Final Project Report

# Team

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **IO** | **Main API’s** | **Opcodes** | **Addressing Modes** |
| **Fatima** | Input for addresses | Pseudocode for pre-planned helper methods | NOP  SUB  AND  ASL  BEQ | * Absolute Long Address * Absolute Word Address |
| **Zhijun** | Outputting the disassembled code |  | * MOVE * MULS * NOT * ASR * JSR | * Address Indirect with Post incrementing * Address Register Indirect with Pre decrementing |
| **Deepali** |  | Pseudocode for the Main Loop function | * MOVEM * DIVU * LSL * BLT * RTS | * Address Register Indirect * Immediate Addressing |
| **Ethan** |  | Implemented main loop function | * ADD * LEA * LSR * BGE * BRA | * Data Register Direct * Address Register Direct |

**How did you do source control, shared code, integration?**

We all worked and shared our code using Github and Git through VSCode. We obviously would code on Easy68k, then we would copy and paste our code or changes, place them in VSCode, then push it to the github repo. From the repo we would pull any code updates.

# 

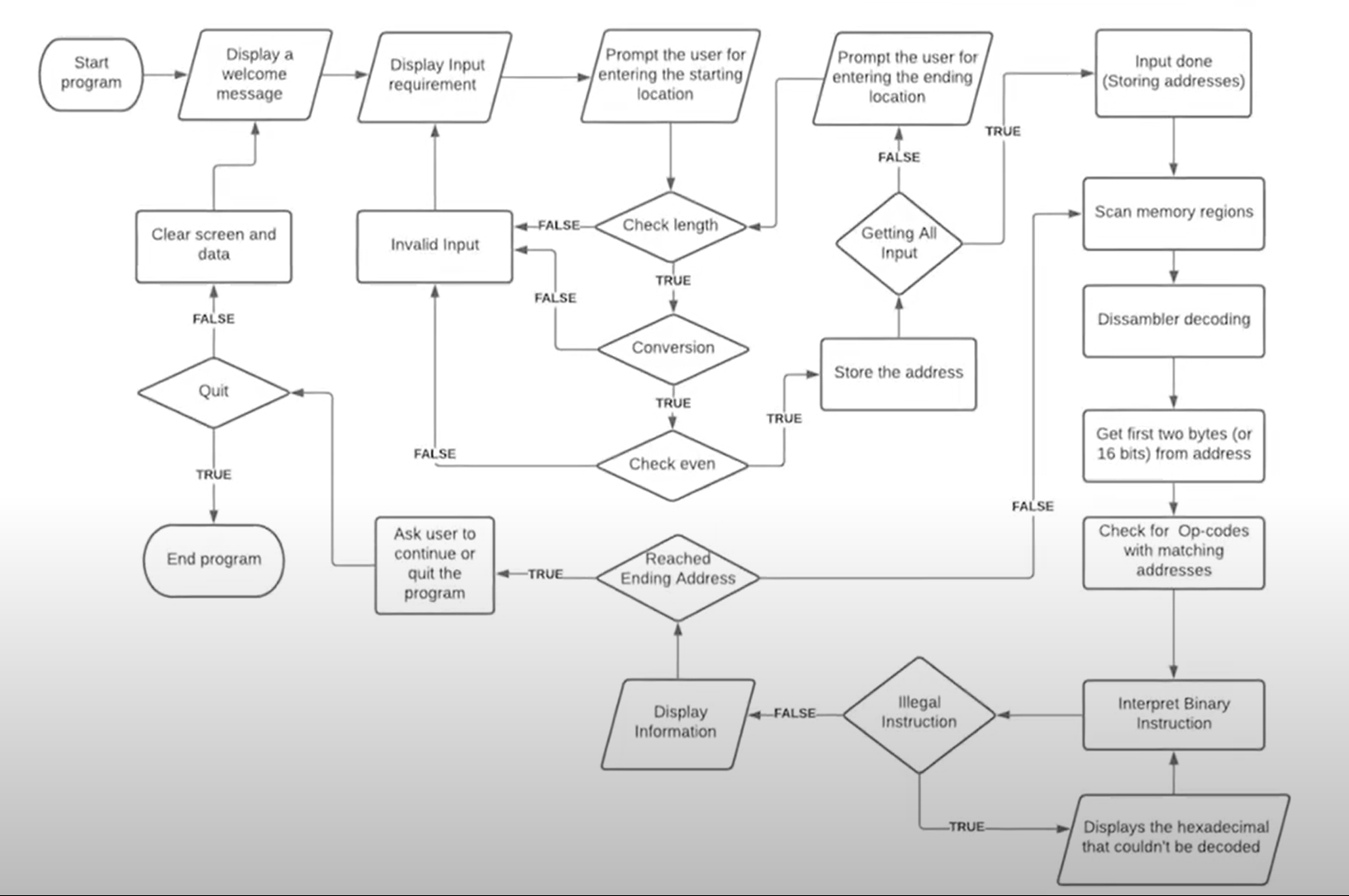
Description

The program begins with simple IO subroutines that prompt the user to enter an address that is 8 digits in hex and the program will convert this to ascii to hex. Then the opcode detection begins which will mask the bits and compare the mask to the opcode bit values to branch to the correct opcode subroutines.

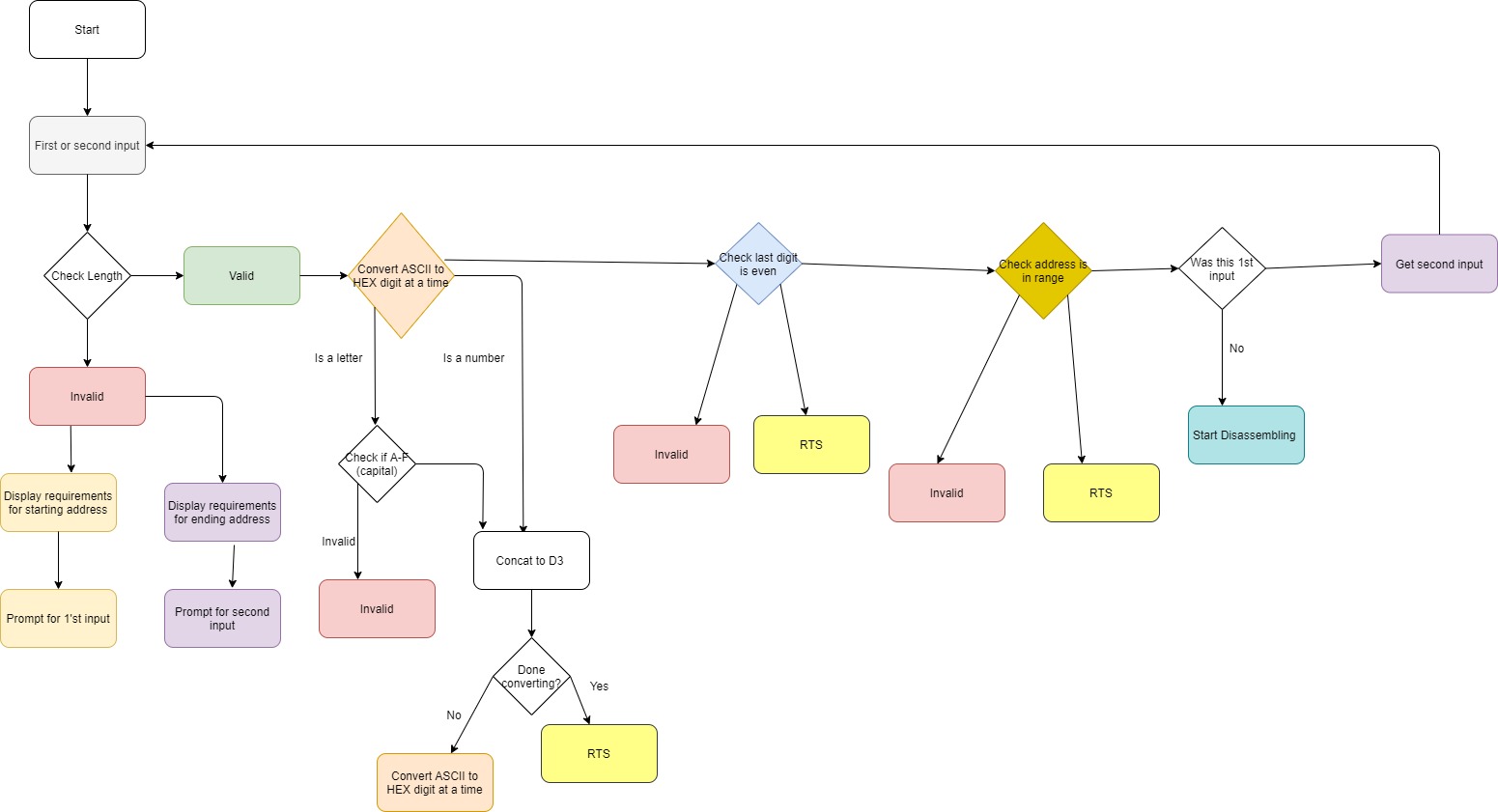
Opcode methods begin with immediate values that will print data for immediate opcodes in hex. Next is the branches which all call the same branch subroutine helpers. Every opcode is handled according to its function, if there needs to be addressing then it will either call address main, or its own custom addressing function such as movem. Output is pushed byte by byte to the 5th address register and the number of bytes to print will be incremented each time a character is added to the register. When the addressing is done it returns and the opcode will then branch to print and after this it will branch back to the start to increment the memory and find the next word to decode if there are any left. Addressing functions also include ways to output addresses that are of a higher bit count. These will shift different bits to output addresses with the output\_operand method. There are two methods for each addressing mode in each bit place so 2 for each mode.

Movem is completely custom addressing with a bit mask that will print the corresponding registers that are marked. The bit mask can be reversed depending on the mode of the opcode.

**Program Flow Chart**



**IO Flowchart**

****

# Specifications

**Registers and Uses:**

D5: stores word of current instruction

D5 holds the current word to decode in memory

D6: stores size of operation

D7 is used mostly for getBits

A0: start address

A2: end address

A3: Current memory location to search in memory

A4: Used for clearing string

A5: Memory location for output of the program / string

**Variables and constants**

hex\_start: stores the memory location that we are reading from

hex\_end: stores the end memory location for reading data

string: stores memory location for a line of instruction output

**Pseudocode**

Opcodes:

Compare A3 to A4 to check when done

Clear all registers

Move A3 to D5

Move D5 to D6

D6 contains 16 bit instruction

While (opcode not determined):

Erase bits and leave only ones related to identifying an opcode (starting from MOVE)

Compare D6 to 16 bits

if(opcode found)

Go to opcode method

If Opcode not found:

Print instruction data

For each opcode:

Append correct letters to (A5)

Determine size by ANDing bit except for size and comparing to D6

Branch to address\_main

Address main:

Move D5 to D7

while(mode not determined)

Compare 3 bits for mode (AND D7 and compare to immediate 16 bit value)

If(mode determined)

Branch mode

Return to subroutine

**Methods:**

|  |  |  |
| --- | --- | --- |
| IO | Opcodes | Addressing |
| Get\_start  Get\_end  toHex\_Start  checkStartAddr  checkAddrEven | * (20+ methods) <opcode\_name>\_op * (3) <size>\_op * getBits5\_7 * getBits8\_10 * getBits11-13 * getBits14-16 * Word\_op * Byte\_op * Long\_op   Used by MOVEM\_op:addr\_list\_to\_mem  addr\_list\_to\_mem\_pre  addr\_mem\_to\_list  addr\_mem\_to\_list\_pre | * Address\_Main: determines the addressing mode of an instruction   DirectData  DirectAddress  Immediate  ImmediateSize  ImmediateByte  ImmediateWord  ImmediateLong  IndirectAddress  PostIncrement  PreDecrement  AbsoluteWord  AbsoluteLong  General helper methods:   * output\_dn * output\_an * output\_pre * output\_post * output\_indirect   Helper methods for absolute addressing:   * print\_address\_word * print\_address\_long * print\_number * print\_letter |

Main Api helper functions:

* printHex: print the hex value of the memory location of the instruction being disassembled
* print\_data\_immediate: print immediate instructions as raw hex in memory.
* print\_output: print a single line of output
* done: finish the program and display message
* Output\_ea\_op
  + Helper methods: output\_operand, output\_address
* Output\_source\_dn: outputs data register

# Test Plan

**IO (8 tests)**

* A test for checking that input is received (1)
* A test for checking prompts print (1)
* A test for checking address is changed to hex (one for numbers, another for letter, and another for combined) (1)
* Checking that start and end addresses are even (2)
* A test for checking start address is later before even address (2)
* A test for checking branching and program flow for IO works correctly (1)

**Opcodes (30 tests)**

* A test for the part that checks for an opcode and identifies it (20 opcodes and 1 invalid data, 21 total)
* A test that sees an opcode correctly updates the buffer A5 with the correct data, determines correct size, and branches (20 opcodes + 3 sizes = 23)
* A test that determines getBit methods save the correct 3 bits to D7 (4 methods \* 2 = 8)
* Move word
* Move long
* Move with immediate
* Move with absolute
* Move with absolute long
* Move with immediate long
* Move with indirect
* Add with indirect
* Add with word, byte, long
* Add with different modes
* Lea
* Muls data
* Divu data
* And immediate
* Not byte
* Left shift
* Arithmetic shift left
* Arithmetic shift right
* Bge
* Bge backwards
* Blt
* Bglt backwards
* Beq
* Nop
* Jsr
* Movem
* etc

**Addressing (30 tests)**

* Test for data register to data register
* Absolute word to absolute word
* Data register to absolute word
* Indirect to absolute long
* Immediate word to data register
* Post Increment to data register
* Pre decrement to data
* Immediate to data
* Immediate hex to data
* Indirect to data
* Address to data
* Immediate to data
* Data register
* Absolute branch jumps with correct displacement
* Examples of the above with different opcodes
* etc.

**Test cases before professor’s tests:**

MOVE.W D3, D5

MOVE.L #$4EDABC5F, D3

MOVE.L $123, $456

MOVE.L D7, $9756

MOVE.L (A4), $A7647321

ADD.W #$02, D7

ADD.W (A2)+, D7

ADD.L -(A2), D7

ADD.W #48, D7

ADD.W #$5000, D7

ADD.B (A2), D7

ADD.W A2, D7

ADD.B D4, D2

SUB.L D3, D2

LEA string, A2

MULS.W D3, D0

DIVU.W D2, D3

AND.B #$E2, D0

NOT.B D0

LSL.L #$02, D2

ASL.B D6, D2

ASR.L #$02, D2

CMP.B D5, D1

BLT done

BGE done

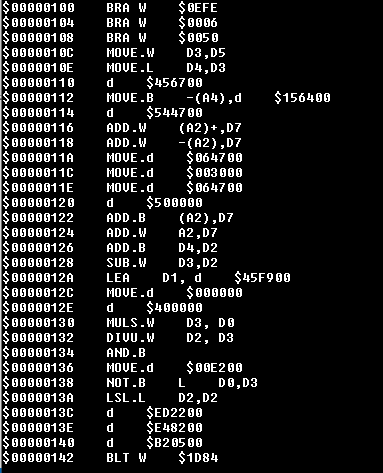
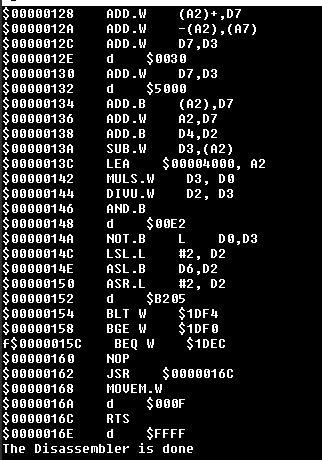
BEQ done

NOP

JSR practice

MOVEM D0-D3, (A5)

**Results:**



# Problems

All parts are believed to be done.

Time management, distribution of work, lack of communication.

# 

# Schedule

**Ethan**

**Opcodes - Time: 25 hours (started in April, completed early/mid may)**

ADD, LEA, LSR, BGE, BRA, MOVEM, LSL, BLT, ASR, JSR, ASL, BEQ, AND, MOVE

**Addressing - Time: 10 hours (started in April, completed mid may)**

* Absolute Long Address
* Absolute Word Address
* Address Indirect with Post incrementing
* Address Register Indirect with Pre decrementing
* Address Register Indirect
* Immediate Addressing
* Data Register Direct
* Address Register Direct

**Testing - Time: 50 hours integration testing (completed late may and june)**

**Fatima**

**IO Implementation - Time:**  30 hours - 2 hours per day, 5 days a week, for 3 weeks

**Design - Time:** 1 week, 10 hours

**Opcodes - Time:** 7 hours

RTS, MULS, DIVU, NOT, NOP, SUB

**Testing - Time:** 40 hours

**Deepali**

**IO implementation - Time: 10 hours**

**Reports - Time: 2 hours**

**Zhijun**

**Reports - Time: 2 hours**

**Opcodes - Time: 10 hours**

Date Deliverable

April 23 (Fri) Project Divided Among Team Members

April 30 (Fri) Start initial test program and decide APIs

May 3 (Monday) Check initial test

May 7 (Friday) Have understanding of the IO and start implementing

May 10 (Monday) Team check-in and start dissembling NOR, RTS

May 14 (Friday) Testing of progress made

May 19 (Wednesday) Team check-in

Testing results analyzed

Getting started on more opcodes and address modes

May 21 (Friday) Team check-in for incremental tests on newer code added

May 25 (Sunday) Team check-in and start testing against TestDasm.X68

May 29 (Thurs) Testing complete

May 30 (Friday) Analyze final results and gather deliverables

June 3 (Thurs) Final Project Submission