Project Description

Team Unknown: Noah McMichael, Zhaoning Qiu, Christopher Bailey

Our program's purpose is to disassemble assembly instructions between a specific memory range that is provided by the user in the form of a .cfg file. The .S68 to be disassembled is loaded into memory by open data function in the Easy68K program. Per the project specification, our disassembler will decode a subset of opcodes and addressing modes that are built into the 68k architecture. The work was split into three main parts: I/O Section,the EA Section, and Opcode Section.

I/O Summary

The I/O section concerned the reading the .cfg file in order to receive the correct starting and ending addresses for the code that is going to be disassembled. The program uses Traptask13 in order to print to the output file.

Opcode Summary

After the IO code has found the starting and ending address and the main method calls the CLASSIFY_ADDRESS subroutine, the Opcode code begins. CLASSIFY_ADDRESS takes the first 4 bits of the subset of opcodes to decode and uses that information to determine what type of code it is. This evaluation is performed by first moving the current to-be decoded opcode stored in (A3) to D2. This prepares the to-be decoded opcode for a bitmask check, where D2 is ANDed with a hex value and is then compared to that same hex value. For example, when ANDing $9 with $9, we then compare the result with $9. If they are equal, then we know that the opcode starts with $9 (indicating SUB in this example).

You may notice that the bitmasks start from high (E) to low (0). This ordering is not just for clarity, it is crucial for the set up of our opcodes. In the previous example, if we had first ANDed $9 with $1 and then compared $1 to the result, we would return a value of $1, incorrectly indicating that this opcode is a MOVE operation, rather than the expected SUB operation.

After finding the first 4 bits of the to-be decoded opcode, further bitmask checks (again ordering high to low) are required to figure out what exact type of opcode we are handling, since some opcodes start with the same first 4 bits. After finding the exact opcode, a foundInsertOpcodeNameHere function is called. Here we push the name of the opcode to the makeshift A2 stack to be printed later. Here we also further parse the size suffix if necessary by calling the SIZE_APPENDER subroutine, and then calling one of 3 APPEND_L/W/B subroutines. After appending the suffix to the A2 stack, the size information is stored in the SIZE variable for EA's use later.

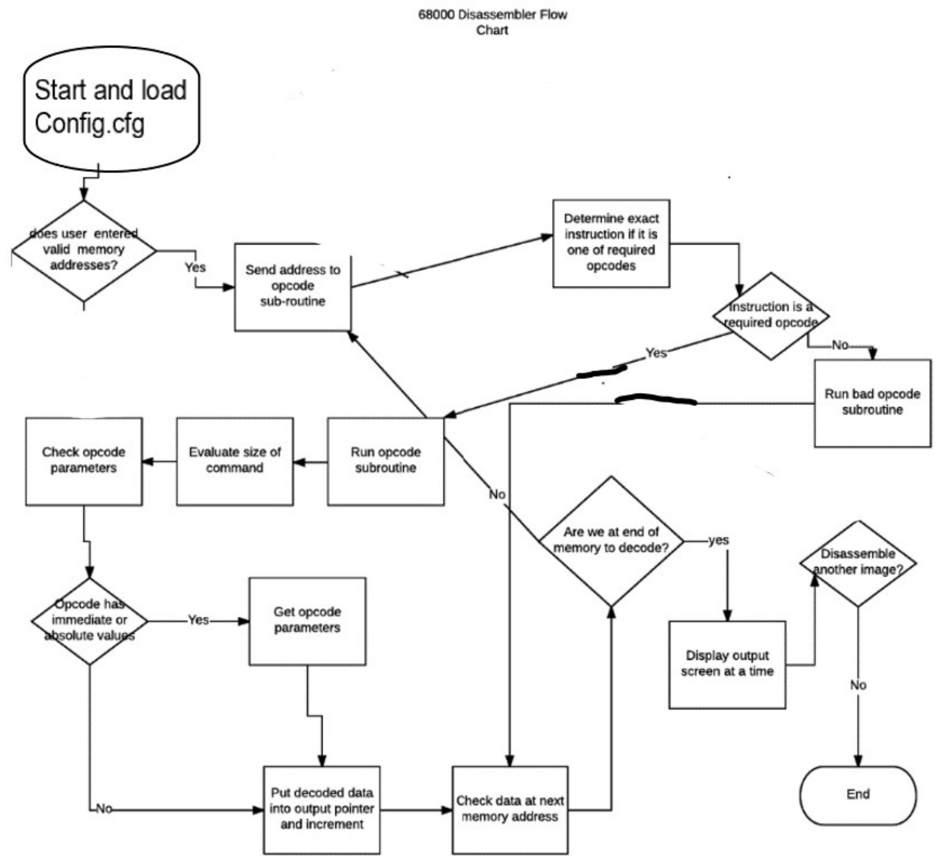
EA Summary

After OP routine finished paring, the OP type would assign to the OP_NAME. It depends on different OP_NAME, the selector will decide to go to which EA parse subroutine. Most of OP type has a group of similar EA sequence, instruction with the same type EA address will share the same EA parsing algorithm.

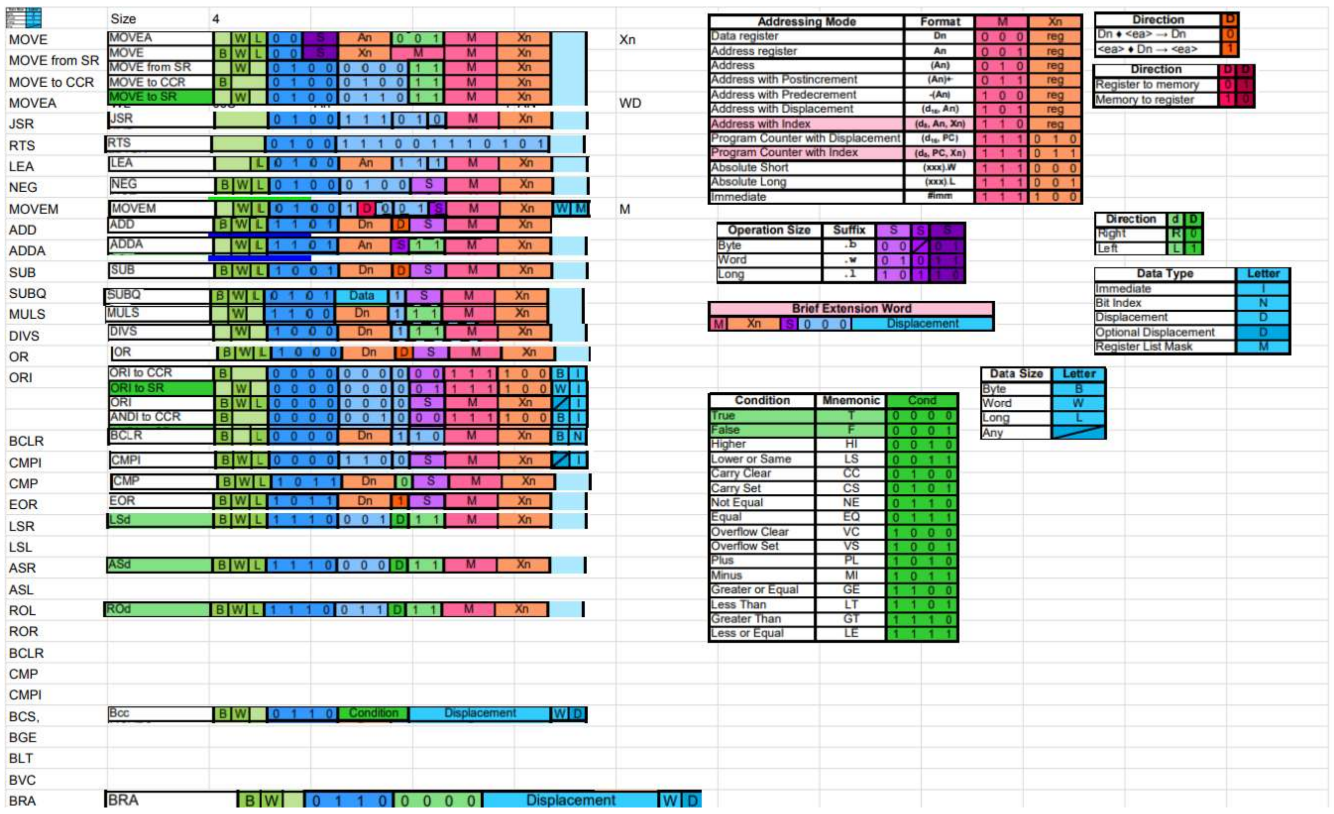
For the common instruction, parsing function will check if it contains immediate value or absolute address for destination and source. If it does, the parsing function will retrieve the next address to get the value, and convert it to ascii, then append the ascii output to output register. If it doesn't, the parsing routine will retrieve the mod and reg for destination and source. Assign each different code for each different mod type, and depends on the reg value, it will append the register word of source and register to the output array.

Some special instructions may not follow the common rule. For these instructions, a proper parsing algorithm will added in each ea parsing subroutine.

Flowchart of Disassembler


68000 Disassembler Flow Chart
Start and load Config.cfg
does user entered valid memory addresses?
Yes
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
Check opcode parameters
Evaluate size of command
Run opcode subroutine
No
Are we at end of memory to decode?
yes
Disassemble another image?
Opcode has immediate or absolute values
Yes
Get opcode parameters
Display output screen at a time
No
No
Put decoded data into output pointer and increment
Check data at next memory address
End

Opcode and EA Pairings



	Size	4							
MOVE	MOVEA	W L	0 0 S	An	0 0 1	M	Xn		Xn
MOVE from SR	MOVE	B W L	0 0 S	Xn	M	M	Xn		
	MOVE from SR	W	0 1 0 0	0 0 0 0	1 1	M	Xn		
MOVE to CCR	MOVE to CCR	B	0 1 0 0	0 1 0 0	1 1	M	Xn		
MOVEA	MOVE to SR	W	0 1 0 0	0 1 1 0	1 1	M	Xn		WD
JSR	JSR		0 1 0 0	1 1 1	0 1 0	M	Xn		
RTS	RTS		0 1 0 0	1 1 1 0 0 1 1 1 0 1 0 1					
LEA	LEA	L	0 1 0 0	An	1 1 1	M	Xn		
NEG	NEG	B W L	0 1 0 0	0 1 0 0	S	M	Xn		
MOVEM	MOVEM	W L	0 1 0 0	1 D 0	0 1 S	M	Xn	W M	M
ADD	ADD	B W L	1 1 0 1	Dn	D S	M	Xn		
ADDA	ADDA	W L	1 1 0 1	An	S 1 1	M	Xn		
SUB	SUB	B W L	1 0 0 1	Dn	D S	M	Xn		
SUBQ	SUBQ	B W L	0 1 0 1	Data	1 S	M	Xn		
MULS	MULS	W	1 1 0 0	Dn	1 1 1	M	Xn		
DIVS	DIVS	W	1 0 0 0	Dn	1 1 1	M	Xn		
OR	OR	B W L	1 0 0 0	Dn	D S	M	Xn		
ORI	ORI to CCR	B	0 0 0 0	0 0 0	0 0 0	1 1 1	1 0 0	B I	
	ORI to SR	W	0 0 0 0	0 0 0	0 0 1	1 1 1	1 0 0	W I	
	ORI	B W L	0 0 0 0	0 0 0	0 S	M	Xn	/ I	
	ANDI to CCR	B	0 0 0 0	0 0 1	0 0 0	1 1 1	1 0 0	B I	
BCLR	BCLR	B L	0 0 0 0	Dn	1 1 0	M	Xn	B N	
CMPI	CMPI	B W L	0 0 0 0	1 1 0	0 S	M	Xn	/ I	
CMP	CMP	B W L	1 0 1 1	Dn	0 S	M	Xn		
EOR	EOR	B W L	1 0 1 1	Dn	1 S	M	Xn		
LSR	LSd	B W L	1 1 1 0	0 0 1	D 1 1	M	Xn		
LSL									
ASR	ASd	B W L	1 1 1 0	0 0 0	D 1 1	M	Xn		
ASL									
ROL	ROd	B W L	1 1 1 0	0 1 1	D 1 1	M	Xn		
ROR									
BCLR									
CMP									
CMPI									
BCS,	Bcc	B W	0 1 1 0	Condition	Displacement			W D	
BGE									
BLT									
BVC									
BRA	BRA	B W	0 1 1 0	0 0 0 0	Displacement			W D	

Addressing Mode	Format	M	Xn
Data register	Dn	0 0 0	reg
Address register	An	0 0 1	reg
Address	(An)	0 1 0	reg
Address with Postincrement	(An)+	0 1 1	reg
Address with Predecrement	-(An)	1 0 0	reg
Address with Displacement	(d_{16}, An)	1 0 1	reg
Address with Index	(d_8, An, Xn)	1 1 0	reg
Program Counter with Displacement	(d_{16}, PC)	1 1 1	0 1 0
Program Counter with Index	(d_8, PC, Xn)	1 1 1	0 1 1
Absolute Short	(xxx).W	1 1 1	0 0 0
Absolute Long	(xxx).L	1 1 1	0 0 1
Immediate	#imm	1 1 1	1 0 0

Direction	D
Dn ♦ <ea> → Dn	0
<ea> ♦ Dn → <ea>	1

Direction	D	D
Register to memory	0	1
Memory to register	1	0

Operation Size	Suffix	S	S	S
Byte	.b	0 0	/	0 1
Word	.w	0 1	0	1 1
Long	.l	1 0	1	1 0

Direction	d	D
Right	R	0
Left	L	1

Data Type	Letter
Immediate	I
Bit Index	N
Displacement	D
Optional Displacement	D
Register List Mask	M

Brief Extension Word				
M	Xn	S	0 0 0	Displacement

Data Size	Letter
Byte	B
Word	W
Long	L
Any	/

Condition	Mnemonic	Cond
True	T	0 0 0 0
False	F	0 0 0 1
Higher	HI	0 0 1 0
Lower or Same	LS	0 0 1 1
Carry Clear	CC	0 1 0 0
Carry Set	CS	0 1 0 1
Not Equal	NE	0 1 1 0
Equal	EQ	0 1 1 1
Overflow Clear	VC	1 0 0 0
Overflow Set	VS	1 0 0 1
Plus	PL	1 0 1 0
Minus	MI	1 0 1 1
Greater or Equal	GE	1 1 0 0
Less Than	LT	1 1 0 1
Greater Than	GT	1 1 1 0
Less or Equal	LE	1 1 1 1