ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION

PROJECT 3 REPORT

GROUP NO : 1

GROUP MEMBERS:

150170095	:	Mehmet ALTUNER
150170026	:	Berkay OLGUN
150170103	:	Atahan ÖZER
150170076	:	Ekrem UĞUR

SPRING 2020

Contents

1 Introduction

As the third and the final project of the computer organization class, we implemented a very primitive computer like structure in logisim. This project is based on the second project, it is the automation of what we could do manually before. It is able to execute instructions that cover 18 different operations from memory. It can make basic arithmetic operations, function calls and more.

To keep track of time, we used a sequence counter, we would be reading the instruction from memory through T0 to T2, and when our operation was finished, we would re-set the counter to zero every time.

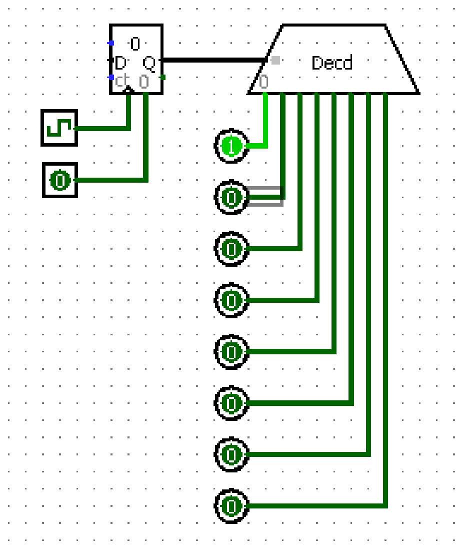

Figure 1: Sequence counter circuit

2 Fetch & Decode

To begin the process of executing instructions, first we needed to read the instruction from memory and put a meaning to it. We managed to load the 16 bit instruction to the IR register in two clock cycles. In the third clock cycle we disabled the IR register and finished the fetch phase of the execution.

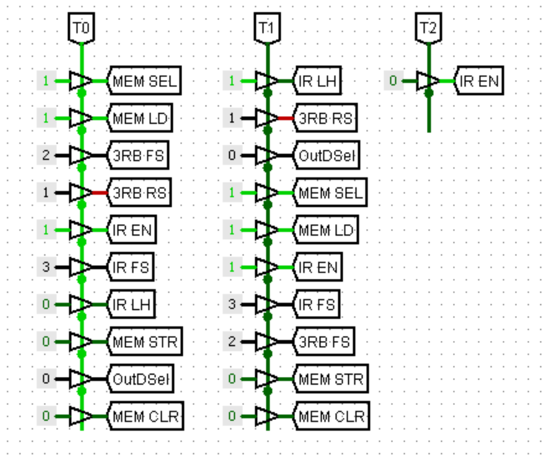

Figure 2: Organization setup for fetching an instruction

Then using splitters and decoders, we decoded the instruction so that only the required operations would be enabled and executed.

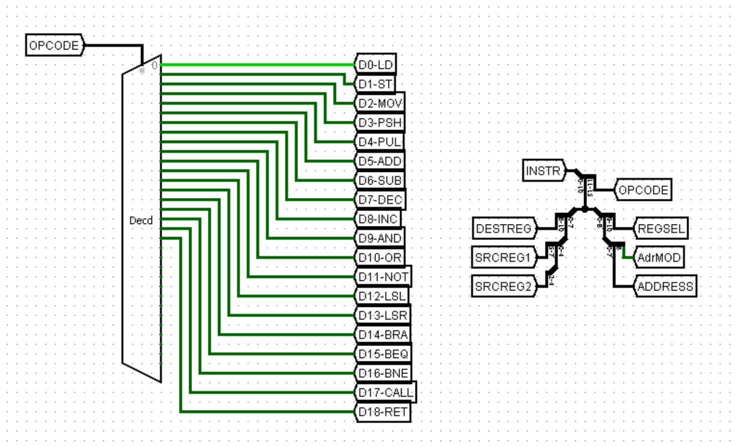

Figure 3: Decoding the instruction

At this point, we were ready to apply various commands, so we did exactly that.

3 Operations

3.1 LD

The operation LD writes a certain value, determined by address mode, to a register that has been selected via two RegSel bits. RegSel selects the register among RF registers. There are two possible outcomes according to address mode.

- RF ↤ M[AR] if address mode is set to 1, D
- RF ↤ ADDRESS Field if address mode is set to 0, IM

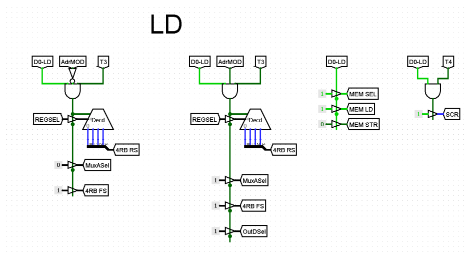

Figure 4: LD Operation Implementation

3.2 ST

The operation ST stores the value of a register selected by the 2-bit RegSel into the memory pointed by AR register. ST operation only accepts address mode D.

- M[AR] ↤ RF

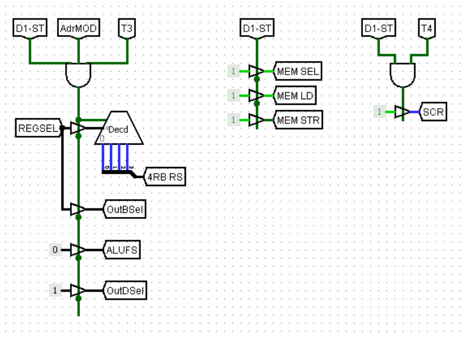

Figure 5: ST Operation Implementation

Note that address mode is always 1, D for the operation ST.

3.3 MOV

The operation MOV moves a value from a register to another register. Registers are selected via 3-bit destreg and 3-bit srcreg1. Destreg is the destination register in which the data in Srcreg1, source register, will be written at the end of the operation. There are four possible outcomes depending to the which type of registers are chosen.

- RF ↤ ARF, if last bit of the destreg is 0 and last bit of the srcreg1 is 1
- ARF ↤ ARF, if last bit of the destreg and last bit of the srcreg1 is 1
- ARF ↤ RF, if last bit of the destreg is 1 and last bit of the srcreg1 is 0
- RF ↤ RF, if last bit of the destreg last bit of the srcreg1 is 0

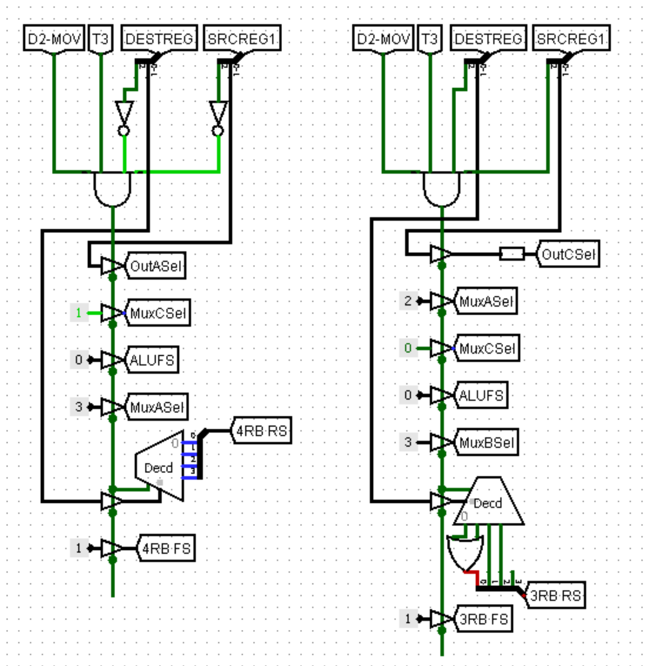

Figure 6: MOV Operation Implementation. Left circuit is RF ↤ RF and right circuit is ARF ↤ ARF

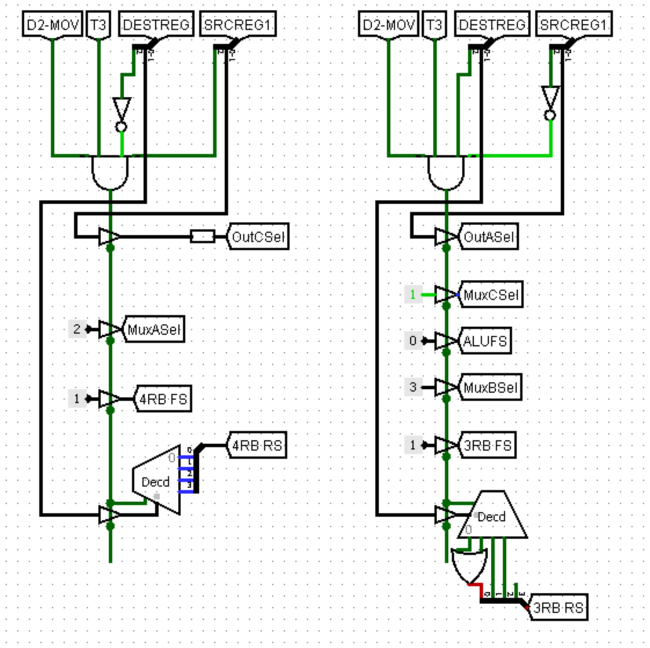

Figure 7: MOV Operation Implementation. Left circuit is RF ↤ ARF and right circuit is ARF ↤ RF

3.4 PSH

The operation of pushing data to the stack (PSH) is paired with the opcode 0x03 and the addressing mode is not aplicable to this operation. Meaning this operation is one of the operations that are type two. In our implementation we used DESTREG (bits from 5 to 8) as an input to this operation. To achieve this function, we first let the value of register SP reach to the address input of the logisim's RAM module, enabled the store, select and load flags of the said module. Then depending on the first bit of the DESTREG input, we either let the data from ARF to reach data input of the module through output C of the address register file, or we allowed the value in register file to go through ALU (which is set to pass the data as it is) and finally we stored the given data in the memory address that SP points to. When the storing is finished, we proceed to increment the value in the register SP via ARF function selector signal and complete the operation.

- M[SP] ↤ Rx
- SP ↤ SP - 1

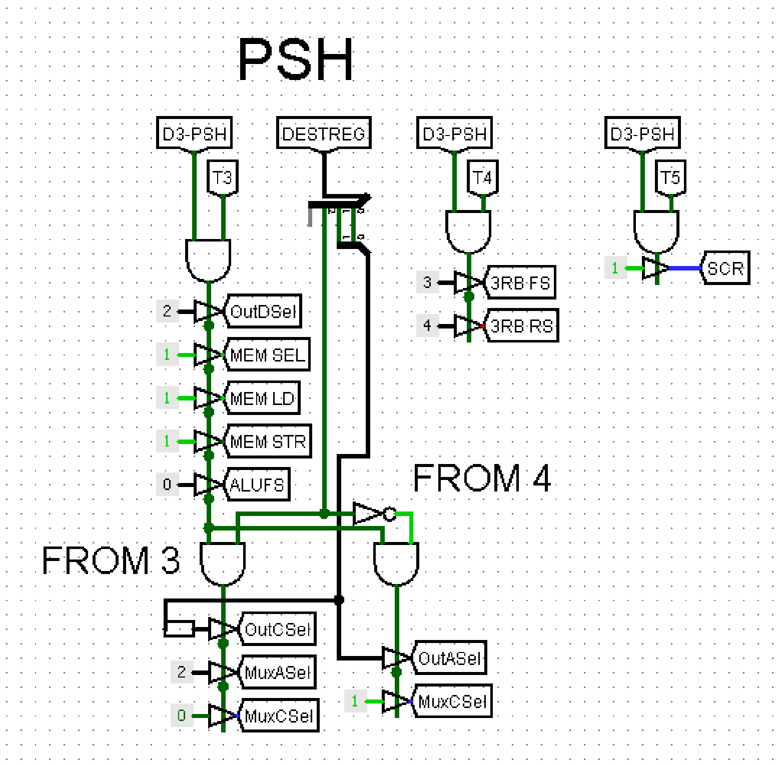

Figure 8: PSH Operation Clock Cycle Configurations

3.5 PUL

PUL operation is paired with the opcode 0x04 and it is again an operation that addressing was not applicable. This operation is just like the pushing operation, the difference between them being the other one is used for pushing and this one is for retrieving the data that was pushed before. To do such an operation, we first allow the memory module to select a memory cell and load the value in it to the output channel. Consequently we increment the value in register SP so that it points to the latest value that was used in a PSH operation. Then we channeled the output of the memory module to the DESTREG that we intend to load with that value.

- SP ↤ SP + 1
- Rx ↤ M[SP]

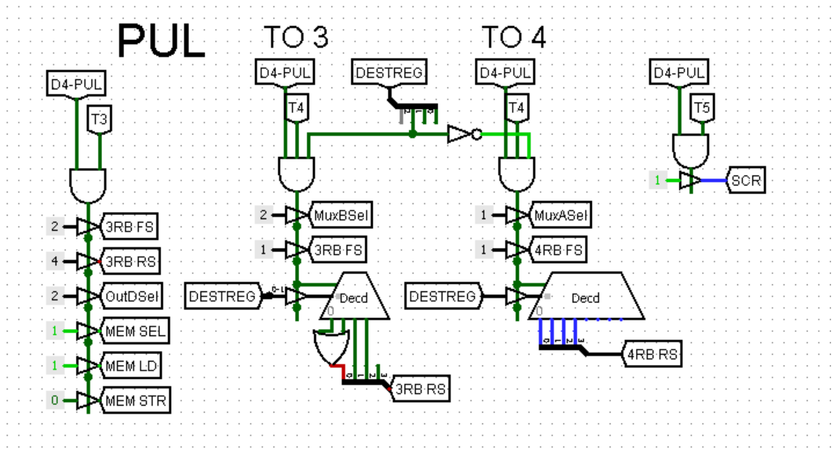

Figure 9: PUL Operation Clock Cycle Configurations

3.6 Binary ALU Operations

3.6.1 ADD

The operation ADD writes the sum of two registers to another register. This operation consist of 6 different cases except operations between 3 register banks such as R0 ↤ PC + AR and SP ↤ PC + AR. These cases are listed below. RF represents any register from register file and ARF represents any register from address register file.

- ARF ↤ RF + RF (PART 1)
- ARF ↤ RF + ARF (PART 1)
- ARF ↤ ARF + RF (PART 1)
- RF ↤ RF + RF (PART 2)
- RF ↤ RF + ARF (PART 2)
- RF ↤ ARF + RF (PART 2)

Fetch and decode operations are same for the whole project. After the fetch and decode operations at the timestamp T3 Add operation is started. Firstly we divided add operation into 2 parts depending on the destination register. In the first part destination is ARF and cycle finishes at T4. As it can be seen above that first part of the add operation consist of three cases. First upper branch belongs to ARF ↤ ARF + RF, lower branch belongs to ARF ↤ RF + ARF and last upper branch belongs to ARF ↤ RF + RF. First part of the circuit can be found below.

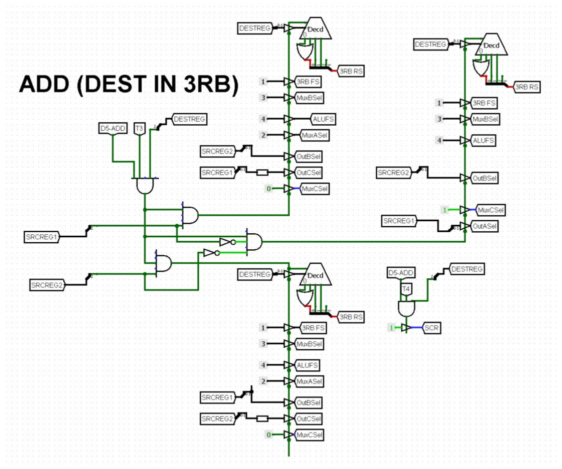

Figure 10: Add destination ARF

When destination is RF, bottleneck occurs at MUX A during RF ↤ RF + ARF and RF ↤ ARF + RF operations therefore we implemented the part2 of the Add operation. We solved bottleneck problem by using stack however this caused complexity to increase since we had to push the value of the ALU into memory and pull it back in the following clock. RF ↤ RF + RF operation does not require stack however it is put under the part 2 since the destination belongs to RF.

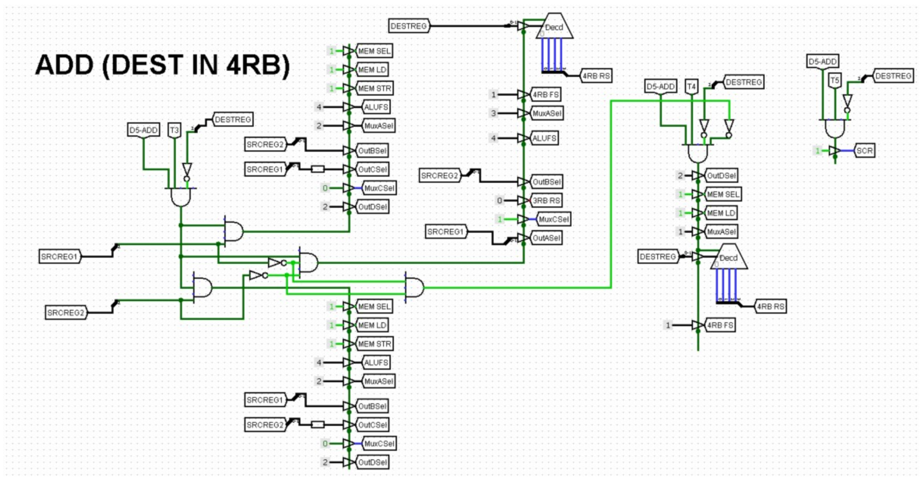

Figure 11: Add destination RF

In instruction T3 first upper branch belongs to RF ↤ ARF + RF, lower branch belongs to RF ↤ RF + ARF and last upper branch belongs to RF ↤ RF + RF. In the first upper branch and the lower branch output of the ALU is pushed into stack in T3. In T4 stack is pulled and the result is written into requested register in RF. Cycle finishes at T5.

3.6.2 SUB

The operation SUB writes the subtraction of two registers to another register.This operation consist of 4 different cases. ARF can not be on the righter side of the subtraction since there is no way from ARF to B input of ALU without overwriting. The other binary ALU operations do not have this problem because those operations have commutative property. Implementable cases are listed below.

- RF ↤ RF - RF
- RF ↤ ARF - RF
- ARF ↤ RF - RF
- ARF ↤ ARF - RF

Implementation is almost same with the ADD only opcodes and funsel are changed therefore figure is not included.

3.6.3 AND

The operation AND writes and gated two registers to another register. Since AND has commutative property, implementation and cases are same with ADD except opcodes and funsel, therefore figure is not included.

3.6.4 OR

The operation OR write or gated two registers to another register. Since or has commutative property, implementation and cases are same with ADD except opcodes and funsel, therefore figure is not included.

3.7 Unary ALU Operations

In this section we have a similar approach for all of our operations, due to their manners being very similar. Just changing the function selectors in the circuits produce the true results.

3.7.1 INC

The operation INC takes the selected SRCREG1 loads it to the DESTREG and decrements it there. This operation consists of 4 different cases because for different destination and source register banks we need to arrange different output selectors. Cases listed below:

- $\mathrm{RF} \mapsfrom \mathrm{RF} + 1$
- $\mathrm{ARF} \mapsfrom \mathrm{RF} + 1$
- $\mathrm{ARF} \mapsfrom \mathrm{ARF} + 1$
- $\mathrm{RF} \mapsfrom \mathrm{ARF} + 1$

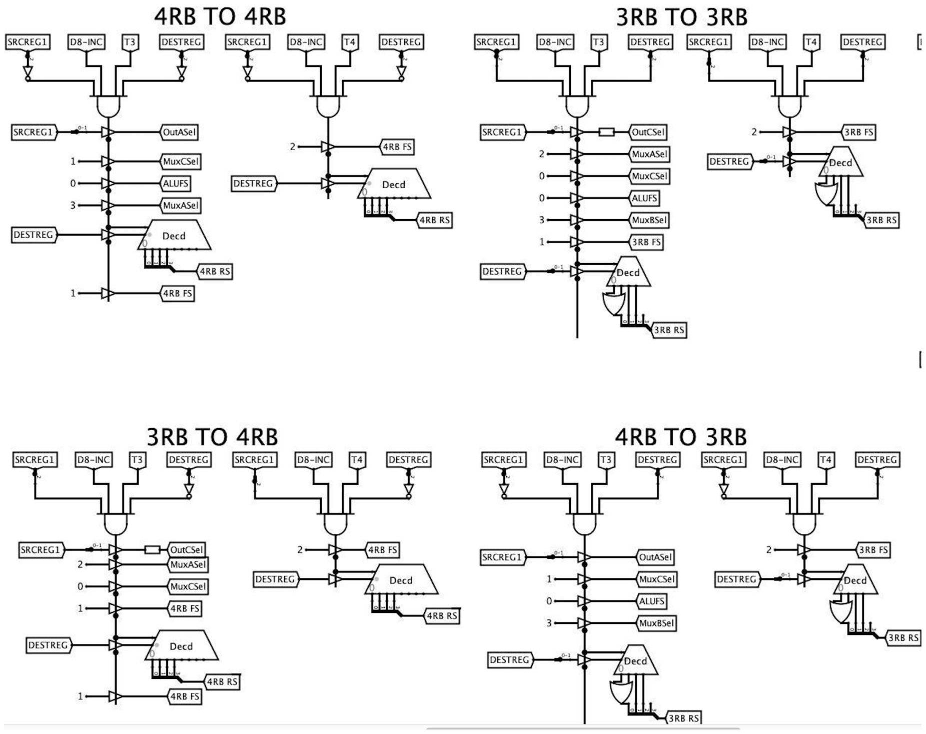

Figure 12: INC Operation Implementation

After fetch & decode of the instruction. We take the input from source register and load it to the destination register. At the other cycle, we enable only the destination register and decrement it there in that cycle. In the last cycle we take the decremented output from our destination register and give it to the ALU using the circuit below in order to observe ZCNO flags. We are using 2 different cases here because we can take the output from the ARF or RF, therefore the circuit differs.

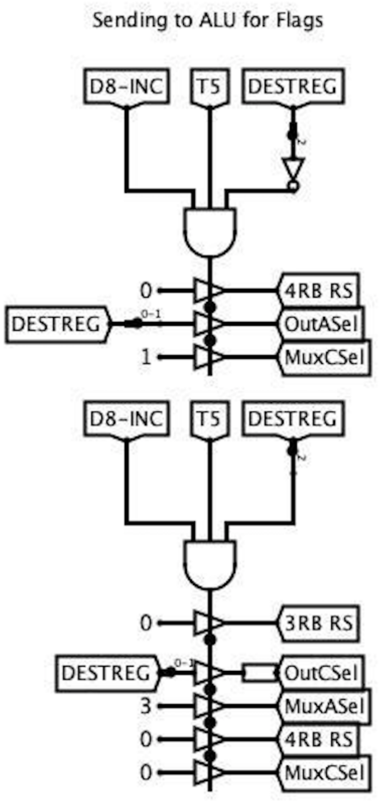

Figure 13: Giving the Result to the ALU

3.7.2 DEC

The operation DEC takes the selected SRCREG1 loads it to the DESTREG and decrements it there. This operation is also consisting of 4 different cases because for different destination and source register banks we need to arrange different output selectors. Cases listed below:

- RF $\hookleftarrow$ RF - 1
- ARF $\hookleftarrow$ RF - 1
- ARF $\hookleftarrow$ ARF - 1
- RF $\hookleftarrow$ ARF - 1

This operation is nearly the same as the INC operation, Figure 12, we just changed the FunSel of the register files in order to decrement the values of the registers. After fetch & decode of the instruction. We take the input from source register and load it to the destination register. At the other cycle, we enable only the destination register and decrement that value there in that cycle. In the last cycle we take the decremented output from our destination register and give it to the ALU using the circuit below in order to observe ZCNO flags. We are using 2 different cases here because we can take the output from the ARF or RF, therefore the circuit differs.

3.7.3 NOT

The NOT operation takes the SRCREG1 value and sends it to the ALU in general. However, there are 4 different cases that we can perform with NOT and not all roads are the same.

- RF ↤ NOT(RF)
- ARF ↤ NOT(RF)
- ARF ↤ NOT(ARF)
- RF ↤ NOT(ARF)

In NOT operation we have extra 1 cycle for the case ARF to RF because we have no ALU's in between ARF to RF directly. You can see below we discriminated the cases with the rightmost circuit and synchronized them differently. In the 3 cases of this operation (except the ARF to RF case), in a cycle we take input from the source register, send the value to the ALU, negate it using the ALU FunSel, then load it to the destination register. In this 3 cases, execution takes only 1 cycle. In ARF to RF case we take the value, load it to RF in the first cycle and in the other cycle we take the value from RF, negate and reload the same value because of the reasons I listed above. We also close the memory cells for this operation since we do not want to operate anything with memory.

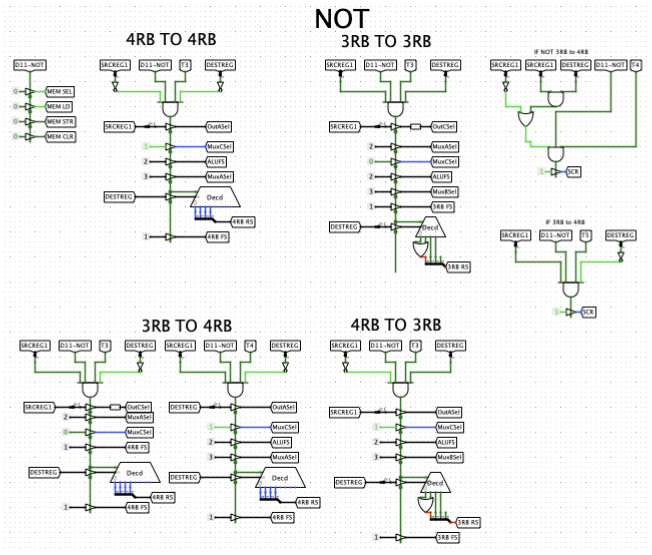

Figure 14: NOT Operation Implementation

3.7.4 LSL

The LSL operation takes the SRCREG1 value and sends it to the ALU in general. However, there are 4 different cases that we can perform with LSL and not all roads are the same.

- RF $\hookleftarrow$ LSL(RF)
- ARF $\hookleftarrow$ LSL(RF)
- ARF $\hookleftarrow$ LSL(ARF)
- RF $\hookleftarrow$ LSL(ARF)

In LSL operation we have extra 1 cycle for the case ARF to RF because we have no ALU's in between ARF to RF directly. You can see below we discriminated the cases with the rightmost circuit and synchronized them differently. In the 3 cases of this operation (except the ARF to RF case), in a cycle we take input from the source register, send the value to the ALU, shift it using the ALU FunSel, then load it to the destination register. In this 3 cases, execution takes only 1 cycle. In ARF to RF case we take the value, load it to RF in the first cycle and in the other cycle we take the value from RF, shift and reload the same value because of the reasons I listed above. We also close the memory cells for this operation since we do not want to operate anything with memory. Implementation is pretty much the same as NOT operation in Figure 14.

3.7.5 LSR

The LSR operation takes the SRCREG1 value and sends it to the ALU in general. However, there are 4 different cases that we can perform with LSR and not all roads are the same.

- RF $\hookleftarrow$ LSR(RF)
- ARF $\hookleftarrow$ LSR(RF)
- ARF $\hookleftarrow$ LSR(ARF)
- RF $\hookleftarrow$ LSR(ARF)

In LSR operation we have extra 1 cycle for the case ARF to RF because we have no ALU's in between ARF to RF directly. You can see below we discriminated the cases with the rightmost circuit and synchronized them differently. In the 3 cases of this operation (except the ARF to RF case), in a cycle we take input from the source register, send the value to the ALU, shift it using the ALU FunSel, then load it to the destination register. In this 3 cases, execution takes only 1 cycle. In ARF to RF case we take the value, load it to RF in the first cycle and in the other cycle we take the value from RF, shift and reload the same value because of the reasons I listed above. We also close the memory cells for this operation since we do not want to operate anything with memory.Implementation is pretty much the same as NOT operation in Figure 14 except the ALU Funsel of course.

3.8 BRA

The operation BRA allows us to set PC register to a value of our chosing. It is represented with the opcode 0x0E and it uses the immediate addressing. It is a very simple operation, takes the input's address field and forwards it to the address register file to be loaded.

- PC ↤ Value

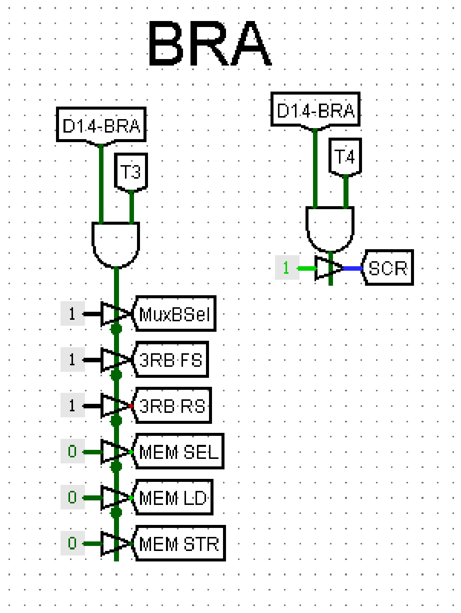

Figure 15: BRA Operation Clock Cycle Configurations

3.9 BEQ & BNE

These operations are very similar, it allows us to re-set the PC register to a desired value. It comes forth especially when evaluating loop conditionals. BEQ stands for branch equal, and it re-sets the PC register to the address field's value if the zero flag that represents the state of the ALU's output is one, meaning that the ALU's output is zero. On the other hand, BNE function works vice versa. If the zero flag is low, meaning the ALU's output is not zero, the value is written into the PC register.

- PC $\hookleftarrow$ Value if Z = 1 (BEQ)
- PC $\hookleftarrow$ Value if Z = 0 (BNE)

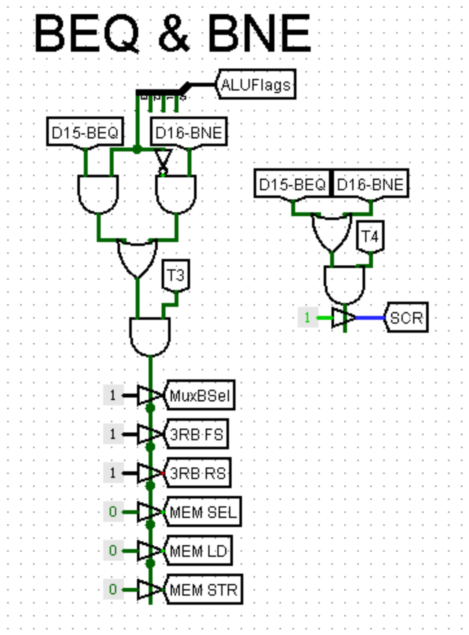

Figure 16: BEQ BNE Operations Clock Cycle Configurations

3.10 CALL

This command pushes the current value of the PC register to the stack in order to remember where to come back, and then changes the value of PC to the address field so that the execution can continue from that address. In the end this is a combination of PSH and BRA operations. First the address field is pushed to the address of the SP register and SP register's value is decremented. Then value of PC register is changed.

- M[SP] ↤ PC
- SP ↤ SP - 1
- PC ↤ Value

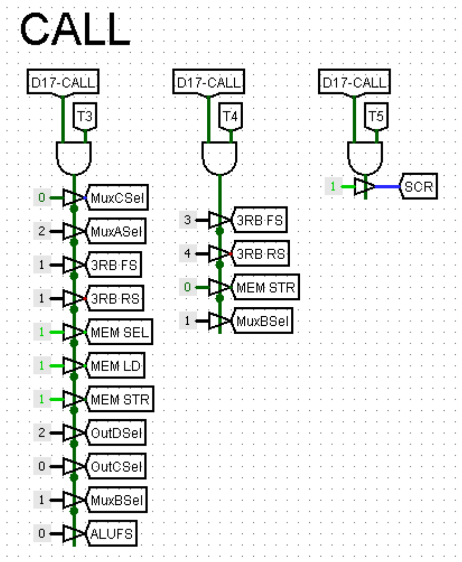

Figure 17: CALL Operation Clock Cycle Configurations

3.11 RET

This command is basically reverse of the CALL command. It pops the value out of the stack and takes the execution of instructions to the popped address value. To sum up this operation can be shown as a combination of PUL and BRA operations. First SP register's value is incremented, after that the value in that address is written into the PC register.

- SP ↤ SP + 1
- PC ↤ M[SP]

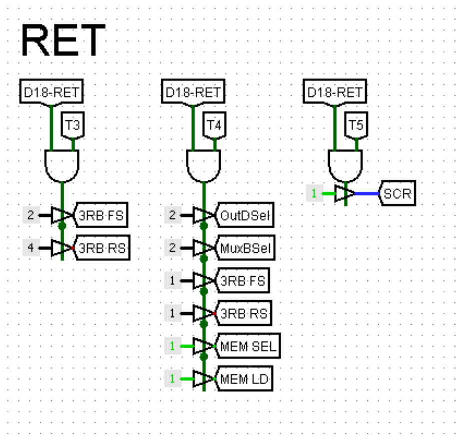

Figure 18: RET Operation Clock Cycle Configurations

4 Results

We have converted the code given to us in the homework file into binary code and run it on our virtual machine. We have also tested numerous cases we have prepared for our operations. We had seen expected results on memory.

5 Discussion

We learned how to build a basic computer. Implementing a concept stack with SP seemed challenging at first but we overcame it. Our collaborative team effort while implementing fetch and decode process' helped us build a baseline that we all could build rest of the project upon.

6 Conclusion

We have faced with red wires in Logisim. Learned floating outputs and tristate buffers. We also learned we can override the red wire in Logisim sometimes. Using the stack seemed hard at first but we have implemented it into our project. Fetch and decode cycles were the most important and time consuming parts. We learned that tristate buffers are very important and effective.