68000 Instruction Set

Assembly language instructions are always listed by mnemonic (or name) in alphabetical order. For each instruction, the following information is provided:

- the instruction name (mnemonic) with a brief description,
- the operation description in register transfer language (RTL),
- the assembler instruction syntax,
- the attributes which indicate the operand size (i.e. byte, word, or longword),
- the description of the instruction operation in words,
- the effect instruction execution has on the condition codes,
- the addressing modes valid for the instruction if not obvious from the syntax,
- sample code to demonstrate instruction use.

Operand Notation

Dn	Data register
An	Address register
Rn	Data or Address register
PC	Program Counter
SR	Status Register
CCR	Condition Code Register. Lower order byte of the Status Register
SSP	Supervisor Stack Pointer Register
USP	User Stack Pointer Register
SP	Active Stack Pointer Register which will be either SSP or USP. Equivalent to A7.
#<data>	An immediate value which may be 8, 16 or 32 bits in length, depending on the instruction. Equivalent to <immediate data>.
d	Address displacement.
d	Direction of data movement for shift or rotate.
source	Source operand.
destination	Destination operand. For unary operand instructions, the operand is always referred to as the destination.
ea	Effective address; can be any valid source/destination address.
[operand]	Contents of the operand.
[[operand]]	Contents of the memory location pointed to by the contents of the operand. Essentially, {contents of {contents of the operand}}.
(xxx).W	Absolute short addressing with a 16-bit address.
(xxx).L	Absolute long addressing with a 32-bit address.

(An)	Address register indirect.
(An)+	Address register indirect with post-increment.
−(An)	Address register indirect with pre-decrement.
d(An) (d,An)	Address register indirect with 16-bit displacement.*
d(An,Xi) (d,An,Xi)	Address register indirect with indexing and 8-bit displacement.*
d(PC) (d,PC)	Program counter indirect with 16-bit displacement.*
d(PC,Xi) (d,PC,Xi)	Program counter indirect with indexing and 8-bit displacement.*

* Two notations are employed for address register indirect with displacement addressing. Some simulators support only the older form which has the displacement outside of the leading bracket.

STOP	Enter the stopped state and wait for interrupts.
TRAP	Execute a TRAP condition interrupt.

The Status Register (SR)

Status = system byte (supervisor only) + user byte
= system status + condition code register

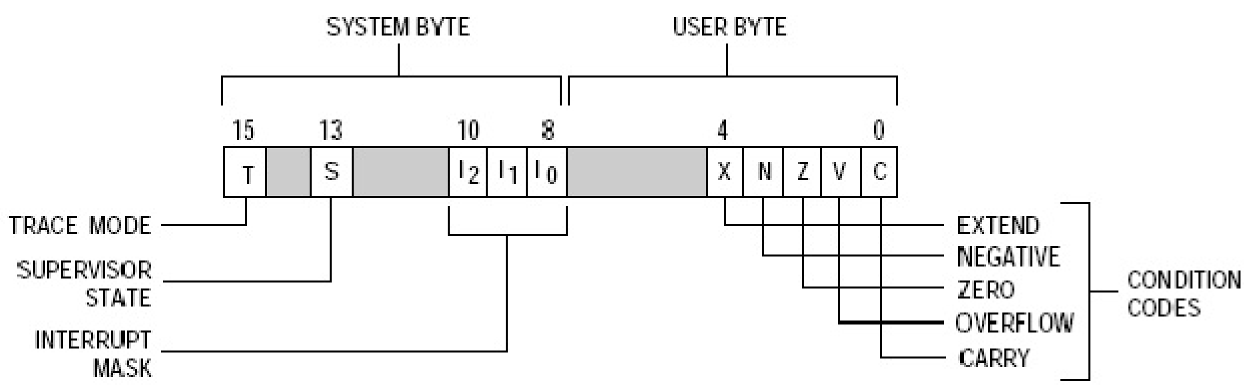

Fig 2-4 from M68000 8-/16-/32-Bit Microprocessors User's Manual

The Condition Code Register (CCR)

The condition code register portion of the status register contains five bits: N, Z, V, C, and X.

N, Z, V, and C are true condition code bits and they reflect the condition generated by the operation. The Extended Carry or Extend bit is a copy of the Carry bit that is used primarily for extended precision arithmetic. In general, X is set the same as C unless there would be negative consequences for extended precision calculations.

Most operations take a source operand and a destination operand to compute a result which is stored in the destination location. Operations with only one operand (referred to as the destination) compute a result which is stored in the destination operand. Computing a result sets the condition codes as follows:

N	Set if the most significant bit of the result is set; cleared otherwise.
Z	Set if the result equals zero; cleared otherwise.
V	Set if there was an arithmetic overflow; cleared otherwise. An arithmetic overflow implies that the result can not be represented in the number of bits used for the operand.
C	Set if a carry (borrow) is generated out of the most significant bit of the operand for an addition (subtraction); cleared otherwise.
X	Set the same as the C (carry) bit for most arithmetic operations but not for data movements or logical operations.

Condition Code Notation

Condition code register representation follows standard convention:

*	Set according to the result of the operation.
-	Not affected by the operation; contents of the bit are not changed.
0	Cleared.
1	Set.
U	Undefined or unpredictable after the operation.

In the instruction details, the condition codes will indicate how the condition flags are modified by the instruction. For example, consider the Logical AND instruction whose condition codes are:

```
X N Z V C
- * * 0 0
```

Using the above notation, this means that

X	Not affected; what was in X remains in X.
N	Set if the most significant bit of the result is set; cleared otherwise.
Z	Set if the result is zero; cleared otherwise.
V	Always cleared, i.e. set to zero.
C	Always cleared, i.e. set to zero.

Note that for the Logical AND instruction, the Extended Carry is not necessarily the same as C.

ABCD — Add decimal with extend — ABCD

Operation: $[destination]_{10} \leftarrow [source]_{10} + [destination]_{10} + [X]$

Syntax:
ABCD Dy, Dx
ABCD −(Ay), −(Ax)

Attributes: Size = byte

Description: Add the source operand to the destination operand along with the extend bit, and store the result in the destination location. The addition is performed using *binary coded decimal* (BCD) arithmetic. The only legal addressing modes are
(a) data register direct to data register direct and
(b) memory to memory using address register indirect with pre-decrementing.

Condition codes:

X	N	Z	V	C
*	U	*	U	*

The Z-bit is cleared if the result is non-zero *and left unchanged otherwise*. The programmer would normally set the Z-bit before the BCD operation(s) to allow testing for a zero result.

Application: The ABCD instruction is used to add two BCD digits. It is assumed that the BCD digits are stored in a packed format with two BCD digits in one byte. This format is required to get the carry to work correctly for multi-digit additions.

Consider the addition of two 8 digit numbers stored in BCD. The numbers are stored so that addition starts at the least significant digit and proceeds to the most significant digit. For the example, the result of the addition is stored back into Num2; result will be $90129999.

```
        LEA      Num1+4,A0    ; point past 1st number
        LEA      Num2+4,A1    ; point past 2nd number
        MOVE.W   #4-1,D0      ; initialize digit counter
        MOVE.W   #$04,CCR     ; clear X and set Z **
Loop    ABCD     -(A0),-(A1)  ; add digits
        DBRA     D0,Loop      ; decrement loop counter til done

        ...
Num1    DC.L     $12345678
Num2    DC.L     $87784321
```

** clear X to prevent a carry-in to first addition; set Z to allow the result to be tested for zero. Test for zero was not done in this code segment.

ADD Add binary ADD

Operation: [destination] ← [source] + [destination]

Syntax: ADD <ea>, Dn
ADD Dn, <ea>

Attributes: Size = byte, word, longword

Description: Add the source operand to the destination operand using binary addition and store the result in the destination location.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The condition codes are not affected when the destination is an address register.

Source effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Destination effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔	✔	✔	✔	✔	✔	✔			

Application: For example, add the contents of D2 to the contents of D4 at word size.

```
ADD.W    D2,D4
```

If D2 = $30008206
and D4 = $100010F0
then result is D4 = $100092F6

Note: As this is a word length operation, only the low words are added and the top word of the destination is not modified.

ADDA **Add address** **ADDA**

Operation: [destination] ← [source] + [destination]

Syntax: ADDA <ea>, An

Attributes: Size = word, longword

Description: Add the source operand to the destination address register, and store the result in the destination address register. The entire destination register is used regardless of the operation size. The source is sign-extended before it is added to the destination address register.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Source effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Application: For example, add the contents of D2 to the contents of A4. For each of the following operations, the value from D2 is sign-extended to do the operation but the actual value in D2 is not changed.

```
ADDA.W     D2,A4
```

If D2 = $00008206
and A4 = $000010F0
sign-extend low word of D2 => $FFFF8206
and then add for result A4 => $FFFF92F6.

If D2 = $00007206
and A4 = $000010F0
sign-extend low word of D2 => $00007206
and then add for result A4 => $000082F6

If D2 = $10008206
and A4 = $000010F0
sign-extend low word of D2 => $FFFF8206
and then add for result A4 => $FFFF92F6.

If D2 = $00008206
and A4 = $100010F0
sign-extend low word of D2 => $FFFF8206
and then add for result A4 => $0FFF92F6.

ADDI Add immediate ADDI

Operation: [destination] ← <immediate data> + [destination]

Syntax: ADDI #<data>, <ea>

Attributes: Size = byte, word, longword

Description: Add the immediate data to the destination operand, and store the result in the destination operand. The size of the immediate data matches the operation size.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: For example, increment a loop counter in D0 by 4.

```
ADDI.B    #4,D0     ; increment loop counter
```

If D0 = $00000012 then result is D0 = $00000016

If D0 = $0000001E then result is D0 = $00000022

If D0 = $000000FF then result is D0 = $00000003

ADDQ — Add quick — ADDQ

Operation: [destination] ← <immediate data> + [destination]

Syntax: ADDQ #<data>, <ea>

Attributes: Size = byte, word, longword
Note: Only word and longword can be used if the destination is an address register.

Description: Add the immediate data to the destination operand, and store the result in the destination location. The immediate data must be in the range 1 to 8. When adding to address registers, the entire register is used regardless of the size attribute used.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The condition codes are not affected if the destination operand is an address register.

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔	✔	✔	✔	✔	✔	✔	✔	✔			

Application: ADDQ is used to add a small constant. ADDQ generates less machine code than ADDI for the comparable addition.

```
        ADDQ.B    #4,D0          ; increment loop counter

If      D0 = $00000012     then result is   D0 = $00000016

If      D0 = $0000001E     then result is   D0 = $00000022

If      D0 = $000000FF     then result is   D0 = $00000003


        ADDQ.W    #4,A0          ; increment address pointer

If      A0 = $00001012     then result is   A0 = $00001016

If      A0 = $0000F0FF     then result is   A0 = $0000F103

If      A0 = $00FFFFFF     then result is   A0 = $01000003
```

ADDX — Add extended — ADDX

Operation: [destination] ← [source] + [destination] + [X]

Syntax:
```
ADDX    Dy, Dx
ADDX    -(Ay), -(Ax)
```

Attributes: Size = byte, word, longword

Description: Add the source operand to the destination operand along with the extend bit, and store the result in the destination location. The only legal addressing modes are
(a) data register direct to data register direct and
(b) memory to memory using address register indirect with pre-decrementing.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The Z-bit is cleared if the result is non-zero and *left unchanged otherwise*. Normally, the programmer sets the Z bit before starting a series of multiple precision operations. The Z-bit can then be used to test for zero after the completion of the multiple precision operations.

Application: ADDX is used for multiple precision arithmetic, e.g. 64 bit addition.

```
        LEA       Num1+8,A0     ; point past 1st number
        LEA       Num2+8,A1     ; point past 2nd number
        MOVE.W    #$04,CCR      ; clear X and set Z **
        ADDX.L    -(A0),-(A1)   ; add digits
        ADDX.L    -(A0),-(A1)   ; add digits
        ...
Num1    DC.L      $12345678,$87654321
Num2    DC.L      $87654321,$12345678
```

Execution of the program will result in Num2 containing $99999999,$99999999.

** Clear X to prevent a carry-in to first addition; set Z to allow the result to be tested for zero. Test for zero was not done in this code segment.

AND — AND logical — AND

Operation: [destination] ← [source] AND [destination]

Syntax:
AND <ea>, Dn
AND Dn, <ea>

Attributes: Size = byte, word, longword

Description: Logically AND the source operand to the destination operand and store the result in the destination location.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Source effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔	✔	✔	✔	✔	✔	✔			

Application: Typically, AND is used to “mask” bits, i.e. clear bits. For example, to clear bits 0 through 3 in D0,

```
        AND.B    #%11110000,D0    ; clear lower nibble
or
MASK    EQU      $F0              ; define mask
        AND.B    #MASK,D0         ; mask out lowest digit
```

For D0 = $12345678 the result will be D0 = $12345670

ANDI — AND logical immediate — ANDI

Operation: [destination] ← <immediate data> AND [destination]

Syntax: ANDI #<data>, <ea>

Attributes: Size = byte, word, longword

Description: Logically AND the immediate data to the destination operand and store the result in the destination location. The size of the immediate data matches the operation size.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: Typically, ANDI is used to mask bits, i.e. clear bits, according to a specified bit pattern or mask. For example, to clear bits 0 through 3 in memory location LIGHTS

```
        ANDI.B    #%11110000,LIGHTS     ; clear lower set of lights
or
        ANDI.B    #$F0,LIGHTS           ; turn off lower light bank


For     LIGHTS    DC.B   %11001010
```

the result will be LIGHTS = %11000000

ANDI to CCR — AND immediate to condition code register — ANDI to CCR

Operation: [CCR] ← <immediate data> AND [CCR]

Syntax: ANDI #<data>, CCR

Attributes: Size = byte

Description: Logically AND the immediate data to the condition code register (i.e., the least-significant byte of the status register) and store the result in the condition code register. The size of the immediate data matches the operation size.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Application: ANDI is used to mask or clear selected bits of the CCR. The example for ADDX instruction can be changed to:

```
        LEA       Num1+8,A0     ; point past 1st number
        LEA       Num2+8,A1     ; point past 2nd number
        ANDI.B    #$00,CCR      ; clear X
        ORI.B     #$04,CCR      ; and set Z
        ADDX.L    -(A0),-(A1)   ; add digits
        ADDX.L    -(A0),-(A1)   ; add digits
        ...
Num1    DC.L      $12345678,$87654321
Num2    DC.L      $87654321,$12345678
```

Execution of the program will result in Num2 containing $99999999,$99999999

ANDI to SR **AND immediate to status register (privileged)** **ANDI to SR**

Operation: If supervisor state
then [SR] ← <immediate data> AND [SR]
else privilege violation

Syntax: ANDI #<data>, SR

Attributes: Size = word

Description: Logically AND the immediate data to the status register and store the result in the status register. All bits of the status register (SR) are affected.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Application: The system half of the status register contains the interrupt mask, the supervisor bit and the trace bit. The user half of the status register contains the condition codes. For example, to clear the trace bit of the status register

```
ANDI.W      #$7FFF,SR     ; clear trace
```

ASL, ASR — Arithmetic shift left/right — ASL, ASR

Operation: [destination] ← [destination] shifted in direction *d* by <count>

Syntax:

AS*d*	#<data>, Dy	where	#<data> is the count
AS*d*	Dx, Dy		Dx contains the count
AS*d*	<ea>		count is 1
		where *d* is the direction, L or R	

Attributes: Size = byte, word, longword
Note: memory locations may be shifted by one bit only and the operand size is restricted to a word.

Description: Arithmetically shift the bits of the operand in the direction (i.e. left or right) specified. The shift count may be specified in one of three ways:

i. The count may be specified as immediate data for a shift range of 1 to 8 bits.
ii. The count is in a data register, Dx, for a shift range of 0 to 63 bits, i.e. the value is modulo 64.
iii. If no count is specified, the shift is assumed to be one bit.

An arithmetic shift left (ASL) shifts all bits in the operand to the left. The most-significant bit of the operand is shifted into the extend and carry bits. A zero is shifted into the least significant bit of the operand. The overflow bit is set if a sign change occurs during shifting.

An arithmetic shift right (ASR) shifts all bits in the operand to the right. The least-significant bit of the operand is shifted into the extend and carry bits. The most-significant bit of the operand is replicated into the left-most bit to preserve the sign of the number. The overflow bit will not be set as the sign does not change for an ASR.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

X and C are set according to the last bit shifted out of the operand. If the shift count is zero, C is cleared. V is set if the most-significant bit is changed *at any time* during the shift operation and cleared otherwise.

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔	✔	✔	✔	✔	✔	✔			

Application: Arithmetic shifts can be used to efficiently multiply and divide by powers of 2.

```
ASL.L   #1,D0        ; multiply by 2
```

Bcc **Branch on condition cc** **Bcc**

Operation: If (condition true)
then [PC] ← [PC] + d

Syntax: Bcc <label>

Attributes: Size = byte, word

Description: The Bcc is a conditional branch where the "cc" in the instruction is replaced by the two letter code for the condition; a list of conditions and their interpretations follow. If the specified condition is true, program execution continues at label; otherwise, execution continues with the instruction immediately after the Bcc.

condition code (cc)	**interpretation**	**how condition is calculated**
CC HS*	carry clear high or same	~C
CS LO*	carry set low	C
NE	not equal	~Z
EQ	equal	Z
VC	overflow clear	~V
VS	overflow set	V
PL	plus	~N
MI	minus	N
GE	greater or equal	(N and V) or (~N and ~V)
LT	less than	(N and ~V) or (~N and V)
GT	greater than	(N and V and ~Z) or (~N and ~V and ~Z)
LE	less or equal	Z or (N and ~V) or (~N and V)
HI	high	~C and ~Z
LS	low or same	C or Z

* Support for these alternate mnemonics depends on the assembler.

The label is specified in the machine code as a displacement, d, relative to the current value of the program counter. The displacement is stored in two's complement; the original [PC] contains the sign-extended instruction location plus two.

The range of the branch is -126 to +128 bytes with an 8-bit displacement (i.e. Bcc.B), and -32K to +32K bytes with a 16-bit displacement (i.e. Bcc.W). Due to the machine code structure for the instruction, a Bcc.B to the next instruction not possible.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Application: Typically, the Bcc is used without a length specifier (defaults to word). If tight efficient machine code is required, a Bcc.B would be used after the code is fully debugged and the displacement is known to be within range.

Conditional branches are used for looping and decision structures. For example, compare two strings of the same length. Execution of ‘Loop’ continues until characters do not match or end of string reached.

```
            LEA         STRING1,A0          ; point to start of first string
            LEA         STRING2,A1          ; and the second string
            MOVE.W      #STRING_LEN,D0      ; initialize string counter
Loop        CMPM.B      (A0)+,(A1)+         ; compare pair of characters
            BNE         NotSame             ; exit loop if not match
            SUBQ.W      #1,D0               ; decrement counter
            BNE         Loop                ; loop until end of strings
Same        ....                            ; at this point strings matched
            ....
NotSame     ...                             ; code to handle mismatched strings
            ....
```

BCHG — Test a bit and change — BCHG

Operation: [Z] ← ~ (<bit number> of [destination])
<bit number> of [destination] ← ~ (<bit number> of [destination])

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

Attributes: Size = byte, longword
Note: A destination of Dn is longword only; all other destinations are byte only.

Description: The source operand specifies the bit number to be tested and changed and can be specified by an immediate data value or by the contents of a data register. Bits are numbered so that bit 0 refers to the least-significant bit. The specified bit in the destination operand is tested and the state of that bit, i.e. zero or not zero, is reflected in the zero flag in the condition code register. After the test operation, the state of the specified bit is changed in the destination, i.e. the bit is toggled.

If a data register is the destination, then the bit numbering is modulo 32, allowing bit manipulation of all bits in the data register. If a memory location is the destination, a byte is read from that location, the bit operation is performed using the bit number modulo 8, and the byte is written back to the memory location.

Condition codes:

X	N	Z	V	C
–	–	*	–	–

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: Used for toggling bits and the state of Z may be irrelevant to the application. For example,

```
BCHG.L    D0,D4
```

Assume D0 = 4
and D4 = $00001234 = %000...0001001000110100,

First bit 4 of D4 is tested. It is a 1 and therefore Z is set to 0.
Then, since bit 4 of D4 is 1, it is changed to 0.

Result is D4 = $00001224 = %000...0001001000100100,

BCLR Test a bit and clear BCLR

Operation: [Z] ← ~ (<bit number> of [destination])
<bit number> of [destination] ← 0

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

Attributes: Size = byte, longword
Note: A destination of Dn is longword only; all other destinations are byte only.

Description: The source operand specifies the bit number to be tested and cleared and can be specified by an immediate data value or by the contents of a data register. Bits are numbered so that bit 0 refers to the least-significant bit. The specified bit in the destination operand is tested and the state of that bit, i.e. zero or not zero, is reflected in the zero flag in the condition code register. After the test operation, the specified bit is cleared, i.e. set to zero.

If a data register is the destination, then the bit numbering is modulo 32, allowing bit manipulation of all bits in the data register. If a memory location is the destination, a byte is read from that location, the bit operation is performed using the bit number modulo 8, and the byte is written back to the memory location.

Condition codes:

X	N	Z	V	C
–	–	*	–	–

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: Used for clearing bits and the state of Z may be irrelevant to the application. For example,

```
BCLR.L    D0,D4
```

Assume D0 = $4
and D4 = $00001234 = %000...0001001000110100,

First bit 4 of D4 is tested. It is a 1 and therefore Z is set to 0.
Then, bit 4 of D4 is cleared, i.e. set to zero.

Result is D4 = $00001224 = %000...0001001000100100,

BRA Branch always BRA

Operation: [PC] ← [PC] + d

Syntax: BRA <label>

Attributes: Size = byte, word

Description: A BRA is an unconditional jump to a label. The label is specified in the machine code as a displacement, d, relative to the current value of the program counter. The displacement is stored in two's complement; the original [PC] contains the sign-extended instruction location plus two.

The range of the branch is -126 to +128 bytes with an 8-bit displacement (i.e. BRA.B), and -32K to +32K bytes with a 16-bit displacement (i.e. BRA.W). Due to the machine code structure for the instruction, a BRA.B to the next instruction not possible.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Application: For unconditional jumps, BRA is used instead of JMP to write position independent code. Typically, the BRA is used without a length specifier (defaults to word). If tight efficient machine code is required, a BRA.B would be used after the code is fully debugged and the displacement is known to be within range.

Conditional branches are used for looping and decision structures. Unconditional branches are used to transfer control, typical around some other segment of code. For example, compare two strings of the same length. Execution of 'Loop' continues until characters do not match or end of string reached.

```
        LEA     STRING1,A0          ; point to start of first string
        LEA     STRING2,A1          ; and the second string
        MOVE.W  #STRING_LEN,D0      ; initialize string counter
Loop    CMPM.B  (A0)+,(A1)+         ; compare pair of characters
        BNE     NotSame             ; exit loop if not match
        SUBQ.W  #1,D0               ; decrement counter
        BNE     Loop                ; loop until end of strings
Same    ....                   ; at this point strings matched
        ....                   ; code to handle matching strings
        ....
        BRA     Cont           ; branch around the code for no match case
NotSame ....                   ; start of code to handle mismatched strings
        ....
        ....                   ; code to handle mismatched strings ends
Cont    ....                   ; do this code after handling the strings
        ....
```

BSET Test a bit and set BSET

Operation: [Z] ← ~ (<bit number> of [destination])
<bit number> of [destination] ← 1

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

Attributes: Size = byte, longword
Note: A destination of Dn is longword only; all other destinations are byte only.

Description: The source operand specifies the bit number to be tested and set and can be specified by an immediate data value or by the contents of a data register. Bits are numbered so that bit 0 refers to the least-significant bit. The specified bit in the destination operand is tested and the state of that bit, i.e. zero or not zero, is reflected in the zero flag of the condition code register. After the test operation, the specified bit is set, i.e. set to 1.

If a data register is the destination, then the bit numbering is modulo 32, allowing bit manipulation of all bits in the data register. If a memory location is the destination, a byte is read from that location, the bit operation is performed using the bit number modulo 8, and the byte is written back to the memory location.

Condition codes:

X	N	Z	V	C
–	–	*	–	–

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: Used for setting bits and the state of Z may be irrelevant to the application. For example,

```
BSET.L    D0,D4
```

Assume D0 = $4
and D4 = $00001234 = %000...0001001000110100,
First bit 4 of D4 is tested. It is a 1 and therefore Z is set to 0.
Then bit 4 of D4 is set and result is D4 = $00001234.
(Since bit 4 was already set, there is no change in the value of D4.)

BSR Branch to subroutine BSR

Operation: [SP] ← [SP] – 4 ; decrement the stack pointer
[[SP]] ← [PC] ; push return address on stack
[PC] ← [PC] + d ; put SR address into PC

Syntax: BSR <label>

Attributes: Size = byte, word

Description: A BSR is a jump to a subroutine that starts a location specified by label. First, the longword address of the instruction immediately following the BSR instruction is pushed onto the system stack. Program execution then continues at location label.

The label of the subroutine is stored in machine code as a displacement relative to the current value of the program counter. The displacement is stored in two's complement; the original [PC] contains the sign-extended instruction location plus two.

The range of the branch is -126 to +128 bytes with an 8-bit displacement (i.e. BSR.B), and -32K to +32K bytes with a 16-bit displacement (i.e. BSR.W). Due to the machine code structure for the instruction, a BSR.B to the next instruction not possible.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Application: For subroutine calls, BSR is used instead of JSR to write position independent code. Typically, the BSR is used without a length specifier (defaults to word). If tight efficient machine code is required, a BSR.B would be used after the code is fully debugged and the displacement is known to be within range.

```
              ....
              BSR    SendChar        ; send character
              ...                    ; return here when routine done
              ...
              ...
              STOP  #$2700

SendChar      ....                   ; code for the subroutine
              ...
              ...
              RTS                    ; return from subroutine
```

BTST — Test a bit — BTST

Operation: [Z] ← ~ (<bit number> of [destination])

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

Attributes: Size = byte, longword
Note: A destination of Dn is longword only; all other destinations are byte only.

Description: The source operand specifies the bit number to be tested and is specified by an immediate data value or by the contents of a data register. Bits are numbered so that bit 0 refers to the least-significant bit. The specified bit in the destination operand is tested and the state of that bit, i.e. zero or not zero, is reflected in the zero flag in the condition code register.

If a data register is the destination, then the bit numbering is modulo 32, allowing testing of all bits in the data register. If a memory location is the destination, a byte is read from that location and the test performed using the bit number modulo 8.

Condition codes:

X	N	Z	V	C
–	–	*	–	–

Destination effective address (for BTST Dn,<ea>) **:**

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Destination effective address (for BTST #<data>,<ea>) **:**

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔		✔	✔

Application: Used for testing the state of a bit. For example, consider BTST.L D0,D4
If D0 = 4 and D4 = $00001234 = %000...0001001000110100,
then bit 4 of D4 is tested. It is a 1 and therefore Z is set to 0.

It is *highly unusual* to have immediate data as a destination. BTST can be used to check for membership in a small set represented by a bit mask. For example, if the set is {1,2,5}, the mask is 00100110 = $26. Test for membership with

```
BTST.B    D4,#$26    ; Does D4 have one of bit {1,2,5} on?
```

CHK — Check register against bounds — CHK

Operation: If [Dn] < 0 or [Dn] > [source]
then issue CHK exception

Syntax: CHK <ea>, Dn

Attributes: Size = word

Description: The contents of the low word in the destination data register are examined and compared with the upper bound at the source effective address. The upper bound is a two's complement integer. If the data register value is less than zero or greater than the upper bound, then the processor initiates exception processing and generates a vector number for the CHK instruction exception vector.

Condition codes:

X	N	Z	V	C
-	*	U	U	U

N is set if [Dn] < 0; cleared if [Dn] > [source]; undefined otherwise.

Source effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Application: The CHK instruction is typically used to test that an array element is within the bounds of the array. If the low end of the actual array bound is not zero, the index is typically shifted to use the test.

For example, if the array is bounded by 0 and #max_bound

```
MOVE.W    INDEX,D1        ; get element index
CHK       #max_bound,D1   ; and trap if not within bounds
                          ; else continue
```

CLR **Clear an operand** **CLR**

Operation: [destination] ← 0

Syntax: CLR <ea>

Attributes: Size = byte, word, longword

Description: The destination is cleared, i.e. zero is moved to the destination location.

Condition codes:

X	N	Z	V	C
-	0	1	0	0

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: For example, to clear an 8 bit memory location,

```
CLR.B    RESULT    ; clear answer space
```

To clear an address register, use subtract or move.

CMP **Compare** **CMP**

Operation: [destination] – [source]

Syntax: CMP <ea>, Dn

Attributes: Size = byte, word, longword

Description: Subtract the source operand from the destination operand and set the condition codes according to the result.

The destination is not modified by this instruction, only flags are set.

Condition codes:

X	N	Z	V	C
–	*	*	*	*

Source effective address:

Dn	An	(An)	(An)+	–(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Application: For example, to test if a register contains the letter “A”

```
        CMP.B    #'A',D0    ; letter A?
        BNE      NOTA       ; no, go do ...
        ADDI.B   #$20,D0    ; yes, convert to lower case
        ....
NOTA    ....                ; stuff to do when not letter A
```

CMPA — Compare address — CMPA

Operation: [destination] – [source]

Syntax: CMPA <ea>, An

Attributes: Size = word, longword

Description: Subtract the source operand from the destination address register and set the condition codes according to the result. The destination address register is not modified, only flags are set. The comparison is always done on 32 bits. Word length source operands (CMPA.W) are sign-extended to 32 bits before the comparison.

Condition codes:

X	N	Z	V	C
–	*	*	*	*

Source effective address:

Dn	An	(An)	(An)+	–(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Application: For example, to test if the address register is equal to the contents of D2. Note, from the examples below, that you normally will be using a longword comparison.

```
        CMPA.W     D2,A0  ; test address
```

If D2 = $00008206
and A4 = $000010F0
sign-extend low word of D2 => $FFFF8206 Note: D2 not modified
and then compare => Z = 0 (not equal)

If D2 = $00008206
and A4 = $FFFF8206
sign-extend low word of D2 => $FFFF8206 Note: D2 not modified
and then compare => Z = 1 (equal)

```
        CMPA.L     D2,A0  ; test address
```

If D2 = $00008206
and A4 = $FFFF8206
then compare => Z = 0 (not equal); N = 1

Carefully, reread the last two cases.

CMPI **Compare immediate** **CMPI**

Operation: [destination] – <immediate data>

Syntax: CMPI #<data>, <ea>

Attributes: Size = byte, word, longword

Description: Subtract the immediate data from the destination operand and set the condition codes according to the result. The destination is not modified. The immediate data is assumed to be the operation size, i.e. there is no sign extension if the data as specified is shorter than the operation size.

Condition codes:

X	N	Z	V	C
_	*	*	*	*

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: For example, to test if the low byte of the data register is equal to zero

```
CMPI.B      #0,D0  ; test count index
```

The low byte of D0 is compared to the immediate data and appropriate flags set.

If D0 = $00000000, then NZVC = 0100

If D0 = $00000012, then NZVC = 0000

If D0 = $0000FF00, then NZVC = 0100

If D0 = $000000FF, then NZVC = 1000

CMPM Compare memory with memory CMPM

Operation: [destination] – [source]

Syntax: CMPM (Ay)+, (Ax)+

Attributes: Size = byte, word, longword

Description: Subtract the source operand from the destination operand and set the condition codes according to the result. The destination is not modified.

Condition codes:

X	N	Z	V	C
–	*	*	*	*

Application: Used to compare the contents of two blocks of memory. For example, to compare two strings

```
            LEA       STRING1,A0        ; point to start of first string
            LEA       STRING2,A1        ; and the second string
            MOVE.W    #STRING_LEN,D0    ; initialize string counter
Loop        CMPM.B    (A0)+,(A1)+       ; compare pair of characters
            BNE       NoMatch           ; branch if strings don't match
            SUBI.W    #1,D0             ; decrement string count
            BNE       Loop              ; loop til done
            ....                        ; some other code here
            ....
NoMatch     ....                        ; handle the no match case
```

DBcc **Test condition, decrement, and branch** **DBcc**

Operation: If (condition false) ; test the condition
then $[Dn] \leftarrow [Dn] - 1$; if false, then decrement counter
If $[Dn] \neq -1$; if counter ≠ -1 then
then $[PC] \leftarrow [PC] + d$; branch to label

Syntax: DBcc Dn, <label>

Attributes: Size = word

Description: The DBcc instruction is a looping primitive that stops when
(1) a condition is met or
(2) the required number of loops have been executed.

Three parameters are required by the DBcc instruction:
(1) a termination condition (specified by 'cc'),
(2) a data register that serves as the loop counter, and
(3) a label that indicates the start of the loop.

The DBcc first tests the termination condition “cc”, and if true, the loop is terminated, i.e. execution continues with the instruction following the DBcc. If the termination condition is not met, the loop counter in Dn.W is decremented by one. If the result is -1, looping is complete and execution continues with the instruction following the Dbcc. If the result is not equal to -1, execution continues at <label>, i.e. [PC] + d where d is the displacement in 2's complement and [PC] is the current location plus two.

Because the test for the required number of loops, tests for -1 and not 0, you must initialize the loop counter to one less than the actual count, e.g. if you want to loop 5 times, the loop counter must be initialized to 4.

condition code (cc)	interpretation	how condition code is calculated
CC / HS	carry clear / high or same	~C
CS / LO	carry set / low	C
NE	not equal	~Z
EQ	equal	Z
VC	overflow clear	~V
VS	overflow set	V
PL	plus	~N
MI	minus	N
GE	greater or equal	(N and V) or (~N and ~V)
LT	less than	(N and ~V) or (~N and V)
GT	greater than	(N and V and ~Z) or (~N and ~V and ~Z)
LE	less or equal	Z or (N and ~V) or (~N and V)
HI	high	~C and ~Z
LS	low or same	C or Z
T	always true	1
F **	never true	0

** Note that many assemblers permit the mnemonic DBF to be expressed as DBRA (i.e. decrement and branch always) when there is no termination condition.

Condition codes: X N Z V C
– – – – –

Application: DBcc allows a loop to terminate when a count condition is met or when some other condition is met. For example, compare two strings. Execution of 'Loop' continues until characters do not match or until there are no more characters. How do you tell which condition caused the exit of the loop? Look at the value of the counter.

```
        LEA      STRING1,A0          ; point to start of first string
        LEA      STRING2,A1          ; and the second string
        MOVE.W   #STRING_LEN-1,D0    ; initialize string counter
Loop    CMPM.B   (A0)+,(A1)+         ; compare pair of characters
        DBNE     D0,Loop             ; loop til not equal or end string
```

DIVS — Signed divide — DIVS

Operation: [$destination_{16:16}$] $\leftarrow$ [$destination_{32}$] / [$source_{16}$]

Syntax: DIVS <ea>, Dn

Attributes: Size = word
Note: <32 bit dividend> / <16 bit divisor> = <16 bit remainder : 16 bit quotient>

Description: Divide the destination operand by the source operand and store the result in the destination location. The destination is a longword and the source is a word. The result in the destination register is a 32-bit value arranged so that the quotient is the lower word and the remainder is the upper word. DIVS performs division assuming that the data is signed, i.e. in two's complement. The sign of the remainder is always the same as the sign of the dividend (unless the remainder is zero).

Attempting to divide a number by zero results in a divide-by-zero exception and exception processing is initiated. If an overflow is detected during division, the operands are unaffected. Overflow occurs if the quotient is larger than a 16-bit signed integer, i.e. if the upper word of the dividend is greater than or equal to the divisor.

Condition codes:

X	N	Z	V	C
-	*	*	*	0

N is set if the quotient is negative.
Z is set if the quotient is zero.
V is set if division overflow occurs (in which case, Z and N are undefined).

Source effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Application: For example, divide the contents of a data register by 2 using signed arithmetic

```
MOVE.L      #5,D0       ; initialize D0 to 5
DIVS        #2,D0       ; divide by 2
```

D0 divided by 2 will result in => $00010002
the quotient in D0 <15:0> => $0002
the remainder in D0 <31:16> => $0001

Note: In general, for a division by powers of 2, a shift or rotate, would be much more efficient. Also, division of large numbers by small powers of 2 may not be doable by a DIVS due to overflow problems.

DIVU **Unsigned divide** **DIVU**

Operation: [$destination_{16:16}$] ← [$destination_{32}$] / [$source_{16}$]

Syntax: DIVU <ea>, Dn

Attributes: Size = word
Note: <32 bit dividend> / <16 bit divisor> = <16 bit remainder : 16 bit quotient>

Description: Divide the destination operand by the source operand and store the result in the destination location. The destination is a longword and the source is a word. The result in the destination register is a 32-bit value arranged so that the quotient is the lower word and the remainder is the upper word. DIVU performs division assuming unsigned arithmetic.

Attempting to divide a number by zero results in a divide-by-zero exception and exception processing is initiated. If an overflow is detected during division, the operands are unaffected. Overflow occurs if the quotient is larger than a 16-bit signed integer, i.e. if the upper word of the dividend is greater than or equal to the divisor.

Condition codes:

X	N	Z	V	C
-	*	*	*	0

N is set if the most significant bit of the quotient is set.
Z is set if the quotient is zero.
V is set if division overflow occurs (in which case, Z and N are undefined).

Source effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Application: For example, divide the contents of a data register by 4 using unsigned arithmetic

```
MOVE.L    #115,D0    ; initialize D0 to 115
DIVU      #4,D0      ; divide by 4
```

D0 divided by 4 will result in => $0003001C
the quotient in D0 <15:0> => $001C
the remainder in D0 <31:16> => $0003

Note: In general, for a division by powers of 2, a shift or rotate, would be much more efficient. Also, division of large numbers by small powers of 2 may not be doable by a DIVU due to overflow problems.

EOR — Exclusive OR logical — EOR

Operation: [destination] ← [source] XOR [destination]

Syntax: EOR Dn, <ea>

Attributes: Size = byte, word, longword

Description: Exclusive OR the source data register contents with the destination operand and store the result in the destination location.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: The exclusive OR is used to toggle a bit(s). For example, to toggle the bit <7> in data register D1

```
MOVE.B    #%10000000,D0    ; set the bit mask
EOR.B     D0,D1            ; and toggle bit <7>
```

EORI — Exclusive OR immediate — EORI

Operation: [destination] ← <immediate data> XOR [destination]

Syntax: EORI #<data>, <ea>

Attributes: Size = byte, word, longword

Description: Exclusively OR the immediate data with the contents of the destination operand, and store the result in the destination location.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: The exclusive OR is used to toggle a bit(s). For example, to toggle the bit <7> in data register D1

```
EORI.B      #$80,D1      ; toggle bit <7>
```

EORI to CCR EOR immediate to condition code register EORI to CCR

Operation: [CCR] ← <immediate data> XOR [CCR]

Syntax: EORI #<data>, CCR

Attributes: Size = byte

Description: Exclusively OR the immediate data with the contents of the condition code register and store the results in the condition code register.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Application: The exclusive OR is used to toggle a bit(s). For example, to toggle the carry and extend bits in the condition code register

```
EORI.B      #%00010001,CCR     ; toggle X and C
```

EORI to SR   EOR immediate to status register (privileged)   EORI to SR

Operation: If supervisor state
then [SR] ← <immediate data> XOR [SR]
else privilege violation

Syntax: EORI #<data>, SR

Attributes: Size = word

Description: Exclusive OR the immediate data with the contents of the status register and store the result in the status register. All bits of the status register are affected.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Application: The system half of the status register contains the interrupt mask, the supervisor bit and the trace bit. The user half of the status register contains the condition codes. For example, to toggle the trace bit of the status register

```
EORI.W    #$8000,SR    ; toogle trace
```

EXG Exchange registers EXG

Operation: [Rx] ↔ [Ry]

Syntax: EXG Rx, Ry

Attributes: Size = longword

Description: Exchange the contents of two registers; the entire 32-bit contents of the two registers are exchanged.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Source effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔	✔										

Destination effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔	✔										

Application: Allows data values to be exchanged without using a temporary register. For the following example, assume that all the registers are in use. Rather than saving D0 to memory, EXG can be used to do an operation on addresses that cannot be done directly on address registers.

```
EXG      D0,A0      ; exchange to allow
ANDI.B   #$0F,D0    ; low byte of A0 to be masked out
EXG      D0,A0      ; without losing contents of D0
```

EXT — Sign extend — EXT

Operation: [destination] ← sign-extended [destination]

Syntax: EXT Dn

Attributes: Size = word, longword

Description: Extend the sign bit of the data register from a byte to a word (.W) or from a word to a longword (.L). If the operation size is word, bit <7> of the data register is copied to bits <15:8>. If the operation size is longword, bit <15> of the data register is copied to bits <31:16>. To sign extend a byte to a longword, EXT must be invoked twice.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Application: Sign extension is used to maintain the sign of a value when increasing the value's size from byte to word or word to longword.

If [DO] = $01234567,

```
        EXT.L       D0      results in $00004567
```

If [DO] = $00008567,

```
        EXT.L       D0      results in $FFFF8567
```

Note that two consecutive sign extensions are required to go from byte to longword. For example,

```
        EXT.W       D0      ; sign extend to word
        EXT.L       D0      ; and then to long
```

If [DO] = $01234567,

```
        EXT.W       D0      results in $01230067
```

and then

```
        EXT.L       D0      results in $00000067
```

If [DO] = $00000087,

```
        EXT.W       D0      results in $0000FF87
```

and then

```
        EXT.L       D0      results in $FFFFFF67
```

ILLEGAL — Take illegal instruction trap — ILLEGAL

Operation:

[SSP] ← [SSP] – 4	; decrement supervisor SP
[[SSP]] ← [PC]	; push PC on supervisor stack
[SSP] ← [SSP] – 2	; decrement supervisor SP
[[SSP]] ← [SR]	; push SR on supervisor stack
[PC] ← Illegal instruction vector address	; go the illegal instruction trap

Syntax: ILLEGAL

Attributes: None

Description: The bit pattern of the illegal instruction, $4AFC_{16}$, causes the illegal instruction trap to be taken. Instruction's sole use is to enable testing of the illegal instruction trap handler code.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Application: The original 68000 had a number of unused instruction bit patterns. These bit patterns were reserved for future extensions of the instruction set. Any pattern of bits read during an instruction read phase that does not correspond to a known instruction causes an illegal instruction trap. The bit pattern corresponding to the ILLEGAL instruction, $4AFC, will always be unused to allow testing of the illegal instruction trap.

JMP — Jump (unconditionally) — JMP

Operation: [PC] ← destination address

Syntax: JMP <ea>

Attributes: Unsized

Description: Program execution continues at the effective address specified by the instruction.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔			✔	✔	✔	✔		✔	✔

Application: JMP allows a simple unconditional jump to an address fixed at compile time, i.e. JMP label. This format acts like an unconditional branch.

The complex addressing modes available for the JMP also allow for the construction of jump tables where jump destinations are dynamically calculated. For example, JMP d(An,Rn) would imply a table of destination addresses located at address An indexed by Rn with displacement d. For example, given

```
JMP     0(A0,D0.W)
```

D0, as the index, would be dynamically changing to access the correct entry in the jump table located at A0.

JSR Jump to subroutine JSR

Operation: [SSP] ← [SSP] − 4 ; decrement system SP
[[SSP]] ← [PC] ; save return address on stack
[PC] ← destination address ; go to subroutine

Syntax: JSR <ea>

Attributes: Unsized

Description: The longword address of the instruction immediately following the JSR is pushed onto the system stack. Program execution then continues at the effective address specified in the instruction.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Destination effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔			✔	✔	✔	✔		✔	✔

Application: JSR allows a simple subroutine call for an address fixed at compile time (i.e. JSR label). In this mode, the JSR behaves like a BSR.

The complex addressing modes available for the JSR also allow for the construction of subroutine jump tables where subroutine addresses are dynamically calculated. For example, JSR d(An,Rn) would imply a table of subroutine addresses located at An indexed by Rn with displacement d. For example,

```
JSR     0(A0,D0.W)
```

D0, as the index, would be dynamically changing to access the correct entry in the subroutine jump table located at A0.

LEA Load effective address LEA

Operation: [An] ← <ea>

Syntax: LEA <ea>, An

Attributes: Size = longword

Description: The effective address is loaded into the address register. Note that it is the address *not* the contents at that address that are moved into the address register.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Source effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔			✔	✔	✔	✔		✔	✔

Application: Used to initialize address registers. Note that it is the address *not* the contents at that address that are moved into the address register. For example, to compare two strings

```
        LEA     STRING1,A0      ; point to start of first string
        LEA     STRING2,A1      ; and the second string
        MOVE.W  #STRING_LEN,D0  ; initialize string counter
Loop    CMPM.B  (A0)+,(A1)+     ; compare pair of characters
        BNE     NoMatch         ; branch if strings don't match
        SUBI.W  #1,D0           ; decrement string count
        BNE     Loop            ; loop til done
        ....
NoMatch ....
```

LEA can be used to write position independent code. For example,

```
        LEA     STRING1(PC),A0
```

calculates the effective address of 'STRING1' with respect to the PC and puts it in A0.

Operation:

[SP]	$\leftarrow$ [SP] − 4	; decrement SP
[[SP]]	$\leftarrow$ [An]	; push stack frame pointer contents on stack
[An]	$\leftarrow$ [SP]	; set current stack frame pointer
[SP]	$\leftarrow$ [SP] + d	; allocate local workspace for procedure

Syntax: LINK An, #<displacement$_{16}$>

Attributes: Size = Unsized

Description: Update the stack pointer to point to the location where the old frame pointer will be stored. The contents of the specified address register, typically the old frame pointer, are pushed onto the stack. Then load the address register with the updated stack pointer. The address register is now the frame pointer and is used to reference the base of the stack frame. The convention is to use A6 for the frame pointer. Finally, the 16-bit sign-extended displacement is added to the stack pointer and the stack pointer points to the top of the stack frame. Since the stack area grows downward in memory, i.e. from a higher memory address to a lower memory address, the displacement must be negative to allocate a local workspace space on the stack for a procedure. The command will accept a positive displacement (unfortunately).

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Application: The LINK and UNLK instructions are used to allocate local workspace on a procedure's stack and then deallocate the space when the procedure is done. The convention is to use A6 as the stack frame pointer. To allocate the local workspace correctly, the displacement must be negative as the stack grows downward in memory. For example,

```
FindChar    LINK   A6,#-8       ; allocate 8 byte stack frame
            ...
            ...
            UNLK   A6           ; deallocate stack frame
            RTS                 ; and return
```

LSL, LSR — Logical shift left/right — LSL, LSR

Operation: [destination] ← [destination] shifted in direction *d* by <count>

Syntax:

LS*d*	#<data>, Dy	where	#<data> is the count
LS*d*	Dx, Dy		Dx contains the count
LS*d*	< ea>		count is 1
		where *d* is the direction, L or R	

Attributes: Size = byte, word, longword
Note: memory locations may be shifted by one bit only and the operand size is restricted to a word.

Description: Logically shift the bits of the operand in the direction (i.e., left or right) specified. The bit shifted out of the operand is copied into the extend and carry bits. A zero is shifted into the low/high bit on a left/right shift. The shift count may be specified in one of three ways:

i. The count may be specified as immediate data for a shift range of 1 to 8 bits.
ii. The count is in a data register, Dx, for a shift range of 0 to 63 bits, i.e. the value is modulo 64.
iii. If no count is specified, the shift is assumed to be one bit.

Condition codes:

X	N	Z	V	C
*	*	*	0	*

X and C are set to the last bit shifted out of the operand. However, a zero shift count clears C but does not affect X.

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔	✔	✔	✔	✔	✔	✔			

Application: Logical shifts can be used to do unsigned multiply and divides by powers of 2. They are also used to isolate a group of bits or to test all the bits in a number consecutively. For example,

```
LSL.B   #4,D0          ; isolate the lowest nibble
LSR.B   #4,D0          ; in the register
```

MOVE — Copy data from source to destination — MOVE

Operation: [destination] ← [source]

Syntax: MOVE <ea>, <ea>

Attributes: Size = byte, word, longword

Description: Copy the contents of the source to the destination location. The data is examined as it is moved and the condition codes set.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Source effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: For example, to toggle the bit <7> in data register D1, the move is used to initialize the mask register D0

```
MOVE.B    #%10000000,D0    ; set the bit mask
EOR.B     D0,D1            ; and toggle bit <7>
```

MOVEA — Copy address — MOVEA

Operation: [An] ← [source]

Syntax: MOVEA <ea>, An

Attributes: Size = word, longword

Description: Copy the contents of the source to the destination address register. Word source operands are sign extended to 32 bits before the move.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Source effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Application: Used to initialize address registers. Function is similar to LEA but MOVEA has a larger range of addressing modes. The intent of LEA is clearer for some applications.

NOTE: syntax for LEA and MOVEA are not interchangeable as the base operation is different. For example, to compare two strings (from the LEA example)

```
         MOVEA.L   #STRING1,A0       ; point to start of first string
         MOVEA.L   #STRING2,A1       ; and the second string
         MOVE.W    #STRING_LEN,D0    ; initialize string counter
Loop     CMPM.B    (A0)+,(A1)+       ; compare pair of characters
         BNE       NoMatch           ; branch if strings don't match
         SUBI.W    #1,D0             ; decrement string count
         BNE       Loop              ; loop til done
         ....
         ....
NoMatch  ....
```

MOVE to CCR — Copy data to the condition code register — MOVE to CCR

Operation: [CCR] ← [source]

Syntax: MOVE <ea>, CCR

Attributes: Size = word

Description: Copy the contents of the *low byte* of the source operand to the condition code register. Although the instruction size is word, only the low-order byte of the source is moved.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Source effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Application: Used to initialize the condition code register. For example,

```
MOVE.W      #0,CCR      ; clear condition codes
```

MOVE from SR   Copy data from status register   MOVE from SR

Operation: [destination] ← [SR]

Syntax: MOVE SR, <ea>

Attributes: Size = word

Description: Copy the contents of the status register to the destination location.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Destination effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: Used to save or investigate the status register. For example,

```
MOVE    SR,-(SP)    ; push SR onto stack
```

MOVE to SR **Copy data to status register (privileged)** **MOVE to SR**

Operation: If supervisor state
then [SR] ← [source]
else privilege violation

Syntax: MOVE <ea>, SR

Attributes: Size = word

Description: Copy the contents of the source operand to the status register.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Source effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Application: The system half of the status register contains the interrupt mask, the supervisor bit and the trace bit. The user half of the status register contains the condition codes. The MOVE to SR instruction allows the programmer to initialize or set the contents of the status register. For example, to clear the condition codes, set the trace bit, clear the supervisor bit, and set the interrupt mask level to 7,

```
MOVE.W      #%1000011100000000,SR
```

MOVE USP — Copy to/from user stack pointer (privileged) — MOVE USP

Operation: If supervisor state
then [destination] ← [source]
else privilege violation

Syntax:
MOVE USP, An
MOVE An, USP

Attributes: Size = longword

Description: Copy the contents of the user stack pointer to an address register or the contents of an address register to the user stack pointer.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Application: Allows the application running in the supervisor mode, e.g. the operating system, to read the contents of the user stack pointer or to intialize the user stack pointer. For example, to switch between different users' tasks, the operating system would have to save and then restore the complete context of the user's system state at the time of the switch.

MOVEM **Move multiple registers** **MOVEM**

Operation: [destination] ← [source]

Syntax:
MOVEM <ea>,<register list>
MOVEM <register list>,<ea>
MOVEM <register list>, -(An) *is a special case*

Attributes: Size = word, longword
Note that either a word or a longword can be moved to memory. When word data is moved from memory to a register, the data is sign-extended to a longword before saving to the destination register.

Description: The contents of a group of registers specified by <register list> are copied to or loaded from consecutive memory locations starting at the effective address location. Any combination of the address and data registers can be copied by a single MOVEM instruction.

The <register list> is specified using
"-" to indicate a range of registers, e.g. D0-D3 = {D0, D1, D2, D3}, and
"/" to indicate a list of registers, e.g. D0/A0/A1 = D0/A0-A1 = {D0, A0, A1}.

The registers can be listed in any order with any number of "-" and/or "/" delimiters, e.g. D0/A5-A7/D1/D2-D3/A1-A2. The MOVEM instruction does not move the registers in the order specified in the register list. Consequently, this register list could also have been written as D0-D3/A1/A2/A5-A7 or A7/D0/A5-A6/A1-A2/D1-D3 or in any other equivalent permutation of these registers.

When copying a group of registers to or from memory the order of transfer is data registers D0 to D7, followed by address registers A0 to A7. Therefore, the previously given example, D0/A5-A7/D1/D2-D3/A1-A2, is actually transferred as {D0, D1, D2, D3, A1, A2, A5, A6, A7}. The one very important exception is that register to memory operations using predecrement addressing, e.g. MOVEM <register list>, -(An) , the order of transfer is address registers A7 down to A0 followed by data registers D7 down to D0.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Source effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔	✔		✔	✔	✔	✔			

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔		✔	✔	✔	✔	✔			

Application: Primary use is to save working registers on entry to a subroutine and to restore them at the end of the subroutine. Saving the registers is also part of saving the context when handling interrupts or switching between user tasks. For example,

```
MoveStr     MOVEM.L     D0-D3/A0-A2,-(SP)    ; save registers
            ...
            ... do stuff
            ...
            MOVEM.L     (SP)+,D0-D3/A0-A2    ; restore registers
            RTS                              ;   and return
```

Note: To minimize errors, use the identical register list for the save/restore. The assembler will sort the registers into the order it wants to store/retrieve the data. To do a store/retrieve and maintain data consistency, use the -(An)/(An)+ addressing modes if placing the data on the stack as the -(An) will save the data in one order and the (An)+ will retrieve it in the opposite order. The programmer is not required to modify/manage the pointers assuming the stack pointer is properly positioned.

MOVEP Move peripheral data MOVEP

Operation: [destination] ← [source]

Syntax:

```
MOVEP    Dx, d(Ay)
MOVEP    d(Ay), Dx
```

Attributes: Size = word, longword

Description: Transfer data between a data register and alternate bytes of memory starting at the location specified and incrementing by two. The data is transferred to/from memory starting at the highest order byte of the data register down to the lowest order byte.

The assumption is that the memory space corresponds to a byte-oriented memory mapped peripheral, i.e. 8-bit peripherals connected to the 68000's 16-bit data bus. If the starting memory address is even, all transfers are made on the high half of the data bus, bits <15:8>. If the starting memory address is odd, all transfers are made on the low half of the data bus, bits <7:0>.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Application: Data transfer is dependent on the instruction size and the starting memory address.
For example, a word transfer to/from an odd address

```
LEA        #$501,A0     ; set odd address
MOVEP.W    D0,0(A0)     ; move a word of data
```

D0.W = $1234 Memory location

500	---
501	$12
502	---
503	$34

For example, a longword transfer to/from an even address

```
LEA        #$500,A0     ; set even address
MOVEP.L    D0,0(A0)     ; move a longword of data
```

D0.L = $12345678 Memory location

500	$12
501	---
502	$34
503	---
504	$56
505	---
506	$78
507	---

MOVEQ — Move quick — MOVEQ

Operation: [destination] ← <immediate data>

Syntax: MOVEQ #<data>, Dn

Attributes: Size = longword

Description: Move the immediate data to a data register. The 8-bit immediate data is sign-extended to 32 bits and all 32 bits are moved to the data register.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Application: Used to load small integers into a data register. Due to the sign extension, results are not the same as the MOVE instruction. For example, compare

```
MOVEQ.L     #$80,D0     ; D0 = $FFFFFF80

MOVE.L      #$80,D0     ; D0 = $00000080
```

MULS **Signed multiply** **MULS**

Operation: [destination] ← [destination] * [source]

Syntax: MULS <ea>, Dn

Attributes: Size = word
Note: <16 bit multiplicand> * <16 bit multiplier> = <32 bit result>

Description: Multiply the 16 bit destination operand by the 16 bit source operand and store the 32 bit result in the destination location. The operation is performed using signed arithmetic.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Source effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Application: For example, multiply the contents of a data register by 2 using signed arithmetic

```
MOVE.W    #$8000,D0    : initialize
MULS      #2,D0        ; multiply by 2
```

D0 <15:0> multiplied by 2 will result in
the product in D0 <31:0>

If D0 = $8000, then a signed multiply by 2 will result in D0 = $FFFF0000

Note: In general, for a multiplication by powers of 2, a shift or rotate, would be much more efficient.

MULU Unsigned multiply MULU

Operation: [destination] ← [destination] * [source]

Syntax: MULU <ea>, Dn

Attributes: Size = word
Note: <16 bit multiplicand> * <16 bit multiplier> = <32 bit result>

Description: Multiply the 16 bit destination operand by the 16 bit source operand and store the 32 bit result in the destination location. The operation is performed using unsigned arithmetic.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Source effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Application: For example, multiply the contents of a data register by 2 using unsigned arithmetic

```
MOVE.W    #$8000,D0    : initialize
MULU      #2,D0        ; multiply by 2
```

D0<15:0> multiplied by 2 will result in
the product in D0 <31:0>

If D0 = $8000, then an unsigned multiply by 2 will result in D0 = $00010000

Note: In general, for a multiplication by powers of 2, a shift or rotate, would be much more efficient.

NBCD — Negate decimal with extend — NBCD

Operation: $[destination]_{10} \leftarrow 0 - [destination]_{10} - [X]$

Syntax: NBCD <ea>

Attributes: Size = byte

Description: Subtract the destination operand and the extend bit from zero and store the result in the destination location. The subtraction is performed using binary coded decimal (BCD) arithmetic.
- If the extend bit is clear (0), the result is the ten's complement of the destination.
- If the extend bit is set (1), the result is the nine's complement of the destination.

Condition codes:

X	N	Z	V	C
*	U	*	U	*

Z is cleared if the result is non-zero *and is unchanged otherwise*. Normally, the programmer sets Z before a series of BCD operations to allow testing for zero result.

Destination effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: The NBCD instruction is used to take the 9's or 10's complement of BCD digits. Consider an 8 decimal digit number, Num1. The negation starts at the least significant digit and proceeds to the highest significant digit. For this example, the result is stored back into Num1.

```
        LEA        Num1+4,A0     ; point past 1st number
        MOVE.W     #4-1,D0       ; initialize digit counter
        MOVE.W     #$04,CCR      ; 10's complement, allow for zero test
Loop    NBCD       -(A0)         ; negate digits
        DBRA       D0,Loop       ; decrement loop counter til done

        ...
Num1    DC.L       $12345678
```

For the above example,
with X = 0 (10's complement), the result in Num1 will be $87654322.
with X = 1 (9's complement), the result in Num1 will be $87654321,

NEG Negate NEG

Operation: [destination] ← 0 − [destination]

Syntax: NEG <ea>

Attributes: Size = byte, word, longword

Description: Subtract the destination operand from 0 and store the result in the destination location.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Destination effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: Negation is done using signed arithmetic and is equivalent to taking the 2's complement of the operand. For example,

```
NEG.B     D0     ; negate
```

If D0.B = $01 then the negation is D0.B = $00 - $01 = $FF = -1.

NEGX Negate with extend NEGX

Operation: [destination] ← 0 − [destination] − [X]

Syntax: NEGX <ea>

Attributes: Size = byte, word, longword

Description: Subtract the destination operand and the extend bit from zero, and store the result in the destination location.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Z is cleared if the result is non-zero *and is unchanged otherwise*. Normally, the programmer sets Z before starting a series of multi-precision operations to allow for zero testing.

Destination effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: Negation is done using signed arithmetic and is equivalent to taking the 2's complement of the operand. NEGX is used for multiple precision arithmetic, e.g. to negate a 64 bit number

```
        LEA       Num+8,A0     ; point past number
        MOVE.W    #$04,CCR     ; clear X and set Z **
        NEGX.L    -(A0)        ; negate digits
        NEGX.L    -(A0)        ; negate digits
        ...
Num     DC.L      $12345678,$76543210
```

Execution of the program will result in Num containing $EDCBA987,$89ABCDF0

** Clear X to prevent a carry-in to first negation; set Z to allow the result to be tested for zero. Test for zero was not done in this code segment.

NOP — No operation — NOP

Operation: None

Syntax: NOP

Attributes: Unsized

Description: The instruction performs no operation. Execution continues with the instruction following the NOP instruction.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Application: Used primarily to insert imprecise delays in code. For example, NOPs may be inserted in a loop to slow down the sending of characters to a display so that (a) they can be seen before they are overwritten or (b) they don't overflow the peripheral's slower input buffer.

NOT Logical complement NOT

Operation: [destination] ← ~ [destination]

Syntax: NOT <ea>

Attributes: Size = byte, word, longword

Description: Take the logical complement of the destination operand and store the result in the destination location.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: Taking the logical complement is equivalent to taking the 1's complement of the operand. For example,

```
NOT.B     D0     ; take complement
```

If D0.B = $01 then the logical complement is D0.B = $FE

OR　　Inclusive OR logical　　OR

Operation: [destination] ← [source] OR [destination]

Syntax:
OR <ea>, Dn
OR Dn, <ea>

Attributes: Size = byte, word, longword

Description: Inclusive OR the source operand to the destination operand, and store the result in the destination location.

Condition codes:

X	N	Z	V	C
–	*	*	0	0

Source effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Destination effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔	✔	✔	✔	✔	✔	✔			

Application: Take the inclusive OR of an operand. For example,

```
OR.B   D1,D0          ; OR low byte of two registers
```

If D1.B = %11001100
and D0.B = %11110000,
then result is
D0.B = %11111100

ORI — Inclusive OR immediate — ORI

Operation: [destination] ← <immediate data> OR [destination]

Syntax: ORI #<data>, <ea>

Attributes: Size = byte, word, longword

Description: Inclusive OR the immediate data with the destination operand, and store the result in the destination location.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: Take the inclusive OR of two operands. ORI allows a greater range of destination addressing modes (as compared to an OR) when the source is immediate data. For example,

```
ORI.B   #%11001100,(A0)    ; OR memory with immediate data
```

If the byte located at memory location (A0) = %11110000,
then the result at that memory location => %11111100.

ORI to CCR   OR immediate to conditional code register   ORI to CCR

Operation: [CCR] ← <immediate data> OR [CCR]

Syntax: ORI #<data>, CCR

Attributes: Size = byte

Description: Inclusive OR the immediate data with the condition code register and store the result in the condition code register.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Application: ORI is used to set selected bits of the CCR. The example for ADDX instruction can be changed to:

```
        LEA      Num1+8,A0    ; point past 1st number
        LEA      Num2+8,A1    ; point past 2nd number
        ANDI.B   #$00,CCR     ; clear X
        ORI.B    #$04,CCR     ; and set Z
        ADDX.L   -(A0),-(A1)  ; add digits
        ADDX.L   -(A0),-(A1)  ; add digits
        ...
Num1    DC.L     $12345678,$87654321
Num2    DC.L     $87654321,$12345678
```

Execution of the program will result in Num2 containing $99999999,$99999999

ORI to SR   Inclusive OR immediate to status register (privileged)   ORI to SR

Operation: If supervisor state
    then [SR] ← <immediate data> OR [SR]
    else privilege violation

Syntax: ORI #<data>, SR

Attributes: Size = word

Description: Inclusive OR the immediate data with the status register, and store the result in the status register. All bits of the status register (SR) are affected.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Application: The system half of the status register contains the interrupt mask, the supervisor bit and the trace bit. The user half of the status register contains the condition codes. For example, to set the trace bit of the status register

```
ORI.W       #$8000,SR     ; set trace
```

PEA — Push effective address — PEA

Operation: [SP] ← [SP] - 4 ; decrement SP
[[SP]] ← <ea> ; push address on stack

Syntax: PEA <ea>

Attributes: Size = longword

Description: The effective address is computed and pushed onto the stack.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔			✔	✔	✔	✔		✔	✔

Application: Used to push an address onto the stack. Note that it is the address *not* the contents at that address that are moved onto the stack. For example,

```
          PEA      STRING1     ; put address of string1 on stack
          ...
STRING1   DC.B     "Hello World"
```

PEA can be used to write position independent code. For example,

```
          PEA   STRING1(PC)
```

calculates the effective address of 'STRING1' with respect to the PC and pushes it on the stack.

RESET — Reset external devices (privileged) — RESET

Operation: If supervisor state
then Assert RESET line
else privilege violation

Syntax: RESET

Attributes: Unsized

Description: The reset line is asserted for 124 clock cycles, causing all external devices connected to the reset pin to be reset. The instruction is used to perform a reset of peripherals under program control.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

ROL, ROR — Rotate left/right (without extend) — ROL, ROR

Operation: [destination] ← [destination] rotated in direction *d* by <count>

Syntax:

RO*d*	#<data>, Dy	where #<data> is the count
RO*d*	Dx, Dy	Dx contains the count
RO*d*	<ea>	count is 1
		where *d* is the direction, L or R

Attributes: Size = byte, word, longword
Note: memory locations may be rotated by one bit only and the operand size is restricted to a word.

Description: Rotate the bits of the operand in the direction, left or right, specified. A rotate is a circular shift where the bit shifted out of the operand is shifted into the other end of the operand. The bit shifted out is also copied to the carry bit. The rotate count may be specified in one of three ways:

i. The count may be specified as immediate data for a rotate range of 1 to 8 bits.
ii. The count is in a data register for a rotate range of 0 to 63 bits, i.e. the value is modulo 64.
iii. If no count is specified, the rotate is assumed to be one bit.

Condition codes:

X	N	Z	V	C
-	*	*	0	*

C is cleared if the shift count is 0.

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔	✔	✔	✔	✔	✔	✔			

Application: For example,

```
MOVE.L    #'HELP',D0    ; initialize the register
ROL.L     #8,D0         ; rotate the contents of D0 by one character
```

the result in D0 is ‘ELPH’

ROXL, ROXR — Rotate left/right through extend — ROXL, ROXR

Operation: [destination] ← [destination]:[X] rotated left by <count> ; rotate left
[destination] ← [X]:[destination] rotated right by <count> ; rotate right

Syntax:
ROX*d* #<data>, Dy where #<data> is the count
ROX*d* Dx, Dy Dx contains the count
ROX*d* <ea> count is 1
where *d* is the direction, L or R

Attributes: Size = byte, word, longword
Note: memory locations may be rotated by one bit only and the operand size is restricted to a word.

Description: Rotate the bits of the operand and the extend bit in the direction, left or right, specified. The rotate is a circular shift where the bit shifted out of the operand is shifted into the extend bit and the content of the extend bit is shifted into the other end of the operand. The bit shifted out is also copied to the carry bit. The rotate count may be specified in one of three ways:

i. The count may be specified as immediate data for a rotate range of 1 to 8 bits.
ii. The count is in a data register for a rotate range of 0 to 63 bits, i.e. the value is modulo 64.
iii. If no count is specified, the rotate is assumed to be one bit.

Condition codes:

X	N	Z	V	C
*	*	*	0	*

C is set to X if the shift count is zero; X is unaffected if the shift count is zero.

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔	✔	✔	✔	✔	✔	✔			

Application: For example,

```
MOVE.L    #'HELP',D0   ; initialize the register
MOVE.W    #0,CCR       ; clear X
ROXL.L    #8,D0        ; rotate the contents of D0 by one character
```

the original contents of D0.L = 'HELP' = $48454C50
after rotation D0.L = $454C5024 with X=0
or D0.L = 'ELP$'
The X=0 bit is moved in before the $48 = %01001000 so the low 8 bits are %00100100 (X:top 7 bits of $48) and the lowest bit of $48 is left in the X bit.

RTE Return from exception (privileged) RTE

Operation:

```
If supervisor state                     ; if supervisor then
    then  [SR] ← [[SP]]                 ;     restore the SR
          [SP] ← [SP] + 2               ;     increment SP
          [PC] ← [[SP]]                 ;     restore the PC
          [SP] ← [SP] + 4               ;     increment SP
    else  privilege violation           ; else privilege violation
```

Syntax: RTE

Attributes: Unsized

Description: The status register and program counter are loaded from the stack and the stack pointer adjusted. The RTE instruction is used to terminate an exception service routine. For later models in the 68000 family, more information is loaded from the stack as all information placed on the stack by the processor for exception handling must be retrieved at the end of exception handling.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The CCR is restored to its pre-exception state.

Application: Used to return from an exception service routine. It is the programmer's responsibility to store any registers that will be modified in the routine so that the user's context is unchanged upon returning from the subroutine. For example,

```
PrtISR    MOVEM.L   ....        ; save context registers
          ....
          .... do stuff
          ...
          MOVEM.L   ...         ; restore context registers
          RTE                   ; and return
```

RTR — Return and restore condition codes — RTR

Operation:

[CCR] ← [[SP]]	; restore CCR
[SP] ← [SP] + 2	; increment SP
[PC] ← [[SP]]	; restore PC
[SP] ← [SP] + 4	; increment SP

Syntax: RTR

Attributes: Unsized

Description: The condition code register and program counter are loaded from the stack and the stack pointer adjusted. The supervisor portion of the status register is not affected. The RTR instruction may be used to terminate a subroutine.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The CCR is restored to its previous state, i.e. pre-subroutine call.

Application: Used to restore the condition codes prior to returning from a subroutine. It is the programmer's responsibility to store the condition codes on the stack so that the RTR can restore them when returning from the subroutine. For example,

```
DispStr     MOVE.W      SR,-(SP)        ; save the condition codes
            MOVEM.L     ....            ; save context registers
            ....
            .... do stuff
            ...
            MOVEM.L     ...             ; restore context registers
            RTR                         ; and return with condition codes
```

Note that this is not a privileged instruction and therefore the system portion of the status register is not affected.

RTS Return from subroutine RTS

Operation: [PC] ← [[SP]] ; restore PC
[SP] ← [SP] + 4 ; increment SP

Syntax: RTS

Attributes: Unsized

Description: The program counter is loaded from the stack and the stack pointer adjusted. The RTS instruction is used to terminate a subroutine.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Application: Used to return from a subroutine. It is the programmer's responsibility to store any registers that will be modified in the routine so that the user's context is unchanged upon returning from the subroutine. For example,

```
DispStr     MOVEM.L    ....          ; save context registers
            ....
            .... do stuff
            ...
            MOVEM.L    ...           ; restore context registers
            RTS                      ; and return
```

SBCD Subtract decimal with extend SBCD

Operation: $[\text{destination}]_{10} \leftarrow [\text{destination}]_{10} - [\text{source}]_{10} - [X]$

Syntax:
SBCD Dy, Dx
SBCD −(Ay), −(Ax)

Attributes: Size = byte

Description: Subtract the source operand and X from the destination operand, and store the result in the destination location. Subtraction is performed using BCD (decimal) arithmetic.

Condition codes:

X	N	Z	V	C
*	U	*	U	*

Z is cleared if result is non-zero *but unchanged otherwise*. Normally, the programmer sets Z before the start of a series of BCD operations to allow for zero testing.

Application: The SBCD instruction is used to subtract BCD digits. Consider the subtraction of Num1 from Num2 where both are eight packed decimal digit numbers. The numbers are stored so that subtraction starts at the least significant digit and proceeds to the highest significant digit. The result of the subtraction is stored back into Num2.

```
        LEA      Num1+4,A0     ; point past 1st number
        LEA      Num2+4,A1     ; point past 2nd number
        MOVE.W   #4-1,D0       ; initialize digit counter
        MOVE.W   #$04,CCR      ; clear X and set Z **
Loop    SBCD     -(A0),-(A1)   ; subtract  digits
        DBRA     D0,Loop       ; decrement loop counter til done

        ...
Num1    DC.L     $12345678
Num2    DC.L     $87784321
```

The program will subtract Num1 from Num2 with the final result $75438643 stored in Num2.

** Clear X to prevent a borrow from first subtraction; set Z to allow the result to be tested for zero. Test for zero was not done in this code segment.

Scc — Set true/false according to condition cc — Scc

Operation: If condition true
then [destination] ← 11111111_2 "True"
else [destination] ← 00000000_2 "False"

Syntax: Scc <ea>

Attributes: Size = byte

Description: The specified condition code, 'cc', is tested. If the condition is true, the bits at the effective address are set to ones (i.e., "TRUE"); otherwise, the bits at the effective address are set to zeros (i.e., "FALSE").

Condition code (cc)	**interpretation**	**how it is calculated**
CC HS*	carry clear high or same	~C
CS LO*	carry set low	C
NE	not equal	~Z
EQ	equal	Z
VC	overflow clear	~V
VS	overflow set	V
PL	plus	~N
MI	minus	N
GE	greater or equal	(N and V) or (~N and ~V)
LT	less than	(N and ~V) or (~N and V)
GT	greater than	(N and V and ~Z) or (~N and ~V and ~Z)
LE	less or equal	Z or (N and ~V) or (~N and V)
HI	high	~C and ~Z
LS	low or same	C or Z
T	always true	1
F	never true	0

* Support for these alternate mnemonics depends on the assembler.

Condition codes: X N Z V C
– – – – –

Destination effective address:

Dn	An	(An)	(An)+	–(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: Used to set flags based on a condition. For example,

```
SUBI.B    #1,D0     ; decrement counter
SEQ       D1        ; if zero, then D1 = true else D1 = false
```

An arithmetic one/zero result may be generated by following the Scc with a NEG instruction.

```
SUBI.B    #1,D0     ; decrement counter
SEQ.B     D1        ; if zero, then D1 = true else D1 = false
NEG.B     D1        ; if zero, then D1 = 1 else D1 = 0
```

STOP Load status register and stop (privileged) STOP

Operation: If supervisor state
then [SR] ← <immediate data>
STOP
else privilege violation

Syntax: STOP #<data>

Attributes: Unsized

Description: The immediate data is moved into the status register, the program counter is advanced to point to the next instruction and the processor stops fetching and executing instructions.

The execution of instructions resumes when a trace, an interrupt, or a reset exception occurs. A trace exception will occur if the trace bit was set when the STOP instruction began execution. If an interrupt request arrives whose priority is higher than the current processor priority as set by the immediate data, an interrupt exception occurs, otherwise the interrupt request has no effect. If the bit of the immediate data corresponding to the supervisor bit is clear (i.e., user mode selected), execution of the STOP instruction will cause a privilege violation. An external reset will always initiate reset exception processing.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The CCR is set according to the immediate data.

Application: Normally, the following stop is issued.

```
STOP     #$2700     ; supervisor, no trace, priority = 7, clear CCR
```

SUB Subtract binary SUB

Operation: [destination] ← [destination] − [source]

Syntax: SUB <ea>, Dn
SUB Dn, <ea>

Attributes: Size = byte, word, longword
Note: If the source is An, size must be word or longword.

Description: Subtract the source operand from the destination operand, and store the result in the destination location.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The condition codes are not affected if a subtraction from an address register is made.

Source effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Destination effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
		✔	✔	✔	✔	✔	✔	✔			

Application: For example, subtract the contents of D2 from the contents of D4 at word size.

```
SUB.W    D2,D4
```

If D4 = $10008206
and D2 = $100010F0,
then result is
D4 = $10007116.

SUBA Subtract address SUBA

Operation: [destination] ← [destination] − [source]

Syntax: SUBA <ea>, An

Attributes: Size = word, longword

Description: Subtract the source operand from the destination operand and store the result in the destination address register. Word source operands are sign extended to 32 bits prior to subtraction.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Source effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Application: Used to subtract values from an address register. For example,

```
SUBA.W    #1,A0        ; decrement address pointer
```

SUBI **Subtract immediate** **SUBI**

Operation: [destination] ← [destination] − <immediate data>

Syntax: SUBI #<data>, <ea>

Attributes: Size = byte, word, longword

Description: Subtract the immediate data from the destination operand, and store the result in the destination location. The immediate data matches the operation size.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Destination effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: Used to subtract constants. For example,

```
SUBI.B    #1,D0       ; decrement counter
```

SUBQ Subtract quick SUBQ

Operation: [destination] ← [destination] − <immediate data>

Syntax: SUBQ #<data>, <ea>

Attributes: Size = byte, word, longword
Note: When destination is An, only word and longword may be used.

Description: Subtract the immediate data from the destination operand and store the result in the destination location. The immediate data must be in the 1 to 8 range. A word operation on an address register affects all 32 bits of the address register.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The condition codes are not affected if a subtraction from an address register is made.

Destination effective address:

Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔	✔	✔	✔	✔	✔	✔	✔	✔			

Application: Used to subtract small constants. SUBQ generates less machine code than SUBI for the comparable subtraction. For example,

```
SUBQ.W    #1,A0        ; decrement address pointer
```

SUBX Subtract with extend SUBX

Operation: [destination] ← [destination] − [source] − [X]

Syntax:
```
SUBX    Dx, Dy
SUBX    -(Ax), -(Ay)
```

Attributes: Size = byte, word, longword

Description: Subtract the source operand and the extend bit from the destination operand, and store the result in the destination location.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Z is cleared if the result is non-zero *but is unchanged otherwise*. Normally, the programmer sets Z before starting a series of multi-precision operations to allow for the testing of zero.

Application: SUBX is used for multiple precision arithmetic, e.g. 64 bit subtraction. For example, subtract number at Num1 from the number at Num2 and store the result in Num2.

```
      LEA       Num1+8,A0     ; point past 1st number
      LEA       Num2+8,A1     ; point past 2nd number
      MOVE.W    #$04,CCR      ; clear X and set Z **
      SUBX.L    -(A0),-(A1)   ; subtract digits
      SUBX.L    -(A0),-(A1)   ; subtract digits
      ...
Num1  DC.L      $12345678,$87654321
Num2  DC.L      $87654321,$12345678
```

Execution of the program will result in Num2 containing $7530ECA8,$8ACF1357

** Clear X to prevent a borrow from first subtraction; set Z to allow the result to be tested for zero. Test for zero was not done in this code segment.

SWAP **Swap register halves** **SWAP**

Operation: [Dn <16:31>] ↔ [Dn <0:15>]

Syntax: SWAP Dn

Attributes: Size = word

Description: Exchange the upper and lower 16-bits of the destination data register.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Application: Used to switch the higher-order word in a register with the lower-order word. Can be used with the MOVEP to put data out to peripherals in a different order, especially useful for initializing LCDs (liquid crystral displays). Can be used with shifts and rotates to isolate substrings.

For example,

```
MOVE.L    #'HELP',D0    ; initialize register
SWAP      D0            ; exchange the words
```

D0.L now contains 'LPHE'.

TAS | Test and set an operand | TAS

Operation: compare [destination] with 0
bit <7> of [destination] ← 1

Syntax: TAS <ea>

Attributes: Size = byte

Description: Test the operand by comparing it with zero and set the negative and zero bits according to the results. Set bit <7> at the destination location. The operation is *indivisible* to allow synchronization between several processors.

Note: Bus error retry is inhibited on the read portion of the TAS read-modify-write bus cycle to ensure system integrity. The bus error exception is always taken.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: The TAS instruction permits one processor in a multiprocessor system to test a resource and claim the resource if it is free. The most-significant bit of the byte at the effective address is used as a semaphore to indicate whether the shared resource is free. The TAS instruction reads the semaphore bit to find the state of the resource. If the resource is free, the TAS sets the semaphore to claim the resource. Because the operation is indivisible, no other processor can access the resource between the testing of the bit and its subsequent setting.

TRAP Trap TRAP

Operation:

[SSP]	← [SSP] – 4	; decrement system SP
[[SSP]]	← [PC]	; push PC on system stack
[SSP]	← [SSP] – 2	; decrement system SP
[[SSP]]	← [SR]	; push SR on system stack
[PC]	← vector address	; initiate TRAP

Syntax: TRAP #<vector>

Attributes: Unsized

Description: The processor initiates TRAP exception processing indexed by the vector number. The operand indicates the vector number in the range #0 to #15.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Application: The TRAP instruction is used to perform operating system calls. The effect of the call depends on how the operating system has mapped the 16 vectors, #0 to #15, to its system calls.

For example, the EASy68K simulator uses Trap #15 for all system calls and expects the task number to be in D0.B. Tasks may require additional parameters to be loaded in other data and address registers.

```
MOVE.B    #9,D0     ; load code for halt simulator task
TRAP      #15       ; and return control to simulator
```

TRAPV — Trap on overflow — TRAPV

Operation: If [V] = 1 then initiate overflow TRAP

Syntax: TRAPV

Attributes: Unsized

Description: If the overflow bit is set, the processor initiates TRAPV exception processing. If the overflow bit is clear, execution continues with the next instruction.

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Application: Used after arithmetic operations to call the operating system if overflow occurs. Allows all overflows to be handled by one system routine.

```
ADD.L     D0,D5     ; add
TRAPV               ; and TRAP if overflow on add
```

TST　　　　Test an operand　　　　TST

Operation: compare [destination] to 0

Syntax: TST <ea>

Attributes: Size = byte, word, longword

Description: Compare the operand with zero and set the contents of the condition code register according to the result.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Destination effective address:

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	(xxx).W	(xxx).L	#<data>	d(PC)	d(PC,Rn)
✔		✔	✔	✔	✔	✔	✔	✔			

Application: Test operand and set condition codes. For example,

```
TST.B    (A0)    ; test byte at location pointed to by A0
```

UNLK **Unlink and deallocate** **UNLK**

Operation: [SP] ← [An] ; collapse local workspace
[An] ← [[SP]] ; restore stack frame pointer contents
[SP] ← [SP] + 4 ; decrement SP

Syntax: UNLK An

Attributes: Unsized

Description: The stack pointer is loaded from the specified address register which (a) sets the stack pointer equivalent to frame pointer and (b) therefore, collapses the local workspace. The address register is then loaded with the longword pulled off the stack thereby restoring the saved register value, typically the old frame pointer.

Typically used in conjunction with the LINK command..

Condition codes:

X	N	Z	V	C
–	–	–	–	–

Application: The LINK and UNLK instructions are used to allocate local workspace on a procedure's stack and then deallocate the space when the procedure is done. The convention is to use A6 as the stack frame pointer. The LINK and UNLK must both use the same register for the frame pointer for consistent allocation and deallocation of the stack frame. For example,

```
FindChar    LINK   A6,#-8      ; allocate 8 byte stack frame
            ...
            ...
            UNLK  A6           ; deallocate stack frame, restore A6
            RTS                ; and return
```

ASCII Character Set

(ASCII = American Standard Code for Information Interchange)

Hex	Value	Comments
00	NUL	Null or tape feed
01	SOH	Start of Header
02	STX	Start of Text
03	ETX	End of Text
04	EOT	End of Transmission
05	ENQ	Enquiry (who are you)
06	ACK	Acknowledgment
07	BEL	Bell
08	BS	Backspace
09	HT	Horizontal Tab
0A	LF	Line Feed
0B	VT	Vertical Tab
0C	FF	Form Feed
0D	CR	Carriage Return
0E	SO	Shift Out (to red ribbon)
0F	SI	Shift In (to black ribbon)
10	DLE	Data Link Escape
11	DC1	XON, Device Control 1
12	DC2	Device Control 2
13	DC3	XOFF, Device Control 3
14	DC4	Device Control 4
15	NAK	Negative Acknowledgement
16	SYN	Synchronous Idle
17	ETB	End of Transmission Block
18	CAN	Cancel
19	EM	End of Medium
1A	SUB	Substitute
1B	ESC	Escape, prefix
1C	FS	File Separator
1D	GS	Group Separator
1E	RS	Request to Send; Record Separator
1F	US	Unit Separator

Hex	Value	Comments
20	SP	Space or Blank
21	!	exclamation mark
22	"	double quote mark
23	#	number sign
24	$	dollar sign
25	%	percent sign
26	&	ampersand
27	'	Apostrophe, closing single quote
28	(	left/opening parenthesis
29	)	right/closing parenthesis
2A	*	asterisk
2B	+	plus sign
2C	,	comma, cedilla
2D	-	Hyphen, minus
2E	.	Period, decimal point
2F	/	forward slash
30	0	digit 0
31	1	digit 1
32	2	digit 2
33	3	digit 3
34	4	digit 4
35	5	digit 5
36	6	digit 6
37	7	digit 7
38	8	digit 8
39	9	digit 9
3A	:	colon
3B	;	semi-colon
3C	<	less than
3D	=	equal sign
3E	>	greater than
3F	?	question mark

Hex	Value	Comments
40	@	AT symbol
41	A	upper-case letter A
42	B	upper-case letter B
43	C	upper-case letter C
44	D	upper-case letter D
45	E	upper-case letter E
46	F	upper-case letter F
47	G	upper-case letter G
48	H	upper-case letter H
49	I	upper-case letter I
4A	J	upper-case letter J
4B	K	upper-case letter K
4C	L	upper-case letter L
4D	M	upper-case letter M
4E	N	upper-case letter N
4F	O	upper-case letter O
50	P	upper-case letter P
51	Q	upper-case letter Q
52	R	upper-case letter R
53	S	upper-case letter S
54	T	upper-case letter T
55	U	upper-case letter U
56	V	upper-case letter V
57	W	upper-case letter W
58	X	upper-case letter X
59	Y	upper-case letter Y
5A	Z	upper-case letter Z
5B	[	left/opening bracket
5C	\	back slash
5D	]	right/closing bracket
5E	^	caret, circumflex
5F	_	underscore

Hex	Value	Comments
60	`	quotation mark
61	a	lower-case letter a
62	b	lower-case letter b
63	c	lower-case letter c
64	d	lower-case letter d
65	e	lower-case letter e
66	f	lower-case letter f
67	g	lower-case letter g
68	h	lower-case letter h
69	i	lower-case letter i
6A	j	lower-case letter j
6B	k	lower-case letter k
6C	l	lower-case letter l
6D	m	lower-case letter m
6E	n	lower-case letter n
6F	o	lower-case letter o
70	p	lower-case letter p
71	q	lower-case letter q
72	r	lower-case letter r
73	s	lower-case letter s
74	t	lower-case letter t
75	u	lower-case letter u
76	v	lower-case letter v
77	w	lower-case letter w
78	x	lower-case letter x
79	y	lower-case letter y
7A	z	lower-case letter z
7B	{	left/opening brace
7C	|	vertical bar
7D	}	right/closing brace
7E	~	tilde, equivalent
7F	DEL	delete