Project Specification

Coding Standards

Commenting

Every line of code must be commented. Comments should be descriptive and not reiterate what can be intuited from the code. For example:

```
MOVE.W   #3,D1    *Move 3 into D1
```

This comment would not be acceptable since it doesn't describe functionality. An acceptable comment would be:

```
MOVE.W   #3,D1    *Move 3 to D1 as a counter for the loop
```

Comments should also additionally be used wherever necessary to explain blocks of code that otherwise would be unclear.

Subroutines

Subroutines must include full documentation above the method. Subroutine names should be prefixed with "sub_" as a namespace, and its name should be descriptive of its function. Subroutines can receive and return variables on the stack or through the registers, but their functionality must be declared in the description. All variables should have their size as part of the definition. Passing on the stack is generally preferred.

Subroutine variables that are passed on the stack should be also declared as offsets at the beginning of the function definition. These offsets provide a way to access the variables from the stack in a more straightforward way.

Any labels within the subroutine should be the whole subroutine name with the label appended to the end. For example, a loop within the "sub_foo_bar" routine could be labeled "sub_foo_bar_loop". Additionally any offset variables declared at the beginning of the method should also be prefixed with the subroutine name for namespacing purposes.

Subroutines should be as modular as possible and not attempt to do too much. Subroutines should be used as much as possible to keep redundancy to a minimum.

All subroutines assume the **caller saved** pattern for backing up registers. Subroutines thus will never backup registers.

Variables, Constants, and Definitions

Commonly used variables (word_size for example) should be declared as constants. “Magic” numbers are strongly discouraged unless it is well documented and logically would be no more clear as a constant.

Global variables must be kept to a minimum and only used where absolutely needed. Passing needed information on the stack, or through registers is preferred for the sake of traceability.

SDL Definitions

All SDL definitions should be at the end of the file. All definitions, operation codes, labels, and ea mode lists should be grouped by the operation they are defining.

Usage Manual

Adding Additional Opcodes

The disassembler makes adding new opcodes extremely easy. To add a new opcode a definition must first be created. See the next section on **SDL** for syntax information.

The addresses of code definitions must be placed in the *op_codes* array. The order of the addresses in the array is not important. The parser will try each code definition until it finds one that matches and doesn’t throw an invalid operation error.

Setting Up Tab Stops

The disassembler supports tab stops to make the output as clean as possible. The tab stop positions are modified by changing the values in the *tab_stops* array. They are defaulted to 12, 24, and 36.

Operating the Disassembler

When the disassembler first starts, it will request an address in hex (inclusive) to begin disassembling from. This address can be anywhere in memory. After entering a starting address and pressing enter, the program will request an address (inclusive) that it should end disassembling at.

Pressing enter will start the disassembler. It will proceed to disassembler one screens worth of code and then prompt for the *Enter* button again. Continue pressing the *Enter* button to display screens of code until the program reaches the ending address.

Information is output in the following format:

```
[Current Address]       [Opcode]    [Fields]
```

If the data at an address is not recognized as an opcode, the program will instead output:

```
[Current Address]       DATA        $[Data]
```

Supported Operation Codes

ADD	**CMP**	MOVE from SR	**ORI**
ADDA	CMPA	MOVE to CCR	PEA
ADDI	**CMPI**	**MOVEA**	**ROL**
ADDQ	CMPM	**MOVEM**	**ROR**
AND	**DIVS**	MOVEQ	RTR
ANDI	DIVU	**MULS**	**RTS**
ANDI to CCR	**EOR**	MULU	Scc *
ASL	EORI	NBCD	**SUB**
ASR	ILLEGAL	**NEG**	SUBA
Bcc *	JMP	NEGX	SUBI
BCLR	**JSR**	**NOP**	**SUBQ**
BRA	**LEA**	NOT	SWAP
BSR	**LSL**	**OR**	TAS
CHK	**LSR**		TST
CLR	**MOVE**		UNLK

* All applicable conditions are available for these modes.
Specification required operations are in bold.

Supported EA Modes

Data Register Direct
Address Register Direct
Address Register Indirect
Address Register Indirect w/ Predecrement
Address Register Indirect w/ Postdecrement
Absolute Word/Long
Immediates

Appendix A:
Structured Disassembler Language

Overview

Our **Structured Disassembler Language (SDL)** is used to describe the behaviour, sizes, and fields of an operation code. By using SDL to describe an opcode, there is generally no need to write specialized code to properly disassemble an operation. For example, defining the disassembly for the opcode MOVE can be reduced to this simple definition:

```
move_s  DC.B   'MOVE',0
move_e  DC.W   $0000,$C000,move_d,op_code_term
move_d  DC.W   move_s,opf_sizes,12,%01,%11,%10,opf_ea,
               op_ea_all,3,0,opf_ea,op_ea_dest_std,6,9,
              opf_term
```

SDL additionally provides very flexible operations and even overrides of default behaviour to handle specialized cases.

SDL Structure

SDL generally uses four parts to define each opcode.

Label	The label is defined as a 0 terminated string of **bits**. This label should have the characters that represent the operation. For example; CMP should be defined as ‘CMP’,0.
Code	The operation code is a string of **words** that define how the code will be identified, and if it is, which definitions to use.
EA Modes List	Available EA modes are defined in **byte** strings.
Definition	The definition is a string of **words** that contains the fields that the operation will use and defines the general functionality of the opcode.

Each operation is required to have one label, one code, and one definition. EA Modes can be defined as needed.

Operation Label <Label>

The opcode label is a 0 terminated string of **bytes** that describe the opcode. This string should not contain any tabs, spaces, periods, or size information. This is an example label for the CMP method.

```
op_string_cmp      DC.B  ‘CMP’,0
```

Operation Code

The operation code is used to match a word string to the operation. The operation code is built of three or more components.

```
<Code> = <Value>,<Mask>,<Definition*>[,<Definition*>,
<Definition*>...],op_code_term
```

The *<Value>* is a hex constant that will match an opcode that has been ANDed with the *<Mask>* constant. If the value then matches, the first definition will be parsed. It will continue to parse definitions until an opcode is successfully output, or *op_code_term* is reached. All operation codes must terminate with *op_code_term*.

Each *<Definition>* should be passed as an address.

The example code for CMP is shown here.

```
op_code_cmp  DC.W   $B000,$F100,op_def_cmp,op_code_term
```

The opcode will be masked with $F100 and then compared to $B100. If there is a match it will attempt to use *op_def_cmp* as a definition.

Operation EA Modes List <EA Mode List>

An *<EA Mode List>* is a string of **bytes** that contains accepted EA modes. The list may contain any of the available EA modes and must be terminated with *ea_term* to specify the end of the list. An *<EA Mode List>* may also just contain the special operator *ea_all* signifying that all EA modes are valid in which case *ea_term* is not required.

```
<EA Mode List> = ([ea_reg_dir],[ea_add_dir], [ea_ari],
[ea_ari_pre],[ea_ari_post], [ea_abs, ea_imm],[ea_abs,
ea_word], [ea_abs, ea_long], ea_term)|ea_all
```

A single opcode definition may use multiple EA Mode lists. This is a common case when the EA field behaves differently as a source and destination operand.

If the operation code doesn't match a mode given in the EA Mode list, the invalid op flag will be set to true. Additionally, any operator of *byte_size* will set the invalid op flag to true if it's used with the ea_add_dir mode.

Available EA Modes <EA Mode>

ea_reg_dir	Register direct. Example: Dn
ea_add_dir	Address direct. Example: An
ea_ari	Address register indirect. Example: (An)
ea_ari_pre	Address register indirect with pre-decrement. Example: -(An)

ea_ari_post	Address register indirect with post-decrement. Example: (An)+
ea_abs	Used to declare either an absolute address or immediate mode. Must be used in conjunction with *ea_long*, *ea_word*, or *ea_imm*.
ea_word	Used to specify an absolute word. Example: $1234
ea_long	Used to specify an absolute long. Example: $12345678
ea_imm	Used to specify an immediate value. Example: #$1234
ea_all	Specifies that all EA modes are accepted

Note: Immediate sizes are generally determined by the size of the operand.

Operation Definition

The behaviour of an opcode is determined by one or more definitions. This is the general definition format, which should be declared as a constant **word** string.

```
<Definition> = <Label*>,[<Cond>],[<Options>],opf_term
```

The only required components of a definition are the address of a label and the *opf_term* keyword. All definitions must have *opf_term* as the very last keyword.

Condition <Cond>

If an opcode has a conditions (such as Bcc and Scc), you can utilize *<Cond>* to handle the condition string building. *<Cond>* is defined as follows:

```
<Cond> = opf_cond,<Bit Pos>,(true|false)
```

*The Boolean value is intended to be **true** for Scc which supports ST and SF, and **false** for Bcc which doesn't support those conditions.*

The *<Bit Pos>* should contain the position for where the condition codes can be found in the opcode string. The second argument is a Boolean that indicates if the branch conditions *True* and *False* are acceptable for the operation.

If an acceptable condition is not matched the invalid op flag will be set true.

Options <Options>

If additional options are provided, the first requirement is a size definition. This determines the acceptable sizes for the opcode, as well as the information necessary for determining the opcode size. Optionally an opcode can include *<Fields>* or *<Flippable Fields>* which hold the information necessary for additional output.

```
<Options> = <Size Def>,[<Fields>|<Flippable Fields>]
```

Size Mode <Size Def>

There are three primary ways to define the size functionality of an opcode.

```
<Size Def> = no_size|<Size>|<Sizes>
```

The *no_size* keyword signifies that size is not a factor in the opcode such as with branching operations.

Use *<Size>* when the opcode is always a single size. In this mode, the size will not be output to the display, but the intrinsic size will be used for certain *<Fields>*. The first keyword must be *opf_size* followed immediately by the size descriptor.

```
<Size> = opf_size,(byte_size|word_size|long_size)
```

The final option for describing an opcodes size is through *<Sizes>*. This method allows for declaring multiple acceptable sizes, and the bit patterns required for recognizing a size.

```
<Sizes> = opf_sizes,<Bit Pos>,(<Size Bits>|no_size),
(<Size Bits>|no_size),(<Size Bits>|no_size)
```

The first required keyword is *opf_sizes* that declares that multiple sizes will be provided. The next parameter should be a *<Bit Pos>* that defines the starting bit location for where the size is determined in the opcode word. It is assumed that 2 bits will be used to determine the size.

Following the bit position, are three arguments that describe *byte_size*, *word_size*, and *long_size* respectively. If an opcode supports a size, two bits should be passed that represent the size. If the opcode does not support a size, *no_size* may be passed in place of the bits.

For example, if an opcode supports bytes and words, then the following descriptor may be used with *%00* representing words and *%10* representing longs.

```
opf_sizes,7,%00,%10,so_size
```

Fields and Flippable Fields <Fields>, <Flippable Fields>

There are two ways that fields can be declared. *<Fields>* is used for normal operation, whereas *<Flippable Fields>* allows for a “flippable” representation.

For discussion on the available fields, see the next section.

The basic format for specifying a field is by providing a comma-separated list of *<Fields>*. There is no limit to the number of fields provided. Each field will be processed and output with a comma separating each.

```
<Fields> = <Field>[,<Field>,<Field>,...]
```

The *<Flippable Fields>* mode is handy for opcodes such as ADD, which requires that one operand is a data register, whereas the other may be an effective address. Whether the register is the source or destination, is determined by a single bit.

```
<Flippable Fields> = opf_flip,<Bit Pos>,<Fields>,
opf_flipped,<Fields>
```

The first argument is the bit position of the determinate bit for flipping. If this bit is set to 0, the first set of *<Fields>* will be used, and if the bit is set to 1, the set of fields after *opf_flipped* will be parsed instead.

Operation Definition Fields <Field>

A <Field> token is used to describe how data is represented in an opcode. Eight supported field modes provide a range of flexibility.

```
<Field> = <EA>|<Count>|<Register>|<Count or
Register>|<String>|<Subroutine>|opf_imm|opf_disp
```

Effective Address <EA>

An effective address mode can be used wherever an effective address exists in an opcode. EA definitions must begin with *opf_ea*, which signifies the beginning of the definition.

```
<EA> = opf_ea,<EA Modes*>,<Bit Pos>,<Bit Pos>
```

The first parameter after *opf_ea* should be either an address to a list of EA Modes or ea_all. The first *<Bit Pos>* should be the starting bit location in the opcode word that defines the EA mode, the second *<Bit Pos>* should be the starting bit location in the opcode word that defines the EA register.

EA Modes <EA Modes>

There are two available options for *<EA Modes>.* The operator function *ea_all* can be used to specify that ALL available EA modes are available. Alternatively, an address to an *<EA Mode List>* may be used to specify all available modes.

```
<EA Modes> = ea_all|<EA Mode List>
```

Count <Count>

Some operands such as SUBQ take a count (or data) directly into their operation codes. To specify this, use the operator function code *opf_count*.

```
<Count> = opf_count,<Bit Pos>
```

Immediately following *opf_count* should be a bit position that signifies the starting location of the count in the opcode. The count size is assumed to be 3 bits.

Register <Register>

Some operands such as ADD and ADDA require that data go directly into a register. This functionality is supported with the *opf_reg* command.

```
<Register> = opf_reg,<Bit Pos>,<EA Mode>
```

Register requires a *<Bit Pos>* that points to the starting bit of the register number, as well as one of the EA modes that specifies what type of register it is.

Count or Register <Count or Register>

Operands that can accept either a count or a register should use *opf_count_reg*.

```
<Count or Register> = opf_count_reg,<Bit Pos>,<Bit Pos>,
<EA Mode>
```

The first argument following *opf_count_reg* should be the bit position of the determinate bit. This is the bit that when set as 0, will make the operator use a count, and 1 use a register.

The second *<Bit Pos>* should be the starting bit position of the count/register with an assumed size of 3 bits.

Finally, *<EA Mode>* will state which mode to use if the determinate bit signifies a register.

String Injection <String>

Injecting a string as a field is done with the following syntax:

```
<String> = opf_string,<Address*>
```

<Address> should be the address of a byte string with a null terminated character. The string at that address will be printed as a field.

Subroutine <Subroutine>

Subroutine mode gives the ability to provide the address to a subroutine to branch to at the given position. When the parser reaches an *opf_sub* statement, it will immediately pass control from to the given subroutine address which may then output freely.

```
<Subroutine> = opf_sub,<Subroutine Address*>
```

The called subroutine can expect two things. The current opcode will be in register D0, and the determined size of the operand will be in D7 holding *byte_size*, *word_size*, *long_size*, or *no_size*.

Immediate opf_imm

The keyword *opf_imm* specifies that an immediate should be printed. The size of the immediate is automatically determined from the operand size. If the operand is *byte_size* or *word_size*, the next addressed word will be printed as hex in the format **#$FFFF**. If the operand is *long_size*, the next addressed long will be printed as hex in the format **#$FFFFFFFF**.

Displacement opf_disp

For opcodes that use displacement such as the branching routines, the keyword *opf_disp* should be used. The displacement value will be calculated and added to the address of the current opcode to determine the branch location. This is then output as a hex address in the format **FFFF**.