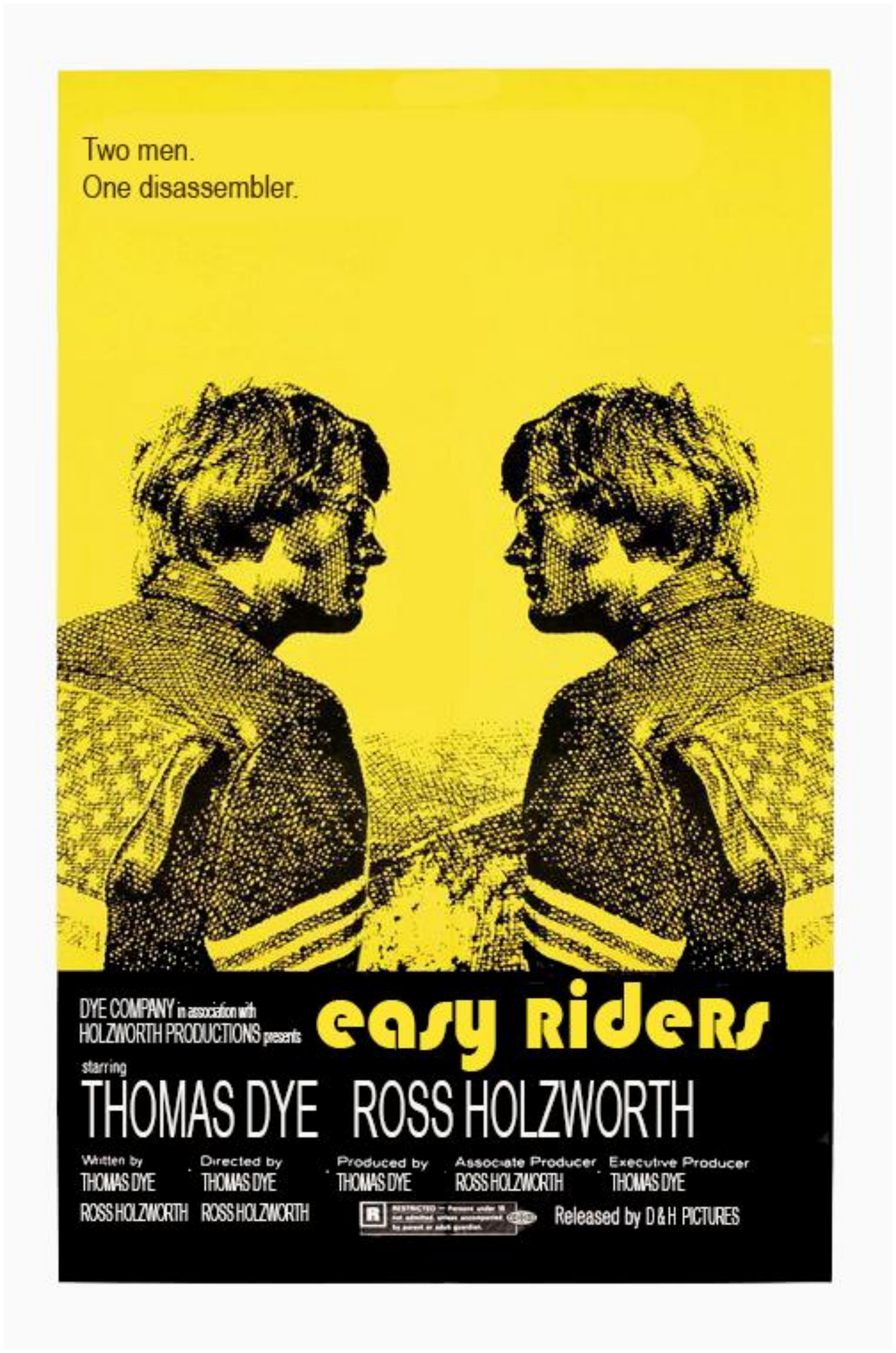
68K DISASSEMBLER FINAL PROJECT


Two men.
One disassembler.
DYE COMPANY in association with HOLZWORTH PRODUCTIONS presents
easy riders
starring
THOMAS DYE ROSS HOLZWORTH
Written by THOMAS DYE ROSS HOLZWORTH
Directed by THOMAS DYE ROSS HOLZWORTH
Produced by THOMAS DYE
Associate Producer ROSS HOLZWORTH
Executive Producer THOMAS DYE
Released by D & H PICTURES

CSS 422 - Hardware - 3/13/2016

Program Description

Not able to fill the traditional I/O, Opcode, and EA person rolls, the approach that we used for a group of two on the disassembler project was divide and conquer. Thomas researched and designed the overall project schema and Ross built the initial prototype that would detect and analyze an arbitrary number of instructions loaded in memory for the opcode NOP. Later, this approach continued to be useful where Thomas took responsibility of all tasks related to client interactivity and opcode detection and Ross tackled EA detection and processing. Eventually, we would meet in the middle and write opcode logic to finish the project.

An early challenge was unfamiliarity with the assembly programming language. There wasn't much of a transition from "Hello, World!" to figuring out how to apply offsets to jump tables and converting values in hex to ASCII for console output.

Before leaving the prototype stage, Thomas worked on designing the end-to-end flow of how instructions would be processed. It proved to be helpful to fully specify all requirements to decode and display instructions while remaining implementation free through detailed flowcharts. Once we solidified the major pieces of code that would need to be written, we decided the best course of action was to tackle the MOVE opcode which would, depending on operands, would make use of any type of effective addressing modes.

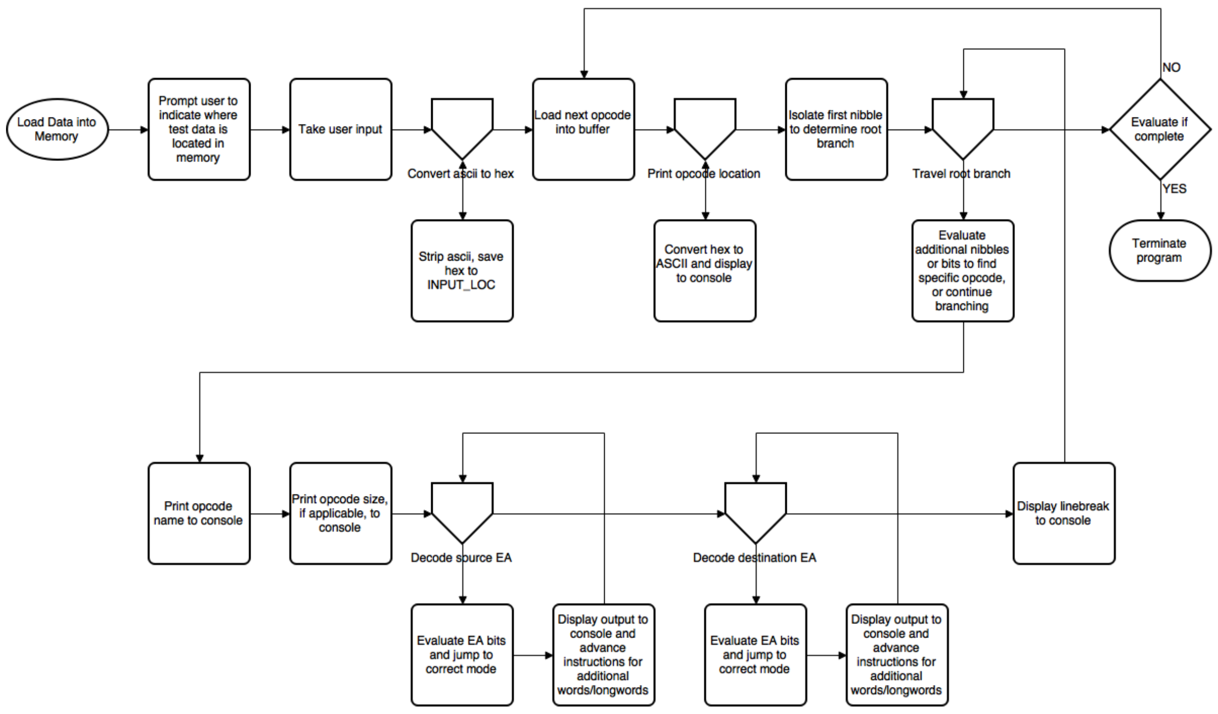

Load Data into Memory
Prompt user to indicate where test data is located in memory
Take user input
Convert ascii to hex
Strip ascii, save hex to INPUT_LOC
Load next opcode into buffer
Print opcode location
Convert hex to ASCII and display to console
Isolate first nibble to determine root branch
Travel root branch
Evaluate additional nibbles or bits to find specific opcode, or continue branching
Evaluate if complete
NO
YES
Terminate program
Print opcode name to console
Print opcode size, if applicable, to console
Decode source EA
Evaluate EA bits and jump to correct mode
Display output to console and advance instructions for additional words/longwords
Decode destination EA
Evaluate EA bits and jump to correct mode
Display output to console and advance instructions for additional words/longwords
Display linebreak to console

Figure 1: Execution Flow Diagram

The program is designed to take input from a user, where the user specifies which block of system memory to scan for opcode instructions using a start and stop address. The ending address must be beyond the starting address. In the event that an odd byte address is supplied for either start or stop locations, the program will display an error message and request the address again. After both addresses are validated, they are stored in memory for later use.

Main execution of the program is in the command parser which loads the next opcode in memory. From there, the memory address is printed to the console and the bits of the opcode are evaluated until each opcode is uniquely and correctly identified.

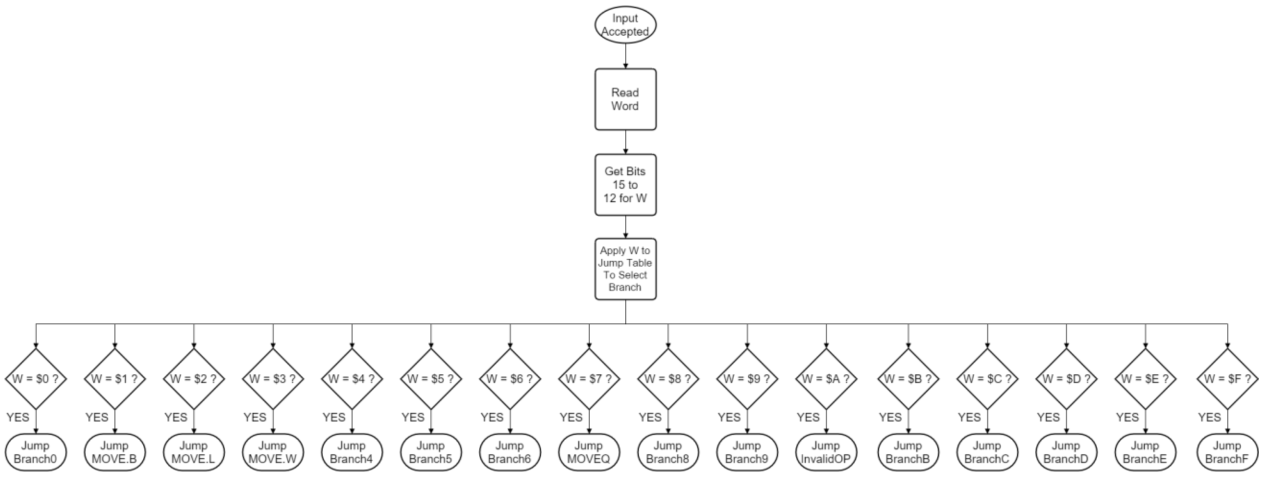


Figure 2: Root Branch Diagram

Core logic for the disassembler is in effective address decoding. For most of the project, this was Ross's focus area. As with all of our subroutines, EA decoding is modular, having a clearly defined interface. The same subroutine is used for both source and destination decoding for all EA modes. Implemented opcodes are essentially an extension of the branch code, continuing to parse the bits supplied in the instruction. The EA decode subroutine is also responsible for advancing the current position locator for the next instruction to be processed where absolute short, long, and immediate data is involved.

A limitation of the current design of this disassembler is that will not accept input for addresses larger than one word. This effectively limits the available range for test code from 0000 to FFFF. While this issue could be fixed, it was deemed low priority and is a deferred bug.

Specification

This project is an inverse assembler that converts a memory image of instructions and data back to 68000 assembly language and outputs the disassembled code to the console. The program is designed to operate between memory addresses $1000 and $3000. Optional test code can also be loaded into memory outside of these address ranges.

The disassembler is designed to run on the Sim68K I/O console provided with the EASy68K Editor/Assembler. If default fonts and console window sizes are maintained, the disassembler will print one page of output at a time.

Instructions are disassembled line-by-line using the following format:

Memory Location   Opcode   Operand(s)

If the disassembler encounters instructions that are not supported, the DATA label will be supplied to the console in the format:

Memory Location   DATA $WXYZ

The output $WXYZ is the hexadecimal number that could not be decoded.

The following required opcodes are supported by the disassembler:

ADD	ASR	DIVU	MOVE	NOP
ADDA	Bcc (BCC, BGT, BLE)	JSR	MOVEM	ROL
ADDI	CLR	LEA	MOVEQ	RTS
AND	CMP	LSL	MULS	SUB

The following optional opcodes are supported by the disassembler:

ADDQ	CMPI	MOVEA	SUBA
ANDI	DIVS	MULU	SUBI
ASL	LSR	ROR	SUBQ

The following Effective Addressing modes are supported:

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

Unsupported EA modes are reported with the label 'BadEA' in the operands field.

Testing

We used test driven development for this project. The first test was to decode the opcode NOP three times in a row, which was achieved with our prototype. The second test utilized the MOVE command exclusively, but was designed to progressively add in effective addressing modes. Thomas constructed a TestMOVE module which systematically tested decoding for size, each supported destination EA mode, each supported source EA mode, and then a variety of special cases found in class homework and exercises.

While Ross implemented MOVE opcode and EA code to allow the TestMOVE module to pass each of the 24 tests, Thomas built another TestBranch module to construct basic cases for supported and unsupported opcodes. When supported features would be added, additional lines of the test modules would be unblocked.

Test modules that extensively tested a single instruction were only created for cases where the implementation or supported features were found to be unique, such as MOVE, ADD. In the case of shift and rotate instructions, it was found that they share a common logic (and implementation) thus a single TestSHIFTROT module would be necessary. Additionally, all opcodes draw on the same the subroutines to print and decode effective addresses, so testing new opcodes is mostly a pass/fail situation where it either completely works, or does not at all.

Before transitioning active development from the prototype to what would become the disassembler, Thomas constructed a framework that specified code formatting, comments, naming, and placement. While coding in assembly is not necessarily self-documenting, labels offered the ability to name the location of subroutines and what they do. For example: All public (meant to be called externally) subroutines begin with the prefix SR. SR_CheckStart is a subroutine that checks the input location address for the test data. Any additional labels associated with that subroutine are prefixed with cs specifying they belong to CheckStart. The beginning and end of each subroutine is clearly marked with a comment bar.

Each comment bar contains the full unabbreviated name of the function. Where applicable, an extended description of what the subroutine does and how it performs is mentioned as notes inside of the comment block. While it is part of the coding standard, neither of us reliably included register comments for our code, instead knowing from prior experience which registers or storage names are global variables. This would have been more important to us if we were collaborating with a larger team, but was easy enough to maintain with only two competent programmers.

Most of the project source code is thoroughly documented explaining either logic behind the command issuance, read and returned values, or pre/post environment changes.

Exception Report

The disassembler will only accept inputs as high as address 0000FFFF for either start or stop values. We suspect fixing this is non-trivial and it was deemed low priority.

MOVEM will display register ranges in reverse order when post decrement is used. The current implementation always reads the register mask from left to right. To reverse the order, we would need a second loop for when the register mask is stored the opposite order. As the compiler optimizes and determines its own read-in and read-out behavior, we opted to not introduce more program complexity to display registers in ascending order at this time.

There are no other known exceptions to report.

Team Assignments

Over the course of this project, Thomas acted as the primary organizer: design flowcharts and system logic, writing reports, as well as programming test modules, client interaction, opcode identification, and multiple opcode logic routines. Ross researched and built the functional prototype, as well as programmed the output subroutines, all effective address modes, and multiple opcode logic routines.

There is not a clear division of, as a percentage, the amount of coding by team member. For content, Thomas committed approximately 4x lines of data (pictures, documents, code lines) according to GitHub, and while that’s important, so is time and energy spent on original concepting and difficult algorithms such as effective address decoding.

Ultimately, both team members chose to do what they wanted to do for this project and contributed the amount they were able to. In the end, the disassembler was built on time, went beyond the required assignment specifications--implementing 12 additional opcodes--and eliminated bugs where they could be found.