Pocket Guide

Assembly Language for the 68000 Series

Robert Erskine

Programming Pocket Guides

Pitman Pocket Guides

PITMAN PUBLISHING LTD
128 Long Acre, London WC2E 9AN

Associated companies
Pitman Publishing Pty Ltd, Melbourne
Pitman Publishing New Zealand Ltd, Wellington
Copp Clark Pitman, Toronto

Consultant Editor: David Hatter

First edition 1984

Printed in Great Britain at The Pitman Press, Bath

ISBN 0 273 02152 4

Index

How to use this Pocket Guide 1
Address errors 34
Address registers 4, 5, 6
Addressing 12, 14, 18
Addressing modes 14, 18, 27
Addressing range 6
Appendix 62
Assembler syntax 13, 27
Bus errors 34
Carry flag 9
Condition codes register 9
Data lengths 3
Data registers 4, 5
Effective addressing 14, 35
Exception processing 30, 34
Extend flag 10
Function code registers 5, 10
Illegal instructions 33
Index registers 5
Instruction formatting 14
Instruction set 35, 38
Instructions 13
Interrupt mask 10, 31
Interrupts 31
MC68000 2
MC68000 series 1
MC68008 2
MC68010 2
Memory addressing 15, 19
Memory organization 12
Multiple exceptions 34
Overflow flag 9
Privilege violations 34
Privileged instructions 11
Program counter 5, 8
Programming model 4
Queues 29
Register direct addressing 15, 18
Reset 31
Sign extension 6, 7, 21
Sources & destinations 4
Special addressing 15, 24
Stack handling 27, 47
Stack pointer 5, 8, 11, 27, 28, 48
Status register 5, 9
Supervisor bit 10, 11
Supervisor state 11
Trace bit 10, 33
Trace mode 32
Traps 33
User state 11
Vector base register 5, 10, 31
Zero flag 9

How to use this Pocket Guide

This Guide is intended for the general programmer rather than for the specialist or systems design programmer and is centred on the MC68000 series programming instruction set, data organization and addressing capabilities rather than on technical aspects of operating functions or on circuit design.

Although readers need not have prior knowledge of the MC68000 series or of other 16-bit microprocessors, it would be helpful to have some understanding of the principles of assembly language programming. Special terms and concepts relating specifically to the 68000 series are explained in the text, but terms which are common in assembly language programming, such as interrupts and stacks, are assumed to be reasonably familiar to the reader.

The Guide covers the system architecture of the 68000 series and explains addressing modes in some detail. Important features such as exception processing and stack handling are also included, while the main part of the Guide consists of a complete glossary of programming instructions, with descriptions of the operational functions of all types of commands and their variants.

The MC68000 series microprocessors

The MC68000 series 16-bit microprocessors have been developed from the M6800 series processors and are designed to be compatible with the M6800 peripherals. The three current processors in the MC68000 range are the MC68000, the MC68008 and the MC68010.

For programming purposes, the three processors are very similar and, with one or two minor exceptions, share the same instruction set. The following paragraphs summarize the main features of these three processors, outlining their main differences. Individual differences in circuit design and function have not been included, except where they relate specifically to topics which are covered in this Guide. These variations are also covered in the main body of the text under the appropriate topic headings.

The MC68000

The MC68000 is a 16-bit microprocessor with eight 32-bit data registers, eight 32-bit address registers and an addressing range of 16 Mbytes. It has memory-mapped I/O, fifty-six instruction types and fourteen addressing modes. Programming operations may be performed on bit, BCD (4-bit), 8-bit, 16-bit and 32-bit data types.

The MC68008

The MC68008 is similar to the MC68000, with the following main exceptions:

- The data bus is of 8 bits rather than 16, with the result that most operations take twice as long to execute than they would on the MC68000.
- The addressing range is limited to 1 Mbyte compared with the 16 Mbytes of the 68000. This is due to the fact that the 68008 uses 20 bits to form an address rather than the 24 bits used by the 68000. Addresses outside the 20-bit range are automatically truncated.
- There are two interrupt control pins rather than three, with the $\overline{\text{IPL0/IPL2}}$ pin being connected internally to the $\overline{\text{IPL0/IPL2}}$ inputs. The result is that the 68008 only recognizes four interrupt priority codes (0, 2, 5 and 7) in comparison to the eight (0 to 7) which are recognized by the 68000.

The instruction set is identical to that of the 68000.

The MC68010

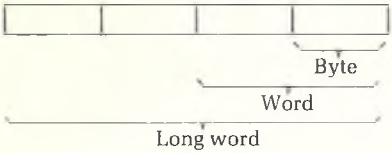
The MC68010 is again similar to the 68000, but with the following main exceptions:

- The 68010 is capable of supporting a virtual memory system — that is, it is able to access data which is not resident in physical memory. This is achieved by suspending data access while the necessary data is fetched from another source and placed in physical memory.

- Certain additional instructions are provided which relate to the special facilities of the 68010. These are MOVEC, MOVES and MOVE from CCR, the functions of which are described in the instruction glossary. In addition, the MOVE from SR instruction is privileged. In other respects the instruction set is identical to that of the 68000.
- The exception vector table may be moved to another location in memory, and additional vector tables may be created. An additional register, the vector base register, is provided for locating the required vector address.
- Some instructions, such as multiplication and division, are executed much faster than those on the other two processors.

Data lengths

The five main data types supported by the MC68000 processors are of 1, 4, 8, 16 and 32 bit lengths, designated as bit, nibble (4-bit binary-coded decimal), byte, word and long word lengths respectively:



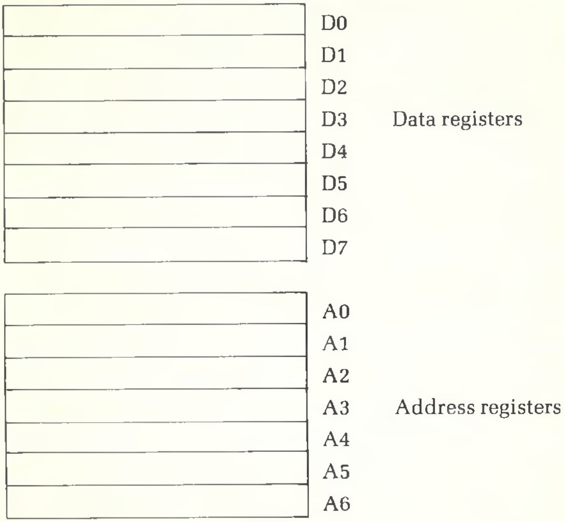
Except in cases where no operands are specified, all instructions incorporate an indication of the size of the operation. For example, MOVE operations may be of byte, word or long word size and the instruction mnemonics are expressed as MOVE.B, MOVE.W or MOVE.L as required. Where no length specification is given, the default length is W. Bit and binary-coded decimal operations are performed on data sections addressed as byte operands.

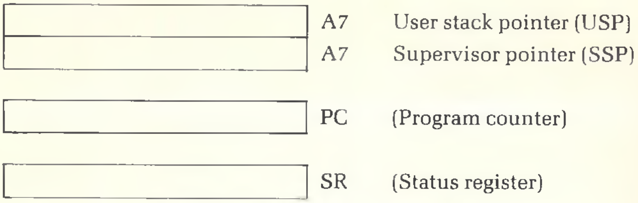
Sources and destinations

The terms source and destination are used throughout this guide to distinguish the operands specified by instructions. In the programming instruction **MOVE.L D1,D2**, for example, D1 is designated the source operand and D2 the destination operand. Effectively the instruction means that data contained in the source operand, to the left of the comma, is to be moved into the destination operand to the right of the comma. This is the opposite of the format used with some other processors such as the Z80, where the source and destination operands are expressed in reverse order.

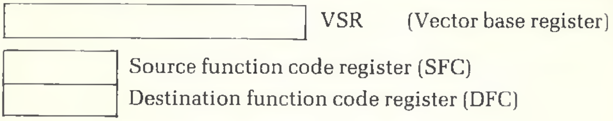
Programming model

The 68000 series processors have two categories of user registers; the data and address registers, plus a stack pointer, a program counter and a status register, as follows:



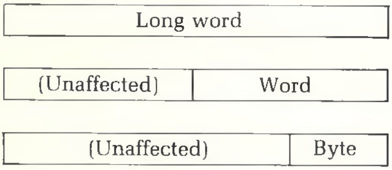


The following additional registers are provided on the MC68010:


VSR (Vector base register)
Source function code register (SFC)
Destination function code register (DFC)

Data registers

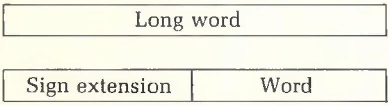
The eight data registers are labelled D0 to D7 and are each 32 bits in length. Data contained in the registers may be of byte, word or long word length, with byte values occupying bits 0 to 7, word values occupying bits 0 to 15 and long word values occupying bits 0 to 31. Note that operations specifically involving the lower-order bits in a data register will leave the higher-order bits unchanged. Thus a word value, for example, placed in a data register, will occupy bits 0 to 15 and leave bits 16 to 31 unaffected, while byte length values will occupy bits 0 to 7 leaving bits 8 to 31 unaffected.



Data registers may be used either as sources or destinations in program instructions and may also be used as data counters and index registers. (Note that the 68000 processors do not incorporate a dedicated set of index registers.)

Address registers

The seven address registers, labelled A0 to A7, are also 32 bits in length and can accommodate word or long word but not byte values. Although the address registers are basically similar to the data registers, there is an important difference. If an address register is being used as a source operand, then, as with the data registers, any bits not involved in the operation will not be affected. However, if an address register is being used as a destination, the entire 32 bits of the register are affected, irrespective of the size of the value transferred. If the value is of word length, it is first *sign extended* to a full 32 bits; that is to say, its most significant bit (bit 15) is copied into bits 16 to 31 of the register.



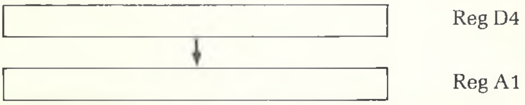
Although an address register contains 32-bit values, only the lower 24 bits are used to specify an actual memory location. The MC68000 and the MC68010 can therefore address up to 16 Mbytes of memory, which in hexadecimal numbering is in the range $000000 to $FFFFFF.

The MC68008 uses only the lower 20 bits of an address register to specify a memory location, giving an addressing range of 1 Mbyte ($00000 to $FFFFF).

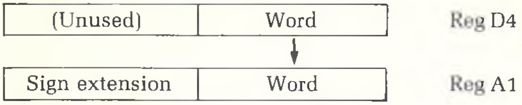
Memory is addressed as a single byte location, a word location or a long word location. A byte length operation accesses an operand in address n, a word operation accesses an operand located in addresses n and n+1 and a long word operation accesses an operand located in addresses n, n+1, n+2 and n+3.

In the case of word and long word operations, the start address of the operand must always be an even number, so that on the 68000 and 68010 the highest memory location for a word operand is $FFFFFE and for a long word operand, FFFFFC. For a byte length operand, the highest addressable memory location is $FFFFFF.

If a 32-bit operand has been loaded into an address register, then the range of memory locations which can be addressed by the register will be 16 Mbytes (or 1 Mbyte in the case of the 68008), as described above. This would be the case with an instruction such as **MOVEA.L D4,A1** which moves the entire 32-bit contents of data register D4 into address register A1, which may then be used to point to a corresponding address.



The instruction **MOVEA.W D4,A1**, however, moves only the lower order 16 bits of D4 into the lower half of A1; and the memory location, which may then be accessed by A1, will depend upon the word value moved.



Because this is a word length operation, the most significant bit of the low-order word in A1 (bit 15) will be sign extended (that is, copied) into the high-order half of A1. Thus, if the value is between $0000 and $7FFF (decimal 0 to 32767) it will be extended to a value between $00000000 and $00007FFF, because bit 15 will be a zero. Since only the first 24 bits are actually used to form an address, then in fact the addressing range of the resulting value will be $000000 to $007FFF ($00000 to $07FFF on the MC68008) — the bottom 32K of memory.

If the word value is between $8000 and $FFFF (decimal 32768 to 65535), then the 15th bit will be 1. After sign extension, therefore, the resulting values will be in the range $FFFF8000 to $FFFFFFFF which is truncated to $FF8000 to $FFFFFF (decimal 16,744,448 to 16,777,215), or, in the MC68008, to $F8000 to $FFFFF (decimal 1,015,808 to 1,048,575). In these cases, therefore, only the top 32K of memory is addressable.

It should be clear, therefore, that to address the entire range of memory locations it is necessary to load the addressing register with a long word rather than with a word operand.

Stack pointer (SP, or USP and SSP)

The stack pointer (SP) is address register A7 and is used to point to either of the two stacks used in the system.

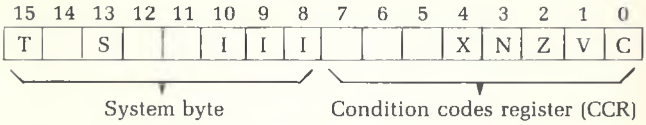
The supervisor stack is normally used by the operating system while the user stack is the one normally used by a user program. In supervisor state the stack pointer is termed the SSP, while in user state it is termed the USP. Only one of these stacks may be used at any one time, and bit 13 of the status register indicates which stack is currently in use. Data stored on a stack should be of word or long word length. Where bytes are to be stacked, they should be incorporated as the high-order half of a word length value to avoid misalignment. See also 'User and supervisor states'.

Program counter (PC)

The program counter holds the address of the next program instruction to be executed.

Status register (SR)

The status register is a two-byte register containing sixteen 'bit' flags which are used to indicate various conditions during the running of a program. Bits 0 to 7 constitute the *condition codes register* (CCR) and contain flags denoting the results of program operations. Bits 8 to 15 constitute the *system byte* and contain flags indicating the current status of the system. Bits 5 to 7 and bits 11, 12 and 14 are not used.



The bit flags may be tested after certain operations have been performed and the results used for conditional call and branch decisions. Their operation is as follows:

Carry flag (C): bit 0 of the status register
The carry flag is 'set' to 1 if an addition operation results in a 'carry' or if a subtraction operation results in a 'borrow'. If no carry is caused by an operation, then the carry flag is 'reset' to zero.

Overflow flag (V): bit 1 of the status register
Any result exceeding an operand's size limits will result in an 'overflow' and will cause the overflow flag to be set.

Zero flag (Z): bit 2 of the status register
The zero flag is set when the result of an operation is zero. This may happen, for example, after a subtraction or decrementation instruction has been used or when a comparison has been made between two numbers of equal value. For results which are other than zero, the zero flag will be reset.

Negative flag (N): bit 3 of the status register
The negative flag is used to indicate whether a number is positive or negative in terms of two's complement arithmetic. The most significant bit of any value in two's complement is used to indicate a positive (reset) or negative (set) value and it is this bit which is copied into bit 3 of the status register.

Extend flag (X): bit 4 of the status register
This operates in a similar way to the carry flag but is used in operations involving larger numbers (for example, in multiprecision arithmetic) or in BCD operations.

Interrupt mask: bits 8 to 10 of the status register
The interrupt mask consists of three bits which indicate which of the seven interrupt levels is currently enabled. Note that the three mask bits are the least significant bits of the 'system' byte of the status register.

Supervisor bit (S): bit 13 of the status register
The supervisor bit is set to indicate supervisor state or reset to indicate user state. For a full explanation of the two states, please refer to 'User and supervisor states'.

Trace bit (T): bit 15 of the status register
This bit is set to indicate that trace mode is in operation, otherwise it is reset. For a full explanation of the trace function, please refer to 'Exception processing'.

Vector base register (VSR)

The vector base register is provided only on the MC68010 and is used in conjunction with an offset value to generate an exception vector address. See 'Exception processing' for a more detailed explanation.

Function code registers (SFC and DFC)

These are 3-bit registers, the source function code register and the destination function code register, which are used with the MOVEC and MOVES instructions on the MC68010. MOVEC sets the registers to permit MOVES to read or write to locations in the supervisor program, user program or user data areas.

User and supervisor states

The 68000 operates in either a 'user' or 'supervisor' state, depending on the types of operations which are currently being executed. In supervisor state, bit 13 of the status register is set. The supervisor state is normally set for operations involving an operating system, including exception processing, and is distinguished from the user state in that certain instructions are 'privileged' and may only be executed when the supervisor state is in force. This system affords the operating system full protection from its user programs and if necessary allows the operating system to obtain exclusive access to certain memory resources and peripherals.

The privileged instructions are ANDI to SR, EORI to SR, MOVE to SR, MOVE USP, ORI to SR, RESET, RTE and STOP. Privileged instructions are illegal in the user state because their use can affect flags and pointers, thus interfering seriously with the execution of a user program. Any attempt to use a privileged instruction from within a user program will result in a TRAP exception. All non-privileged instructions are executable from within both user and supervisor states.

Both states use different stack pointers: the user stack pointer (USP) for the user state and the supervisor stack pointer (SSP) for the supervisor state. Both pointers can only be referenced from their own state although both are addressed as address register A7 (an exception to this rule is the instruction MOVE USP, which permits an operating system functioning in the supervisor state to set the stack pointer for a user program).

Memory organization

Address organization in memory

Memory locations may be accessed in byte, word or long word lengths as specified by the particular instructions used. All word and long word length references must address even numbered memory locations whereas byte references may address both odd and even locations.

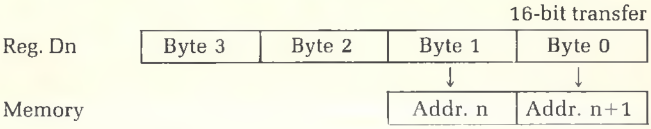
Addr.		Bytes	Words	Long words
0		0	0	0
1		1		
2		2	1	
3		3		
4		4	2	1
5		5		
6		6	3	
7		7		
8		8	4	2
9		9		

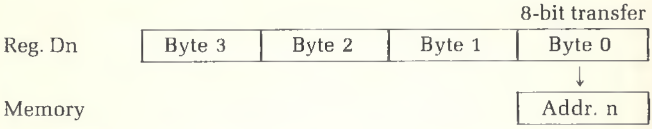
For long word operations, the first two bytes of the address constitute the high-order word and the second two bytes the low-order word.

Where data is loaded into memory, for example from a data register, the data bytes are transferred as follows:

32-bit transfer

Reg. Dn	Byte 3	Byte 2	Byte 1	Byte 0
	↓	↓	↓	↓
Memory	Addr. n	Addr. n+1	Addr. n+2	Addr. n+3





Data transferred from memory to a register is moved in the reverse order. For example, the first word value at a particular address will be transferred in two bytes to the low-order half of a register.

MC68000 instructions

Assembler syntax

Newcomers to assembly language programming should note that it is usual to use an assembler program for constructing programs, rather than calculating the machine code for each instruction individually.

An assembler accepts assembly language mnemonics and compiles them into machine code automatically. For example, the instruction mnemonic **ADD D4,D6** means 'add the contents of data register D4 to data register D6 and let D6 hold the result'. If this instruction were to be entered into an assembler program, it would be converted into its numeric form automatically, ready for execution.

The exact form of each mnemonic may vary among assemblers but will always follow a similar type of format, which is referred to as the *assembler syntax*. In this Guide, all instructions are expressed in this type of format although in many cases a generic version of the syntax is used. For example, the above instruction was derived from the general syntactic form: **ADD <ea>,Dn** which means add a source operand of a type permitted by the definition <ea> (meaning 'effective address'); in this case, the contents of register D4, to a destination operand specified by Dn (meaning any of the eight data registers); in this case, D6.

Before going on to look at the ways in which these mnemonic instructions are used it is worth looking in some detail at how instructions are assembled and how the assembler mnemonics relate to them.

Instruction formatting

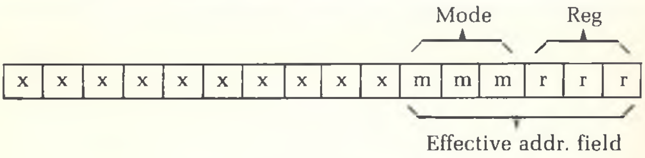
MC68000 instructions are between one and five words in length, the first word always specifying the length of the instruction and the operation which is to be performed. The remaining words, if any, consist of immediate operands or of 'extension words', which further qualify the addressing mode used by the instruction.

Types of addressing modes

There are three distinct forms in which operands are specified by an instruction:

Register specification — in which a specific register is referred to and identified by a unique 3-bit code within the instruction word.
Implicit reference — in which the instruction implicitly refers to specific registers such as the stack pointer (SP) or the status register (SR).
Effective addressing — in which the operand is specified by the contents of an 'effective address field' within the instruction.

The effective address field occupies the 6 low-order bits of an operation word and consists of 3 'mode' bits (3 to 5), which specify the type of addressing mode to be used, and 3 'register' bits (0 to 2), which specify the identity of the register to be used.



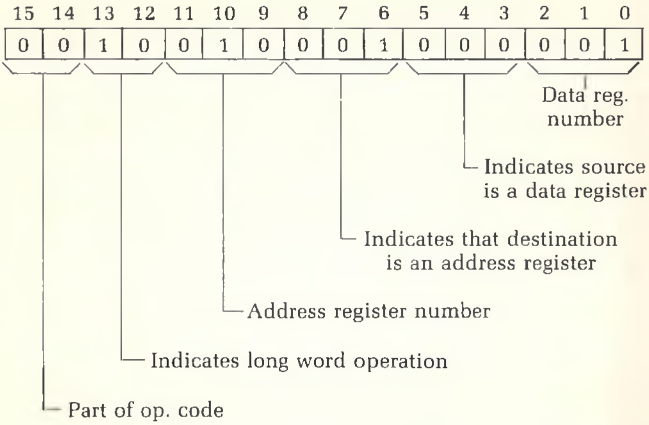
In fact, most instructions specify operands by means of an effective address code, and these can be divided into three categories as follows:

Register direct — in which one of the data or address registers is specified in the register section of the effective address field.

Memory addressing — in which the mode section of the effective address field indicates that the operand is to be found in memory. The instruction also incorporates an indication of the operand's actual location.
Special addressing — in which a code indicating the special mode required is used in place of a register specification in the effective address field.

The following are examples of the constitution of a selection of instruction formats, showing how the fields within the instruction word are used to specify the required operands.

Example 1:

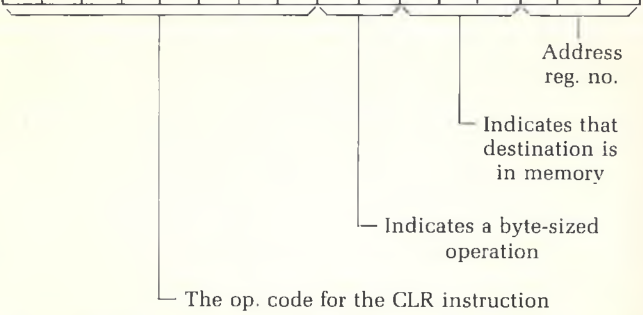


This corresponds to the instruction **MOVEA.L D1,A2** which moves the entire contents of data register D1 into address register A2. Bits 3 to 5 of the instruction word contain the binary code 000 which specifies that the source operand is a data register. This addressing mode is of the *register direct* type. Bits 0 to 2 indicate the actual number of the data register, which is 1. Bits 6 to 8 contain the binary code 001 which specifies that the destination operand is an address register and bits 9 to 11 give the actual number of the register, which is 2. Bits 12 and 13 contain the binary operation code 10, indicating that the instruction is of long word length. Bits 14 and 15 contain zeros and these also form part of the operation code.

The above instruction may be described in terms of the generic assembler syntax as **MOVEA <ea>,An** (move data from a source operand specified by an effective address to a destination operand specified as an address register).

Example 2:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0	0	0	1	0	1	0	0


Address reg. no.
Indicates that destination is in memory
Indicates a byte-sized operation
The op. code for the CLR instruction

This corresponds to the instruction **CLR.B (A4)** which clears (sets to zero) the memory byte whose address is specified by address register A4. The code contained in bits 3 to 5 of the instruction word indicates that the destination operand is in a memory address pointed to by one of the address registers and bits 0 to 2 contain the actual number of the register, 4. This effective address is therefore of the *memory addressing* type and the instruction belongs to an addressing mode referred to as Address Register Indirect because the operand is indirectly referred to via the register containing its address.

Bits 6 and 7 contain the binary code 00, indicating that this instruction references a byte length operand. Bits 8 to 15 specify that this is a CLR instruction.

Example 3:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	0	1	0	0	0	1	1	1	0	0	1	= operation word
a	a	a	a	a	a	a	a	b	b	b	b	b	b	b	b	= 1st ext. word
x	x	x	x	x	x	x	x	y	y	y	y	y	y	y	y	= 2nd ext. word

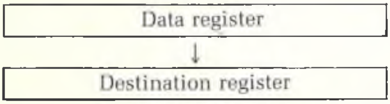
This corresponds to the instruction **CLR.B $nnnn**, a *special addressing* instruction in which the content of a memory address specified by the absolute numeric value nnnn is cleared. This instruction is three words long, with the first word being similar to the instruction in Example 2. Bits 8 to 15 are identical to those in Example 2 and bits 6 to 7 contain zeros, indicating that this is a byte length operation. Bits 0 to 5, however, which form the effective address, contain the binary code 111001 which indicates that this is an Absolute Long addressing mode instruction, meaning that the entire 32 bits of the absolute value nnnn are to be used to address the required location. The first extension word forms the high part of the address and the second extension word forms the low part of the address.

Addressing modes

Note that all examples given are based on the 68000 and therefore addresses are derived from the first 24 bits of an address value. On the 68008, only the first 20 bits would be taken into account. There are otherwise no differences between the processors with respect to the following addressing modes.

Register direct modes

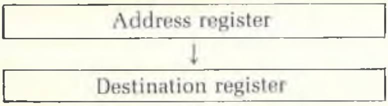
Data register direct
The operand is located in the data register specified in the effective address register field.



For example, the instruction **MOVE.L D5,D4** moves the 32-bit value stored in data register D5 to data register D4:

Reg. D5
Reg. D4

Address register direct
The operand is located in the address register specified in the effective address register field.

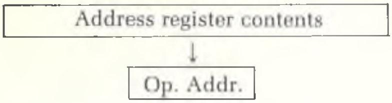


For example, the instruction **MOVE.L A4,D5** moves the 32-bit value stored in data register A4 to address register D5:

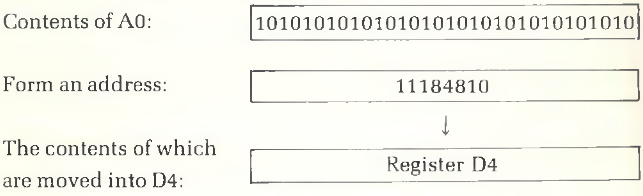
Reg. A4
Reg. D5

Memory addressing modes

Address register indirect
The operand is located in a memory address specified by an address register.



For example, **MOVE.B (A0),D4** moves the byte length value located in the memory address pointed to by A0 to the least significant byte of data register D4:



MOVE.B D1,(A2) moves the value contained in the least significant byte of data register D1 to the memory address pointed to by address register A2.

Address register indirect with postincrement
The operand is located in a memory address specified by an address register. The register is afterwards incremented by 1 following operations on byte-sized operands, by 2 after operations on word-sized operands and by 4 after operations on long word-sized operands. If the stack pointer is being used to hold the address of the operand, the register is incremented by 2 after operations on byte-sized operands.

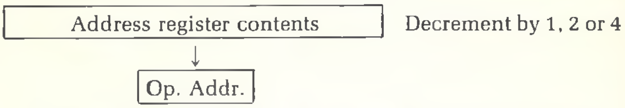
$$\begin{tikzcd}\boxed{\text{Address register contents}} \arrow[d] \\ \boxed{\text{Op. Addr.}}\end{tikzcd}$$

Then increment address register by 1, 2 or 4.

For example, **MOVE.L (A1)+,D2** moves the contents of the memory address pointed to by A1, together with the contents of the three following memory addresses, to data register D2. It then increments the value of address register A1 by four (because this is a long word length operation):

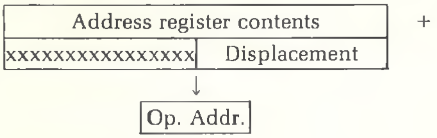
Contents of A1:	00000000000000011001011000111010			
Specify an address:	103994			
Whose contents plus those of the three following addresses:	103994	103995	103996	103997
	↓	↓	↓	↓
Are moved into D2:	Byte 3	Byte 2	Byte 1	Byte 0
Then A1 is incremented by four:	00000000000000011001011000111110			

Address register indirect with predecrement
The operand is located in a memory address specified by a register. Before execution, however, the register is decremented by 1 prior to operations on byte-sized operands, by 2 prior to operations on word-sized operands and by 4 prior to operations on long word-sized operands. If the stack pointer is being used to hold the address of the operand, the register is decremented by 2 before operations on byte-sized operands.



For example, **MOVE.B D1,–(A1)** automatically decrements register A1 by one (since this is a byte-length operation) and then moves the contents of the least significant byte of data register D1 into the memory address pointed to by address register A1.

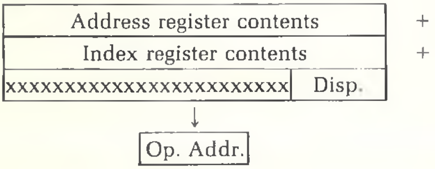
Address register indirect with displacement
The operand is located in a memory address specified as the sum of an address in an address register plus a sign extended 16-bit displacement integer.



For example, **MOVE.B D1,—1(A1)** moves the contents of the least significant byte of data register D1 into the memory address pointed to by address register A1 plus the sign extended displacement value, which in this case is −1. The original value of A1 is not altered by the operation:

Register A1:	10101010101010101010101010101010
And a two's complement displacement:	1111111111111111
(which is sign extended)	11111111111111111111111111111111
Are added together giving:	10101010101010101010101010101001
Specifying an address:	11184809
Whose contents are passed to register D2:	Register D2

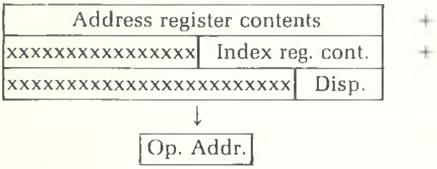
Address register indirect with displacement and index
The operand is located in a memory address specified as the sum of an address in an address register plus the least significant byte of an extension word (sign extended to 16 bits) plus the contents of an index register.



For example, **MOVE.B $18(A2,D3.L),D2** takes the contents of address register A2, the contents of address register D3 (the index register) and the absolute value $18 (the displacement) and adds them together. The contents of the address represented by their sum are then copied into data register D2. Note that the displacement is first sign extended to 32 bits. The 'L' after D3 indicates that the index register is of long word size:

Address register A2:	10101010101010101010101010101010
And the index register (D3):	00000000000000001101101011101010
And an 8-bit signed displacement value:	00011000
(which is sign extended)	00000000000000000000000000011000
Are added together giving:	10101010101010111000010110101100
Which is address:	11240876
Whose contents are moved to D2:	Register D2

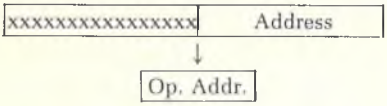
or:



MOVE.B $18(A2,D3),D2 operates in the same way as the previous example except that the index register D3 is of size 'W' by default and is therefore sign extended to 32 bits.

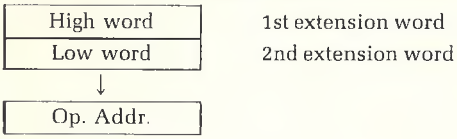
Special address modes

Absolute short address
The operand is located in a memory address specified by a 16-bit extension word, sign extended to 32 bits.



For example, **ADDA.W $0700,A4** adds the values stored at memory address $0700 and $0701 to the current value of address register A4.

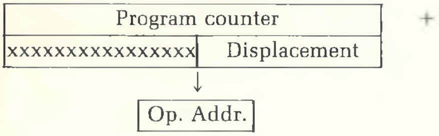
Absolute long address
The operand is located in a memory address specified by two extension words. The first word represents the high-order part of the address and the second represents the low-order part.



For example, **ADDA.L $FF2A56,A4** adds the long word value starting at memory address 16,722,518 to the current value of address register A4.

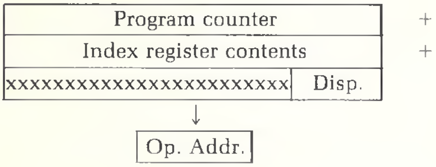
The difference between the short and long absolute forms is that the long form is used to address the entire memory range while the short form addresses only the top 32K and bottom 32K of memory. (If the reason for this is not clear, please refer back to the 'Address registers' section.)

Program counter relative with displacement
The address is specified as the sum of the address in the program counter register and a sign extended 16-bit displacement integer. The address of the displacement extension word is the value contained in the program counter.

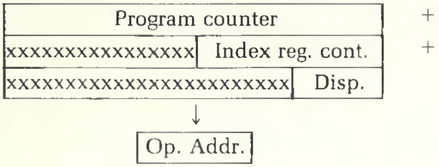


For example, **JMP *+18** adds the hex value 18 to the current value of the program counter and causes program execution to 'jump' to the corresponding address.

Program counter relative with displacement and index
The address of the operand is specified as the sum of the address contained in the program counter plus a sign extended 8-bit displacement integer plus the contents of the index register. The displacement integer is located in the address contained in the program counter.



For example, **MOVE.B $18(PC,A2.L),D4** adds together the program counter, an index register (A2) and a displacement value ($18) and moves the contents of the resulting address into D4. The 'L' after the index register indicates that its long word value is to be used; or:



MOVE.B $18(PC,A2),D4 is the same as the previous example except that by default the index register value is of word length and is therefore sign extended before addition.

Immediate data
The operand is specified as the low-order byte of an extension word in the case of byte operations, a complete extension word in the case of word operations and two extension words in the case of long word operations. In the latter case, the first extension word represents the high-order part and the second the low-order part.

For example, **MOVE.B #$24,D4** moves the hex value 24 into the low-order byte of register D4. **MOVE.L #$F23A820B,D4** moves the specified value into the entire 32 bits of D4.

Implicit reference
An implicit instruction is one in which the instruction itself implicitly points to the operand. For example, RTS meaning return from subroutine. In this case, no operand needs to be given.

Note that the instructions which incorporate the letter Q (for 'quick') may be used for transferring signed 8-bit values. For example, **MOVEQ #−5,D4**. This is a long word operation and sets the whole of D4 to the value indicated. ADDQ and SUBQ can be used in a similar fashion for adding or subtracting values to or from a register. In these cases, the value is confined to a range of 1 to 8. For example, **ADDQ.W #7,D4**.

Summary

The following table lists the addressing modes, together with the assembler syntax and the contents of the effective address field for instructions in each mode:

Addressing mode	Assembler syntax	Mode field	Register field
Data register direct	Dn	000	Reg. no.
Addr. reg. direct	An	001	Reg. no.
Addr. reg. indirect	(An)	010	Reg. no.
Addr. reg. ind. with postincrement	(An)+	011	Reg. no.
Addr. reg. ind. with predecrement	−(An)	100	Reg. no.
Addr. reg. ind. with disp.	d(An)	101	Reg. no.
Addr. reg. ind. with disp. + index	d(An,Ri)	110	Reg. no.
Absolute short	$xxxx	111	000
Absolute long	$xxxxxx	111	001
PC relative with displacement	d(PC)	111	010
PC relative with index	d(PC,Ri)	111	011
Immediate	#$xxx	111	100

Stack handling

All eight address registers may be used as stack pointers in the MC68000, although register A7, commonly distinguished as the SP, is the one most regularly used for stack operations.

Registers A0 to A7 may each be used to form stack pointers by selecting an appropriate area of memory to hold the stack and loading its base address into one of the registers. It is possible by this means to reference eight separate stacks, although it is advisable for practical reasons to leave at least one address register free for normal programming purposes.

If address number $8000 is selected as the base address of a user stack, then register A0 can be used as its stack pointer by loading it directly with the base value:

```
MOVEA.L #$8000,A0
```

Stacks normally grow downwards in memory, therefore it is necessary when placing an item of data on a stack to decrement its stack pointer by 2 or 4 bytes, depending on whether a word or long word value is being placed on it. This is most easily achieved using the address register indirect with predecrement mode (reverse stacks use postincrement mode). For example:

MOVE.L D4,−(A0)

which decrements stack pointer A0 by 4 bytes to point to address $7FFC ($8000 minus 4) and then moves the 32 bits contained in data register D4 into memory addresses 7FFC, 7FFD, 7FFE and 7FFF.

The instruction **MOVE.L D5,−(A0)** would then add a second 32-bit value to the stack, at addresses 7FF8, 7FF9, 7FFA and 7FFB, with the stack pointer now pointing to address 7FF8.

To remove values from the stack, the address register indirect with postincrement mode is used:

MOVE.L (A0)+,D5 (transfer the top of stack value into D5 and increment A0 by 4)
MOVE.L (A0)+,D4 (transfer the top of stack value into D4 and increment A0 by 4)

Note that because the original value of D5 was the last to be placed on the stack, it is the first one to be removed.

Register A7 differs from the other address registers with respect to stack operations in that because of its special status as a dual stack pointer (the USP in the user state and the SSP in the supervisor state) it is implicitly assumed by some of the MC68000 instructions to be the one and only stack pointer. For this reason it is advisable to regard the stack pointed to by A7 as the primary general-purpose stack and to use other stacks as data stores for specific user applications.

It is often useful to be able to store the values of several registers on the stack for later retrieval, for example where a call is made to a subroutine and the current register values need to be preserved for use after the subroutine has been completed.

To this end, the MOVEM command is provided, which specifies a list of registers and a pointer to the stack area in which they are to be stored. (MOVEM may also be used to transfer data to nominated memory addresses.)

For example, **MOVEM.L D0–D3/D5/A0–A6,–(A4)** stores the specified data and address register values in stack addresses pointed to by address register A4, which is predecremented by 4 bytes before each transfer of data. This instruction may also be implemented as a word (but not a byte) sized operation, although data integrity can only be guaranteed upon retrieval if the operation is of long word size.

The above values may be retrieved by the instruction **MOVEM.L (A4)+,D0–D3/D5/A0–A6**.

Queues

A queue is similar in operation to a stack, except that values are removed from a queue in the same order as that in which they were originally stored. A queue may run from high to low memory or vice versa.

Two address register pointers are necessary for each queue: one to point to the address at which the next value can be stored and one to point to the current head of the queue.

If the queue runs from low to high memory, then data is stored by means of the 'address register indirect with postincrement' addressing mode, using a data 'put' pointer. Data is removed by means of the same addressing mode, using a data 'get' pointer.

Addr. x	A0 used as 'get' pointer
Addr. x+1	
Addr. x+2	
Addr. x+3	A1 used as 'put' pointer
Addr. x+4	

In the above example, if the byte value is placed in address x+3, register A1 is then autoincremented by 1 to point to address x+4. Likewise, if a byte value is removed from address x, register A0 is then autoincremented to point to address x+1.

In the case of queues running from high to low memory, the 'address register indirect with predecrement' mode is used for both 'put' and 'get' operations. Note that although byte values are stacked in the above example, it is advisable to stack data in word or long word lengths to avoid address misalignment.

A problem that is likely to arise with a queue is that as data is added and removed, the section of memory occupied by the queue will gradually creep through the memory space, thus overwriting other stored data. To overcome this problem, it is advisable to create a 'circular' queue, which is essentially a reserved buffer space beyond which the queue data is not permitted to move. The reference register is tested to establish whether it points to an address beyond the permitted boundary and, if it does, it can be adjusted by subtracting the length of the buffer.

Exception processing

An exception is an occurrence in which program execution is automatically diverted, or 'vectored', to an address outside the programmed execution sequence. This situation may arise when an error has occurred or when an external event such as a keyboard entry or an input from some other device has signalled an interruption requiring an immediate response.

When an exception occurs, it is necessary for the processor to store the current environment (that is, the parameters of its current status) so that execution can be resumed after the exception has been dealt with.

An exception process involves four distinct phases:

(1) A copy is taken of the current state of the status register.
(2) The status register is altered in readiness for exception processing.
(3) The current processor environment is saved (the program counter and status register are pushed on to the supervisor stack).
(4) A jump is made to the appropriate exception vector address.

The address to which execution is diverted depends on the origin of the exception and is obtained from an internally generated 'vector number' or from one supplied by an external device. The vector number is multiplied by four, giving the address of an exception vector which is stored in a table in the supervisor data space. The vector table supplies the address of a routine which is designed to handle the particular cause of the exception. The new execution address is then stored in the program counter register and the processor commences execution from that point.

The address to which program execution is vectored during an exception depends upon the origin of the exception and in each case it is one of a number of routines designed to handle the specified type of interruption. If the processor is in Trace mode, for example, an exception takes place after the execution of every single instruction and execution is temporarily diverted to a monitoring routine.

External exceptions may be caused by interrupts from external devices, bus errors or external resets, and internal exceptions may be caused by illegal instructions, address errors, privilege violations and TRAP, TRAPV, CHK and DIV instructions.

The MC68010 uses a movable vector base register (VSR) and the vector number is used as an offset to reference items in the vector table. The operating system may move the vector table elsewhere in memory, even to RAM, where it may be used to call exception handling routines incorporated in applications software. Multiple vector tables may also be created for multitasking purposes and referenced via the VSR.

Reset

A reset is a process in which the entire system is re-initialized and normally takes place either when the system is first powered up or when a complete system failure has taken place from which recovery is impossible.

Interrupts

An interrupt is a signal from an external device which causes an exception, the exception vector address being calculated from the value input from the interrupting device.

Since an interrupt may be competing with a process which is currently taking place, an interrupt priority system is used in which each interrupt is given a priority code between 0 and 7: level 7 representing the highest level of priority. Priority code 0 signifies no interrupt. On the MC68008, only priority levels 0, 2, 5 and 7 are used. When an interrupt takes place, the following sequence of events is initialized:

(1) The priority level of the interrupt is placed on the interrupt request lines.
(2) The interrupt is placed in a pending state while the current instruction is processed.
(3) The interrupt priority code is compared with the current processor priority code (indicated by bits 8 to 10 of the status register).
(4) If the current priority code is higher than or equal to the interrupt priority code, the pending state continues while the next instruction is processed.
(5) If the interrupt priority code is higher than the current priority code, an exception is forced.
(6) The interrupt priority code is placed in bits 8 to 10 of the status register.
(7) The interrupt is acknowledged and an exception vector byte is obtained from the interrupting device.
(8) The program counter and status register are pushed on to the supervisor stack.
(9) The exception vector address is calculated.
(10) Execution then continues from the address pointed to by the program counter.

Trace mode

The purpose of the trace mode is to allow a program to be executed one instruction at a time, each individual instruction execution being followed by an exception process in which a debugging monitor routine is called.

Trace mode is selected by setting bit 15 (the T bit) of the status register and is terminated by resetting it. During a trace exception, the processor initializes the following set of events:

(1) The current values of the status register are copied.
(2) The system is set to supervisor privilege status (the S bit of the status register is set).
(3) The T bit is reset, to prohibit further trace exceptions.
(4) The trace exception vector address is obtained.
(5) The program counter and status register are pushed on to the stack.
(6) Execution commences from the address of the trace routine.

After a trace exception has been completed, the former conditions are reinstated and execution re-commences with the next instruction in the user program.

Traps

A trap is a type of exception caused by the use of the TRAP instruction, by unusual conditions encountered during execution or by instructions which incorporate a trap mechanism, such as TRAPV.

Trap-generated exceptions follow the standard procedure for internally-generated exceptions and execution is vectored to an address calculated from data contained in the exception vector table between addresses 080 and 0BF.

Illegal instructions

An illegal instruction is one in which the bit pattern of the first word does not correspond to that of any legal instruction.

Instructions in which bit patterns 12 to 15 inclusive are equal to the binary values 1010 or 1111 are defined as unimplemented instructions and are used to emulate unimplemented instructions in software. Those incorporating the value 1010 are vectored to an address calculated from data contained in the vector table at address 028 and those incorporating the value 1111 are vectored to an address calculated from data contained in the vector table at address 02C.

Privilege violations

Privilege violations are attempts to use one of the following privileged supervisor instructions from within the user state: STOP, RESET, RTE, MOVE to SR, MOVE USP, ANDI to SR, EORI to SR, and ORI to SR. Execution during privilege violation exceptions is vectored to an address calculated from data contained in the vector table at address 020.

Bus errors

Bus errors are caused by attempts to access incorrect destinations external to the processor such as non-existent memory addresses. A bus error exception is vectored to an address calculated from data contained in the vector table at address 008 and the current environment is saved on the stack in the usual manner before exception processing begins. However, since a bus error occurs in mid-instruction rather than between instructions, it is necessary to save additional information concerning the details of the instruction currently being processed.

Address errors

Address error exceptions are caused by attempts to access word or long word operands, or instructions, at odd rather than even addresses. The exception is vectored to an address calculated from data contained in the vector table at address 00C and the same data is stacked as that for a bus error.

Multiple exceptions

Where several exceptions occur at the same time there is a system of priority grouping which determines their order of precedence. The highest priority group is 0 which includes reset, bus error and address error. Group 1 is next in priority order and includes trace interrupt, illegal instruction and privilege violation. Group 2, which has the lowest priority, includes TRAP, TRAPV, CHK and division by zero. Within these groupings the exceptions are arranged in a secondary order of priority as indicated above.

Group 0 exceptions are initiated within two clock cycles of their occurrence. Group 1 exceptions are initiated before the next instruction is executed while group 2 exceptions are initiated in accordance with the operation of the instructions which cause them.

The MC68000 instruction set

In the following section, each MC68000 instruction is listed with details of its assembler syntax, operand lengths and a summary of the control code flags affected by the operation. A short description of the function of each instruction is given.

Effective addressing modes are divisible into four categories which distinguish which types of operands may be used by a particular instruction. These are as follows: *Data operands* are those which reference data rather than the contents of address registers. *Memory operands* are those which are not contained in registers. *Alterable operands* are those which can be written to and therefore exclude, for example, immediate data operands. *Control operands* are memory operands of an unspecified size.

Clearly these categories overlap and it is therefore possible to define operands as being 'memory alterable' or 'control alterable'. The following table indicates the categories applicable to each of the addressing modes:

Addressing mode	Data	Mem.	Cont.	Alter.	Syntax
Data reg. dir.	X			X	Dn
Addr. reg. dir.				X	An
Addr. reg. ind.	X	X	X	X	(An)
Addr. reg. ind. + postinc.	X	X		X	(An)+
Addr. reg. ind. + predec.	X	X		x	−(An)
Addr. reg. ind. + disp.	X	X	X	X	d(An)
Addr. reg. ind. + index	X	X	X	X	d(An,Ri)
Absolute short	X	X	X	X	xxx
Absolute long	X	X	X	X	xxxxxx
PC with disp.	X	X	X		d(PC)
PC with index	X	X	X		d(PC,Ri)
Immediate	X	X			#xxx

The combination of data lengths, addressing modes and instruction types in the 68000 processors can be used to generate a very large number of individual instructions. The addressing modes are so consistently applied that it is possible to define the structure of any instructions by means of the generic assembler syntax and these mnemonics are included under each instruction type heading.

Key to notation

(1) *Condition flags*

–	unaffected
0	zero (reset)
1	one (set)
a	altered except when destination is an address register
A	altered according to value
?	may be altered according to value
X	not defined

(2) *Addressing syntax*

An, Ax and Ay	any address registers
Dn, Dx and Dy	any data registers
Rn, Rx and Ry	any registers
(An)	address register indirect
d(An)	address register indirect with displacement
–(An)	address register indirect with predecrement
(An)+	address register indirect with postincrement
<data>	immediate data

(3) *Addressing mode categories*

<ea>	any effective address mode
<a/ea>	alterable mode
<c/ea>	control mode
<d/ea>	data mode
<ca/ea>	control alterable mode
<da/ea>	data alterable mode
<ma/ea>	memory alterable mode

Note that most instructions incorporate a suffix (B, W or L) to indicate the size of the operation. Where no suffix is given, the default size is W, except in cases where no operand is given; for example, in RTS. Where appropriate, the legitimate operand sizes for each instruction are given in the glossary.

Add decimal

Syntax ABCD Dy,Dx
ABCD −(Ay),−(Ax)

Mnemonic	Operation	Data size	Flags: N Z V C X
ABCD	Decimal addition	B.	X ? X A A

Description Adds two binary-coded decimal bytes together plus the value of the extend bit of the status register. The operation may be performed between the contents of two data registers or between the contents of two memory addresses using predecrement addressing mode.

Add binary

Syntax ADD <ea>,Dn
ADD Dn,<ma/ea>
ADDA <ea>,An
ADDI #<data>,<da/ea>
ADDQ #<data>,<a/ea>
ADDX Dy.Dx
ADDX −(Ay),−(Ax)

Mnemonic	Operation	Data size	Flags: N Z V C X
ADD	Binary addition	B.W.L.	A A A A A
ADDA	Add address	W.L.	− − − − −
ADDI	Add immediate	B.W.L.	A A A A A
ADDQ	Add quick	B.W.L.	a a a a a
ADDX	Add extended	B.W.L.	A ? A A A

Description The ADD instruction is used to add a source operand to a destination operand, with either the source or destination being one of the data registers. Where the source is an address register, operations must be of word or long size. Where the source is a data register, operands may be of byte, word or long size and the destination specification is restricted to a memory alterable address mode. Where the destination is a data register, any of the address modes may be used.

The ADDA instruction is similar to ADD, except that the destination must be an address register.

ADDI adds an immediate value to a destination specified by a data alterable addressing mode.

ADDQ is similar to ADDI but adds a specified value between 1 and 8 to an alterable address. When used with address registers, either word or long word sizes are used.

ADDX operates similarly to ADD, except that the value of the extend flag in the SR is added to the result.

AND

Syntax	AND <d/ea>,Dn	ANDI #<data>,CCR
	AND Dn,<ma/ea>	ANDI #<data>,SR
	ANDI #<data>,<da/ea>	

Mnemonic	Operation	Data size	Flags: N Z V C X
AND	Logical AND	B.W.L.	A A 0 0 –
ANDI	AND immediate	B.W.L.	A A 0 0 –
ANDI to CCR	AND immediate to CCR	B.	? ? ? ? ?
ANDI to SR	AND immediate to SR	W.	? ? ? ? ?

Description AND is a logical operation which ANDs a source operand, specified by a data addressing mode, with a data register. The result is stored in the destination. Note that the destination operand may not be in an address register.

ANDI performs an AND operation between a specified immediate data value and a destination location, with the result being stored in the destination location.

ANDI to CCR performs an AND operation between a specified immediate data value and the low-order byte of the status register, with the result being stored in the low-order byte of the status register (the CCR).

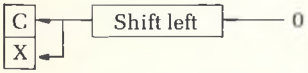
ANDI to SR performs an AND operation between a specified immediate data value and the current contents of the status register with the result being stored in the status register.

Arithmetic shift

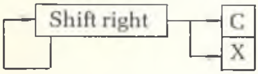
Syntax ASd Dx,Dy
ASd #<data>,Dy
ASd <ma/ea>

Mnemonic	Operation	Data size	Flags: NZ V C X
ASL	Arithmetic shift left	B.W.L.	A A A A A
ASR	Arithmetic shift right	B.W.L.	A A A A A

Description ASL shifts all the bits to the left, depositing the value of the leftmost bit into the carry flag and the extend flag and replacing the rightmost bit with a zero:



ASR shifts all the bits to the right, depositing the value of the rightmost bit into the carry flag and the extend flag and replicating the sign bit into the high-order bit. Two's complement arithmetic is assumed, and for negative numbers 1s are shifted in from the left:



Branching instructions

Syntax Bcc <label>
BRA <label>
BSR <label>

Mnemonic	Operation	Data size	Flags: NZVCX
BCC	Branch if carry clear	B.W.	-----
BCS	Branch if carry set	B.W.	-----
BEQ	Branch if equal	B.W.	-----
BGE	Branch if greater or equal	B.W.	-----
BGT	Branch if greater	B.W.	-----
BHI	Branch if high	B.W.	-----
BLE	Branch if less or equal	B.W.	-----
BLS	Branch if low or same	B.W.	-----
BLT	Branch if less than	B.W.	-----
BMI	Branch if minus	B.W.	-----
BNE	Branch if not equal	B.W.	-----
BPL	Branch if plus	B.W.	-----
BVS	Branch if overflow set	B.W.	-----
BVC	Branch if overflow reset	B.W.	-----
BRA	Branch always	B.W.	-----
BSR	Branch to subroutine	B.W.	-----

Description Bcc instructions branch to execution addresses calculated using a displacement value if specified conditions are true. The displacement is an 8-bit two's complement integer, unless it is zero, in which case the displacement is the 16-bit word immediately following the instruction.

The conditions will vary according to the Bcc variant used and these are summarized as follows:

BCC	if C=0
BCS	if C=1
BEQ	if Z=1
BGE	if either (N=1 and V=0) or (N=0 and V=1)
BGT	if either (N=1 and V=1 and Z=0) or (N=0 and V=0 and Z=0)
BHI	if C=0 and Z=0
BLE	if (N=1 and V=0) or if (N=0 and V=1) or if Z=1
BLS	if C=1 or Z=1
BLT	if either (N=1 and V=0) or (N=0 and V=1)
BMI	if N=1
BNE	if Z=0
BPL	if N=0
BVS	if V=1
BVC	if V=0

BRA is the unconditional branch instruction and uses a displacement in the same way as the conditional branching instructions.

BSR branches execution to a subroutine located at an address expressed as a two's complement offset from the program counter, measured in bytes. If it is zero, then the 16-bit number following the instruction is used. The return address is pushed on to the stack.

Bit testing

Syntax	BCHG Dn,<da/ea>	BSET Dn,<ea>
	BCHG #<data>,<da/ea>	BSET #<data>,<ea>
	BCLR Dn,<ea>	BTST Dn,<d/ea>
	BCLR #<data>,<ea>	BTST #<data>,<d/ea>

Mnemonic	Operation	Data size	Flags: N Z V C X
BCHG	Bit test and change	B.L.	– A– – –
BCLR	Bit test and clear	B.L.	– A– – –
BSET	Bit test and set	B.L.	– A– – –
BTST	Bit test	B.L.	– A– – –

Description BCHG is used to test the value of a bit which is specified either by an immediate data value or by a data register. The destination operand is addressed by a data alterable address mode and the bit tested is copied to the Z flag of the status register. following which the original bit is complemented (changed to its opposite value). Where the destination operand is a data register, then the bit numbering is modulo 32 while in the case of memory address destinations, the bit numbering is modulo 8.

BCLR is the same as BCHG, except that after the bit is copied to the Z flag it is reset to zero in the destination.

BSET is the same as BCHG except that after the bit is copied to the Z flag it is set to 1 in the destination.

BTST is the same as BCHG except that the destination operand is addressed by any data addressing mode, and after the bit is copied into the Z flag it is left unaltered in the destination.

Check

Syntax CHK <d/ea>,Dn

Mnemonic	Operation	Data size	Flags: NZVCX
CHK	Check and trap	W.	XXXX–

Description CHK is used to compare the two's complement integer which forms the source operand with the low-order byte of a data register destination. An exception is initiated if the destination is either less than zero or greater than the source operand.

Clear

Syntax CLR <da/ea>

Mnemonic	Operation	Data size	Flags: NZVCX
CLR	Reset destination byte(s)	B.W.L.	0 1 0 0 –

Description CLR resets the bits in the destination location to zero.

Comparison

Syntax CMP <ea>,Dn CMPI #<data>,<da/ea>
CMPA <ea>,An CMPM (Ay)+,(Ax)+

Mnemonic	Operation	Data size	Flags: NZVCX
CMP	Compare source and destination	B.W.L.	AAAA–
CMPA	Compare address	W.L.	AAAA–
CMPI	Compare immediate	B.W.L.	AAAA–
CMPM	Compare memory	B.W.L.	AAAA–

Description CMP compares two values by subtracting the first from the second without altering the value of the second. The source operand may be anything except an address register with a byte value and the destination may only be a data register. The instruction affects all the condition codes except X.

CMPA is the same as CMP but may only be used with an address register as the destination. Any address mode may be used and word length operations are sign extended to 32 bits prior to comparison.
CMPI is the same as CMP but the source operand is always an immediate data value and the destination may be any data alterable mode.
CMPM is used to compare memory locations and both the source and destination operands may only be addressed in postincrement modes.

Decrement and branch

Syntax DBcc Dn,<label>
DBRA

Mnemonic	Operation	Data size	Flags: N Z V C X
DBCC	Dec. & branch if carry clear	W.	- - - - -
DBCS	Dec. & branch if carry set	W.	- - - - -
DBEQ	Dec. & branch if =	W.	- - - - -
DBF	Dec. & branch if false	W.	- - - - -
DBGE	Dec. & branch if > or =	W.	- - - - -
DBGT	Dec. & branch if >	W.	- - - - -
DBHI	Dec. & branch if high	W.	- - - - -
DBLE	Dec. & branch if < or =	W.	- - - - -
DBLS	Dec. & branch if low or same	W.	- - - - -
DBLT	Dec. & branch if <	W.	- - - - -
DBMI	Dec. & branch if minus	W.	- - - - -
DBNE	Dec. & branch if <>	W.	- - - - -
DBPL	Dec. & branch if plus	W.	- - - - -
DBVS	Dec. & branch if overflow set	W.	- - - - -
DBVC	Dec. & branch if overflow reset	W.	- - - - -
DBT	Dec. & branch if true	W.	- - - - -
DBRA	Unconditional branch	W.	- - - - -

Description DBcc instructions check the contents of a data register against specified termination conditions. If the conditions are not met, the register's low order word is decremented and if the result is not −1, execution branches to a location calculated as the PC plus a 16 bit sign extended displacement word.

The termination conditions will vary according to the DBcc variant used and these are summarized as follows:

DBCC	if C=0
DBCS	if C=1
DBEQ	if Z=1
DBF	if false
DBGE	if either (N=1 and V=0) or (N=0 and V=1)
DBGT	if either (N=1 and V=1 and Z=0) or (N=0 and V=0 and Z=0)
DBHI	if C=0 and Z=0
DBLE	if (N=1 and V=0) or if (N=0 and V=1) or if Z=1
DBLS	if C=1 or Z=1
DBLT	if either (N=1 and V=0) or (N=0 and V=1)
DBMI	if N=1
DBNE	if Z=0
DBPL	if N=0
DBVS	if V=1
DBVC	if V=0
DBT	if true

DBRA operates in the same way as DBcc but without a conditional termination.

Division

Syntax DIVS <d/ea>,Dn
DIVU <d/ea>,Dn

Mnemonic	Operation	Data size	Flags: N Z V C X
DIVS	Signed division	W.	A A A 0 –
DIVU	Unsigned division	W.	A A A 0 –

Description DIVS divides a 32-bit destination operand in a data register by a 16-bit source operand, with the result being stored in the destination. Unless the remainder is equal to zero, its sign will be the same as that of the dividend. Note that any overflow will be flagged during computation without the operands being affected. A trap exception will be initiated by attempts to divide by zero (an illegal instruction).

$$\boxed{\text{Data register Dn}} \; / \; \boxed{\text{Source op.}}$$
$$= \boxed{\text{Remainder} \;\big|\; \text{Quotient}} \; = \text{Dn}$$

DIVU is similar to DIVS except that it is performed using unsigned arithmetic.

Exclusive OR

Syntax	EOR Dn,<da/ea>	EORI #<data>,CCR
	EORI #<data>,<da/ea>	EORI #<data>,SR

Mnemonic	Operation	Data size	Flags: N	Z	V	C	X
EOR	Logical exclusive OR	B.W.L.	A	A	0	0	–
EORI	EOR immediate	B.W.L.	A	A	0	0	–
EORI to CCR	EOR immediate to CR	B.	?	?	?	?	?
EORI to SR	EOR immediate to SR	W.	?	?	?	?	?

Description EOR performs an exclusive OR operation between a data register source operand and a destination operand, with the result being stored in the destination.

EORI performs an exclusive OR operation between a specified immediate data operand and a destination location, with the result being stored in the destination.

EORI to CCR performs an exclusive OR operation between a specified immediate data byte and the low-order byte of the status register, with the result being stored in the low-order byte of the status register.

EORI to SR is a privileged instruction which performs an exclusive OR operation between a specified immediate data word and the status register with the result being stored in the status register.

Exchange registers

Syntax EXG Rx,Ry

Mnemonic	Operation	Data size	Flags: N Z V C X
EXG	Exchange registers	L.	– – – – –

Description EXG exchanges the 32-bit contents of two specified registers, which may be two data registers, two address registers, or one of each type.

Sign extension

Syntax EXT Dn

Mnemonic	Operation	Data size	Flags: N Z V C X
EXT	Sign extend	W.L.	A A 0 0 –

Description The sign bit of a data register is copied in order to extend the length of the operand. To extend a byte to word length, bit 7 is copied to bits 8 to 15. To extend a word to long word length, bit 15 is copied to bits 16 to 31.

Jump operations

Syntax JMP <c/ea>
JSR <c/ea>

Mnemonic	Operation	Data size	Flags: N Z V C X
JMP	Jump		– – – – –
JSR	Jump to subroutine		– – – – –

Description The JMP instruction directs execution to an address specified by a control addressing mode.

The JSR instruction redirects execution to a subroutine at a specified address, after first pushing the return address on.to the system stack.

Load effective address

Syntax LEA <c/ea>,An

Mnemonic	Operation	Data size	Flags: NZVCX
LEA	Load effective address	L.	-----

Description Loads an effective address into a specified address register.

Link operations

Syntax LINK An,#<displacement>

Mnemonic	Operation	Data size	Flags: NZVCX
LINK	Link to subroutine		-----

Description The LINK operation initiates the following sequence of actions:

(1) The contents of the address register specified in the instruction are pushed on to the stack.
(2) The stack pointer (A7) is loaded into the specified address register.
(3) The stack pointer is decremented by an amount specified by a sign extended 16-bit two's complement displacement integer.

The effect of these actions is to create an area within the stack which may be used for the storage of data. For example, in the instruction **LINK A4,#−4** the contents of address register A4 are first saved on the stack. The stack pointer (A7) is then loaded into A4 so that A4 becomes the base index register for the reserved data area which is about to be created. A7 is then decremented by 4 bytes, thus moving the top of the stack by a long word length.

The stack area between the addresses contained in A4 and A7 is thus reserved for the storage of data:

Address	Stack contents	Stack pointer contents (A7)	Base index (A4) contents	Comment
n			#xxxx	Store A4 on stack
n−1	x			
n−2	x			
n−3	x			
n−4	x	n−4	n−4	Load A7 into A4
n−5				
n−6				
n−7				
n−8		n−8		Decrement A7 by 4
n−9				
n−10				
n−11				

There are now effectively two stack pointers. The normal stack pointer, A7, points to the new top of the ordinary user stack while the additional stack pointer, A4, points to the data area within the ordinary stack. Register A4 retains its contents and acts as a base index pointer. Thus data is entered in the data area to an address relative to A4, defined by a negative displacement value (since the stack extends downwards in memory). For example, **MOVE.L D2,−4(A4)** which means move the contents of D2 into a memory area starting at an address which is four less than the address contained in A4 (in this case addresses n−8 to n−5).

Data is stored at the top of the stack in the normal way. For example, **MOVE.L D2,−(A7)** which means decrement the stack pointer by 4 (since this is a long word operation) and move the contents of D2 into the stack area starting from the address pointed to by A7.

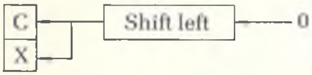
The data area may be de-allocated by means of the UNLK instruction which reverses the actions of the LINK instruction (see UNLK).

Logical shifts

Syntax LSd Dx,Dy
LSd #<data>,Dn
LSd <ma/ea>

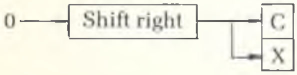
Mnemonic	Operation	Data size	Flags: N Z V C X
LSL	Logical shift left	B.W.L.	A A 0 A A
LSR	Logical shift right	B.W.L.	A A 0 A A

Description LSL shifts all the bits to the left, depositing the value of the leftmost bit in the carry flag and the extend flag and replacing the rightmost bit with a zero:



Shift operations in memory are always of word size and the bits are shifted by 1. Where the operands are in data registers they may be of byte, word or long word size and the bits are shifted by between 1 and 8 places as specified by a constant in the instruction (for example, **LSL.W #2,Dn**) or by the value indicated in another data register, modulo 64 (for example, **LSL.W D0,D1**).

LSR shifts all the bits to the right, depositing the value of the rightmost bit in the carry and extend flags and replacing the leftmost bit with a zero:



LSR operations on data register operands are subject to the same length specifications as those of LSL.

Move operations

Syntax	MOVE <ea>,<da/ea>	MOVEQ #<data>,Dn
	MOVEA <ea>,An	MOVE <d/ea>,CCR
	MOVEM <register list>,<ca/ea>	MOVE <d/ea>,SR
	MOVEM <c/ea>,<register list>	MOVE SR,<da/ea>
	MOVEM (An)+,<reg. list>	MOVE USP,An
	MOVEM <reg. list>,−(An)	MOVE An,USP
	MOVEP Dx,d(Ay)	MOVEC Rn,(Cr)
	MOVEP d(Ay),Dx	MOVES <ea>,Dn
	MOVEC (Cr),Rn	MOVE CCR,<ea>
	MOVES DN,<ea>	

Mnemonic	Operation	Data size	Flags: NZVCX
MOVE	Move data	B.W.L.	AA00–
MOVEA	Move address	W.L.	–––––
MOVEM	Move multiple	W.L.	–––––
MOVEP	Move to peripheral	W.L.	–––––
MOVEQ	Move quick	L.	AA00–
MOVE to CCR		W.	?????
MOVE from CCR		W.	–––––
MOVE to SR		W.	?????
MOVE from SR		W.	–––––
MOVE USP	Move user stack pointer	L.	–––––
MOVEC	Move control register	L.	–––––
MOVES	Move alternate addr. space	B.W.L.	–––––

Description MOVE is used to move data between two effective addresses, and operations may be of byte, word or long word length.

MOVEA is the same as MOVE except that it is used to move data from an effective address to an address register. Operations may only be of word or long word length and word length source operands are sign extended to 32 bits before the move takes place.

MOVEM is used to stack or unstack the contents of multiple registers. See example under 'Stack handling'.

MOVEP is used for moving data between a data register and a location pointed to by an address register with displacement. For example, **MOVEP L. Dn,d(An)** or **MOVEP W. d(An),Dn**. Data is transferred one byte at a time, to or from *alternate* locations. For example, in a long word length MOVEP operation, **MOVEP L. Dn,d(An)**, if d(An) points to address n (which may be an odd or even address), data is transferred to addresses n, n+2, n+4 and n+6, starting with the high-order byte of data held in the source register. This instruction may be used for transferring data to or from peripheral control ports.

MOVEQ is used for the quick transfer of data into data registers and can only be used as a long word operation. It moves a specified signed byte value into the least significant byte of the data register, which is then sign extended to a full 32 bits.

MOVE to CCR takes a word length value and moves it into the condition codes register (CCR), which is the low-order word of the status register. Thus bits 0 to 7 of the status register may be set to specific values.

MOVE from CCR is used on the MC68010 to move data from the condition codes register.

MOVE to SR is a privileged instruction which moves a long word length value into the whole of the status register.

MOVE from SR moves the contents of the status register to a data register or to a data alterable address. On the MC68010, this is classified as a privileged instruction.

MOVE USP is a privileged instruction which allows data to be moved into or out of the user stack pointer while the processor is in supervisor state. This is used, for example, by the operating system to set the initial value of the USP.

MOVEC is a privileged instruction used on the MC68010 to move data to or from a control register such as the VSR or the function code registers.

MOVES is a privileged instruction used with the MC68010 to allow a program in supervisor state to access areas which are normally inaccessible to the system such as the supervisor program area, the user program area and the user data area. The area accessed by a MOVES instruction is specified by codes placed in the function code registers by means of a MOVEC instruction.

Multiply

Syntax MULS <d/ea>,Dn
MULU <d/ea>,Dn

Mnemonic	Operation	Data size	Flags: N Z V C X
MULS	Signed multiplication	W.	A A 0 0 –
MULU	Unsigned multiplication	W.	– – – – –

Description MULS multiplies a signed 16-bit operand and the low-order word of a data register, the 32-bit result being stored in the destination register.

MULU operates in the same way as MULS but uses unsigned operands.

Negate decimal

Syntax NBCD <da/ea>

Mnemonic	Operation	Data size	Flags: N Z V C X
NBCD	Decimal negation	B.	X ? X A A

Description NBCD is a decimal arithmetic operation which subtracts both the destination operand and the extend bit from zero, with the result being placed in the destination. The result will either be the ten's complement of the destination or the nine's complement, depending on whether the extend flag is reset or set. The Z flag should be set beforehand if required subsequently.

Negate binary

Syntax NEG <da/ea>
NEGX <da/ea>

Mnemonic	Operation	Data size	Flags: N Z V C X
NEG	Binary negation	B.W.L.	A A A A A
NEGX	Binary negation with extend	B.W.L.	A ? A A A

Description The NEG instruction subtracts the destination operand from zero and stores the result in the destination.
NEGX operates similarly except that both the destination operand and the extend bit are subtracted from zero.

No operation

Syntax NOP

Mnemonic	Operation	Data size	Flags: N Z V C X
NOP	No operation		– – – – –

Description The NOP instruction has no effect and execution continues with the following instruction.

NOT operations

Syntax NOT <da/ea>

Mnemonic	Operation	Data size	Flags: N Z V C X
NOT	Logical complement	B.W.L.	A A 0 0 –

Description NOT is used to invert all the bits in an operand from ones to zeros and vice versa.

OR operations

Syntax OR <d/ea>,Dn
OR Dn,<ma/ea>
ORI #<data>,<da/ea>
ORI #<data>,CCR
ORI #<data>,SR

Mnemonic	Operation	Data size	Flags: N Z V C X
OR	Logical OR	B.W.L.	A A 0 0 –
ORI	OR immediate	B.W.L.	A A 0 0 –
ORI to CCR	OR immediate to CCR	B.	? ? ? ? ?
ORI to SR	OR immediate to SR	W.	? ? ? ? ?

Description OR performs a logical OR operation between the source and destination operand, with the result being stored in the destination. The source may not be an address register.

ORI performs a logical OR operation between an immediate data value and the destination operand, with the result being stored in the destination.

ORI to CCR performs a logical OR operation between an immediate data value and the lower half of the status register, with the result being stored in the destination.

ORI to SR performs a logical OR operation between an immediate data value and the entire status register, with the result being stored in the destination. This is a privileged instruction.

Push effective address

Syntax PEA <c/ea>

Mnemonic	Operation	Data size	Flags: NZVCX
PEA	Push effective address	L.	– – – – –

Description Computes the effective address and pushes it on to the stack.

Reset

Syntax RESET

Mnemonic	Operation	Data size	Flags: NZVCX
RESET	Reset		– – – – –

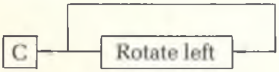
Description RESET is a privileged instruction which resets all external devices. If the processor is not in the supervisor mode when RESET occurs, a trap exception takes place.

Rotation

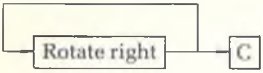
Syntax	ROd Dx,Dy	ROXd Dx,Dy
	ROd #<data>,Dy	ROXd #<data>,Dy
	ROd <ma/ea>	ROXd <ma/ea>

Mnemonic	Operation	Data size	Flags: N Z V C X
ROL	Rotate left	B.W.L.	A A 0 A –
ROXL	Rotate left with extend	B.W.L.	A A 0 A A
ROR	Rotate right	B.W.L.	A A 0 A –
ROXR	Rotate right with extend	B.W.L.	A A 0 A A

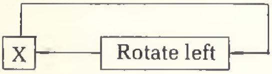
Description ROL rotates the bits of an operand to the left by a specified amount, depositing a copy of the leftmost bit in the carry flag and also transferring it to the rightmost bit position:



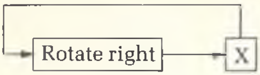
ROR rotates the bits of an operand to the right by a specified amount, depositing a copy of the rightmost bit in the carry flag and also transferring it to the leftmost bit position:



ROXL operates in a similar manner to ROL, except that the leftmost bit is copied into the extend flag and the previous value of the extend flag is transferred to the rightmost bit position:



ROXR is similar to ROXL but shifts the bits to the right, copying the rightmost bit into the extend flag and transferring the previous value of the extend flag into the leftmost bit position:



Return operations

Syntax RTE
RTR
RTS

Mnemonic	Operation	Data size	Flags: NZVCX
RTE	Return from exception		AAAAA
RTR	Return and restore CCR		AAAAA
RTS	Return from subroutine		-----

Description RTE is a privileged instruction which pulls the status register and program counter from the supervisor stack following an exception process, thus restoring the previous execution sequence and conditions.

RTR is similar to RTE except that instead of retrieving the whole of the status register it retrieves only the user section (CCR) containing the condition flag codes. This is an unprivileged instruction.

RTS is used to return from a subroutine. The old program counter and status register values are returned from either the user or supervisor stack and execution continues from the address in the restored program counter.

Subtract decimal

Syntax SBCD Dy,Dx
SBCD −(Ay),−(Ax)

Mnemonic	Operation	Data size	Flags: N Z V C X
SBCD	Decimal subtraction	B.	X ? X A A

Description SBCD is a decimal arithmetic operation in which the source operand and the extend bit are subtracted from the destination operand. The zero flag should be set beforehand if required subsequently.

Set from condition

Syntax Scc <da/ea>

Mnemonic	Operation	Data size	Flags: N Z V C X
SCC	Set if carry clear	B.	- - - - -
SCS	Set if carry set	B.	- - - - -
SEQ	Set if equal	B.	- - - - -
SF	Set if false	B.	- - - - -
SGE	Set if greater or equal	B.	- - - - -
SGT	Set if greater	B.	- - - - -
SHI	Set if high	B.	- - - - -
SLE	Set if less or equal	B.	- - - - -
SLS	Set if low or same	B.	- - - - -
SLT	Set if less than	B.	- - - - -
SMI	Set if minus	B.	- - - - -
SNE	Set if not equal	B.	- - - - -
SPL	Set if plus	B.	- - - - -
ST	Set if true	B.	- - - - -
SVS	Set if overflow set	B.	- - - - -
SVC	Set if overflow reset	B.	- - - - -

Description Scc tests a specified condition code. The byte defined as the destination is set to FF if the condition is satisfied, else it is zeroed. Scc can only be used to set single bytes specified by data alterable addressing modes.

The conditions under which the destination byte is set will vary according to the Scc variant used and these are summarized as follows:

SCC	if C=0
SCS	if C=1
SEQ	if Z=1
SF	if false
SGE	if either (N=1 and V=0) or (N=0 and V=1)
SGT	if either (N=1 and V=1 and Z=0) or (N=0 and V=0 and Z=0)
SHI	if C=0 and Z=0
SLE	if (N=1 and V=0) or if (N=0 and V=1) or if Z=1
SLS	if C=1 or Z=1
SLT	if either (N=1 and V=0) or (N=0 and V=1)
SMI	if N=1
SNE	if Z=0
SPL	if N=0
ST	if true
SVS	if V=1
SVC	if V=0

Stop

Syntax STOP #<data>

Mnemonic	Operation	Data size	Flags: N Z V C X
STOP	Stop execution		A A A A A

Description STOP is a privileged instruction which causes a program to stop until a trace, a high-priority interrupt or reset exception is initiated. Upon commencement of the STOP operation, the immediate data operand following the stop instruction is loaded into the status register and the program counter is incremented to point to the following execution address.

Subtract binary

Syntax SUB <ea>,Dn
SUB Dn,<ma/ea>
SUBA <ea>,An
SUBI #<data>,<da/ea>
SUBQ #<data>,<a/ea>
SUBX Dy,Dx
SUBX −(Ay),−(Ax)

Mnemonic	Operation	Data size	Flags: NZVCX
SUB	Binary subtraction	B.W.L.	AAAAA
SUBA	Subtract address	W.L.	–––––
SUBI	Subtract immediate	B.W.L.	AAAAA
SUBQ	Subtract quick	B.W.L.	AAAAA
SUBX	Subtract with extend	B.W.L.	A? AAA

Description SUB the source operand is subtracted from the destination operand, with the result being stored in the destination.

SUBA is similar to SUB, except that the destination must be an address register. Byte operations are therefore disallowed.

SUBI subtracts immediate data from a destination operand, with the result being stored in the destination.

SUBQ subtracts immediate data between 1 and 8 from a destination operand, with the result being stored in the destination.

SUBX subtracts both the source operand and the extend bit from the destination operand, with the result being stored in the destination. The Z flag should be set beforehand if required subsequently.

Swap

Syntax SWAP Dn

Mnemonic	Operation	Data size	Flags: NZVCX
SWAP	Exchange register words	W.	AA0 0 –

Description SWAP exchanges the upper and lower halves of a data register.

Test bit

Syntax TAS <da/ea>

Mnemonic	Operation	Data size	Flags: NZVCX
TAS	Test bit and set	B.	AA00–

Description TAS is used to test and set the most significant bit of a byte operand. If the most significant bit of the operand is set, then the N flag is set, and if the operand is zero, the Z flag is set. The most significant bit is then set, irrespective of its previous value.

In systems where several computers share the same memory resources, TAS may be used to test a flag in a particular byte to establish whether certain resources are accessible and to set the flag to exclude access by other systems. Thus, if a TAS instruction encounters a set flag, the N flag will be set, which may itself be tested and used to trigger off an alternative course of action. If a TAS instruction encounters a reset flag, it will claim access to the resource by setting the flag and thus excluding access by other systems. No other system is able to access the operand containing the flag bit while a TAS operation is being executed.

Trap

Syntax TRAP #<vector>
TRAPV

Mnemonic	Operation	Data size	Flags: NZVCX
TRAP	Initiate trap exception		–––––
TRAPV	Initiate trap exception if overflow		–––––

Description TRAP forces a TRAP exception which is vectored to an address calculated from bits 0 to 3 of the instruction word.

TRAPV causes a TRAP exception vectored to the TRAPV exception vector address if the overflow flag is set, otherwise it has no effect.

Test against zero

Syntax TST <da/ea>

Mnemonic	Operation	Data size	Flags: N Z V C X
TST	Compare with zero	B.W.L.	A A 0 0 –

Description TST compares the operand with zero and sets the condition flags according to the result.

Unlink

Syntax UNLK An

Mnemonic	Operation	Data size	Flags: N Z V C X
UNLK	Unlink subroutine		– – – – –

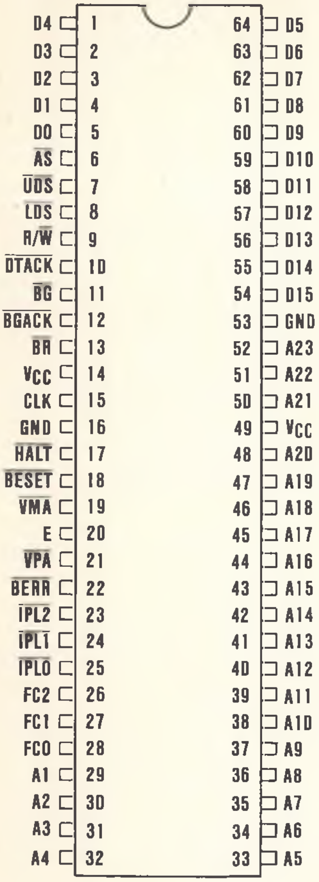
Description UNLK is used to de-allocate data space which has been reserved within the stack (*see* LINK). The series of actions initiated by an UNLK instruction is as follows:

(1) The contents of a specified address register (An) are moved into the stack pointer (A7).
(2) The long word at the base of the stack is loaded into An.

If we take the example shown under 'LINK', where A4 has been used as the base index register for the data area, the UNLK operation will restore the address originally held in the stack pointer (A4 is moved into A7) and the original value of A4 (#xxxx) is retrieved and replaced in A4.

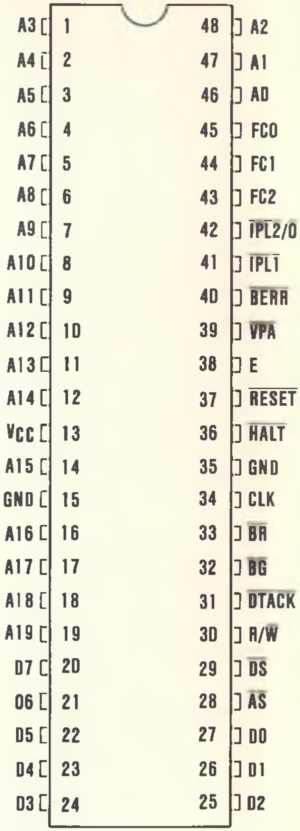
Appendix
Pin assignments

MC68000 and MC68010
64-Pin Dual-in-Line Package

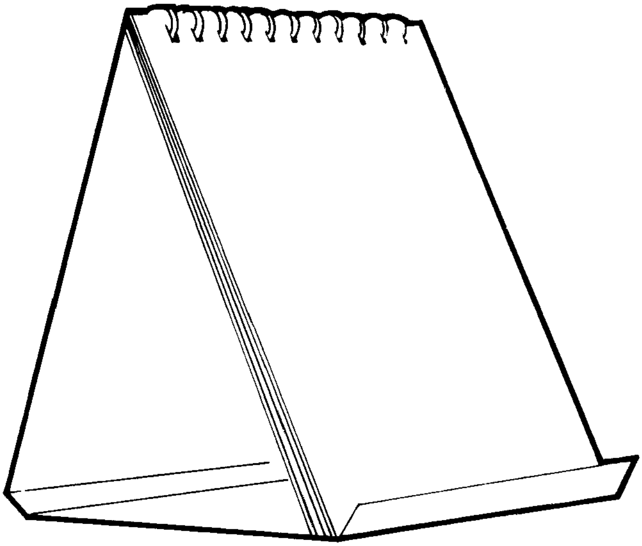


D4 1 64 D5
D3 2 63 D6
D2 3 62 D7
D1 4 61 D8
D0 5 60 D9
$\overline{AS}$ 6 59 D10
$\overline{UDS}$ 7 58 D11
$\overline{LDS}$ 8 57 D12
R/$\overline{W}$ 9 56 D13
$\overline{DTACK}$ 10 55 D14
$\overline{BG}$ 11 54 D15
$\overline{BGACK}$ 12 53 GND
$\overline{BR}$ 13 52 A23
V_{CC} 14 51 A22
CLK 15 50 A21
GND 16 49 V_{CC}
$\overline{HALT}$ 17 48 A20
$\overline{BESET}$ 18 47 A19
$\overline{VMA}$ 19 46 A18
E 20 45 A17
$\overline{VPA}$ 21 44 A16
$\overline{BERR}$ 22 43 A15
$\overline{IPL2}$ 23 42 A14
$\overline{IPL1}$ 24 41 A13
$\overline{IPL0}$ 25 40 A12
FC2 26 39 A11
FC1 27 38 A10
FC0 28 37 A9
A1 29 36 A8
A2 30 35 A7
A3 31 34 A6
A4 32 33 A5

MC68008
48-Pin Dual-in-Line Package



A3 1 48 A2
A4 2 47 A1
A5 3 46 A0
A6 4 45 FC0
A7 5 44 FC1
A8 6 43 FC2
A9 7 42 $\overline{IPL2/0}$
A10 8 41 $\overline{IPL1}$
A11 9 40 $\overline{BERR}$
A12 10 39 $\overline{VPA}$
A13 11 38 E
A14 12 37 $\overline{RESET}$
V_{CC} 13 36 $\overline{HALT}$
A15 14 35 GND
GND 15 34 CLK
A16 16 33 $\overline{BR}$
A17 17 32 $\overline{BG}$
A18 18 31 $\overline{DTACK}$
A19 19 30 R/$\overline{W}$
D7 20 29 $\overline{DS}$
D6 21 28 $\overline{AS}$
D5 22 27 D0
D4 23 26 D1
D3 24 25 D2



The diagram shows, how to arrange the Pocket Guide in an upright position.

Bend at score mark indicated by arrow.

Pitman

The Modern Book Co.

2.95

ISBN 0-273-02152-4

9 780273 021520