ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION PROJECT REPORT

PROJECT NO : 4

PROJECT DATE : 14.07.2020

GROUP NO : 31

GROUP MEMBERS:

150180001 : MEHMET ARİF DEMİRTAŞ

150180002 : BARIŞ EMRE MİŞE

150180007 : ÖMER FARUK ÖZKAN

150180009 : ERCE CAN BEKTÜRE

SPRING 2020

Contents

FRONT COVER

CONTENTS

1 INTRODUCTION **1**

2 PROJECT PARTS **1**

2.1 Control Logic 1

2.1.1 Branching Logic 2

2.1.2 Mapping Logic 2

2.2 Type-1 Micro-operations 5

2.2.1 READ 5

2.2.2 MREAD 5

2.2.3 STORE 6

2.2.4 BRANCH 6

2.2.5 FLAG 6

2.2.6 SETSTACK & GETSTACK 8

2.2.7 ALUA Operations: MOVM, NOTM, LSLM, LSRM 8

2.2.8 ALUAB Operations: ADDM, SUBM, ANDM, ORM 9

2.2.9 INCR & DECR 10

2.2.10 CHNGSTACK & STRSTACK 11

2.2.11 PCTIRH & PCTIRL 12

2.3 Type-2 Micro-operations 12

2.3.1 INCPC & INCSTACK & DECSTACK 12

3 RESULT **13**

3.1 Operations 14

3.2 Subroutines 16

3.2.1 FETCH 16

3.2.2 DIRECT 16

3.3 The Complete Microprogram 16

4 DISCUSSION **17**

5 CONCLUSION **17**

1 INTRODUCTION

In this fourth project, we implemented a software-based control unit for our ALU and register system from the second project. This design has 19 operations that are specified in the project PDF, and it has access to four numbered registers R_0, R_1, R_2, R_3 and registers SP, AR, PC. We also use 16 bit instructions register IR.

We used 24 micro-operations to realize these operations in our control unit, a ROM with 19-bit data and 8-bit address, with a total of 256 addresses, with three branching conditions and four branching types.

File name is as follows:

- **hw4.circ**

Also, these files are included from the first and second project:

- **hw2part2.circ**
- **hw2part1.circ**
- **hw1part1_8.circ**
- **hw1part2a.circ**
- **hw1part2b.circ**
- **hw1part2c.circ**

2 PROJECT PARTS

2.1 Control Logic

Our initial parts are similar in design, we divide the instruction into meaningful parts such as OPCODE, REGSEL and such. But we are using software-based control in this design, as opposed to the hardwired control of the previous project.

This means that instead of hard-coding the control signals for operations themselves, we sent control signals for micro-operations. Micro-operations are simple commands that are done in a single clock cycle. Since they take single clock, we do not need a sequence counter.

We implemented 24 micro-operations in total, and we divided them into 2 categories. There are 21 type-1 micro-operations, so we denote them with a 5-bit code. Most of our micro-operations are in this category. There are 3 type-2 instructions, so we denote them

with 2 bits. The micro-operations in this category are simpler and used with some of our other micro-operations simultaneously.

We used these micro-operations as building blocks in writing micro-instructions. Each micro-instruction of ours are 19-bit. The control word, which determines the control signals for the required micro-instructions, are 7-bit, as we have a 5-bit and a 2-bit category for micro-operations. Remaining 12-bit space is made of 2-bit CD: condition for branching, 2-bit BR: type of branching, and the 8-bit address. We decide where to go next in our control memory based on this 12-bit sequencing word.

Our micro-instructions make up the micro-program, which contains all the code for different operations. Micro-program is inside the Control Memory, which is a ROM with data width 19 and address width 8, which makes this a ROM with 256 address cells. Our address is held in a 8-bit register called CAR.

2.1.1 Branching Logic

We have 3 conditions for branching, unconditional branch (which is a condition that is always true), addressing mode branching, and zero flag branching. If condition is false, we move on to the next address in the control memory. If condition is true, we do the desired type of branching.

There are 4 branching types, which are JMP, CALL, RET and MAP. JMP jumps into the specified address. CALL also goes to the address, but it also stores the next address in subroutine register SBR. RET returns us to the address hold in the SBR. These two types can be used to emulate function calls. Finally, MAP does the mapping to find the correct address to go.

CD	**Condition**	**Symbol**
00	Unconditional	**U**
01	Addressing Mode	**I**
10	Zero flag	**Z**
11	Unconditional	**U**

BR	**Branch**
00	JMP
01	CALL
10	RET
11	MAP

Table 1: Branching logic and types

2.1.2 Mapping Logic

Since we have a 256-bit ROM, it has 8-bit addresses. Since our OPCODEs are 5-bit, we put two zeroes in the right and a one in the left. So our OPCODEs become addresses that start at 128(hex 80) and each opcode have 4 lines until the next opcodes. First 128 bits of the control memory are reserved for subroutines.

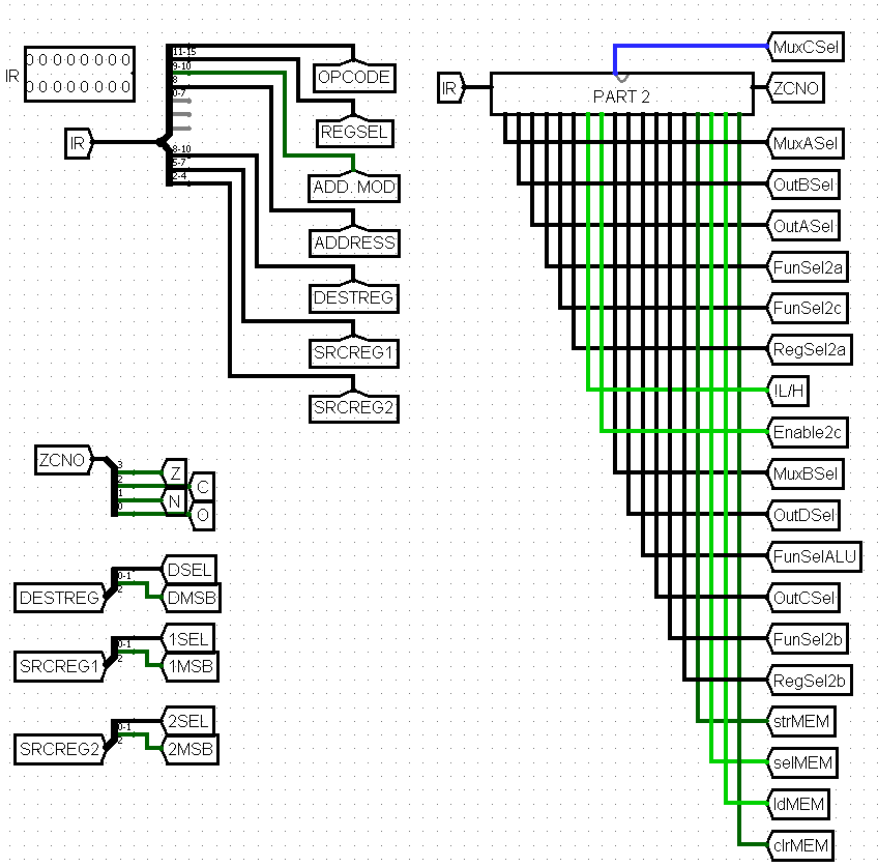

Figure 1: Initial setup for inputs-outputs

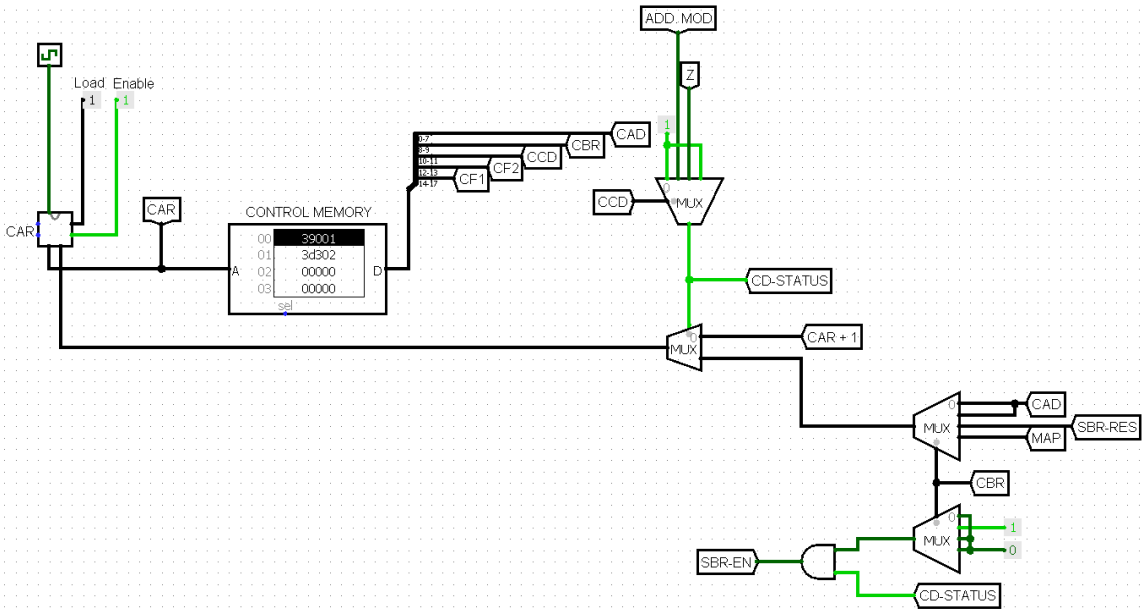

Figure 2: Control Memory and Address Designation Circuit

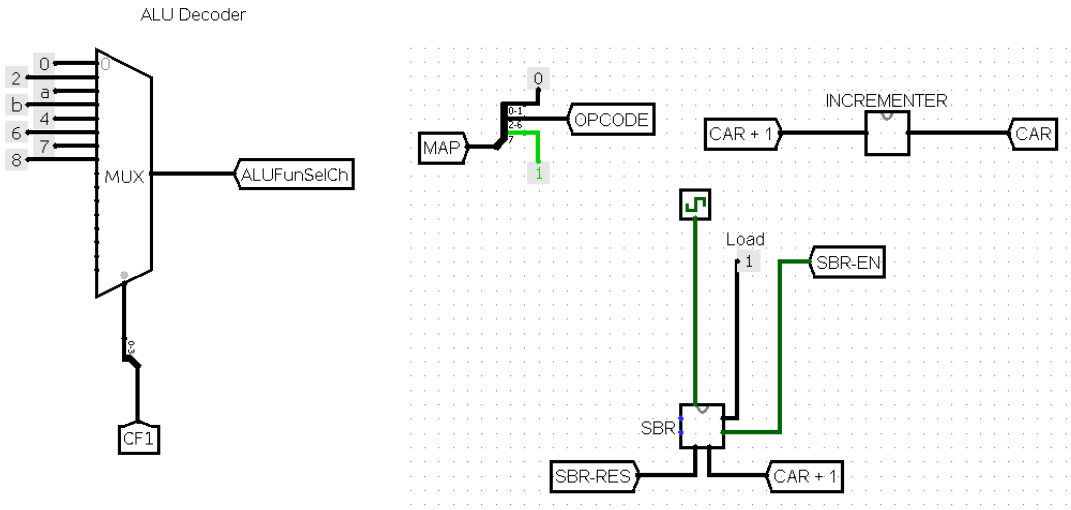

Figure 3: ALU Decoder, Mapping Logic, Incrementer, Subroutine Register

ALU Decoder is used to have multiple ALU microoperations that uses the same control signal. It will be explained in ALUA and ALUAB subsections.

μ-op	Code
NOP	00000
READ	00001
MREAD	00010
STORE	00011
BRANCH	00100
FLAG	00101
SETSTACK	00110
INCR	01001
DECR	01010
GETSTACK	01011
CHNGSTACK	01100
STRSTACK	01101
PCTIRH	01110
PCTIRL	01111

μ-op	Code
MOVM	10000
NOTM	10001
LSLM	10010
LSRM	10011
ADDM	10100
SUBM	10100
ANDM	10101
ORM	10110

μ-op	Code
NOP	00
INCPC	01
INCSTACK	10
DECSTACK	11

Table 2: Our microoperations for type-1(5-bit) and type-2(2-bit)

2.2 Type-1 Micro-operations

2.2.1 READ

This micro-operation reads the instructions rightmost 8-bits from instruction register into the register in part2a $Rx \leftarrow IR(0-7)$. The required register is selected by REGSEL.

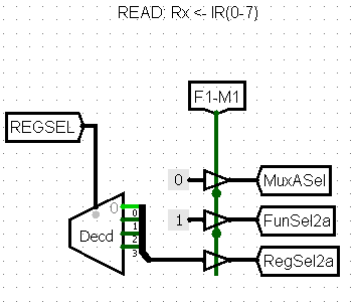

Figure 4: Circuit of READ micro-operation

2.2.2 MREAD

This micro operation is used for direct addressing, and it reads from memory to IR, from $M[AR]$ to $IR(0-7)$. For this micro-op we need to access the memory and get output of it therefore we set ldMEM and selMEM to 1.

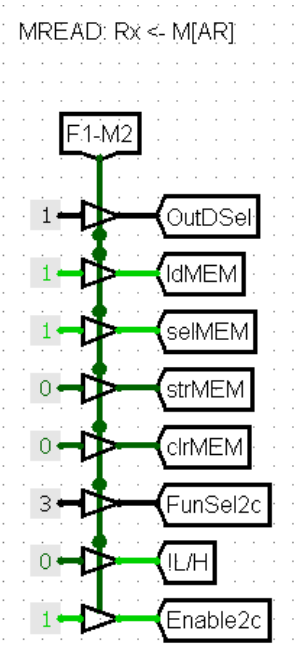

Figure 5: Circuit of MREAD micro-operation

2.2.3 STORE

In this micro-operation we read the value in Rx to $M[AR]$. For this one we need to access the memory but since we read input to the memory this time instead of ldMEM we set strMEM to 1.

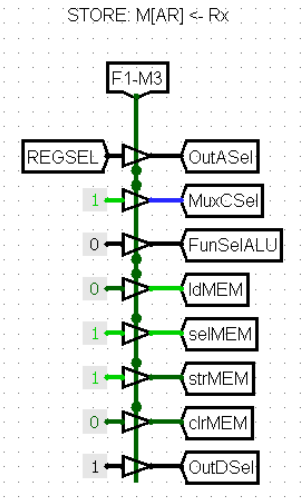

Figure 6: Circuit of STORE micro-operation

2.2.4 BRANCH

This is a rather short micro-operation, in this one we simply read the value into the PC register. Since this is an immediate operation we do not need to access the memory.

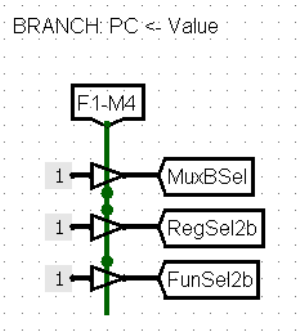

Figure 7: Circuit of BRANCH micro-operation

2.2.5 FLAG

This micro operation is designed to check the zero and negative flags. This is done by running the value through the ALU. This micro-operation is only used in increment and decrement instructions.

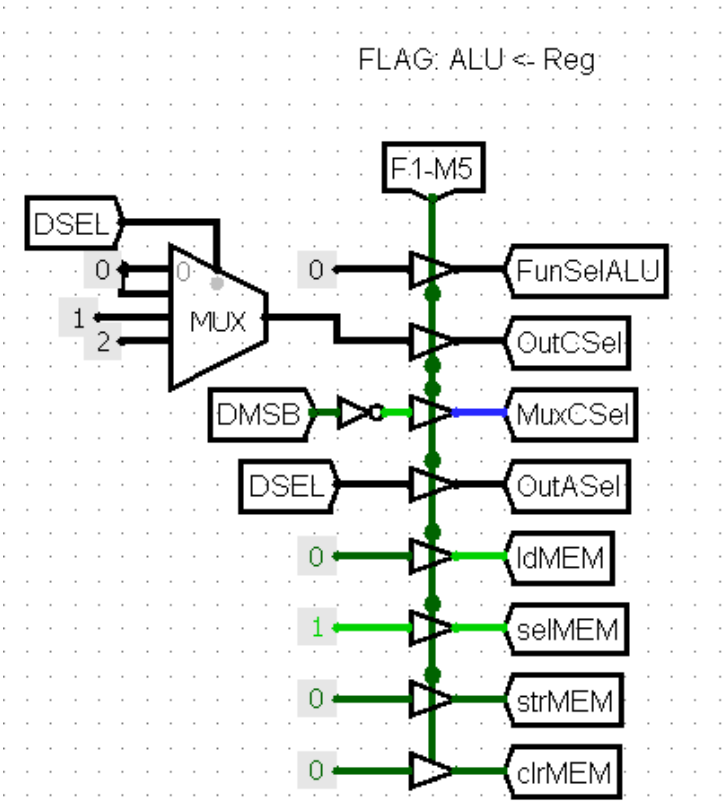

Figure 8: Circuit of FLAG micro-operation

2.2.6 SETSTACK & GETSTACK

These micro operations are opposites of each other. SETSTACK is $M[SP] \leftarrow PC$ and GETSTACK is $PC \leftarrow M[SP]$. These micro-operations are used in CAL(setstack) and RET (getstack) instructions. We store the current value of PC to the stack, we go to a subroutine and we return with RET, reading the top of the stack to PC.

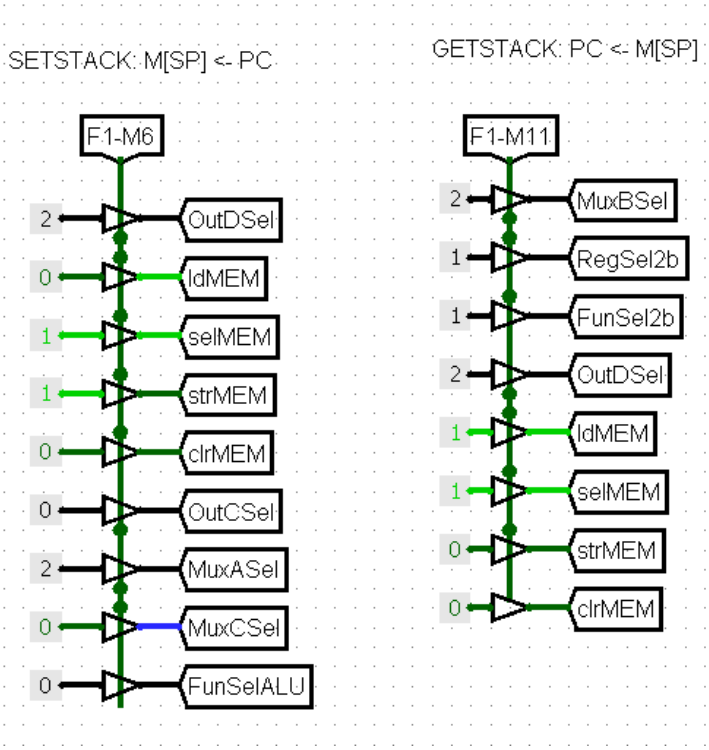

Figure 9: Circuits of SETSTACK and GETSTACK micro-operations

2.2.7 ALUA Operations: MOVM, NOTM, LSLM, LSRM

This micro-operation is used for basic ALU operations like LSL, LSR, NOT and another operation MOV. In this micro-operation we have 4 different moving states. These states are decided according to the DMSB and 1MSB, which are MSB of DESTREG and MSB of SRCREG1. The ALU operations are decided by the ALUFunSelCh, which is connected to a decoder which checks for which micro operation we selected. Our circuit takes a register, goes through the ALU and writes it into a third register, and what ALU does is determined by the ALUFunSelCh.

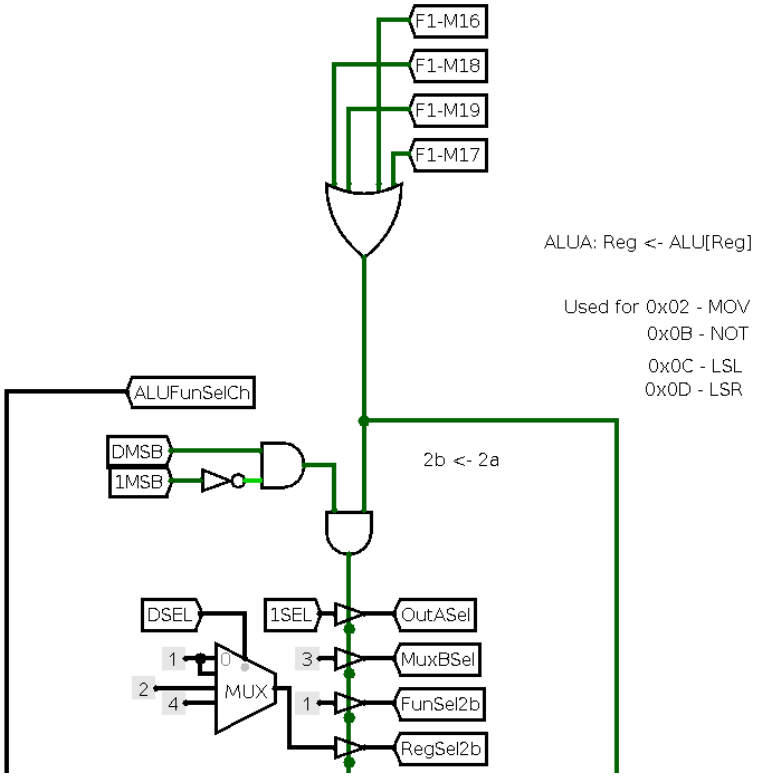

Figure 10: Circuit of ALUA micro-operations

2.2.8 ALUAB Operations: ADDM, SUBM, ANDM, ORM

In this micro operation we do the simple ALU operations with two operands like ADD,SUB, OR, AND. Just like in ALUA we have 4 different moving states for the output. To decide which operation we are doing we check ALUFunSelCh and to decide where we want to move the output we check 2MSB, 1MSB and DMSB.

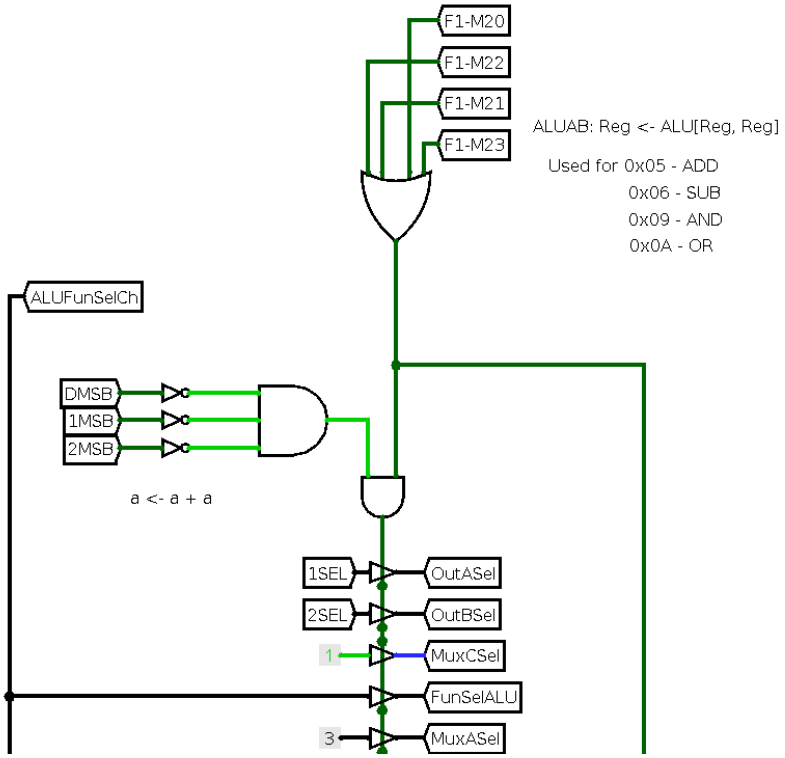

Figure 11: Circuit of ALUAB micro-operation

2.2.9 INCR & DECR

These two operations are quite simple to implement by DMSB we see if the register is part2a or part2b then according to that we do the incrementation or decrementation with funsel.

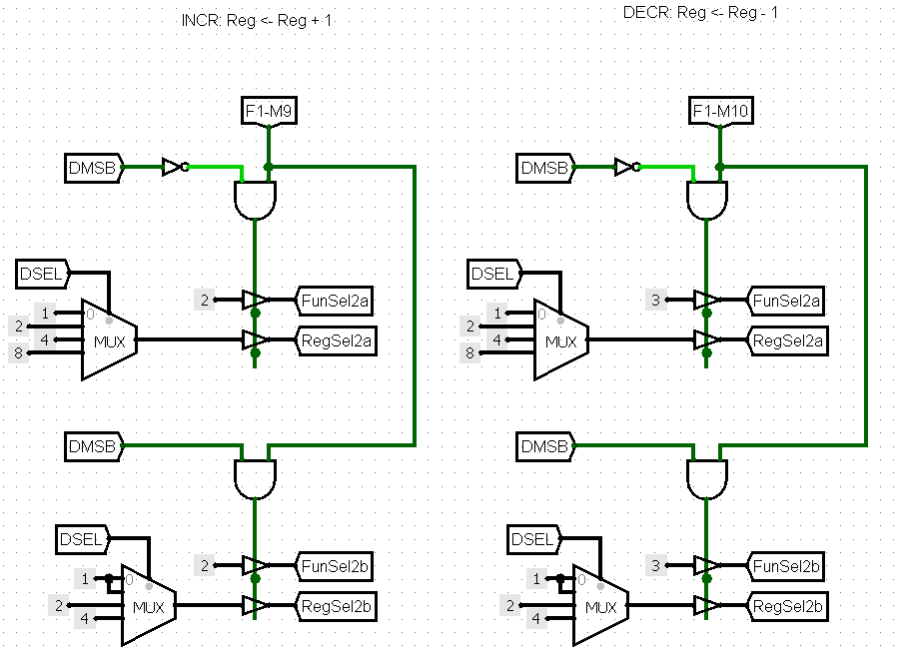

Figure 12: Circuits of INCR and DECR micro-operations

2.2.10 CHNGSTACK & STRSTACK

These micro-operations are used in PSH and PUL instructions which lets us push values to stack from registers in part2a or read values from stack to the registers in part2a.

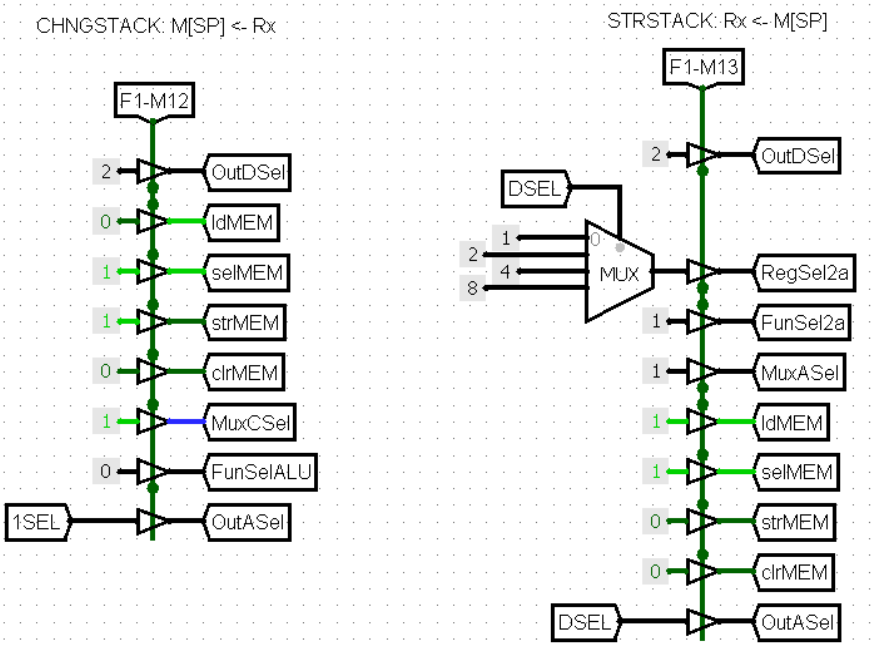

Figure 13: Circuits of CHNGSTACK and STRSTACK micro-operations

2.2.11 PCTIRH & PCTIRL

These micro-operations are used in fetch cycle. In these micro-ops we simply write the instructions from progtam counter (PC) into the instruction register.

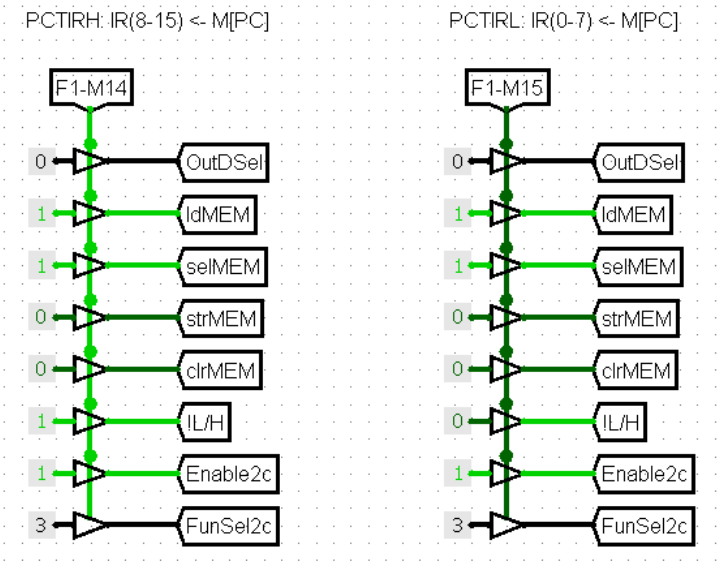

Figure 14: Circuits of PCTIRH and PCTIRL micro-operations

2.3 Type-2 Micro-operations

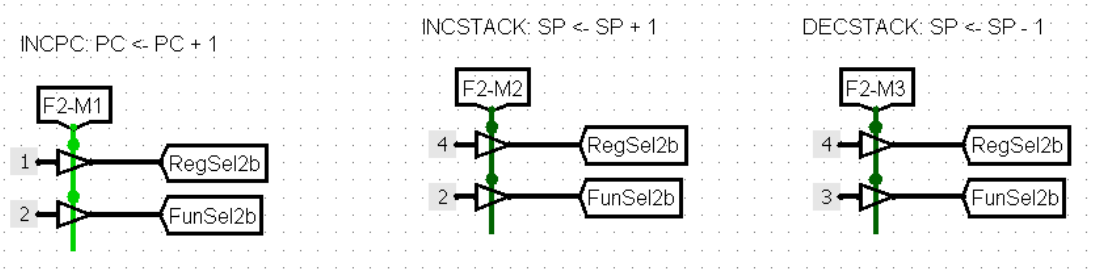

Figure 15: Circuits of INCPC, INCSTACK and DECSTACK micro-operations

2.3.1 INCPC & INCSTACK & DECSTACK

These micro-operations are Type-2 (F2) Micro-operations because some of these micro-operations' registers clash therefore they can not be Type-1. INCSTACK and DECSTACK operations are used in PSH and PUL instructions. When pushing to the stack we use DECSTACK and when pulling out of the stack we use INCSTACK. We use INCPC to jump to the next instruction in the fetch cycle.

3 RESULT

The following are our test cases from the third project. They still work without any problem.

Our first test case is the example program from PDF, which sums up from M[A0] to M[A4] and writes the output to M[A6]. PDF says result is written to M[A5], but to do this, we should delete the increment line, which is the instructions before the last one.

Listing 1: Example code in human readable form.

```
BRA PC IM 0x20
SKIP 15
LD R0 IM 0x05
LD R1 IM 0x00
LD R2 IM 0xA0
MOV AR R2
LD R2 D 0x00
INC AR AR
ADD R1 R1 R2
DEC R0 R0
BNE PC IM 0x28
INC AR AR
ST R1 D 0x00
```

Listing 2: Example code in hex logisim memory image form.

```
v2.0 raw
70 20 30*00 00 05 02 00 04 a0 16 40 05 00 46 c0 29 28 38 00 80 28 46 c0 0b 00
```

Our test case for bonus points, which subtracts R1 value from AR value and writes it to R3.

Listing 3: Example code in human readable form.

```
LD R0 IM 0x09
LD R1 IM 0x03
MOV AR R0
SUB R3 AR R1
```

Listing 4: Example code in hex logisim memory image form.

```
v2.0 raw
```

```
00 09 02 03 16 00 33 c4
```

Another example code we have written writes Fibonacci sequence to memory, starting from 0x30 address.

Listing 5: Example code in human readable form.

```
BRA PC D 0x10
SKIP 7
LD R0 IM 0x30
LD R1 IM 0x01
LD R2 IM 0x01
MOV AR R0
ADD R3 R1 R2
ST R3 D 0x00
INC AR AR
MOV R1 R2
MOV R2 R3
BRA PC D 0x18
```

Listing 6: Example code in hex logisim memory image form.

```
v2.0 raw
71 10 14*00 00 30 02 01 04 01 16 00 2b 28 0f 00 46 c0 11 40 12 60 71 18
```

3.1 Operations

We implemented all 19 operations in the following code. We start at address 128 and each operation is stored 4 addresses later.

Listing 7: Machine code for operations.

```
ORG 128
LOAD: NOP I CALL DIRECT
      READ U JMP FETCH
ORG 132
STORE: STORE U JMP FETCH
ORG 136
MV: MOVM U JMP FETCH
ORG 140
PSH: CHNGSTACK U JMP NEXT
     DECSTACK U JMP FETCH
```

```
11 ORG 144
12 PUL: INCSTACK U JMP NEXT
13      STRSTACK U JMP FETCH
14 ORG 148
15 ADD: ADDM U JMP FETCH
16 ORG 152
17 SUB: SUBM U JMP FETCH
18 ORG 156
19 DEC: DECR U JMP NEXT
20      FLAG U JMP FETCH
21 ORG 160
22 INC: INCR U JMP NEXT
23      FLAG U JMP FETCH
24 ORG 164
25 AND: ANDM U JMP FETCH
26 ORG 168
27 OR: ORM U JMP FETCH
28 ORG 172
29 NOT: NOTM U JMP FETCH
30 ORG 176
31 LSL: LSLM U JMP FETCH
32 ORG 180
33 LSR: LSRM U JMP FETCH
34 ORG 184
35 BRA: BRANCH U JMP FETCH
36 ORG 188
37 BEQ: NOP Z JMP BRA
38      NOP U JMP FETCH
39 ORG 192
40 BNE: NOP Z JMP FETCH
41      NOP U JMP BRA
42 ORG 196
43 CALL: SETSTACK, DECSTACK U JMP NEXT
44       BRANCH U JMP FETCH
45 ORG 200
46 RET: NOP, INCSTACK U JMP NEXT
47      GETSTACK U JMP FETCH
```

3.2 Subroutines

We implemented 2 subroutines, FETCH and DIRECT.

Listing 8: Subroutine declarations in machine code.

```
ORG 0
FETCH: PCTIRH, INCPC U JMP NEXT
       PCTIRL, INCPC U MAP EMPTY
ORG 4
DIRECT: MREAD U RET EMPTY
```

3.2.1 FETCH

FETCH subroutine takes two lines of micro-instructions and uses PCTIRL, PCTIRH and INCPC to fetch an instruction from the computer memory (RAM) and MAP to the corresponding OPCODE.

3.2.2 DIRECT

DIRECT subroutine uses MREAD micro-operation to read the direct value into the IR so in other micro-operations we can use it as an immediate value such as the LOAD operation.

3.3 The Complete Microprogram

Listing 9: The hex file for the whole microprogram.

```
v2.0 raw
39001 3d302 00000 00000 08205 00000 00000 00000 00000 00000 00000 00000 00000
    00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
    00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
    00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
    00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
    00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
    00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
    00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
    00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
    00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
    00000 00000 00000 00000 00000 00000 00000 00504 04000 00000 00000 0c000
    00000 00000 00000 40000 00000 00000 00000 3008d 00000 00000 00000 00091
    34000 00000 00000 50000 00000 00000 00000 54000 00000 00000 00000 2809d
```

```
14000 00000 00000 240a1 14000 00000 00000 58000 00000 00000 00000 5c000
00000 00000 00000 44000 00000 00000 00000 48000 00000 00000 00000 4c000
00000 00000 00000 10000 00000 00000 00000 008b8 00000 00000 00000 00800
000b8 00000 00000 1b0c5 10000 00000 00000 020c9 2c000
```

4 DISCUSSION

In this project, we completely implemented a basic CPU with software-based (micro-programmed) control unit. This project was mainly on control unit and control memory, and we used different Logisim gates and units together. We learned using tunnels to represent connections, and this project would be a lot more chaotic without tunnel usage and ROM usage. We also heavily used controlled buffers as we did in the third project. This was another big organisation and we designed it more carefully. Designing micro operations was easy but we spent time creating a well-crafted micro program, watching out for edge cases in programming. We believe we avoided such errors.

5 CONCLUSION

In conclusion of this project we have learned a lot about software side of a basic computer by implementing lots and lots of instructions on Logisim and gained many information about basic central processing units.