The 68000's Instruction Set

We have included this appendix to save you the task of having to turn to secondary material when writing 68000 assembly language programs. Since most programmers are not interested in the encoding of instructions, details of instruction encoding have been omitted (i.e., the actual op-code bit patterns). Applications of some of the instructions have been provided to demonstrate how they can be used in practice.

Instructions are listed by mnemonic in alphabetical order. The information provided about each instruction is: its assembler syntax, its attributes (i.e., whether it takes a byte, word, or longword operand), its description in words, the effect its execution has on the condition codes, and the addressing modes it may take. The effect of an instruction on the CCR is specified by the following codes:

- `U` The state of the bit is undefined (i.e., its value cannot be predicted)
- `-` The bit remains unchanged by the execution of the instruction
- `*` The bit is set or cleared according to the outcome of the instruction.

Unless an addressing mode is implicit (e.g., `NOP`, `RESET`, `RTS`, etc.), the legal source and destination addressing modes are specified by their assembly language syntax. The following notation is used to describe the 68000's instruction set.

`Dn, An`	Data and address register direct.
`(An)`	Address register indirect.
`(An)+, -(An)`	Address register indirect with post-incrementing or pre-decrementing.
`(d,An), (d,An,Xi)`	Address register indirect with displacement, and address register indirect with indexing and a displacement.
`ABS.W, ABS.L`	Absolute addressing with a 16-bit or a 32-bit address.
`(d,PC), (d,PC,Xi)`	Program counter relative addressing with a 16-bit offset, or with an 8-bit offset plus the contents of an index register.
`imm`	An immediate value (i.e., literal) which may be 16 or 32 bits, depending on the instruction.

Two notations are employed for address register indirect addressing. The notation originally used to indicate address register indirect addressing has been superseded. However, the Teesside 68000 simulator supports only the older form.

Old notation	Current notation
d(An), d(An,Xi)	(d,An), (d,An,Xi)
d(PC), d(PC,Xi)	(d,PC), (d,PC,Xi)

ABCD Add decimal with extend

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 BCD arithmetic. The only legal addressing modes are data register direct and memory to memory with address register indirect using pre-decrementing.

Application: The `ABCD` instruction is used in chain arithmetic to add together strings of BCD digits. Consider the addition of two nine-digit numbers. Note that the strings are stored so that the least-significant digit is at the high address.

```
        LEA  Number1,A0     A0 points at first string
        LEA  Number2,A1     A1 points at second string
        MOVE #8,D0          Nine digits to add
        MOVE #$04,CCR       Clear X-bit and Z-bit of the CCR
LOOP    ABCD -(A0),-(A1)    Add a pair of digits
        DBRA D0,LOOP        Repeat until 9 digits added
```

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 Z-bit is normally set by the programmer before the BCD operation, and can be used to test for zero after a chain of multiple-precision operations. The C-bit is set if a decimal carry is generated.

ADD Add binary

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 and store the result in the destination location.

Condition codes:
```
X  N  Z  V  C
*  *  *  *  *
```

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

ADDA Add address

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 source is sign-extended before it is added to the destination. For example, if we execute `ADDA.W D3,A4` where A4 = 00000100_{16} and D3.W = 8002_{16}, the contents of D3 are sign-extended to $FFFF8002_{16}$ and added to 00000100_{16} to give $FFFF8102_{16}$, which is stored in A4.

Application: To add to the contents of an address register and not update the CCR. Note that `ADDA.W D0,A0` is the same as `LEA (A0,D0.W),A0`.

Condition codes:

X	N	Z	V	C
-	-	-	-	-

An `ADDA` operation does not affect the state of the CCR.

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

ADDI Add immediate

Operation: `[destination] ← <literal> + [destination]`

Syntax: `ADDI #<data>,<ea>`

Attributes: Size = byte, word, longword

Description: Add immediate data to the destination operand. Store the result in the destination operand. `ADDI` can be used to add a literal directly to a memory location. For example, `ADDI.W #$1234,$2000` has the effect $[M(2000_{16})] \leftarrow [M(2000_{16})] + 1234_{16}$.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

ADDQ Add quick

Operation: `[destination] ← <literal> + [destination]`

Syntax: `ADDQ #<data>,<ea>`

Sample syntax: `ADDQ #6,D3`

Attributes: Size = byte, word, longword

Description: Add the immediate data to the contents of the destination operand. The immediate data must be in the range 1 to 8. Word and longword operations on address registers do not affect condition codes. Note that a word operation on an address register affects all bits of the register.

Application: `ADDQ` is used to add a small constant to the operand at the effective address. Some assemblers permit you to write `ADD` and then choose `ADDQ` *automatically* if the constant is in the range 1 to 8.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Note that the CCR is not updated if the destination operand is an address register.

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔	✔	✔	✔	✔	✔	✔	✔	✔			

ADDX Add extended

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 data register direct and memory to memory with address register indirect using pre-decrementing.

Application: The `ADDX` instruction is used in chain arithmetic to add together strings of bytes (words or longwords). Consider the addition of

two 128-bit numbers, each of which is stored as four consecutive longwords.

```
        LEA  Number1,A0     A0 points at first number
        LEA  Number2,A1     A1 points at second number
        MOVE #3,D0          Four longwords to add
        MOVE #$00,CCR       Clear X-bit and Z-bit of the CCR
LOOP    ADDX -(A0),-(A1)    Add pair of numbers
        DBRA D0,LOOP        Repeat until all added
```

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The Z-bit is cleared if the result is non-zero, and left unchanged otherwise. The Z-bit can be used to test for zero after a chain of multiple precision operations.

AND AND logical

Operation: `[destination] ← [source].[destination]`

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

Attributes: Size = byte, word, longword

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

Application: `AND` is used to mask bits. If we wish to clear bits 3 to 6 of data register D7, we can execute `AND #%10000111,D7`. Unfortunately, the `AND` operation cannot be used with an address register as either a source or a destination operand. If you wish to perform a logical operation on an address register, you have to copy the address to a data register and then perform the operation there.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

ANDI AND immediate

Operation: `[destination] ← <literal>.[destination]`

Syntax: `ANDI #<data>,<ea>`

Attributes: Size = byte, word, longword

Description: AND the immediate data to the destination operand. The `ANDI` permits a literal operand to be ANDed with a destination other than a data register. For example, `ANDI #$FE00,$1234` or `ANDI.B #$F0,(A2)+`.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

ANDI to CCR AND immediate to condition code register

Operation: `[CCR] ← <data>.[CCR]`

Syntax: `ANDI #<data>,CCR`

Attributes: Size = byte

Description: AND the immediate data to the condition code register (i.e., the least-significant byte of the status register).

Application: `ANDI` is used to clear selected bits of the CCR. For example, `ANDI #$FA,CCR` clears the Z- and C-bits, i.e., XNZVC = X N 0 V 0.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

X: cleared if bit 4 of data is zero
N: cleared if bit 3 of data is zero
Z: cleared if bit 2 of data is zero
V: cleared if bit 1 of data is zero
C: cleared if bit 0 of data is zero

ANDI to SR AND immediate to status register

Operation:

```
IF [S] = 1
   THEN
        [SR] ← <literal>.[SR]
   ELSE TRAP
```

Syntax: `ANDI #<data>,SR`

Attributes: Size = word

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

Application: This instruction is used to clear the interrupt mask, the S-bit, and the T-bit of the SR. `ANDI #<data>,SR` affects both the status byte of the SR and the CCR. For example, `ANDI #$7FFF,SR` clears the trace bit of the status register, while `ANDI #$7FFE,SR` clears the trace bit and also clears the carry bit of the CCR.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

ASL,ASR Arithmetic shift left/right

Operation: `[destination] ← [destination] shifted by <count>`

Syntax:

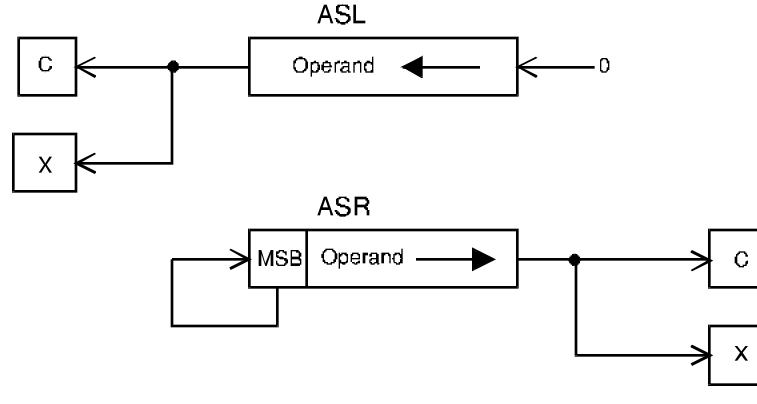
```
ASL Dx,Dy
ASR Dx,Dy
ASL #<data>,Dy
ASR #<data>,Dy
ASL <ea>
ASR <ea>
```

Attributes: Size = byte, word, longword

Description: Arithmetically shift the bits of the operand in the specified direction (i.e., left or right). The shift count may be specified in one of three ways. The count may be a literal, the contents of a data register, or the value 1. An immediate (i.e., literal) count permits a shift of 1 to 8 places. If the count is in a register, the value is modulo 64 (i.e., 0 to 63). If no count is specified, one shift is made (i.e., `ASL <ea>` shifts the contents of the *word* at the effective address one place left).

The effect of an arithmetic shift left is to shift a zero into the least-significant bit position and to shift the most-significant bit out into both the X- and the C-bits of the CCR. The overflow bit of the CCR is set if a sign change occurs during shifting (i.e., if the most-significant bit changes value during shifting).

The effect of an arithmetic shift right is to shift the least-significant bit into both the X- and C-bits of the CCR. The most-significant bit (i.e., the sign bit) is *replicated* to preserve the sign of the number.



Application: `ASL` multiplies a two's complement number by 2. `ASL` is almost identical to the corresponding logical shift, `LSR`. The only difference between `ASL` and `LSL` is that `ASL` sets the V-bit of the CCR if overflow occurs, while `LSL` clears the V-bit to zero. An `ASR` divides a two's complement number by 2. When applied to the contents of a memory location, all 68000 shift operations operate on a word.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

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

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

Bcc Branch on condition cc

Operation: `If cc = 1 THEN [PC] ← [PC] + d`

Syntax: `Bcc <label>`

Sample syntax:
```
BEQ Loop_4
BVC *+8
```

Attributes: `BEQ` takes an 8-bit or a 16-bit offset (i.e., displacement).

Description: If the specified logical condition is met, program execution continues at location [PC] + displacement, d. The displacement is a two's complement value. The value in the PC corresponds to the current location plus two. The range of the branch is -126 to +128 bytes with an 8-bit offset, and -32K to +32K bytes with a 16-bit offset. A short branch to the next instruction is impossible, since the branch code 0 indicates a long branch with a 16-bit offset. The assembly language form `BCC *+8` means branch to the point eight bytes from the current PC if the carry bit is clear.

BCC	branch on carry clear	$\overline{C}$
BCS	branch on carry set	C
BEQ	branch on equal	Z
BGE	branch on greater than or equal	$N.V + \overline{N}.\overline{V}$
BGT	branch on greater than	$N.V.\overline{Z} + \overline{N}.\overline{V}.\overline{Z}$
BHI	branch on higher than	$\overline{C}.\overline{Z}$
BLE	branch on less than or equal	$Z + N.\overline{V} + \overline{N}.V$
BLS	branch on lower than or same	C + Z
BLT	branch on less than	$N.\overline{V} + \overline{N}.V$
BMI	branch on minus (i.e., negative)	N
BNE	branch on not equal	$\overline{Z}$
BPL	branch on plus (i.e., positive)	$\overline{N}$
BVC	branch on overflow clear	$\overline{V}$
BVS	branch on overflow set	V

Note that there are two types of conditional branch instruction:

those that branch on an unsigned condition and those that branch on a signed condition. For example, $FF is greater than $10 when the numbers are regarded as unsigned (i.e., 255 is greater than 16). However, if the numbers are signed, $FF is less than $10 (i.e., -1 is less than 16).

The signed comparisons are:

BGE	branch on greater than or equal
BGT	branch on greater than
BLE	branch on lower than or equal
BLT	branch on less than

The unsigned comparisons are:

BHS	BCC	branch on higher than or same
BHI		branch on higher than
BLS		branch on lower than or same
BLO	BCS	branch on less than

The *official* mnemonics BCC (branch on carry clear) and BCS (branch on carry set) can be renamed as BHS (branch on higher than or same) and BLO (branch on less than), respectively. Many 68000 assemblers support these alternative mnemonics.

Condition codes:

X	N	Z	V	C
-	-	-	-	-

BCHG Test a bit and change

Operation: [Z] ← $\overline{\text{<bit number> OF [destination]}}$
<bit number> OF [destination] ← $\overline{\text{<bit number> OF [destination]}}$

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

Attributes: Size = byte, longword

Description: A bit in the destination operand is tested and the state of the specified bit is reflected in the condition of the Z-bit in the CCR. After the test operation, the state of the specified bit is changed in the destination. If a data register is the destination, then the bit numbering is modulo 32, allowing bit manipulation of all bits in a data register. If a memory location is the destination, a byte is

read from that location, the bit operation performed using the bit number modulo 8, and the byte written back to the location. Note that bit zero refers to the least-significant bit. The bit number for this operation may be specified either *statically* by an immediate value or *dynamically* by the contents of a data register.

Application: If the operation `BCHG #4,$1234` is carried out and the contents of memory location $1234 are 10101010_2, bit 4 is tested. It is a 0 and therefore the Z-bit of the CCR is set to 1. Bit 4 of the destination operand is changed and the new contents of location 1234_{16} are 10111010_2.

Condition codes:

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

Z: set if the bit tested is zero, cleared otherwise.

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

Note that data register direct (i.e., Dn) addressing uses a longword operand, while all other modes use a byte operand.

BCLR Test a bit and clear

Operation:

```
[Z] ← <bit number> OF [destination]
<bit number> OF [destination] ← 0
```

Syntax:

```
BCLR Dn,<ea>
BCLR #<data>,<ea>
```

Attributes: Size = byte, longword

Description: A bit in the destination operand is tested and the state of the specified bit is reflected in the condition of the Z-bit in the condition code. After the test, the state of the specified bit is cleared in the destination. If a data register is the destination, the bit numbering is modulo 32, allowing bit manipulation of all bits in a data register. If a memory location is the destination, a byte is read from that location, the bit operation performed using the bit number modulo 8, and the byte written back to the location.

Bit zero refers to the least-significant bit. The bit number for this operation may be specified either by an immediate value or dynamically by the contents of a data register.

Application: If the operation `BCLR #4,$1234` is carried out and the contents of memory location $1234 are 11111010_2, bit 4 is tested. It is a 1 and therefore the Z-bit of the CCR is set to 0. Bit 4 of the destination operand is cleared and the new contents of $1234 are: 11101010_2.

Condition codes:

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

Z: set if the bit tested is zero, cleared otherwise.

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

Note that data register direct (i.e., Dn) addressing uses a longword operand, while all other modes use a byte operand.

BRA Branch always

Operation: `[PC] ← [PC] + d`

Syntax:

```
BRA <label>
BRA <literal>
```

Attributes: Size = byte, word

Description: Program execution continues at location [PC] + d. The displacement, d, is a two's complement value (8 bits for a short branch and 16 bits for a long branch). The value in the PC corresponds to the current location plus two. Note that a short branch to the next instruction is impossible, since the branch code 0 is used to indicate a long branch with a 16-bit offset.

Application: A `BRA` is an unconditional relative jump (or goto). You use a `BRA` instruction to write position independent code, because the destination address (*branch target address*) is specified with respect to the current value of the PC. A `JMP` instruction does not produce position independent code.

Condition codes:

X	N	Z	V	C
-	-	-	-	-

BSET Test a bit and set

Operation: [Z] ← <bit number> OF [destination]
<bit number> OF [destination] ← 1

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

Attributes: Size = byte, longword

Description: A bit in the destination operand is tested and the state of the specified bit is reflected in the condition of the Z-bit of the condition code. After the test, the specified bit is set in the destination. If a data register is the destination then the bit numbering is modulo 32, allowing bit manipulation of all bits in a data register. If a memory location is the destination, a byte is read from that location, the bit operation performed using bit number modulo 8, and the byte written back to the location. Bit zero refers to the least-significant bit. The bit number for this operation may be specified either by an immediate value or dynamically by the contents of a data register.

Condition codes:

X	N	Z	V	C
-	-	*	-	-

Z: set if the bit tested is zero, cleared otherwise.

Destination operand addressing mode for BSET Dn,<ea> form

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

Note that data register direct (i.e., Dn) addressing uses a longword operand, while all other modes use a byte operand.

BSR Branch to subroutine

Operation: [SP] ← [SP] - 4; [M([SP])] ← [PC]; [PC] ← [PC] + d

Syntax:
```
BSR <label>
BSR <literal>
```

Attributes: Size = byte, word

Description: The longword address of the instruction immediately following the BSR instruction is pushed onto the system stack pointed at by A7. Program execution then continues at location [PC] + displacement. The displacement is an 8-bit two's complement value for a short branch, or a 16-bit two's complement value for a long branch. The value in the PC corresponds to the current location plus two. Note that a short branch to the next instruction is impossible, since the branch code 0 is used to indicate a long branch with a 16-bit offset.

Application: BSR is used to call a procedure or a subroutine. Since it provides relative addressing (and therefore position independent code), its use is preferable to JSR.

Condition codes:
```
X  N  Z  V  C
-  -  -  -  -
```

BTST Test a bit

Operation: [Z] ← <bit number> OF [destination]

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

Attributes: Size = byte, longword

Description: A bit in the destination operand is tested and the state of the specified bit is reflected in the condition of the Z-bit in the CCR. The destination is not modified by a BTST instruction. If a data register is the destination, then the bit numbering is modulo 32, allowing bit manipulation of all bits in a data register. If a memory location is the destination, a byte is read from that location, the bit operation performed. Bit 0 refers to the least-significant bit. The bit number for this operation may be specified either statically by an immediate value or dynamically by the contents of a data register.

Condition codes:
```
X  N  Z  V  C
-  -  *  -  -
```
Z: set if the bit tested is zero, cleared otherwise.

Destination operand addressing modes for `BTST Dn,<ea>` form

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	

Note that data register direct (i.e., Dn) addressing uses a longword operand, while all other modes use a byte operand.

CHK Check register against bounds

Operation: `IF [Dn] < 0 OR [Dn] > [<ea>] THEN TRAP`

Syntax: `CHK <ea>,Dn`

Attributes: Size = word

Description: The contents of the low-order word in the data register specified in the instruction are examined and compared with the upper bound at the 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 contained in the operand word, then the processor initiates exception processing.

Application: The `CHK` instruction can be used to test the bounds of an array element before it is used. By performing this test, you can make certain that you do not access an element outside an array. Consider the following fragment of code:

```
MOVE.W  subscript,D0     Get subscript to test
CHK     #max_bound,D0    Test subscript against 0 and upper bound
*                        TRAP on error ELSE continue if ok
```

Condition codes:

X	N	Z	V	C
-	*	U	U	U

N: set if `[Dn] < 0`; cleared if `[Dn] > [<ea>]`; undefined otherwise.

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

CLR Clear an operand

Operation: `[destination] ← 0`

Syntax: `CLR <ea>`

Sample syntax: `CLR (A4)+`

Attributes: Size = byte, word, longword

Description: The destination is cleared — loaded with all zeros. The `CLR` instruction can't be used to clear an address register. You can use `SUBA.L A0,A0` to clear A0. Note that a side effect of `CLR`'s implementation is a *read* from the specified effective address before the clear (i.e., write) operation is executed. Under certain circumstances this might cause a problem (e.g., with write-only memory).

Condition codes:

X	N	Z	V	C
-	0	1	0	0

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

CMP Compare

Operation: `[destination] - [source]`

Syntax: `CMP <ea>,Dn`

Sample syntax: `CMP (Test,A6,D3.W),D2`

Attributes: Size = byte, word, longword

Description: Subtract the source operand from the destination operand and set the condition codes accordingly. The destination must be a data register. The destination is not modified by this instruction.

Condition codes:

X	N	Z	V	C
-	*	*	*	*

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

CMPA Compare address

Operation: `[destination] - [source]`

Syntax: `CMPA <ea>,An`

Sample syntax:
```
CMPA.L #$1000,A4
CMPA.W (A2)+,A6
CMPA.L D5,A2
```

Attributes: Size = word, longword

Description: Subtract the source operand from the destination address register and set the condition codes accordingly. The address register is not modified. The size of the operation may be specified as word or longword. Word length operands are sign-extended to 32 bits before the comparison is carried out.

Condition codes:
```
X  N  Z  V  C
-  *  *  *  *
```

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

CMPI Compare immediate

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 accordingly — the destination is not modified. `CMPI` permits the comparison of a literal with memory.

Condition codes:

X	N	Z	V	C
-	*	*	*	*

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	

CMPM Compare memory with memory

Operation: `[destination] - [source]`

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

Attributes: Size = byte, word, longword

Sample syntax: `CMPM.B (A3)+,(A4)+`

Description: Subtract the source operand from the destination operand and set the condition codes accordingly. The destination is not modified by this instruction. The only permitted addressing mode is address register indirect with post-incrementing for both source and destination operands.

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

```
*       Compare two blocks of memory for equality
        LEA     Source,A0       A0 points to source block
        LEA     Destination,A1  A1 points to destination block
        MOVE.W  #Count-1,D0     Compare Count words
 RPT    CMPM.W  (A0)+,(A1)+     Compare pair of words
        DBNE    D0,RPT          Repeat until all done
        .
        .
```

Condition codes:

X	N	Z	V	C
-	*	*	*	*

DBcc Test condition, decrement, and branch

Operation:

```
IF(condition false)
   THEN [Dn] ← [Dn] - 1 {decrement loop counter}
     IF [Dn] = -1 THEN [PC] ← [PC] + 2 {fall through to next instruction}
                  ELSE [PC] ← [PC] + d {take branch}
   ELSE [PC] ← [PC] + 2 {fall through to next instruction}
```

Syntax: `DBcc Dn,<label>`

Attributes: Size = word

Description: The `DBcc` instruction provides an automatic looping facility and replaces the usual decrement counter, test, and branch instructions. Three parameters are required by the `DBcc` instruction: a branch condition (specified by 'cc'), a data register that serves as the loop down-counter, and a label that indicates the start of the loop. The `DBcc` first tests the condition 'cc', and if 'cc' is true the loop is terminated and the branch back to `<label>` not taken. The 14 branch conditions supported by `Bcc` are also supported by `DBcc`, as well as `DBF` and `DBT` (F = false, and T = true). Note that many assemblers permit the mnemonic `DBF` to be expressed as `DBRA` (i.e., decrement and branch back).

It is important to appreciate that the condition tested by the `DBcc` instruction works in the *opposite* sense to a `Bcc`, conditional branch, instruction. For example, `BCC` means branch on carry clear, whereas `DBCC` means continue (i.e., exit the loop) on carry clear. That is, the `DBcc` condition is a loop terminator. If the termination condition is not true, the low-order 16 bits of the specified data register are decremented. If the result is -1, the loop is not taken and the next instruction is executed. If the result is not -1, a branch is made to 'label'. Note that the label represents a 16-bit signed value, permitting a branch range of -32K to +32K bytes. Since the value in Dn decremented is 16 bits, the loop may be executed up to 64K times.

We can use the instruction `DBEQ`, decrement and branch on zero, to mechanize the high-level language construct `REPEAT...UNTIL`.

```
LOOP   ...                REPEAT
       ...
       ...                  [D0] := [D0] - 1
       ...
       DBEQ   D0,REPEAT  UNTIL [D0] = - 1 OR [Z] = 1
```

Application: Suppose we wish to input a block of 512 bytes of data (the data is returned in register D1). If the input routine returns a value zero in D1, an error has occurred and the loop must be exited.

```
        LEA    Dest,A0   Set up pointer to destination
        MOVE.W #511,D0   512 bytes to be input
AGAIN   BSR    INPUT     Get the data in D1
        MOVE.B D1,(A0)+  Store it
        DBEQ   D0,AGAIN  REPEAT until D1 = 0 OR 512 times
```

Condition codes:

X	N	Z	V	C
-	-	-	-	-

Not affected

DIVS, DIVU Signed divide, unsigned divide

Operation: [destination] ← [destination]/[source]

Syntax:
```
DIVS <ea>,Dn
DIVU <ea>,Dn
```

Attributes: Size = longword/word = longword result

Description: Divide the destination operand by the source operand and store the result in the destination. The destination is a longword and the source is a 16-bit value. The result (i.e., destination register) is a 32-bit value arranged so that the quotient is the lower-order word and the remainder is the upper-order word. `DIVU` performs division on unsigned values, and `DIVS` performs division on two's complement values. An attempt to divide by zero causes an exception. For `DIVS`, 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. If overflow is detected during division, the operands are unaffected. Overflow is checked for at the start of the operation and occurs if the quotient is larger than a 16-bit signed integer. If the upper word of the dividend is greater than or equal to the divisor, the V-bit is set and the instruction terminated.

Application: Consider the division of D0 by D1, `DIVU D1,D0`, which results in:

```
[D0(0:15)] ← [D0(0:31)]/[D1(0:15)]
[D0(16:31)] ← remainder
```

Condition codes:

X	N	Z	V	C
-	*	*	*	0

The X-bit is not affected by a division. The N-bit is set if the quotient is negative. The Z-bit is set if the quotient is zero. The V-bit is set if division overflow occurs (in which case the Z- and N-bits are undefined). The C-bit is always cleared.

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

EOR Exclusive OR logical

Operation: `[destination] ← [source] ⊕ [destination]`

Syntax: `EOR Dn,<ea>`

Sample syntax: `EOR D3,-(A3)`

Attributes: Size = byte, word, longword.

Description: EOR (exclusive or) the source operand with the destination operand and store the result in the destination location. Note that the source operand must be a data register and that the operation `EOR <ea>,Dn` is not permitted.

Application: The EOR instruction is used to *toggle* (i.e., change the state of) selected bits in the operand. For example, if [D0] = 00001111, and [D1] = 10101010, the operation `EOR.B D0,D1` toggles bits 0 to 3 of D1 and results in [D1] = 10100101.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

EORI EOR immediate

Operation: [destination] ← <literal> ⊕ [destination]

Syntax: EORI #<data>,<ea>

Attributes: Size = byte, word, longword

Description: EOR the immediate data with the contents of the destination operand. Store the result in the destination operand.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

EORI to CCR EOR immediate to CCR

Operation: [CCR] ← <literal> ⊕ [CCR]

Syntax: EORI #<data>,CCR

Attributes: Size = byte

Description: EOR the immediate data with the contents of the condition code register (i.e., the least-significant byte of the status register).

Application: Used to toggle bits in the CCR. For example, EORI #$0C,CCR toggles the N- and Z-bits of the CCR.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

X:= toggled if bit 4 of data = 1; unchanged otherwise
N:= toggled if bit 3 of data = 1; unchanged otherwise
Z:= toggled if bit 2 of data = 1; unchanged otherwise
V:= toggled if bit 1 of data = 1; unchanged otherwise
C:= toggled if bit 0 of data = 1; unchanged otherwise

EORI to SR EOR immediate to status register

Operation:
```
IF [S] = 1
   THEN
       [SR] ← <literal> ⊕ [SR]
   ELSE TRAP
```

Syntax: `EORI #<data>,SR`

Attributes: Size = word

Description: EOR (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
*	*	*	*	*

X:= toggled if bit 4 of data = 1; unchanged otherwise
N:= toggled if bit 3 of data = 1; unchanged otherwise
Z:= toggled if bit 2 of data = 1; unchanged otherwise
V:= toggled if bit 1 of data = 1; unchanged otherwise
C:= toggled if bit 0 of data = 1; unchanged otherwise

EXG Exchange registers

Operation: `[Rx] ← [Ry]; [Ry] ← [Rx]`

Syntax: `EXG Rx,Ry`

Sample syntax:
```
EXG D3,D4
EXG D2,A0
EXG A7,D5
```

Attributes: Size = longword

Description: Exchange the contents of two registers. The size of the instruction is a longword because the entire 32-bit contents of two registers are exchanged. The instruction permits the exchange of address registers, data registers, and address and data registers.

Application: One application of `EXG` is to load an address into a data register and then process it using instructions that act on data registers. Then the reverse operation can be used to return the result to the

address register. Doing this preserves the original contents of the data register.

Condition codes:

```
X  N  Z  V  C
-  -  -  -  -
```

EXT Sign-extend a data register

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

Syntax:

```
EXT.W Dn
EXT.L Dn
```

Attributes: Size = word, longword

Description: Extend the least-significant byte in a data register to a word, or extend the least-significant word in a data register to a longword. If the operation is word sized, bit 7 of the designated data register is copied to bits (8:15). If the operation is longword sized, bit 15 is copied to bits (16:31).

Application: If `[D0] = $12345678, EXT.W D0` results in 12340078_{16}
If `[D0] = $12345678, EXT.L D0` results in 00005678_{16}

Condition codes:

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

ILLEGAL Illegal instruction

Operation:

```
[SSP] ← [SSP] - 4; [M([SSP])] ← [PC];
[SSP] ← [SSP] - 2; [M([SSP])] ← [SR];
[PC] ← Illegal instruction vector
```

Syntax:

```
ILLEGAL
```

Attributes: None

Description: The bit pattern of the illegal instruction, $4AFC_{16}$ causes the illegal instruction trap to be taken. As in all exceptions, the contents of the program counter and the processor status word are pushed onto the supervisor stack at the start of exception processing.

Application: Any *unknown* pattern of bits read by the 68000 during an instruction read phase will cause an illegal instruction trap. The `ILLEGAL` instruction can be thought of as an *official* illegal instruction. It can be used to test the illegal instruction trap and will always be an illegal instruction in any future enhancement of the 68000.

Condition codes:

```
X  N  Z  V  C
-  -  -  -  -
```

JMP Jump (unconditionally)

Operation: `[PC] ← destination`

Syntax: `JMP <ea>`

Attributes: Unsized

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

Application: Apart from a simple unconditional jump to an address fixed at compile time (i.e., `JMP label`), the `JMP` instruction is useful for the calculation of *dynamic* or *computed* jumps. For example, the instruction `JMP (A0,D0.L)` jumps to the location pointed at by the contents of address register A0, offset by the contents of data register D0. Note that `JMP` provides several addressing modes, while `BRA` provides a single addressing mode (i.e., PC relative).

Condition codes:

```
X  N  Z  V  C
-  -  -  -  -
```

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
		✔			✔	✔	✔	✔	✔	✔	

JSR Jump to subroutine

Operation:

```
[SP] ← [SP] - 4; [M([SP])] ← [PC]
[PC] ← destination
```

Syntax: `JSR <ea>`

Attributes: Unsized

Description: `JSR` pushes the longword address of the instruction immediately following the `JSR` onto the system stack. Program execution then continues at the address specified in the instruction.

Application: `JSR (Ai)` calls the procedure pointed at by address register Ai. The instruction `JSR (Ai,Dj)` calls the procedure at the location [Ai] + [Dj] which permits dynamically computed addresses.

Condition codes:

X	N	Z	V	C
-	-	-	-	-

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
		✔			✔	✔	✔	✔	✔	✔	

LEA Load effective address

Operation: `[An] ← <ea>`

Syntax: `LEA <ea>,An`

Sample syntax:

```
LEA Table,A0
LEA (Table,PC),A0
LEA (-6,A0,D0.L),A6
LEA (Table,PC,D0),A6
```

Attributes: Size = longword

Description: The effective address is computed and loaded into the specified address register. For example, `LEA (-6,A0,D0.W),A1` calculates the sum of address register A0 plus data register D0.W sign-extended to 32 bits minus 6, and deposits the result in address register A1. The difference between the `LEA` and `PEA` instructions is that `LEA` calculates an effective address and puts it in an address register, while `PEA` calculates an effective address in the same way but pushes it on the stack.

Application: LEA is a very powerful instruction used to calculate an effective address. In particular, the use of LEA facilitates the writing of position independent code. For example, LEA (TABLE,PC),A0 calculates the effective address of 'TABLE' with respect to the PC and deposits it in A0.

```
        LEA  (Table,PC),A0     Compute address of Table with respect to PC
        MOVE (A0),D1           Pick up the first item in the table
        .                      Do something with this item
        MOVE D1,(A0)           Put it back in the table
        .
        .
Table DS.B  100
```

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
		✔			✔	✔	✔	✔	✔	✔	

Condition codes:

X	N	Z	V	C
-	-	-	-	-

LINK Link and allocate

Operation: [SP] ← [SP] - 4; [M([SP])] ← [An];
[An] ← [SP]; [SP] ← [SP] + d

Syntax: LINK An,#<displacement>

Sample syntax: LINK A6,#-12

Attributes: Size = word

Description: The contents of the specified address register are first pushed onto the stack. Then, the address register is loaded with the updated stack pointer. Finally, the 16-bit sign-extended displacement is added to the stack pointer. The contents of the address register occupy two words on the stack. A *negative displacement* must be used to allocate stack area to a procedure. At the end of a LINK instruction, the old value of address register An has been pushed on the stack and the new An is pointing at

the base of the stack frame. The stack pointer itself has been moved up by d bytes and is pointing at the top of the stack frame. Address register An is called the f*rame pointer* because it is used to reference data on the stack frame. By convention, programmers often use A6 as a frame pointer.

Application: The `LINK` and `UNLK` pair are used to create local workspace on the top of a procedure's stack. Consider the code:

```
Subrtn LINK A6,#-12    Create a 12-byte workspace
       .
       MOVE D3,(-8,A6) Access the stack frame via A6
       .
       .
       UNLK A6         Collapse the workspace
       RTS             Return from subroutine
```

Condition codes:

X	N	Z	V	C
-	-	-	-	-

The `LINK` instruction does not affect the CCR.

LSL, LSR Logical shift left/right

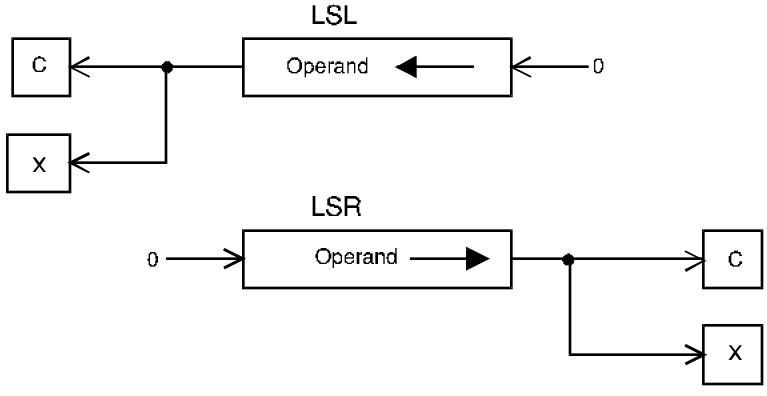
Operation: `[destination] ← [destination] shifted by <count>`

Syntax:

```
LSL Dx,Dy
LSR Dx,Dy
LSL #<data>,Dy
LSR #<data>,Dy
LSL <ea>
LSR <ea>
```

Attributes: Size = byte, word, longword

Description: Logically shift the bits of the operand in the specified direction (i.e., left or right). A zero is shifted into the input position and the bit shifted out is copied into both the C- and the X-bit of the CCR. The shift count may be specified in one of three ways. The count may be a literal, the contents of a data register, or the value 1. An immediate count permits a shift of 1 to 8 places. If the count is in a register, the value is modulo 64 — from 0 to 63. If no count is specified, one shift is made (e.g., `LSL <ea>` shifts the *word* at the effective address one position left).



Application: If [D3.W] = 1100110010101110_2, the instruction `LSL.W #5,D3` produces the result 1001010111000000_2. After the shift, both the X-and C-bits of the CCR are set to 1 (since the last bit shifted out was a 1).

Condition codes:

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

The X-bit is set to the last bit shifted out of the operand and is equal to the C-bit. However, a zero shift count leaves the X-bit unaffected and the C-bit cleared.

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

MOVE Copy data from source to destination

Operation: `[destination] ← [source]`

Syntax: `MOVE <ea>,<e>`

Sample syntax:

```
MOVE (A5),-(A2)
MOVE -(A5),(A2)+
MOVE #$123,(A6)+
MOVE Temp1,Temp2
```

Attributes: Size = byte, word, longword

Description: Move the contents of the source to the destination location. The data is examined as it is moved and the condition codes set accordingly. Note that this is actually a *copy* command because the source is not affected by the move. The move instruction has the widest range of addressing modes of all the 68000's instructions.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

MOVEA Move address

Operation: [An] ← [source]

Syntax: MOVEA <ea>,An

Attributes: Size = word, longword

Description: Move the contents of the source to the destination location. The destination is an address register. The source must be a word or longword. If it is a word, it is sign-extended to a longword. The condition codes are not affected.

Application: The MOVEA instruction is used to load an address register (some assemblers simply employ the MOVE mnemonic for both MOVE and MOVEA). Note that the instruction LEA can often be used to perform the same operation (e.g., MOVEA.L #$1234,A0 is the same as LEA $1234,A0).

Take care because the `MOVEA.W #$8000,A0` instruction sign-extends the source operand to $FFFF8000 before loading it into A0, whereas `LEA $8000,A0` loads A0 with $00008000.

You should appreciate that the `MOVEA` and `LEA` instructions are not interchangeable. The operation `MOVEA (Ai),An` cannot be implemented by an `LEA` instruction, since `MOVEA (Ai),An` performs a memory access to obtain the source operand, as the following RTL demonstrates.

```
LEA   (Ai),An  =  [An] ← [Ai]
MOVEA (Ai),An  =  [An] ← [M([Ai])]
```

Condition codes:

X	N	Z	V	C
-	-	-	-	-

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

MOVE to CCR Copy data to CCR from source

Operation: `[CCR] ← [source]`

Syntax: `MOVE <ea>,CCR`

Attributes: Size = word

Description: Move the contents of the source operand to the condition code register. The source operand is a *word*, but only the low-order *byte* contains the condition codes. The upper byte is neglected. Note that `MOVE <ea>,CCR` is a word operation, but `ANDI`, `ORI`, and `EORI to CCR` are all byte operations.

Application: The move to CCR instruction permits the programmer to preset the CCR. For example, `MOVE #0,CCR` clears all the CCR's bits.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

MOVE from SR Copy data from SR to destination

Operation: `[destination] ← [SR]`

Syntax: `MOVE SR,<ea>`

Attributes: Size = word

Description: Move the contents of the status register to the destination location. The source operand, the status register, is a word. This instruction is not privileged in the 68000, but is privileged in the 68010, 68020, and 68030. Executing a `MOVE SR,<ea>` while in the user mode on these processors results in a privilege violation trap.

Condition codes:

X	N	Z	V	C
-	-	-	-	-

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

MOVE to SR Copy data to SR from source

Operation:

```
IF [S] = 1
   THEN [SR] ← [source]
ELSE TRAP
```

Syntax: `MOVE <ea>,SR`

Attributes: Size = word

Description: Move the contents of the source operand to the status register. The source operand is a word and all bits of the status register are affected.

Application: The `MOVE to SR` instruction allows the programmer to preset the contents of the status register. This instruction permits the trace mode, interrupt mask, and status bits to be modified. For example, `MOVE #$2700,SR` moves 00100111 00000000 to the status register which clears all bits of the CCR, sets the S-bit, clears the T-bit, and sets the interrupt mask level to 7.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

MOVE USP Copy data to or from USP

Operation 1:
```
IF [S] = 1                          {MOVE USP,An form}
   THEN [USP] ← [An]
ELSE TRAP
```

Operation 2:
```
IF [S] = 1                          {MOVE An,USP form}
   THEN [An] ← [USP]
ELSE TRAP
```

Syntax 1: `MOVE USP,An`
Syntax 2: `MOVE An,USP`

Attributes: Size = longword

Description: Move the contents of the user stack pointer to an address register or vice versa. This is a privileged instruction and allows the operating system running in the supervisor state either to read the contents of the user stack pointer or to set up the user stack pointer.

Condition codes:

X	N	Z	V	C
-	-	-	-	-

MOVEM Move multiple registers

Operation 1 :
```
REPEAT
   [destination_register] ← [source]
UNTIL all registers in list moved
```

Operation 2:
```
REPEAT
   [destination] ← [source_register]
UNTIL all registers in list moved
```

Syntax 1: `MOVEM <ea>,<register list>`
Syntax 2: `MOVEM <register list>,<ea>`

Sample syntax:
```
MOVEM.L D0-D7/A0-A6,$1234
MOVEM.L (A5),D0-D2/D5-D7/A0-A3/A6
MOVEM.W (A7)+,D0-D5/D7/A0-A6
MOVEM.W D0-D5/D7/A0-A6,-(A7)
```

Attributes: Size = word, longword

Description: The group of registers specified by `<register list>` is copied to or from consecutive memory locations. The starting location is provided by the effective address. Any combination of the 68000's sixteen address and data registers can be copied by a single `MOVEM` instruction. Note that either a word or a longword can be moved, and that a word is sign-extended to a longword when it is moved (even if the destination is a data register).

When a group of registers is transferred to or from memory (using an addressing mode other than pre-decrementing or post-incrementing), the registers are transferred starting at the specified address and up through higher addresses. The order of transfer of registers is data register D0 to D7, followed by address register A0 to A7.

For example, `MOVEM.L D0-D2/D4/A5/A6,$1234` moves registers `D0,D1,D2,D4,A5,A6` to memory, starting at location $1234 (in which D0 is stored) and moving to locations $1238, $123C,... Note that the address counter is incremented by 2 or 4 after each move according to whether the operation is moving words or longwords, respectively.

If the effective address is in the pre-decrement mode (i.e., `-(An)`), only a register to memory operation is permitted. The registers are stored starting at the specified address minus two (or four for longword operands) and down through lower addresses. The order of storing is from address register A7 to address register A0, then from data register D7 to data register

D0. The decremented address register is updated to contain the address of the last word stored.

If the effective address is in the post-increment mode (i.e., `(An)+`), only a memory to register transfer is permitted. The registers are loaded starting at the specified address and up through higher addresses. The order of loading is the inverse of that used by the pre-decrement mode and is D0 to D7 followed by A0 to A7. The incremented address register is updated to contain the address of the last word plus two (or four for longword operands).

Note that the `MOVEM` instruction has a side effect. An extra bus cycle occurs for memory operands, and an operand at one address higher than the last register in the list is accessed. This extra access is an 'overshoot' and has no effect as far as the programmer is concerned. However, it could cause a problem if the overshoot extended beyond the bounds of physical memory. Once again, remember that `MOVEM.W` sign-extends words when they are moved to data registers.

Application: This instruction is invariably used to save working registers on entry to a subroutine and to restore them at the end of a subroutine.

```
          BSR     Example
          .
          .
Example   MOVEM.L D0-D5/A0-A3,-(SP)  Save registers
          .
          .
          Body of subroutine
          .
          .
          MOVEM.L (SP)+,D0-D5/A0-A3  Restore registers
          RTS                        Return
```

Condition codes:

X	N	Z	V	C
-	-	-	-	-

Source operand addressing modes (memory to register)

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
		✔	✔		✔	✔	✔	✔	✔	✔	

Destination operand addressing modes (register to memory)

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
		✔		✔	✔	✔	✔	✔			

MOVEP Move peripheral data

Operation: `[destination] ← [source]`

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

Sample syntax:
```
MOVEP D3,(Control,A0)
MOVEP (Input,A6),D5
```

Attributes: Size = word, longword

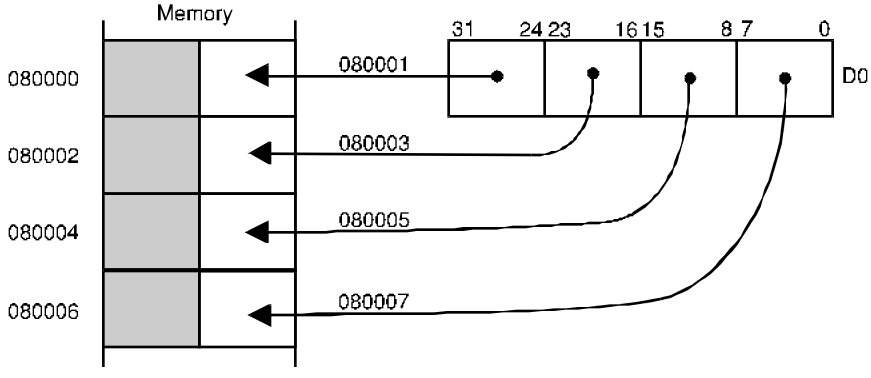
Description: The `MOVEP` operation moves data between a data register and a byte-oriented memory mapped peripheral. The data is moved between the specified data register and *alternate bytes* within the peripheral's address space, starting at the location specified and incrementing by two. This instruction is designed to be used in conjunction with 8-bit peripherals connected to the 68000's 16-bit data bus. The high-order byte of the data register is transferred first and the low-order byte transferred last. The memory address is specified by the address register indirect mode with a 16-bit offset. If the address is even, all transfers are to or from the high-order half of the data bus. If the address is odd, all the transfers are made to the low-order half of the data bus.

Application: Consider a memory-mapped peripheral located at address $08 0001 which has four 8-bit internal registers mapped at addresses $08 0001, $08 0003, $08 0005, and $08 0007. The longword in data register D0 is to be transferred to this peripheral by the following code.

```
LEA      $080001,A0
MOVEP.L  D0,0(A0)
```

This code results in the following actions:

```
[M(080001)] ← [D0(24:31)]
[M(080003)] ← [D0(16:23)]
[M(080005)] ← [D0(8:15)]
[M(080007)] ← [D0(0:7)]
```



Condition codes: X N Z C V
- - - - -

MOVEQ Move quick (copy a small literal to a destination)

Operation: `[destination] ← <literal>`

Syntax: `MOVEQ #<data>,Dn`

Attributes: Size = longword

Description: Move the specified literal to a data register. The literal data is an eight-bit field within the `MOVEQ` op-code and specifies a signed value in the range -128 to +127. When the source operand is transferred, it is sign-extended to 32 bits. Consequently, although only 8 bits are moved, the `MOVEQ` instruction is a *longword* operation.

Application: `MOVEQ` is used to load small integers into a data register. Beware of its sign-extension. The two operations `MOVE.B #12,D0` and `MOVEQ #12,D0` are not equivalent. The former has the effect [D0(0:7)] ← 12, while the latter has the effect [D0(0:31)] ← 12 (with sign-extension).

Condition codes:

X	N	Z	V	C
-	*	*	0	0

MULS, MULU Signed multiply, unsigned multiply

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

Syntax:
```
MULS <ea>,Dn
MULU <ea>,Dn
```

Attributes: Size = word (the product is a longword)

Description: Multiply the 16-bit destination operand by the 16-bit source operand and store the result in the destination. Both the source and destination are 16-bit word values and the destination result is a 32-bit longword. The product is therefore a correct product and is not truncated. `MULU` performs multiplication with unsigned values and `MULS` performs multiplication with two's complement values.

Application: `MULU D1,D2` multiplies the low-order words of data registers D1 and D2 and puts the 32-bit result in D2. `MULU #$1234,D3` multiplies the low-order word of D3 by the 16-bit literal $1234 and puts the 32-bit result in D3.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

NBCD Negate decimal with sign extend

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

Syntax:
```
NBCD <ea>
```

Attributes: Size = byte

Description: The operand addressed as the destination and the extend bit in the CCR are subtracted from zero. The subtraction is performed using binary coded decimal (BCD) arithmetic. This instruction calculates the ten's complement of the destination if the X-bit is clear, and the nine's complement if X = 1. This is a byte-only operation. Negating a BCD number (with X = 0) has the effect of subtracting it from 100_{10}.

Condition codes:

X	N	Z	V	C
*	U	*	U	*

The Z-bit is cleared if the result is non-zero and is unchanged otherwise. The C-bit is set if a decimal borrow occurs. The X-bit is set to the same value as the C-bit.

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

NEG Negate

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. The difference between `NOT` and `NEG` instructions is that `NOT` performs a bit-by-bit logical complementation, while a `NEG` performs a two's complement arithmetic subtraction. All bits of the condition code register are modified by a `NEG` operation. For example, if D3.B = 11100111_2, the logical operation `NEG.B D3` results in D3 = 00011001_2 (XNZVC=10001) and `NOT.B D3` = 00011000_2 (XNZVC=-0000).

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Note that the X-bit is set to the value of the C-bit.

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

NEGX Negate with extend

Operation: `[destination] ← 0 - [destination] - [X]`

Syntax: `NEGX <ea>`

Attributes: Size = byte, word, longword

Description: The operand addressed as the destination and the extend bit are subtracted from zero. `NEGX` is the same as `NEG` except that the X-bit is also subtracted from zero.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

The Z-bit is cleared if the result is non-zero and is unchanged otherwise. The X-bit is set to the same value as the C-bit.

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

NOP No operation

Operation: None

Syntax: `NOP`

Attributes: Unsized

Description: The no operation instruction, `NOP` performs no *computation*. Execution continues with the instruction following the `NOP` instruction. The processor's state is not modified by a `NOP`.

Application: NOPs can be used to introduce a *delay* in code. Some programmers use them to provide space for *patches* — two or more NOPs can later be replaced by branch or jump instructions to fix a bug. This use of the NOP is seriously frowned upon, as errors should be corrected by re-assembling the code rather than by patching it.

Condition codes:

X	N	Z	V	C
-	-	-	-	-

NOT Logical complement

Operation: [destination] ← [destination]

Syntax: NOT <ea>

Attributes: Size = byte, word, longword

Description: Calculate the logical complement of the destination and store the result in the destination. The difference between NOT and NEG is that NOT performs a bit-by-bit logical complementation, while a NEG performs a two's complement arithmetic subtraction. Moreover, NEG updates all bits of the CCR, while NOT clears the V- and C-bits, updates the N- and Z-bits, and doesn't affect the X-bit.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

OR OR logical

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

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

Attributes: Size = byte, word, longword

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

Application: The `OR` instruction is used to set selected bits of the operand. For example, we can set the four most-significant bits of a longword operand in D0 by executing:

```
OR.L #$F0000000,D0
```

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

ORI OR immediate

Operation: `[destination] ← <literal> + [destination]`

Syntax: `ORI #<data>,<ea>`

Attributes: Size = byte, word, longword

Description: OR the immediate data with the destination operand. Store the result in the destination operand.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Application: `ORI` forms the logical OR of the immediate source with the effective address, which may be a memory location. For example,

```
ORI.B #%00000011,(A0)+
```

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

ORI to CCR Inclusive OR immediate to CCR

Operation: [CCR] ← <literal> + [CCR]

Syntax: ORI #<data>,CCR

Attributes: Size = byte

Description: OR the immediate data with the condition code register (i.e., the least-significant byte of the status register). For example, the Z flag of the CCR can be set by ORI #$04,CCR.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

X is set if bit 4 of data = 1; unchanged otherwise
N is set if bit 3 of data = 1; unchanged otherwise
Z is set if bit 2 of data = 1; unchanged otherwise
V is set if bit 1 of data = 1; unchanged otherwise
C is set if bit 0 of data = 1; unchanged otherwise

ORI to SR Inclusive OR immediate to status register

Operation:

```
IF [S] = 1
   THEN
      [SR] ← <literal> + [SR]
   ELSE TRAP
```

Syntax: ORI #<data>,SR

Attributes: Size = word

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

Application: Used to set bits in the SR (i.e., the S, T, and interrupt mask bits). For example, `ORI #$8000,SR` sets bit 15 of the SR (i.e., the trace bit).

Condition codes:

```
X  N  Z  V  C
*  *  *  *  *
```

X is set if bit 4 of data = 1; unchanged otherwise
N is set if bit 3 of data = 1; unchanged otherwise
Z is set if bit 2 of data = 1; unchanged otherwise
V is set if bit 1 of data = 1; unchanged otherwise
C is set if bit 0 of data = 1; unchanged otherwise

PEA Push effective address

Operation: `[SP] ← [SP] - 4; [M([SP])] ← <ea>`

Syntax: `PEA <ea>`

Attributes: Size = longword

Description: The longword effective address specified by the instruction is computed and pushed onto the stack. The difference between `PEA` and `LEA` is that `LEA` calculates an effective address and puts it in an address register, while `PEA` calculates an effective address in the same way but pushes it on the stack.

Application: `PEA` calculates an effective address to be used later in address register indirect addressing. In particular, it facilitates the writing of position independent code. For example, `PEA (TABLE,PC)` calculates the address of `TABLE` with respect to the PC and pushes it on the stack. This address can be read by a procedure and then used to access the data to which it points. Consider the example:

```
            PEA     Wednesday     Push the parameter address on the stack
            BSR     Subroutine    Call the procedure
            LEA     (4,SP),SP     Remove space occupied by the parameter
            .
Subroutine  MOVEA.L (4,SP),A0     A0 points to parameter under return address
            MOVE.W  (A0),D2       Access the actual parameter — Wednesday
            .
            RTS
```

Condition codes:

```
X  N  Z  V  C
-  -  -  -  -
```

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
		✔			✔	✔	✔	✔	✔	✔	

RESET Reset external devices

Operation:
```
IF [S] = 1 THEN
             Assert RESET* line
           ELSE TRAP
```

Syntax:
```
RESET
```

Attributes: Unsized

Description: The reset line is asserted, causing all external devices connected to the 68000's RESET* output to be reset. The `RESET` instruction is privileged and has no effect on the operation of the 68000 itself. This instruction is used to perform a programmed reset of all peripherals connected to the 68000's RESET* pin.

Condition codes:
```
X  N  Z  V  C
-  -  -  -  -
```

ROL, ROR Rotate left/right (without extend)

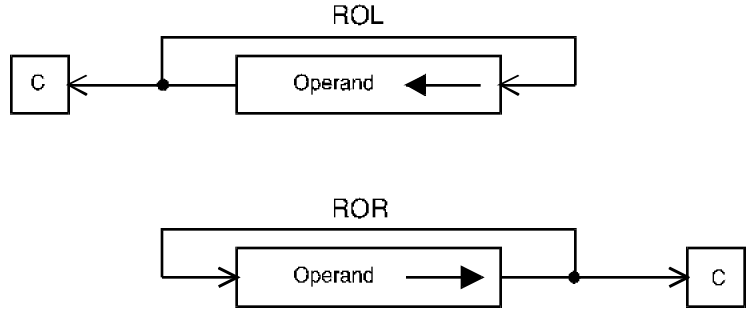
Operation: `[destination] ← [destination] rotated by <count>`

Syntax:
```
ROL Dx,Dy
ROR Dx,Dy
ROL #<data>,Dy
ROR #<data>,Dy
ROL <ea>
ROR <ea>
```

Attributes: Size = byte, word, longword

Description: Rotate the bits of the operand in the direction indicated. The extend bit, X, is not included in the operation. A rotate operation is circular in the sense that the bit shifted out at one end is shifted into the other end. That is, no bit is lost or destroyed by a rotate

operation. The bit shifted out is also copied into the C-bit of the CCR, but not into the X-bit. The shift count may be specified in one of three ways: the count may be a literal, the contents of a data register, or the value 1. An immediate count permits a shift of 1 to 8 places. If the count is in a register, the value is modulo 64, allowing a range of 0 to 63. If no count is specified, the *word* at the effective address is rotated by one place (e.g., `ROL <ea>`).



Condition codes:

X	N	Z	V	C
-	*	*	0	*

The X-bit is not affected and the C-bit is set to the last bit rotated out of the operand (C is set to zero if the shift count is 0).

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

ROXL, ROXR Rotate left/right with extend

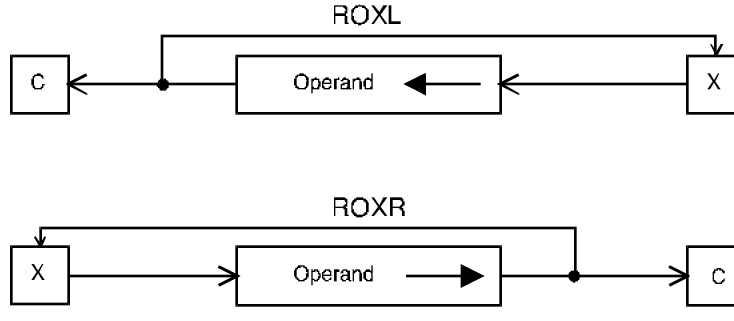
Operation: `[destination] ← [destination] rotated by <count>`

Syntax:

```
ROXL Dx,Dy
ROXR Dx,Dy
ROXL #<data>,Dy
ROXR #<data>,Dy
ROXL <ea>
ROXR <ea>
```

Attributes: Size = byte, word, longword

Description: Rotate the bits of the operand in the direction indicated. The extend bit of the CCR is included in the rotation. A rotate operation is circular in the sense that the bit shifted out at one end is shifted into the other end. That is, no bit is lost or destroyed by a rotate operation. Since the X-bit is included in the rotate, the rotation is performed over 9 bits (`.B`), 17 bits (`.W`), or 33 bits (`.L`). The bit shifted out is also copied into the C-bit of the CCR as well as the X-bit. The shift count may be specified in one of three ways: the count may be a literal, the contents of a data register, or the value 1. An immediate count permits a shift of 1 to 8 places. If the count is in a register, the value is modulo 64 and the range is from 0 to 63. If no count is specified, the word at the specified effective address is rotated by one place (i.e., `ROXL <ea>`).



Condition codes:

X	N	Z	V	C
*	*	*	0	*

The X- and the C-bit are set to the last bit rotated out of the operand. If the rotate count is zero, the X-bit is unaffected and the C-bit is set to the X-bit.

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

RTE Return from exception

Operation:

```
IF [S] = 1 THEN
                [SR] ← [M([SP])]; [SP] ← [SP] + 2
                [PC] ← [M([SP])]; [SP] ← [SP] + 4
              ELSE TRAP
```

Syntax: `RTE`

Attributes: Unsized

Description: The status register and program counter are pulled from the stack. The previous values of the SR and PC are lost. The `RTE` is used to terminate an exception handler. Note that the behavior of the `RTE` instruction depends on the nature of both the exception and processor type. The 68010 and later models push more information on the stack following an exception than the 68000. The processor determines how much to remove from the stack.

Condition codes:

```
X  N  Z  V  C
*  *  *  *  *
```

The CCR is restored to its pre-exception state.

RTR Return and restore condition codes

Operation: `[CCR] ← [M([SP])]; [SP] ← [SP] + 2`
`[PC] ← [M([SP])]; [SP] ← [SP] + 4`

Syntax: `RTR`

Attributes: Unsized

Description: The condition code and program counter are pulled from the stack. The previous condition code and program counter are lost. The supervisor portion of the status register is not affected.

Application: If you wish to preserve the CCR after entering a procedure, you can push it on the stack and then retrieve it with `RTR`.

```
        BSR     Proc1       Call the procedure
        .                   .
        .                   .
Proc1   MOVE.W  SR,-(SP)    Save old CCR on stack
        .                   .
        .                   Body of procedure
        .                   .
        RTR                 Return and restore CCR (not SR!)
```

Condition codes:

```
X  N  Z  V  C
*  *  *  *  *
```

The CCR is restored to its pre-exception state.

RTS Return from subroutine

Operation: [PC] ← [M([SP])]; [SP] ← [SP] + 4

Syntax: RTS

Attributes: Unsized

Description: The program counter is pulled from the stack and the previous value of the PC is lost. RTS is used to terminate a subroutine.

Condition codes:

X	N	Z	V	C
-	-	-	-	-

SBCD Subtract decimal with extend

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

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

Attributes: Size = byte

Description: Subtract the source operand from the destination operand together with the X-bit, and store the result in the destination. Subtraction is performed using BCD arithmetic. The only legal addressing modes are data register direct and memory to memory with address register indirect using auto-decrementing.

Condition codes:

X	N	Z	V	C
*	U	*	U	*

Z: Cleared if result is non-zero. Unchanged otherwise. The Z-bit can be used to test for zero after a chain of multiple precision operations.

Scc Set according to condition cc

Operation: IF cc = 1 THEN [destination] ← 11111111_2
ELSE [destination] ← 00000000_2

Syntax: Scc <ea>

Attributes: Size = byte

Description: The specified condition code is tested. If the condition is true, the bits at the effective address are all set to one (i.e., $FF). Otherwise, the bits at the effective address are set to zeros (i.e., $00).

SCC	set on carry clear	$\overline{C}$
SCS	set on carry set	C
SEQ	set on equal	Z
SGE	set on greater than or equal	$N.V + \overline{N}.\overline{V}$
SGT	set on greater than	$N.V.\overline{Z} + \overline{N}.\overline{V}.\overline{Z}$
SHI	set on higher than	$\overline{C}.\overline{Z}$
SLE	set on less than or equal	$Z + N.\overline{V} + \overline{N}.V$
SLS	set on lower than or same	C + Z
SLT	set on less than	$N.\overline{V} + \overline{N}.V$
SMI	set on minus (i.e., negative)	N
SNE	set on not equal	$\overline{Z}$
SPL	set on plus (i.e., positive)	$\overline{N}$
SVC	set on overflow clear	$\overline{V}$
SVS	set on overflow set	V
SF	set on false (i.e., set never)	0
ST	set on true (i.e., set always)	1

Condition codes:

X	N	Z	V	C
-	-	-	-	-

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

STOP Load status register and stop

Operation:

```
IF [S] = 1 THEN
              [SR] ← <data>
              STOP
         ELSE TRAP
```

Syntax:

```
STOP #<data>
```

Sample syntax:

```
STOP #$2700
STOP #SetUp
```

Attributes: Unsized

Description: The immediate operand is copied into the entire status register (i.e., both status byte and CCR are modified), and the program counter advanced to point to the next instruction to be executed. The processor then suspends all further processing and halts. That is, the privileged `STOP` instruction stops the 68000.

The execution of instructions resumes when a trace, an interrupt, or a reset exception occurs. A trace exception will occur if the trace bit is set when the `STOP` instruction is encountered. If an interrupt request arrives whose priority is higher than the current processor priority, an interrupt exception occurs, otherwise the interrupt request has no effect. If the bit of the immediate data corresponding to the S-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
*	*	*	*	*

Set according to the literal.

SUB Subtract binary

Operation: `[destination] ← [destination] - [source]`

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

Attributes: Size = byte, word, 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
*	*	*	*	*

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

SUBA Subtract address

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 operations are sign-extended to 32 bits prior to subtraction.

Condition codes:

X	N	Z	V	C
-	-	-	-	-

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

SUBI Subtract immediate

Operation: [destination] ← [destination] - <literal>

Syntax: SUBI #<data>,<ea>

Attributes: Size = byte, word, longword

Description: Subtract the immediate data from the destination operand. Store the result in the destination operand.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

SUBQ Subtract quick

Operation: `[destination] ← [destination] - <literal>`

Syntax: `SUBQ #<data>,<ea>`

Attributes: Size = byte, word, longword

Description: Subtract the immediate data from the destination operand. The immediate data must be in the range 1 to 8. Word and longword operations on address registers do not affect condition codes. A word operation on an address register affects the entire 32-bit address.

Condition codes:

X	N	Z	V	C
*	*	*	*	*

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔	✔	✔	✔	✔	✔	✔	✔	✔			

SUBX Subtract extended

Operation: `[destination] ← [destination] - [source] - [X]`

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

Attributes: Size = byte, word, longword

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

The only legal addressing modes are data register direct and memory to memory with address register indirect using auto-decrementing.

Condition codes:

```
X  N  Z  V  C
*  *  *  *  *
```

Z: Cleared if the result is non-zero, unchanged otherwise. The Z-bit can be used to test for zero after a chain of multiple precision operations.

SWAP Swap register halves

Operation: `[Register(16:31)] ← [Register(0:15)]; [Register(0:15)] ← [Register(16:31]`

Syntax: `SWAP Dn`

Attributes: Size = word

Description: Exchange the upper and lower 16-bit words of a data register.

Application: The `SWAP Dn` instruction enables the higher-order word in a register to take part in word operations by moving it into the lower-order position. `SWAP Dn` is effectively equivalent to `ROR.L Di,Dn`, where [Di] = 16. However, `SWAP` clears the C-bit of the CCR, whereas `ROR` sets it according to the last bit to be shifted into the carry bit.

Condition codes:

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

The N-bit is set if most-significant bit of the 32-bit result is set and cleared otherwise. The Z-bit is set if 32-bit result is zero and cleared otherwise.

TAS Test and set an operand

Operation: `[CCR] ← tested([operand]); [destination(7)] ← 1`

Syntax: `TAS <ea>`

Attributes: Size = byte

Description: Test and set the byte operand addressed by the effective address field. The N- and Z-bits of the CCR are updated accordingly. The

high-order bit of the operand (i.e., bit 7) is set. This operation is *indivisible* and uses a read-modify-write cycle. Its principal application is in multiprocessor systems.

Application: The `TAS` instruction permits one processor in a multiprocessor system to test a resource (e.g., shared memory) 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, and then sets the semaphore to claim the resource (if it was free). Because the operation is indivisible, no other processor can access the memory between the testing of the bit and its subsequent setting.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

TRAP Trap

Operation:

```
S ← 1;
[SSP] ← [SSP] - 4; [M([SSP])] ← [PC];
[SSP] ← [SSP] - 2; [M([SSP])] ← [SR];
[PC] ← vector
```

Syntax: `TRAP #<vector>`

Attributes: Unsized

Description: This instruction forces the processor to initiate exception processing. The vector number used by the `TRAP` instruction is in the range 0 to 15 and, therefore, supports 16 traps (i.e., `TRAP #0` to `TRAP #15`).

Application: The `TRAP` instruction is used to perform operating system calls and is system independent. That is, the effect of the call depends on the particular operating environment. For example, the University of Teesside 68000 simulator uses `TRAP #15` to perform

I/O. The ASCII character in D1.B is displayed by the following sequence.

```
MOVE.B  #6,D0  Set up the display a character parameter in D0
TRAP    #15    Now call the operating system
```

Condition codes:

X	N	Z	V	C
-	-	-	-	-

TRAPV Trap on overflow

Operation:
```
IF V = 1  THEN:
          [SSP] ← [SSP] - 4;  [M([SSP])] ← [PC];
          [SSP] ← [SSP] - 2;  [M([SSP])] ← [SR];
          [PC] ← [M($01C)]
          ELSE no action
```

Syntax: `TRAPV`

Attributes: Unsized

Description: If the V-bit in the CCR is set, then initiate exception processing. The exception vector is located at address $01C_{16}$. This instruction is used in arithmetic operations to call the operating system if overflow occurs.

Condition codes:

X	N	Z	V	C
-	-	-	-	-

TST Test an operand

Operation:
```
[CCR] ← tested([operand])
```
i.e., `[operand] - 0; update CCR`

Syntax: `TST <ea>`

Attributes: Size = byte, word, longword

Description: The operand is compared with zero. No result is saved, but the contents of the CCR are set according to the result. The effect of `TST <ea>` is the same as `CMPI #0,<ea>` except that the `CMPI` instruction also sets/clears the V- and C-bits of the CCR.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	

UNLK Unlink

Operation: [SP] ← [An]; [An] ← [M([SP])]; [SP] ← [SP] + 4

Syntax: UNLK An

Attributes: Unsized

Description: The stack pointer is loaded from the specified address register and the old contents of the stack pointer are lost (this has the effect of collapsing the stack frame). The address register is then loaded with the longword pulled off the stack.

Application: The UNLK instruction is used in conjunction with the LINK instruction. The LINK creates a stack frame at the start of a procedure, and the UNLK collapses the stack frame prior to a return from the procedure.

Condition codes:

X	N	Z	V	C
-	-	-	-	-