CSS 422 Final Project Documentation

To	:	CSS 422, Spring 2017
From	:	WeMissJava Group
Date	:	29 May 2017
Subject	:	MC68000 Microprocessor Disassembler program Documentation

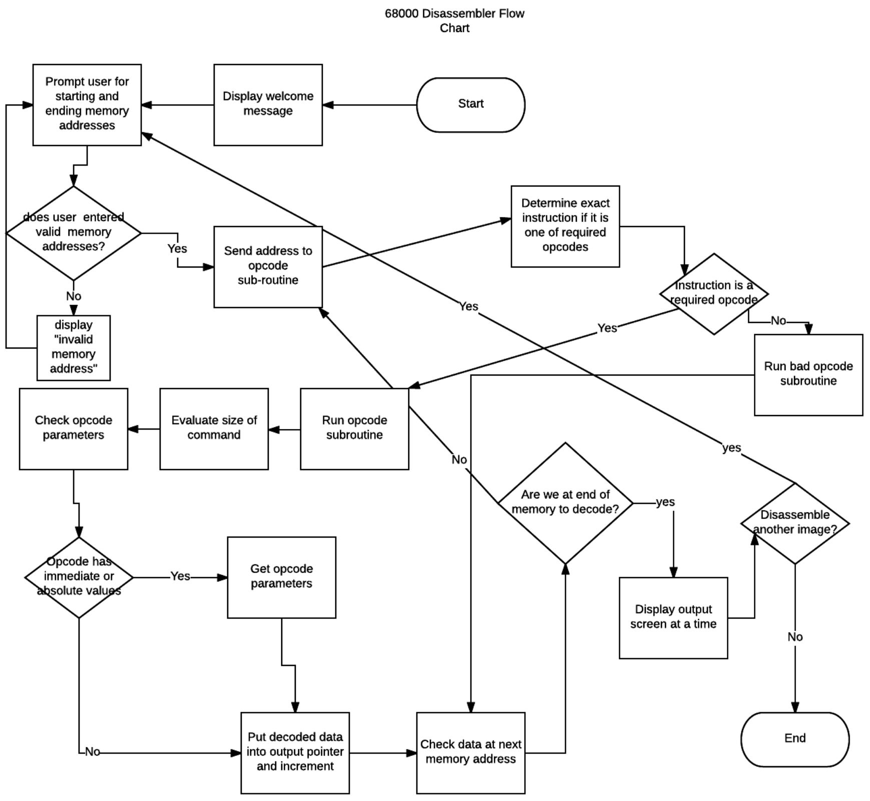
1 Program Description

The disassembler program converts a memory image of instructions and data back to 68000 assembly language and output the disassembled code to the display. It will only disassemble the list of opcodes and effective addressing modes listed under the specification section.

1.1 Program Flow

The program starts by displaying a welcome message, then it prompts the user to enter starting and ending memory addresses. If user inputs valid address, it is sent to the op code subroutine to determine if instruction is one of the required opcodes. If user does not input a valid memory address then it will print error message stating that the address is invalid.

If the instruction is one of the required opcodes, then the opcode subroutine runs to evaluate size of the command and check for the opcode parameters. If instruction is not in the list of required opcodes, then the bad opcode subroutine runs and checks the data at the next memory address. The opcode we want to decode then will be checked for absolute or immediate values, in which case will get the opcode data, decode it and put it into the output pointer and increment the pointer. Program then checks the next memory address to decode. Program checks if the memory is completely decoded in which case it displays output on screen one at a time. If there is more to decode in memory then program sends it to the opcode subroutine and the cycle continues until there is no more instruction to decode. Program terminates when the user enters 'n' for no more addresses.


68000 Disassembler Flow Chart
Start
Display welcome message
Prompt user for starting and ending memory addresses
does user entered valid memory addresses?
Yes
No
display "invalid memory address"
Send address to opcode sub-routine
Determine exact instruction if it is one of required opcodes
Instruction is a required opcode
Yes
No
Run bad opcode subroutine
Run opcode subroutine
Evaluate size of command
Check opcode parameters
Opcode has immediate or absolute values
Yes
No
Get opcode parameters
Put decoded data into output pointer and increment
Check data at next memory address
Are we at end of memory to decode?
yes
No
Display output screen at a time
Disassemble another image?
yes
No
End

1.2 Design Philosophy

The CmdIdentify file contains the subroutine for identifying which command a single opcode is, then calls the subroutine specific for that command, which will read more of the input if necessary (for immediate values and addresses). This lets us have a separate file for each command, partition the work between us and measure our progress. We noticed some repetition in how each command worked, so we wrote re-usable subroutines for things like printing the size, source, destination, a single number, etc. Encapsulation and abstraction are pretty basic concepts and not exactly advanced software design, but I (James) am pretty proud of implementing them this well in assembly.

2 Specification

The inverse assembler (disassembler) program converts a memory image of instructions and data back to 68000 assembly language and output the disassembled code to the display. It will not disassemble all of the instructions and addressing modes in 68000. The list of instructions and addressing modes the disassembler program converts to recognizes are given below

Effective Addressing Modes:

1. Data Register Direct
2. Address Register Direct
3. Address Register Indirect
4. Immediate Data

5. Address Register Indirect with Post-incrementing
6. Address Register Indirect with Pre-decrementing
7. Absolute Long Address
8. Absolute Word Address

Instructions:

1. NOP
2. MOVE
3. MOVEA
4. MOVEQ
5. MOVEM
6. ADD
7. ADDA
8. ADDI
9. ADDQ
10. SUB
11. SUBI
12. MULS
13. DIVU
14. LEA
15. AND
16. OR
17. LSL
18. LSR
19. ASR
20. ASL
21. ROL
22. ROR
23. Bcc (BCC, BLT, BGE)
24. JSR
25. RTS
26. BRA

In addition the program should be written with the following specifications:

1. The simulators text I/O function should be used for all I/O needs. Trap function 15 and tasks with ID 0-14 of the trap function are allowed. Trap function higher than trap 15 are not allowed.
2. The program should be written from the start in 68000 assembly language. Do not write it in C or C++ and then cross-compile it to 68000 code.
3. The program should be ORG'ed at $1000.
4. At startup, the program should display welcome message and then prompt the User for the starting and ending location in hexadecimal format of the code to be disassembled
5. The program need to clearly specify the expected input format for user inputs.
6. The program should scan the memory region and output the memory addresses of the instructions and the assembly language instructions contained in that region to the display. The program should be able to actually disassemble its own to the display!
7. The display should show one screen of data at a time, hitting the ENTER key should display the next screen of information.
8. The program should be able to realize when it has an illegal instruction (i.e, data), and be able to deal with it until it can find instructions again to decode. Instructions that cannot be decoded, either because they do not disassemble as op codes or because you aren't able to decode them should be displayed as:

   ```
   1000    DATA    $WXYZ
   ```

   where `$WXYZ` is the hexadecimal number that couldn't be decoded. The program should not crash because it can't decode an instruction. it is perfectly legal to have data and instructions interspersed, so it is very possible that you will hit data, and not an instruction.
9. Address displacements or offsets should be properly displayed as the address of the branch and display that value. It's the absolute address value, not the displacement value. For example:

   ```
   1000    BRA 993    * Branch to address 993
   ```

10. The program performs a line by line disassembly, displaying the following columns:

    a. Memory location   b. Op-code   c. Operand

11. When it completes the disassembly, the program prompts the user to disassemble another memory image, or prompt the user to quit.

3 Test Plan

A test main file was created that took input starting and ending addresses and outputted the decoded commands to the console. A test code file that was ORGed at $7000 got included into our main with operations that we wanted to test written into it. Different sizes, effective addressing, and variations of the operation were tested to check basic functionality of the decoding. Coding standards included following naming conventions so that code could be easily integrated, as we used a multiple file format for each separate opcode command. Documentation was provided for the helper functions that printed out different components of the opcode commands. That made working on separate github branches possible while being able to test each others code at the same time. We debugged each function as we moved along to lessen the workload after all the opcodes were integrated and final testing was done.

Final checking was done to make sure the disassembler meets assignment requirements:

1. Program cannot assemble into listing file
2. Program crashes on non-required opcode or EA
3. Program crashes on each required opcode or EA
4. Each required op-code or EA which is not properly disassembled
5. Program does not print the address of each instruction
6. Program does not print results as one screen at a time
7. Program does not have options to restart or finish

The final test phase was done using the following test file: Test code file

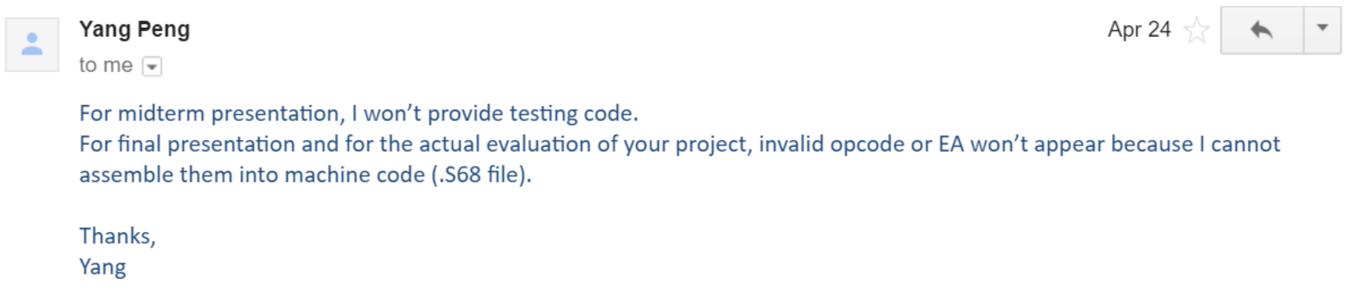
Operations were tested for different source and destination EA, such as different size absolute addressing and immediate values, address register dereferencing, data registers, and address registers. Boundaries got checked for ADDQ and bit operators that rotated an immediate value number of times. Branching forward and backwards was checked for all branch operations. Movem and bit shifts were tested for all different formats. If a bug was found, the person who was responsible for the opcode used the test information to fix their code, and then the fixed version was integrated into their branch.

4 Exception Report

We decided that instead of printing the text as we go, we should generate a large string of the entire output in memory, than print it once it's done. Our rationale behind this was that "printing" a character would just be printing a byte onto a stack with `MOVE.B #'R',OUT_CURR_A`, which is one line, and more human-readable than a trap task. But I'm not incredibly confident in this decision, it takes up a decent chunk of memory, and on more careful inspection it turns out we don't print the entire output at once.

Due to getting clarification via e-mail, extensive error-checking involving invalid instruction commands are not being handled. For example, the following code produces some decoded instructions that are not actually valid commands. Specifically, the size bits for `MOVEA` are not checked for size bits=00. This is an invalid size and our disassembler still decodes it as a valid `MOVEA` operation.

```
DATA_test DC.B 'MOVE', 0
DATA_test1 DC.B 'I AM DATA', 0
DATA_test2 DC.B 'TESTCODECHECKOKHAHA', 0
```


Yang Peng
to me
Apr 24

For midterm presentation, I won't provide testing code.
For final presentation and for the actual evaluation of your project, invalid opcode or EA won't appear because I cannot assemble them into machine code (.S68 file).

Thanks,
Yang

5 Team assignments and report

Opcodes were broken up as follows:

Bisrat:	LEA, BCC, BLT, BGE, JSR, RTS, BRA
Stephen:	MOVE (old version), MULS, DIVU, LSL, LSR, ASR, ASL, ROL, ROR
James:	NOP, MOVE (final version), MOVEA, MOVEQ, MOVEM, ADD, ADDA, ADDI, ADDQ, SUB, SUBI, AND, OR

Opcodes were worked on in each members individual branches, tested, and then integrated together into the master branch. Bisrat was responsible for his opcodes, documentation,debugging, and input. James was responsible for his opcodes, documentation, debugging, helper functions,and output. Stephen was responsible for his opcodes, documentation,debugging, test file, and integration of members opcodes in each individual branch onto the master branch.

Approximate percentages of coding done by each member in group is as follows:

Stephen:	30%
Bisrat:	30%
James:	40%