ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION PROJECT REPORT

PROJECT NO : 3

DUE DATE : 03.06.2020

GROUP NO : G26

GROUP MEMBERS:

150170725 : FATİH UZUNOĞLU

150170041 : UZAY DALYAN

150170068 : BERDAN ÇAĞLAR AYDIN

150170071 : YİĞİT EMRE YILMAZ

SPRING 2020

Contents

1 INTRODUCTION

In the third and last project, a hardwired control unit which is capable of executing instructions listed below was designed. The control unit works on 16-bit fixed size instructions.

- LD (LOAD)
- ST (STORE)
- MOV (MOVE)
- PSH (PUSH)
- PUL (PULL)
- ADD
- SUB (SUBTRACT)
- DEC (DECREMENT)
- INC (INCREMENT)
- AND
- OR
- NOT
- LSL
- LSR
- BRA
- BEQ
- BNE
- CALL
- RET

Logisim (v3.3.4 / Logisim-Evolution) was used for the implementations since the original logisim caused problems in our computers because of it being very old and not maintained for a very long time. Note that it requires the latest version of Java runtimes to run. Please read README.txt if you have troubles opening the project.

2 PROJECT PARTS

Since there were a lot of instructions to implement, a modular design approach was chosen to reduce complexity of the control unit. This means that every instruction implementation resides under a module (subproject).

2.1 Sequence Counter

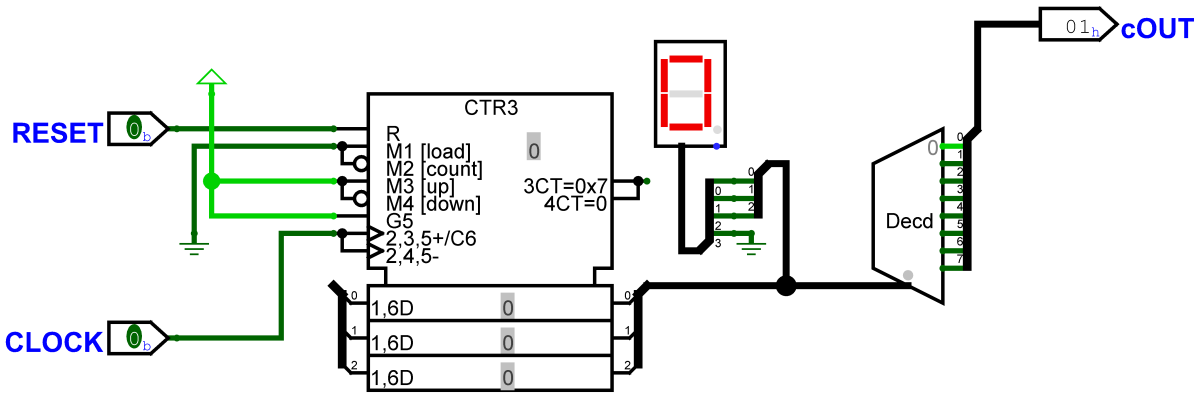

Figure 1: Sequence counter implementation.

Sequence counter is a 3-bit circular counter that counts from decimal 0 to decimal 7. It was initially designed as 4-bit but because no instruction needed more than a few clock cycles, it was reduced to be 3-bit counter.

Output of the sequence counter is decoded with a 3-to-8 bit decoder for instruction implementations to selectively enable a specific cycle.

Output of sequence counter is as such:

```
High bit denotes the active cycle:
00000001
00000010
00000100
...
10000000
00000001
```

Listing 1: Sequence Counter Output

2.2 Fetch Module

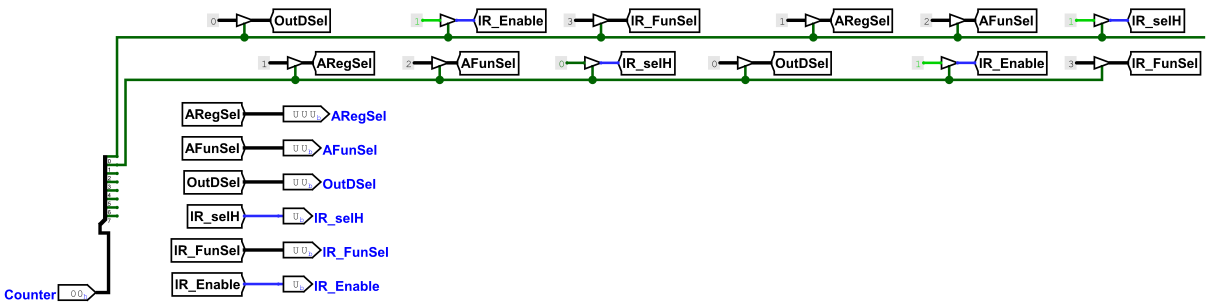

Figure 2: Fetch module.

Fetch module, or fetcher, does the job of retrieving 16-bit instruction to the IR register. Since the memory used in the organization is **asynchronous read**, fetching is done in only 2 cycles.

Fetching prepares a potential operation execution for the basic computer by founding an initial bridge between memory (RAM) and control unit. Without fetching, the computer would not know which instruction to execute. Since the computer is **big-endian**, fetcher first writes into IR_H then in the next cycle it writes into IR_L.

Fetcher increments PC by 2 to point to the next instruction in the memory for the next fetch procedure.

Fetch module does the operations listed below:

- First Cycle (Retrieve first 1-byte into IR_H)
  - Set memory address input to PC.
  - Enable IR.
  - Set IR_H to high.
  - Increment PC by 1.
- Second Cycle (Retrieve second 1-byte into IR_L)
  - Set memory address input to PC.
  - Enable IR.
  - Set IR_L to high.
  - Increment PC by 1.

2.3 ALU Operations Module

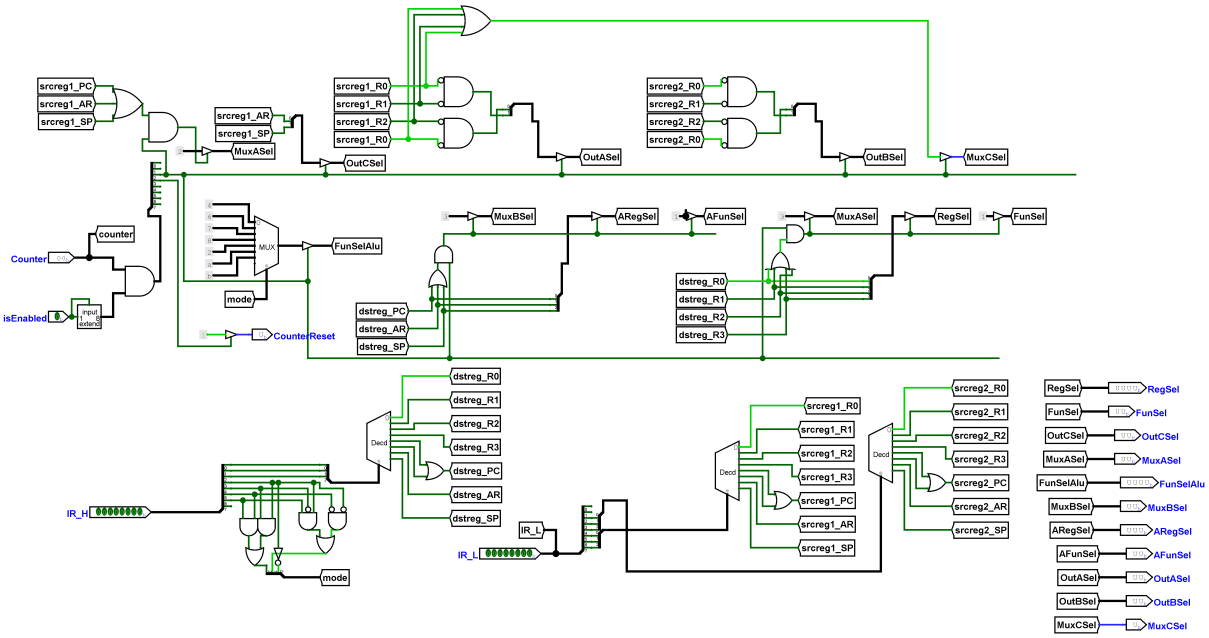

Figure 3: ALU Operations Module

All operations that require ALU to be used are executed under this module. This was done to reduce repetition. Consider two operations {AND and OR}, the only difference between them is different FunSel signals for the ALU and all the other parts are common. This approach allowed to make one module for all ALU operations.

ALU Operations module has an internal 'mode' signal which points to what operation to be executed. This mode signal is mapped from IR_H. The mapping is done as given:

Table 1: ALU Operations internal mode mapping table.

OpCode (HEX)	**mode (DEC)**
0x05 [ADD]	0
0x06 [SUB]	1
0x09 [AND]	2
0x0A [OR]	3
0x0B [NOT]	4
0x0C [LSL]	5
0x0D [LSR]	6

The part of the circuit that does this mapping is located at the bottom left corner of the circuit. Mapping part is implemented by solving a karnaugh map.

This mode signal is connected to the select input of a multiplexer which has required FunSel parameters for the ALU. For example, when OpCode is 0x0A mode becomes 3

which points 4. input of the multiplexer. 4. input of the multiplexer is decimal 8, which corresponds to OR operation in terms of FunSel for ALU.

ALU Operations module does the operations listed below (cycles are relative to fetch):

- First Cycle
  - Check if any of SRCREG1 is in AR file and set MuxASel to 2 if true.
  - Set OutCSel according to SRCREG1.
  - Set OutASel according to SRCREG1.
  - Set OutBSel according to SRCREG2.
  - Set MuxCSel to low if SRCREG1 is in AR File, otherwise set it high.
  - Set FunSelAlu according to mode.
  - If DSTREG is in AR File, set MuxBSel to 3 to forward ALU output to AR File, set AR RegFile's RegSel and FunSel to load the input signal into the respective register.
  - If DSTREG is not in AR File, set MuxASel to 3 to forward ALU output to general purpose register file, set register file RegSel and FunSel to load the input signal into the respective register.
- Second Cycle
  - Reset sequence counter.

2.3.1 Complex ALU Operations Module (Extra)

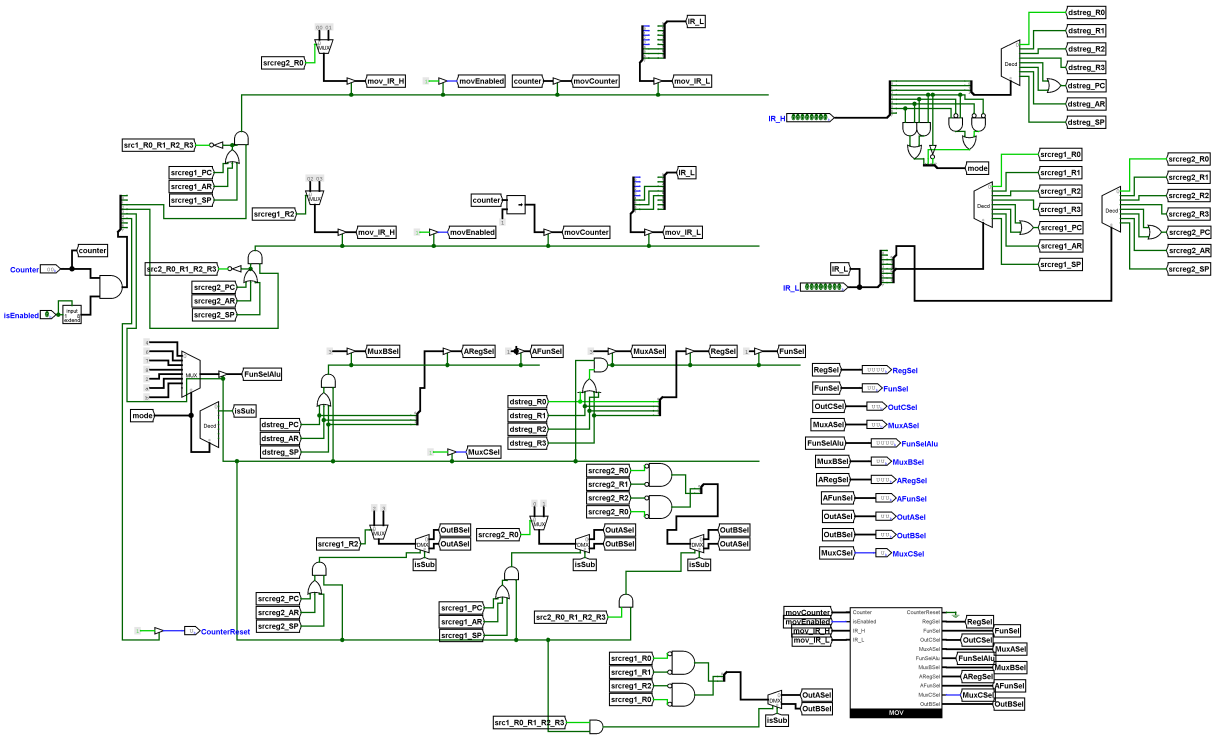

Figure 4: Complex ALU Operations Module.

An alternative ALU operations module was implemented to be able to do operations that are not normally allowed. Normally certain operations are described illegal. For example when both SRCREG is from AR File such as R0 <- AR + PC, it can not be done because only one of AR File registers can be forwarded to ALU input. The other illegal operation is R1 <- R0 - PC because ALU does subtracting as A - B and PC can not be forwarded into ALU input B.

This alternative version of ALU operation modules works by first copying required AR File registers into free registers in the general purpose register file (free means a register that is not pointed either by SRCREG1 or SRCREG2) then forwarding these registers to ALU inputs. If both SRCREGs are in AR File, both of them are copied into general purpose register file. The rest of what it does is the same as normal implementation.

It was not required to implement this module as it is too complex and takes a lot of time to implement. It was implemented for showing off skills and getting **bonus points**.

2.3.2 Old Approach: ADD Module (Alternative)

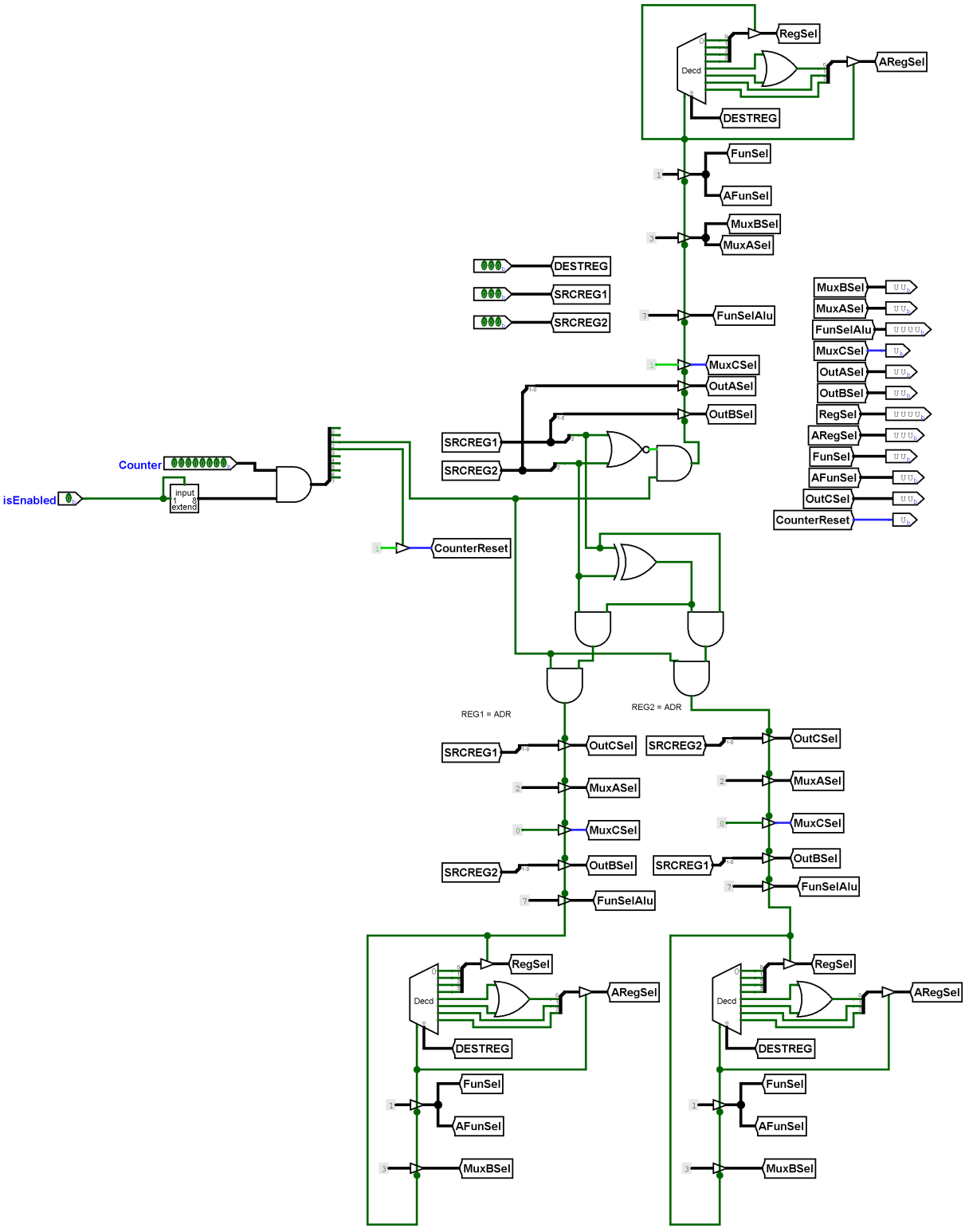

Figure 5: Alternative ADD Operation Module from the old approach.

In our older design of ALU Operations such as ADD, OR, AND etc., we used to implement the same circuit again and again by just switching the **FunSelAlu** to make

the different kind of operations. These modules does the operations listed below:

Since this approach has disadvantages such as cost etc., new solutions used in making the ALU Operations.

- First Cycle
  - Check the locations of SRCREG1 and SRCREG2 in order to determine the path to follow. There are three cases at this point:
    - **case 1:** Both SRCREG1 and SRCREG2 are in the Register File.
    - **case 2:** SRCREG1 is in the Register file and SRCREG2 is in the Address Register File.
    - **case 3:** SRCREG1 is in the Address Register File and SRCREG2 is in the Register File.

    case 1:

    - Set OutASel and OutCSel according to SRCREG1 and SRCREG2.
    - Set MuxCSel to high in order to choose right inputs.
    - Set FunSelAlu according to the operation.
    - Set MuxBSel and MuxASel as *11* in order to send the output to the Address Register File or Register File.
    - Set FunSel and AFunSel as *01* in order to make load operation to Address Register File or Register File.
    - Send DESTREG as the selector bit of 3x8 Decoder, if DESTREG is in Register File, we set ARegSel as *000* in order to not pick any register in Address Register File and prevent unwanted load operations. Same way, if DESTREG is in the Address Register File, we set RegSel as *0000*.

    case 2 and 3:

    - Set OutCSel and OutBSel according to SRCREG1 or SRCREG2. The register inside the Address Register File is used to select OutCSel, other one is used to select OutBSel.
    - Set MuxASel as *10* and MuxCSel as *0* in order to send the output which comes from the Address Register File.
    - Set MuxCSel as *0* in order to send the output which comes from the Address Register File.
    - Set FunSelAlu according to the operation.
    - Set MuxBSel as *11* in order to write to the Address Register File.

* Send DESTREG as the selector bit of 3x8 Decoder, if DESTREG is in Register File, we set ARegSel as *000* in order to not pick any register in Address Register File and prevent unwanted load operations. Same way, if DESTREG is in the Address Register File, we set RegSel as *0000*.

- Second Cycle
  - Reset sequence counter.

2.3.3 Old Approach: LSR Module (Alternative)

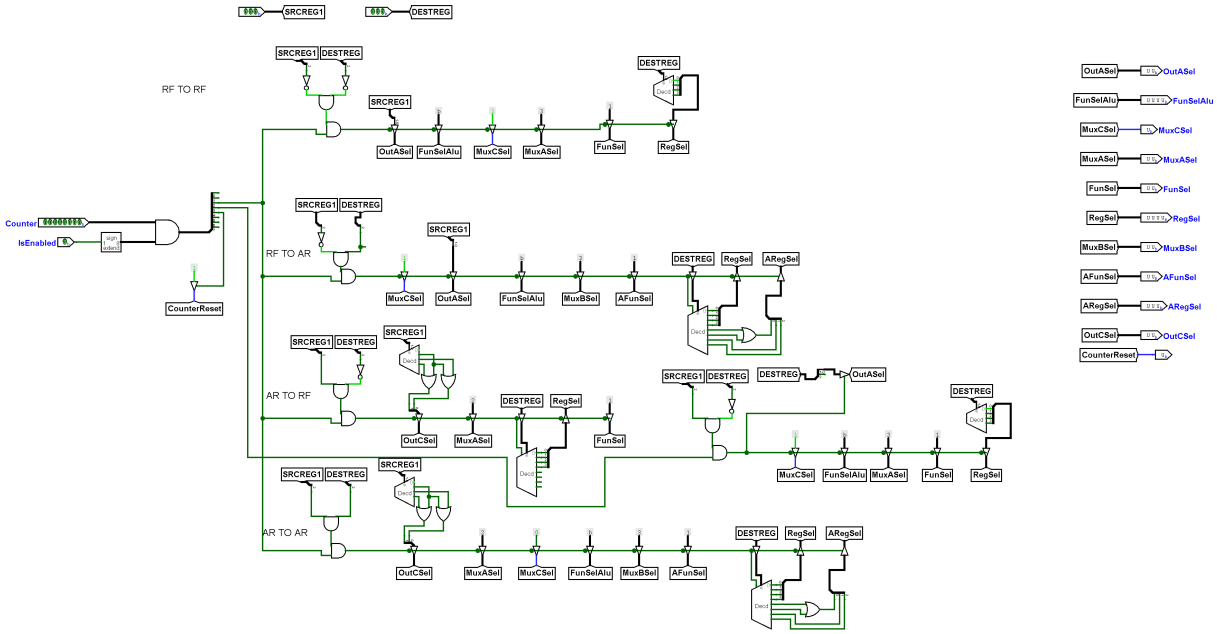

Figure 6: Alternative LSR Operation Module from the old approach.

There is an another way to build Logic Shift Operations modules. But since it costs higher than ours, we builded one of them as an alternative. This module takes srcreg1 and destreg inputs, takes the value in the register that is shown by srcreg1 code, shifts it to right, and writes it to the register that is shown by the destreg code. There are different cases for srcreg1 and destreg being RF or AR registers. Firstly, there is a control part for choosing the case.

- For RF to RF:
  - First Cycle
    * Set OutASel according to SRCREG1.
    * Set FunSelAlu to select Logic Shift Right operation.
    * Set MuxCSel to select OutASel.

        * Set MuxASel to select Alu output.
        * Set Set Funsel and RegSel to load Alu output to register that is chosen according to DESREG.
    - Second Cycle
        * Set CounterReset to reset counter.

- For RF to AR:
    - First Cycle
        * Set MuxCSel to select OutASel.
        * Set OutASel according to SRCREG1.
        * Set FunSelAlu to select Logic Shift Right operation.
        * Set MuxBSel to select Alu output.
        * Set AFunsel and ARegSel to load Alu output to address register that is chosen according to DESREG.
    - Second Cycle
        * Set CounterReset to reset counter.

- For AR to RF:
    - First Cycle
        * Set OutCSel according to SRCREG1.
        * Set MuxASel to select AR output.
        * Set Funsel and RegSel to load SRCC1 to register that is chosen according to DESREG temporarily.
    - Second Cycle
        * Set OutASel according to SRCREG1.
        * Set MuxCSel to select OutASel.
        * Set FunSelAlu to select Logic Shift Right operation.
        * Set MuxASel to select Alu output.
        * Set Funsel and RegSel to load Alu output to register that is chosen according to DESREG.
    - Third Cycle
        * Set CounterReset to reset counter.
    - For AR to AR:

* First Cycle
  - Set OutCSel according to SRCREG1.
  - Set MuxASel to select AR output.
  - Set MuxCSel to select output of MuxA.
  - Set FunSelAlu to select Logic Shift Right operation.
  - Set MuxBSel to select Alu output.
  - Set AFunsel and ARegSel to load Alu output to address register that is chosen according to DESREG.
* Second Cycle
  - Set CounterReset to reset counter.

2.3.4 Old Approach: NOT Module (Alternative)

This design first looks at whether it is from SRCREG1 and DESTREG General Purpose Register (GPR) or from Adress Register File (ARF). Depending on this, 4 different situations arise.

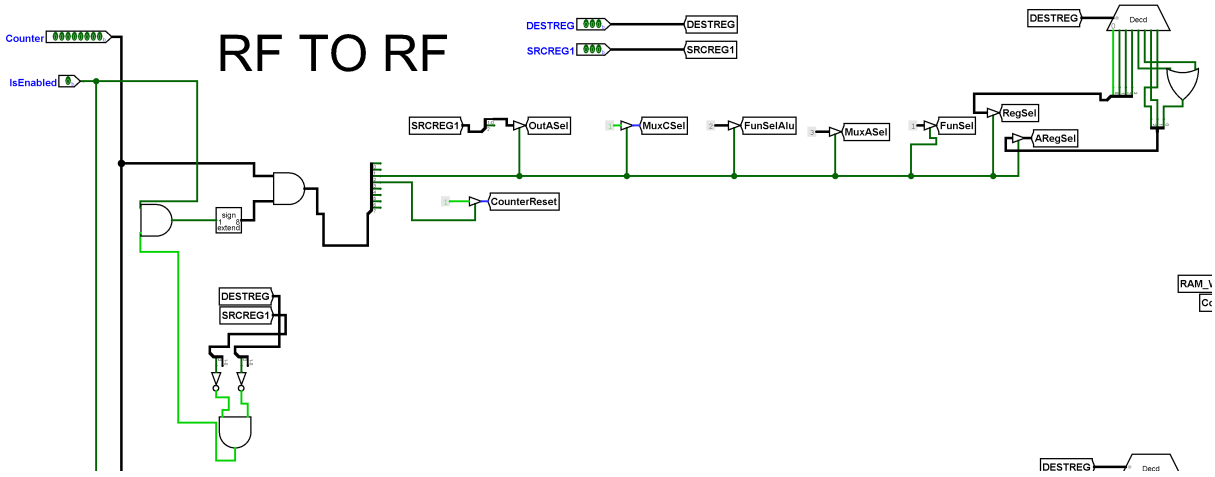

Figure 7: If SRCREG1 and DESTREG are in GPR

If SRCREG1 and DESTREG are in GPR:

In the first clock cycle, the value in SRCREG1 is given to OutA by selecting the appropriate OutASel and sent to the A input of ALU by selecting muxCSel 1. From here, FUNSEL of ALU is set to NOT and the result is sent to the input of GPR with MuxASel. Then, the register that is represented by the appropriate FUNSEL and REGSEL is selected to write to DESTREG.

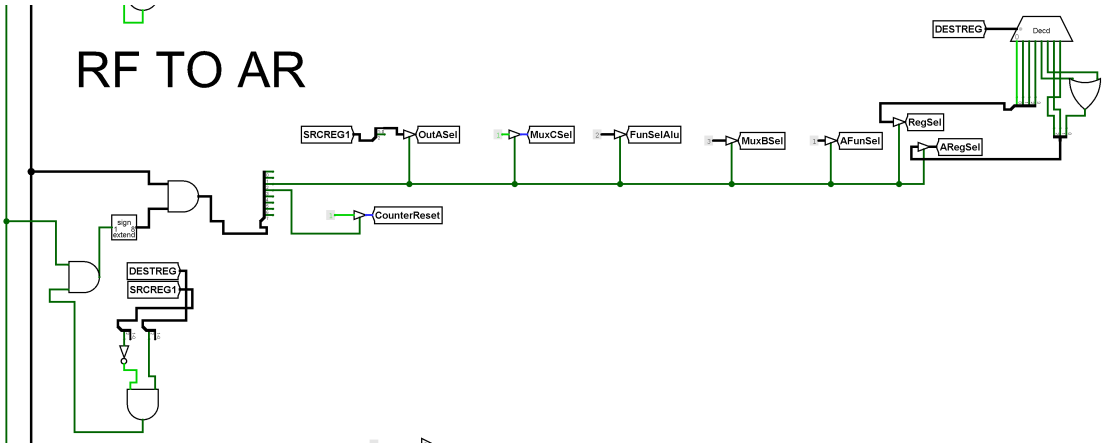

Figure 8: If SRCREG1 is in GPR and DESTREG is in ARF

If SRCREG1 is in GPR and DESTREG is in ARF:

By selecting the appropriate OutASel in the first clock cycle, the value represented by SRCREG is sent to OutA, after it is set as ALU Note, it is sent to the output of ALU. After that, MuxBSel is selected and sent to the input of ARF, this value is printed to DESTREG by selecting the appropriate AREGSEL and AFUNSEL.

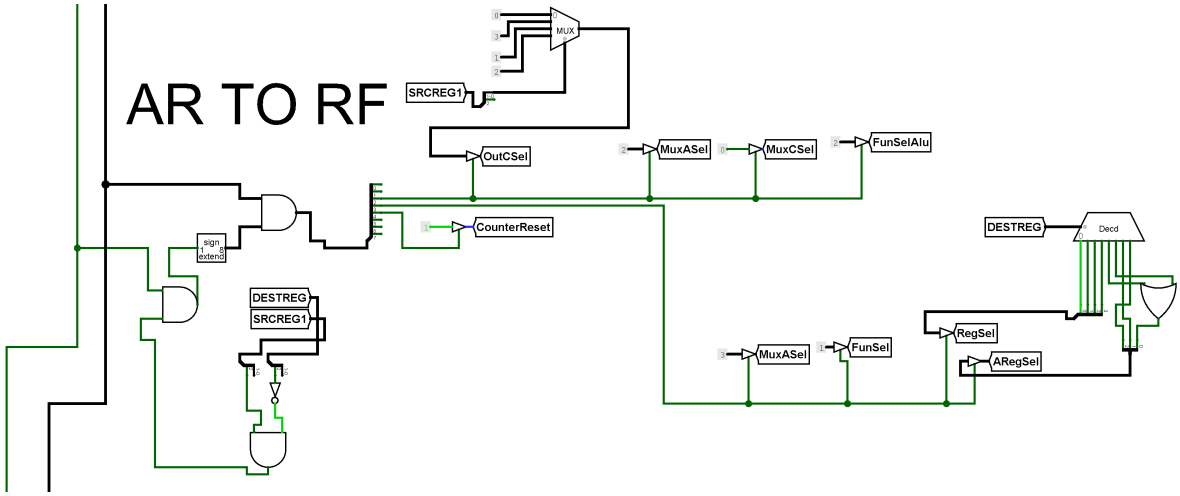

Figure 9: If SRCREG1 is in ARF and DESTREG is in GPR

If SRCREG1 is in ARF and DESTREG is in GPR

In the first clock cycle, the value represented by SRCREG1 is directed to OutCSel. MuxASel MuxCSel is selected and moved to ALU and "NOT" operation is applied. In the second clock cycle, MuxASel is selected and the value that is moved to the input of the ARF is printed by selecting AFUNSEL and AREGSEL according to DESTREG.

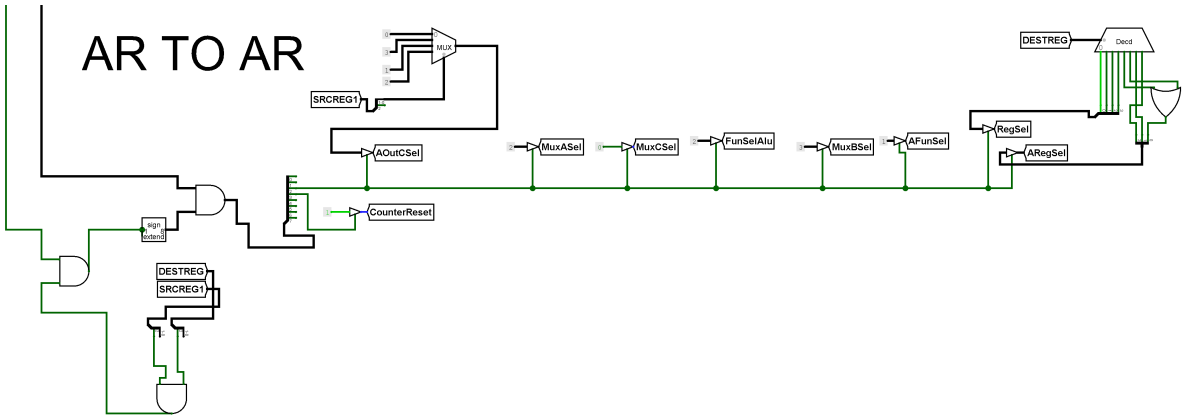

Figure 10: If SRCREG1 and DESTREG are in ARF

If SRCREG1 and DESTREG are in ARF: In this clock cycle, as I explained in the previous section, the value represented by SRCREG1 is passed through ALU and "NOT" operation is applied. Selecting the value from ALU, MuxBSel is selected and redirects to the input of ARF and is printed from the appropriate DESTREG.

2.4 LD Module

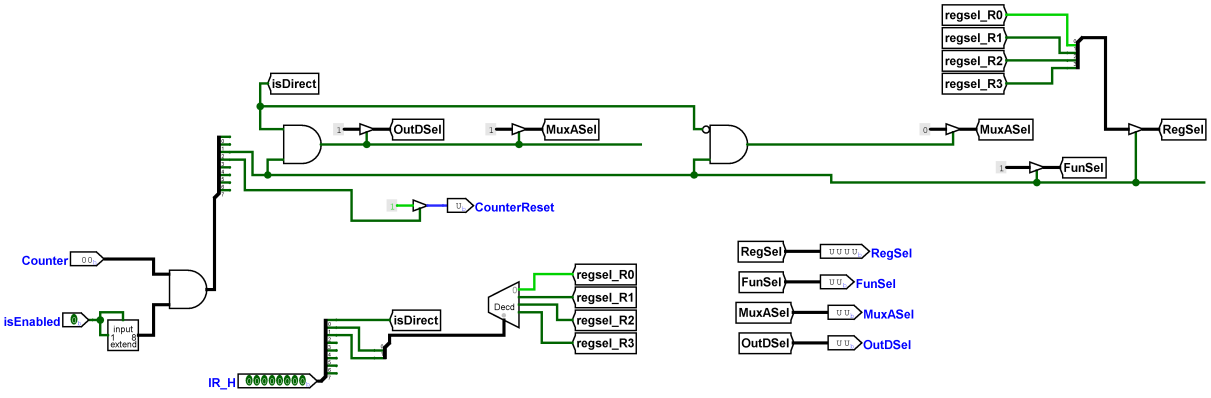

Figure 11: Load Module.

LD Module does LOAD operation. LOAD operation is described as loading a value into a general purpose register. The loaded value is determined by the addressing mode. If it is immediate mode, the value in the address field of the instruction will be directly loaded into the register. If it is direct mode, the value will be the memory content that is pointed by the AR register.

LD module does the operations listed below (cycles are relative to fetch):

- First Cycle
  - If addressing is direct:

* Set memory address input as AR register.
* Set MuxASel to select memory output.

– If addressing is immediate:

* Set MuxASel to select IR_L (content of address field).

– Set RegSel according to register destination.

– Set FunSel to make register LOAD.

- Second Cycle

– Reset sequence counter.

2.5 ST Module

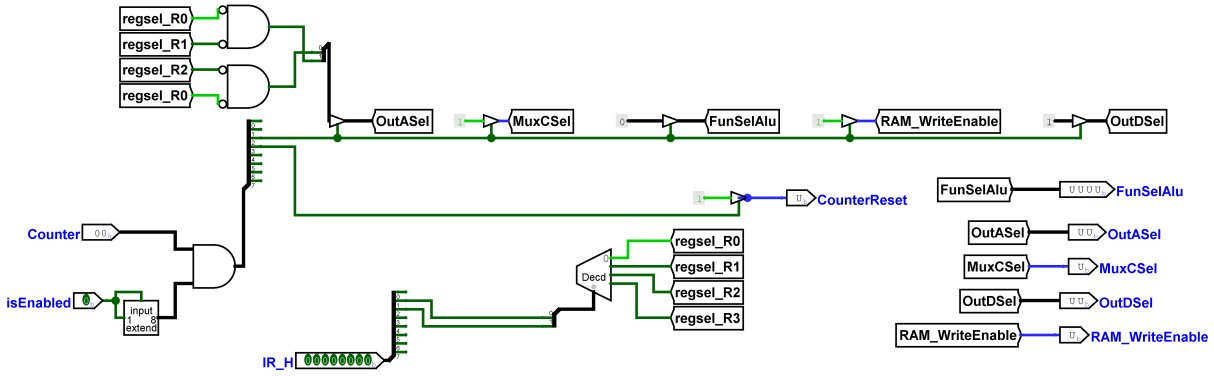

Figure 12: Store Module

ST Module does STORE operation. STORE operation is described as storing a value into a specific address of the memory. The value that is stored in the memory comes from the register that is specified in REGSEL field. The address of the memory is pointed by AR register.

ST module does the operations listed below (cycles are relative to fetch):

- First Cycle

– Set OutASel according to REGSEL.

– Set MuxCSel to forward OutASel into ALU input A.

– Set FunSelAlu 0 to make ALU act as a buffer for input A.

– Activate RAM WriteEnable signal.

– Set OutDSel to forward AR into memory address input.

- Second Cycle

– Reset sequence counter.

2.6 MOV Module

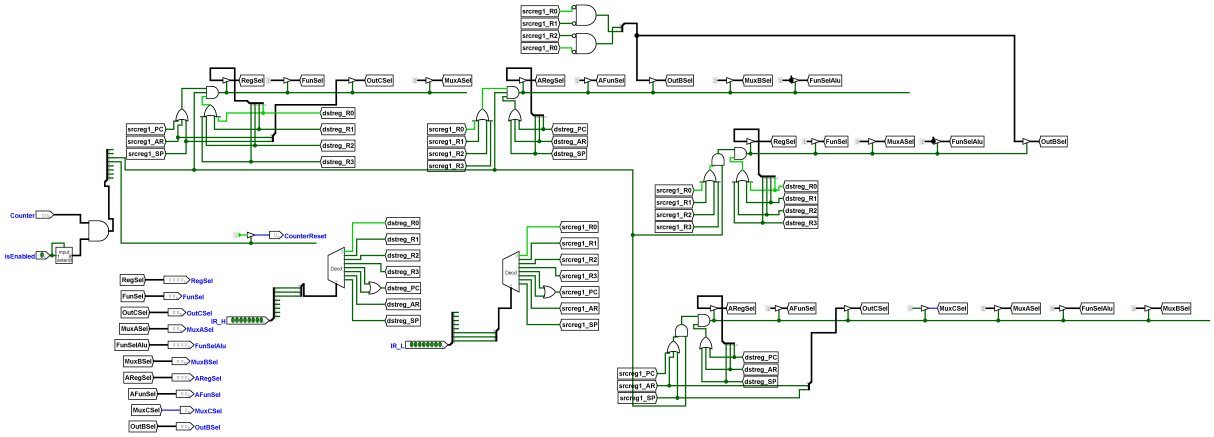

Figure 13: Move Module

MOV Module does MOVE operation. MOVE operation is an operation where the destination register's content gets replaced by source register's content. MOVE is a destructive operation because after MOVE operation, destination register's content becomes permanantly lost.

MOV module does the operations listed below (cycles are relative to fetch):

- First Cycle
  - If SRCREG1 is in AR file and DSTREG is a general purpose register:
    - Set RegSel according to DstReg.
    - Set FunSel to load.
    - Set OutCSel to forward SRCREG1.
    - Set MuxASel to forward incoming SRCREG1 to general purpose register file input.
  - If SRCREG1 is a general purpose register and DSTREG is in AR File:
    - Set ARegSel according to DstReg.
    - Set AFunSel to load.
    - Set OutBSel to forward SRCREG1.
    - Set MuxBSel to forward ALU output to address register file input.
    - Set FunSelAlu to 1 to make ALU act as a buffer for input B.
  - If both SRCREG1 and DSTREG are general purpose registers:
    - Set RegSel according to DstReg.
    - Set FunSel to load.

* Set MuxASel forward ALU output to general purpose register file input.
* Set FunSelAlu to 1 to make ALU act as a buffer for input B.
* Set OutBSel to forward SRCREG1.

– If both SRCREG1 and DSTREG are in AR File:

* Set ARegSel according to DstReg.
* Set AFunSel to load.
* Set OutCSel to forward SRCREG1.
* Set MuxCSel to forward incoming SRCREG1 into ALU input A.
* Set MuxASel to forward incoming SRCREG1 into MuxCSel.
* Set FunSelAlu to 0 to make ALU act as a buffer for input A.
* Set MuxBSel to forward ALU output into AR file input.

- Second Cycle

– Reset sequence counter.

2.7 PSH Module

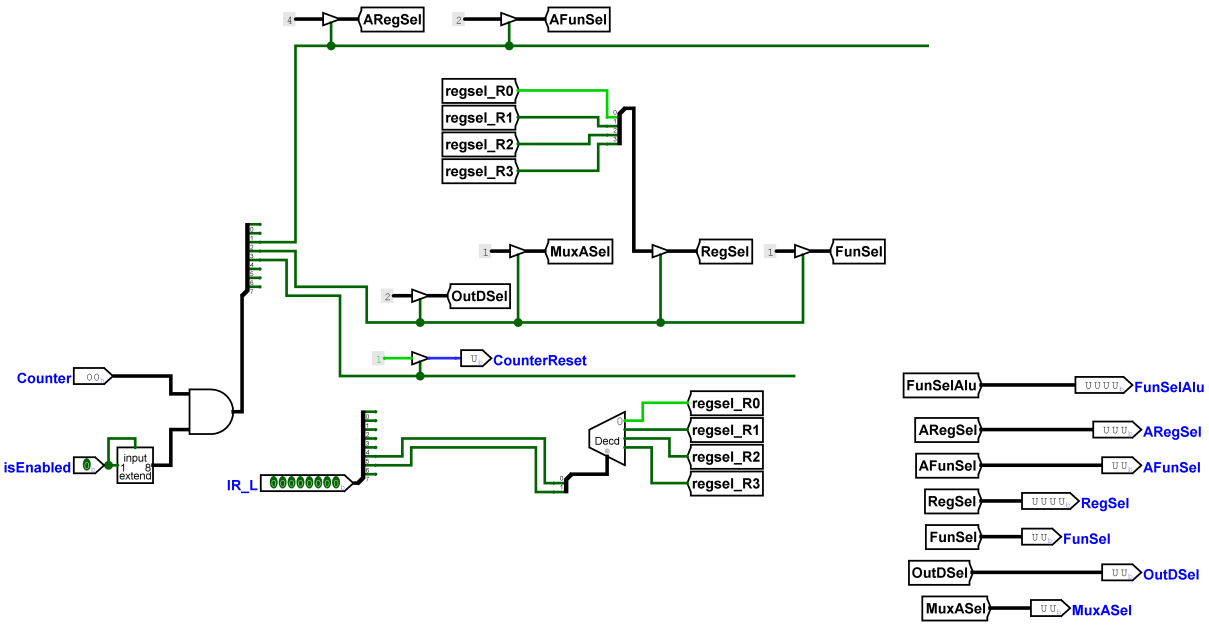

Figure 14: Push Module

PSH Module does PUSH operation. PUSH operation writes REGSEL register's content into memory with address pointed by SP register then it decrements SP by 1.

PSH module does the operations listed below (cycles are relative to fetch):

- First Cycle

- Set OutBSel according to RegSel.
- Set FunSelAlu to 1 to make ALU act as a buffer for input B.
- Set OutDSel to forward SP into memory address input.
- Enable RAM write.

- Second Cycle
  - Set ARegSel to select SP.
  - Set AFunSel to decrement operation.

- Third Cycle
  - Reset sequence counter.

2.8 PUL Module

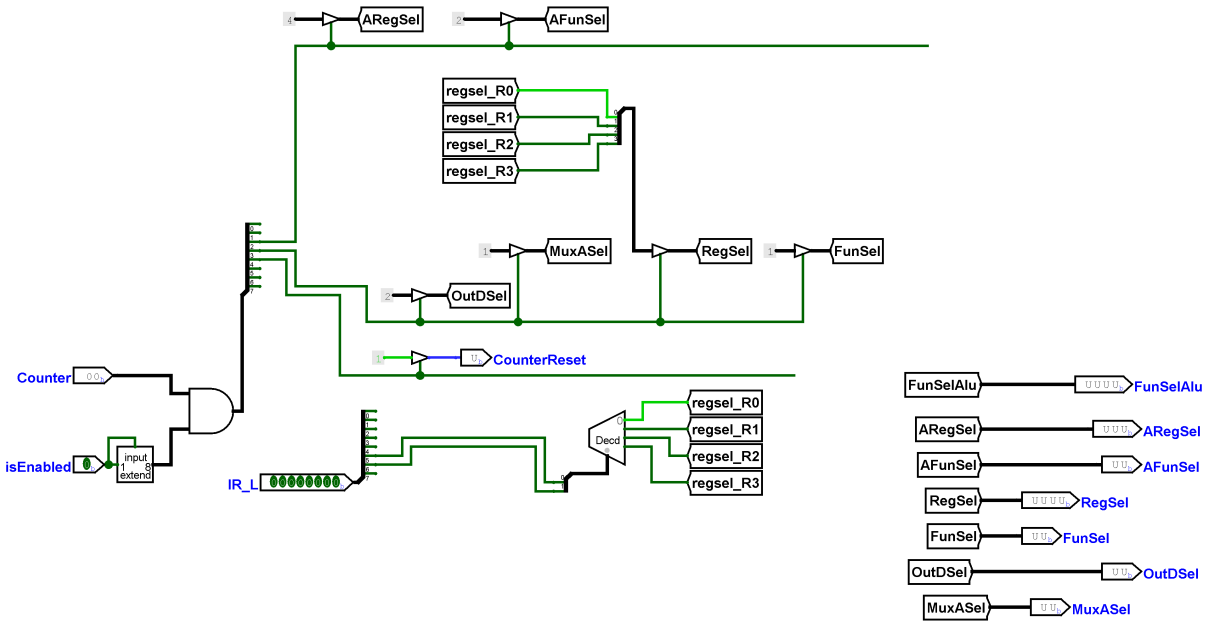

Figure 15: Pull Module

PUL Module does PULL operation. PULL operation first increments SP by 1, then it writes the memory content with address pointed by SP to the register pointed by REGSEL.

PUL module does the operations listed below (cycles are relative to fetch):

- First Cycle
  - Set ARegSel to select SP.

- Set AFunSel to increment operation.

- Second Cycle
  - Set OutDSel forward SP into memory address input.
  - Set MuxASel to forward memory output to general purpose register file input.
  - Set RegSel according to REGSEL.
  - Set FunSel to load.

- Third Cycle
  - Reset sequence counter.

2.9 INC/DEC Module

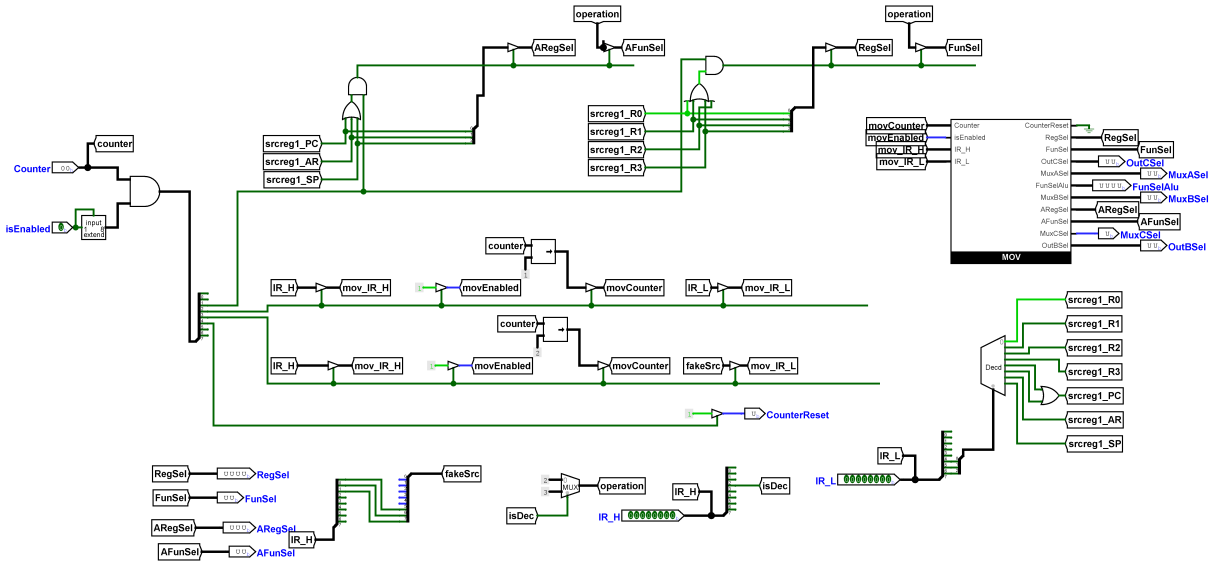

Figure 16: INC/DEC Module

INC/DEC module does increment or decrement operation. Increment and decrement operations are operations that increments or decrements SRCREG1 by 1, then moves the content to DESTREG.

This implementation of increment and decrement operation does the moving by using MOV operation. And it does MOV two times (first, from SRCREG1 to DSTREG; then, from DSTREG to DSTREG) to ensure that the FLAGs are updated. By moving two times, chances of flags being not updated are eliminated completely. Consider the case where DSTREG is R0 and SRCREG is AR. If MOV was being done only one time, flags would not update because SRCREG would not pass through the ALU. When MOV is done two times, there is no way SRCREG not passed through the ALU.

INC/DEC module does the operations listed below (cycles are relative to fetch):

- First Cycle (increment or decrement)
  - If SRCREG1 is in address register file:
    - Set ARegSel according to SRCREG1.
    - Set AFunSel increment or decrement depending on the mode.
  - If SRCREG1 is a general purpose register:
    - Set RegSel according to SRCREG1.
    - Set FunSel increment or decrement depending on the mode.
- Second Cycle (move SRC to DST)
  - Set IR_H of MOV to IR_H (to set destination register of MOV DESTREG).
  - Enable MOV module.
  - Adjust counter for MOV module. (Shift right by 1 to only do first cycle of MOV module).
  - Set IR_L of MOV to IR_L (to set source register SRCREG1).
- Third Cycle (move DST to DST to ensure FLAGS are up-to-date)
  - Set IR_H of MOV to IR_H (to set destination register of MOV DESTREG).
  - Enable MOV module.
  - Adjust counter for MOV module. (Shift right by 2 to only do first cycle of MOV module).
  - Set IR_L of MOV to fakeSrc which points DESTREG (to set source register DESTREG).
- Fourth Cycle
  - Reset sequence counter.

2.9.1 Old Approach: INC Module (Alternative)

When we first designed the Increment module, we did not design the MOV module. Therefore, the first module, the alternative module, has a larger structure than the increment module in our circuit. The alternative module first looks at the SRCREG1 and DESTREG entries. According to DESTREG and SRCREG1 entries, 4 different states occur.

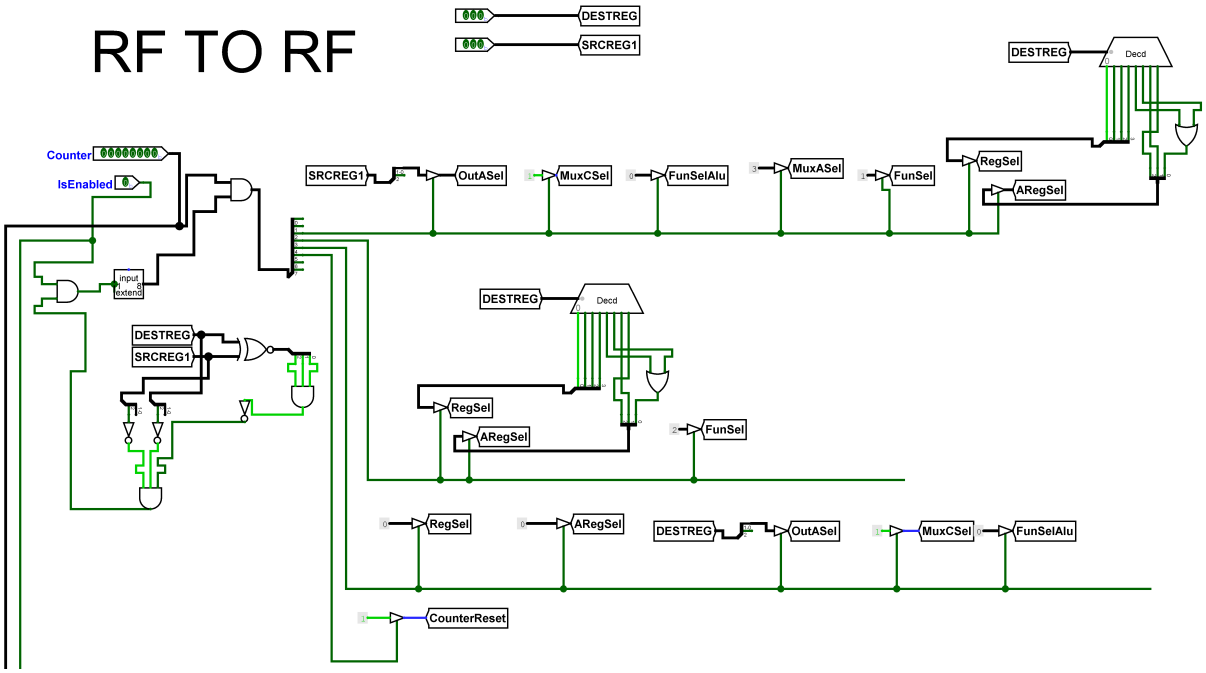

Figure 17: If DESTREG and SRCREG1 are in General Purpose Register File (GPR) and differ from each other:

If DESTREG and SRCREG1 are in General Purpose Register File (GPR) and differ from each other:
In the first clock cycle, the value of the register represented by SRCREG1 is given to outputA. Then the data is sent to A part of ALU by selecting this value muxCsel 1. It is passed through the ALU without being processed, and the alu output is selected with muxAsel and sent to the GPR. Then the register that DESTREG represents is selected and the load is selected with FUNSEL to GPR and SRCREG1 is written in DESTREG. In the 2nd clock cycle, the value in the register represented by DESTREG is selected by selecting FUNSEL and incremented. In the 3rd clock cycle, the value of the register represented by DESTREG is given outA and sent to the ALU via the muxCsel selection and passed through the ALU without processing. The purpose of this process is to trigger the Z flag after the increment or decrement process.

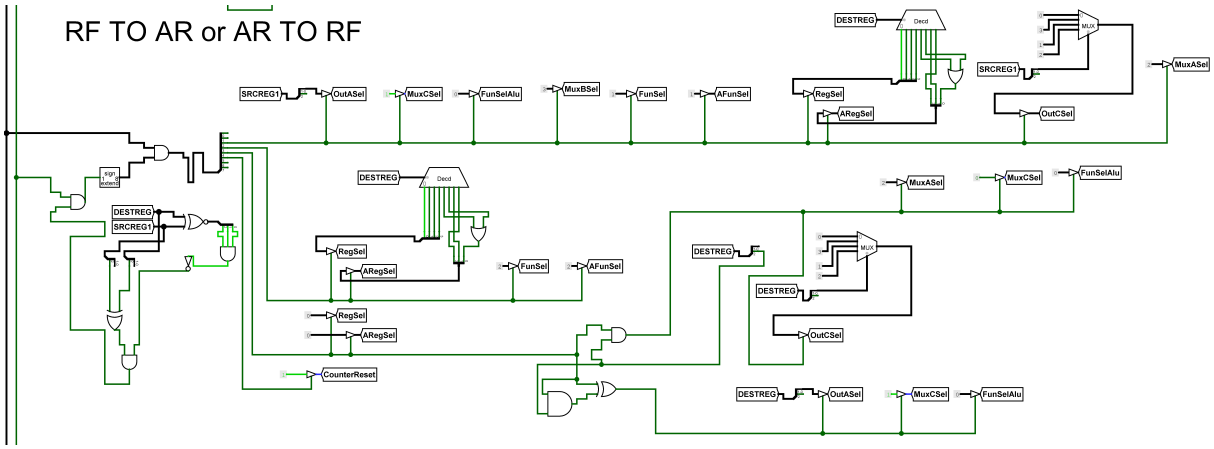

Figure 18: If SRCREG1 ARF and DESTREG GRP or SRCREG1 GPR and DESTREG ARF and differ from each other

If SRCREG1 ARF and DESTREG GRP or SRCREG1 GPR and DESTREG ARF and differ from each other:
In the first clock cycle, as described in the previous section, SRCREG1 is written to DESTREG. In the 2nd clock cycle, FUNSEL and REGSEL suitable for DESTREG are selected and incremented. In the third clock cycle, there is a system that is divided into 2 ways depending on whether DESTREGin is in GPR or ARF. If it is in DESTREG GPR, as in the previous section, ALU migration is done. If DESTREG is in ARF, the appropriate AREGSEL for DESTREGE is selected and given to the outC. OutC is transported to MuxC with MuxASel MuxCsel is selected by selecting 0 and passing the ALU and flag is triggered.

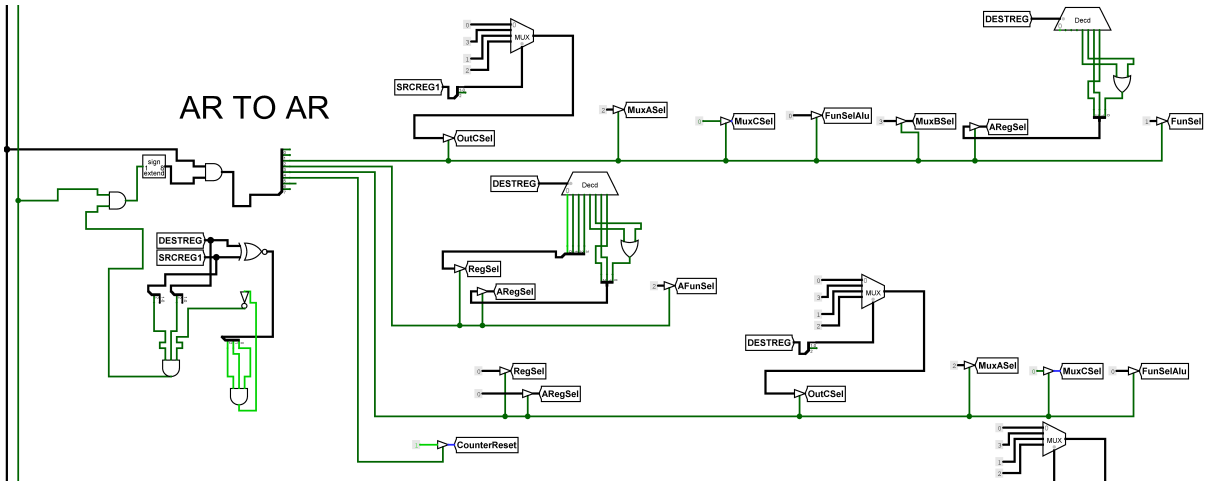

Figure 19: If SRCREG1 and DESTREG ARF and they are different from each other

If SRCREG1 and DESTREG ARF and they are different from each other:
In the first clock loop the same procedure as in the previous section is done and the

value of SRCREG1 is written to DESTREG. In the 2nd clock cycle, DESTREG is selected and incremented. In the third clock cycle, the incremented value is passed through ALU.

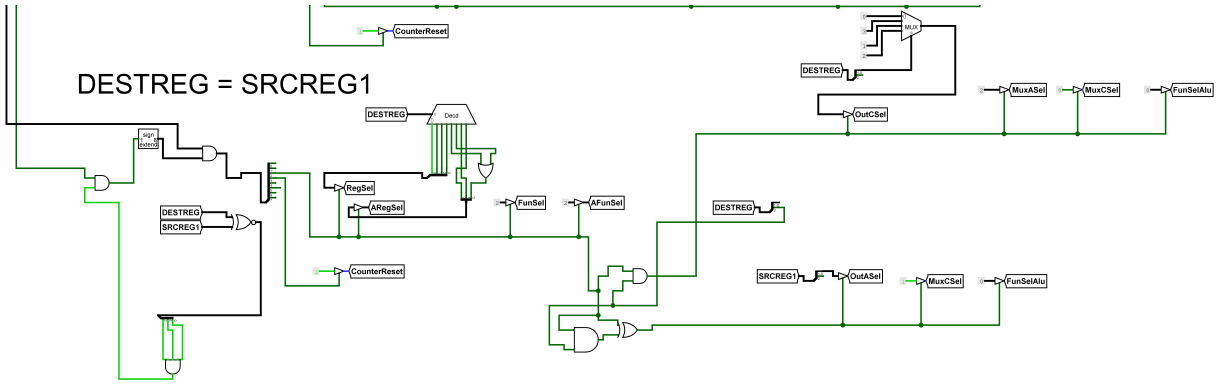

Figure 20: If DESTREG and SRCREG1 are equal

If DESTREG and SRCREG1 are equal:

SRCREG1 is directly selected and incremented without being printed on DESTREG. It is then passed through ALU.

2.10 BRA Module

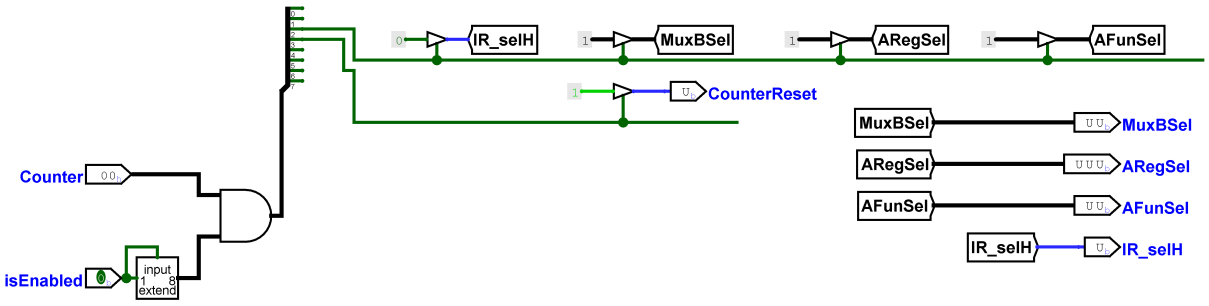

Figure 21: BRA Module

BRA Module does BRANCH operation. BRANCH operation is an operation that is used to change PC register's content with a different value. It is used to divert into subprograms during program execution. Branching or jumping are one of the essential operations that makes the CPU turing complete. It also allows recursive programs to be developed.

- First Cycle
  - Set IR_H to 0.
  - Set MuxBSel to forward IR out to address register file input.

- Set ARegSel to select PC.
- Set AFunSel to select load operation.

- Second Cycle
  - Reset sequence counter.

2.11 BEQ Module

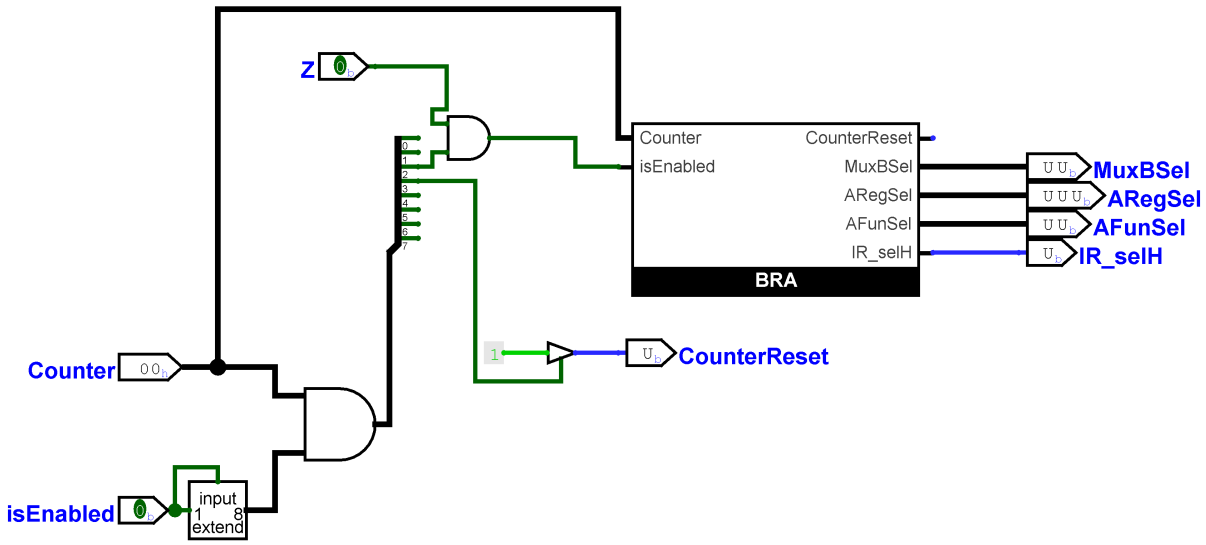

Figure 22: BEQ Module

BEQ Module does BRANCH if EQUAL operation. It does branch operation if only Z flag is set to 1. To reduce repetition and complexity, already implemented BRA module is used to implement BEQ.

- First Cycle
  - Execute 'BRA' if Z is 1.

- Second Cycle
  - Reset sequence counter.

2.12 BNE Module

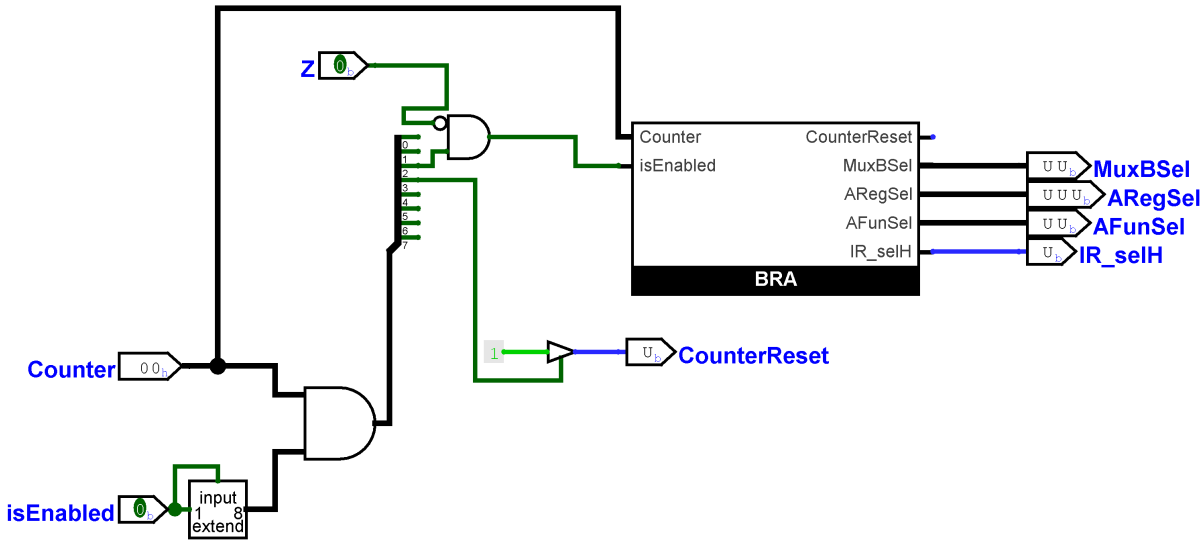

Figure 23: BNE Module

BNE Module does BRANCH if not EQUAL operation. It does branch operation if only Z flag is set to 0. To reduce repetition and complexity, already implemented BRA module is used to implement BNE.

- First Cycle
  - Execute 'BRA' if Z is 0.
- Second Cycle
  - Reset sequence counter.

2.13 CALL Module

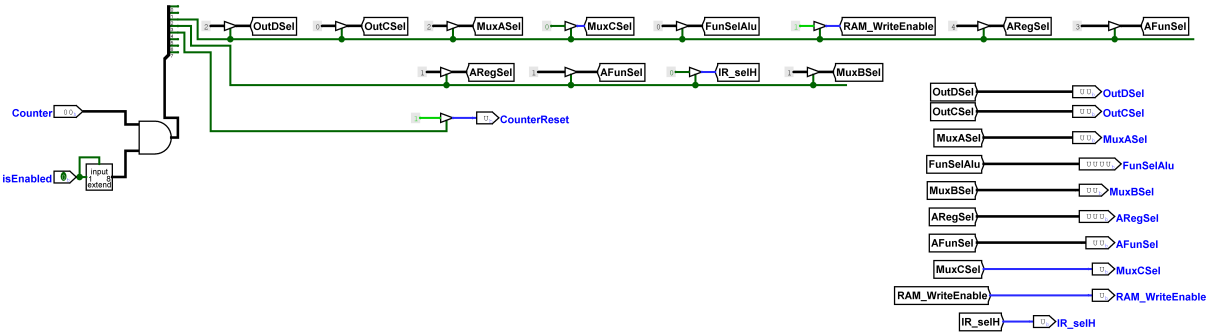

Figure 24: CALL Module

CALL Module does CALL operation. CALL is a specific type of branching operation which saves the content of PC when the diversion occurs. This behaviour of CALL allows us to implement functions that we can return from. With BRA, returning from the diverted path is impossible because return address is unknown. However, since CALL saves return address, it becomes trivial to return to that address later on.

CALL saves current address (PC) to the stack. Due to this reason, stack pointer (SP) must be configured properly before using this operation.

- First Cycle
  - Set OutDSel to forward SP into memory address input.
  - Set OutCSel to forward PC into MuxASel.
  - Set MuxASel to forward PC into MuxCSel.
  - Set MuxCSel to forward PC into ALU input A.
  - Set FunSelAlu to 0 to make ALU act as a buffer for input A.
  - Enable write for RAM.
  - Set ARegSel to select SP.
  - Set AFunSel to select decrement operation.
- Second Cycle
  - Set IR_H to 0.
  - Set MuxBSel to forward IR out to address register file input.
  - Set ARegSel to select PC.
  - Set AFunSel to select load operation.
- Third Cycle
  - Reset sequence counter.

2.14 RET Module

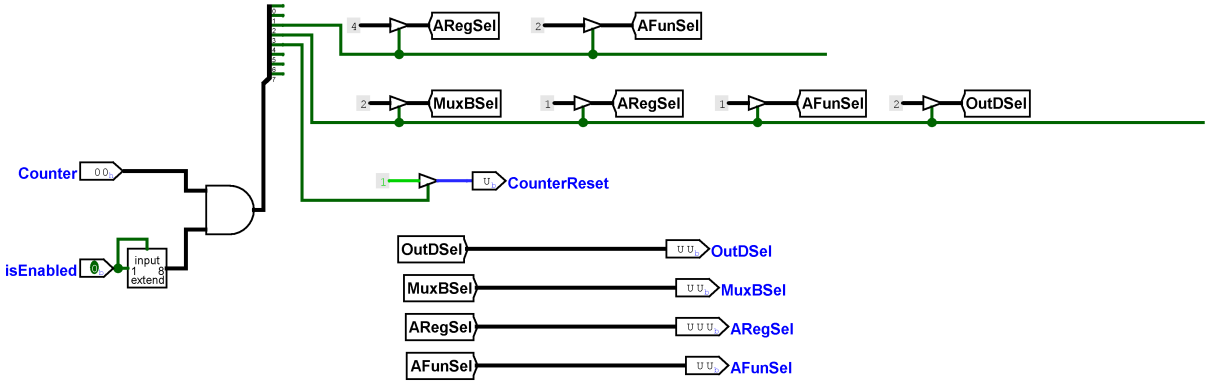

Figure 25: RET Module

RET Module does RETURN operation. RETURN operation is used to return back to the address for execution where branching (call) happened. Return operation should be used after Call operation is used.

RETURN increments stack pointer (SP) by 1. Then it overwrites PC with the content of RAM with address pointed by the stack pointer (SP).

- First Cycle
  - Set ARegSel to select SP.
  - Set AFunSel to select increment operation.
- Second Cycle
  - Set MuxBSel to forward memory out into address register file input.
  - Set ARegSel to select PC.
  - Set AFunSel to select load operation.
  - Set OutDSel to forward SP into memory address input.
- Third Cycle
  - Reset sequence counter.

2.15 The Control Unit

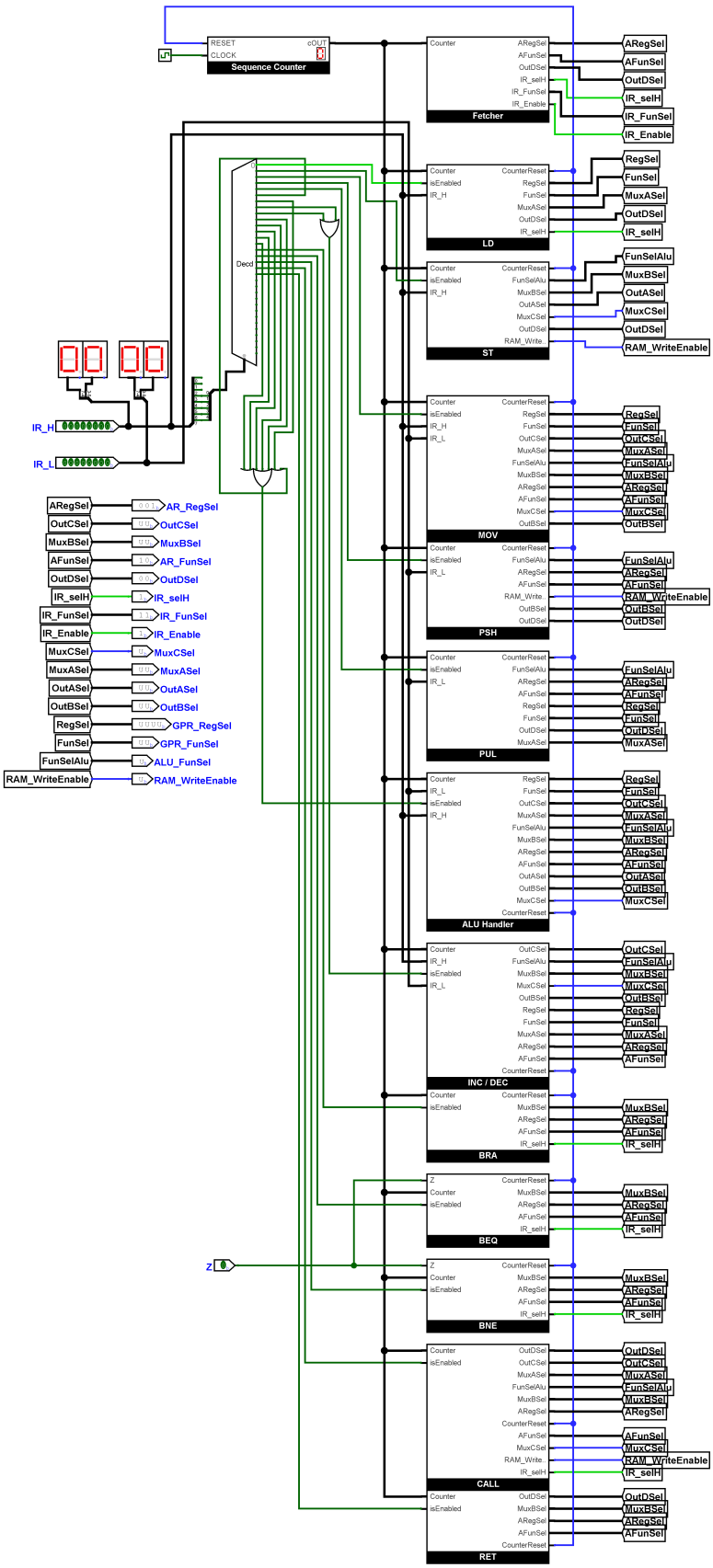

Figure 26: The Control Unit

Since a modular approach was used to implement all instructions, it was rather convenient to wrap-up all of the modules and create the control unit.

Note that the decode phase happens here with a 5-to-32 decoder. Decoder outputs are fed into modules as 'isEnabled' signal. The rest is self explanatory.

2.16 Computer Organization

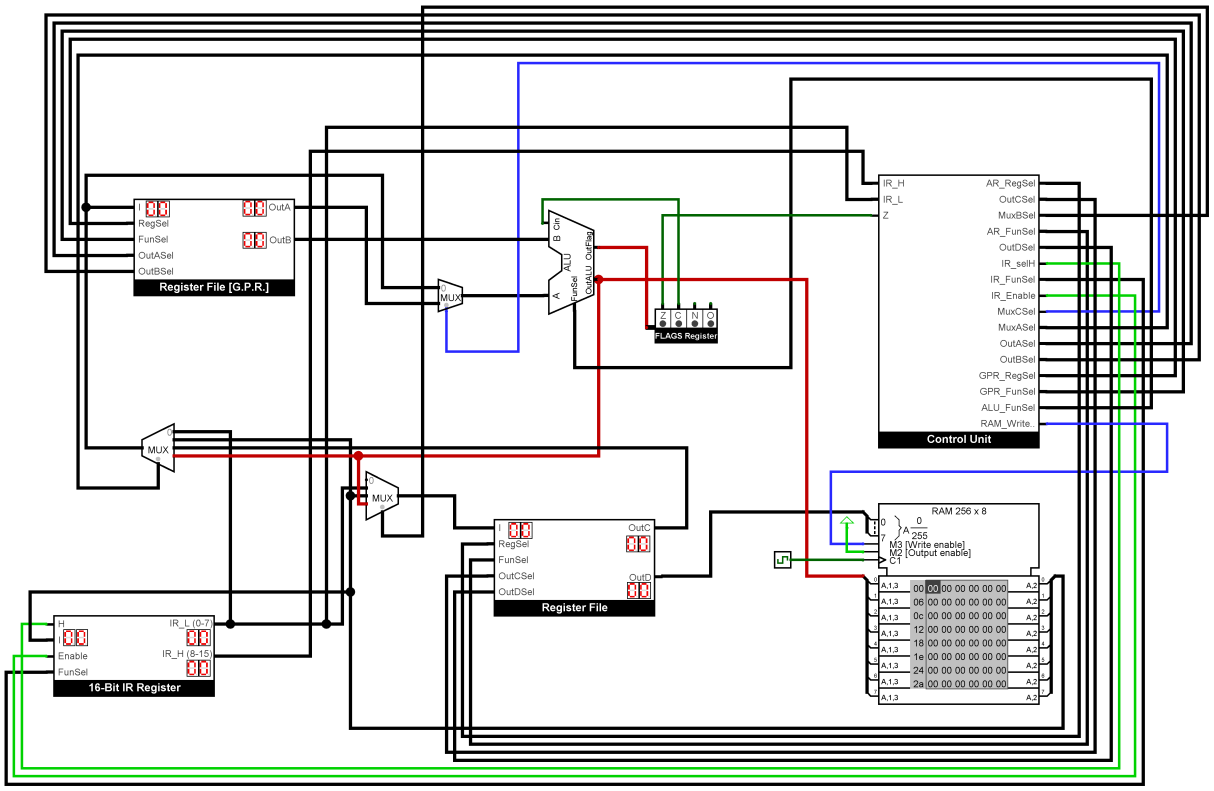

Figure 27: The Control Unit

The given computer organization schematic with the second project was used as the template for the basic computer. It was not modified. The designed control unit was put inside the organization and needed pins were connected.

3 RESULTS

All modules were tested case by case, and all of them worked properly.

3.1 Example Program

An example program which calculates M[A0]+M[A1]+M[A2]+M[A3]+M[A4] and stores the result at M[A6] was given in the project documentation file.

```
    ORG 0x20 # Write the program starting from the address 0x20
    LD R0 IM 0x05 # R0 is used for iteration number
    LD R1 IM 0x00 # R1 is used to store total
    LD R2 IM 0xA0
    MOV AR R2 # AR is used to track data adrress: starts from 0xA0
LABEL:  LD R2 D # R2 <- M[AR] (AR = 0xA0 to 0xA4)
    INC AR AR # AR <- AR + 1 (Next Data)
    ADD R1 R1 R2 # R1 <- R1 + R2 (Total = Total + M[AR])
    DEC R0 R0 # R0 <- R0 - 1 (Decrement Iteration Counter)
    BNE IM LABEL # Go back to LABEL if Z=0 (Itertaion Counter > 0)
    INC AR AR # AR <- AR + 1 (Total will be written to 0xA5)
    ST R1 D # M[AR] <- R1 (Store Total at 0xA5)
```

Listing 2: Example Program Source Code

Example program was assembled manually and the machine code was put into the computer's memory. Note that an extra BRA 0x20 was added into the beginning to jump to the program's first command. We may think of this additional command as the **bootloader**. When a computer starts it starts executing the program in the memory starting by 0 (because PC is initially 0). ORG directive dictates where the program must reside in the memory. Since in this case it was 0x20, BRA 0x20 was put in M[0] to set PC (program counter) to 0x20 at the beginning.

```
v3.0 hex words addressed
00: 71 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20: 00 05 02 00 04 a0 16 40 05 00 46 c0 29 28 38 00
30: 80 28 46 c0 0b 00 00 00 00 00 00 00 00 00 00 00
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Listing 3: Assembled Example Program

Then the program was executed with the following input values on the computer:

```
M[A0] = 01
M[A1] = 01
```

```
M[A2] = 01
M[A3] = 01
M[A4] = 01
```

Listing 4: Example Program Input

And as expected, the computer successfully computed the sum of these inputs as 5 (1+1+1+1+1=5). Which can be seen here:

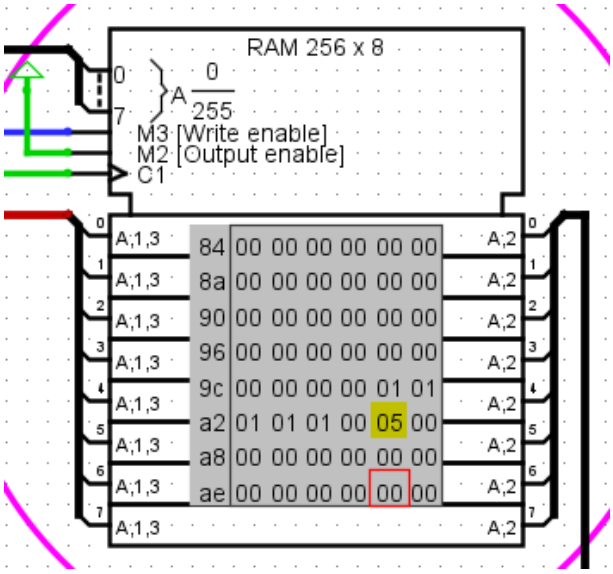

Figure 28: Result of the example program execution (highlighted part is where the result stored).

4 DISCUSSION

Control Unit is the core part of the CPU that founds a bridge between programs and the computer. Control Unit is a sequential circuitry that makes the computer do certain operations on a sequential order. A control unit may have a microprogram in a separate EEPROM to drive output signals to perform operations or it may be hardwired.

In this project, a hardwired control unit was asked to be implemented. It was done so by following a modular design approach. This design methodology was chosen because of its advantages such as being able to test each module completely separately and avoiding repetition by using modules in other modules if needed.

5 CONCLUSION

We learned how to design a working Control Unit that is capable of executing 19 different instructions. We learned how to carry out basic operations such as MOV, CALL

and BRA as well as addressing modes and how addressing works. We did not encounter with any abnormal result. All of the modules that were designed worked properly.

REFERENCES

[1] Computer Organization. Computer Organization Guidelines. *Design of basic computer*.