Disassembler Project

By: Team Bazinga (Hyeon Hong, Jewel Chu, Kevin Kang)

Program Description

Program Algorithm

1. Main

1. When the program starts, it first displays the welcome message.
2. The call is made to I/O subroutine, which processes the user input for the start address.
3. After I/O subroutine is returned, the memory variable (start_addr) is set.
4. The memory variable (which_add) is flagged to indicate that it is now the turn for the end address.
5. The call is made to I/O subroutine, which processes the user input for the end address.
6. After I/O subroutine is returned, the memory variable (end_addr) is set.
7. The call is made to Op-Code subroutine, which writes one line of instruction to the buffer.
8. Depending on the instruction, EA subroutine gets called during the span of Op-Code subroutine, and it writes EA to the buffer.
9. After Op-Code subroutine is returned, the buffer is filled with one line of instruction and its operands.
10. The buffer is printed to console.
11. Check if the memory pointer has reached the end address.
12. If not, loop through the step 7-11 until it reaches the end address.
13. In every 25 calls to Op-Code subroutine, the message for pause (i.e., page break) is displayed.
14. After the program reaches the end address, it asks the user to continue or not.
15. If continue, go through the step 1 - 14 again. Otherwise the program terminates.

2. I/O subroutine

1. Save the working registers on stack.
2. Check for the flag to see if it's the start/end address.
3. Prompt the user for entering the start/end address depending on the flag.
4. Read the input from the user and save it into 'reader' memory variable.
5. Check for the length of the input characters.
6. If it's more than 8, display the message stating it's too long, then go back to the beginning to start over.
7. If it's less than 8, fill the 0s in the front of characters to make it 8 characters.

8. Check if the input characters are valid or not. (that is, hexadecimal or not)
9. If there is a character that is non-hexadecimal, display the message stating it's invalid, then go back to the beginning to start over.
10. Convert the input from the ascii-hex to the hex value and write it into the 'start_addr'/'end_addr' memory variable.
11. Check if the address is even number or not.
12. If the address is odd and it's the start address, convert it to an even number and display the message stating the program has changed the address boundary.
13. Check for the memory range and if it's not in the range of 0 - FFFFFF (16MB), display the message stating it's out of range, then go back to the beginning to start over.
14. If the flag is set as the end address, check if the end address is greater or equal to the start address.
15. If not, display the message stating it's illegal to have a smaller end address, then go back to the beginning to start over.
16. Restore the working registers from stack and return from subroutine.

3. Op-Code subroutine

1. Save the working registers on stack.
2. Read the memory location (Word size) that will be decoded.
3. Write this memory address to the buffer.
4. Apply the appropriate mask pattern (ANDing) to the word in order to remove the non op-code bits in the word.
5. Read the first word in current row of the look-up table and compare it with the filtered op-code bits.
5.1.1. If it's not the same word, the pointer moves to next row in the table.
5.1.2. Check if the program has reached the next case in the look-up table. If not, it goes back to the beginning of the current loop. If it has reached, it goes out of current loop to compare with the next case.
5.2. If it's the match, the pointer moves to the next column in the table.
6. Read the next 4 words to the column direction and write them to the buffer, which are the characters of instruction name.
7. The pointer moves to the next column to read the syntax pattern number.
8. Go to the branch that matches the syntax pattern number.
9. Write to the buffer the operands in a proper syntax form depending on the syntax pattern number.
10. If E.A. operand exists in the instruction, the call is made to EA subroutine, which writes EA to the buffer and return the memory pointer that's pointing at the next instruction.
11. Restore the working registers from stack

4. EA subroutine

1. Save the working registers on stack.
2. Check if the instruction is MOVE or not.

2.1. If it's non-MOVE instruction, parse the EA bits into 1 set of mode and register number bits.
2.2. If it's MOVE instruction, parse the EA bits into 2 sets of mode and register number bits, since there will be 2 EAs.
3. Check if the mode is valid for the specified instruction, by comparing it with the value in the EA table.
3.1. If the mode is valid, go to the next step.
3.2. If the mode is not valid, set the bad_flag to 1 and get out of EA subroutine.
4. Before reading additional memory for values (such as immediate values), check to see if the memory was already processed by Op-Code subroutine. (This is implemented by memory variable 'imme_flag')
5. If there is additional memory that was already processed, then skip to next memory location.
6. Process the mode and register bits and write the values to the buffer.
7. Restore the working registers from stack.

Borrowed or Inspired by the Alan Clements Book

1. Hex - ASCII conversion

The idea of Hex and ASCII conversion routine is from Clements' book (page 145) where it explains the algorithmic approach to convert 4-bit hex digit (HexVal) into ASCII char (HexChar).

```
HexChar := HexVal + $30;
If HexChar > $39 THEN HexChar := HexChar + $07
```

2. DIVU & SWAP instruction

The technique we used for implementing the page break (25 lines in our case) is to use DIVU instruction and check for the remainder. The DIVU (divide) idea is explained in Clements' book page 185 and the routine we borrowed is in page 346.

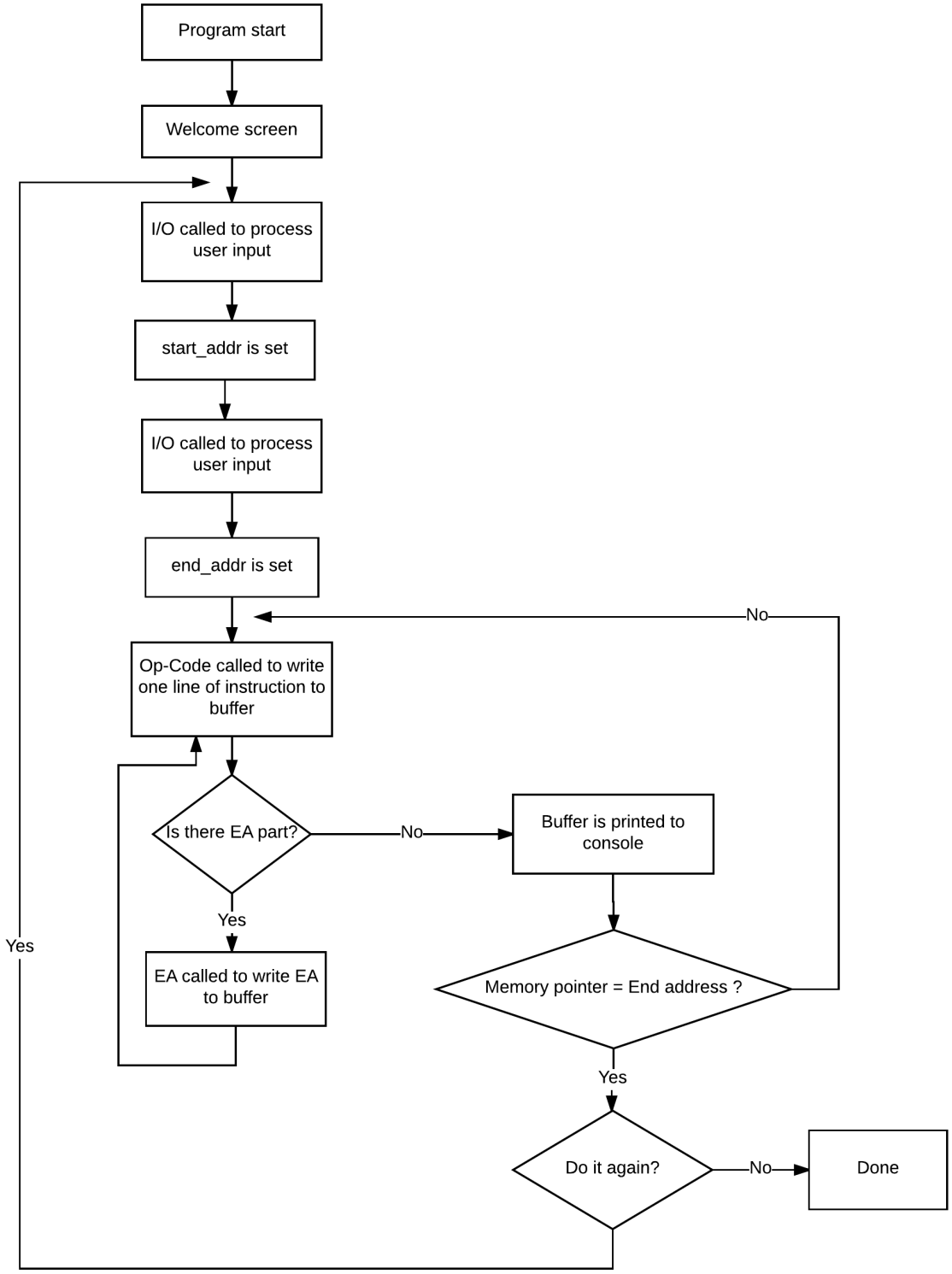
```
DIVU        #25, D2     * Calculate quotient and remainder
SWAP        D2          * move remainder to the lower-order
ANDI.L      #$00FF, D2 * Remove upper-order word (quotient)
```

3. Macros and Labels

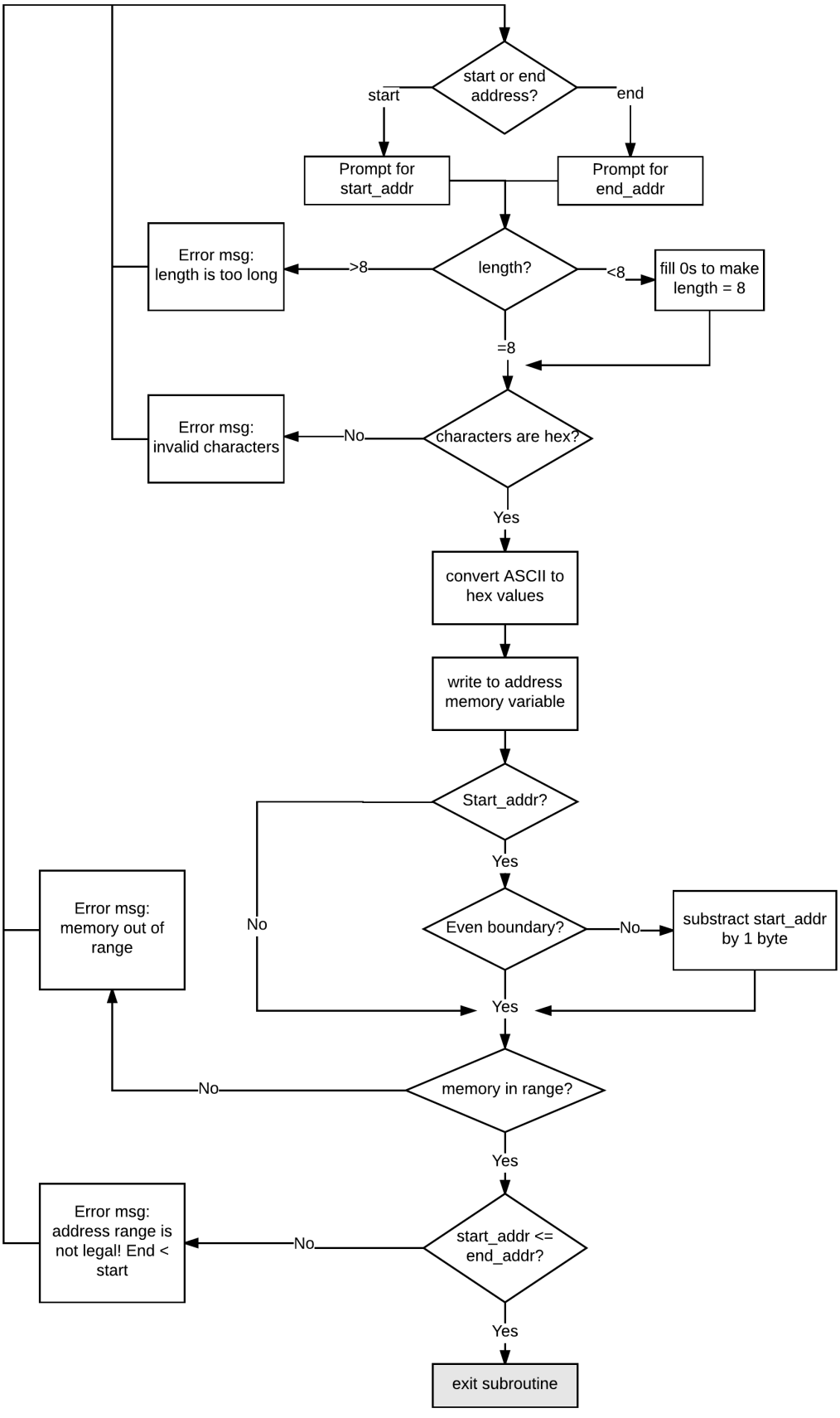
Macro routines are called multiple times in this program. In writing macro routines we often used labels inside the macro. Since the macro itself is replaced by the code block when it is called, the same label could appear in multiple places in the code, which would cause errors. Macro and label usage is introduced in Clements' book page 343 where it shows how the macro assembler automatically renames the label each time the macro is called. The label in macro should be in the form of a user-defined character string with **\@** such as `SUM\@` and the label will be translated into the form SUM.nnn where .nnn represents a 3 digit decimal number automatically chosen by the assembler.

Flowcharts

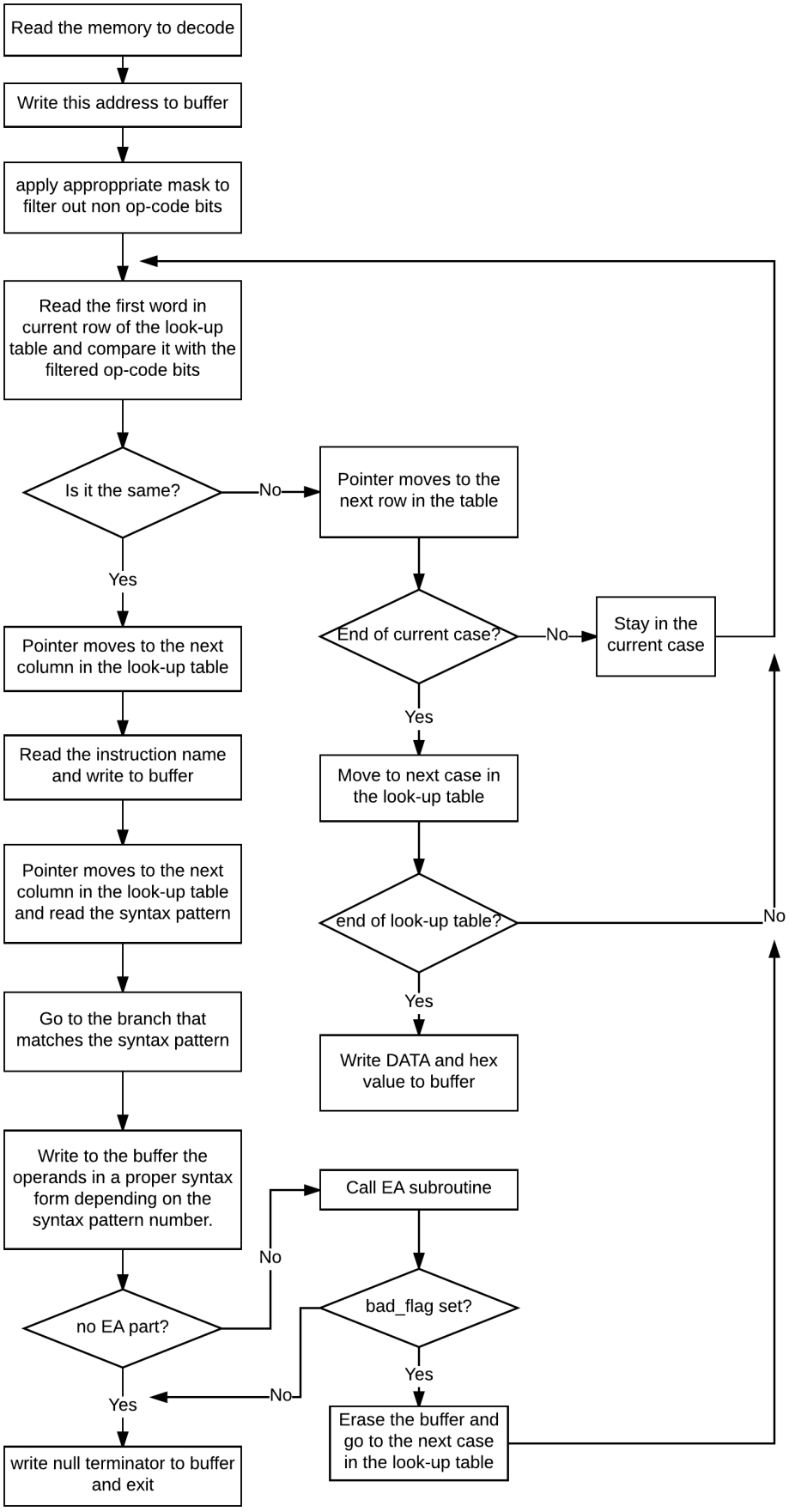
1. Main



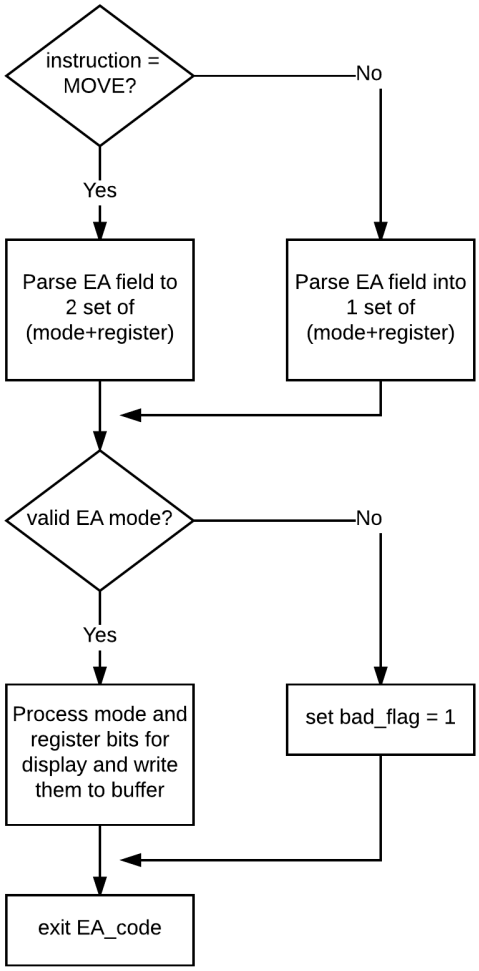
2. I/O subroutine



3. Op-Code subroutine



4. EA subroutine



Design Features

- I/O subroutine handles the operation of taking the user input and saving it to memory variable.

- Op-Code handles all the operations of writing to buffer.

- Op-Code also handles the non-EA operands.

- EA subroutine is called inside the Op-Code subroutine. (multi-layered)

- Op-Code subroutine writes into the buffer the ASCII-hex values.

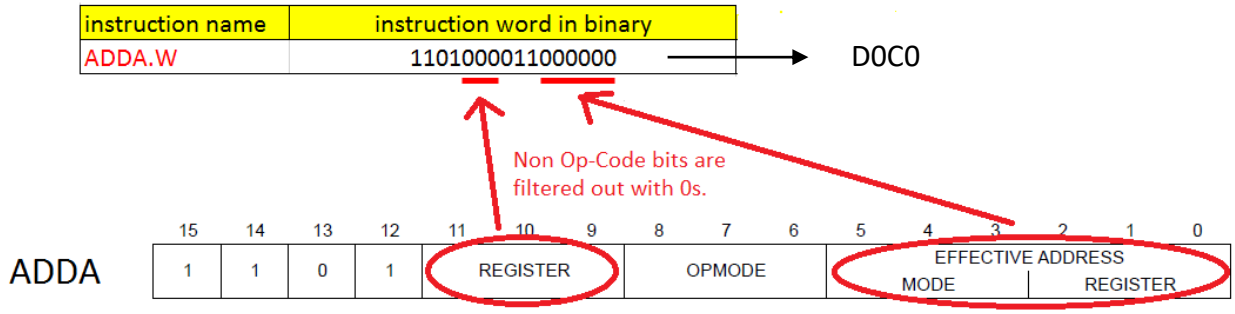
- All immediate values are printed to console as the form of hexadecimal only.

- A lot of MACROs are utilized in our program to reuse blocks of code.

- Two tables are utilized in this program: 1. Look-up table for instruction, 2. EA table

- The look-up table is carefully designed in a way that instructions of similar patterns are grouped together. The instructions are categorized into seven cases in the look-up table depending on the location of op-code, E.A., and register numbers. This approach minimizes the total volume of look-up table and hence, decreases the search time.

- Look-up table implementation




```
DC.W    $D180,'ADD.L   ',$0001,$0000,$0178
DC.W    $D0C0,'ADDA.W  ',$0002,$0000,$0180
DC.W    $D1C0,'ADDA.L  ',$0002,$0001,$0188
```

Instruction name

Syntax pattern number

Flag for Immediate value

Displacement to the beginning address of EA table

- EA table implementation

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Designated Number
for each mode in EA_Table

➔ 0
➔ 1
➔ 2
➔ 3
➔ 4
➔ 5
➔ 6

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011

➔ 7
➔ 8
➔ 11
➔ 9
➔ 10

ADDI

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

instruction name	invalid mode numbers				
ADDI.W	01	11	09	10	15

The number 15 indicates there's no more invalid modes.

Specification

- The program is loaded into memory at $1000 and display a welcome screen

- Prompt user for the memory region they wish to decode (starting address and ending address)

- Scan the memory region and output the memory addresses of the instructions and the assembly language instructions contained in that region to the console. The decoded instruction will be displayed in this format line by line:
  [Memory location] [Opcode] [Operand]

- Supported Opcode instructions:
  ADD, ADDA, ADDI, AND, ANDI, ASL, ASR, BSR
  CLR, CMP, CMPA, CMPI, EOR, EORI, EXG
  JMP, JSR, LEA, LSR,LSL, MOVE, MOVEA, MOVEM
  NEG, NOP, NOT, OR, ORI, ROL, ROR, RTS
  SUB, SUBA, SUBI, SWAP

- Supported Effective Addressing modes:
  - Data Register Direct
  - Address Register Direct
  - Address Register Indirect
  - Immediate Data
  - Absolute Long Address
  - Absolute Word Address
  - Address Register Indirect with Post incrementing
  - Address Register Indirect with Pre decrementing
  - Program counter with displacement
  - Address register indirect with index

- Illegal instruction (ie, data) and illegal addressing mode detection. If there are instructions or addressing mode that cannot be decoded, they will be displayed as:
  [Memory location] DATA $WXYZ

Test Plan

1. Write all possible combinations of instructions for the project using Excel and formulas. The instructions are listed as one instruction per column. On the left side of the instructions, all 12 op-code formats are listed and replicated across the instructions to generate an entire spectrum of instructions for testing the project. Not all combinations would apply to every instruction, but that is expected.
2. Copy and paste instructions into EASy68K Assembler's test instructions area. Eliminate the instructions that generate errors in the Assembler.
3. Run the compiled program, and compare the output of the decompiled instructions against the original instructions.
4. For instructions showing discrepancies, prepare a Word doc per instruction that has the instructions used along with screenshots of errors. Taking screenshots is the only way to capture the errors since cut-and-paste functionality is not available from the output window of the decompiler.
5. Save the Word documents in a Debug folder of the version controlled repository, and notify EA and OP-code programmers to review and correct.
6. Once an updated version of the decompiler is made available, test that showed errors
7. Also, check for any regression bugs by testing the instructions that had passed previously
8. Repeat steps 2 through 7 until all errors are eliminated.

Excerpts from the Excel spreadsheet for the test commands

CMPI	EOR	EORI
CMPI.B #$33, D0	EOR.B D3, D0	EORI.B #$33, D0
CMPI.W #$33, D0	EOR.W D3, D0	EORI.W #$33, D0
CMPI.L #$33, D0	EOR.L D3, D0	EORI.L #$33, D0
CMPI.B #$33, A0	EOR.B D3, A0	EORI.B #$33, A0
CMPI.W #$33, A0	EOR.W D3, A0	EORI.W #$33, A0
CMPI.L #$33, A0	EOR.L D3, A0	EORI.L #$33, A0
CMPI.B #$33, (A0)	EOR.B D3, (A0)	EORI.B #$33, (A0)
CMPI.W #$33, (A0)	EOR.W D3, (A0)	EORI.W #$33, (A0)
CMPI.L #$33, (A0)	EOR.L D3, (A0)	EORI.L #$33, (A0)
CMPI.B #$33, (A0)+	EOR.B D3, (A0)+	EORI.B #$33, (A0)+
CMPI.W #$33, (A0)+	EOR.W D3, (A0)+	EORI.W #$33, (A0)+
CMPI.L #$33, (A0)+	EOR.L D3, (A0)+	EORI.L #$33, (A0)+
CMPI.B #$33, -(A0)	EOR.B D3, -(A0)	EORI.B #$33, -(A0)
CMPI.W #$33, -(A0)	EOR.W D3, -(A0)	EORI.W #$33, -(A0)
CMPI.L #$33, -(A0)	EOR.L D3, -(A0)	EORI.L #$33, -(A0)

Exception Report

- The instruction MOVEM has a shortened form for displaying operands, such as A1-A3 instead of A1/A2/A3, but we couldn't implement this feature to the program.

- We found that the data immediately following the unrecognized instruction was frequently misinterpreted as the instruction ORI. We tried to come up with the solution for this issue, but due to time constraints we decided to leave out the instruction ORI to improve the overall performance of the program.

Team Assignments

Jewel (Yun-Hsuan) Chu
- I/O design and implementation
- Project Report write-up
- Managing the team schedule and milestone

Hyeon Hong
- Op-Code & EA subroutine design and implementation
- Overall code structure, system logic
- Code integration, unit testing

Kevin Kang
- Testing plans & executions for every possible combination of instructions
- Discovering bugs and other exceptions in the program
- Regression testing