ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION PROJECT REPORT

PROJECT NO : 3

PROJECT DATE : 03.06.2020

GROUP NO : 31

GROUP MEMBERS:

150180001 : MEHMET ARİF DEMİRTAŞ

150180002 : BARIŞ EMRE MİŞE

150180009 : ERCE CAN BEKTÜRE

150180007 : ÖMER FARUK ÖZKAN

SPRING 2020

Contents

1 INTRODUCTION

In the third project, we implemented a control unit for our ALU and register system from the second project. This control unit uses 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.

File name is as follows:

- **hw3.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 Initial Operations

These operations are happening regardless of instruction. We decode OPCODE to meaningful parts, such as OPCODE, REGSEL, ADD. MOD which is addressing mode, ADDRESS, DESTREG, SRCREG1 and SRCREG2. We use a splitter to select relevant bits.

DESTREG, SRCREG1 and SRCREG2 are also decoded into smaller parts, such as DSEL and DMSB, or 1SEL, 1MSB, 2SEL, 2MSB. xMSB helps us understand if we are working on the register bank in part2a or part2b. xSEL is used to set the OutxSel of a register bank.

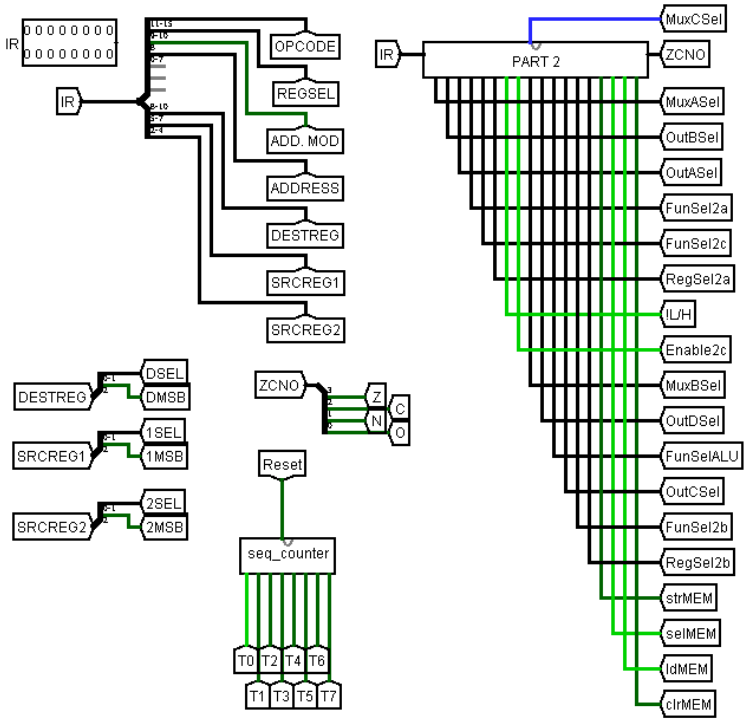
We have a seq_counter subcircuit, which is a modified 8-bit register that clears itself if reset equals 1, or increments itself if reset equals 0. Its output is connected to a 8-bit decoder, and the outputs of decoder is connected to tunnels from $T0$ to $T7$, which are our timing signals.

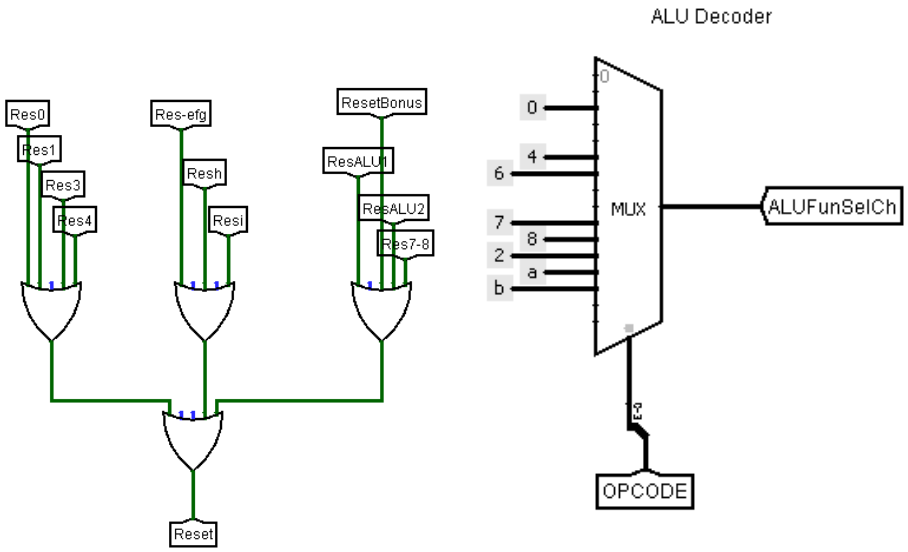
We have our PART 2 circuit, which we extended all inputs and outputs to tunnels.

We have our fetch decode cycle and then our OPCODE decoder. We use our decoder to create signals $D0, D1, \ldots$ up to Dh. There is a tunnel for each of the operations. If D_i is 1, all other D_j are zero and we are doing i_{th} operation.

Then we have our reset logic, which uses OR to combine different reset signals from all over our circuit and makes Reset of the sequence counter 1 if any of the resets are 1.

Finally, we have an ALU decoder, which takes OPCODE and outputs ALUFunSelCh - a choice for the ALU. We did this since there are multiple operations such as ADD, SUB, AND etc. that does the same operations but with different ALU FunSel. This part uses a MUX that sends constants to its outputs.





2.2 Fetch Decode Cycle

Our fetch-decode cycle takes up 3 timing signals, $T0 - T2$. Our instructions start at $T3$. We also considered making it two timing signals, but we have faced some errors.

In $T0$, we go to the memory location using the address in PC, we copy it to IR(15-8) and we increment PC. In $T1$, we do the same but copy the value into IR(7-0). In $T2$, we send in a clock with enable of IR set to 0.

After this cycle, we start running our instructions. Our instructions divided into two categories, instructions with address references, which have addressing mode and address. Other category are without address references, which have register selections such as DESTREG, SRCREG1, SRCREG2.

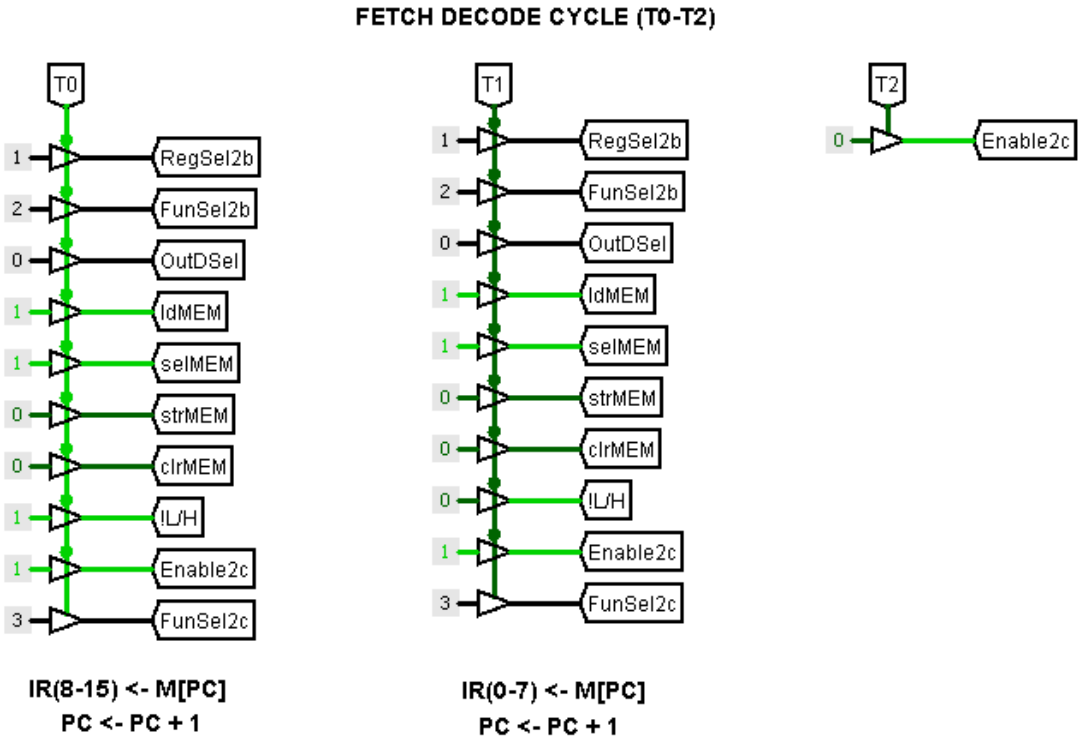


Figure 1: Circuit of Fetch and Decode

2.3 Instructions with address references

2.3.1 LD(0x00)

In this operation if the addressing mode (8th bit) is 0 we directly write the value ,that is the first 8 bits, into the desired register which was given in the instructions bits 9-10 . If the addressing mode is 1 then we need to access to the memory location of the address register and write the value at that location of the memory into the desired register.

This operation is made in 2 time signals.

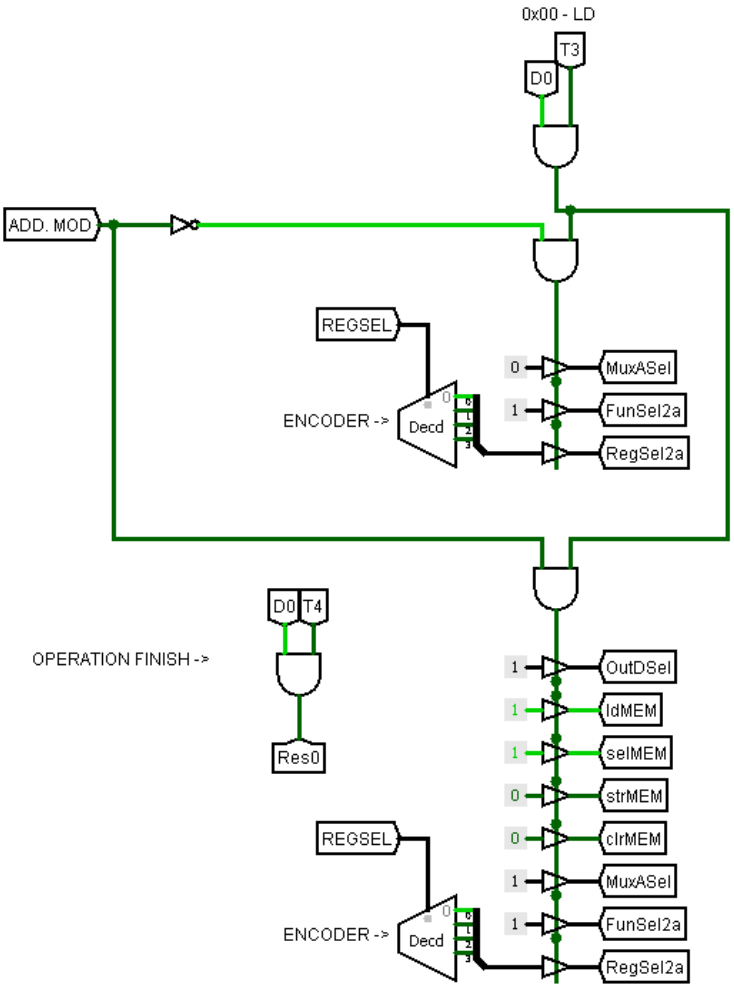


Figure 2: Circuit of LD operation

2.3.2 ST(0x01)

This operation is an address referenced operation but it only has direct addressing mode ,meaning that the 8th bit of the instruction is 1. In this operation we need to write the value of desired register which is given in REGSEL into the memory location of AR.

This operation is made in 2 time signals.

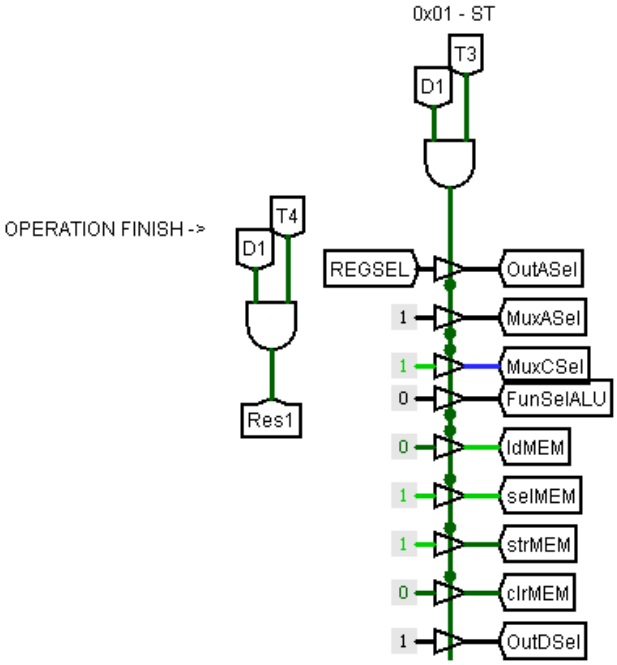


Figure 3: Circuit of ST operation

2.3.3 BRA(0x0E),BEQ(0x0F),BNE(0x10)

These operations are address referenced operations yet they only have immediate addressing mode ,meaning that the 8th bit of the instructions is 0. These 3 operations are quite similar the way they are implemented.Since these operations are with immediate addressing mode, we do not need to access the memory. In this operation we just write first 8 bits of the instruction into the PC in part2b via MuxBSel. The other operations are done almost exactly the same.

BEQ operation is executed if Zero flag in hw2part2.circ is 1. After checking the Zero (Z) flag, rest of the BEQ operation is the same as BRA operation ($PC \leftarrow Value$) .

BNE operation is executed if Zero flag in hw2part2.circ is 0. After checking the Zero (Z) flag, rest of the BNE operation is the same as BRA operation ($PC \leftarrow Value$).

This operation is made in 2 time signals.

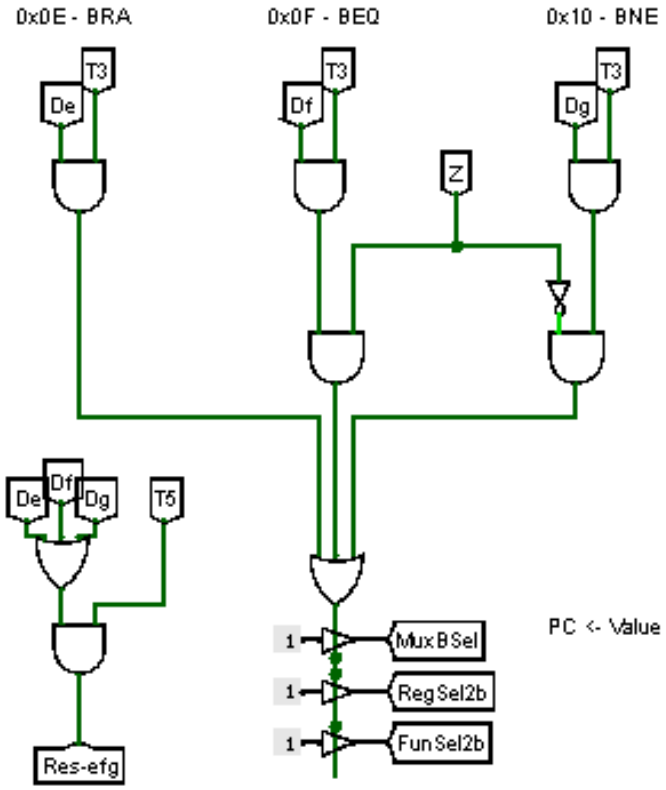


Figure 4: Circuits of BRA, BEQ and BNE operations

2.4 Instructions without address references

2.4.1 MOV(0x02),NOT(0x0B),LSL(0x0C) and LSR(0x0D)

These four operations are all similar, because they move a register's value through ALU to another register. So we implemented a circuit that checks DMSB and 1MSB, so we understand if we are moving the value from part2a to part2a, or part2b to part2b, or part2a to part2b. We set required FunSel and output choice inputs and we set ALU FunSel to ALUFunSelCh. So the value is moved with ALU = 0000 for MOV, 0010 for NOT, 1010 for LSL and 1011 for LSR.

Note that only MOV works when moving from 2b to 2a, since other variants of this operation is not considered valid according to project descriptions.

This operation is made in 2 time signals.

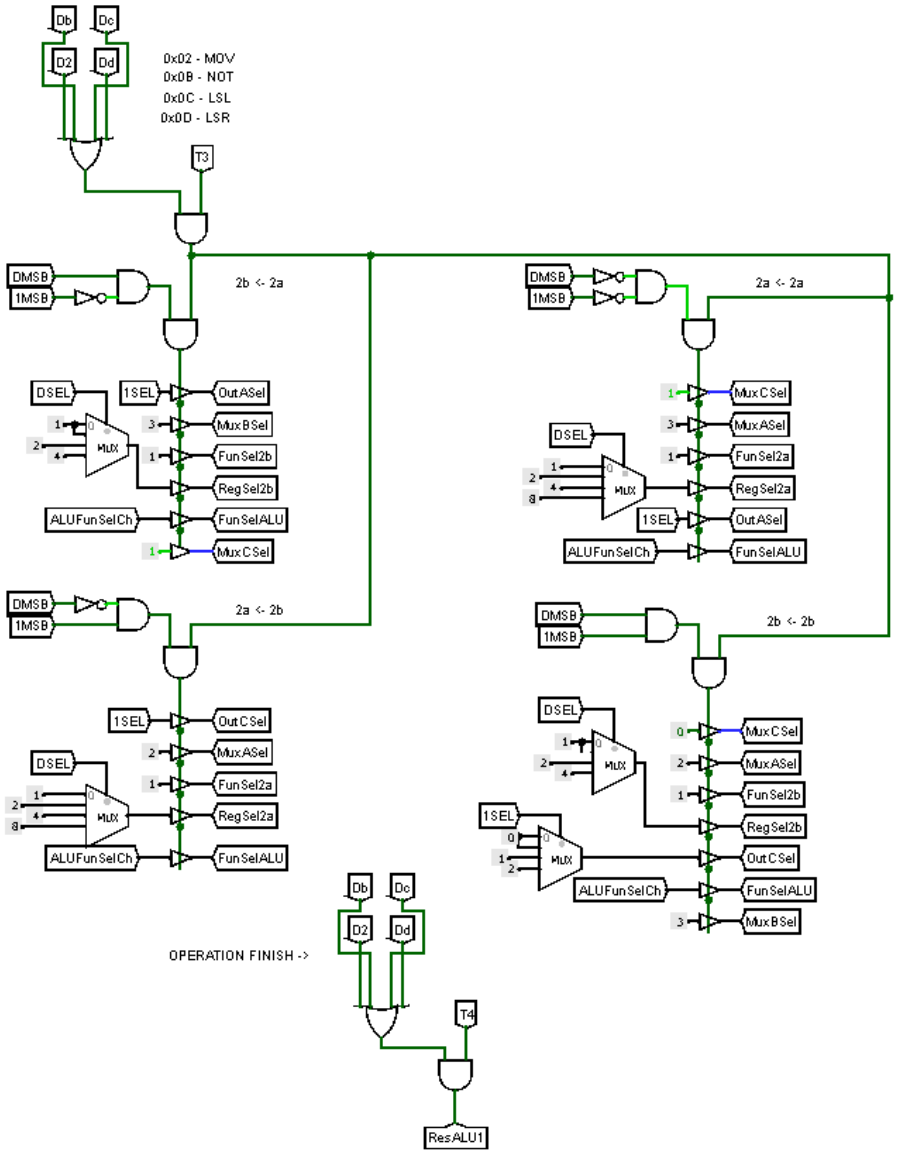


Figure 5: Circuits of MOV, NOT, LSL and LSR operations

2.4.2 ADD(0x05),SUB(0x06),AND(0x09) and OR(0x0A)

These are another set of four relevant operations. They are all operations where we take two operands from two registers, and after going through ALU, we store the output at another register. So we choose ALU FunSel using ALUFunSelCh and we check which way we are copying, such as $2a \leftarrow 2a + 2a$, $2b \leftarrow 2a - 2a$, $2b \leftarrow 2b \wedge 2a$. These are made in two time signals. We also have $2a \leftarrow 2b \vee 2a$, which we implemented for bonus points. This is made in 3 time signals, since we push the value to the stack and pull it from there.

This operation is made in 2 time signals.

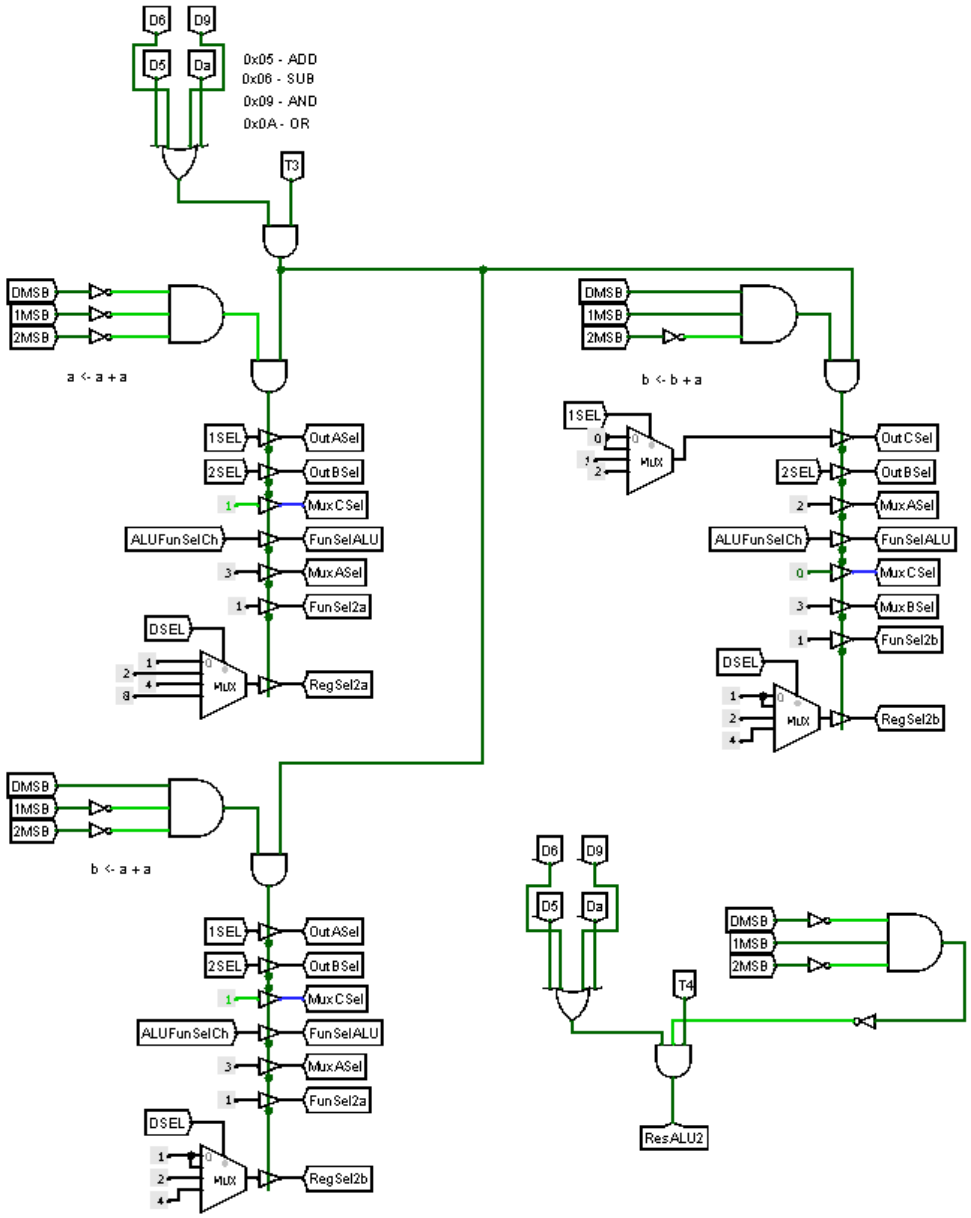


Figure 6: Circuits of ADD, SUB, AND and OR operations

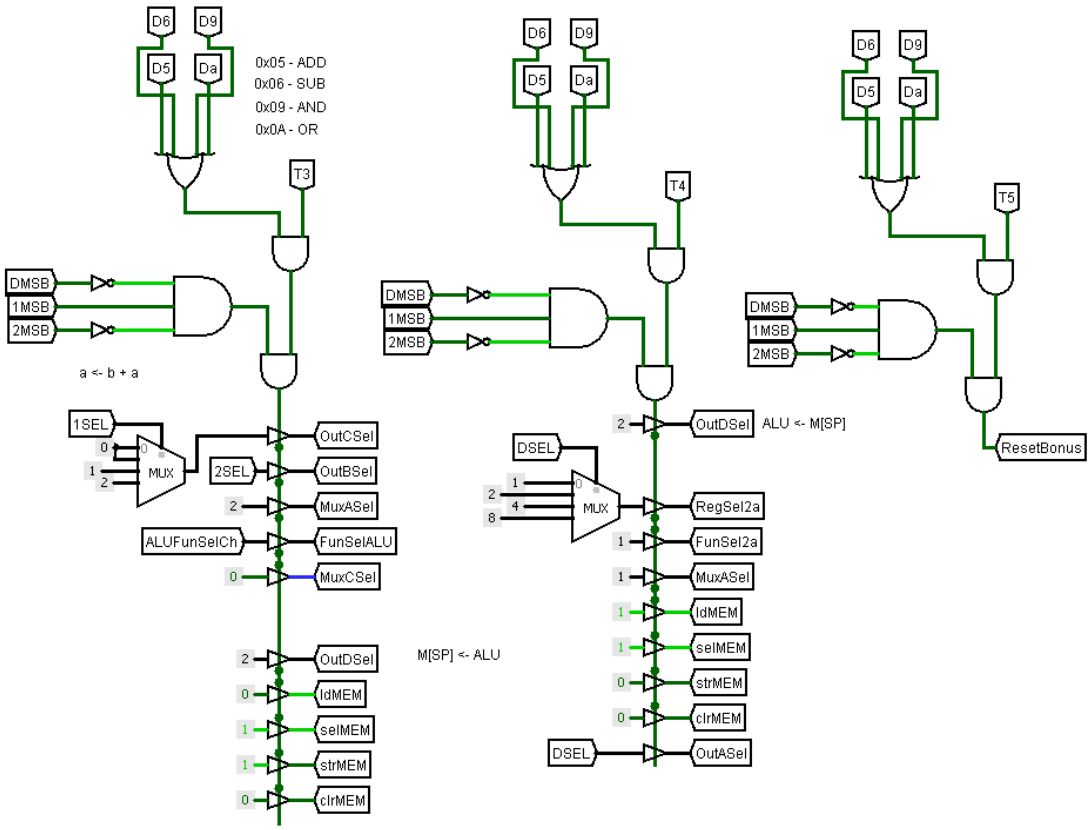


Figure 7: $2a \leftarrow 2b + 2a$, bonus implementation.

2.4.3 DEC(0x07) and INC(0x08)

These operations are also quite similar the way they are implemented since the only difference between decrement and increment is only different in Funsel2b or Funsel2a. For these operations we implemented a circuit checking the type of destination register with DMSB and source register with 1MSB in order to understand which operation we are doing such as $2b \leftarrow 2a$, $2b \leftarrow 2b$, $2a \leftarrow 2b$, $2a \leftarrow 2a$.Firstly we check if the DESTREG and SRCREG1 is the same, if that is the case we do not move the value and get straight to the increment/decrement operation else we first move the value to the desired register in one clock signal. After that we do the increment and decrement operations on the destination register in one clock cycle ,then in order to update the zero flag we simply run the incremented/decremented value in the destination register through the ALU in another clock cycle.

This operation is made in 3 time signals if DESTREG is SRCREG1, or 4 time signals if they are different.

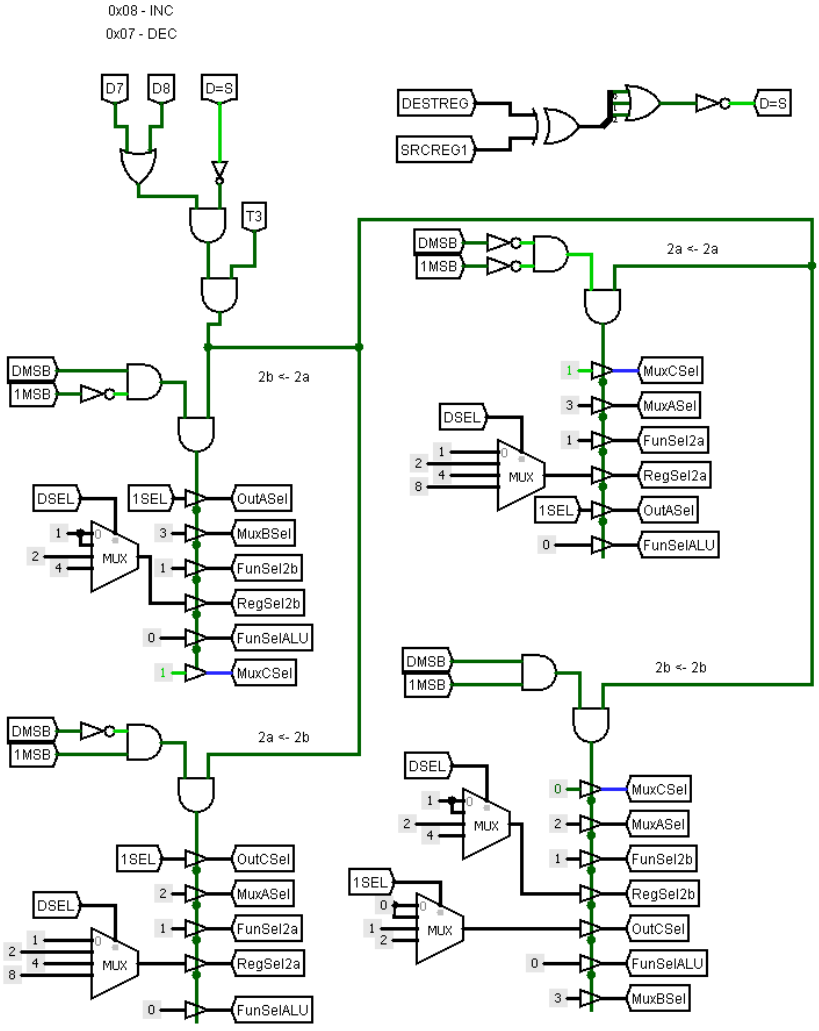


Figure 8: Control circuit of different DESTREG and SRCREG case

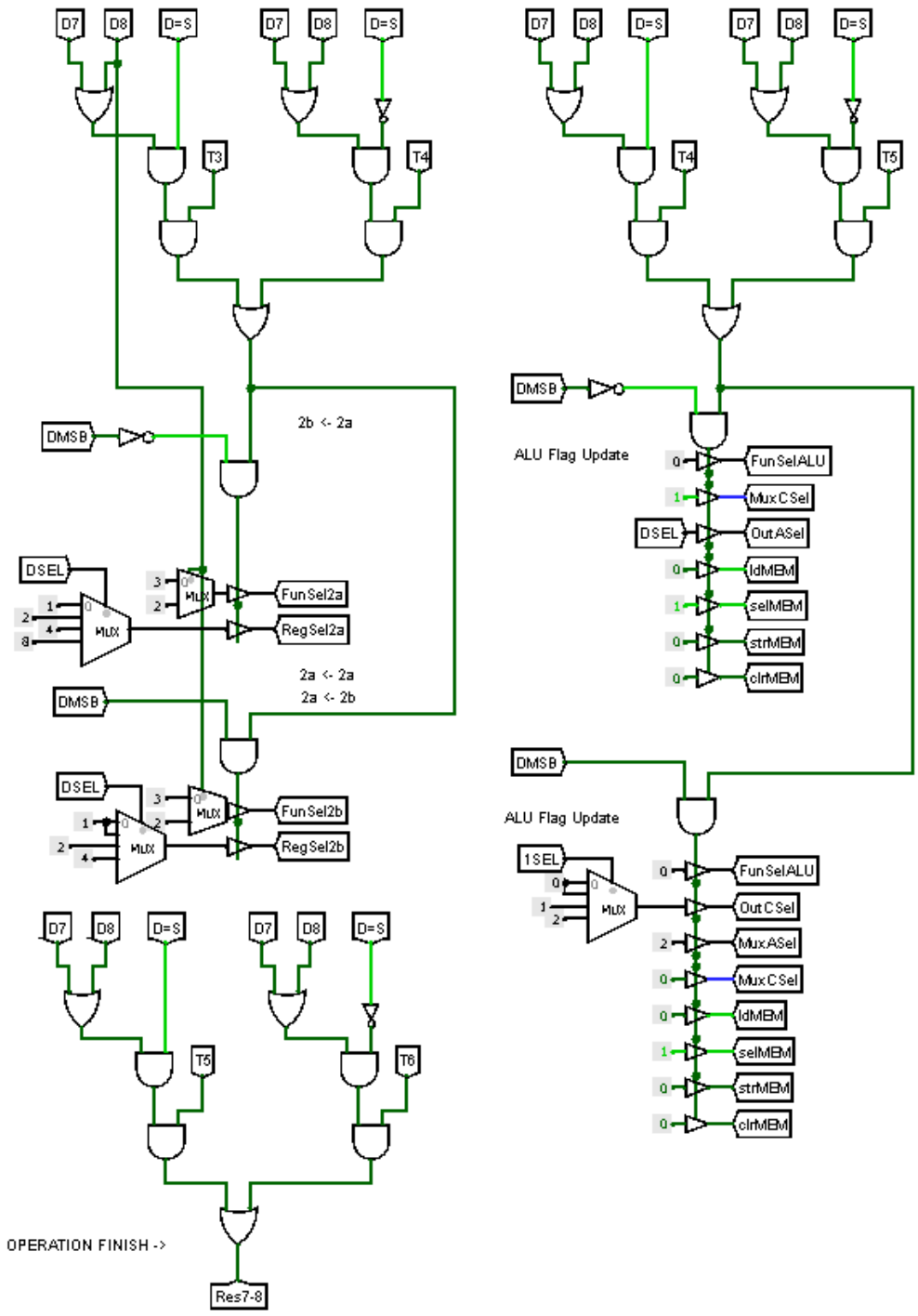


Figure 9: Direct increment/decrement operation and flag update

2.5 Instructions that store values in memory

2.5.1 PSH(0x03) and PUL(0x04)

These operations act like actual stack operations in data structures. In push operation we first write the value in a register in part2a in the memory location of SP (stack pointer) then decrement the value of SP.
PUL operation acts like stacks pop operation. In this operation we increment the value in SP then locate the slot of memory SP points and write the value in there to the location of the memory where the desired register points $Rx \leftarrow M[SP]$.

PSH operation is made in 3 time signals, PUL operation is made in 4 time signals.

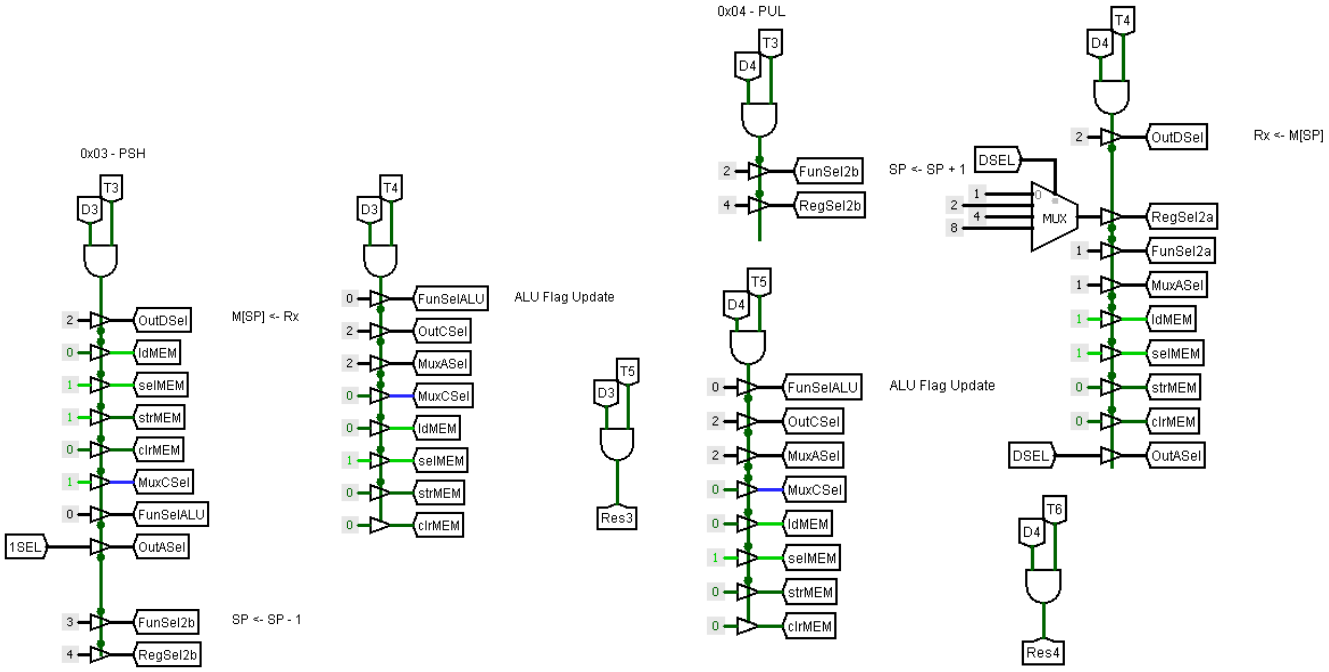


Figure 10: Circuits of PSH and PUL operations

2.5.2 CALL(0x11) and RET(0x12)

These instructions are like PSH and PUL, but they work with PC. Instead of a Rx value, these operations store the value of PC to M[SP] (for CALL), and store the value of M[SP] to PC (for RET). They increment or decrement SP accordingly. CALL also uses an address reference to copy a given value to PC, different than PSH.

CALL operation is made in 3 time signals, RET operation is made in 4 time signals.

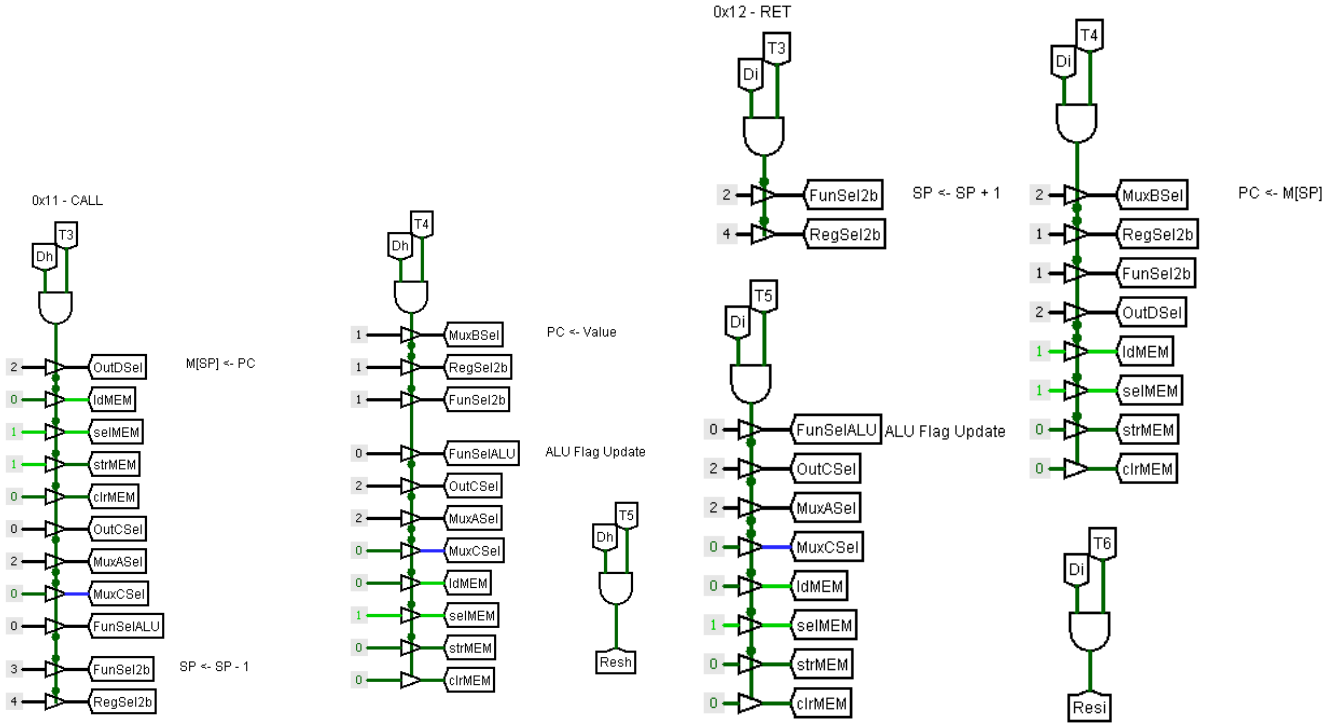


Figure 11: Circuits of CALL and RET operations

3 RESULT

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
LD R2 IM 0x10
MOV SP R2
MOV AR R0
SUB R3 AR R1
```

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

```
v2.0 raw
00 09 02 03 04 10 17 40 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
```

4 DISCUSSION

While designing this project, we completely implemented a basic CPU. In consecutive parts, we implemented registers, ALU and a control unit. This project was mainly on control unit, 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. We also heavily used controlled buffers to send in predetermined constant inputs to our part 2 circuit. This was a big organisation and we designed it carefully, watching out for edge cases in implementation. We believe we avoided such errors.

5 CONCLUSION

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