**Final CSS 422 Project : 68K Disassembler**

**Chelsea Ip, Marcela Gomez and YunJae Cho**

**Professor Woo Young Kim**

**Testing Disassembler**

* Open Team6.X68
* Using Open Data: load TEAM6TESTING.S68 file into memory
* Enter Starting Address: 00009000
* Enter Ending Address: 00009700

**OPCODE MOVEM TESTING**

* Download MOVEM\_TEST.X68
* Using Open Data: OFFICIAL\_MOVEM\_TEST.S68 file into memory
* Enter Starting Address: 00008752
* Enter Ending Address: 00009000

**INDIVIDUAL TEST LSR/LSL/ASL/ASR/RTS/ROR/CLR**

* Download MOVEM\_TEST.X68
* Using Open Data: OFFICIAL\_MOVEM\_TEST.S68 file into memory
* Enter Starting Address: 00008752
* Enter Ending Address: 00009000

**Program Specification**

“A **disassembler** scans a section of memory and attempts to convert the memory’s contents to a listing of valid assembly language instructions. Most disassemblers cannot recreate symbolic, or label information” (CSS 422 Project Overview).

Our ***I/O philosophy*** starts with asking the user for a starting address, we first read the string input using trap task #2 from the keyboard and store it in D0. Then, we would branch into the CONVERT\_STRING\_TO\_HEX subroutine where we would check if that char is in valid range (0-9 and A-F in ASCII = 30-39 and 41-46 in hex). If the character is greater than 46, we branch into INVALID\_INPUT. We subtract 30 from 39-30 to retrieve numbers 0-9 and subtract 37 from 41 to retrieve A-F. If the user attempts to enter an address larger than the starting, an invalid output will be shown.

To begin describing our ***Opcode philosophy*** -- we load the first 4 hexabits into D3 before comparing the first hexabit. For example, LEA/JSR/RTS instructions start with ‘4’. Therefore, we will branch into the root subroutine. From there, we would check for slight differences between each via the “CMP.B” function before officially printing it to the console.

**Figure 1.0** shows a table of instructions grouped by their first hexabit.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| FIRST HEXABIT | | | | | | | | | | |
| ‘D’ = ADD | ‘B’=  CMP | ‘8’= OR | ‘9’=  SUB | ‘C’=  AND | ‘1’= MOVEB | ‘3’ = MOVEAW | ‘2’ =  MOVEL/  MOVEAL | ‘6’= BCC  BGT  BLT | ‘4’ = LEA  RTS  JST/  MOVEM | ‘E’=  LSL  ASR |

To maintain reusability in our code, we simply invented mini subroutines to analyze all 12 bits behind the first four hexabits. For each 3 bits, we equate it to addresses ranging from 350 to 500. The first three bits after the 4 hexabits in front are labeled as ‘FIRST\_THREE\_BITS\_IN\_TWELVE\_BITS EQU $350, SECOND\_THREE\_BITS\_IN\_TWELVE\_BITS EQU $400, THIRD\_THREE\_BITS\_IN\_TWELVE\_BITS EQU $450 etc. Depending on what we are trying to extract from the 3 bits, we can easily use LSR/LSL to isolate specific bits for comparison. Since we pre-made subroutines, we are able to concentrate more on the logic behind instructions instead of shifting bits around.

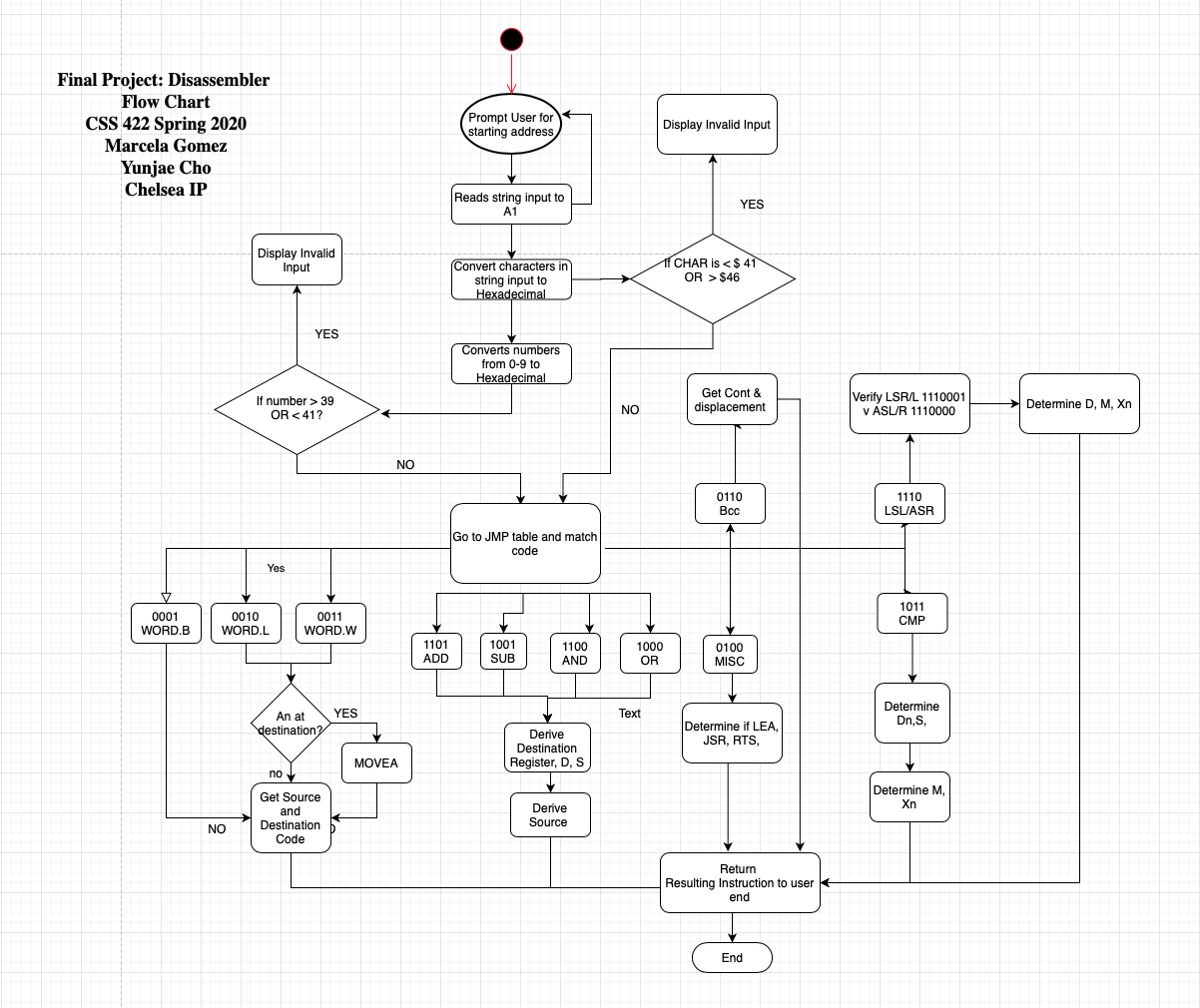
Our ***EA philosophy*** mainly relies on a few key JMP tables namely :- OUTPUT\_SOURCE/PRINT\_DESTINATION/DERIVING\_OPCODES which assists in scanning thoroughly for effective mode/effective address/destination mode/destination register. To start comparing, we move the targeted set number of bits (1st/2nd/3rd/4th) into D3 before entering with JSR OUTPUT\_SOURCE. For instructions that do not require destination registers, we are able to print the output address registers directly without interfering with bit shifting.

Any unrequired opcodes or an unexisting instruction will branch into ‘OPCODE\_IS\_INVALID”. When opcode is invalid (for example, ADD.W #$1234,D1 (0641 1234) this will be printed out as: \*DATA $WXYZ where $WXYZ is the hexadecimal number that couldn't be decoded, so that the program can continue.

Furthermore, our program only prints 20 instructions per screen-- it enables the user to press any key to assemble the next 20 instructions until there is none left. After successfully assembling the instructions, a user can enter ‘yes’ for more decoding or ‘no’ to exit the program completely. Else, we end the program with DATA $FFFF indicating that there is no more data to be disassembled.

**Overall Program Flowchart**

**Figure 1.1**  shows a higher level design of our general program.



**Special Algorithm Designs**

i) **LSL/ASR** - Writing this algorithm was pretty challenging as there are register and memory shifts for each.

**Figure 1.2**  shows the bits considered in a LSL/ASR instruction from start till end.

|  |  |  |
| --- | --- | --- |
| Are bits 6th & 7th ==11? | [YES] -> | BRA INTO MEMORY SHIFT AND ABSOLUTE ADDRESS |
|  | [NO] -> | Are bits 3rd & 4th ==00? |

|  |  |  |
| --- | --- | --- |
| Are bits 3rd & 4th == 00? | [YES] -> | BRA INTO ASR REGISTER SHIFT  OR IMMEDIATE DATA |
|  | [NO] -> | BRA into LSL REGISTER SHIFT OR IMMEDIATE DATA |

|  |  |  |
| --- | --- | --- |
| Are bits 3-5 == 111? | [YES] -> | Are bits 9-11 == 000? |
|  | [NO] -> | Check if LSL absolute address or ASR absolute address ($)  Are bits 3-5 == 111? |

|  |  |  |
| --- | --- | --- |
| Are bits 9-11 == 000? | [YES] -> | PRINT ASR MEMORY REGISTER |
|  | [NO] -> | PRINT LSL MEMORY REGISTER |

|  |  |  |
| --- | --- | --- |
| Are bits 3-5 == 111? | [YES] -> | PRINT ASR ABSOLUTE ADDRESS |
|  | [NO] -> | PRINT LSL ABSOLUTE ADDRESS |

|  |  |  |
| --- | --- | --- |
| Are bits 3-5 == 000? | [YES] -> | PRINT ASR REGISTER SHIFT |
|  | [NO] -> | PRINT ASR IMMEDIATE DATA |

|  |  |  |
| --- | --- | --- |
| Are bits 3 to 5 == 001? | [YES] -> | PRINT LSL REGISTER SHIFT |
|  | [NO] -> | PRINT LSL IMMEDIATE DATA |

**What we worked on**

INSTRUCTIONS

**Figure 1.3**  shows all 14 required instructions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| MOVE | MOVEA | MOVEM | AND | SUB |
| LEA | AND | OR | LSL | ASR |
| CMP | BCC, BGE, BLE | JSR | RTS |  |

Effective Addressing Modes:

**Figure 1.4**  shows a higher level design of our general program.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Register | Address Register Direct (An) | Address Register Indirect | Immediate Data (#$) | Address Register Indirect with Post incrementing (An)+ |
|  |  | Absolute Word Address | Absolute Long Address | Address Register Indirect with Pre decrementing |

Miscellaneous Functions :

**Figure 1.5**  shows important subroutines our program logic mainly uses.

|  |  |  |
| --- | --- | --- |
| DERIVING\_OPCODE | ANALYZE\_TWELVE\_BITS | GRAP\_LAST\_TWO\_BITS\_IN\_OPMODE |
| OUTPUT\_SOURCE | PRINT\_DESTINATION | DN\_IS\_EITHER\_SOURCE\_OR\_MODE |
| OPCODE\_IS\_INVALID | START\_TO\_HANDLE\_OPCODE | INVALID\_INPUT |

**How we collaborated**

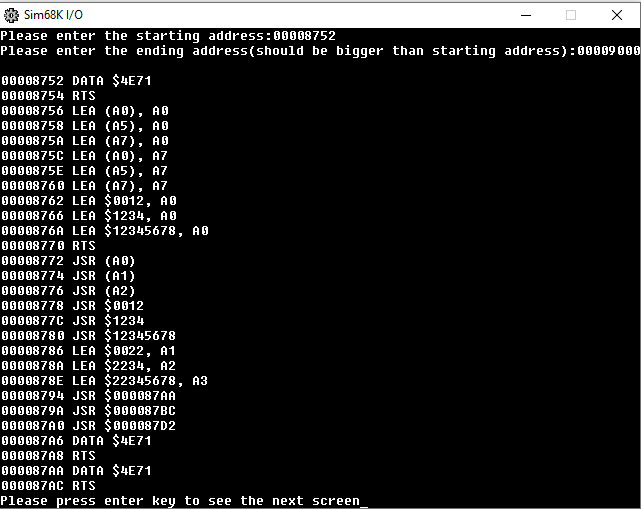
Our team chose to follow agile methodologies to complete the project. Before diving into implementation, we held virtual video calls to conceptualize how we will process hexadecimal to binary and vice versa until we were able to finish one instruction. After gaining an understanding of what was ahead, we started assigning individual tasks to each team member. Ideally, our goal was dividing instructions into 5-5-4 proportions.

The majority of our collaborative efforts was through communication via discord group discussion and github submissions. Due to the pandemic, face to face meetings took place over zoom. Continual discussion over google drive documents and discord channel.

**Figure 1.6**  shows the work breakdown structure over the course of four weeks.

|  |  |  |  |
| --- | --- | --- | --- |
| **Meeting Dates** | **Project Iteration** | **Description** | **Goal** |
| 5/7/2020   * Zoom * discord | * Welcome Screen * Jump Tables * Input & updates * Input conversion | Construct the infrastructure around which everything else will be built.  Testing | Start & end address handling  Input handling |
| 5/18/2020   * Zoom * discord | * Print & disassembling * ADD,AND, CMP, MOVE, BCC | Fine tune input conversions and added instructions, continual testing | Progress onto instruction focus |
| 5/25/2020   * Zoom * discord | * LEA, OR, RTS, JSR, SUB, MOVE, MOVEA, LSR/ASL * TEST SCREEN | Continue adding instructions testing and integrating into main file | Build and test more instructions then add to main file |
| 6/5/2020   * Zoom * discord | * MOVEM * Final Instruction code added to main file * Testing Plan * Complete reports | Bring all final instructions to the main file for testing and fine tuning.  Test and adjust code as it is added to the main report. | Finish all tasks and submit the report with findings. |

**Test Plan**

We tested our program using “OurOwnTesting.X” file. First, the file contained only a small amount of instructions, but as our program grew, we increased the amount of instructions gradually. When we were testing, we ran our disassembler program and clicked ‘open\_data’ and selected our OurOwnTesting.X to load data to our program starting from the given address. And after running our program, we checked if output matches the data loaded in the machine from that starting address. We first tested with only required opcodes, and then also added non-required opcodes to check if it prints out DATA $XYZZ correctly. For the coding standards, our team constantly did “iteration testing” to make sure everything works as we go. We also organized our program in a way that ‘DERIVING\_OPCODE’ subroutine is like a MAIN function in other languages. In this DERIVING\_OPCODE subroutine, we dealt with 14 different opcodes using CMP instruction. 

**Exception Report**

When we were testing, we found that when the program is decoding the instructions that were not required, it will just crash. So we added the ‘OPCODE\_IS\_INVALID’ subroutine to handle this case so that it will print as DATA $ABCD and continue with the next instructions. After several attempts over 3 days to construct a proper bit mask handler for the movem operation an alternative method was created to get the address printed. It uses the hexabits after the instruction address to process the address registers. There are some “non-required” opcodes that are printing remaining hexabits onto other addresses. The program still assembles other instructions despite this, there is no crash.

Due to version control issues, bringing MOVEM instruction into the main file conflicted with opcode\_invalid directions for all other instructions. A separate file is included for testing the output.

**TASKS ORGANIZATION**

**Yun Jae Cho**

* Added Subroutines: ‘ANALYZE\_TWELVE\_BITS’,‘PRINT\_COMMON\_SIZE’,‘PRINT\_MEMORY\_ADDRESS’,’OUPUT\_SOURCE’, ‘OUPUT\_DESTINATION’ to be able to analyze opcde 16 bits and print out <EA>s.
* Added subroutine ‘IS\_NEW\_SCREEN\_NEEDED’ to show one screen of data at a time, hitting the ENTER key will display the next screen of data.
* When program has illegal instruction, it will print out as memory address DATA $WXYZ using OPCODE\_IS\_INVALID
* When the program completes, it will prompt the user to restart to disassemble or quit.
* Handling invalid address/converting from hex to ASCII and vice versa.
* MOVE
* MOVEA
* ADD
* SUB
* AND
* OR
* CMP
* BCC/BGE/BLE
* Testing plan, exception report in final report

Total Percentage: 40%

**Marcela**

* MOVE: SOURCE, DESTINATION, OPCODE
* BCC
* BGE
* BLE
* MOVEM: OPCODE INSTRUCTION, INSTRUCTION SIZE, MEMORY ADDRESS
  + - * Weekly Report1: Instruction Table
      * Weekly Report3:Flow Chart in week 3 report
      * Ongoing Testing

Total Percentage: 5 tasks total 23-25%

**Chelsea Ip**

* + I/O - Asking for starting address from user
  + Progress report 1,2,3, 4 + flowchart
  + LEA
    - LEA <ea>, An
  + RTS
  + JSR
    - JSR <ea>
  + LSL
    - LSd Dx,Dy
    - LSd # <data>, Dy
    - LSd <ea>
  + ASR
    - ASd Dx,Dy
    - ASd # <data>, Dy
    - ASd <ea>
  + 10% of MOVEM
  + Dealing with LSR + ASL (unrequired instructions) and printing them in the form of DATA $XXXX.
  + Testing
  + 80% of the final report including the general program, special algorithms, and overall .

Total Percentage : (5/14) + 80% Progress Report = 35.7%

**REFERENCES**

[CSS 422 Project Overview](https://canvas.uw.edu/courses/1387438/files/63521638?module_item_id=10488691)

<http://mrjester.hapisan.com/04_MC68/> <http://goldencrystal.free.fr/M68kOpcodes-v2.3.pdf>