Disassembler Project Documentation

# 

# 

# 

# By Luke Bushey, Josh Trygg, Sasha Stavila

## **Introduction**

Our program’s purpose is to disassemble assembly instructions between a specific memory range that is provided by the user. Per the project specification, our disassembler will decode a subset of opcodes and addressing modes that are built into the 68k architecture.

## **Design Approach**

Our team chose to follow agile methodologies to complete the project. First, we began with conceptualizing exactly what the disassembler does. After gaining an understanding of what was ahead, we designed flow charts for the different aspects of the project. At this point, we set-up a schedule and assigned roles to each individual.

Figure 1 shows our team’s schedule for working on the disassembler:

|  |  |  |  |
| --- | --- | --- | --- |
| **Meeting Dates** | **Project Iteration** | **Description** | **Goal** |
| 11/17/2016 | Specs and initial design | Wade through specifications / initial flow-chart theory | Complete designs before next meeting |
| 11/19/2016 | Saturday session | Discuss intricacies of our designs with professor | To begin implementation |
| 11/21/2016 | Skeleton | Set-up NOP | Print out NOP |
| 11/22/2016 | APIs | Discuss API integration | Move on to developing test program |
| 11/28/2016 | Implementation | Establish coding standards and continue implementation | Eat lots of turkey this week |
| 12/01/2016 | Debugging, testing | Debug & fix. | Test, Test, Test, more tests |
| 12/06/2016 | Last iteration of implementation | Group code integration | Have a walking, talking, disassembler |
| 12/10/2016 | Documentation | Start official documentation | Complete a rough draft |
| 12/13/2016 | Complete Documentation | Finish official documentation | Completion! |

Figure 1: Disassembler Schedule

Although it outlines clearly what what must be done in the project and when, our team did not meet all deadlines on the schedule due to a variety of reasons. Among other unforeseen circumstances, Josh was very sick for almost a week, and that set us back in our development life cycle. However, the schedule did help us keep track of what to expect for each team meeting

**Program** **Description**

* *Design philosophy*
  + ***I/O philosophy***

First priority was to deal with any potential exceptions thrown regarding the user’s input. After this was ensured, installing our flag checker and buffer management was key. In fact, we decided to only use a single buffer (data register) to handle both good & bad cases. This was accomplished by resetting the buffer’s starting address at the beginning of the decode loop for OpCode, and since we always ended with a null terminator before printing the buffer, we were able to effectively “reset” the buffer using this process.

Lastly, setting up our looping technique to output only a specific amount of lines to the user was taken care of.

* + ***Opcode philosophy***

The design of the opcode section was to first use a jump table to branch into each section of opcodes, as defined by the first byte of the opcode word. Next, we would copy the instruction into a separate data register, and then manipulate the instruction by shifting and masking to further determine which opcode within the branch.

Some basic routines had to be developed and used throughout the decode process, such as: a getSize routine- determined the size in bits 6-7, highRegBits - converted bits 9-11 to register numbers, getDirBit - determines the directionality of the opcode in bit 8. Other routines also expanded their functionality by using the values returned to further the scope, such as using the size to print into the buffer.

* + **EA philosophy**

Initial theory was to isolate the low order bits 0-5 (mode & Xn) and utilize a jump table to decide which addressing mode was necessary. This worked out well, but when the mode was (111), I should have perhaps used a second jump table to separate the second layer of modes.

At this point determining the logic for each mode was not overly complicated, once we were able to merge our EA section of code into our master code base, our opcode development propelled forward.

* *Algorithms*
  + We are very proud of our ability to hack away at our multitude of bugs. :)
* *Limitations*
  + **I/O limitations**

No known limitations.

* + **EA limitations**

The only addressing mode that isn’t working AT ALL, is the *Address with Index* mode. Due to our desire to test our program further, we decided to stop spending time on figuring out how to make it work.

The immediate addressing works, but in a hacky way. The EA functionality for immediate addressing only addresses the MOVE opcode commands when an immediate operand is needed. The other opcodes such as ANDI, SUBI, etc, use different logic we have built.

* + **OpCode limitations**

All of the eligible OpCodes are able to be decoded, with the exception of the BSR and MOVEM commands. Due to the design of the the opcode branches, the handling of instructions within the same branch (first hex value of the opcode instruction) may not be handled properly, and in turn result in inaccurate results. Example, the ADDX function gets interpreted as the ADD function, the ABCD function gets interpreted as EXG, however ADD and EXG both get decoded accurately as well.

Due to the nature of the above limitation, in that certain opcodes aren’t handled properly, the pointer incrementation can also be affected. Example, some of the branch statements will include a displacement of an additional word, whereas others don’t, in turn our pointer incrementation from the badInstruction routine will have adjusted the pointer incorrectly. Eventually we will land on another valid instruction, but the error may result in some missed instructions or misinterpreted instructions.

## **Specification**

The following list explains the function of the Disassembler:

1. Program initiates with a display.
2. Program prompts user to enter a starting address
3. Program prompts user to enter an ending address, error checking ensues.
4. Program checks addresses to ensure it meets requirements: valid size, boundaries, and difference.
5. Are we at the end of the test addressing space? If not, continue, if so, end program
6. Jump to OpCode and start decoding opcode word.
7. If necessary, jump to EA to decode the addressing mode.
8. If the “bad” flag is set, the buffer is filled with the appropriate bad data.
9. If the “good” flag is set, print out the appropriate opcode & increment the address pointer
10. I/O handles printing out 20 lines at a time.
11. The address is again checked for validity (are we at the end of the test space)
12. Prompt user to enter 1 for more decoding, or 0 to exit program
    1. If 1, go back to step 2
    2. Else, print exit message and stop program

**Test** **Plan**

Our test plan was to incrementally test the functionality of each release iteration. Releases typically included modifications or additions to: opcodes, IO prompts, or address modes. At each release, we then added a new test suite to our code within a defined testing area. Note, these additional test cases were added into the pre-existing list test cases, thus allowing regression testing to be achieved.

As bugs were noticed, they were to be entered into a bug log with the intent of resolving them within the next iteration. If there were any patches that caused previous test cases to fail, they were then rejected and the last stable version was reverted as our master. Lastly, we kept the list of bugs that weren’t resolved by the project deadline and added them into our limitations within this report.

# **Team Assignments**

|  |  |  |
| --- | --- | --- |
| **Assignment** | **Team Members** | **Description** |
| I/O | Project Manager | Sasha | Sasha: wrote intro/exit screen; checked addresses for even/odd, upper/lower bound, and inequalities; wrote loop to print 20 lines at a time.  Josh: wrote ASCII-to-hex converter.  Luke, Josh, Sasha: wrote hex-to-ASCII converter |
| OpCode | Luke | Developed main opcode decoding functionality, which was used to determine each opcode along with underlying logic to differentiate between opcodes. |
| EA | Josh | In charge of EA coding, and used as an all-around debugger and contributed to opcode implementation |