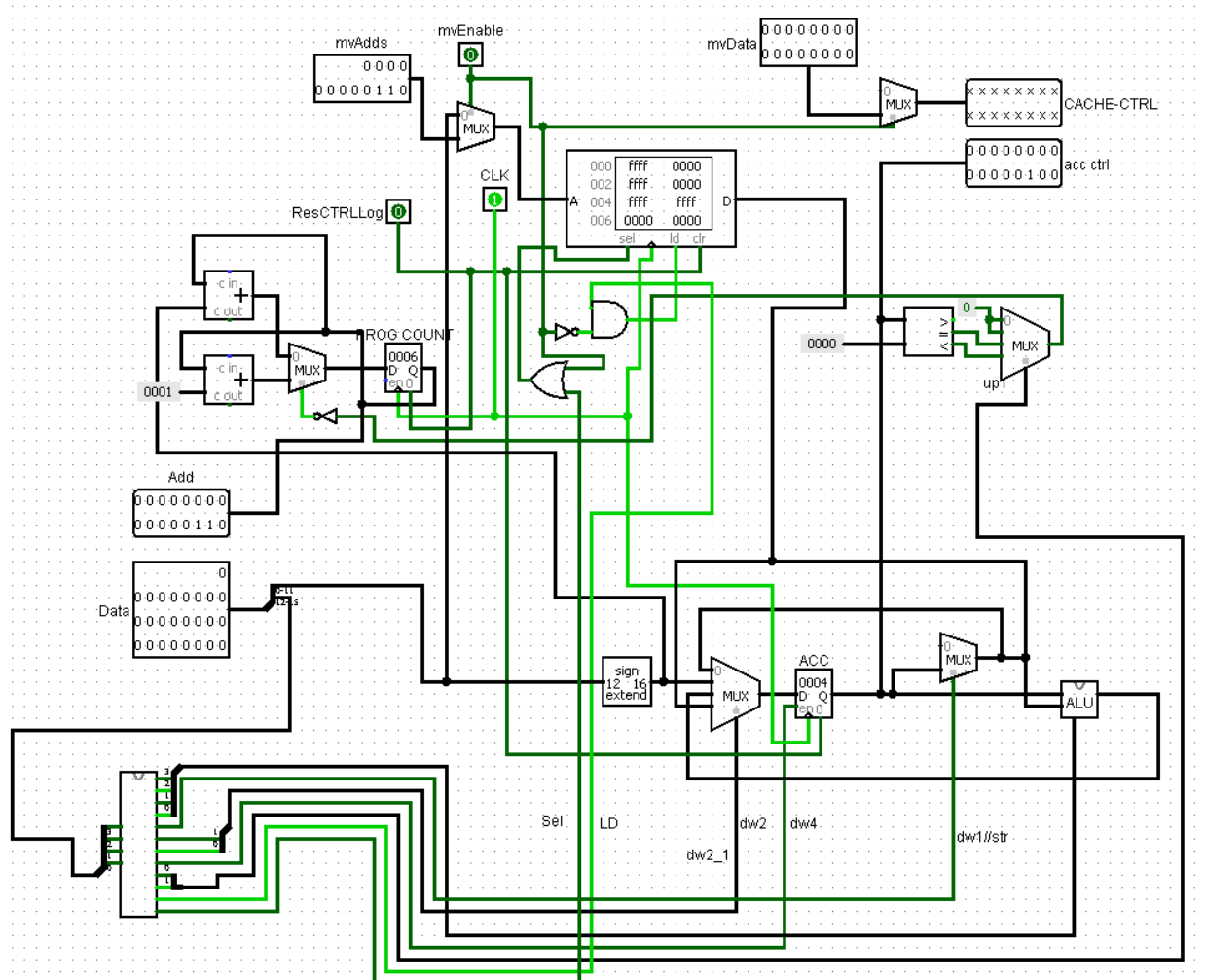


Figure 5: CPU Core utilizing “control” with memory write error

## Why Two Control Logics?

Amid difficulties and late-night work sessions the truth table of the control logic capable of writing to memory was lost and through many tries could not be recovered. We have spotted that there is a problem with the “up1” signal. Then another control logic was devised from the earlier revisions of the said control logic to fix the bug where both cores could not operate. This fix worked albeit with a caveat. The memory write would not work. So as a team we had to come to a compromise to ensure that we can present our efforts in the most **functional** way we can.

We devised two identical dual core circuits which one utilizes “ff7controledition” and shows our CPUs memory writing instruction handling capabilities and the other utilizes “control” shows our CPUs instruction execution while utilizing both cores capabilities.

That way we could show two semi-functional CPU’s instead of a wholly broken one.

## The CMU (Core Management Unit)

The CMU adjusts the workload of two cores by distributing them. This way one core gets the majority of the operations and they work in parallel.

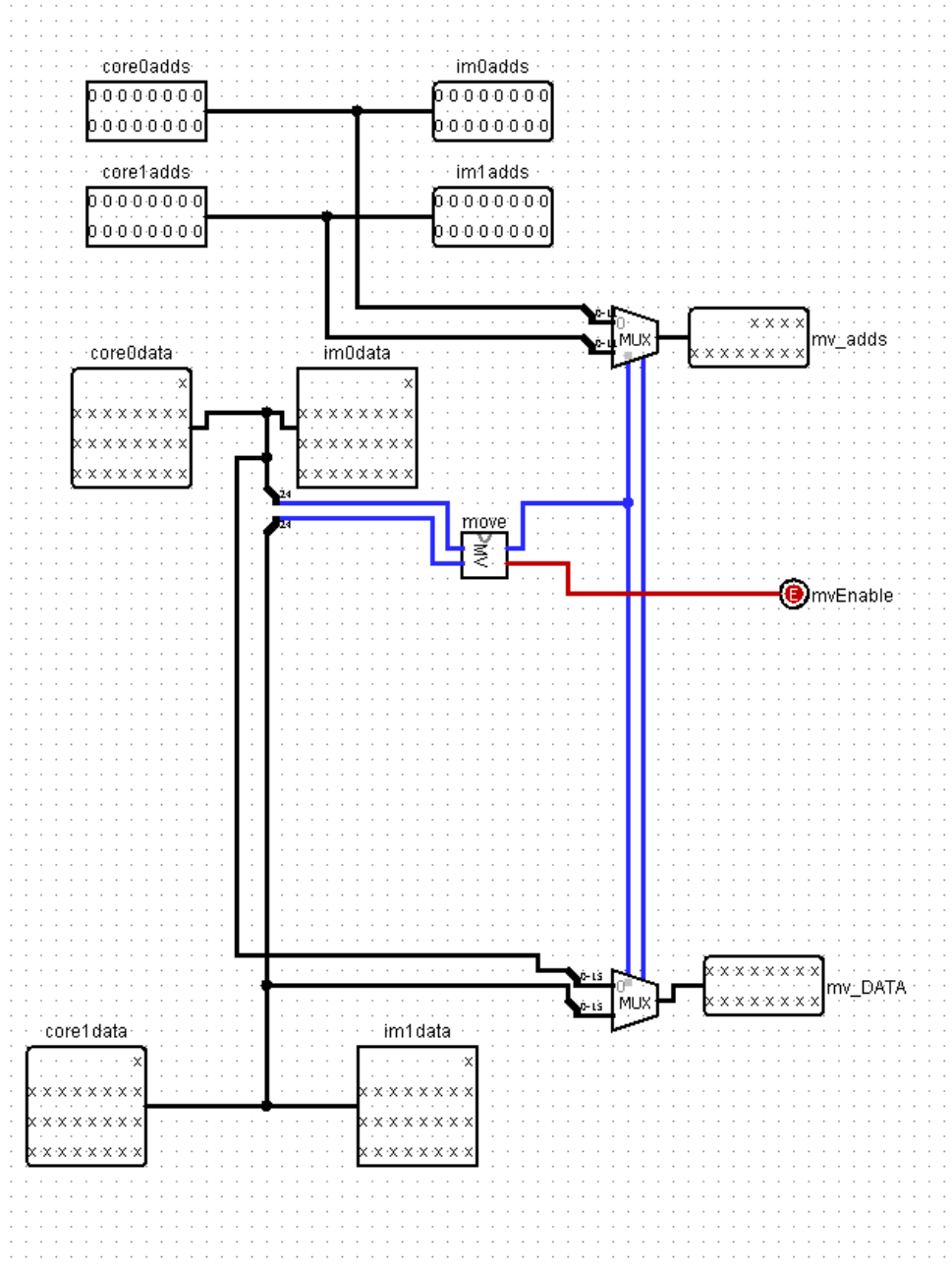


Figure 6: The CMU (Core Management Unit)

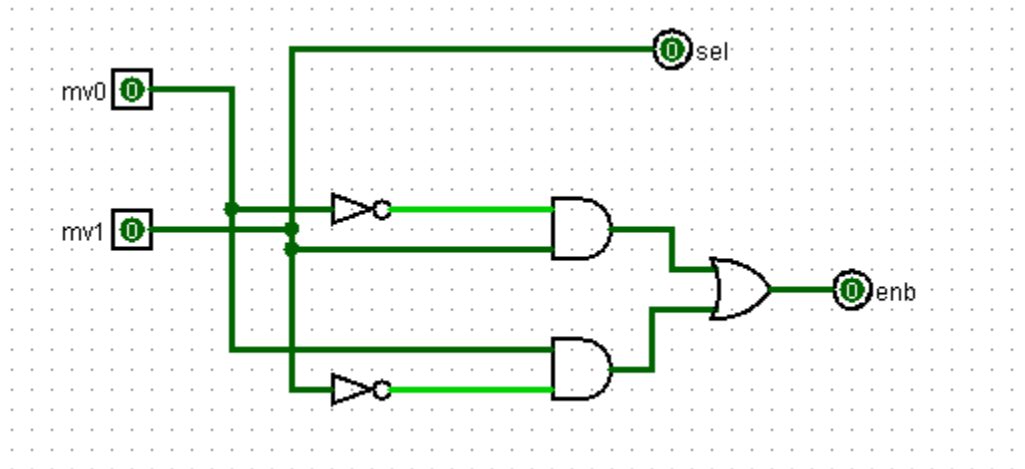


Figure 7: Move Control utilized inside the CMU

## Dual Core CPU – 1

This CPU utilizes the control logic “control” so it can read from both memories and utilize both cores albeit not being able to write into the RAM.

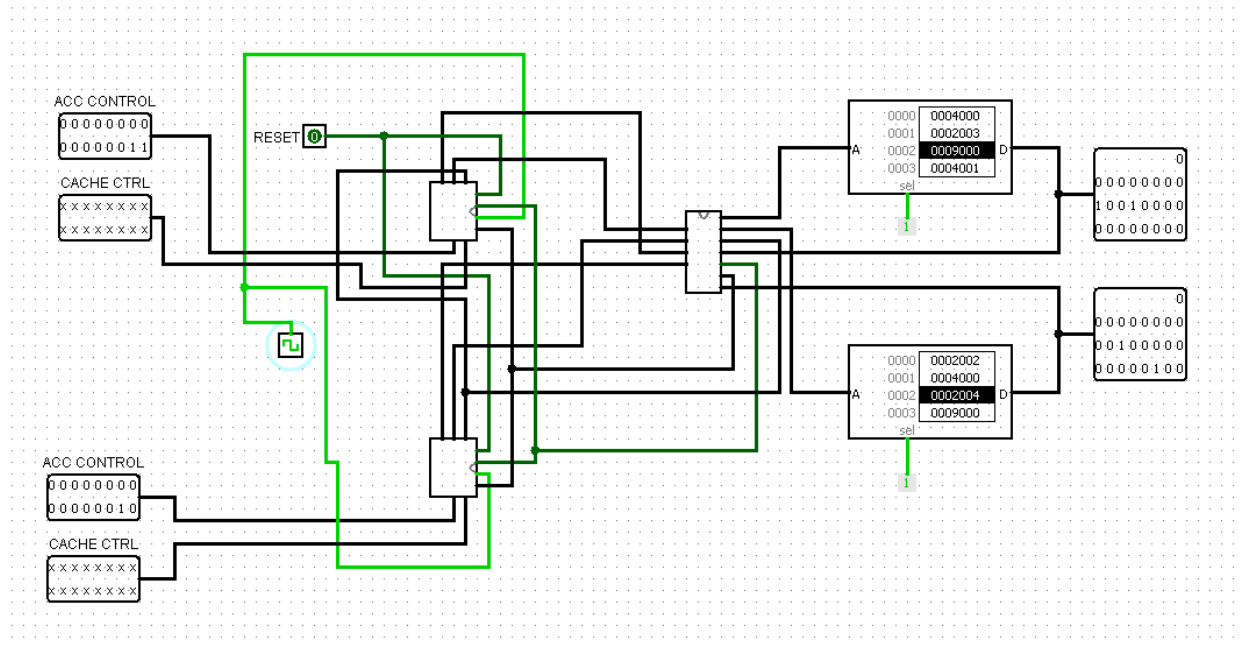


Figure 8: Dual Core CPU that utilizes “control”

## Dual Core CPU – 2

The CPU that utilizes “ff7controledition” and cannot utilize 2 cores. In this circuit there is an instruction skipping problem. When the up1 signal is hooked up to the “1” signal of the ACC, both cores work but they skip instructions. As broken as this is, we decided to put this into the final draft of the project too just to show that we managed to make the STR instruction work because this architecture **can** write to RAM.

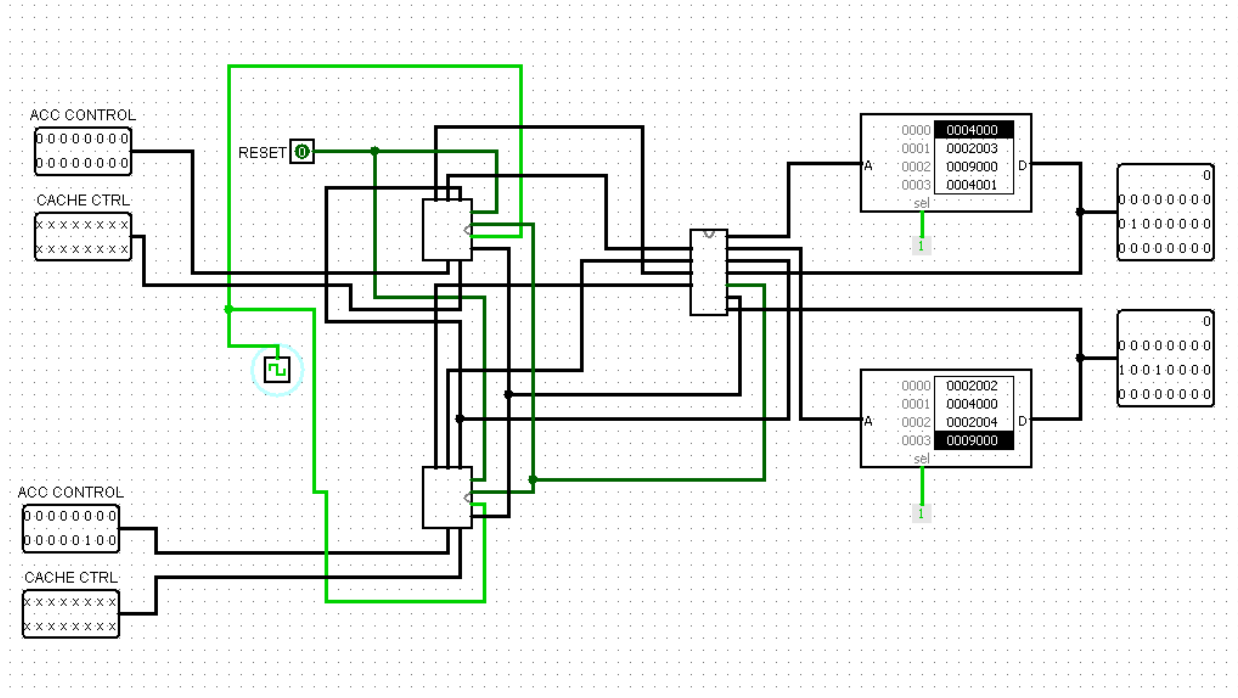


Figure 9: Dual Core CPU that utilizes “ff7controledition”

## The Assembler

The assembler for this project was written on C in Linux.

**For unknown reasons, this code only and only works on Linux operating systems.**

The code takes a text file containing the instructions and the arguments and translates them into machine code that can be used in the CPU to load into the ROM.

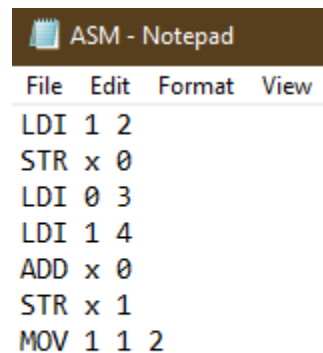
The code increments the int “i” to find the opcode.

File\_con chooses which CPU to relegate the work to.

Then the code generates two files called ASCII0.txt and ASCII1.txt.

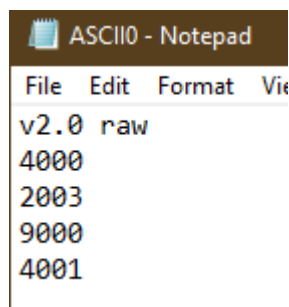
These text files can now be inserted into the CPUs ROM.

### Sample Input File:

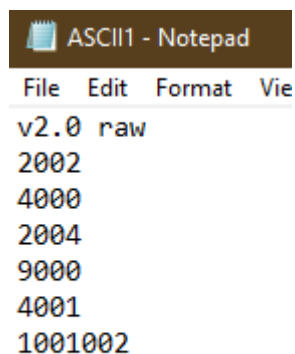


```
File Edit Format View
LDI 1 2
STR x 0
LDI 0 3
LDI 1 4
ADD x 0
STR x 1
MOV 1 1 2
```

### Sample Output File(s):



```
File Edit Format View
v2.0 raw
4000
2003
9000
4001
```



```
File Edit Format View
v2.0 raw
2002
4000
2004
9000
4001
1001002
```

## The Code:

```
main.c x
1  #include <stdio.h>
2  #include <string.h>
3  #include <ctype.h>
4
5  #define NEG_FILTER 0x00000FFF
6
7  char *op_codes_str[16] = {
8      "BRZ",
9      "BRN",
10     "LDI",
11     "LDM",
12     "STR",
13     "XOR",
14     "NOT",
15     "AND",
16     "ORR",
17     "ADD",
18     "SUB",
19     "MUL",
20     "DIV",
21     "NEG",
22     "LSL",
23     "LSR"
24 };
25
26 int main()
27 {
28     printf("COMP206 - Dual Core Assembler\n\n");
29     FILE* fin;
30     FILE* fcore0;
31     FILE* fcore1;
32     char buff[255];
33
34     fin = fopen("ASM.txt", "r"); //INPUT
35     fcore0 = fopen("ASCII0.txt", "w"); //OUTPUT FOR CORE0
36     fcore1 = fopen("ASCII1.txt", "w"); //OUTPUT FOR CORE1
37
38     fprintf(fcore0, "v2.0 raw\n");
39     fprintf(fcore1, "v2.0 raw\n");
40
41     printf("Reading assembly file as....\n\n");
42
43     while (fgets(buff, 255, (FILE*)fin)) {
44         int instruction = 0;
```

```

45     char op[3];
46     char file_con;
47     int y, x;
48
49     /*
50      * CMU 1 YYYYYYYYYYYY XXXXXXXXXXXXX
51      * CPU 0 00000000PPPP YYYYYYYYYYYY
52      * */
53
54     if (strstr(buff, "MOV")) {
55         sscanf(buff, "%s %s %d %d", op, &file_con, &y, &x);
56         printf("%s %c %d %d\n", op, file_con, y, x);
57
58         y &= NEG_FILTER;
59         x &= NEG_FILTER;
60
61         instruction = 0x1000000;
62         instruction |= y << 12 | x;
63
64         printf("\t %X\n", instruction);
65     } else {
66         sscanf(buff, "%s %s %d", op, &file_con, &y);
67         printf("%s %c %d\n", op, file_con, y);
68
69         int i = 0;
70         for (; i < 16; i++) {
71             if (strstr(op, op_codes_str[i])) {
72                 break;
73             }
74         }
75
76         y &= NEG_FILTER;
77
78         instruction = i << 12;
79         instruction |= y;
80
81         printf("\t %X\n", instruction);
82     }
83     //Core0
84     if (file_con == '0') {
85         fprintf(fcore0, "%X\n", instruction);
86     }

```



```

        printf("\t %X\n", instruction);
    }
    //Core0
    if (file_con == '0') {
        fprintf(fcore0, "%X\n", instruction);
    }

    //Core 1
    if (file_con == '1') {
        fprintf(fcore1, "%X\n", instruction);
    }

    //Operation is for Both Cores, write to both files.
    if (file_con == 'X' || file_con == 'x') {
        fprintf(fcore0, "%X\n", instruction);
        fprintf(fcore1, "%X\n", instruction);
    }
}
printf("\n\nASCII file for Core #0 created as...'ASCII0.txt'...\n");
printf("\nASCII file for Core #1 created as...'ASCII1.txt'...\n\n");

//Close file pointers
fclose(fin);
fclose(fcore0);
fclose(fcore1);

return 0;
}

```

## Work Acknowledgement

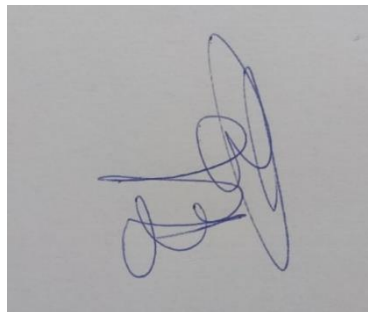
**Selçuk Coşkun:** Did the assembler, helped with CPU core, did the CMU, helped with Report, did the control logics.

**Tolga Kaan Tütün:** Helped with CPU core, helped with Report, helped with control logics.

Selçuk Coşkun

A handwritten signature in blue ink, consisting of several loops and a long horizontal stroke at the end.

Tolga Kaan Tütün

A handwritten signature in blue ink, featuring a large, stylized 'T' and 'K' with a long horizontal stroke at the bottom.