# Progress Report 2

Date: 11/8/2020
To: CSS 422, Autumn2020
From: Youssef Beltagy, Jasdeep Brar, Temesgen Habte, Jordan Quigtar
Subject: Implemented the BCC, BRA, and default subroutines.

---

This document is on google drive:

https://docs.google.com/document/d/1450EJKB9z8hQeOK3M9xS6A5_RJA3P8JXc37F4rcYook/edit?usp=sharing

## Work Completed

We improved the main loop so it prints the addresses of the instructions. We made the program test and ensure that the starting address is even. We implemented the default subroutine that prints the data at the addresses. The default subroutine prints the data word by word.

As for op-codes, we implemented BRA and BCC. BRA is an extension of BCC with the condition of TRUE. These two op-codes are really just one.

JSR, RTS, and NOT are currently in progress and expected to be finished by Wednesday.

This is the current output.

```
Please Enter Starting Address: 00001000
Please Enter Ending Address: 00001030
1000
1030
00001000     BRA      0000105E
00001004     BRA      0000138A
00001008     NOP
0000100A     BEQ      000010AC
0000100E     BRA      00001000
00001010     BEQ      000010F8
00001014     BRA      000012A2
00001018     BLT      000012A2
0000101C     BEQ      000012A2
00001020     DATA    $6E00
00001022     DATA    $0280
00001024     BGE      000012A2
00001028     DATA    $4840
0000102A     DATA    $5288
0000102C     DATA    $5680
0000102E     DATA    $0601
00001030     DATA    $0017
Program Terminated
```

We made a table to help us decode the op-codes. Our focus in the  table was to find signatures that identify op-codes. We didn't go into the details of how to decode the op-codes because it is too early for that. It is much more important to focus on distinguishing the op-codes right now.

## Legend:

- X - Register
- D - Data Register
- A - Address Register
- M - Mode/Opmode
- d - Direction
- S - Size

| OP-Code | Binary | Notes |
|---|---|---|
| NOP | 0100 1110 0111 0001 | All 16 bits are exact<br><br>You identify NOP by this exact signature. |
| MOVE | 00SS XXXM MMMM MXXX | The first two bits are exact. The two bits for the size. Then three for the destination register; three for the destination mode; three for the source mode; and three for the source register.<br><br>You identify MOVE by ensuring that the first two bits are zeros but the second two bits contain at least one set bit. |
| MOVEM | 0100 1d00 1SMM MXXX | d represents whether MOVEM is moving from register to memory or from memory to register. The M and X bits represent the mode and register for MOVEM.<br><br>You identify MOVEM by checking that the first five bits are 0100 1; d doesn't matter; then the next three bits are 001. |
| ADD | 1101 DDDd SSMM MXXX | ADD instructions are the only op-codes that start with 1101. But ADD shares this signature with ADDX and ADDA.<br><br>We can eliminate ADDA because it has 11 for its SS. ADD can't have both its SS set at the same time.<br><br>It doesn't seem possible to distinguish between ADD and ADDX especially when |

| | | |
|---|---|---|
| | | they are moving from register to register. We need clarification on that. |
| SUB | 1001 DDDd SSMM MXXX | SUB instructions are the only op-codes that starts with 1001. SUB is similar to ADD.<br><br>SUB has the same issue with SUBA and SUBX that ADD has with ADDA and ADDX. |
| MULS | 1100 DDD1 11MM MXXX | You identify MULs by 1100 and the following 111. |
| LEA | 0100 AAA1 11MM MXXX | AAA represents the destination register. MMMXXX represents the mode and location of the source.<br><br>You identify LEA by 0100 and the 111 that follows AAA. |
| AND | 1100 DDDd SSMM MXXX | You identify AND by the first four bits, d is zero, and that the SS bits will never have both bits set.<br><br>OR<br><br>The first four bits are 1100, d is 1, SS are not both set, and MMM is anything other than 000 and 001.<br><br>And is very complicated to identify. We have to be careful with it. |
| NOT | 0100 0110 SSMM MXXX | You identify NOT by the first 8 bits that are exact. No other op-code has the 5th-8th bits as 0110 or even has the possibility of getting it. |
| LSL | 1110 CCC1 SSR0 1DDD<br>1110 0011 11MM MXXX | CCC represents either the count of rotations when R is 0. Or the register to get the number of rotations from if R is 1.<br><br>LSL has two Signatures. It is too complicated. We need to have a better grasp of decoding before finding its signature. If we forcefully do it now, we are likely to do it incorrectly and introduce bugs. |
| ASR | 1110 CCC0 SSR0 0DDD<br>1110 0000 11MM MXXX | CCC represents either the count of rotations when R is 0. Or the register to get the number of rotations from if R is 1. |

| | | |
|---|---|---|
| | | ASR has the same issue as LSL. We need to approach them carefully and with caution. |
| BLT | 0110 1101 displacement | Identify that the first 4 bits are 0110 for a branching statement.<br><br>Identify that the next 4 bits are 1101 for a Less than conditional statement.<br><br>Identify the next 8 bits for displacement of pc counter. |
| BGE | 0110 1100 displacement | Identify that the first 4 bits are 0110 for a branching statement.<br><br>Identify that the next 4 bits are 1100 for a Greater or equal conditional statement.<br><br>Identify the next 8 bits for displacement of pc counter. |
| BEQ | 0110 0111 displacement | Identify that the first 4 bits are 0110 for a branchings statement.<br><br>Identify that the next 4 bits are 0111 for an equals conditional statement.<br><br>Identify the next 8 bits for displacement of pc counter. |
| BRA | 0110 0000 displacement | Identify that the first 4 bits are 0110 for a branching statement<br><br>Identify that the next 4 bits are 0000 for a branch always statement<br><br>Identify the next 8 bits for displacement of pc counter |
| JSR | 0100 1110 10MM MXXX | The first 10 bits are exact. Then the effective address mode, then effective address register.<br><br>You identify this JSR by the 10th bit being a 0. This separates it from JMP. |
| RTS | 0100 1110 0111 0101 | All 16 bits are exact<br><br>You identify RTS by this exact signature. |

# Problems

LEA makes an address register **point** to a memory location. We thought we understood that and made A1 point to a Buffer. We never changed A1, but only incremented it so that we don't ruin other values in memory. Yet, we still had I/O issues because we forgot to reload the buffer into A1.

We don't know whether we should disassemble ADDA and ADDX (similarly for SUB and MOVE). We don't know how to distinguish between ADD and ADDX. Especially when ADD is moving from register to register. We need clarification on that.

LSL and ASR stumped us. We decided not to include them in this table until we have a better understanding of disassembling. We don't want to introduce bugs by pushing through it.

# Work Scheduled

For the next couple of weeks, we want to work on:

- MOVE
- MOVEM
- Finish JSR
- Finish RTS
- Finish NOT
- ADD
- SUB

# Evaluation

Right now, the completed work is by Youssef. JSR, RTS, and NOT got delayed a little but should be done by Wednesday. We are developing in different branches, so the master branch only shows Youssef's work.