Disassembler Specification

CSS 422

3/19/2009

Ethan Crawford, Michael Grimm, Paula Haddad

# Program description

Our final project for CSS 422 was to implement a Motorola 68K disassembler in 68K assembly language. Our primary design consideration was modularity, to allow for efficient parallel development and incremental step-wise testing throughout the development process. Prior to beginning implementation, the team met twice to define coding and calling conventions and to hammer out interfaces between I/O, opcode, and effective address. We agreed on a GNU-like subroutine calling convention, in which D0 is reserved for return values and all other parameters are passed on the stack, and a Microsoft-like coding convention, in which datatypes are preceded by a "Hungarian" prefix to unambiguously identify their types.

From an I/O perspective, we needed the ability to convert between integer and string, read user input, print string buffers, and copy strings from one location to another. Homework 6 provided the forcing function to implement most of those methods before starting the assembler project. Paula was the chief I/O programmer, and the rest of the team collaborated with her.

At startup, the program displays a welcome message and then prompts the user for the starting address and the ending address of the code to be disassembled. Then the program scans the section of memory specified by the user and outputs the address and the instructions contained in that memory space to the display.

Output is displayed one screen of data at a time, and the user must hit the **ENTER** key to display the next screen.

The program prints illegal or unrecognizable instructions as DATA, printing them to the screen in the format ' xxxx    DATA    $YYYY'.

The program takes user input in ASCII format and converts the number to hexadecimal. For printing, it can also convert the output from Hexadecimal back to ASCII.

We used different modes of effective addressing including Data register Direct, Address Register Direct, Address Register Indirect, Immediate Data, Stack, Address Register post-increment, and many more.

The buffer we set in memory is filled with information as we get it, and then the opcode routine sends a good or bad flag to the I/O routine to print the appropriate buffer to the display.

On the opcodes front, we wanted, as much as possible, to avoid code duplication and "spaghetti code". To accomplish this, we felt that an in-depth understanding of the instruction set was required. Before writing any code, We made a list of every assigned opcode and identified the common elements within them. One example:

**ADD**

<ea>,Dn \* Two ways of ordering operands

Dn,<ea>

1101 reg oo(0,1) sz(00,01,10) effadd

The final list showed many opcodes with identical operands, suggesting that a set of common operand subroutines would be useful. To keep things orderly, we chose not to use jump tables, instead opting for a central set of subroutines driven by an array of InstructionInfo structs (arrays of words). The parent function, decodeInstruction(), loops through each struct, retrieves and applies constant and variable masks to the current instruction to identify opcodes, calls decodeSize() to print the size suffix, and jumps to a stored function pointer to decode the operands. Individual operand decoders invoke decodeEA() as-needed and leverage simple macros for printing registers.

This design contained a few interesting elements. One was a technique that used the position of bits in a word to encode the valid ranges for a variable mask. This range was used to disambiguate between two instructions where the only difference might be, for example, the two bits that encoded the valid size. In this encoding, a '1' at bit position 0 meant that 0 was a valid value, and so on. Another useful technique was directly mapping the size bits of an instruction to indices in an array of strings. This allowed us to simply define one string pointer array per equivalence class and store a pointer to that array in the associated struct. It also enabled the decodeSize()implementation to be very simple, as its only job was to jump to an offset in the string array, using the size code as the offset.

The effective addresses are decoded through simple masking techniques and branching logic. The decodeEA() function is provided a byte on the stack by the individual operand decoder functions that call it. This byte contains size information (if necessary on bits 7-6), mode information (bits 5-3) and register information (bits 2-0). Each of these bit patterns are separated into their own data register for easy access. Each effective address request is differentiated by mode and directed to a separate label for specific handling.

We used the Boolean method of returning -1 when an error occurs and greater than -1 on success. Each subroutine is documented with the parameters it takes and the ones it returns – making sure all the description, parameters, and remarks are included. We also took advantage of Macros to make our programming lives easier.

No algorithms were copied and no canned routines were used.

# Specification

1. Gets user input for starting and ending addresses.
2. Validates user input (checks for out-of-range characters, odd numbers, and ending addresses less than or equal to starting addresses)
3. Loops over a range of in-memory instructions, printing the disassembled values.
4. Prints one page at a time and prompts for user interaction to print the next page.
5. Prints DATA and the hex value for any indecipherable operands.

# Test Plan and Coding Conventions:

The disassembler was tested by embedding a test program, ORGed at $B0000, into the main assembly file. Initial instructions were selected to provide minimum coverage for each particular opcode. Additional instructions were added to cover operand order variations, individual opcode special cases, and internal boundary conditions such as last operand in the struct array, all registers and one register specified for MOVEM, etc. Further instructions were added to cover all known syntactic combinations of supported effective addressing modes with both positive and negative values.

The test program is provided below:

\*\* TEST PROGRAM

\*\* TODO: DELETE this section before turning in

\*\* Hi Arnie!

STARTTEST ORG $B0000

CMPA.L #$FF,A0

ADDA.W (7,A0,A1),A1 \* ADDA

ADDI.B #$B,(-1023,A4) \* ADDI

MOVE.B #13,D0

MOVEM.L D0/A1,-(SP) \* push

LEA $1220,A1 \* Load string pointer

MOVE.B #13,D0 \* Print with newline

MOVEM.L (SP)+,D0/A1 \* pop

TRAP #15 \* DATA $4E4F

MOVE.L #$DEADBEEF,D0

MOVE.W $B0030(PC),D2

MOVE.W $B0002(PC),D2

MOVE.W $B0004(PC),D2

MOVE.W $B0006(PC),D2

MOVE.L $B0006(PC),D2

MOVE.L $B0006(PC),D3

ADD.W ($B0040,PC,A2.L),D2

ADD.W ($B0030,PC,A2.L),D3

ADD.L ($B0000,PC,A3.L),D2

ADD.L ($B00A4,PC,A4.L),D2

LEA stack,SP \* Load the SP

BEQ STARTTEST \* DATA $67FE

ADD.L #1,D0 \* ADDQ: DATA $5280

BSR.B $B0004

BSR.W $B0100

BSR.L $B8000

BSR DELETEME \* BSR

AND.B D3,D7 \* AND

AND.W D3,(A7) \* AND

AND.L (A7),D3 \* AND

AND (A0)+,D0

AND D0,(A0)

SWAP D7 \* SWAP

MOVE #2,D0

ADDI.L #$DEADBEEF, (8,A5)

ADDI.W #$DEAD,D1

ADDI.B #$FF,(127,A4)

ADDI.B #$FE,(127,A4)

ADDI.W #$FE,(127,A4)

ADDI.B #$FE,(255,A4)

ADDI.W #$FFFF, (1023,A4)

ADDI.L #$F0F0F0F0,(2047,A5)

ADDI.L #$FFFF000F,(32767,A1)

ADDI.W #$BEEF,(8,A4) \* ADDI

ADDI.L #$DEADBEEF,(32,A4) \* ADDI

LSR.W (A0)

LSR.B D0,D1 \*\* LSR

LSR.W D5,D6

LSR.B #7,D0

LSR.B #8,D3

LSR.L D0,D1

MOVE.L A0,A1

MOVE.L D0,D1

MOVE.L $AAAAAAAA,$55555555

MOVE.L ($AAAAAAAA),($55555555)

ADD.B D0,D1 \* ADD

ADD.L #$DEADBEEF,D0

ADDA.W (7,A2,D1),A1

ADDA.W (7,A2,A2),A1

ADDA.L (127,A2,A3),A1

ADDA.W (7,A2,A3),A2

ADDA.W (10,A2,A3),A2

ADDA.W (10,A2,A3),A3

ADDA.W (10,A2,A3),A4

ADDA.W (15,A2,A3),A4 \*15 bits is the highest displacement

ADDA.L (127,A2,A3),A4

ANDI #$C0,D4 \* ANDI

ASL #4,D7 \* ASL

ASR D4,D4 \* ASR

CLR.B D0 \* CLR

CLR.W (A0)

CLR.L 8(A0)

CMP D0,A5 \* CMP

CMPA.L A3,A2 \* CMPA

CMPI.B #$F,D0 \* CMPI

EOR D0,(A3)+ \* EOR

EORI.L #4,-(A2) \* EORI

EXG D5,D1 \* EXG

EXG A0,A7

EXG D7,A3

JMP DELETEME \* JMP

JSR DELETEME \* JSR

LEA DELETEME,A5 \* LEA

DELETEME

NOP

LSL D3,D7 \* LSL

MOVE.L D5,A5 \* MOVE

\* MOVEA

MOVEM.L (A7)+,A1

MOVEM.L A0-A6/D0-D6,-(A7) \* MOVEM

MOVEM.L (A7)+,A2/D2

MOVEM.L A0/D1,-(A7) \* MOVEM

MOVEM.L (A7)+,A0-A7

NEG -(A0) \* NEG

NOP \* NOP

NOT (A4) \* NOT

OR -(A7),D0 \* OR

ORI #$0F0F,D5 \* ORI

ROL.B D3,D2 \* ROL

ROL.W #2,D4

ROL ($B0000).L

ROR D0,D1 \* ROR

ROR #7,D0

ROR ($C0000).W

RTS \* RTS

SUB.B D0,D3 \*SUB

SUB.W A7,A6

SUB.L D0,A1

ENDTEST

We defined multiple coding standards prior to beginning implementation. In addition to the calling convention and variable naming convention discussed above, we also set up revision control and agreed on a subroutine documentation format that included parameters and registers used. During our meetings, we would discuss progress made, share learning and plan the next steps.

# Exception report

We did not encounter unfixable issues and are not aware of any bugs.

# Team Assignments

* Ethan: opcodes and main()
* Michael: effective addresses and integration testing
* Paula: I/O, documentation, and QA

# List file