SOFTWARE SUPPORT FOR
MOTOROLA 68000 MICROPROCESSOR AT CERN

M68MIL CROSS MACRO ASSEMBLER

Horst von Eicken

GENEVA
1983
Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l'autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s'il s'agit d'utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée.
Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document; ceux-ci peuvent être librement utilisés par les instituts de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s'opposer à toute revendication qu'un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrits dans le présent document.

Literary and scientific copyrights reserved in all countries of the world. This report, or any part of it, may not be reprinted or translated without written permission of the copyright holder, the Director-General of CERN. However, permission will be freely granted for appropriate non-commercial use.

If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.

CERN — Service d'information scientifique — RD/618 - 2600 - Décembre 1983
ABSTRACT

This document is a user's guide for programming the Motorola 68000 microprocessor in assembly language. It describes the programming model, addressing modes and instruction set of the M 68000 as well as the use of the M68mil cross macro assembler. Version 3.6 of the assembler has been installed at CERN on CDC, DEC VAX, IBM, Norsk Data and Siemens computers. The source code of the assembler is available from CERN on request.
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>M 68000 Programming Model</td>
<td>2</td>
</tr>
<tr>
<td>Privilege States</td>
<td>2</td>
</tr>
<tr>
<td>Memory Addressing Scheme</td>
<td>3</td>
</tr>
<tr>
<td>Register Set</td>
<td>4</td>
</tr>
<tr>
<td>Addressing Modes</td>
<td>5</td>
</tr>
<tr>
<td>Implied Addressing</td>
<td>6</td>
</tr>
<tr>
<td>Register Direct Addressing</td>
<td>7</td>
</tr>
<tr>
<td>Immediate Addressing</td>
<td>9</td>
</tr>
<tr>
<td>Direct Addressing</td>
<td>10</td>
</tr>
<tr>
<td>Address Register Indirect Addressing</td>
<td>11</td>
</tr>
<tr>
<td>Address Register Indirect with Postincrement</td>
<td>12</td>
</tr>
<tr>
<td>Address Register Indirect with Predecrement</td>
<td>13</td>
</tr>
<tr>
<td>Address Register Indirect with Displacement</td>
<td>15</td>
</tr>
<tr>
<td>Address Register Indirect with Index and Displacement</td>
<td>15</td>
</tr>
<tr>
<td>Program Counter Relative Addressing</td>
<td>16</td>
</tr>
<tr>
<td>Program Counter Relative with Displacement</td>
<td>17</td>
</tr>
<tr>
<td>Program Counter Relative with Index and Displacement</td>
<td>18</td>
</tr>
<tr>
<td>Machine Instructions</td>
<td>20</td>
</tr>
<tr>
<td>ABCD - Add Decimal With Extend</td>
<td>20</td>
</tr>
<tr>
<td>ADD - Add binary</td>
<td>20</td>
</tr>
<tr>
<td>ADDA - Add Address</td>
<td>21</td>
</tr>
<tr>
<td>ADDI - Add Immediate</td>
<td>21</td>
</tr>
<tr>
<td>ADDQ - Add Quick</td>
<td>21</td>
</tr>
<tr>
<td>ADDX - Add Extended</td>
<td>22</td>
</tr>
<tr>
<td>AND - AND Logical</td>
<td>22</td>
</tr>
<tr>
<td>ANDI - AND Immediate</td>
<td>23</td>
</tr>
<tr>
<td>ASL - Arithmetic Shift Left</td>
<td>23</td>
</tr>
<tr>
<td>ASR - Arithmetic Shift Right</td>
<td>23</td>
</tr>
<tr>
<td>Bcc - Branch Conditionally</td>
<td>24</td>
</tr>
<tr>
<td>BCHG - Test a bit and change</td>
<td>25</td>
</tr>
<tr>
<td>BCLR - Test a bit and clear</td>
<td>25</td>
</tr>
<tr>
<td>BRA - Branch always</td>
<td>26</td>
</tr>
<tr>
<td>BSET - Test a Bit and Set</td>
<td>26</td>
</tr>
<tr>
<td>BSR - Branch to subroutine</td>
<td>26</td>
</tr>
<tr>
<td>BTST - Test a Bit</td>
<td>27</td>
</tr>
<tr>
<td>CHK - Check Register against Bounds</td>
<td>27</td>
</tr>
<tr>
<td>CLR - Clear an Operand</td>
<td>28</td>
</tr>
<tr>
<td>CMP - Compare</td>
<td>28</td>
</tr>
<tr>
<td>CMPA - Compare address</td>
<td>29</td>
</tr>
<tr>
<td>CMPI - Compare Immediate</td>
<td>29</td>
</tr>
<tr>
<td>CNPM - Compare Memory</td>
<td>29</td>
</tr>
<tr>
<td>DBcc - Test Condition, Decrement and Branch</td>
<td>30</td>
</tr>
<tr>
<td>DIVS - Signed divide</td>
<td>30</td>
</tr>
<tr>
<td>DIVU - Unsigned Divide</td>
<td>31</td>
</tr>
<tr>
<td>EOR - Exclusive OR Logical</td>
<td>31</td>
</tr>
</tbody>
</table>
SOFTWARE SUPPORT FOR MOTOROLA 68000
M68MIL CROSS MACRO ASSEMBLER

TABLE OF CONTENTS

EORI - Exclusive OR Immediate ........................................... 32
EXG - Exchange Registers .................................................. 32
EXT - Sign Extend ............................................................ 33
JMP - Jump ........................................................................ 33
JSR - Jump to Subroutine ...................................................... 33
LEA - Load Effective Address .................................................. 34
LINK - Link and Allocate .......................................................... 34
LSL - Logical Shift Left .......................................................... 35
LSR - Logical Shift Right .......................................................... 35
MOVE - Move Data from Source to Destination .................................. 35
MOVE - Special MOVES ............................................................. 36
MOVEA - Move Address ............................................................. 36
MOVEM - Move Multiple Registers ............................................. 37
MOVEP - Move peripheral data .................................................... 38
MOVEQ - Move Quick .............................................................. 38
MULS - Signed Multiply ............................................................ 38
MULU - Unsigned Multiply ........................................................ 39
NBCD - Negate Decimal With Extend ........................................... 39
NEG - Negate ....................................................................... 40
NEGX - Negate with Extend ........................................................ 40
NOP - No Operation ................................................................. 40
NOT - Logical Complement ......................................................... 41
OR - Inclusive OR Logical ......................................................... 41
ORI - Inclusive Or Immediate ...................................................... 42
PEA - Push Effective Address ..................................................... 42
RESET - Reset External Devices ................................................ 43
ROL - Rotate Left (without Extend) ............................................. 43
RDR - Rotate Right (without Extend) ........................................... 43
ROXL - Rotate Left with Extend ................................................ 43
ROXR - Rotate Right with Extend ............................................... 44
RTE - Return from Exception ..................................................... 44
RTR - Return and Restore Condition Codes .................................. 45
RTS - Return from Subroutine .................................................. 45
SBCD - Subtract Decimal with Extend ....................................... 45
Scc - Set According to Condition .............................................. 45
STOP - Load Status Register and Stop ....................................... 46
SUB - Subtract binary ................................................................ 47
SUBA - Subtract Address .......................................................... 47
SUBI - Subtract Immediate ........................................................ 48
SUBQ - Subtract Quick ............................................................. 48
SUBX - Subtract with Extend ...................................................... 48
SWAP - Swap Register Halves ................................................... 49
TAS - Test and Set an Operand .................................................. 49
TRAP - Trap ........................................................................ 50
TRAPV - Trap on Overflow ........................................................ 50
TST - Test an Operand .............................................................. 50
UNLK - Unlink ....................................................................... 51
Assembler Notations ................................................................. 52
Source Statement Format ........................................................... 52

- VI -
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Column</td>
<td>52</td>
</tr>
<tr>
<td>Location Field</td>
<td>52</td>
</tr>
<tr>
<td>Operation Field</td>
<td>52</td>
</tr>
<tr>
<td>Operand Field</td>
<td>53</td>
</tr>
<tr>
<td>Comment Field</td>
<td>53</td>
</tr>
<tr>
<td>Comment Statement</td>
<td>53</td>
</tr>
<tr>
<td>Statement Continuation</td>
<td>53</td>
</tr>
<tr>
<td>Micro Substitution</td>
<td>54</td>
</tr>
<tr>
<td>Symbols</td>
<td>54</td>
</tr>
<tr>
<td>Identifiers</td>
<td>54</td>
</tr>
<tr>
<td>Numbers</td>
<td>54</td>
</tr>
<tr>
<td>Character Strings</td>
<td>55</td>
</tr>
<tr>
<td>Assembler Symbols</td>
<td>55</td>
</tr>
<tr>
<td>Absolute Symbol</td>
<td>55</td>
</tr>
<tr>
<td>Relative Symbol</td>
<td>55</td>
</tr>
<tr>
<td>External Symbol</td>
<td>56</td>
</tr>
<tr>
<td>Undefined Symbol</td>
<td>56</td>
</tr>
<tr>
<td>Operators</td>
<td>56</td>
</tr>
<tr>
<td>Arithmetic Operators</td>
<td>56</td>
</tr>
<tr>
<td>Shift Operators</td>
<td>56</td>
</tr>
<tr>
<td>Logical Operators</td>
<td>56</td>
</tr>
<tr>
<td>Operator Precedence</td>
<td>57</td>
</tr>
<tr>
<td>Expressions</td>
<td>57</td>
</tr>
<tr>
<td>Assembler Directives</td>
<td>58</td>
</tr>
<tr>
<td>Module Identification</td>
<td>59</td>
</tr>
<tr>
<td>IDENT - Module Identification</td>
<td>59</td>
</tr>
<tr>
<td>END - End of Module</td>
<td>59</td>
</tr>
<tr>
<td>Section Control</td>
<td>59</td>
</tr>
<tr>
<td>Absolute Section</td>
<td>59</td>
</tr>
<tr>
<td>Relative Section</td>
<td>60</td>
</tr>
<tr>
<td>G (general) Section</td>
<td>60</td>
</tr>
<tr>
<td>R (relocatable) Section</td>
<td>61</td>
</tr>
<tr>
<td>C (common) Section</td>
<td>62</td>
</tr>
<tr>
<td>U (unique) Section</td>
<td>62</td>
</tr>
<tr>
<td>&quot;Previous&quot; Section</td>
<td>62</td>
</tr>
<tr>
<td>Symbol Definition</td>
<td>63</td>
</tr>
<tr>
<td>EQU - Equate Symbol Value</td>
<td>63</td>
</tr>
<tr>
<td>SET - Set or Reset Symbol Value</td>
<td>63</td>
</tr>
<tr>
<td>Module Linkage</td>
<td>63</td>
</tr>
<tr>
<td>ENTRY - Declare Entry Symbols</td>
<td>63</td>
</tr>
<tr>
<td>EXTERN - Declare External Symbols</td>
<td>64</td>
</tr>
<tr>
<td>Data Generation and Storage Reservation</td>
<td>64</td>
</tr>
<tr>
<td>ALIGN - Align on Word Boundary</td>
<td>64</td>
</tr>
<tr>
<td>DC - Define Constant</td>
<td>65</td>
</tr>
<tr>
<td>DS - Define Storage</td>
<td>66</td>
</tr>
<tr>
<td>Conditional Assembly</td>
<td>66</td>
</tr>
<tr>
<td>ENDF - End of IF Range</td>
<td>66</td>
</tr>
<tr>
<td>ELSE - Reverse Effects of IF</td>
<td>67</td>
</tr>
<tr>
<td>IF - Test Attribute of Symbol</td>
<td>67</td>
</tr>
</tbody>
</table>

- VII -
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFC</td>
<td>IFNC - Compare two Character Strings</td>
</tr>
<tr>
<td>IFcc - Test Expression against Zero</td>
<td>68</td>
</tr>
<tr>
<td>Macro Operations</td>
<td>69</td>
</tr>
<tr>
<td>Macro Definition</td>
<td>69</td>
</tr>
<tr>
<td>ENDM - End Macro Definition</td>
<td>70</td>
</tr>
<tr>
<td>LOCAL - Local Symbols</td>
<td>70</td>
</tr>
<tr>
<td>MACRO - Macro Heading</td>
<td>70</td>
</tr>
<tr>
<td>MEXIT - End Macro Expansion</td>
<td>71</td>
</tr>
<tr>
<td>NARG - Number of Macro Arguments</td>
<td>71</td>
</tr>
<tr>
<td>Macro Calls</td>
<td>71</td>
</tr>
<tr>
<td>Source Stream Control</td>
<td>72</td>
</tr>
<tr>
<td>DISTINCT - Define Handling of Letters in Identifiers</td>
<td>72</td>
</tr>
<tr>
<td>DUP</td>
<td>ENDDUP - Duplicate Code Sequence</td>
</tr>
<tr>
<td>INSERT - Insert Secondary Source</td>
<td>72</td>
</tr>
<tr>
<td>SYSTEXT - Insert system text information</td>
<td>73</td>
</tr>
<tr>
<td>Listing Control</td>
<td>73</td>
</tr>
<tr>
<td>FAIL - Generate an Error Message</td>
<td>73</td>
</tr>
<tr>
<td>G - List Assembled Code</td>
<td>74</td>
</tr>
<tr>
<td>LLLEN - Set Line Length and Format Listing</td>
<td>74</td>
</tr>
<tr>
<td>LIST - Select List Options</td>
<td>74</td>
</tr>
<tr>
<td>NOL</td>
<td>NOLIST - Cancel Listing</td>
</tr>
<tr>
<td>NOPAGE - Turn off Paging</td>
<td>76</td>
</tr>
<tr>
<td>PAGE - Top of Page</td>
<td>76</td>
</tr>
<tr>
<td>PLEN - Define Page Length</td>
<td>76</td>
</tr>
<tr>
<td>SPC - Space Between Source Lines</td>
<td>76</td>
</tr>
<tr>
<td>TTL - Assembly Listing Title</td>
<td>76</td>
</tr>
<tr>
<td>STTL - Assembly Listing Subtitle</td>
<td>76</td>
</tr>
<tr>
<td>Object Code Control</td>
<td>77</td>
</tr>
<tr>
<td>BLONG - Use Two-Word Branch</td>
<td>77</td>
</tr>
<tr>
<td>BSHORT - Use One-Word Branch</td>
<td>77</td>
</tr>
<tr>
<td>FLONG - Force Direct Long Address</td>
<td>77</td>
</tr>
<tr>
<td>FSHORT - Force Direct Short Address</td>
<td>77</td>
</tr>
<tr>
<td>NOOBJ - Suppress CUFGM Output</td>
<td>78</td>
</tr>
<tr>
<td>Date and Time Stamp</td>
<td>78</td>
</tr>
<tr>
<td>STAMP - Date and Time Stamp</td>
<td>78</td>
</tr>
<tr>
<td>Assembler Maintenance</td>
<td>78</td>
</tr>
<tr>
<td>DEBUG - Print Assembler Debug Information</td>
<td>79</td>
</tr>
<tr>
<td>Symbolic Debugging</td>
<td>80</td>
</tr>
<tr>
<td>SYMDEBUG - Produce Symbol Table Information</td>
<td>80</td>
</tr>
<tr>
<td>Programming Convention Support</td>
<td>81</td>
</tr>
<tr>
<td>Standard Stack Initialisation</td>
<td>81</td>
</tr>
<tr>
<td>Definition of Global Variables</td>
<td>84</td>
</tr>
<tr>
<td>Start Main Program and Initialise Stack</td>
<td>85</td>
</tr>
<tr>
<td>Standard Subprogram Call</td>
<td>86</td>
</tr>
<tr>
<td>Standard Subprogram Entry</td>
<td>89</td>
</tr>
<tr>
<td>Formal Parameter Definition</td>
<td>89</td>
</tr>
<tr>
<td>Definition of Local Variables</td>
<td>90</td>
</tr>
<tr>
<td>Enter a Subprogram</td>
<td>90</td>
</tr>
<tr>
<td>Return from Subprogram</td>
<td>91</td>
</tr>
</tbody>
</table>

- VIII -
Example of a Standard Subprogram .................................. 91
Switches Controlling the $\text{macros} .................................. 94
Implementations ............................................................... 95
Acknowledgements ............................................................ 96
References ................................................................. 97
1 INTRODUCTION

M68MIL, the Cross Macro Assembler described in this user's guide can be used to translate assembler source programs for the Motorola 68000 microprocessor into CUFOM [4][5], the CERN Universal Format for relocatable Object Modules. A link editor subsequently allows the combination and linking of several such modules into a new CUFOM module. It also permits the construction of CUFOM libraries, which can be placed as input to the link editor to resolve unsatisfied external references once all other input to the link editor has been processed. A pusher finally translates a CUFOM module to the Motorola 'S' format needed for down-line loading from the host computer into the memory of the target machine.

The assembler is upward compatible with the M68000 Cross Macro Assembler [2] provided by Motorola. Additional assembler directives are provided to allow the generation of relocatable object modules. Although this user's guide gives some information about the principles of operation for the M 68000 microprocessor, the user is advised to consult the Motorola publication


to obtain detailed information on the use of the M 68000.

The assembler has been designed as a two pass assembler and is written in PASCAL. Several independent source modules may form the input source stream for the assembler. In pass one the assembler will read a source module, develop the symbol table for this module, expand all macros, perform source code duplication and write the information on a scratch file for processing by pass two. In pass two the assembler reads the scratch file, uses the symbol table to translate the source code into CUFOM based M 68000 code, produces a listing and, if requested, a cross reference table. It will then check if further source modules are contained in the input source stream and repeat pass one and two until all modules are processed.

The assembler has been installed at CERN on CDC, DEC VAX, IBM, Norsk Data and Siemens computer systems. The source code of the assembler contains additional information to direct a pre-processor to extract the version for a particular computer system. Outside CERN the assembler has been installed on a PRIME computer and on a M 68000 operating under the control of a UNIX [1] operating system.

This user's guide describes version 3.6 of M68MIL.

---

1 UNIX is a Trademark of Bell Laboratories

- 1 -
2 M 68000 PROGRAMMING MODEL

For the convenience of the user of this guide the programming model of the M 68000 will be described here before we enter the description of the M68MIL cross assembler. It should however be understood, that the description given by Motorola Inc. in its publication


is the primary source of this information and should always be consulted, whenever doubts about the functioning of the M 68000 or any of its machine instructions arise.

Throughout the description of the M 68000 programming model the following abbreviations will be used:

- ea = effective address
- An = address register
- Dn = data register
- Rn = any An or Dn
- SR = status register
- PC = program counter
- SP = stack pointer
- d8 = 8 bit displacement
- d16 = 16 bit displacement
- abs.w = absolute short direct address
- abs.l = absolute long direct address
- ( ) = contents of
- => = replaces
- $nnn = hexadecimal number
- C, N, V, X and Z: condition codes

2.1 PRIVILEGE STATES

The M 68000 microprocessor allows two modes of operation

- supervisor or system mode
- user or normal mode

In supervisor mode all instructions can be executed, including those changing the mode of operation, while in user mode a few "privileged" instructions such as "RESET", "STOP" and access to the "system" byte of the status register are inhibited. The M 68000 is automatically put into supervisor mode whenever it is started using the external reset line and when an exception (interrupt, error condition) occurs.
2.2 MEMORY ADDRESSING SCHEME

The smallest directly addressable unit of memory is a byte (8-bits). Since memory addresses are specified using 24-bits, byte addresses are numbered from 000000 to FFFFFFF in hexadecimal notation. This allows a directly addressable memory array of up to 16,777,216 bytes.

\[ b_x \]

\[ \begin{array}{c}
    \text{byte} \\
    0 & 7
\end{array} \]

The next larger directly addressable unit of memory is a word (16-bits). A word consists of two consecutive bytes and must be aligned on even byte addressing boundaries.

\[ b_0 \ b_1 \]

\[ \begin{array}{c}
    \text{word} \\
    0 & 7 & 8 & 15
\end{array} \]

The largest directly addressable unit of memory is a long word (32-bits). A long word consists of two consecutive words (or four consecutive bytes) and must be aligned on even byte addressing boundaries.

\[ b_0 \ b_1 \ b_2 \ b_3 \]

\[ \begin{array}{c}
    \text{long word} \\
    0 & 15 & 16 & 31
\end{array} \]

Note:

- The byte with the even address occupies the upper part of the word (bits 15 through 8) while the byte with the odd address occupies the lower part of the word (bits 7 through 0). (i.e. the byte in the upper part is the one with the lower memory address)

- The word with the lower even address occupies the upper part of the long word (bits 31 through 16) while the word with the higher even address occupies the lower part of the long word (bits 15 through 0).

- The memory unit used by the M 68000 for instruction processing is the word, hence instructions must always start at even addresses.

- If memory is accessed for word or long word operations with an odd address an address error exception (see section on exceptions) will occur.
2.3 REGISTER SET

The register set offered by the M 68000 comprises

- eight 32-bit general purpose data registers, which may hold byte (8-bit), word (16-bit) and long word (32-bit) operands. They can be used as accumulators, index registers and counters. Their names are: D0, D1, ..., D7.

```
31 15 7 0

```

- seven 32-bit general purpose address registers, which are typically used to hold memory addresses allowing various types of memory access (see section on addressing modes). The contents of an address register may be changed using word (16-bit) or long word (32-bit) operations. Changing of the contents on a byte (8-bit) boundary is not possible. Their names are: A0, A1, ..., A6.

```
31 16 15 0

```

- two 32-bit stack pointers, one of which (called the system stack pointer: SSP) is active in supervisor mode, the other (called the user stack pointer: USP) is active in user mode. Since they have the same characteristics as the other address registers and since only one of them can be active at any given time, they are named A7.

- one 32-bit program counter, named PC.

- a 16-bit status register, named SR. The top byte of the status register is called the system byte, changeable only in supervisor mode, while the lower byte is called the user byte (or the condition code register: CCR).
Data registers and address registers may be used as source or destination operands. The following rules apply:

- When a data register is used as a source operand, only the appropriate low order portion is used (byte, word or long word).

- When a data register is used as a destination operand (register direct addressing mode), only the appropriate low order portion of the register is changed (byte, word or long word); the remaining high order portion, if any, remains unchanged. The condition code register is affected and reflects the change in the appropriate portion of the data register.

- When an address register is used as a source operand either the entire long word (32-bits) or the low order word (16-bits) are used, depending on the size of the operation.

- When an address register is used as a destination operand (register direct addressing mode), then the entire register is affected, i.e. during a word access bit 15 is replicated into bits 16 through 31 (sign extension). The condition code register remains unaffected.

### 2.4 ADDRESSING MODES

The M 68000 allows up to 14 different addressing modes, which can be divided into the following six groups:

- implied (or inherent) addressing
- register direct addressing
- immediate addressing
- direct (or absolute data) addressing
• address register indirect addressing
• program counter relative addressing

Throughout the description of the programming model we will use the term 'effective address' to describe the actual physical address to be used by the microprocessor in order to fetch or store an operand. This effective address, abbreviated <ea>, is always the result of one of the addressing modes mentioned above. Since not all addressing modes are allowed for a given machine instruction, we use the term <ea> in the description of the machine instruction and add a table depicting those addressing modes allowed as source operand and those as destination operand.

In the subsequent description of the addressing modes we will use the MOVE instruction for illustration since it allows almost all address modes.

The MOVE instruction always has two operands, one operand specifies the address of the data to be moved (also called: source operand), the other specifies the address to which to move the data (also called: destination operand). Since the M 68000 allows three different sizes of data to be handled, namely bytes, words and long words, a size option may be added to those machine instructions, that allow different data sizes. This is done by adding the suffix

.B for operations on bytes
.W for operations on words
.L for operations on long words

to the machine instruction. If no size code is added, the assembler assumes operations on words as the default operation size. The general form of the MOVE instruction is therefore:

MOVE.n <source>,<destination>

where "n" represents the data size code described above.

Implied Addressing

Implied addressing, sometimes also called inherent addressing, means the machine instruction itself already implies certain registers of the microprocessor, hence they are not specified as operands in the instruction itself. Typical examples for the M 68000 are the following instructions:

JMP <address> <address> => PC
BSR <address> (PC) => stack, <address> => PC

just to mention a few.
Register Direct Addressing

Register direct addressing, if it is used to describe the source operand, implies that the data for the operation is contained either in a data register or in an address register. When used to specify the destination operand, it means that the result of the machine instruction is to be stored into a data or address register. The general form of a MOVE instruction using register direct addressing modes for both operands would be

\[ \text{MOVE} \quad Rn, Rn \]

To illustrate some of the properties let us assume the following register contents:

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
</table>
| FF | FF | FF | FF | in register D0

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
</table>
| 12 | 34 | 56 | 78 | in register D1

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
</table>
| 00 | 01 | EF | 02 | in register A0

If we now execute the following machine instruction:

\[ \text{MOVE.L} \quad D1, D0 \]

we will find

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
</table>
| 12 | 34 | 56 | 78 | in register D0.

If we however had executed

\[ \text{MOVE} \quad D1, D0 \]

we would find

- 7 -
after execution.

Note:

- The number of locations allocated after the instruction word to hold the data depends on the size option used with the machine instruction and not on the actual size of the data.

- Since instructions must start on a word boundary immediate operands of size byte are stored in the lower byte of a word, the upper byte of which is zero.

Direct Addressing

In direct addressing the address to be used to access or store data is itself specified as an operand. It is stored as an extension to the instruction word using the direct addressing mode for one or both of its operands. The M 68000 allows two forms of direct addressing modes depending on the size of the address used

- in the long direct addressing mode a long word (32-bits) is used to hold the address, thus data in the entire address range from $000000 to $FFFFFF may be accessed.

- in the short direct addressing mode a word (16-bits) is used to hold the address. The same rule as already outlined for the address registers applies also for the short direct addressing mode; the 16-bit address is expanded to a full 32-bit address (of which for the current M 68000 chip only the low order 24-bits will be used). This means that with the short direct addressing mode data in the address ranges $000000 to $007FFF and $FF8000 to $FFFFFF may be accessed, or, to put it differently, data in the address range $008000 to $FF7FFF cannot be accessed using the short direct addressing mode.

The general form of a MOVE instruction using direct addressing modes for both operands would be

MOVE <address 1>,<address 2>

When during instruction assembly M68MIL encounters an operand a label symbol (see chapter: Assembler Notations) in direct addressing mode it does not always know the size of the final address to be used. It for instance has to take a decision as to which form to assume whenever it encounters a forward referenced symbol, i.e. a symbol which has not yet been defined, or a relative symbol, i.e. a symbol the value of which may be changed by the link-editor. By default it will use the long direct addressing mode to ensure correct handling of the symbol. The assembler directives SECTION, FSHORT and FLONG will allow the programmer to override the assembler defaults (see chapter: Assembler Directives).
To show an example, let us suppose that we want to move the data word stored at address $6000$ to address $8002$. The instruction would be

```
MOVE.W  $6000,$8002
```

which would be assembled to

```
b_0  b_1
    13  F8   instruction word
    15  8  7  0
b_0  b_1
    60  00   extension holding source address
    15  8  7  0
```

```
b_0  b_1  b_2  b_3
    00  00  80  02   extension holding destination address
```

```
  31  16 15  0
```

Note:

- The size of the instruction extension depends only on the size of the address specified and not on the size of the operation to be performed.

*Address Register Indirect Addressing*

There are actually five different address register indirect addressing modes. All of them use the contents of an address register as the base to calculate the effective address. The effective address is then used by the microprocessor to fetch or store an operand, or, in the case of a jump or jump subroutine instruction, as a new address value for the program counter.

*Address Register Indirect:*

The general form of a MOVE instruction using address register indirect addressing modes for both operands would be

```
MOVE  (An),(An)
```

This is the simplest form where the address register holds the effective address itself, in other words: the address register "points" to the address in memory that holds the data or the next instruction to be executed.

- 11 -
Assume register A4 contains

\[ \begin{array}{cccc}
31 & 16 & 15 & 0 \\
00 & D3 & 0E & 02 \\
\end{array} \]

register A4

and memory at address $00D30E02$ contains the word

\[ \begin{array}{cc}
b_0 & b_1 \\
1F & 01 \\
\end{array} \]

data word in memory

15 8 7 0

then the instruction

MOVE.W (A4),D0

would move $1F01$ into register D0.

Address Register Indirect with Postincrement:

The general form of a MOVE instruction using address register indirect with postincrement addressing modes for both operands would be

MOVE (An)+,(An)+

In this form the address register again points to the address in memory, but is automatically incremented by one, two or four depending on the size of the data operation (byte, word or long word) after the address contained in the register has been used by the microprocessor.

Assume we want to move a block of three long words beginning at address $1000$ to another block beginning at address $2000$, then we could write

\[
\begin{align*}
\text{MOVEA} & \quad \#1000,A0 & \text{start address of source block} \\
\text{MOVEA} & \quad \#2000,A1 & \text{start address of destin. block} \\
\text{MOVE.L} & \quad (A0)+,(A1)+ & \text{move and increment addresses} \\
\text{MOVE.L} & \quad (A0)+,(A1)+ \\
\text{MOVE.L} & \quad (A0)+,(A1)+ \\
\end{align*}
\]

The first two instructions using immediate addressing for the source operand and register direct addressing for the destination operand would load the block addresses into registers A0 (source block) and A1 (destination block).
The next instruction uses address register indirect with postincrement for source and destination operand. The following operations will occur:

- fetch the long word pointed to by A0 from memory.
- add four (we are loading four bytes) to the contents of register A0
- store the long word into memory at the address indicated by register A1
- add four (we are storing four bytes) to the contents of register A1

When we now start the next instruction both registers point to the next operands (source and destination respectively).

Address Register Indirect with Predecrement:

The general form of a MOVE instruction using address register indirect with predecrement addressing modes for both operands would be

```
MOVE    -(An),-(An)
```

This address mode is similar to the one just described, but the M 68000 first subtracts one, two or four (again depending on the size of the operation) from the contents of the address register and then uses this contents as an address for data access.

The above coding sequence to copy three long words could well have been written using address register indirect with predecrement as follows:

```
MOVE    #$1000,C,A0        fourth item in source block
MOVE    #$2000,C,A0        fourth item in destin. block
MOVE.L   -(A0),-(A1)       decrement addresses and move
MOVE.L   -(A0),-(A1)
MOVE.L   -(A0),-(A1)
```

These two addressing modes, the address register indirect with postincrement or predecrement allow easily to maintain stacks with push and pull operations and to manipulate queues. The stack of the M 68000 (system stack or user stack, depending on privilege state) is addressed using address register A7. As a consequence of the M 68000 design it grows from high addresses to low addresses and shrinks from low addresses to high addresses (see in chapter Machine Instructions: JSR and RTS instructions). In other words, to push information on to the stack we use

```
MOVE.W  #$1000,-(A7)    pushes value $1000
```

---

- 13 -
The following operations will occur:

- fetch the word containing the value $1000$ (immediate addressing mode)
- subtract two from the contents of address register A7 (we want to push a word)
- store the word into memory at the address contained in address register A7

The value $1000$ is now stored on the stack and register A7 is pointing to it, that is to say, register A7 always points to last valid entry on the stack which is also called: the top of the stack. If we now want to inspect the word on the top of the stack, we may write

```
TST (A7) TST sets condition codes
```

if however we want to pull the word off the stack, we may write

```
MOVE (A7)+,D0 top of stack => D0
```

Note:

If the information to be pushed on or pulled off the M 68000 stack is a byte, i.e. if we write

```
MOVE.B D0,-(A7) push bits 0-7 from D0
```

or

```
MOVE.B (A7)+,D0 pull byte into bits 0-7 of D0
```

the M 68000 will behave as follows

**push:**

- in order to keep the M 68000 stack pointer aligned on word boundaries, two rather than one will be subtracted from address register A7 although we just want to push one byte
- the byte contained in bits 0-7 of D0 will be stored as upper byte of the memory word register A7 is pointing to, the low order byte of that memory word remains unchanged

**pull:**

- the byte at the memory location pointed to by the stack pointer A7 will be loaded into bits 0-8 of the data register D0
- the stack pointer A7 will be incremented by two in order to keep it aligned on word boundaries
Address Register Indirect with Displacement:

The general form of a MOVE instruction using address register indirect with displacement addressing modes for both operands would be

\[ \text{MOVE} \quad d16(\text{An}),d16(\text{An}) \]

Here the M 68000 uses the contents of the address register as the base address and allows the addition of a signed 16 bit displacement integer to obtain the effective address. Thus we can access an address range of

\[ [\text{<base address> - 32 768} \ldots \text{<base address> + 32 767}] \]

where \text{<base address> is the contents of the address register.}

This addressing mode is particularly useful to handle tables. Assume we have in register A3 the base address of a table containing the addresses of peripheral equipment we have to monitor, each address is supposed to be 32-bits long. If we now want to load the address of the first peripheral equipment contained in that table into register A4, we can write

\[ \text{MOVE.L} \quad 4\times(5-1)(\text{A3}),\text{A4} \]

and could now use address register A4 to read for instance the status register of that equipment.

Note:

* address register A3 not only contains the base address of the table, but it also points to the first element in the table

* each element in the table is four bytes long; in order to access the \(n\)-th element in the table we have to calculate \(4\times(n-1)\) as the displacement to be used

* for clarity of documentation we can leave it to the assembler to evaluate such an expression

Address Register Indirect with Index and Displacement:

The general form of a MOVE instruction using address register indirect with index and displacement addressing modes for both operands would be

\[ \text{MOVE} \quad d8(\text{An},\text{Rn.S}),d8(\text{An},\text{Rn.S}) \]

where the optional size designator \".S\" for the register Rn could be either \".W\" or \".L\".
The M68MIL assembler will calculate the displacement of "<label symbol>" (see chapter: Assembler Notations) from the current value of the program counter and use this to form the correct instruction code. The M 68000 uses the contents of the program counter as the base address and adds the 16-bit sign extended displacement (range: [-32768 .. 32767]) to it to form the effective address to be used.

Program Counter Relative with Index and Displacement:

The general form of a MOVE instruction using program counter relative with index and displacement addressing mode for the source operand would be

MOVE <label symbol>(PC,Rn.S),D0

where the optional size designator ".S" for the register Rn could be either ".W" or ".L".

The M68MIL assembler will calculate the displacement of "<label symbol>" (see chapter: Assembler Notations) from the current value of the program counter and use this to form the correct instruction code. In addition to adding a signed integer displacement, here restricted to the range [-128 .. +127], it allows to add the contents of either a data register or an address register. This contents might be the entire register (size designator ".L") or just the sign extended low order word (default or size designator ".W") of the register.

Let us now terminate the description of the M 68000 addressing modes with a more complicated example using the program counter relative addressing modes. We are in a subprogram where we have to take a multi-way branch, here according to a request code; one might see it as a computed GOTO in FORTRAN, or a CASE statement in PASCAL. The request code on which we have to take the branch is of size byte and on top of the stack. The start addresses of the program parts to be executed are kept in a table, here labelled "CtRequest". The strategy now is

• load the request code, extend it to word size and verify it
• multiply it by two to allow access of table elements, which are kept as words
• load the start address of the program part to be executed according to the request code
• transfer control to it using a JMP instruction

We use program counter relative addressing in order to access the table. But in addition we stored the offsets of the program parts in the table, rather than their direct address (we would have had to allocate 32-bit table entries to hold their direct address). Thus we again have to use program counter relative
addressing when we jump to it.

```
...  
MOVE.B (A7),D0 load request code
EXT.W D0 extend to word size
BLT.S Aci_Ct1 bad request code
CMPI.W #CtMax,D0 compare against maximum code possible
BGT.S Aci_Ct1 bad request code
LSL.W #1,D0 shift to allow word access
MOVE.W CtRequest(PC,D0.W),D1 load handler offset
JMP    CtRequest(PC,D1) and call it
CtRequest DC.W Aci_Init-CtRequest reset hardware and software
DC.W Aci_Ct10-CtRequest receiver wake up
DC.W Aci_Ct15-CtRequest put receiver to sleep
DC.W Aci_Ct20-CtRequest transmitter wake up
DC.W Aci_Ct25-CtRequest put transmitter to sleep
IFNE   (X-CtRequest-2)/2-CtMax,1  
FAIL   'Control requests listed do not correspond with CtMax'

Aci_Ctl MOVEQ #BadCtReq,D0 set bad control request
RTS    and return
...  
```

The example is quoted from the control part of a driver handling a Motorola M 6850 ACIA chip to communicate with a terminal. Explanations of the various machine and assembler instructions used can be found in the corresponding chapters of this user's guide. The conditional assembly part used at the end of the table should ensure that the table contains entries for all possible correct values of the request code.
2.5 MACHINE INSTRUCTIONS

This section lists the M 68000 instruction set in alphabetic order. The description of each instruction has been abbreviated to the absolute minimum necessary for its use.

**ABCD**

*Add Decimal With Extend*

**ABCD**

Syntax:

ABCD \(D_x, D_y\)  
\(\{D_x\} + \{D_y\} + X \Rightarrow D_y\)

ABCD \(-(A_x), -(A_y)\)  
\(-\{A_x\} + -\{A_y\} + X \Rightarrow \{A_y\}\)

Attributes: byte

Condition codes:

\[\begin{align*}
N & : \text{Undefined.} \\
Z & : \text{Set if result is non-zero. Unchanged otherwise.} \\
V & : \text{Undefined.} \\
C & : \text{Set if a carry (decimal) is generated. Cleared otherwise.} \\
X & : \text{Same as carry.}
\end{align*}\]

**ADD**

*Add binary*

**ADD**

Syntax:

ADD \(<e1>, D_n\)  
\(\{<e1>\} + \{D_n\} \Rightarrow D_n\)

ADD \(D_n, <e2>\)  
\(\{D_n\} + \{<e2>\} \Rightarrow <e2>\)

\(<e1>:\)

<table>
<thead>
<tr>
<th>D_n</th>
<th>An</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immed.</td>
</tr>
</tbody>
</table>

Except for address register direct if operand size is byte.

\(<e2>:\)

<table>
<thead>
<tr>
<th>--</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Attributes: byte, word, long

Condition codes:

\[\begin{align*}
N & : \text{Set if the result is negative. Cleared otherwise.} \\
Z & : \text{Set if the result is zero. Cleared otherwise.} \\
V & : \text{Set if overflow is generated. Cleared otherwise.} \\
C & : \text{Set if a carry is generated. Cleared otherwise.} \\
X & : \text{Same as carry.}
\end{align*}\]
ADDA

Add Address

Syntax:
ADD <ea>, An

\( [<ea>] + \{An\} \rightarrow An \)

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>An</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.l</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immed.</td>
<td></td>
</tr>
</tbody>
</table>

Attributes: word, long

Condition codes:
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.
X: Not affected.

ADDI

Add Immediate

Syntax:
ADD $<data>, <ea>

\( [<ea>] + <data> \rightarrow <ea> \)

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.l</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>

Attributes: byte, word, long

Condition codes:
N: Set if the result is negative. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Set if overflow is generated. Cleared otherwise.
C: Set if a carry is generated. Cleared otherwise.
X: Same as carry.

ADDQ

Add Quick

Syntax:
ADDQ $<data>, <ea>

\( [<ea>] + <data> \rightarrow <ea> \)

<data> : 1..8

- 21 -
\texttt{\textless ea\textgreater :} \\
\begin{tabular}{|c|c|c|c|c|}
\hline
Dn & An & (An) & (An)+ & -(An) & d16(An) \\
\hline
d8(An,Rn) & abs.w & abs.1 & \text{---} & \text{---} & \text{---} \\
\hline
\end{tabular}

Except for address register direct if operand size is byte.

Attributes: byte, word, long

Condition codes:
- \textit{N}: Set if the result is negative. Cleared otherwise.
- \textit{Z}: Set if the result is zero. Cleared otherwise.
- \textit{V}: Set if overflow is generated. Cleared otherwise.
- \textit{C}: Set if a carry is generated. Cleared otherwise.
- \textit{X}: Same as carry.

\texttt{ADDX} \hspace{1cm} \textbf{Add Extended} \hspace{1cm} \texttt{ADDX}

Syntax:
- \texttt{ADDX Dy} \quad \{Dx\}+\{Dy\}+X \Rightarrow Dy
- \texttt{ADDX -(Ax),-(Ay)} \quad \{-(Ax)\} + \{-(Ay)\} + X \Rightarrow \{Ay\}

Attributes: byte, word, long

Condition codes:
- \textit{N}: Set if the result is negative. Cleared otherwise.
- \textit{Z}: Set if the result is zero. Unchanged otherwise.
- \textit{V}: Set if overflow is generated. Cleared otherwise.
- \textit{C}: Set if a carry is generated. Cleared otherwise.
- \textit{X}: Same as carry.

\texttt{AND} \hspace{1cm} \textbf{AND Logical} \hspace{1cm} \texttt{AND}

Syntax:
- \texttt{AND <ea1>,Dn} \quad \{<ea1>\}.AND.(Dn) \Rightarrow Dn
- \texttt{AND Dn,<ea2>} \quad \{Dn\}.AND.\{<ea2>\} \Rightarrow <ea2>

\texttt{<ea1>:} \\
\begin{tabular}{|c|c|c|c|c|}
\hline
Dn & ---- & (An) & (An)+ & -(An) & d16(An) \\
\hline
d8(An,Rn) & abs.w & abs.1 & d16(PC) & d8(PC,Rn) & \text{immed.} \\
\hline
\end{tabular}

\texttt{<ea2>:} \\
\begin{tabular}{|c|c|c|c|c|}
\hline
- & ---- & (An) & (An)+ & -(An) & d16(An) \\
\hline
d8(An,Rn) & abs.w & abs.1 & \text{---} & \text{---} & \text{---} \\
\hline
\end{tabular}
Attributes: byte, word, long

Condition codes:
N: Set if most significant bit of result is set. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Always cleared.
C: Always cleared.
X: Not affected.

**ANDI**

**Syntax:**
ANDI #<data>,<ea>     [<ea>].AND.<data> => <ea>
ANDI #<data>,CCR     [CCR].AND.<data> => CCR
ANDI #<data>,SR      [SR].AND.<data> => SR (privileged)

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>An</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>---</td>
<td></td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Attributes: byte, word, long

Condition codes:
N: Set if most significant bit of result is set. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Always cleared.
C: Always cleared.
X: Not affected.

When accessing the status or condition code register the condition codes are affected according to the operation.

**ASL**

**Arithmetic Shift Left**

**ASR**

**Arithmetic Shift Right**

**Syntax:**
A5d  Dx,Dy       [Dy] shifted by [Dx] bits => Dy
A5d  #<count>,Dx   [Dx] shifted by <count> bits => Dx
A5d   <ea>          [<ea>] shifted by one bit => <ea>

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>--</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>---</td>
<td></td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
Notes:
ASL: zero shifted in
most significant bit shifted out into C and X
ASR: most significant bit replicated into high order bits
least significant bit shifted out into C and X

Attributes: byte, word, long

Condition codes:
N: Set if most significant bit of result is set. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Set if the most significant bit is changed at any time during
   the shift operation. Cleared otherwise.
C: Last bit shifted out of operand, cleared if shiftcount = 0.
X: Last bit shifted out of operand, unaffected if shiftcount = 0.

Bcc Branch Conditionally Bcc

Syntax:
Bcc <label> If cc then <label> => PC
c:
  CC: Carry clear
  CS: Carry set
  EQ: Equal
  F: false
  GE: Greater or equal
  GT: Greater than
  HI: High
  LE: Less or equal
  LS: Low or same
  LT: Less than
  MI: Minus
  NE: Not equal
  PL: Plus
  T: true
  VC: Overflow clear
  VS: Overflow set

Attributes: byte, word

Condition codes:
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.
X: Not affected.
BCHG

Test a bit and change

Syntax:
BCHG Dn,<ea>
BCHG #<data>,<ea>

\[ \text{NOT.\{bit\} of <ea>} \rightarrow Z \]
\[ \text{NOT.\{bit\} of <ea>} \rightarrow \text{bit\} of <ea}\]

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
</tbody>
</table>

Attributes: byte, long

Condition codes:
N: Not affected.
Z: Set if the tested bit is zero. Cleared otherwise.
V: Not affected.
C: Not affected.
X: Not affected.

BCLR

Test a bit and clear

Syntax:
BCLR Dn,<ea>
BCLR #<data>,<ea>

\[ \text{NOT.\{bit\} of <ea>} \rightarrow Z \]
\[ 0 \rightarrow \text{bit\} of <ea}\]

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
</tbody>
</table>

Attributes: byte, long

Condition codes:
N: Not affected.
Z: Set if the bit tested is zero. Cleared otherwise.
V: Not affected.
C: Not affected.
X: Not affected.
BRA

Branch always

Syntax:
BRA <label> <label> => PC

Attributes: byte, word

Condition codes:
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.
X: Not affected.

BSET

Test a Bit and Set

Syntax:
BSET Dn,<ea>
BSET #<data>,<ea>

.NOT.(<bit #> of <ea>) => Z
1 => <bit #> of <ea>

<ea>:

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.l</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Attributes: byte, long

Condition codes:
N: Not affected.
Z: Set if the bit tested is zero. Cleared otherwise.
V: Not affected.
C: Not affected.
X: Not affected.

BSR

Branch to subroutine

Syntax:
BSR <label> {PC} => -(SP); <label> => PC

Attributes: byte, word

Condition codes:
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.
X : Not affected.

**BTST**

*Test a Bit*

**Syntax:**
- **BTST Dn,<ea>**
- **BTST #<data>,<ea>**

```
.NOT.(<bit #> of <ea>) => Z
```

<table>
<thead>
<tr>
<th>&lt;ea&gt; :</th>
<th>Dn</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>

**Attributes:** byte, long

**Condition codes:**
- **N** : Not affected.
- **Z** : Set if the bit tested is zero. Cleared otherwise.
- **V** : Not affected.
- **C** : Not affected.
- **X** : Not affected.

**CHK**

*Check Register against Bounds*

**Syntax:**
- **CHK <ea>,Dn**

```
If (Dn)<0.OR.(Dn)>{<ea>} then TRAP
```

<table>
<thead>
<tr>
<th>&lt;ea&gt; :</th>
<th>Dn</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
</table>
| d8(An,Rn) | abs.w | abs.1 | d16(PC) | d8(PC,Rn) | imm |}

**Attributes:** word

**Condition codes:**
- **N** : Set if (Dn)<0. Cleared if (Dn){<ea>}. Undefined otherwise.
- **Z** : Undefined.
- **V** : Undefined.
- **C** : Undefined.
- **X** : Not affected.
CLR

Clear an Operand

Syntax:

CLR <ea> 0 => <ea>

<ea>:

<table>
<thead>
<tr>
<th></th>
<th>Dn</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Note:

CLR first performs a read cycle before clearing <ea> with a write cycle. (Do not use to clear device registers!)

Attributes: byte, word, long

Condition codes:

N: Always cleared.
Z: Always set.
V: Always cleared.
C: Always cleared.
X: Not affected.

CMP

Compare

Syntax:

CMP <ea>, Dn (Dn) - <ea>

<ea>:

<table>
<thead>
<tr>
<th></th>
<th>Dn</th>
<th>An</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immad.</td>
<td></td>
</tr>
</tbody>
</table>

Except for address register direct if operand size is byte.

Attributes: byte, word, long

Condition codes:

N: Set if the result is negative. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Set if overflow is generated. Cleared otherwise.
C: Set if a borrow is generated. Cleared otherwise.
X: Not affected.
CMPA

Compare address

Syntax:

\[ \text{CMPA } <ea>,\text{An} \quad \{\text{An}\} - \{<ea>\} \]

\<ea\>:

<table>
<thead>
<tr>
<th></th>
<th>Dn</th>
<th>An</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immed.</td>
<td></td>
</tr>
</tbody>
</table>

Attributes: word, long

Condition codes:

N: Set if the result is negative. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Set if overflow is generated. Cleared otherwise.
C: Set if a borrow is generated. Cleared otherwise.
X: Not affected.

CMPI

Compare Immediate

Syntax:

\[ \text{CMP } #<data>,<ea> \quad \{<ea}\} - <data> \]

\<ea\>:

<table>
<thead>
<tr>
<th></th>
<th>Dn</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Attributes: byte, word, long

Condition codes:

N: Set if the result is negative. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Set if overflow is generated. Cleared otherwise.
C: Set if a borrow is generated. Cleared otherwise.
X: Not affected.

CMPM

Compare Memory

Syntax:

\[ \text{CMP } (Ax)+,(Ay)+ \quad \{(Ay)\} - \{(Ax)\} \]

Attributes: byte, word, long
SOFTWARE SUPPORT FOR MOTOROLA 68000  CERN
M68MIL CROSS MACRO ASSEMBLER  M 68000 PROGRAMMING MODEL

Condition codes:
N: Set if the result is negative. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Set if overflow is generated. Cleared otherwise.
C: Set if a borrow is generated. Cleared otherwise.
X: Not affected.

DBcc  Test Condition  Decrement and Branch  DBcc

Syntax:
DBcc  Dn,<label>  If .NOT.(cc) then
          (Dn) - 1 => Dn
          If (Dn)<>-1 then <label> => PC
else continue with next instruction

cc:
CC: Carry clear  .NOT. C
CS: Carry set    C
EQ: Equal       Z
F: false        0
GE: Greater or equal
              (.NOT.(C).AND..NOT.(V)) .OR.
              (.NOT.(N).AND..NOT.(V))
GT: Greater than
              (.NOT.(C).AND..NOT.(V)) .OR.
              (.NOT.(N).AND..NOT.(V))
HI: High        .NOT.(C).AND..NOT.(Z)
LE: Less or equal
              Z .OR. (.NOT.(C).AND..NOT.(V)) .OR.
              (.NOT.(N).AND.V)
LS: Low or same C .OR. Z
LT: Less than
              (.NOT.(C).AND..NOT.(V)) .OR.
              (.NOT.(N).AND.V)
MI: Minus       N
NE: Not equal   .NOT. Z
PL: Plus        .NOT. N
T: true         1
VC: Overflow clear
              .NOT. V
VS: Overflow set V

Condition codes:
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.
X: Not affected.

DIVS  Signed divide  DIVS

Syntax:
DIVS  <ea>,Dn  (Dn)/{<ea>} => Dn
Quotient in lower word.
Remainder in upper word.
Division by zero causes trap.
If overflow then condition flagged

- 30 -
but operands unaffected.

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>-----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.l</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immed.</td>
</tr>
</tbody>
</table>

Attributes: word

Condition codes:
N: Set if the result is negative. Cleared otherwise.
Undefined if overflow.
Z: Set if the result is zero. Cleared otherwise.
Undefined if overflow.
V: Set if division overflow is detected. Cleared otherwise.
C: Always cleared.
X: Not affected.

DIVU

Unsigned Divide

Syntax:
DIVU <ea>,Dn

{Dn}/<{ea}> => Dn
Same remarks as DIVS

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>-----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.l</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immed.</td>
</tr>
</tbody>
</table>

Attributes: word

Condition codes:
N: Set if most significant bit of result is set. Cleared otherwise.
Undefined if overflow.
Z: Set if the result is zero. Cleared otherwise.
Undefined if overflow.
V: Set if division overflow. Cleared otherwise.
C: Always cleared.
X: Not affected.

EOR

Exclusive OR Logical

Syntax:
EOR Dn,<ea>

{Dn} .EXOR. {<ea>} => <ea>
<ea> :  

<table>
<thead>
<tr>
<th></th>
<th>Dn</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Attributes : byte, word, long

Condition codes :
N : Set if most significant bit of result is set. Cleared otherwise.
Z : Set if the result is zero. Cleared otherwise.
V : Always cleared.
C : Always cleared.
X : Not affected.

**EORI**

*Exclusive OR Immediate*

**Syntax :**
EORI #<data>,<ea>  
EORI #<data>,CCR  
EORI #<data>,SR

**<ea> :**  

<table>
<thead>
<tr>
<th></th>
<th>Dn</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Attributes : byte, word, long

Condition codes :
N : Set if most significant bit of result is set. Cleared otherwise.
Z : Set if the result is zero. Cleared otherwise.
V : Always cleared.
C : Always cleared.
X : Not affected.

When accessing the status or condition code register the condition codes are affected according to the operation.

**EXG**

*Exchange Registers*

**Syntax :**
EXG Xx,Xy

Attributes : long
Condition codes:
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.
X: Not affected.

EXT

Syntax:
EXT Dn
If word then bit [7] => into bits [15,8]
If long then bit [15] => into bits [31,16]

Attributes: word, long

Condition codes:
N: Set if the result is negative. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Always cleared.
C: Always cleared.
X: Not affected.

JMP

Syntax:
JMP <ea> {<ea>} => PC

Attributes: unsized

Condition codes:
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.
X: Not affected.

JSR

Syntax:
JSR <ea> {PC} => -(SP) ; {<ea>} => PC

- 33 -
<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>---</th>
<th>----</th>
<th>(An)</th>
<th>---</th>
<th>---</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>

Attributes: unsized

Condition codes:
- N: Not affected.
- Z: Not affected.
- V: Not affected.
- C: Not affected.
- X: Not affected.

**LEA**

*Load Effective Address*

Syntax:

```
LEA <ea>,An
<ea> => An
```

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>---</th>
<th>----</th>
<th>(An)</th>
<th>---</th>
<th>---</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>

Attributes: long

Condition codes:
- N: Not affected.
- Z: Not affected.
- V: Not affected.
- C: Not affected.
- X: Not affected.

**LINK**

*Link and Allocate*

Syntax:

```
LINK An,#<d16>
{An} => -(SP); {SP} => An; {SP}+d16 => PC
```

Attributes: unsized

Condition codes:
- N: Not affected.
- Z: Not affected.
- V: Not affected.
- C: Not affected.
- X: Not affected.
**LSL**  
Logical Shift Left  
LSL

**LSR**  
Logical Shift Right  
LSR

**Syntax:**
- L5d Dx, Dy  \( \text{[Dy] shifted by [Dx] bits} \Rightarrow Dy \)
- L5d #<count>, Dx  \( \text{[Dx] shifted by <count> bits} \Rightarrow Dx \)
- L5d <ea>  \( \text{<ea>} \text{ shifted by one bit} \Rightarrow <ea> \)

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{<ea>} & -- & ---- & (An) & (An)+ & -(An) & d16(An) \\
\hline
d8(An,Rn) & abs.w & abs.1 & -- & -- & -- \\
\hline
\end{array}
\]

**Notes:**
- LSL: zero shifted in  
  most significant bit shifted out into C and X
- LSR: zero shifted in  
  least significant bit shifted out into C and X

**Attributes:** byte, word, long

**Condition codes:**
- N: Set if the result is negative. Cleared otherwise.
- Z: Set if the result is zero. Cleared otherwise.
- V: Always cleared.
- C: Last bit shifted out. Cleared if count = 0.
- X: Last bit shifted out. Unaffected if count = 0.

**MOVE**  
Move Data from Source to Destination  
MOVE

**Syntax:**
- MOVE <ea1>, <ea2>  \( \{<ea1>\} \Rightarrow <ea2> \)

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{Dn} & \text{An} & (An) & (An)+ & -(An) & d16(An) \\
\hline
d8(An,Rn) & abs.w & abs.1 & d16(PC) & d8(PC,Rn) & \text{immed.} \\
\hline
\end{array}
\]

Except for address register direct if operand size is byte.

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{Dn} & ---- & (An) & (An)+ & -(An) & d16(An) \\
\hline
d8(An,Rn) & abs.w & abs.1 & -- & -- & -- \\
\hline
\end{array}
\]

**Attributes:** byte, word, long
Condition codes:
N: Set if the result is negative. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Always cleared.
C: Always cleared.
X: Not affected.

MOVE

Syntax:
a. MOVE <ea1>,CCR  
   {<ea1>} => CCR
b. MOVE <ea1>,SR    
   {<ea1>} => SR (privileged)
c. MOVE SR,<ea2>    
   {SR} => <ea2>
d. MOVE USP,An      
   {USP} => An (privileged)
e. MOVE An,USP      
   {An} => USP (privileged)

<ea1>:

<table>
<thead>
<tr>
<th>Dn</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immed.</td>
</tr>
</tbody>
</table>

<ea2>:

<table>
<thead>
<tr>
<th>Dn</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
</tbody>
</table>

Attributes: a_, b_, c_: word; d_, e_: long

Condition codes:
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.
X: Not affected.

When accessing the status or condition code register the condition codes are affected according to the operation.

MOVEA

Syntax:
MOVEA <ea>,An  
{<ea>} => An

- 36 -
<ea>:

<table>
<thead>
<tr>
<th></th>
<th>Dn</th>
<th>An</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immed.</td>
<td></td>
</tr>
</tbody>
</table>

Attributes: word, long

Condition codes:

N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.
X: Not affected.

MOVEM

Move Multiple Registers

Syntax:

MOVEM <reg. list>,<ea1>  [Registers] => <ea1>
MOVEM <ea2>,<reg. list>  [<ea2>] => Registers

<ea1>:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(An)</th>
<th></th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<ea2>:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(An)</th>
<th>(An)+</th>
<th></th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td></td>
</tr>
</tbody>
</table>

<reg. list>: Ri/Rj/... or Ri-Rj or any combination thereof

Attributes: word, long

Condition codes:

N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.
X: Not affected.
**MOVEP**  
Move peripheral data

Syntax:
- MOVEP Dx,d16(Ay)  
  \[Dx \Rightarrow d16(Ay)\]
- MOVEP d16(Ax),Dy  
  \[d16(Ay) \Rightarrow Dy\]

Notes:
Data is transferred between a data register and alternate bytes of memory. The high order byte of the data register is transferred first.

Attributes: word, long

**MOVEQ**  
Move Quick

Syntax:
- MOVEQ #<data>,Dn  
  \[<data> \Rightarrow Dn\]
  \[<data> : [ -128 .. 127 ] (sign extended)\]

Attributes: long

Condition codes:
- N: Set if the result is negative. Cleared otherwise.
- Z: Set if the result is zero. Cleared otherwise.
- V: Always cleared.
- C: Always cleared.
- X: Not affected.

**MULS**  
Signed Multiply

Syntax:
- MULS <ea>,Dn  
  \[(DN) \times (<ea>) \Rightarrow Dn\]

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immem.</td>
</tr>
</tbody>
</table>

Note: 16 bits \(\times\) 16 bits \(\Rightarrow\) 32 bits.

Attributes: word
Condition codes:
N: Set if the result is negative. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Always cleared.
C: Always cleared.
X: Not affected.

MULU

UnSigned Multiply

Syntax:
MULU <ea>,Dn \[\{DN\} \times \{<ea>\} \Rightarrow Dn\]

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>-----</th>
<th>(An)</th>
<th>(An)⁺</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immed.</td>
<td></td>
</tr>
</tbody>
</table>

Note: 16 bits \times 16 bits \Rightarrow 32 bits.

Attributes: word

Condition codes:
N: Set if most significant bit of result is set. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Always cleared.
C: Always cleared.
X: Not affected.

NBCD

Negate Decimal With Extend

Syntax:
NBCD <ea> \[0 \rightarrow \{<ea>\} \rightarrow X \rightarrow <ea>\]

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>-----</th>
<th>(An)</th>
<th>(An)⁺</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Attributes: byte

Condition codes:
N: Undefined.
Z: Cleared if result is non-zero. Unchanged otherwise.
V: Undefined.
C: Set if borrow (decimal). Cleared otherwise.
X: Same as carry.
NEG

Syntax:

\[
\text{NEG} \quad \langle ea \rangle \\
\quad 0 - \{\langle ea \rangle \} \Rightarrow \langle ea \rangle
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
\langle ea \rangle & Dn & \text{----} & (An) & (An)+ & -(An) & d16(An) \\
\hline
\text{d8(An,Rn)} & \text{abs.w} & \text{abs.1} & \text{---} & \text{---} & \text{---} \\
\hline
\end{array}
\]

Attributes: byte, word, long

Condition codes:

- **N**: Set if the result is negative. Cleared otherwise.
- **Z**: Set if the result is zero. Cleared otherwise.
- **V**: Set if overflow is generated. Cleared otherwise.
- **C**: Set if borrow. Cleared otherwise.
- **X**: Same as carry.

NEGX

Syntax:

\[
\text{NEGX} \quad \langle ea \rangle \\
\quad 0 - \{\langle ea \rangle \} - X \Rightarrow \langle ea \rangle
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\langle ea \rangle & Dn & \text{----} & (An) & (An)+ & -(An) & d16(An) \\
\hline
\text{d8(An,Rn)} & \text{abs.w} & \text{abs.1} & \text{---} & \text{---} & \text{---} \\
\hline
\end{array}
\]

Attributes: byte, word, long

Condition codes:

- **N**: Set if the result is negative. Cleared otherwise.
- **Z**: Set if the result is zero. Unchanged otherwise.
- **V**: Set if overflow is generated. Cleared otherwise.
- **C**: Set if borrow. Cleared otherwise.
- **X**: Same as carry.

NOP

Syntax:

\[
\text{NOP}
\]

Condition codes:

- **N**: Not affected.
- **Z**: Not affected.
- **V**: Not affected.
- **C**: Not affected.
X : Not affected.

**NOT**

**Logical Complement**

**NOT**

**Syntax**:

```
NOT <ea>  .NOT.<<ea>> => <ea>
```

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

**Attributes**: byte, word, long

**Condition codes**:

N : Set if the result is negative. Cleared otherwise.
Z : Set if the result is zero. Cleared otherwise.
V : Always cleared.
C : Always cleared.
X : Not affected.

**OR**

**Inclusive OR Logical**

**OR**

**Syntax**:

```
OR Dn,<ea1>  {[<ea1>].OR.(Dn) => <ea1>}
OR <ea2>,Dn  (Dn).OR.<<ea2>> => Dn
```

<table>
<thead>
<tr>
<th>&lt;ea1&gt;</th>
<th>Dn</th>
<th>An</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;ea2&gt;</th>
<th>Dn</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immed.</td>
<td></td>
</tr>
</tbody>
</table>

**Attributes**: byte, word, long

**Condition codes**:

N : Set if most significant bit of result is set. Cleared otherwise.
Z : Set if the result is zero. Cleared otherwise.
V : Always cleared.
C : Always cleared.
X : Not affected.

- 41 -
**ORI**

Inclusive Or Immediate

Syntax:

ORI #<data>,<ea>  
{<ea>}.OR.<data> => <ea>

ORI #<data>,CCR  
[CCR].OR.<data> => CCR

ORI #<data>,SR  
{SR}.OR.<data> => SR (privileged)

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>Dn</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Attributes: byte, word, long

Condition codes:

N: Set if most significant bit of result is set. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Always cleared.
C: Always cleared.
X: Not affected.

When accessing the status or condition code register the condition codes are affected according to the operation.

**PEA**

Push Effective Address

Syntax:

PEA <ea>  
<ea> => -(SP)

<table>
<thead>
<tr>
<th>&lt;ea&gt;</th>
<th>---</th>
<th>----</th>
<th>(An)</th>
<th>---</th>
<th>---</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>

Attributes: long

Condition codes:

N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.
X: Not affected.
RESET

Reset External Devices

Syntax:
RESET

(privileged)

Attributes: unsized

Condition codes:
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.
X: Not affected.

ROL

Rotate Left without Extend

ROR

Rotate Right without Extend

Syntax:
R0d D0,Dy
{Dy} rotated by (Dx) bits => Dy
R0d #<count>,Dx
{Dx} rotated by <count> bits => Dx
R0d <ea>
{<ea>} rotated by one bit => <ea>

<ea>:

<table>
<thead>
<tr>
<th>d8(An,Rn)</th>
<th>abs.w</th>
<th>abs.l</th>
<th>---</th>
<th>---</th>
<th>---</th>
</tr>
</thead>
</table>

Notes:
ROL: most significant bit shifted out into C and
least significant bit.
ROR: least significant bit shifted out into C and
most significant bit.

Attributes: byte, word, long

Condition codes:
N: Set if most significant bit of result is set. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Always cleared.
C: Set according to the last bit shifted out of the operand.
   Cleared if shift count equal zero.
X: Not affected.
ROXL

Rotate Left with Extend

ROXL

Syntax:

ROXd  Dx, Dy       {Dy} rotated by {Dx} bits => Dy
ROXd  #<count>, Dx  {Dx} rotated by <count> bits => Dx
ROXd  <ea>           {<ea>} rotated by one bit => <ea>

<ea>:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.l</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Notes:

ROXL: X shifted in
most significant bit shifted out into C and X
ROXR: X shifted in
least significant bit shifted out into C and X

Attributes: byte, word, long

Condition codes:

N: Set if most significant bit of result is set. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Always cleared.
C: Set according to the last bit shifted out of the operand.
    Set to the value of the extend bit for a count of zero.
X: Set according to the last bit shifted out of the operand.
    Unaffected if shift count equal zero.

RTE

Return from Exception

RTE

Syntax:

RTE

{(SP)+} => SR ;
{(SP)+} => PC (privileged)

Attributes: unsized

Condition codes:

N: Set according to the content of the word on the stack
Z: Set according to the content of the word on the stack
V: Set according to the content of the word on the stack
C: Set according to the content of the word on the stack
X: Set according to the content of the word on the stack
RTR  

*Return and Restore Condition Codes*

**Syntax:**

\[
\text{RTR} \quad \{(SP)+\} \Rightarrow \text{CC} ; \{(SP)+\} \Rightarrow \text{PC}
\]

**Attributes:** unsized

**Condition codes:**

- N: Set according to the content of the word on the stack
- Z: Set according to the content of the word on the stack
- V: Set according to the content of the word on the stack
- C: Set according to the content of the word on the stack
- X: Set according to the content of the word on the stack

RTS  

*Return from Subroutine*

**Syntax:**

\[
\text{RTS} \quad \{(SP)+\} \Rightarrow \text{PC}
\]

**Attributes:** unsized

**Condition codes:**

- N: Not affected.
- Z: Not affected.
- V: Not affected.
- C: Not affected.
- X: Not affected.

SBCD  

*Subtract Decimal with Extend*

**Syntax:**

\[
\text{SBCD} \quad \text{Dx,Dy} \quad \{\text{Dy}\} - \{\text{Dx}\} - \text{X} \Rightarrow \text{Dy}
\]
\[
\text{SBCD} \quad -\{\text{Ax}\},-\{\text{Ay}\} \quad \{-\{\text{Ay}\}\} - \{-\{\text{Ax}\}\} \Rightarrow \text{Ay}
\]

**Attributes:** byte

**Condition codes:**

- N: Undefined.
- Z: Cleared if result is non-zero. Unchanged otherwise.
- V: Undefined.
- C: Set if borrow (decimal). Cleared otherwise.
- X: Same as carry.

Scc  

*Set According to Condition*

**Syntax:**

\[
\text{Scc} \quad \text{<ea>} \quad \text{If cc then 1's} \Rightarrow \text{<ea>}
\]
\[
\text{cc} ~ \text{Else 0's} \Rightarrow \text{<ea>}
\]

\[
-45-
\]
CC : Carry clear .NOT. C
CS : Carry set C
EQ : Equal Z
F : false 0
GE : Greater or equal ( N.AND.V ).OR.
     ( .NOT.(N).AND..NOT.(V) )
GT : Greater than ( N.AND.V.AND..NOT.(Z) ).OR.
     ( .NOT.(N).AND..NOT.(V).AND..NOT.(Z) )
HI : High .NOT.(C).AND..NOT.(Z)
LE : Less or equal Z.OR.(N.AND..NOT.(V)).OR.(.NOT.(N).AND.V)
LS : Low or same C.OR.Z
LT : Less than ( N.AND..NOT.(V) ).OR.( .NOT.(N).AND.V )
MI : Minus N
NE : Not equal .NOT. Z
PL : Plus .NOT. N
T : true 1
VC : Overflow clear .NOT. V
VS : Overflow set V

\<ea\> :  
<table>
<thead>
<tr>
<th></th>
<th>Dn</th>
<th>---</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.l</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>

Attributes : byte

Condition codes :
N : Not affected.
Z : Not affected.
V : Not affected.
C : Not affected.
X : Not affected.

STOP

Load Status Register and Stop

STOP

Syntax :
STOP #xx  
xx => SR ; STOP (privileged)

Attributes : unsized

Condition codes :
N : Set according to the immediate operand.
Z : Set according to the immediate operand.
V : Set according to the immediate operand.
C : Set according to the immediate operand.
X : Set according to the immediate operand.
**SUB**

*Subtract binary*

**Syntax:**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SUB &lt;ea1&gt;,Dn</code></td>
<td><code>(Dn) - (&lt;ea1&gt;) =&gt; Dn</code></td>
</tr>
<tr>
<td><code>SUB Dn,&lt;ea2&gt;</code></td>
<td><code>(&lt;ea2&gt;) - (Dn) =&gt; &lt;ea2&gt;</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>&lt;ea1&gt;</code>:</th>
<th>Dn</th>
<th>An</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immed.</td>
<td></td>
</tr>
</tbody>
</table>

Except for address register direct if operand size is byte.

<table>
<thead>
<tr>
<th><code>&lt;ea2&gt;</code>:</th>
<th>--</th>
<th>----</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

**Attributes:** byte, word, long

**Condition codes:**

- **N:** Set if the result is negative. Cleared otherwise.
- **Z:** Set if the result is zero. Cleared otherwise.
- **V:** Set if overflow is generated. Cleared otherwise.
- **C:** Set if borrow. Cleared otherwise.
- **X:** Same as carry.

**SUBA**

*Subtract Address*

**Syntax:**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SUBA &lt;ea&gt;,An</code></td>
<td><code>(An) - (&lt;ea&gt;) =&gt; An</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>&lt;ea&gt;</code>:</th>
<th>Dn</th>
<th>An</th>
<th>(An)</th>
<th>(An)+</th>
<th>-(An)</th>
<th>d16(An)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d8(An,Rn)</td>
<td>abs.w</td>
<td>abs.1</td>
<td>d16(PC)</td>
<td>d8(PC,Rn)</td>
<td>immed.</td>
<td></td>
</tr>
</tbody>
</table>

**Attributes:** word, long

**Condition codes:**

- **N:** Not affected.
- **Z:** Not affected.
- **V:** Not affected.
- **C:** Not affected.
- **X:** Not affected.

---

- 47 -
\textbf{SUBI}  \hspace{1cm} \textit{Subtract Immediate} \hspace{1cm} \textbf{SUBI}

**Syntax:**
\begin{verbatim}
  SUBI  #<data>,<ea>   [(<ea>) - <data> => <ea>]
\end{verbatim}

\begin{tabular}{|c|c|c|c|c|}
\hline
  <ea>  & Dn & ---- & (An) & (An)+ & -(An) & d16(An) \\
\hline
  d8(An,Rn) & abs.w & abs.1 & --- & --- & --- \\
\hline
\end{tabular}

**Attributes:** byte, word, long

**Condition codes:**
- N: Set if the result is negative. Cleared otherwise.
- Z: Set if the result is zero. Cleared otherwise.
- V: Set if overflow is generated. Cleared otherwise.
- C: Set if borrow. Cleared otherwise.
- X: Same as carry.

\textbf{SUBQ}  \hspace{1cm} \textit{Subtract Quick} \hspace{1cm} \textbf{SUBQ}

**Syntax:**
\begin{verbatim}
  SUBQ  #<data>,<ea>   [(<ea>) - <data> => <ea>]
  <data> : 1..8
\end{verbatim}

\begin{tabular}{|c|c|c|c|c|}
\hline
  <ea>  & Dn & An & (An) & (An)+ & -(An) & d16(An) \\
\hline
  d8(An,Rn) & abs.w & abs.1 & --- & --- & --- \\
\hline
\end{tabular}

Except for address register direct if operand size is byte.

**Attributes:** byte, word, long

**Condition codes:**
- N: Set if the result is negative. Cleared otherwise.
- Z: Cleared if result is non-zero. Cleared otherwise.
- V: Set if overflow is generated. Cleared otherwise.
- C: Set if a carry is generated. Cleared otherwise.
- X: Same as carry.

\textbf{SUBX}  \hspace{1cm} \textit{Subtract with Extend} \hspace{1cm} \textbf{SUBX}

**Syntax:**
\begin{verbatim}
  SUBX  Dx,Dy   {Dy} - {Dx} - X => Dy
  SUBX  -(Ax),-(Ay)  {-(Ay)} - {-(Ax)} => Ay
\end{verbatim}

- 48 -
Attributes: byte, word, long

Condition codes:
N: Set if the result is negative. Cleared otherwise.
Z: Cleared if result is non-zero. Unchanged otherwise.
V: Set if overflow is generated. Cleared otherwise.
C: Set if a carry is generated. Cleared otherwise.
X: Same as carry.

**SWAP**

**Swap Register Halves**

Syntax:
SWAP Dn

\[ \text{bits [31,16]} \leftrightarrow \text{bits [15,0]} \]

Attributes: word

Condition codes:
N: Set if most significant bit of result is set. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Always cleared.
C: Always cleared.
X: Not affected.

**TAS**

**Test and Set an Operand**

Syntax:
TAS <ea>

Test \(<ea>\); 1 => bit 7 of Dn

\[
\begin{array}{|c|c|c|c|c|}
\hline
<ea> & Dn & (An) & (An)+ & -(An) & d16(An) \\
\hline
\text{d8(An,Rn)} & \text{abs.w} & \text{abs.l} & \text{---} & \text{---} & \text{---} \\
\hline
\end{array}
\]

Note:
TAS performs an uninterruptible read-modify-write cycle!

Attributes: byte

Condition codes:
N: Set if most significant bit of result is set. Cleared otherwise.
Z: Set if the result is zero. Cleared otherwise.
V: Always cleared.
C: Always cleared.
X: Not affected.
**TRAP**

Syntax:
```
TRAP  #<vector>
```

Attributes: unsized

Condition codes:
- N: Not affected.
- Z: Not affected.
- V: Not affected.
- C: Not affected.
- X: Not affected.

**TRAPV**

Syntax:
```
TRAPV
```

Attributes: unsized

Condition codes:
- N: Not affected.
- Z: Not affected.
- V: Not affected.
- C: Not affected.
- X: Not affected.

**TST**

Syntax:
```
TST  <ea>
```

Attributes: byte, word, long

Condition codes:
- N: Set if the result is negative. Cleared otherwise.
- Z: Set if the result is zero. Cleared otherwise.
- V: Always cleared.
- C: Always cleared.
X : Not affected.

UNLK

Syntax :
UNLK An

Attributes : unsized

Condition codes :
N : Not affected.
Z : Not affected.
V : Not affected.
C : Not affected.
X : Not affected.
3.2 MICRO SUBSTITUTION

The micro substitution capability enables the programmer to convert the decimal value of a label symbol into the corresponding ASCII character string. Wherever a label symbol between micro marks (tilde "~") appears in a statement line, the assembler substitutes the ASCII string corresponding to the current decimal value of the symbol before it interprets the statement.

The assembler will issue an error message

- if the symbol is not a label symbol
- if the symbol is undefined
- if it does not find the ending micro mark in the current statement line

The assembler will ignore micro substitution

- while skipping information in an IF range
- while scanning the body of a MACRO definition
- while scanning the symbol embraced by the micro marks

Micro substitution is especially useful when tables are to be constructed, using macros to define the table entries.

3.3 SYMBOLS

Identifiers

Identifiers recognized by the assembler consist of one or more characters, the first sixteen of which are significant. The first character must be a letter (A through Z; and a through z) or an underscore ("_"). Each remaining character may be a letter, a digit (0 through 9) or an underscore. The names for registers (A0 through A7; D0 through D7; SP, USP, CCR, SR), instructions (ABCD .. UNLNX) and assembler directives (ALIGN .. TTL) are predefined symbols and may not be redefined by the user. Please note that lower case letters and upper case letters are not distinct in identifiers. This is necessary to avoid conflicts with Pascal generated calls for procedures written in assembler and for symbolic debugging under the MoniCa monitor [6]. The assembler directive DISTINCT however allows the user to override this assembler default.

Numbers

Numbers recognized by the assembler include decimal, hexadecimal and octal values. Decimal numbers are specified by a string of decimal digits (0 through 9); hexadecimal numbers are specified by a dollar sign ("$"), followed by a string of hexadecimal digits (0 through 9, A through F); octal numbers are
specified by a colon ("":"), followed by a string of octal digits (0 through 7).

**Character Strings**

One or more characters enclosed by apostrophes ("'"') constitute a character string. Character strings are left-adjusted and zero-filled (if necessary), whether stored or used as immediate operands. Only strings of four or fewer characters may be used as immediate operands. (In order to specify an apostrophe within a character string, two successive apostrophes must appear where the single apostrophe is intended to appear.)

**Assembler Symbols**

All symbols except absolute symbols are relative, i.e. they represent relocatable addresses. Whenever the assembler encounters a relative symbol it will generate a direct address (see: Direct addressing) and related relocation information. To yield a program counter relative address write

```
<r sym>(PC)   or    <r sym>(PC,<reg>)
```

respectively (see: Program Counter Relative Addressing).

Symbols defined within an assembly as absolute or relative symbols may be exported by occurring in an ENTRY assembler directive, i.e. their value and their type are available to the link editor so that the program may be loaded with others that reference these symbols.

The assembler has four types of symbols:

**Absolute Symbol:**

- The symbol is equated (EQU) or SET to an absolute value.

- The symbol is defined in the absolute section of the program. The start of an absolute section is defined either by an ORG pseudo instruction or by a SECTION assembler directive, the type of which is absolute. Its value is unaffected by any possible future applications of the link editor to the module.

**Relative Symbol:**

- The symbol is equated (EQU) or SET to a relative symbol.

- The symbol is defined in a relative section of the program. The assembler recognizes the following relative sections:
  
  - general section
  - relocatable sections
  - common sections
  - unique sections

- 55 -
The different section types are explained in the chapter on pseudo instructions. The assembler will by default start each assembly in the general section. The RORR assembler directive initializes the general section.

External Symbol:
The symbol is listed in an EXTERN assembler directive and is not defined in the current assembly. Its value is set to zero and must be defined during a subsequent link editor run.

Undefined Symbol:
The symbol is neither defined in the current assembly nor listed in an EXTERN pseudo instruction. The occurrence of such a symbol is indicated as an error.

3.4 OPERATORS

The assembler recognizes three classes of operators:

* Arithmetic Operators*

+  addition
-  subtraction
\*  multiplication
/  division, produces truncated integer result
-  unary minus, causes a term in the expression to be negated or subtracted from zero.

* Shift Operators*

<< shift left causes the left expression to be left-shifted by the number of bits specified in the right expression. The left expression is filled with zeros from the right.

>> shift right causes the left expression to be right-shifted by the number of bits specified in the right expression. The left expression is filled with zeros from the left.

* Logical Operators*

\&  logical AND causes each bit in the left expression to be logically anded with the corresponding bit in the right expression

\|  logical OR causes each bit in left expression to be logically ored with the corresponding bit in the right expression.
Operator Precedence

Expressions are evaluated with the following operator precedence:
- parenthetical expression (innermost first)
- unary minus
- shift
- and, or
- multiplication, division
- addition, subtraction

Operators of the same precedence are evaluated from left to right.

3.5 EXPRESSIONS

An expression is a combination of symbols, constants, algebraic operators, and parentheses. The expression is used to specify a value which is to be used as an operand. Expressions follow the conventional rules of algebra. There are no restrictions in the use of operators between these symbols. It may not be possible for the assembler to fully evaluate an expression containing relative symbols. Such an expression is forwarded in the CUFDM module and will finally be resolved by the loader (pusher).

Some assembler directives like EQU and SET however need expressions, which can be fully evaluated by the assembler. Such expressions are called "simple expressions" and must fulfill the following rules:

- Relative symbols or expressions cannot be multiplied, divided, added, or operated on with the logical operators.

- A relative symbol or expression may have an absolute value added to or subtracted from it. The result is relative.

- A relative symbol or expression may be subtracted from another relative symbol or expression provided they are both defined in the same section. The result is absolute. External symbols do not belong to any specific section.
4 ASSEMBLER DIRECTIVES

Assembler directives (or pseudo instructions) discussed in this chapter are classed according to application as follows:

- Module identification (IDENT and END)
- Section control (ORG, RORG and SECTION)
- Symbol definition (EQU and SET)
- Module linkage (ENTRY and EXTERN)
- Data generation and storage reservation (ALIGN, DC and DS)
- Conditional assembly (ELSE, ENDC, ENDIF, IF, IFC, IFNC and IFcc)
- Macro operations (ENDM, LOCAL, MACRO, MEXIT, NARG)
- Source stream control (DISTINCT, DUP, ENDDUP, INSERT and SYSTEXT)
- Listing control (FAIL, G, LIST, LLEN, NOL, NOLIST, NOPAGE, PAGE, PLEN, SPC, STTL and TTL)
- Object code control (BLONG, BSHORT, FLONG, FSHORT, NOOBJ)
- Date and time stamp (STAMP)
- Assembler maintenance (DEBUG)
- Symbolic debugging (SYMDEBUG)

The next chapter describes the definition and use of MACRO's.

The format description for the pseudo instructions uses symbols which have the following syntactical meaning:

- [...] Enclose optional fields of the pseudo instruction.
- <..> Enclose a symbol, called a 'syntactic variable', e.g. <number>.
4.1 MODULE IDENTIFICATION

IDENT Module Identification

IDENT <symbol>

The symbol defines the name of the CUFOM module.

An IDENT pseudo instruction is the first statement of a module recognized by the assembler. It defines the name to be given to the module.

END End of Module

END [<trasym>]

<trasym>

An optional symbol. If specified, it declares the module to be a main program and the symbol to be the transfer symbol (defining the place at which execution is to start) for the loader in the target machine. The symbol, as well as being quoted in the END pseudo instruction, must be defined in the module, or an assembly error will be generated.

An END pseudo instruction must be the last instruction of each module. It causes the assembler to terminate all counters, conditional assembly, or macro generation. It also causes the CUFOM module to be terminated.

4.2 SECTION CONTROL

Section control pseudo instructions allow the programmer to divide a source module into separately controlled regions of a program, providing him with a means of changing program counters. They establish a new section or resume use of an already established section. The section in use is the section into which code is subsequently assembled. A user may establish up to 16 sections. By default the assembler will always start with the general section using long direct address mode. The assembler basically allows two types of section, namely:

Absolute Section

The following pseudo instructions cause the assembler to establish or resume assembly in the absolute section:

SECTION A,[[<expression>]]
SECTION.L A,[[<expression>]]
SECTION.S A,[[<expression>]]
or (for compatibility with the Motorola assembler)

ORG  [<expression>]
ORG.L [<expression>]
ORG.S [<expression>]

The optional expression may not contain any forward references and must evaluate to an absolute value. It specifies the value to be assigned to the program counter.

If an expression is not defined, the program counter will be set to zero unless an absolute section has already been specified, in which case the program counter will resume with its last value.

If the suffix [.S] is appended to the pseudo instruction, the assembler will assume that as of now any forward reference can be achieved by using the short direct addressing form of the instruction (this might be dangerous if the forward reference is to a symbol in a relative section). If no suffix or the suffix [.L] is appended it will use the long direct addressing form for forward references.

Relative Section

In order to allow structuring of the relative section, the assembler and finally the link editor and pusher provide for different relocatable sections within the relative section. Whenever a new section is established by a SECTION pseudo instruction, the location counter will be set to zero. If the suffix .S is appended to the pseudo instruction establishing the section, the assembler will assume that this section will finally be loaded into low address memory so that direct addressing of the symbols defined in that section may be accomplished through direct short addresses. Whenever an already established section is resumed by a SECTION pseudo, the location counter will be reset to its last value.

Note:
The suffix .S does not imply that forward references in that section will also be resolved with direct short addresses. (A forward reference might lead to an address in another section.) The pseudo instruction FSHORT must be used for this purpose.

G (general) Section:

The section is relocatable and the link editor joins all general sections into a single one in the order in which they are processed. In a module, only one general section can be defined; but in different modules, G-sections may be named differently. The following pseudo instruction cause the assembler to establish or to resume assembly in the G-section:
[<symbol>] SECTION[.S] G[,<expression>]

or (for compatibility with the Motorola assembler)

RORG[.S]

<symbol>
An optional symbol to define a name for the G-section. By default the assembler will start generating code in the general section assuming a blank name. The first subsequent use of a SECTION pseudo instruction for the general section might change the default name.

<expression>
The optional expression may not contain any forward references and must evaluate to an absolute value. It specifies the value to be assigned to the program counter.

If an expression is not defined, the program counter will be set to zero unless a general section has already been specified, in which case the program counter will resume with its last value.

Note:
The general section is established by the assembler and it is assumed that it will not necessarily be loaded in low memory, hence long direct addresses are generated. The user can override this by using a SECTION.S pseudo for the general section prior to any code generation for this section.

R (relocatable) Section:

The section is relocatable. If R-sections that have been processed by the link editor have the same name, they are joined into a single one (with that name) in the order in which they were processed. Unnamed R-sections are not joined. The following pseudo instruction causes the assembler to establish or resume assembly in an R-section:

[<symbol>] SECTION[.S] R[,<expression>]

<symbol>
An optional symbol to define a name for the R-section.

<expression>
The optional expression may not contain any forward references and must evaluate to an absolute value. It specifies the value to be assigned to the program counter.

If an expression is not defined, the program counter will be set to zero unless the R-section has already been specified, in which case the program counter will resume with its last value.
C (common) Section:

The section is relocatable. If C-sections processed by the link editor have the same name, they are overlapped. Unnamed C-sections (blank common) are also overlapped. A warning message is printed by the link editor if the size of a common section is increased.

[<symbol>] SECTION[.S] C[,]<expression>]

<symbol>
An optional symbol to define a name for the C-section.

<expression>
The optional expression may not contain any forward references and must evaluate to an absolute value. It specifies the value to be assigned to the program counter.
If an expression is not defined, the program counter will be set to zero unless the C-section has already been specified, in which case the program counter will resume with its last value.

U (unique) Section:

The section is relocatable and must have a name. If other U-sections processed by the link editor have the same name, a warning message is printed.

<symbol> SECTION[.S] U[,]<expression>]

<symbol>
A mandatory symbol to define the name of the U-section.

<expression>
The optional expression may not contain any forward references and must evaluate to an absolute value. It specifies the value to be assigned to the program counter.
If an expression is not defined, the program counter will be set to zero unless the U-section has already been specified, in which case the program counter will resume with its last value.

"Previous" Section:

SECTION *

The assembler maintains an initial section definition, which is the general section, and places all subsequent section definitions into a window that can hold the last nine definitions. Use of a SECTION * instruction resumes use of the most recent entry and removes it from the window. If there are no more window entries the initial general section will be used. This option of the SECTION pseudo instruction is useful if one wants to switch to different sections in a macro body without knowing the section currently in
use.

4.3 SYMBOL DEFINITION

EQU  

EQU  

Equate Symbol Value  

An EQU pseudo instruction permanently defines the symbol in the location field as having the value and attributes indicated by the simple expression in the variable field.

<symbol> EQU <expression>

A location symbol following the naming rules must be defined.

<expression>

A simple expression with at most one relative symbol in the final expression. Forward references are not allowed.

SET  

SET  

Set or Reset Symbol Value  

A SET pseudo instruction defines the symbol in the location field as having the value and attributes indicated by the simple expression in the variable field. A subsequent SET using the same symbol redefines the symbol to the new value and attributes.

<symbol> SET <expression>

A location symbol following the naming rules must be defined.

<expression>

A simple expression with at most one relative symbol in the final expression. Forward references are not allowed.

4.4 MODULE LINKAGE

The pseudo instructions ENTRY and EXTERN do not define symbols but either declare symbols defined within a module as being available outside the module or declare symbols referred to in the module as being defined outside the module.

ENTRY  

Declare Entry Symbols  

The ENTRY pseudo instruction specifies which of the symbolic addresses defined in the module can be referred to by modules assembled independently; ENTRY lists entry points to the current module.

ENTRY <sym.1>,<sym.2>,...,<sym.n>
<sym.i>
Linkage symbol. Each symbol must be defined in the module as nonexternal
(must not be listed on an EXTERN pseudo instruction).

A list of all entry points declared in the module precedes the assembly listing.

Note:
If the ENTRY pseudo declaring a symbol precedes the actual definition of the
symbol, the assembler will handle the symbol as if it were going to be
defined in the current section. With other words, if the current section is
declared to be loaded in lower memory, absolute addresses for the symbols dec-
clared in the list of the ENTRY pseudo instruction will be short addresses,
else they are long addresses. The user is therefore advised to use the ENTRY
pseudo within the section for which the entry symbols are to be declared.

EXTERN Declare External Symbols EXTERN

The EXTERN pseudo instruction lists symbols that are defined as entry points
in independently assembled modules for which references can appear in the module
being assembled.

EXTERN[.S] <sym.1>,<sym.2>,....,<sym.n>

<sym.i>
Linkage symbol. These symbols must not be defined within the module.

The suffix .S, if appended to the EXTERN pseudo instruction, indicates to the
assembler, that all symbols contained in the variable field will be finally
located in lower memory and can be accessed using the direct short address form.
A list of all external symbols declared in the module precedes the assembly
listing.

4.5 DATA GENERATION AND STORAGE RESERVATION

ALIGN Align on Word Boundary ALIGN

ALIGN
Whenever the assembler finds a label definition in a machine instruction it
will make certain, that the label is given an even address to allow word access.
For the two pseudo instructions
DC.B and DS.B
however bytes are allocated and hence
Label_1 DC.B 1
may well be given an odd address.
The same is true for the use of the EQU pseudo instruction with a label, namely

    Label_2 EQU 'X'

The assembler cannot necessarily know, for which type of access the label will be used, hence it will allocate the current value of the location counter, which might be even or odd. If a label must have an even address, like in the following code sequence

    DC.B 'One'
    ALIGN
    Label_3 EQU 'X'

the ALIGN pseudo will make certain that Label_3 has an even address.

**DC**

**Define Constant**

**DC**

```
[<symbol>] DC <opr.1>,<opr.2>,....,<opr.n>
[<symbol>] DC[.B] <opr.1>,<opr.2>,....,<opr.n>
[<symbol>] DC[.W] <opr.1>,<opr.2>,....,<opr.n>
[<symbol>] DC[.L] <opr.1>,<opr.2>,....,<opr.n>
```

A symbol following the naming rules may be defined.

**<opr.i>**

The operand can be a symbol, an ASCII, decimal or hexadecimal value or an expression evaluating to such a value.

The function of the DC pseudo instruction is to define a constant in memory. The DC directive may have one operand, or multiple operands which are separated by commas. The operand field may contain the actual value (decimal, octal, hexadecimal, or character string). Alternatively, the operand may be a symbol or expression. The constant is aligned on a word boundary if word (.W) or long word (.L) is specified, or a byte boundary if byte (.B) is specified.

The following rules apply to size specifications on character strings:

**DC.B**

If an odd number of bytes (characters) are entered, the odd byte on the right will be zero filled unless the next source statement is another DC.B or DS.B. In this case the next DC.B or DS.B will start in the odd byte on the right.

**DC.W**

If an odd number of bytes (characters) are entered, the last word will be zero filled on the right to force an even byte count.

**DC.L**

If less than a multiple of four bytes are entered, the last long word will be zero filled on the right to a multiple of four bytes.
Define Storage

[<symbol>] DS <expression>
[<symbol>] DS[.B] <expression>
[<symbol>] DS[.W] <expression>
[<symbol>] DS[.L] <expression>

A symbol following the naming rules may be defined.

The expression must evaluate to an absolute value. Forward references are not allowed.

The DS pseudo instruction is used to reserve memory locations. The contents of the memory reserved are not initialized in any way. The expression must evaluate to an absolute value. Forward references are not allowed.

4.6 CONDITIONAL ASSEMBLY

The pseudo instructions IF, IFC, IFNC and IFcc permit optional assembly or skipping of source code. The instructions immediately following the test instruction are assembled if the tested condition is true and skipped if the condition is false. Skipping is terminated either by a source statement count on the IF instruction, or by an ENDIF, an ELSE, or an END.

The statement count, when used, is decremented for instruction lines only; comment lines (identified by * in column one) are not counted. Determining the IF range with a statement count produces slightly faster assembly than using the ENDIF.

The result of an IF test is determined by the value of the expression in pass one of the assembler; the value of a relative symbol is relative to the origin of the section in which it was defined. The value of an external symbol is zero if the symbol was declared as external. If the symbol was defined relative to a declared external, the value is the relative value.

IF's may be nested up to ten levels deep.

ENDIF End of IF Range ENDIF

[<if_name>] ENDIF

<if_name>

<if_name>, an optional symbol, defines the name of an IF, IFC, IFNC, IFcc, or ELSE sequence; or blank.
An ENDIF pseudo instruction (or ENDC for compatibility with the Motorola assembler) causes termination of skipping and assembly to resume. When the sequence containing the ENDIF is being assembled, or is controlled by a statement count, the ENDIF has no effect other than to be included in the count.

Skipped instructions such as macro references are not expanded. Thus, any ENDIF that would have resulted from an expansion is not detected.

Skipping of a sequence initiated by an IF, IFC, IFNC, IFcc, or ELSE that is assigned a name is terminated by an ENDIF specifying the same name. Skipping of a sequence initiated by an unnamed IF, IFC, IFNC, IFcc or ELSE is terminated by an unnamed ENDIF.

ELSE

Reverse Effects of IF

ELSE

[<if_name>] ELSE

<if_name>

<if_name>, an optional symbol, defines the name of an IF, IFC, IFNC, IFcc, or ENDIF sequence; or blank.

By means of the ELSE instruction, the assembler provides the facility to reverse the effects of an IF test within the IF range. An ELSE detected during skipping causes assembly to resume at the instruction following the ELSE. An ELSE detected while a sequence is being assembled initiates skipping of source code following the ELSE. Skipping continues until either an END or an ENDIF for the sequence is detected.

An ELSE specifying the sequence by name terminates skipping of a sequence initiated by an IF, IFC, IFNC or IFcc with the same name. An unnamed ELSE terminates skipping of a sequence initiated by an unnamed IF, IFC, IFNC or IFcc. Skipped instructions such as macro references are not expanded; any else that would have resulted from the expansion is not detected.

IF

Test Attribute of Symbol

IF

[<if_name>] IF <attribute>,<symbol>[,<line_count>]

<if_name>

<if_name>, an optional symbol, defines the name of the IF, IFC, IFNC or IFcc sequence; or blank

<attribute>

<attribute> specifies attribute test. A minus prefix to the attribute causes assembly on the false rather than the true condition.

The following attributes may be tested:

DEF

- 67 -
backslash character ('\')", followed by a digit (0 through 9) or an upper case letter (A through Z). The assembler recognizes substitutable arguments in all fields of the source line. The macro argument \0 however can only be used in the operation field for referring to the data size subparameter in an opcode or pseudo instruction. The arguments \1 through \9 and \A through \Z (upper case letters) can appear anywhere in a source line. Macro calls may be nested up to ten levels deep.

- An ENDM pseudo instruction terminates a macro definition.

The following assembler directives allow to create a macro definition:

```
ENDM                  End Macro Definition
ENDM
```

An ENDM pseudo instruction terminates the macro definition.

```
LOCAL                 Local Symbols
LOCAL <sym.1>,<sym.2>,....,<sym.10>
```

List of local symbols. Symbols must be separated by commas. A blank terminates the list. The maximum number of local symbols is 10.

Description
The LOCAL pseudo instruction, which lists symbols local to the definition optionally follows the MACRO pseudo instruction.

A symbol in the list is considered local to the macro; that is, it is known only within the macro definition. On each expansion of the macro, the assembler creates a new symbol for each local symbol and substitutes it for each occurrence of the local symbol in the definition. Thus invented symbols replace LOCAL-named symbols wherever they appear in a macro definition in a manner similar to the way substitutable parameters are replaced.

The LOCAL pseudo instruction is especially useful to avoid the problem of multiply defined labels resulting from multiple calls to a macro using labels in its body.

```
MACRO                  Macro Heading
MACRO

<m_name> MACRO

<m_name>, a mandatory symbol, defines the name of the macro.
```

Description
A MACRO pseudo instruction tells the assembler to place the instructions forming the body of the macro in a table of macro definitions for assembly upon call, and to place the macro name in the symbol table.

MEXIT
End Macro Expansion
MEXIT

The MEXIT pseudo instruction terminates the macro source statement generation during expansion. It may be used within a conditional assembly structure to skip any remaining source lines up to the ENDM pseudo instruction. Together with the FAIL pseudo instruction it provides the user with a mechanism to report errors and terminate macro expansion should the macro be called with wrong parameters.

NARG
Number of Macro Arguments
NARG

The symbol NARG is a special symbol when referenced within a macro body. The assembler assigns the index of the last argument passed to the macro in the parameter list as value to NARG. The symbol NARG is undefined outside of a macro body.

Macro Calls

A macro headed by a MACRO pseudo instruction can be called by an instruction in the following format:

{<symbol>} <m_name> <p.1>,<p.2>,...,<p.i>
{<symbol>} <m_name>[.S] <p.1>,<p.2>,...,<p.i>
{<symbol>} <m_name>[.B] <p.1>,<p.2>,...,<p.i>
{<symbol>} <m_name>[.W] <p.1>,<p.2>,...,<p.i>
{<symbol>} <m_name>[.L] <p.1>,<p.2>,...,<p.i>

<symbol>
An optional location symbol.

<m_name>
Name of a previously defined macro.

<p.i>
Parameter list composed of strings of characters. Parameters are separated by commas and terminated by a blank. Two consecutive commas constitute a null parameter. An explicit zero, if desired, must be entered.

If null parameters are interspersed with legal parameters, the correct positions must be established with commas. When the list terminates before the last possible parameter, all remaining parameters a considered null.
When the first character of a parameter is a left angular bracket ("<"), the assembler considers all the characters between it and the matching right angular bracket (">") as an embedded parameter. The assembler removes the outer pair of angular brackets before substituting the enclosed character string in a line. Embedded parenthetical items must be properly paired. A parenthetical item can contain blanks and commas.

4.8 SOURCE STREAM CONTROL.

DISTINCT Define Handling of Letters in Identifiers DISTINCT

DISTINCT

The M68MIL assembler does not distinguish between lower and upper case letters in identifiers. The DISTINCT pseudo instruction allows the user to override this assembler default. To avoid confusion inside an assembly module (starting with an IDENT and ending with an END pseudo instruction) the DISTINCT pseudo instruction must be used prior to the beginning of each assembly module for which it should be effective.

DUP ENDDUP Duplicate Code Sequence DUP ENDDUP

DUP <rep_count>
ENDDUP

<rep_count>

Absolute evaluable expression specifying the integer number of times statements in the DUP range are to be assembled. If <rep_count> is zero, the statements in the DUP range are skipped.

The DUP pseudo instruction specifies repeated assembly of the statements immediately following. The range of the DUP is terminated by the ENDDUP pseudo instruction and may not contain more than 16 statements.

INSERT Insert Secondary Source INSERT

INSERT

The INSERT pseudo instruction provides a means of obtaining source statements from a file other than that being used for input. The assembler transfers the text from this file and assembles it before taking the next statement from the interrupted source of statements.

There are no parameters for the INSERT pseudo instruction. The file to be used is specified when the assembler is called. The file will be rewound before using it.
SYSTEXT Insert system text information SYSTEXT

SYSTEXT [<op.1>, <op.2>, ..., <op.n>]

<op.i>
Optional parameter, usually used to select specific parts of the system text information which are skipped otherwise.

The following option is currently available:

_MoniCaText
When _MoniCaText is selected, all definitions and macros which are needed to interface from an assembler program to the services of the MoniCa [6] monitor are assembled, else they are skipped when the system text file is read by the assembler.

The SYSTEXT pseudo instruction causes the assembler to read text from an additional input source, the system text file. It contains in its first part the definitions of identifiers and macros necessary to aid an assembly language program to adhere to the CERN convention for programming the MC68000 family [8]. These identifiers and macros are listed and explained in chapter: Programming convention support. Although this chapter gives some information about the convention, the user is advised to consult the working party report.

The system text file may contain additional information subject to conditional assembly. The identifier used to control the conditional assembly may be specified as a parameter to the SYSTEXT assembler directive. A first such collection are the identifiers and macros needed to interface to the MoniCa [6] monitor.

4.9 LISTING CONTROL

The pseudo instructions described in this section permit extensive control of the assembly listing format.

FAIL Generate an Error Message FAIL

FAIL '<text>'

<text>
<text> specifies the text to be printed as a user generated error message. Please note, that the text must be enclosed in apostrophes.
For compatibility with the Motorola cross assembler the pseudo instruction \textit{G} is accepted. Its effect is identical with the effect of \texttt{LIST DC}.

\texttt{LLEN} \hspace{1cm} \textit{Set Line Length and Format Listing}\hspace{1cm} \texttt{LLEN}

\texttt{LLEN} \hspace{1cm} \textit{<count>}

For compatibility with the Motorola cross assembler the pseudo instruction \texttt{LLEN} is accepted. Its effect is identical with the effect of \texttt{LIST FORM}.

the count to control the line length is ignored.

\texttt{LIST} \hspace{1cm} \textit{Select List Options}\hspace{1cm} \texttt{LIST}

\texttt{LIST} \hspace{1cm} \texttt{[<op.1>,<op.2>,...,<op.n>]}\hspace{1cm}

\texttt{<op.i>}

Optional parameter. A list option or a list option prefixed by a minus sign. The unprefixe d option selects the option; the prefixed option cancels it. Options are separated by commas and terminated by a blank.

The following options are available:

\texttt{CREF}

When \texttt{CREF} is selected, the assembler will print a cross reference list of all symbols used in the assembled module.

-\texttt{CREF} is the default.

\texttt{DC}

When \texttt{DC} is selected, the source line of the DC pseudo instruction and its expansion are listed, otherwise only the source line will be listed.

-\texttt{DC} is the default.

\texttt{DUP}

When \texttt{DUP} is selected, then all source lines in a DUP range will be listed; otherwise only the first iteration of the DUP range is listed.

-\texttt{DUP} is the default.

\texttt{FORM}

When \texttt{FORM} is selected, the assembler will try to align the opcode, operand and comment field in the printed listing.
-FORM is the default.

IF
When IF is selected, the source lines of the IF, IFC, IFNC, IFcc, ELSE, or ENDIF pseudo instructions and the skipped source statements in the IF range are listed, otherwise the pseudo instructions are listed, but not the skipped source statements.
-IF is the default.

MACRO
When MACRO is selected, the source line of the macro reference and the fully expanded macro body are listed, otherwise only the source line of the outermost macro reference of possibly nested macro calls is listed.
-MACRO is the default.

XOPC
When XOPC and CREF are selected, the assembler will list the use of all opcodes in the cross reference list.
-XOPC is the default.

XPSE
When XPSE and CREF are selected, the assembler will list the use of all pseudo instruction in the cross reference list.
-XPSE is the default.

XREG
When XREG and CREF are selected, the assembler will list the use of all registers in the cross reference list.
-XREG is the default.

The assembler maintains, similarly to the window for sections, a window for list options. Thus one can change the list options and return to the one in use before without explicitly knowing its options.

The LIST pseudo instruction controls the content and format of the assembler listing. Use of the LIST pseudo instruction is optional. If not specified in a module, or if specified without parameters, the assembler will produce an output according to the default for each possible option.

NOL NO LIST Cancel Listing NOL NOLIST

NOLIST

The NOLIST or NOL pseudo instruction suppresses the printing of the assembly listing until a LIST pseudo instruction is encountered.

- 75 -
NOPAGE

NOPAGE

The NOPAGE pseudo instruction suppresses paging to the output device. Page and line numbers in the cross reference table will be meaningless.

PAGE

PAGE

The PAGE pseudo instruction advances printer paper to a new page before printing. Then page headings are printed and listing continues. The PAGE pseudo instruction does not appear on the program listing.

PLEN

PLEN

<count>
An absolute value.

The assembler by default formats its output pages such, that they contain a 5 line heading and 50 lines of user code. The PLEN pseudo instruction now allows the user to change the number of lines of user code to be placed on an output page (this does not affect the heading).

SPC

SPC

<count>
An absolute value.

The SPC pseudo instruction causes the assembler to output <count> blank lines to the assembly listing. The SPC pseudo instruction does not appear on the program listing.

TTL

Assembly Listing Title

TTL

STTL

Assembly Listing Subtitle

STTL

TTL '〈text〉'

STTL '〈text〉'

〈text〉
〈text〉 specifies the text to be used as title or subtitle. Please note, that the text must be enclosed in apostrophes and cannot contain more than 60 characters.
The TTL and STTL pseudo instructions allow the programmer to print a title and a subtitle on the top of each page of the listing. To this effect the assembler maintains internally two text strings which are set to blank at the beginning of pass one. In pass two, whenever a new page is started, these two text strings together with other information are printed in the page header. Specifying a title or subtitle merely means, that the contents of the corresponding internal text string is changed to the one specified with the TTL or STTL pseudo instruction. It does not imply an automatic start of a new page. The first specified title is in addition kept in a third internal text string and is copied into the title text string at the start of pass two. Neither the TTL nor the STTL pseudo instruction are listed in the assembly listing.

4.10 OBJECT CODE CONTROL

The pseudo instructions BLONG, BSHORT, FLONG, or FSHORT allow the programmer to influence the assembler's choice whenever forward references or relative symbols are encountered, be it for direct addresses or relative branching instructions.

**BLONG**

Use Two Word Branch

**BSHORT**

Use One Word Branch

BLONG

BSHORT

The two pseudo instructions BLONG and BSHORT allow the programmer to influence the assembler whenever it is assembling a forward reference for a branch instruction. By default the assembler will use the two-word instruction form allowing a larger relative address range. After a BSHORT pseudo instruction the assembler will generate the one-word relative branch instruction, unless the suffix `.l` has been appended to that branch instruction. The occurrence of a BLONG pseudo instruction forces the two-word relative branch instruction to be generated, unless the suffix `.5` has been appended to that branch instruction.

**FLONG**

Force Direct Long Address

**FSHORT**

Force Direct Short Address

FLONG

FSHORT

The two pseudo instructions FLONG and FSHORT allow the programmer to influence the assembler whenever it is assembling a direct address the label of which contains a forward reference. By default the assembler will use the long direct address form. After an FSHORT pseudo instruction the assembler will generate the direct short address form. The occurrence of a FLONG pseudo instruction forces the direct long address form to be generated.
Note:
The selected option, long or short direct addresses, is only valid until the next occurrence of a FLONG, FSHORT or SECTION pseudo instruction.

NOOBJ  Suppress CUFO M Output

NOOBJ

The pseudo instruction NOOBJ suppresses the generation of a CUFO M module.

4.11 DATE AND TIME STAMP

For many applications it is useful and necessary to place a date and time stamp into the code generated. To allow maximum flexibility for the programmer the assembler provides a single assembler directive STAMP which causes declaration of identifiers representing year, month, day, hour, minute and second in numeric form. The programmer may then use these identifiers to produce his preferred format for the date and time stamp in his program.

STAMP  Date and Time Stamp  STAMP

STAMP

The assembler directive STAMP will generate the following identifiers as if they had been declared using the EQU pseudo instruction:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>EQU</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Year</td>
<td>&lt;year&gt;</td>
<td>83</td>
</tr>
<tr>
<td>_Month</td>
<td>&lt;month&gt;</td>
<td>1</td>
</tr>
<tr>
<td>_Day</td>
<td>&lt;day&gt;</td>
<td>1</td>
</tr>
<tr>
<td>_Hour</td>
<td>&lt;hour&gt;</td>
<td>0</td>
</tr>
<tr>
<td>_Minute</td>
<td>&lt;minute&gt;</td>
<td>0</td>
</tr>
<tr>
<td>_Second</td>
<td>&lt;second&gt;</td>
<td>0</td>
</tr>
</tbody>
</table>

representing the date and time when the assembler directive STAMP was encountered in pass one of the assembly. The information will not be changed when the assembler directive STAMP is found in pass two. STAMP may only be used once in an assembly.

4.12 ASSEMBLER MAINTENANCE

To aid in the development and maintenance of the M68MIL assembler a DEBUG pseudo instruction is provided which will force the assembler, depending on the options selected, to provide additional output in the source listing. Based on this output it proved relatively easy to find the reason for errors and to implement new features into the assembler. The normal user, not concerned with the maintenance of the assembler, has no need for this pseudo unless he wants to find out more about the internal workings of the assembler.
Just in the case the assembler "blows" up during pass one of the assembly, he might place a DEBUG pseudo requesting a pass one listing (the listing is produced during pass two) immediately following his IDENT pseudo and might, in this way, obtain an idea on which subsequent statement the assembler gave up. (This should however never happen (in theory, of course).)

```
DEBUG       Print Assembler Debug Information       DEBUG

DEBUG       <op.1>,<op.2>,....,<op.n>]

<op.i>
Assembler debug option.

The following options are available:

CHEACH
Each character scanned by the assembler from one of its sources (original code, inserted code, copied code, macro expansion, duplicated code and substituted code) will be printed.
Warning: This option produces an enormous amount of output!

FULL
Select all DEBUG options, except "CHEACH".

LISTONE
Produces a listing of the pass one operation of the assembler. This option is very useful, if the assembler "blows" up during pass one of the assembly.

OPERAND
During expression evaluation the assembler will produce detailed information about the operands used.

SYMBOLS
Whenever the assembler has scanned an identifier, it will produce information detailing the attributes of the identifier.

OFF
Switch off debug output.

TABLE
Forces the assembler to produce a listing of its symbol table at the end of pass one.

TRACE
On entry of a "strategic" procedure or function the assembler will print a line containing the name of the procedure and on exit from an even more "strategic" procedure it will print information about the findings of this procedure.

- 79 -
Figure 1: Stack layout after initialisation by main program

Many of the higher level languages use a "display" as an aid to address variables local to subprograms (procedures or functions). Each display entry is four bytes long and points to a stack frame allocated to such a subprogram. Space is allocated for 29 such entries.

During the execution it might be necessary to allocate and deallocate dynamically space which is not part of the global or local workspace. This space is made available on the heap which grows from low addresses to high addresses. The "free list pointer" is an aid for garbage collection while the heap origin pointer serves as an aid for space management.

As the name suggests, the stack limit pointer is used to detect stack/heap overflow conditions. On subprogram entry this pointer should be checked before any space is allocated on the stack as local workspace. To allow run-time support routines to use a few bytes of stack space without formality a safety
margin of 256 bytes is maintained between the stack limit pointer and the top of
the allocated heap. Figure 2 shows the same stack/heap as figure 1, but now the
program has requested and obtained space on the heap, hence the stack limit

<table>
<thead>
<tr>
<th>stack origin --&gt;</th>
<th>&quot;display&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>static level pointers</td>
</tr>
<tr>
<td></td>
<td>(higher level lang.)</td>
</tr>
<tr>
<td>A5, A6 --&gt;</td>
<td>+12(A5)</td>
</tr>
<tr>
<td>free list pointer</td>
<td>+8(A5)</td>
</tr>
<tr>
<td>heap origin pointer</td>
<td>+4(A5)</td>
</tr>
<tr>
<td>stack limit pointer</td>
<td>(A5)</td>
</tr>
<tr>
<td>entry point address</td>
<td>-4(A5)</td>
</tr>
<tr>
<td>zero</td>
<td>-8(A5)</td>
</tr>
<tr>
<td>except. handler addr.</td>
<td>-12(A5)</td>
</tr>
<tr>
<td>A7 --&gt;</td>
<td>global variables</td>
</tr>
<tr>
<td></td>
<td>last global variable</td>
</tr>
<tr>
<td>stack limit --&gt;</td>
<td>available stack space</td>
</tr>
<tr>
<td></td>
<td>256 bytes safety margin</td>
</tr>
<tr>
<td>heap top --&gt;</td>
<td>allocated heap space</td>
</tr>
<tr>
<td>heap origin --&gt;</td>
<td>&lt;-- low address</td>
</tr>
</tbody>
</table>

Figure 2: Stack layout once space has been allocated on heap

pointer has moved up, less space is available.

The first three long words of each stack frame are reserved by the convention
and must be allocated, even if the program or subprogram does not use them. The
first one contains the entry point address of the program or subprogram to which
the stack frame belongs. This information is extremely useful for monitors and
run-time systems in case of errors. It allows meaningful diagnostics to be
given. The next long word is reserved for higher level languages to maintain
the display while the last one is reserved for those languages, that provide
exception handlers to cope with run-time error conditions.

Last, but not least, we have the global variables which are accessible to all
subprograms by using <displacement>(A5) address mode. Please note that the dis-
placement is negative and that A5 is the register that permanently points to the
global stack frame while register A6 will be used to point to the stack frame of
the subprogram being active.

Definition of Global Variables

Global variables are those variables that can be addressed using the contents
of register A5 and the displacement of the variable within the global stack
frame. The macro S_Global allows the user to define his global variables as fol-
loWS:

S_Global  <name>,<size>

where

<name> is the name of the global variable
<size> is the number of bytes to be allocated

Example:
Suppose we want to use three variables, the first one being a 32-bit integer
named "Result" while the other two are two 16-bit counters named "Channels"
and "Index". The first of the two following sequences shows their definition,
while the second illustrates their use:

S_Global  Result,4 displacement is -16
S_Global  Channels,2 displacement is -18
S_Global  Index,2 displacement is -20
....

CLR.L    Result(A5) set result to zero
MOVE.W   #10,Channels(A5) set channels to 10

To be able to use the same global variables in a subprogram as in the main
program, they must be defined in the same order. To ease this, the user is
advised to keep the definitions on a separate source file and use the INSERT
assembler directive in each of the separately assembled subprograms of the
entire program.

Note:

• It is essential in a main program to define all global variables before the
  S_StartMain macro is used.

• The S_Global macro defines the displacements to be used to address the
  variables on the stack, it does however not allocate the space on the
  stack. This must be done using the S_StartMain macro.
Since memory is allocated in bytes extreme care must be taken to align variables to be addressed as words or long words on word boundaries. If in doubt write

```objective-c
S_Align   GLOBAL
S_Global   <variable>,<size>
```

- It is not possible to preset stack locations using the S_Global macro. Either they are initialised to zero using the S_StartMain macro or they must be explicitly initialised at run-time.

- Variables in the global stack frame are "alive" as long as the main program to which the frame belongs is active.

**Start Main Program and Initialise Stack**

The main program is usually defined using the assembler directives

```objective-c
IDENT   <name>
ENTRY   <name>
</name>  EQU   *

* code representing the main program

<return to monitor>
END   <name>
```

If we now want to structure this program as a main program following the programming convention we would change it to:

```objective-c
IDENT   <name>
*
define all global variables
S_Global   Var1,4
...          
S_Global   VarN,2
S_StartMain <name>,Stack_Org,Heap_Org,0

* code representing the main program

<return to monitor>
*
reserve work space
Heap_Org EQU   *
DS.L   $2000    sample space allocation
Stack_Org EQU   *
END   <name>
```

The calling sequence for the S_StartMain macro is:

```objective-c
S_StartMain <name>,<Stack_Org>,<Heap_Org>,<Init>,<Except>
```
where

<name> is the name of the main program
<Stack_Org> is the upper boundary of the work space
<Heap_Org> is the lower boundary of the work space
<Init> if zero, work space is initialised to zero
else left unchanged
<Except> is the address of an exception handler (optional)

Note:

• The macro declares the name of the main program as entry point.

• It expects that all global variables are declared using the S_Global macro
prior to the call of S_StartMain.

• It calls the subprogram _StackInit to initialise the stack/heap environ-
ment. This subprogram is part of the MoniCa monitor stack management pack-
age the source of which may be obtained from the author of this guide.

• If selected, the entire stack space is cleared to zero. This will preset
variables in the global stack frame, but no such assumptions can be made
for local stack frames.

5.2 STANDARD SUBPROGRAM CALL

The convention proposes that parameters are passed to subprograms using the
stack. It states:

Parameter descriptions are stacked from last to first
immediately before calling a subroutine. This means that the
last parameter description has a higher address than the
first parameter description.

The last-to-first approach of stacking parameters is
important because it allows the called routine to locate the
first parameter. This is essential for languages (such as C
and P+) in which the number of parameters is allowed to
vary. No count of the number of actual parameters is passed.

The value of a function is returned in D0 if it occupies
no more than 4 bytes. If the returned value needs more space
than 32 bits, an implicit first call-by-reference parameter
is introduced.

Languages like FORTRAN and Nodal always pass the address of their parameter
(call by reference) while other languages like Pascal also allow to pass the
value itself (call by value). The size of an address parameter is always four
bytes, a value parameter however may have different sizes (a byte for a charac-
ter, four bytes for an integer, six bytes for a 48-bit floating point number,
etc.). In order to reduce the "chaos" it has been agreed that a value parameter is passed as value if it occupies no more than four bytes, else its address must be passed.

There are two additional rules to be observed when calling a subprogram:

- The calling routine not only stacks the actual parameters immediately before calling a subprogram but also must remove them after the called routine has returned.

- If the calling routine wants any registers other than A5, A6 or A7 to be preserved, it must save them prior to the call and restore them after return. In other words, the called routine may freely use all registers except registers A5, A6 or A7.

The macro provided to call a subprogram is:

\[
\text{S_Call} \quad <\text{name}>,<\text{reg. list}>,<\text{par. 1}>,...,<\text{par. N}> 
\]

where

- \(<\text{name}>\) is the name of the subprogram to be called
- \(<\text{reg. list}>\) is the list of registers to be saved prior to and restored after the call
- \(<\text{par. i}>\) are the parameters to be handed over to the subprogram

Since parameters handed over to a subprogram can either be addresses (call by reference) or values (call by value) the S_Call macro needs additional information from the user to be able to generate the correct code for both cases. Parameters therefore must be specified as follows:

- call by reference: \(<\text{AP parameter}>\)
- call by value: \(<\text{VP.n parameter}>\)

where

- \(\text{VP.n}\) is: \(\text{VP.B}\) or \(\text{VP.W}\) or \(\text{VP.L}\)

In the case of byte data passed by value, a full word is allocated on the stack since the stack pointer is always aligned on word boundaries (see address register indirect with predecrement mode in chapter M 68000 programming model).

Example:
Suppose we want to call the subprogram PRESET expecting three parameters, namely:

- the address of the array to be preset
- the size of the array as a 16-bit value
- the integer value (32-bits) with which to preset the array
Figure 3: Stack layout after the call to subprogram PRESET

Let us furthermore assume we defined the array with name SCALAR using the S_Global macro, stored its size into the global variable LENGTH and want to preset it to the decimal value 25. If in addition we wanted to preserve register D3 we would write the following macro call:

```
EXTERN PRESET
S_Call PRESET,D3,<AP SCALAR(A5)>,<VP.W LENGTH(A5)>,<VP.L #25>
```
Note:

- the user must declare the name of the subprogram to be called
- never preserve D0 if you are calling a function returning its result in D0!

Figure 3 now represents the layout of our stack once the call to subprogram PRESET has been executed.

5.3 STANDARD SUBPROGRAM ENTRY

When designing a subprogram following the CERN programming convention one has to consider that

- parameters are passed to the subprogram using the stack
- the return address is passed on the stack
- a stack frame has to be generated containing the three standard entries for entry point address, display management and exception handler address as well as the space for the local variables
- register A5 should only be used for referencing purposes and registers A6 and A7 must be reset to their initial value when returning to the calling (sub)program

The four macros available to carry out this task are now described and a complete example on how to use these macros is given at the end of the section.

Formal Parameter Definition

The names and sizes of the parameters expected by a subprogram may be specified once per subprogram entry by using the macro

\[
\text{S_Param} \quad [<\text{name}>,<\text{size}>]
\]

where

- \(<\text{name}>\) is the symbolic name of the parameter
- \(<\text{size}>\) is the size of the parameter in bytes

\(<..>,<..>\) indicates this specification should be repeated for each parameter expected

It is essential that all parameters expected by a subprogram are specified with one and only one call of the \(S\_Param\) macro and that they are listed in order, beginning with the first parameter.

Parameters are accessed on the stack using the address register indirect with displacement addressing mode. The displacement is relative to the contents of register A6, which is the current stack frame pointer. The macro simply assigns
the correct displacement values to the symbolic parameter names.

**Definition of Local Variables**

Local variables are those variables that can be addressed using the contents of register A6 (the current stack frame pointer), and the displacement of the variable within the local stack frame. The macro $\text{S\_Local}$ allows the user to define his local variables as follows:

$$
\text{s\_Local} \quad <\text{name}>,<\text{size}>
$$

where

$<\text{name}>$ is the name of the local variable

$<\text{size}>$ is the number of bytes to be allocated

**Note:**

- All local variables must be defined before the $\text{S\_Enter}$ macro is used.

- The $\text{S\_Local}$ macro defines the displacements to be used to address the variables on the stack, it does however not allocate the space on the stack. This must be done using the $\text{S\_Enter}$ macro.

- Since memory is allocated in bytes extreme care must be taken to align variables to be addressed as words or long words on word boundaries. If in doubt write

$$
\text{s\_Align} \quad \text{LOCAL}
\text{s\_Local} \quad <\text{variable}>,<\text{size}>
$$

- It is not possible to preset stack locations using the $\text{S\_Local}$ macro. They must be explicitly initialised at run-time.

- Variables in the local stack frame are "alive" as long as the subprogram to which the frame belongs is active.

**Enter a Subprogram**

The entry point of a subprogram may be specified using the $\text{S\_Enter}$ macro. It has the following parameters:

$$
\text{s\_Enter} \quad <\text{name}>,<\text{except}>
$$

where

$<\text{name}>$ is the name of the subprogram

$<\text{except}>$ is the address of an exception handler (optional)

**Note:**
• The macro declares the name of the subprogram as an entry point.

• It expects that the user has declared all his local variables prior to the call of S_Enter using the S_Local macro.

• Unless instructed otherwise (see section: Switches Controlling the S_macros) it will call the subprogram _FrameInit to construct the local stack frame. This subprogram is part of the MoniCa monitor stack management package. Its source may be obtained from the author of this guide.

• Neither the S_Enter macro nor the S_RETURN macro described subsequently know anything about the purpose of the subprogram. It is therefore left to the user to take the appropriate action if the subprogram has to return a function result.

Return from Subprogram

The macro S_RETURN, which does not have any parameters, will remove the local stack frame from the stack, reset registers A6 and A7 to their initial value and return to the calling (sub)program. It does this by simply using machine instructions available on the M 68000 for this purpose.

Example of a Standard Subprogram

The subprogram shown here comes from an earlier version of the Pascal runtime library and is displayed solely to illustrate the use of the various macros described in this chapter.

Figure 4 depicts the layout of the stack once the S_Enter macro in the subprogram _PIMul has been executed. In this picture it is assumed, that _PIMul has been called from another subprogram, hence we not only have the global stack frame on the stack but also at least one local stack frame. The calling routine might have had the following sequence

* reserve local space for variables
S_Local Multicand,4
S_Local Multiplier,4
S_Local Result,4

...

* call integer multiply routine
EXTERN _PIMul
S_Call _PIMul,,<VP.L Multicand(A6)>,<VP.L Multiplier(A6)>
MOVE.L D0,Result(A6)

...

to prepare the call of the integer multiply routine and to store the result
SWAP     D1     D1 = P_upper | P_lower
BPL.S    IMul_2 did we multiply a sign bit?
SUB.L    M_plicand(A6),D0     yes, take corrective action
TST.B    M_plicand(A6)
BPL.S    IMul_3 did we multiply a sign bit?
SUB.L    D1,D0     yes, take corrective action
IMul_3   MOVE.L D0,Result(A6) store Result_upper_32
TST.B    Result+4(A6) check low order 32-bit result
BPL.S    IMul_4 is it positive?
CMP.L    #-1,D0     no, is Result_upper_32 = -1?
BNE.S    Overflow no, overflow
BRA.S    Exit yes, all ok then
IMul_4   TST.L    D0     yes, is Result_upper_32 = 0 ?
BEQ.S    Exit yes, all ok
Overflow Error E_Int0vf call MoniCa with error
Exit     MOVE.L Result+4(A6),D0 load 32-bit function result
S_RETURN and return

5.4 SWITCHES CONTROLLING THE S_MACROS

Following the programming convention the code generated for assembly language (sub)program entry is as follows:

DC.W     1 language code
Trace    IFNE _Trace switch to control name field
         DC.W ’<name of routine>’
         DC.B <length of name> always even
Trace    ELSE
         DC.B 0 no name generated
Trace    ENDF
         DC.B $FF no static level

which in the case of a subprogram is followed by the sequence

Stack    IFNE _Stack
         * call subprogram to check stack space and create stack frame
Stack    ELSE
         * do not verify stack, just create stack frame
Stack    ENDF

The default setting for both switches, _Trace and _Stack, is such, that the name field will be generated and the stack will be verified.
6 IMPLEMENTATIONS

Documentation on how to use the M68mil assembler at CERN on different computers under various operating systems has been prepared separately for each implementation thus making it easier to keep the information up to date. The documentation is accessible on the central IBM/Siemens installation under the Wylibur editor using the HELP facility. To find out which documents exist please login under Wylibur and type:

HELP PRIAM MICDOC R

This command line will bring you into the document retrieval part of the PRIAM documentation scheme. If you now type:

C <cr>

to select the "Classification selection" you will obtain a prompt to retrieve on affiliation or on type. Please answer:

A <cr>

in which case you are now almost there. Reply with:

XSOFT <cr>

to the next question and you will obtain a list of all the documents pertaining to cross software. You may now select the ones you are interested in, scan them and eventually print them on the Laser printer.
7 ACKNOWLEDGEMENTS

I should like to thank K. Eggert from the UA1 experiment who in early 1980 stimulated the development of this cross assembler. My very first (and very patient) customers, P.S. Anderssen and G. Shering, SPS, helped much in debugging the first version of the assembler. Many of today's improvements are due to their continuous demand and support.

Motorola publications [1][2] have been used as a source for this user's guide. Thanks are due to my colleagues in the Software Support Group, in particular to J.D. Blake, for suggestions, criticism and proofreading of the guide and to my son Thorsten who composed the section on M 68000 machine instructions.
8 REFERENCES

Motorola Inc

Motorola Inc

Motorola Inc

[4] CUFOM, The CERN universal format for object modules
Jean Montuelle, CERN, Data Handling Division
Part 1 and 2 of DD/US/84, Feb. 16, 1982

Jean Montuelle, CERN, Data Handling Division
DD/US/83, Feb. 16, 1982

User's Guide
Horst von Eicken, CERN, Data Handling Division
In preparation.

Horst von Eicken, CERN/DD/80/26,
write-up M68MIL, September 1980

[8] CERN convention for programming the MC68000 family.
Working party reported by B. Carpenter
CERN/PRIAM - 68K/H2, Version 3, 12 December 1983