

**MC680x0**

**COLLABORATORS**

	<i>TITLE :</i> MC680x0		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		February 28, 2022	

**REVISION HISTORY**

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>MC680x0</b>	<b>1</b>
1.1	MC680x0 Reference 0.9, ©April 1995 by Flint/DARKNESS. . . . .	1
1.2	Move Instructions . . . . .	3
1.3	Mathematical Instructions . . . . .	3
1.4	Logic Instructions . . . . .	4
1.5	Flow Control Instructions . . . . .	5
1.6	Miscellaneous Instructions . . . . .	6
1.7	Shift and Rotate Instructions . . . . .	6
1.8	Bit Manipulation Instructions . . . . .	6
1.9	Alphabetical Index . . . . .	7
1.10	Add Binary Coded Decimal (w/extend) . . . . .	11
1.11	ADD integer . . . . .	12
1.12	ADD Address . . . . .	13
1.13	ADD Immediate . . . . .	13
1.14	ADD 3-bit immediate Quick . . . . .	14
1.15	ADD integer with eXtend . . . . .	15
1.16	Logical AND . . . . .	15
1.17	Logical AND Immediate . . . . .	16
1.18	Logical AND immediate to CCR . . . . .	17
1.19	Logical AND Immediate to SR (privileged) . . . . .	17
1.20	Arithmetic Shift Left and Arithmetic Shift Right . . . . .	18
1.21	Conditional branch . . . . .	18
1.22	Bit CHanGe . . . . .	19
1.23	Bit CLeaR . . . . .	20
1.24	Bit Field CHanGe . . . . .	21
1.25	Bit Field CLeaR . . . . .	21
1.26	Bit Field Signed EXTRACT . . . . .	21
1.27	Bit Field Unsigned EXTRACT . . . . .	22
1.28	Bit Field Find First One set . . . . .	22
1.29	Bit Field INSert . . . . .	22

---

1.30 Bit Field SET . . . . .	22
1.31 Bit Field TeST . . . . .	23
1.32 BreaK-PoinT . . . . .	23
1.33 Unconditional BRAnch . . . . .	24
1.34 Bit SET . . . . .	24
1.35 Branch to SubRoutine . . . . .	25
1.36 Bit TeST . . . . .	26
1.37 CALL Module . . . . .	26
1.38 Compare And Swap . . . . .	27
1.39 Compare And Swap (two-operand) . . . . .	27
1.40 CHecK bounds . . . . .	27
1.41 CHecK register against upper and lower bounds . . . . .	28
1.42 CLeaR . . . . .	28
1.43 CoMPare . . . . .	29
1.44 CoMPare register against upper and lower bounds . . . . .	30
1.45 CoMPare Address . . . . .	30
1.46 CoMPare Immediate . . . . .	31
1.47 CoMPare Memory . . . . .	31
1.48 Decrement and Branch Conditionally . . . . .	32
1.49 Signed DIVide . . . . .	34
1.50 Unsigned DIVide . . . . .	35
1.51 Exclusive logical OR . . . . .	36
1.52 Exclusive OR Immediate . . . . .	37
1.53 Exclusive OR Immediate to CCR . . . . .	37
1.54 Exclusive OR immediated with SR (privileged) . . . . .	38
1.55 Register EXchanGe . . . . .	38
1.56 Sign EXTend . . . . .	39
1.57 Illegal processor instruction . . . . .	39
1.58 Unconditional far JuMP . . . . .	40
1.59 Jump to far SubRoutine . . . . .	40
1.60 Load Effective Address . . . . .	41
1.61 Create local stack frame . . . . .	41
1.62 Logical Shift Left and Logical Shift Right . . . . .	42
1.63 Move Source -> Destination . . . . .	43
1.64 Move Address Source -> Destination . . . . .	43
1.65 CCR -> Destination . . . . .	44
1.66 Source -> CCR . . . . .	44
1.67 Move from SR (privileged) . . . . .	45
1.68 Move to SR (privileged) . . . . .	45

---

---

1.69 Move to/from USP (privileged) . . . . .	46
1.70 Move to/from control register . . . . .	46
1.71 MOVE Multiple registers . . . . .	47
1.72 MOVE Peripheral data . . . . .	48
1.73 MOVE signed 8-bit data Quick . . . . .	48
1.74 MOVE address Space (privileged) . . . . .	49
1.75 Signed and Unsigned multiply . . . . .	49
1.76 Negate Binary Coded Decimal with extend . . . . .	50
1.77 neg . . . . .	51
1.78 NEGate with eXtend . . . . .	51
1.79 No OPeration . . . . .	52
1.80 Logical complement . . . . .	52
1.81 Logical OR . . . . .	53
1.82 Logical OR Immediate . . . . .	53
1.83 Logical OR immediate to CCR . . . . .	54
1.84 Logical OR immediated to SR (privileged) . . . . .	55
1.85 PACK binary coded decimal . . . . .	55
1.86 Push Effective Address . . . . .	56
1.87 RESET external devices . . . . .	56
1.88 ROTate Left and ROTate Right . . . . .	56
1.89 ROTate Left with eXtend and ROTate Right with eXtend . . . . .	57
1.90 ReTURn and Deallocate parameter stack frame . . . . .	58
1.91 ReTURn from Exception (privileged) . . . . .	58
1.92 ReTURn from process Module . . . . .	59
1.93 ReTURn and Restore CCR . . . . .	59
1.94 ReTURn from Subroutine . . . . .	60
1.95 Subtract Binary Coded Decimal with extend . . . . .	60
1.96 Conditional Set . . . . .	61
1.97 Stop processor execution (privileged) . . . . .	62
1.98 SUBtract . . . . .	62
1.99 SUBtract Address . . . . .	63
1.100SUBtract Immediate . . . . .	63
1.101SUBtract 3-bit immediate Quick . . . . .	64
1.102SUBtract with eXtend . . . . .	65
1.103SWAP register upper and lower words . . . . .	66
1.104Test And Set operand . . . . .	66
1.105Initiate processor TRAP . . . . .	67
1.106Conditional trap . . . . .	67
1.107TeST operand for zero . . . . .	67

---

---

1.108Free stack frame created by LINK . . . . .	68
1.109Unpack binary coded decimal . . . . .	68
1.110Move Instruction Execution Times . . . . .	69
1.111Standard Instruction Execution Times . . . . .	70
1.112Immediate Instruction Execution Times . . . . .	71
1.113Single Operand Instruction Execution Times . . . . .	71
1.114Rotate Instruction Execution Times . . . . .	72
1.115Bit Manipulation Instruction Execution Times . . . . .	72
1.116Specification Instruction Execution Times . . . . .	73
1.117JMP, JSR, LEA, PEA and MOVEM Instruction Execution Times . . . . .	73
1.118Multi-Precision Instruction Execution Times . . . . .	74
1.119Miscellaneous Instruction Execution Times . . . . .	75
1.120Move Peripheral Instruction Execution Times . . . . .	75
1.121Exception Processing Execution Times . . . . .	76
1.122ASP68K PROJECT, Sixth Edition . . . . .	76
1.123Generic Assembler Documentation . . . . .	98

---

# Chapter 1

## MC680x0

### 1.1 MC680x0 Reference 0.9, ©April 1995 by Flint/DARKNESS.

MC 680x0 Reference 0.9

~~~~~  
 ©April 1995 by Flint/DARKNESS.

Generic Asm Doc

MC680x0 Optimizations

~~~~~

MC680x0 Instruction types:

~~~~~

Move Instructions

Mathematical Instructions

Logic Instructions

Flow Control Instructions

Shift and Rotate Instructions

Bit Manipulation Instructions

Miscellaneous Instructions

Alphabetical Index

MC68000 Instructions timing:

~~~~~

To calculate the timings of most 68000 instructions, you will need to first find the number of cycles used by the addressing mode in the table below ('Effective Address Operand Calculation Timing') and then the timing for the actual instruction in the appropriate table.

Move Instruction Execution Times

Standard Instruction Execution Times

Immediate Instruction Execution Times

Single Operand Instruction Execution Times

Rotate Instruction Execution Times

Bit Manipulation Instruction Execution Times

Specificational Instruction Execution Times

JMP, JSR, LEA, PEA and MOVEM Instruction Execution Times

Multi-Precision Instruction Execution Times

Miscellaneous Instruction Execution Times

Move Peripheral Instruction Execution Times

Exception Processing Execution Times

Effective Address Operand Calculation Timing

This table lists the number of clock periods required to compute an instruction's effective address. It includes fetching of any extension words, the address computation, and fetching of the memory operand. The number of bus read and write cycles is shown in parenthesis as (r/w). Note there are no write cycles involved in processing the effective address.

#### Effective Address Calculation Times

register            Byte,Word Long

Dn	data register direct	0 (0/0)	0 (0/0)
An	address register direct	0 (0/0)	0 (0/0)
memory			
(An)	address register indirect	4 (1/0)	8 (2/0)
(An)+	address register indirect with post-increment	4 (1/0)	8 (2/0)
-(An)	address register indirect with predec.	6 (1/0)	10 (2/0)
d(An)	address register indirect with displacement	8 (2/0)	12 (3/0)
d(An,ix)	address register indirect with index	10 (2/0)	14 (3/0)
xxx.W	absolute short	8 (2/0)	12 (3/0)
xxx.L	absolute long	12 (3/0)	16 (4/0)
d(PC)	program counter with displacement	8 (2/0)	12 (3/0)
d(PC,ix)	program counter with index	10 (2/0)	14 (3/0)
#xxx	immediate	4 (1/0)	8 (2/0)

The size of the index register (ix) does not affect execution time



## 1.2 Move Instructions

Normal:

MOVE

MOVEA

MOVEC

MOVEM

MOVEP

MOVEQ

MOVES

MOVE to CCR

MOVE from CCR

MOVE to SR

MOVE from SR

MOVE USP

Special:

LEA

PEA

## 1.3 Mathematical Instructions

Integer:

ADD

ADDI

ADDQ

ADDA

SUB

SUBI

SUBQ

SUBA

DIVS

DIVU  
MULS  
MULU  
EXT  
NEG  
NEGX  
CLR  
EXG  
SWAP  
CMP  
CMP2  
CMPI  
CMPA  
CHK  
CHK2  
CAS  
CAS2  
Multi-Precision Integer:  
ABCD  
ADDX  
SBCD  
SUBX  
NBCD  
CMPM  
PACK  
UNPK

## 1.4 Logic Instructions

---

Bit-wise:

AND

ANDI

ANDI to CCR

ANDI to SR

EOR

EORI

EORI to CCR

EORI to SR

OR

ORI

ORI to CCR

ORI to SR

Byte-wise:

TST

TAS

ScC

NOT

## 1.5 Flow Control Instructions

Subroutine:

BSR

JSR

RTS

RTD

RTR

Local:

BRA

JMP

Conditional:

---

Bcc  
DBcc  
System:  
ILLEGAL  
BKPT  
TRAP  
TRAPV  
RESET  
STOP  
RTE  
CALLM  
RTM

## 1.6 Miscellaneous Instructions

Stack Frame Maintenance:  
LINK  
UNLK  
Processor:  
NOP

## 1.7 Shift and Rotate Instructions

Shifts:  
ASd  
LSd  
Rotates:  
ROd  
ROXd

## 1.8 Bit Manipulation Instructions

---

Single Bit:

BCHG

BCLR

BSET

BTST

Bit Field:

BFCHG

BFCLR

BFSET

BFINS

BFEXTS

BFEXTU

BFFFO

BFTST

## 1.9 Alphabetical Index

ABCD

ADD

ADDA

ADDI

ADDQ

ADDX

AND

ANDI

ANDI CCR

ANDI SR

ASL

ASR

Bcc

---

BCHG  
BCLR  
BFCHG  
BFCLR  
BFEXTS  
BFEXTU  
BFFFO  
BFINS  
BFSET  
BFTST  
BKPT  
BRA  
BSET  
BSR  
BTST  
CALLM  
CAS  
CAS2  
CHK  
CHK2  
CLR  
CMP  
CMP2  
CMPA  
CMP I  
CMPM  
DBcc  
DIVS

---

---

DIVU  
EOR  
EORI  
EORI CCR  
EORI SR  
EXG  
EXT  
ILLEGAL  
JMP  
JSR  
LEA  
LINK  
LSL  
LSR  
MOVE  
MOVEA  
MOVE from CCR  
MOVE to CCR  
MOVE from SR  
MOVE to SR  
MOVE from/to USP  
MOVEC  
MOVEM  
MOVEP  
MOVEQ  
MOVES  
MUL  
NBCD  
NEG

---

---

NEGX  
NOP  
NOT  
OR  
ORI  
ORI CCR  
ORI SR  
PACK  
PEA  
RESET  
ROL  
ROR  
ROXL  
ROXR  
RTD  
RTE  
RTM  
RTR  
RTS  
SBCD  
Scc  
STOP  
SUB  
SUBA  
SUBI  
SUBQ  
SUBX  
SWAP

---



TAS  
TRAP  
TRAPcc  
TRAPV  
TST  
UNLK  
UNPK

## 1.10 Add Binary Coded Decimal (w/extend)

### NAME

ABCD -- Add binary coded decimal

### SYNOPSIS

ABCD Dy, Dx  
ABCD -(Ay), -(Ax)

Size = (Byte)

### FUNCTION

Adds the source operand to the destination operand along with the extend bit, and stores the result in the destination location. The addition is performed using binary coded decimal arithmetic. The operands, which are packed BCD numbers, can be addressed in two different ways:

1. Data register to data register: The operands are contained in the data registers specified in the instruction.
2. Memory to memory: The operands are addressed with the predecrement addressing mode using the address registers specified in the instruction.

This operation is a byte operation only.

Normally the Z condition code bit is set via programming before the start of an operation. That allows successful tests for zero results upon completion of multiple-precision operations.

### RESULT

X - Set the same as the carry bit.  
N - Undefined  
Z - Cleared if the result is non-zero. Unchanged otherwise.  
V - Undefined  
C - Set if a decimal carry was generated. Cleared otherwise.

### SEE ALSO

ADD

---

ADDI  
ADDQ  
ADDX  
SUB  
SUBI  
SUBQ  
SBCD  
SUBX

## 1.11 ADD integer

NAME

ADD -- Add integer

SYNOPSIS

ADD <ea>,Dn

ADD Dn,<ea>

Size = (Byte, Word, Long)

FUNCTION

Adds the source operand to the destination operand using binary addition, and stores the result in the destination location. The size of the operation may be specified as byte, word, or long. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

RESULT

X - Set the same as the carry bit.  
N - Set if the result is negative. Cleared otherwise.  
Z - Set if the result is zero. Cleared otherwise.  
V - Set if an overflow is generated. Cleared otherwise.  
C - Set if a carry is generated. Cleared otherwise.

SEE ALSO

ADDI  
ADDQ  
ADDX  
SUB  
SUBI  
SUBQ

SUBX

## 1.12 ADD Address

NAME  
ADDA -- Add address

### SYNOPSIS

ADDA <ea>,An

Size = (Word, Long)

### FUNCTION

Adds the source operand to the destination address register, and stores the result in the destination address register. The size of the operation may be specified as word or long. The entire destination operand is used regardless of the operation size.

### RESULT

None.

### SEE ALSO

ADDQ

SUBQ

SUBA

## 1.13 ADD Immediate

NAME  
ADDI -- Add immediate

### SYNOPSIS

ADDI #<data>,<ea>

Size = (Byte, Word, Long)

### FUNCTION

Adds the immediate data to the destination operand, and stores the result in the destination location. The size of the operation may be specified as byte, word, or long. The size of the immediate data matches the operation size.

### RESULT

X - Set the same as the carry bit.  
N - Set if the result is negative. Cleared otherwise.  
Z - Set if the result is zero. Cleared otherwise.  
V - Set if an overflow is generated. Cleared otherwise.  
C - Set if a carry is generated. Cleared otherwise.

---

SEE ALSO

ADD

ADDQ

ADDX

SUB

SUBI

SUBQ

SUBX

## 1.14 ADD 3-bit immediate Quick

NAME

ADDQ -- Add 3-bit immediate quick

SYNOPSIS

ADDQ #<data>, <ea>

Size = (Byte, Word, Long)

FUNCTION

Adds the immediate value of 1 to 8 to the operand at the destination location. The size of the operation may be specified as byte, word, or long. When adding to address registers, the condition codes are not altered, and the entire destination address register is used regardless of the operation size.

RESULT

X - Set the same as the carry bit.

N - Set if the result is negative. Cleared otherwise.

Z - Set if the result is zero. Cleared otherwise.

V - Set if an overflow is generated. Cleared otherwise.

C - Set if a carry is generated. Cleared otherwise.

SEE ALSO

ADD

ADDI

SUB

SUBI

SUBQ

---

## 1.15 ADD integer with eXtend

NAME

ADDX -- Add integer with extend

SYNOPSIS

ADDX Dy,Dx

ADDX -(Ay),-(Ax)

Size = (Byte, Word, Long)

FUNCTION

Adds the source operand to the destination operand along with the extend bit, and stores the result in the destination location. The addition is performed using binary coded decimal arithmetic. The operands, which are packed BCD numbers, can be addressed in two different ways:

1. Data register to data register: The operands are contained in the data registers specified in the instruction.
2. Memory to memory: The operands are addressed with the predecrement addressing mode using the address registers specified in the instruction.

The size of operation can be specified as byte, word, or long.

Normally the Z condition code bit is set via programming before the start of an operation. That allows successful tests for zero results upon completion of multiple-precision operations.

RESULT

X - Set the same as the carry bit.

N - Set if the result is negative. Cleared otherwise.

Z - Cleared if the result is non-zero. Unchanged otherwise.

V - Set if an overflow is generated. Cleared otherwise.

C - Set if a carry is generated. Cleared otherwise.

SEE ALSO

ADD

ADDI

SUB

SUBI

SUBX

## 1.16 Logical AND

NAME

AND -- Logical AND

---

## SYNOPSIS

AND <ea>,Dn  
AND Dn,<ea>

Size = (Byte, Word, Long)

## FUNCTION

Performs a bit-wise AND operation with the source operand and the destination operand and stores the result in the destination. The size of the operation can be specified as byte, word, or long. The contents of an address register may not be used as an operand.

## RESULT

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

## SEE ALSO

ANDI

## 1.17 Logical AND Immediate

## NAME

ANDI -- Logical AND immediate

## SYNOPSIS

ANDI #<data>,<ea>

Size = (Byte, Word, Long)

## FUNCTION

Performs a bit-wise AND operation with the immediate data and the destination operand and stores the result in the destination. The size of the operation can be specified as byte, word, or long. The size of the immediate data matches the operation size.

## RESULT

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

## SEE ALSO

AND

ANDI to CCR

---

ANDI to SR

## 1.18 Logical AND immediate to CCR

NAME

ANDI to CCR -- Logical AND immediate to condition code register

SYNOPSIS

ANDI #<data>,CCR

Size = (Byte)

FUNCTION

Performs a bit-wise AND operation with the immediate data and the lower byte of the status register.

RESULT

X - Cleared if bit 4 of immed. operand is zero. Unchaned otherwise.  
N - Cleared if bit 3 of immed. operand is zero. Unchaned otherwise.  
Z - Cleared if bit 2 of immed. operand is zero. Unchaned otherwise.  
V - Cleared if bit 1 of immed. operand is zero. Unchaned otherwise.  
C - Cleared if bit 0 of immed. operand is zero. Unchaned otherwise.

SEE ALSO

AND

ANDI

ANDI to SR

## 1.19 Logical AND Immediate to SR (privileged)

NAME

ANDI to SR -- Logical AND immediate to status register (privileged)

SYNOPSIS

ANDI #<data>,SR

Size = (Word)

FUNCTION

Performs a bit-wise AND operation with the immediate data and the status register. All implemented bits of the status register are affected.

RESULT

X - Cleared if bit 4 of immed. operand is zero. Unchaned otherwise.  
N - Cleared if bit 3 of immed. operand is zero. Unchaned otherwise.  
Z - Cleared if bit 2 of immed. operand is zero. Unchaned otherwise.  
V - Cleared if bit 1 of immed. operand is zero. Unchaned otherwise.  
C - Cleared if bit 0 of immed. operand is zero. Unchaned otherwise.

---

SEE ALSO

AND

ANDI

ANDI to CCR

## 1.20 Arithmetic Shift Left and Arithmetic Shift Right

NAME

ASL, ASR -- Arithmetic shift left and arithmetic shift right

SYNOPSIS

ASd Dx,Dy

ASd #<data>,Dy

ASd <ea>

where d is direction, L or R

Size = (Byte, Word, Long)

FUNCTION

Performs an arithmetic shifting bit operation in the indicated direction, with an immediate data, or with a data register. If you shift address contents, you only can do ONE shift, and your operand is ONE word exclusively.

RESULT

X - Set according to the list bit shifted out of the operand. Unaffected for a shift count of zero.

N - Set if the most-significant bit of the result is set. Cleared otherwise.

Z - Set if the result is zero. Cleared otherwise.

V - Set if the most significant bit is changed at any time during the shift operation. Cleared otherwise.

C - Set according to the list bit shifted out of the operand. Cleared for a shift count of zero.

SEE ALSO

ROd

ROXd

## 1.21 Conditional branch

NAME

Bcc -- Conditional branch

SYNOPSIS

Bcc <label>

---



Size = (Byte, Word)

#### FUNCTION

If condition true then program execution continues at:  
(PC) + displacement.

PC value is instruction address more two.

Displacement is the relative value in bytes which separate  
Bcc instruction of mentioned label.

Condition code 'cc' specifies one of the following:

CC	carry clear	C'	LS	low or same	C+Z
CS	carry set	C	LT	less than	N'·V+N'V
EQ	equal	Z	MI	minus	N
GE	greater or equal	N·V+N'V'	NE	not equal	Z'
GT	greater than	N·V·Z'+N'V'·Z'	PL	plus	N'
HI	high	C'·Z'	VC	overflow clear	V'
LE	less or equal	Z+N'·V+N'V	VS	overflow set	V

#### RESULT

None.

#### SEE ALSO

BRA

DBcc

Scc

## 1.22 Bit CHanGe

#### NAME

BCHG -- Bit change

#### SYNOPSIS

BCHG Dn, <ea>

BCHG #<data>, <ea>

Size = (Byte, Long)

#### FUNCTION

Tests a bit in the destination operand and sets the Z  
condition code appropriately, then inverts the bit in the destination.  
If the destination is a data register, any of the 32 bits can be  
specifice by the modulo 32 number. When the distination is a memory  
location, the operation must be a byte operation, and therefore the  
bit number is modulo 8. In all cases, bit zero is the least  
significant bit. The bit number for this operation may be specified  
in either of two ways:

1. Immediate -- The bit number is specified as immediate data.
2. Register -- The specified data register contains the bit number.

#### RESULT

X - not affected  
N - not affected  
Z - Set if the bit tested is zero. Cleared otherwise.  
V - not affected  
C - not affected

SEE ALSO

BCLR  
BSET  
BTST  
EOR  
BFCHG

## 1.23 Bit CLear

NAME

BCLR -- Bit clear

SYNOPSIS

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

Size = (Byte, Long)

FUNCTION

Tests a bit in the destination operand and sets the Z condition code appropriately, then clears the bit in the destination. If the destination is a data register, any of the 32 bits can be specified by the modulo 32 number. When the destination is a memory location, the operation must be a byte operation, and therefore the bit number is modulo 8. In all cases, bit zero is the least significant bit. The bit number for this operation may be specified in either of two ways:

1. Immediate -- The bit number is specified as immediate data.
2. Register -- The specified data register contains the bit number.

RESULT

X - not affected  
N - not affected  
Z - Set if the bit tested is zero. Cleared otherwise.  
V - not affected  
C - not affected

SEE ALSO

BCHG  
BSET

---

BTST  
AND  
BFCLR

## 1.24 Bit Field CHanGe

### NAME

BFCHG -- Bit field change

### SYNOPSIS

BFCHG (X) <ea> (OFFSET : WIDTH) (68020+)

### FUNCTION

### RESULT

### SEE ALSO

## 1.25 Bit Field CLearR

### NAME

BFCLR -- Bit field clear

### SYNOPSIS

BFCLR (X) <ea> (OFFSET : WIDTH) (68020+)

### FUNCTION

### RESULT

### SEE ALSO

## 1.26 Bit Field Signed EXTract

### NAME

BFEXTS -- Bit field signed extract

### SYNOPSIS

BFEXTS (X) <ea> (OFFSET : WIDTH) (68020+)

### FUNCTION

### RESULT

### SEE ALSO

---

## 1.27 Bit Field Unsigned EXtract

### NAME

BFEXTU -- Bit field unsigned extract"

### SYNOPSIS

BFEXTU (X) <ea> (OFFSET : WIDTH) (68020+)

### FUNCTION

### RESULT

### SEE ALSO

## 1.28 Bit Field Find First One set

### NAME

BFFFO -- Bit field find first one set

### SYNOPSIS

BFFFO (X) <ea> (OFFSET : WIDTH) (68020+)

### FUNCTION

### RESULT

### SEE ALSO

## 1.29 Bit Field INSert

### NAME

BFINS -- Bit field insert

### SYNOPSIS

BFINS (X) <ea> (OFFSET : WIDTH) (68020+)

### FUNCTION

### RESULT

### SEE ALSO

## 1.30 Bit Field SET

### NAME

BFSET -- Bit field set

### SYNOPSIS

BFSET (X) <ea> (OFFSET : WIDTH) (68020+)

---

FUNCTION

RESULT

SEE ALSO

### 1.31 Bit Field TeST

NAME

BFTST -- Bit field test

SYNOPSIS

BFTST (X) <ea> (OFFSET : WIDTH) (68020+)

FUNCTION

RESULT

SEE ALSO

### 1.32 Break-Point

NAME

BKPT -- Break-point

SYNOPSIS

BKPT #<data>

FUNCTION

This instruction is used to support the program breakpoint function for debug monitors and real-time hardware emulators, and the operation will be dependent on the implementation. Execution of this instruction will cause the MC68010 to run a breakpoint acknowledge bus cycle and zeros on all address lines, but an MC68020 will place the immediate data on lines A2, A3, and A4, and zeros on lines A0 and A1.

Whether the breakpoint acknowledge cycle is terminated with (DTACK)', (BERR)', or (VPA)' the processor always takes an illegal instruction exception. During exception processing, a debug monitor can distinguish eight different software breakpoints by decoding the field in the BKPT instruction.

For the MC68000 and the MC68HC000, this instruction causes an illegal instruction exception, but does not run the breakpoint acknowledge bus cycle.

There are two possible responses on an MC68020: normal and exception. The normal response is an operation word (typically the instruction the BKPT originally replaced) on the data lines with the (DSACKx)' signal asserted. The operation word is the executed in place of the

breakpoint instruction.

For the exception response, a bus error signal will cause the MC68020 to take an illegal instruction exception, just as an MC68010 or MC68000 would do.

RESULT

None.

SEE ALSO

ILLEGAL

### 1.33 Unconditional BRAnch

NAME

BRA -- Unconditional branch

SYNOPSIS

BRA <label>

Size = (Byte, Word)

Size = (Byte, Word, Long) (68020+)

FUNCTION

Program execution continues at location (PC) + displacement.

The PC contains the address of the instruction word of the BRA instruction plus two. The displacement is a two's complement integer that represents the relative distance in bytes from the current PC to the destination PC.

RESULT

None.

SEE ALSO

JMP

Bcc

### 1.34 Bit SET

NAME

BSET -- Bit set

SYNOPSIS

BSET Dn, <ea>

BSET #<data>, <ea>

Size = (Byte, Long)

FUNCTION

---

Tests a bit in the destination operand and sets the Z condition code appropriately, then sets the bit in the destination. If the destination is a data register, any of the 32 bits can be specified by the modulo 32 number. When the destination is a memory location, the operation must be a byte operation, and therefore the bit number is modulo 8. In all cases, bit zero is the least significant bit. The bit number for this operation may be specified in either of two ways:

1. Immediate -- The bit number is specified as immediate data.
2. Register -- The specified data register contains the bit number.

#### RESULT

X - not affected  
N - not affected  
Z - Set if the bit tested is zero. Cleared otherwise.  
V - not affected  
C - not affected

#### SEE ALSO

BCHG  
  
BCLR  
  
BTST  
  
OR  
  
BFSET  
  
BFINS

## 1.35 Branch to SubRoutine

#### NAME

BSR -- Branch to subroutine

#### SYNOPSIS

BSR <label>

Size = (Byte, Word)  
Size = (Byte, Word, Long) (68020+)

#### FUNCTION

Pushes the long word address of the instruction immediately following the BSR instruction onto the system stack. The PC contains the address of the instruction word plus two. Program execution continues at location (PC) + displacement.

#### RESULT

None.

#### SEE ALSO

---

JSR

BRA

RTS

RTD

RTR

## 1.36 Bit TeST

NAME

BTST -- Bit test

SYNOPSIS

BTST Dn, <ea>

BTST #<data>, <ea>

Size = (Byte, Long)

FUNCTION

Tests a bit in the destination operand and sets the Z condition code appropriately. If the destination is a data register, any of the 32 bits can be specified by the modulo 32 number. When the destination is a memory location, the operation must be a byte operation, and therefore the bit number is modulo 8. In all cases, bit zero is the least significant bit. The bit number for this operation may be specified in either of two ways:

1. Immediate -- The bit number is specified as immediate data.
2. Register -- The specified data register contains the bit number.

RESULT

X - not affected

N - not affected

Z - Set if the bit tested is zero. Cleared otherwise.

V - not affected

C - not affected

SEE ALSO

BFTST

BFFFO

## 1.37 CALL Module

NAME

CALLM -- Call module (68020+)

SYNOPSIS

---



FUNCTION

RESULT

SEE ALSO

### 1.38 Compare And Swap

NAME

CAS -- Compare and swap (68020+)

SYNOPSIS

FUNCTION

RESULT

SEE ALSO

### 1.39 Compare And Swap (two-operand)

NAME

CAS2 --Compare and swap (two-operand) (68020+)

SYNOPSIS

FUNCTION

RESULT

SEE ALSO

### 1.40 CHeck bounds

NAME

CHK -- Check bounds

SYNOPSIS

CHK <ea>,Dn

Size = (Word)

FUNCTION

Compares the value in the data register specified to zero and to the upper bound. The upper bound is a twos complement integer. If the register value is less than zero or greater than the upper bound, a CHK instruction, vector number 6, occurs.

RESULT

---

X - Not affected  
N - Set if Dn < 0; cleared if Dn > <ea>. Undefined otherwise.  
Z - Undefined.  
V - Undefined.  
C - Undefined.

SEE ALSO

CMP

CMPI

CMPA

CHK2

## 1.41 CHecK register against upper and lower bounds

NAME

CHK2 -- Check register against upper and lower bounds (68020+)

SYNOPSIS

CHK2

Size = ()

FUNCTION

RESULT

X -

N -

Z -

V -

C -

SEE ALSO

CMP

CMPI

CMPA

CHK

## 1.42 CLeaR

NAME

CLR -- Clear

SYNOPSIS

CLR <ea>

---

Size = (Byte, Word, Long)

#### FUNCTION

Clears the destination operand to zero.

On an MC68000 and MC68HC000, a CLR instruction does both a read and a write to the destination. Because of this, this instruction should never be used on custom chip registers.

#### RESULT

X - Not affected  
N - Always cleared  
Z - Always set  
V - Always cleared  
C - Always cleared

#### SEE ALSO

MOVE  
MOVEQ  
BCLR  
BFCLR

## 1.43 CoMPare

#### NAME

CMP -- Compare

#### SYNOPSIS

CMP <ea>,Dn

Size = (Byte, Word, Long)

#### FUNCTION

Subtracts the source operand from the destination data register and sets the condition codes according to the result. The data register is NOT changed.

#### RESULT

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

#### SEE ALSO

CMPI  
CMPA

---

CMPM

CMP2

TST

CHK

CHK2

## 1.44 CoMPare register against upper and lower bounds

NAME

Cmp2 -- Compare register against upper and lower bounds (68020+)

SYNOPSIS

FUNCTION

RESULT

SEE ALSO

## 1.45 CoMPare Address

NAME

CMPA -- Compare address

SYNOPSIS

CMPA <ea>,An

Size = (Word, Long)

FUNCTION

Subtracts the source operand from the destination address register and sets the condition codes according to the result. The address register is NOT changed. Word sized source operands are sign extended to long for comparison.

RESULT

X - Not affected

N - Set if the result is negative. Cleared otherwise.

Z - Set if the result is zero. Cleared otherwise.

V - Set if an overflow occurs. Cleared otherwise.

C - Set if a borrow occurs. Cleared otherwise.

SEE ALSO

CMP

CMPI

---

CMP2

## 1.46 CoMPare Immediate

NAME

CMPI -- Compare immediate

SYNOPSIS

CMP #<data>, <ea>

Size = (Byte, Word, Long)

FUNCTION

Subtracts the source operand from the destination operand and sets the condition codes according to the result. The destination is NOT changed. The size of the immediate data matches the operation size.

RESULT

X - Not affected

N - Set if the result is negative. Cleared otherwise.

Z - Set if the result is zero. Cleared otherwise.

V - Set if an overflow occurs. Cleared otherwise.

C - Set if a borrow occurs. Cleared otherwise.

SEE ALSO

CMP

CMPA

CMPM

CMP2

TST

CHK

CHK2

## 1.47 CoMPare Memory

NAME

CMPM -- Compare memory

SYNOPSIS

CMPM (Ay)+, (Ax)+

Size = (Byte, Word, Long)

FUNCTION

---

Subtracts the source operand from the destination operand and sets the condition codes according to the result. The destination operand is NOT changed. Operands are always addressed with the postincrement mode.

#### RESULT

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

#### SEE ALSO

CMP

CMPI

CMPA

CMP2

TST

CHK

CHK2

## 1.48 Decrement and Branch Conditionally

#### NAME

DBcc -- Decrement and branch conditionally

#### SYNOPSIS

DBcc Dn,<label>

Size = (Word)

#### FUNCTION

Controls a loop of instructions. The parameters are: a condition code, a data register (counter), and a displacement value. The instruction first tests the condition (for termination); if it is true, no operation is performed. If the termination condition is not true, the low-order 16 bits of the counter are decremented by one. If the result is -1, execution continues at the next instruction, otherwise, execution continues at the specified address.

Condition code 'cc' specifies one of the following:

CC	carry clear	C'	LS low or same	C+Z
CS	carry set	C	LT less than	N'·V+N'V
EQ	equal	Z	MI minus	N
GE	greater or equal	N·V+N'V'	NE not equal	Z'
GT	greater than	N·V·Z'+N'V'·Z'	PL plus	N'

```

HI high      C'·Z'   VC overflow clear  V'
LE less or equal Z+N'·V+N'V  VS overflow set  V
T True              1                F False              0

```

Keep the following in mind when using DBcc instructions:

1. A DBcc acts as the UNTIL loop construct in high level languages. E.g., DBMI would be "decrement and branch until minus".
2. Most assemblers accept DBRA or DBF for use when no condition is required for termination of a loop.

The DBcc will, unlike the Bcc instruction, take the branch only if the set of conditions are NOT satisfied. The loop will be terminated if the condition is true, or the counter is zero BEFORE a decrement, and wrap to -1. This means that if you execute code like:

```
move.w #30,d0
```

```
.Loop
move.l (a0)+,(a1)+
dbf d0,.Loop
```

then the copy will be executed \*31\* times, and 124 bytes of memory will be copied, not 120.

A good practice is therefore to write:

```
move.w #31-1,d0
```

```
.Loop
move.l (a0)+,(a1)+
dbf d0,.Loop
```

To compare two strings that may be in excess of 64k length for being equal, you could execute the following code:

...

```

move.l #$53452-1,d0
beq.s .NoLength ; Zero length!
bra.s .StartOuterLoop ; The upper word contain count of 65536's...

```

```
.OuterLoop
swap d0
```

```
.InnerLoop
cmp.b (a0)+,(a1)+
dbne d0,.InnerLoop ; Remember, drop through on condition TRUE.
```

```
.StartOuterLoop ; d0 will be $FFFF on 2.+ run-through
bne.s .NotEqual ; Dropped through due to Not Equal?
swap d0 ; Get upper part of word...
dbf d0,.OuterLoop
...
```

It would not be possible to use two sets of DBNEs, as SWAP

changes the condition codes - and we don't want the drop-though to be on account of D0, instead of the compares...

Another neat trick is to use instruction as a conditional decrementer; this code will decrement d0.w if the last instruction returned NOT EQUAL:

```
...
dbeq d0,.Next
```

```
.Next
...
```

RESULT  
None.

SEE ALSO

Bcc

Sc

## 1.49 Signed DIVide

NAME

DIVS, DIVSL -- Signed divide

SYNOPSIS

DIVS.W	<ea>,Dn	32/16	->	16r:16q	
DIVS.L	<ea>,Dq	32/32	->	32q	(68020+)
DIVS.L	<ea>,Dr:Dq	64/32	->	32r:32q	(68020+)
DIVSL.L	<ea>,Dr:Dq	32/32	->	32r:32q	(68020+)

Size = (Word, Long)

FUNCTION

Divides the signed destination operand by the signed source operand and stores the signed result in the destination.

The instruction has a word form and three long forms. For the word form, the destination operand is a long word and the source operand is a word. The resultant quotient is placed in the lower word of the destination and the resultant remainder is placed in the upper word of the destination. The sign of the remainder is the same as the sign of the dividend.

In the first long form, the destination and the source are both long words. The quotient is placed in the longword of the destination and the remainder is discarded.

The second long form has the destination as a quadword (eight bytes), specified by any two data registers, and the source is a long word. The resultant remainder and quotient are both long words and are placed in the destination registers.



The final long form has both the source and the destination as long words and the resultant quotient and remainder as long words.

#### RESULT

- X - Not affected
- N - Set if the quotient is negative, cleared otherwise. Undefined if overflow or divide by zero occurs.
- Z - Set if the quotient is zero, cleared otherwise. Undefined if overflow or divide by zero occurs.
- V - Set if overflow occurs, cleared otherwise. Undefined if divide by zero occurs.
- C - Always cleared.

#### Notes:

1. If divide by zero occurs, an exception occurs.
2. If overflow occurs, neither operand is affected.

#### SEE ALSO

DIVU  
MULS MULU

## 1.50 Unsigned DIVIDE

#### NAME

DIVU, DIVUL -- Unsigned divide

#### SYNOPSIS

```

DIVU.W <ea>,Dn      32/16 -> 16r:16q
DIVU.L <ea>,Dq      32/32 -> 32q      (68020+)
DIVU.L <ea>,Dr:Dq   64/32 -> 32r:32q   (68020+)
DIVUL.L <ea>,Dr:Dq  32/32 -> 32r:32q   (68020+)

```

Size = (Word, Long)

#### FUNCTION

Divides the unsigned destination operand by the unsigned source operand and stores the unsigned result in the destination.

The instruction has a word form and three long forms. For the word form, the destination operand is a long word and the source operand is a word. The resultant quotient is placed in the lower word of the destination and the resultant remainder is placed in the upper word of the destination. The sign of the remainder is the same as the sign of the dividend.

In the first long form, the destination and the source are both long words. The quotient is placed in the longword of the destination and the remainder is discarded.

The second long form has the destination as a quadword (eight bytes), specified by any two data registers, and the source is a long word. The resultant remainder and quotient are both long words and are placed in the destination registers.

The final long form has both the source and the destination as long words and the resultant quotient and remainder as long words.

#### RESULT

X - Not affected  
N - See below.  
Z - Set if the quotient is zero, cleared otherwise. Undefined if overflow or divide by zero occurs.  
V - Set if overflow occurs, cleared otherwise. Undefined if divide by zero occurs.  
C - Always cleared.

#### Notes:

1. If divide by zero occurs, an exception occurs.
2. If overflow occurs, neither operand is affected.

According to the Motorola data books, the N flag is set if the quotient is negative, but in an unsigned divide, this seems to be impossible.

#### SEE ALSO

DIVS  
MULS MULU

## 1.51 Exclusive logical OR

#### NAME

EOR -- Exclusive logical OR

#### SYNOPSIS

EOR Dn, <ea>

Size = (Byte, Word, Long)

#### FUNCTION

Performs an exclusive OR operation on the destination operand with the source operand.

#### RESULT

X - Not Affected  
N - Set to the value of the most significant bit.  
Z - Set if the result is zero.  
V - Always cleared  
C - Always cleared

#### SEE ALSO

EORI  
BCHG

---

## 1.52 Exclusive OR Immediate

NAME

EORI -- Exclusive OR immediate

SYNOPSIS

EORI #<data>,<ea>

Size = (Byte, Word, Long)

FUNCTION

Performs an exclusive OR operation on the destination operand with the source operand.

RESULT

X - Not Affected

N - Set to the value of the most significant bit.

Z - Set if the result is zero.

V - Always cleared

C - Always cleared

SEE ALSO

EOR

EORI to CCR

EORI to SR

BCHG

## 1.53 Exclusive OR Immediate to CCR

NAME

EORI to CCR -- Exclusive OR immediate to the condition code register

SYNOPSIS

EORI #<data>,CCR

Size = (Byte)

FUNCTION

Performs an exclusive OR operation on the condition codes register with the source operand.

RESULT

X - Changed if bit 4 of the source is set, cleared otherwise.

N - Changed if bit 3 of the source is set, cleared otherwise.

Z - Changed if bit 2 of the source is set, cleared otherwise.

V - Changed if bit 1 of the source is set, cleared otherwise.

C - Changed if bit 0 of the source is set, cleared otherwise.

SEE ALSO

---

EOR

EORI

EORI to SR

## 1.54 Exclusive OR immediated with SR (privileged)

NAME

EORI to SR -- Exclusive OR immediated to the status register (privileged)

SYNOPSIS

EORI #<data>,SR

Size = (Word)

FUNCTION

Performs an exclusive OR operation on the status register with the source operand.

RESULT

X - Changed if bit 4 of the source is set, cleared otherwise.

N - Changed if bit 3 of the source is set, cleared otherwise.

Z - Changed if bit 2 of the source is set, cleared otherwise.

V - Changed if bit 1 of the source is set, cleared otherwise.

C - Changed if bit 0 of the source is set, cleared otherwise.

SEE ALSO

EOR

EORI

EORI to CCR

## 1.55 Register EXchanGe

NAME

EXG -- Register exchange

SYNOPSIS

EXG Rx,Ry

Size = (Long)

FUNCTION

Exchanges the contents of any two registers

RESULT

None.

SEE ALSO

---

SWAP

## 1.56 Sign EXTend

NAME

EXT, EXTB -- Sign extend

SYNOPSIS

EXT.W Dn Extend byte to word  
EXT.L Dn Extend word to long word  
EXTB.L Dn Extend byte to long word (68020+)

Size = (Word, Long)

FUNCTION

Extends a byte to a word, or a word to a long word in a data register by copying the sign bit through the upper bits. If the operation is from byte to word, bit 7 is copied to bits 8 through 15. If the operation is from word to long word, bit 15 is copied to bits 16 through 31. The EXTB copies bit 7 to bits 8 through 31.

RESULT

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

SEE ALSO

BFEXTS

BFEXTU

## 1.57 Illegal processor instruction

NAME

ILLEGAL -- Illegal processor instruction

SYNOPSIS

ILLEGAL

FUNCTION

This instruction forces an Illegal Instruction exception, vector number 4. All other illegal instruction bit patterns, including, but not limited to, \$fxxx and \$axxx, are reserved for future expansion.

The Hexadecimal code of this instruction is: \$4AFC

RESULT

None.

---

SEE ALSO

BKPT

## 1.58 Unconditional far JuMP

NAME

JMP -- Unconditional far jump

SYNOPSIS

JMP <ea>

FUNCTION

Program execution continues at the address specified by the operand.

RESULT

None.

SEE ALSO

BRA

JSR

## 1.59 Jump to far SubRoutine

NAME

JSR -- Jump to far subroutine

SYNOPSIS

JSR <ea>

FUNCTION

Pushes the long word address of the instruction immediately following the JSR instruction onto the stack. The PC contains the address of the instruction word plus two. Program execution continues at location specified by <ea>.

RESULT

None.

SEE ALSO

BSR

BRA

RTS

RTD

---

RTR

## 1.60 Load Effective Address

NAME

LEA -- Load effective address

SYNOPSIS

LEA <ea>,An

Size = (Long)

FUNCTION

Places the specified address into the destination address register. Note: All 32 bits of An are affected by this instruction.

RESULT

None.

SEE ALSO

MOVEA

ADDA

SUBA

## 1.61 Create local stack frame

NAME

LINK -- Create local stack frame

SYNOPSIS

LINK An, #<data>

Size = (Word)

Size = (Word, Long) (68020+)

FUNCTION

This instruction saves the specified address register onto the stack, then places the new stack pointer in that register. It then adds the specified immediate data to the stack pointer. To allocate space on the stack for a local data area, a negative value should be used for the second operand.

The use of a local stack frame is critically important to the programmer who wishes to write re-entrant or recursive functions. The creation of a local stack frame on the MC680x0 family is done through the use of the LINK and UNLK instructions. The LINK instruction saves the frame pointer onto the stack, and places a pointer to the new stack frame in it. The UNLK instruction

restores the old stack frame. For example:

```
link  a5,#-8    ; a5 is chosen to be the frame
        ; pointer. 8 bytes of local stack
        ; frame are allocated.
...
unlk  a5        ; a5, the old frame pointer, and the
        ; old SP are restored.
```

Since the LINK and UNLK instructions maintain both the frame pointer and the stack pointer, the following code segment is perfectly legal:

```
link  a5,#-4

movem.l d0-a4,-(sp)
pea (500).w
move.l d2,-(sp)
bsr.b Routine

unlk  a5
rts
```

RESULT  
None.

SEE ALSO

UNLK

## 1.62 Logical Shift Left and Logical Shift Right

NAME

LSL, LSR -- Logical shift left and logical shift right

SYNOPSIS

```
LSd Dx,Dy
LSd #<data>,Dy
LSd <ea>
where d is direction, L or R
```

Size = (Byte, Word, Long)

FUNCTION

Shift the bits of the operand in the specified direction. The carry bit set set to the last bit shifted out of the operand. The shift count for the shifting of a register may be specified in two different ways:

1. Immediate - the shift count is specified in the instruction (shift range 1-8).
2. Register - the shift count is contained in a data register specified in the instruction (shift count mod 64)

For a register, the size may be byte, word, or long, but for a memory



location, the size must be a word. The shift count is also restricted to one for a memory location.

**RESULT**

X - Set according to the last bit shifted out of the operand.  
N - Set if the result is negative. Cleared otherwise.  
Z - Set if the result is zero. Cleared otherwise.  
V - Always cleared  
C - Set according to the last bit shifted out of the operand.

**SEE ALSO**

ASL, ASR

ROL, ROR

## 1.63 Move Source -> Destination

NAME

MOVE -- Source -> Destination

**SYNOPSIS**

MOVE <ea>, <ea>

Size = (Byte, Word, Long)

**FUNCTION**

Move the content of the source to the destination location. The data is examined as it is moved, and the condition codes set accordingly.

**RESULT**

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

**SEE ALSO**

MOVEA

## 1.64 Move Address Source -> Destination

NAME

MOVEA -- Source -> Destination

**SYNOPSIS**

MOVEA <ea>, An

Size = (Word, Long)

---

## FUNCTION

Move the content of the source to the destination address register. Word sized operands are sign extended to 32 bits before the operation is done.

## RESULT

None.

## SEE ALSO

MOVE

LEA

## 1.65 CCR -> Destination

## NAME

MOVE from CCR -- CCR -> Destination

## SYNOPSIS

MOVE CCR,<ea>

Size = (Word)

## FUNCTION

The content of the status register is moved to the destination location. The source operand is a word, but only the low order byte contains the condition codes. The high order byte is set to all zeros.

## RESULT

None.

## SEE ALSO

Move To CCR

## 1.66 Source -> CCR

## NAME

MOVE to CCR -- Source -> CCR

## SYNOPSIS

MOVE <ea>,CCR

Size = (Word)

## FUNCTION

The content of the source operand is moved to the condition codes. The source operand is a word, but only the low order byte is used to update the condition codes. The high order byte is ignored.

---

## RESULT

- X - Set the same as bit 4 of the source operand.
- N - Set the same as bit 3 of the source operand.
- Z - Set the same as bit 2 of the source operand.
- V - Set the same as bit 1 of the source operand.
- C - Set the same as bit 0 of the source operand.

## SEE ALSO

Move From CCR

## 1.67 Move from SR (privileged)

## NAME

MOVE from SR -- Move from status register (privileged)

## SYNOPSIS

MOVE SR,<ea>

Size = (Word)

## FUNCTION

The content of the status register is moved to the destination location. The operand size is a word.

## RESULT

None.

## SEE ALSO

Move To SR

## 1.68 Move to SR (privileged)

## NAME

MOVE to SR -- Move to status register (privileged)

## SYNOPSIS

MOVE <ea>,SR

Size = (Word)

## FUNCTION

The content of the source operand is moved to the status register. The source operand size is a word and all bits of the status register are affected.

## RESULT

- X - Set the same as bit 4 of the source operand.
  - N - Set the same as bit 3 of the source operand.
-

Z - Set the same as bit 2 of the source operand.  
 V - Set the same as bit 1 of the source operand.  
 C - Set the same as bit 0 of the source operand.

SEE ALSO

MOVE to CCR

MOVE from SR

## 1.69 Move to/from USP (privileged)

NAME

MOVE USP -- Move to/from user stack pointer (privileged)

SYNOPSIS

MOVE USP, An

MOVE An, USP

Size = (Long)

FUNCTION

The contents of the user stack pointer are transferred either to or from the specified address register.

RESULT

None.

SEE ALSO

MOVE from CCR

MOVE to CCR

MOVE from SR

MOVE to SR

## 1.70 Move to/from control register

NAME

MOVEC -- Move to/from control register

SYNOPSIS

MOVEC Rc, Rn

MOVEC Rn, Rc

Size = (Long)

FUNCTION

Copy the contents of the specified control register to the specified general register or copy from the

general register to the control register. This is always a 32-bit transfer even though the the control register may be implemented with fewer bits.

RESULT  
None.

SEE ALSO

MOVE to/from USP

## 1.71 MOVE Multiple registers

NAME

MOVEM -- Move multiple registers

SYNOPSIS

MOVEM <register list>, <ea>

MOVEM <ea>, <register list>

Size = (Word, Long)

FUNCTION

Registers in the register list are either moved to or fetched from consecutive memory locations at the specified address. Data can be either word or long word, but if the register list is destination and the size is word, each register is filled with the source word sign extended to 32-bits.

Also, in the case that the register list is the destination, register indirect with predecrement is not a valid source mode. If the register list is the source, then the destination may not be register indirect with postincrement.

```
MOVEM.L D0/D1/A0, (A2)+    ;invalid
MOVEM.W -(A1), D5/D7/A4    ;invalid
```

The register list is accessed with D0 first through D7, followed by A0 through A7.

RESULT  
None.

SEE ALSO

MOVE

MOVEA

MOVEP

---

## 1.72 MOVE Peripheral data

NAME

MOVEP -- Move peripheral data

SYNOPSIS

MOVEP Dx, (d,Ay)

MOVEP (d,Ay),Dx

Size = (Word, Long)

FUNCTION

Data is transferred between a data register and ever-other byte of memory at the selected address. Example:

```
LEA port0,A0 ; A0 -> $FFFFFFFFFFFFFFFF
MOVEQ #0,D0
MOVEP.L D0,(0,A0) ; A0 -> $FF00FF00FF00FF00
MOVE.L #$55554444,D0
MOVEP.L D0,(1,A0) ; A0 -> $FF55FF55FF44FF44
```

RESULT

None.

SEE ALSO

MOVEM

## 1.73 MOVE signed 8-bit data Quick

NAME

MOVEQ -- Move signed 8-bit data quick

SYNOPSIS

MOVEQ #<data:8>,Dn

Size = (Long)

FUNCTION

Move signed 8-bit data to the specified data register. The specified data is sign extended to 32-bits before it is moved to the register

RESULT

X - Not affected.

N - Set if the result is negative. Cleared otherwise.

Z - Set if the result is zero. Cleared otherwise.

V - Always cleared.

C - Always cleared.

SEE ALSO

MOVE

MOVEA

LEA

## 1.74 MOVE address Space (privileged)

NAME

MOVES -- Move address space (privileged)

SYNOPSIS

MOVES Rn, &lt;ea&gt; (68010+)

MOVES &lt;ea&gt;, Rn (68010+)

Size = (Byte, Word, Long)

FUNCTION

Moves contents (Byte, Word, Long) of register Rn to the addressed space by effective address in the addressable space specified by DFC.

RESULT

None.

SEE ALSO

MOVE from CCR

MOVE to CCR

MOVE from SR

MOVE to SR

MOVE to/from USP

MOVEM

MOVEC

MOVEP

## 1.75 Signed and Unsigned multiply

NAME

MULS -- Signed multiply

MULU -- Unsigned multiply

SYNOPSIS

MULS.W &lt;ea&gt;, Dn 16\*16-&gt;32

MULS.L &lt;ea&gt;, Dn 32\*32-&gt;32 68020+

MULS.L &lt;ea&gt;, Dh:Dl 32\*32-&gt;64 68020+

MULU.W &lt;ea&gt;, Dn 16\*16-&gt;32

```
MULU.L <ea>,Dn 32*32->32 68020+
MULU.L <ea>,Dh:Dl 32*32->64 68020+
```

```
Size = (Word)
Size = (Word, Long) 68020+
```

#### FUNCTION

Multiply two signed (MULS) or unsigned (MULU) integers to produce either a signed or unsigned, respectively, result.

This instruction has three forms. They are basically word, long word, and quad word. The first version is the only one available on a processor lower than a 68020. It will multiply two 16-bit integers and produce a 32-bit result. The second will multiply two 32-bit integers and produce a 32-bit result.

The third form needs some special consideration. It will multiply two 32-bit integers, specified by <ea> and Dl, the result is (sign) extended to 64-bits with the low order 32 being placed in Dl and the high order 32 being placed in Dh.

#### RESULT

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

#### SEE ALSO

DIVS

DIVU

## 1.76 Negate Binary Coded Decimal with extend

#### NAME

NBCD -- Negate binary coded decimal with extend

#### SYNOPSIS

```
NBCD <ea>
```

```
Size = (Byte)
```

#### FUNCTION

The specified BCD number and the extend bit are subtracted from zero. Therefore, if the extend bit is set a nines complement is performed, else a tens complement is performed. The result is placed back in the specified <ea>.

It can be use full to set the zero flag before performing this operation so that multi precision operations can



be correctly tested for zero.

**RESULT**

X - Set the same as the carry bit.  
N - Undefined.  
Z - Cleared if the result is non-zero, unchanged otherwise.  
V - Undefined.  
C - Set if a borrow was generated, cleared otherwise.

**SEE ALSO**

NEG

NEGX

## 1.77 neg

**NAME**

NEG -- Negate

**SYNOPSIS**

NEG <ea>

Size = (Byte, Word, Long)

**FUNCTION**

The operand specified by <ea> is subtracted from zero. The result is stored in <ea>.

**RESULT**

X - Set the same as the carry bit.  
N - Set if the result is negative, otherwise cleared.  
Z - Set if the result is zero, otherwise cleared.  
V - Set if overflow, otherwise cleared.  
C - Cleared if the result is zero, otherwise set.

**SEE ALSO**

NBCD

NEGX

## 1.78 NEGate with eXtend

**NAME**

NEGX -- Negate with extend

**SYNOPSIS**

NEGX <ea>

Size = (Byte, Word, Long)

---

**FUNCTION**

Perform an operation similar to a NEG, with the exception that the operand and the extend bit are both subtracted from zero. The result then being place in the given ea.

As with ADDX, SUBX, ABCD, SBCD, and NBCD, it can be useful to set the zero flag before performing this operation so that multi precision operations can be tested for zero.

**RESULT**

X - Set the same as the carry bit.  
N - Set if the result is negative, otherwise cleared.  
Z - Cleared if the result is non-zero, otherwise unchanged.  
V - Set if an overflow is generated, cleared otherwise.  
C - Set if a borrow is generated, otherwise cleared.

**SEE ALSO**

NEG

NBCD

ADDX

SUBX

## 1.79 No Operation

**NAME**

NOP -- No operation

**SYNOPSIS**

NOP

**FUNCTION**

Nothing happens! This instruction will basically wait until all pending bus activity is completed. This allows synchronization of the pipeline and prevents instruction overlap.

**RESULT**

None.

**SEE ALSO**

## 1.80 Logical complement

**NAME**

NOT -- Logical complement

## SYNOPSIS

NOT <ea>

Size = (Byte, Word, Long)

## FUNCTION

All bits of the specified operand are inverted and placed back in the operand.

## RESULT

X - Not affected.

N - Set if the result is negative, otherwise cleared.

Z - Set if the result is zero, otherwise cleared.

V - Always cleared.

C - Always cleared.

## SEE ALSO

NEG

## 1.81 Logical OR

NAME

OR -- Logical OR

## SYNOPSIS

OR <ea>,Dn

OR Dn,<ea>

Size = (Byte, Word, Long)

## FUNCTION

Performs an OR operation on the destination operand with the source operand.

## RESULT

X - Not Affected

N - Set to the value of the most significant bit.

Z - Set if the result is zero.

V - Always cleared

C - Always cleared

## SEE ALSO

ORI

BSET

## 1.82 Logical OR Immediate

NAME

ORI -- Logical OR immediate

## SYNOPSIS

ORI #<data>, <ea>

Size = (Byte, Word, Long)

## FUNCTION

Performs an OR operation on the destination operand with the source operand.

## RESULT

X - Not Affected  
N - Set to the value of the most significant bit.  
Z - Set if the result is zero.  
V - Always cleared  
C - Always cleared

## SEE ALSO

OR

ORI to CCR

ORI to SR

BSET

## 1.83 Logical OR immediate to CCR

## NAME

ORI to CCR -- Logical OR immediate to the condition code register

## SYNOPSIS

ORI #<data>, CCR

Size = (Byte)

## FUNCTION

Performs an OR operation on the condition codes register with the source operand.

## RESULT

X - Set if bit 4 of the source is set, cleared otherwise.  
N - Set if bit 3 of the source is set, cleared otherwise.  
Z - Set if bit 2 of the source is set, cleared otherwise.  
V - Set if bit 1 of the source is set, cleared otherwise.  
C - Set if bit 0 of the source is set, cleared otherwise.

## SEE ALSO

OR

ORI

---

ORI to SR

## 1.84 Logical OR immediated to SR (privileged)

NAME

ORI to SR -- Logical OR immediated to the status register (privileged)

SYNOPSIS

ORI #<data>,SR

Size = (Word)

FUNCTION

Performs an OR operation on the status register with the source operand, and leaves the result in the status register.

RESULT

X - Set if bit 4 of the source is set, cleared otherwise.  
 N - Set if bit 3 of the source is set, cleared otherwise.  
 Z - Set if bit 2 of the source is set, cleared otherwise.  
 V - Set if bit 1 of the source is set, cleared otherwise.  
 C - Set if bit 0 of the source is set, cleared otherwise.

SEE ALSO

OR

ORI

ORI to CCR

## 1.85 PACK binary coded decimal

NAME

PACK -- Pack binary coded decimal (68020+)

SYNOPSIS

PACK -(Ax),-(Ay),#<adjustment>

PACK Dx,Dy,#<adjustment>

Size = (Byte, Word, Long)

FUNCTION

Convert byte-per-digit unpacked BCD to packed two-digit-per-byte BCD.

RESULT

None.

SEE ALSO

UNPK

## 1.86 Push Effective Address

NAME

PEA -- Push effective address

SYNOPSIS

PEA <ea>

Size = (Long)

FUNCTION

Effective address is stored to stack. Stack is decreased.

RESULT

None.

SEE ALSO

LEA

## 1.87 RESET external devices

NAME

RESET -- Reset external devices

SYNOPSIS

RESET

FUNCTION

RESET line is set, then external circuitry is reset.  
Processor state isn't modified, except PC, which allows restart  
of execution to the next instruction.

RESULT

None.

SEE ALSO

STOP

## 1.88 ROTate Left and ROTate Right

NAME

ROL, ROR -- Rotate left and rotate right

SYNOPSIS

ROd Dx,Dy

ROd #<data>,Dy

ROd <ea>  
 where d is direction, L or R

Size = (Byte, Word, Long)

#### FUNCTION

Rotate the bits of the operand in the specified direction.  
 The rotation count may be specified in two different ways:

1. Immediate - the rotation count is specified in the instruction
2. Register - the rotation count is contained in a data register specified in the instruction

For a register, the size may be byte, word, or long, but for a memory location, the size must be a word. The rotation count is also restricted to one for a memory location.

#### RESULT

X - Not affected  
 N - Set if the result is negative. Cleared otherwise.  
 Z - Set if the result is zero. Cleared otherwise.  
 V - Always cleared  
 C - Set according to the last bit shifted out of the operand.

#### SEE ALSO

ROXL, ROXR

ASL, ASR

LSL, LSR

## 1.89 ROTate Left with eXtend and ROTate Right with eXtend

#### NAME

ROXL, ROXD -- Rotate left with extend and rotate right with extend

#### SYNOPSIS

ROXd Dx,Dy  
 ROXd #<data>,Dy  
 ROXd <ea>  
 where d is direction, L or R

Size = (Byte, Word, Long)

#### FUNCTION

A rotation is made on destination operand bits.  
 Rotation uses bit X.

#### RESULT

X - Set by the last bit out of operand.  
     Not changed if rotation is zero.  
 N - Set if the result is negative. Cleared otherwise.  
 Z - Set if the result is zero. Cleared otherwise.

V - Always cleared  
C - Set according to the last bit shifted out of the operand.

SEE ALSO

ROL, ROR

ASL, ASR

LSL, LSR

## 1.90 ReTurn and Deallocate parameter stack frame

NAME

RTD -- Return and deallocate parameter stack frame (68010+)

SYNOPSIS

RTD #<displacement>

FUNCTION

PC is subtracted from stack and replace old PC address.  
Then displacement is added to SP value.

This instruction is useful to restore reserved space memory of stored arguments at time sub-routine is called.

RESULT

None.

SEE ALSO

RTS

RTE

RTM

## 1.91 ReTurn from Exception (privileged)

NAME

RTE -- Return from exception (privileged)

SYNOPSIS

RTE

FUNCTION

SR and PC are restored by SP. All SR bits are affected.

RESULT

SR is set following to the restored word taken from SP.

SEE ALSO

---



RTS

RTD

RTM

## 1.92 ReTurn from process Module

NAME

RTM -- Return from process module (68020+)

SYNOPSIS

RTM Rn

FUNCTION

Return from a process module called with CALLM.

RESULT

Don't know!

SEE ALSO

RTS

RTE

RTD

## 1.93 ReTurn and Restore CCR

NAME

RTR -- Return and restore condition code register

SYNOPSIS

RTR

FUNCTION

CCR and PC are restored by SP.  
Supervisor byte of SR isn't affected.

RESULT

SR is set following to the restored word taken from SP.

SEE ALSO

RTS

RTE

RTD

---

## 1.94 ReTurn from Subroutine

NAME

RTS -- Return from subroutine

SYNOPSIS

RTS

FUNCTION

PC is restored by SP.

RESULT

None.

SEE ALSO

RTM

RTE

RTD

## 1.95 Subtract Binary Coded Decimal with extend

NAME

SBCD -- Subtract binary coded decimal with extend

SYNOPSIS

SBCD Dy, Dx

SBCD -(Ay), -(Ax)

Size = (Byte)

FUNCTION

Subtracts the source operand to the destination operand along with the extend bit, and stores the result in the destination location.

The subtraction is performed using binary coded decimal arithmetic.

The operands, which are packed BCD numbers, can be addressed in two different ways:

1. Data register to data register: The operands are contained in the data registers specified in the instruction.
2. Memory to memory: The operands are addressed with the predecrement addressing mode using the address registers specified in the instruction.

Normally the Z condition code bit is set via programming before the start of an operation. That allows successful tests for zero results upon completion of multiple-precision operations.

RESULT

X - Set the same as the carry bit.

N - Undefined

---

Z - Cleared if the result is non-zero. Unchanged otherwise.  
 V - Undefined  
 C - Set if a decimal carry was generated. Cleared otherwise.

SEE ALSO

SUB

ADDI

ADDQ

ADDX

ADD

SUBI

SUBQ

ABCD

SUBX

## 1.96 Conditional Set

NAME

Scc -- Conditional set

SYNOPSIS

Scc <ea>

Size = (Byte)

FUNCTION

If condition is true then byte addressed by <ea> is set to \$FF,  
 else byte addressed by <ea> is set to \$00.

Condition code 'cc' specifies one of the following:

CC	carry clear	C'	LS low or same	C+Z
CS	carry set	C	LT less than	N'·V+N'V
EQ	equal	Z	MI minus	N
GE	greater or equal	N·V+N'V'	NE not equal	Z'
GT	greater than	N·V·Z'+N'V'·Z'	PL plus	N'
HI	high	C'·Z'	VC overflow clear	V'
LE	less or equal	Z+N'·V+N'V	VS overflow set	V
T	True	1	F False	0

RESULT

None.

SEE ALSO

Bcc

DBcc

## 1.97 Stop processor execution (privileged)

NAME

STOP -- Stop processor execution (privileged)

SYNOPSIS

STOP #<data:16>

FUNCTION

Immediate data is moved to SR. PC is set to next instruction, and the processor stops fetch and execution of instruction. Execution restarts if if a TRACE exception, an interruption, or a RESET takes place.

When STOP is executed, a TRACE exception is generated (if T = 1). An interruption is allowed if it level is higher than current one. An external RESET always will generate a RESET exception. If bit S is set to zero by the immediate data, execution of this instruction will generate a "privilege violation".

RESULT

SR is set according to immediate data.

SEE ALSO

RESET

## 1.98 SUBtract

NAME

SUB -- Subtract

SYNOPSIS

SUB <ea>,Dn

SUB Dn,<ea>

Size = (Byte, Word, Long)

FUNCTION

Subtracts source operand to destination operand. Result is stored to destination's place.

RESULT

X - Set the same as the carry bit.

N - Set if the result is negative. Cleared otherwise.

Z - Set if the result is zero. Cleared otherwise.

V - Set if an overflow is generated. Cleared otherwise.

C - Set if a carry is generated. Cleared otherwise.

SEE ALSO

---

ADDI

ADDQ

ADDX

ADD

SUBI

SUBQ

SUBX

## 1.99 SUBtract Address

NAME

SUBA -- Subtract address

SYNOPSIS

SUBA <ea>,An

Size = (Word, Long)

FUNCTION

Subtracts source operand to destination operand.  
Source operand with a Word size is extended to 32 bits before operation. Result is stored to destination's place.

RESULT

None.

SEE ALSO

ADDA

SUBI

SUBQ

SUBX

## 1.100 SUBtract Immediate

NAME

SUBI -- Subtract immediate

SYNOPSIS

SUBI #<data>,<ea>

Size = (Byte, Word, Long)

---

## FUNCTION

Subtracts the immediate data to the destination operand, and stores the result in the destination location. The size of the operation may be specified as byte, word, or long.

The size of the immediate data matches the operation size.

## RESULT

X - Set the same as the carry bit.  
N - Set if the result is negative. Cleared otherwise.  
Z - Set if the result is zero. Cleared otherwise.  
V - Set if an overflow is generated. Cleared otherwise.  
C - Set if a carry is generated. Cleared otherwise.

## SEE ALSO

ADD

ADDQ

ADDX

SUB

ADDI

SUBQ

SUBX

## 1.101 SUBtract 3-bit immediate Quick

## NAME

SUBQ -- Subtract 3-bit immediate quick

## SYNOPSIS

SUBQ #<data>, <ea>

Size = (Byte, Word, Long)

## FUNCTION

Subtracts the immediate value of 1 to 8 to the operand at the destination location. The size of the operation may be specified as byte, word, or long. When subtracting to address registers, the condition codes are not altered, and the entire destination address register is used regardless of the operation size.

## RESULT

X - Set the same as the carry bit.  
N - Set if the result is negative. Cleared otherwise.  
Z - Set if the result is zero. Cleared otherwise.  
V - Set if an overflow is generated. Cleared otherwise.  
C - Set if a carry is generated. Cleared otherwise.

## SEE ALSO

ADD  
ADDI  
SUB  
SUBI  
ADDQ

## 1.102 SUBtract with eXtend

NAME

SUBX -- Subtract with extend

SYNOPSIS

SUBX Dy,Dx  
SUBX -(Ay),-(Ax)

Size = (Byte, Word, Long)

FUNCTION

Subtracts the source operand to the destination operand along with the extend bit, and stores the result in the destination location. The subtraction is performed using binary coded decimal arithmetic. The operands, which are packed BCD numbers, can be addressed in two different ways:

1. Data register to data register: The operands are contained in the data registers specified in the instruction.
2. Memory to memory: The operands are addressed with the predecrement addressing mode using the address registers specified in the instruction.

The size of operation can be specified as byte, word, or long.

Normally the Z condition code bit is set via programming before the start of an operation. That allows successful tests for zero results upon completion of multiple-precision operations.

RESULT

X - Set the same as the carry bit.  
N - Set if the result is negative. Cleared otherwise.  
Z - Cleared if the result is non-zero. Unchanged otherwise.  
V - Set if an overflow is generated. Cleared otherwise.  
C - Set if a carry is generated. Cleared otherwise.

SEE ALSO

ADD  
ADDI

---

SUB

SUBI

ADDX

### 1.103 SWAP register upper and lower words

NAME

SWAP -- Swap register upper and lower words

SYNOPSIS

SWAP Dn

Size = (Word)

FUNCTION

Swaps between 16 low bits and 16 high bits of register.

RESULT

X - Not affected

N - Set if the most-significant bit of the result was set. Cleared otherwise.

Z - Set if the 32 bits result was zero. Cleared otherwise.

V - Always cleared.

C - Always cleared.

SEE ALSO

EXG

### 1.104 Test And Set operand

NAME

TAS -- Test and set operand

SYNOPSIS

TAS <ea>

Size = (Byte)

FUNCTION

Test of a byte addressed by <ea>, bits N and Z of SR are set according to result of test.

Bit 7 of byte is set to 1. This instruction uses read-modify-write cycle, which is not dividable and allows synchronisation of several processors. But this instruction is NOT ALLOWED ON AMIGA!  
DON'T USE IT.

RESULT

X - Not affected.

N - Set if MSB of byte is set. Cleared otherwise.

---



Z - Set if byte is zero. Cleared otherwise.  
V - Always cleared.  
C - Always cleared.

SEE ALSO

## 1.105 Initiate processor TRAP

NAME

TRAP -- Initiate processor trap

SYNOPSIS

TRAP #<number>

FUNCTION

Processor starts an exception process. TRAP number is pointed out by 4 bits into the instruction. 16 vectors are free to be used for TRAP.

PC and SR are stored to SSP, and Vector is written to PC.

RESULT

None.

SEE ALSO

TRAPcc

## 1.106 Conditional trap

NAME

TRAPcc -- Conditional trap (68020+)

TRAPv -- Trap on overflow

SYNOPSIS

TRAPcc

TRAPV

FUNCTION

If overflow capacity condition is true (V = 1) then there's generation of a level 7 exception, else execution continue normally.

RESULT

None.

SEE ALSO

TRAP

## 1.107 TeST operand for zero

---

## NAME

TST -- Test operand for zero

## SYNOPSIS

TST <ea>

Size = (Byte, Word, Long)

## FUNCTION

Operand is compared with zero.  
Conditionnal codes are set according to the result.

## RESULT

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

## SEE ALSO

BTST

BFTST

## 1.108 Free stack frame created by LINK

## NAME

UNLK -- Free stack frame created by LINK

## SYNOPSIS

UNLK An

## FUNCTION

Address register specified is moved in SP.  
Contents of SP is moved into address register.

## RESULT

None.

## SEE ALSO

LINK

## 1.109 Unpack binary coded decimal

## NAME

UNPK -- Unpack binary coded decimal (68020+)

## SYNOPSIS

UNPK -(Ax), -(Ay), #<adjustment>

---

UNPK Dx,Dy,#<adjustment>

Size = (Byte, Word, Long)

#### FUNCTION

Convert packed two-digit-per-byte BCD to byte-per-digit unpacked BCD.

#### RESULT

None.

#### SEE ALSO

PACK

## 1.110 Move Instruction Execution Times

These following two tables indicate the number of clock periods for the move instruction. This data includes instruction fetch, operand reads, and operand writes. The number of bus read and write cycles is shown in parenthesis as (r/w).

#### Move Byte and Word Instruction Execution Times

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,ix)	xxx.W	xxx.L
Dn	4(1/0)	4(1/0)	8(1/1)	8(1/1)	8(1/1)	12(2/1)	14(2/1)	12(2/1)	16(3/1)
An	4(1/0)	4(1/0)	8(1/1)	8(1/1)	8(1/1)	12(2/1)	14(2/1)	12(2/1)	16(3/1)
(An)	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)
(An)+	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)
-(An)	10(2/0)	10(2/0)	14(2/1)	14(2/1)	14(2/1)	18(3/1)	20(4/1)	18(3/1)	22(4/1)
d(An)	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
d(An,ix)	14(3/0)	14(3/0)	18(3/1)	18(3/1)	18(3/1)	22(4/1)	24(4/1)	22(4/1)	26(5/1)
xxx.W	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
xxx.L	16(4/0)	16(4/0)	20(4/1)	20(4/1)	20(4/1)	24(5/1)	26(5/1)	24(5/1)	28(6/1)
d(PC)	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
d(PC,ix)	14(3/0)	14(3/0)	18(3/1)	18(3/1)	18(3/1)	22(4/1)	24(4/1)	22(4/1)	26(5/1)
#xxx	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)

The size of the index register (ix) does not affect execution time

#### Move Long Instruction Execute Times

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,ix)	xxx.W	xxx.L
Dn	4(1/0)	4(1/0)	12(1/2)	12(1/2)	12(1/2)	16(2/2)	18(2/2)	16(2/2)	20(3/2)
An	4(1/0)	4(1/0)	12(1/2)	12(1/2)	12(1/2)	16(2/2)	18(2/2)	16(2/2)	20(3/2)
(An)	12(3/0)	12(3/0)	20(3/2)	20(3/2)	20(3/2)	24(4/2)	26(4/2)	24(4/2)	28(5/2)
(An)+	12(3/0)	12(3/0)	20(3/2)	20(3/2)	20(3/2)	24(4/2)	26(4/2)	24(4/2)	28(5/2)
-(An)	14(3/0)	14(3/0)	22(3/2)	22(3/2)	22(3/2)	26(4/2)	28(4/2)	26(4/2)	30(5/2)
d(An)	16(4/0)	16(4/0)	24(4/2)	24(4/2)	24(4/2)	28(5/2)	30(5/2)	28(5/2)	32(6/2)
d(An,ix)	18(4/0)	18(4/0)	26(4/2)	26(4/2)	26(4/2)	30(5/2)	32(5/2)	30(5/2)	34(6/2)
xxx.W	16(4/0)	16(4/0)	24(4/2)	24(4/2)	24(4/2)	28(5/2)	30(5/2)	28(5/2)	32(6/2)

```

xxx.L 20(5/0) 20(5/0) 28(5/2) 28(5/2) 28(5/2) 32(6/2) 34(6/2) 32(6/2) 36(7/2)
d(PC) 16(4/0) 16(4/0) 24(4/2) 24(4/2) 24(4/2) 28(5/2) 30(5/2) 28(5/2) 32(5/2)
d(PC,ix)18(4/0) 18(4/0) 26(4/2) 26(4/2) 26(4/2) 30(5/2) 32(5/2) 30(5/2) 34(6/2)
#xxx 12(3/0) 12(3/0) 20(3/2) 20(3/2) 20(3/2) 24(4/2) 26(4/2) 24(4/2) 28(5/2)

```

The size of the index register (ix) does not affect execution time

## 1.111 Standard Instruction Execution Times

The number of clock periods shown in this table indicates the time required to perform the operations, store the results and read the next instruction. The number of bus read and write cycles is shown in parenthesis as (r/w). The number of clock periods and the number of read and write cycles must be added respectively to those of the effective address calculation where indicated.

In the following table the headings have the following meanings:

An = address register operand, Dn = data register operand, ea = an operand specified by an effective address, and M = memory effective address operand.

### Standard Instruction Execution Times

instruction	Size	op<ea>,An ^ op<ea>,Dn	op Dn, <M>
ADD	byte,word	8(1/0) + 4(1/0) + 8(1/1) +	
	long	6(1/0) +** 6(1/0) +** 12(1/2) +	
AND	byte,word	- 4(1/0) + 8(1/1) +	
	long	- 6(1/0) +** 12(1/2) +	
CMP	byte,word	6(1/0) + 4(1/0) + -	
	long	6(1/0) + 6(1/0) + -	
DIVS	-	-	158(1/0) +*
DIVU	-	-	140(1/0) +*
EOR	byte,word	- 4(1/0) +** 8(1/1) +	
	long	- 8(1/0) +** 12(1/2) +	
MULS	-	-	70(1/0) +*
MULU	-	-	70(1/0) +*
OR	byte,word	- 4(1/0) +** 8(1/1) +	
	long	- 6(1/0) +** 12(1/2) +	
SUB	byte,word	8(1/0) + 4(1/0) + 8(1/1) +	
	long	6(1/0) +** 6(1/0) +** 12(1/2) +	

notes: + Add effective address calculation time

^ Word or long only

\* Indicates maximum value

\*\* The base time of six clock periods is increased to eight if the effective address mode is register direct or immediate (effective address time should also be added)

\*\*\* Only available effective address mode is data register direct

DIVS, DIVU - The divide algorithm used by the MC68000 provides less than 10% difference between the best and the worst case timings.

MULS, MULU - The multiply algorithm requires  $38+2n$  clocks where  $n$  is defined as:

MULU: n = the number of ones in the <ea>  
 MULS: n = concatenate the <ea> with a zero as the LSB;  
 n is the resultant number of 10 or 01 patterns  
 in the 17-bit source; i.e., worst case happens  
 when the source is \$5555

## 1.112 Immediate Instruction Execution Times

The number of clock periods shown in this table includes the time to fetch immediate operands, perform the operations, store the results and read the next operation. The number of bus read and write cycles is shown in parenthesis as (r/w). The number of clock periods and the number of read and write cycles must be added respectively to those of the effective address calculation where indicated.

### Immediate Instruction Execution Times

instruction	size	op #, Dn	op #, An	op #, M
ADDI	byte, word	8 (2/0)	-	12 (2/1) +
	long	16 (3/0)	-	20 (3/2) +
ADDQ	byte, word	4 (1/0)	8 (1/0) *	8 (1/1) +
	long	8 (1/0)	8 (1/0)	12 (1/2) +
ANDI	byte, word	8 (2/0)	-	12 (2/1) +
	long	16 (3/0)	-	20 (3/1) +
CMPI	byte, word	8 (2/0)	-	8 (2/0) +
	long	14 (3/0)	-	12 (3/0) +
EORI	byte, word	8 (2/0)	-	12 (2/1) +
	long	16 (3/0)	-	20 (3/2) +
MOVEQ	long	4 (1/0)	-	-
ORI	byte, word	8 (2/0)	-	12 (2/1) +
	long	16 (3/0)	-	20 (3/2) +
SUBI	byte, word	8 (2/0)	-	12 (2/1) +
	long	16 (3/0)	-	20 (3/2) +
SUBQ	byte, word	4 (1/0)	8 (1/0) *	8 (1/1) +
	long	8 (1/0)	8 (1/0)	12 (1/2) +

+ Add effective address calculation time

\* word only

## 1.113 Single Operand Instruction Execution Times

This table indicates the number of clock periods for the single operand instructions. The number of bus read and write cycles is shown in parenthesis as (r/w). The number of clock periods and the number of read and write cycles must be added respectively to those of the effective address calculation where indicated.

### Single Operand Instruction Execution Times

	instruction size	register	memory
CLR	byte, word 4 (1/0)	8 (1/1) +	
	long 6 (1/0)	12 (1/2) +	
NBCD	byte 6 (1/0)	8 (1/1) +	
NEG	byte, word 4 (1/0)	8 (1/1) +	
	long 6 (1/0)	12 (1/2) +	
NEGX	byte, word 4 (1/0)	8 (1/1) +	
	long 6 (1/0)	12 (1/2) +	
NOT	byte, word 4 (1/0)	8 (1/1) +	
	long 6 (1/0)	12 (1/2) +	
Scc	byte, false 4 (1/0)	8 (1/1) +	
	byte, true 6 (1/0)	8 (1/1) +	
TAS #	byte 4 (1/0)	10 (1/1) +	
TST	byte, word 4 (1/0)	4 (1/0) +	
	long 4 (1/0)	4 (1/0) +	

+ add effective address calculation time

# This instruction should never be used on the Amiga as its invisible read/write cycle can disrupt system DMA.

## 1.114 Rotate Instruction Execution Times

This table indicates the number of clock periods for the shift and rotate instructions. The number of read and write cycles is shown in parenthesis as (r/w). The number of clock periods and the number of read and write cycles must be added respectively to those of the effective address calculation where indicated.

### Shift/Rotate Instruction Execution Times

	instruction size	register	memory
ASR, ASL	byte, word 6+2n (1/0)	8 (1/1) +	
	long 8+2n (1/0)	-	
LSR, LSL	byte, word 6+2n (1/0)	8 (1/1) +	
	long 8+2n (1/0)	-	
ROR, ROL	byte, word 6+2n (1/0)	8 (1/1) +	
	long 8+2n (1/0)	-	
ROXR, ROXL	byte, word 6+2n (1/0)	8 (1/1) +	
	long 8+2n (1/0)	-	

+ add effective address calculation time

n is the shift or rotate count

## 1.115 Bit Manipulation Instruction Execution Times

This table indicates the number of clock periods required for the bit manipulation instructions. The number of read and write cycles is shown in parenthesis as (r/w). The number of clock periods and the number of read and write cycles must be added respectively to those of the effective address

calculation where indicated.

#### Bit Manipulation Instruction Execution Times

instruction	size	dynamic		static	
		register	memory	register	memory
BCHG	byte	-	8 (1/1) +	-	12 (2/1) +
	long	8 (1/0) *	-	12 (2/0) *	-
BCLR	byte	-	8 (1/1) +	-	12 (2/1) +
	long	10 (1/0) *	-	14 (2/0) *	-
BSET	byte	-	8 (1/1) +	-	12 (2/1) +
	long	8 (1/0) *	-	12 (2/0) *	-
BTST	byte	-	4 (1/0) +	-	8 (2/0) +
	long	6 (1/0)	-	10 (2/0)	-

+ add effective address calculation time

\* indicates maximum value

## 1.116 Specificational Instruction Execution Times

This table indicates the number of clock periods for the conditional instructions. The number of read and write cycles is shown in parenthesis as (r/w). The number of clock periods and the number of read and write cycles must be added respectively to those of the effective address calculation where indicated.

#### Conditional Instruction Execution Times

instruction	displacement	branch	
		taken	not taken
Bcc	byte	10 (2/0)	8 (1/0)
	word	10 (2/0)	12 (1/0)
BRA	byte	10 (2/0)	-
	word	10 (2/0)	-
BSR	byte	18 (2/2)	-
	word	18 (2/2)	-
DBcc	CC true	-	12 (2/0)
	CC false	10 (2/0)	14 (3/0)

## 1.117 JMP, JSR, LEA, PEA and MOVEM Instruction Execution Times

This Table indicates the number of clock periods required for the jump, jump-to-subroutine, load effective address, push effective address and move multiple registers instructions. The number of bus read and write cycles is shown in parenthesis as (r/w).

#### JMP, JSR, LEA, PEA and MOVEM Instruction Execution Times

instr	size	(An)	(An)+	-(An)	d(An)
JMP	-	8(2/0)	-	-	10(2/0)
JSR	-	16(2/2)	-	-	18(2/2)
LEA	-	4(1/0)	-	-	8(2/0)
PEA	-	12(1/2)	-	-	16(2/2)
MOVEM	word	12+4n	12+4n	-	16+4n
M->R		(3+n/0)	(3+n/0)	-	(4+n/0)
	long	12+8n	12+8n	-	16+8n
		(3+2n/0)	(3+2n/0)	-	(4+2n/0)
MOVEM	word	8+4n	-	8+4n	12+4n
R->M		(2/n)	-	(2/n)	(3/n)
	long	8+8n	-	8+8n	12+8n
		(2/2n)	-	(2/2n)	(3/2n)

instr	size	d(An,ix)+	xxx.W	xxx.L	d(PC)	d(PC,ix)*
JMP	-	14(3/0)	10(2/0)	12(3/0)	10(2/0)	14(3/0)
JSR	-	22(2/2)	18(2/2)	20(3/2)	18(2/2)	22(2/2)
LEA	-	12(2/0)	8(2/0)	12(3/0)	8(2/0)	12(2/0)
PEA	-	20(2/2)	16(2/2)	20(3/2)	16(2/2)	20(2/2)
MOVEM	word	18+4n	16+4n	20+4n	16+4n	18+4n
M->R		(4+n/0)	(4+n/0)	(5+n/0)	(4+n/0)	(4+n/0)
	long	18+8n	16+8n	20+8n	16+8n	18+8n
		(4+2n/0)	(4+2n/0)	(5+2n/0)	(4+2n/0)	(4+2n/0)
MOVEM	word	14+4n	12+4n	16+4n	-	-
R->M		(3/n)	(3/n)	(4/n)	-	-
	long	14+8n	12+8n	16+8n	-	-
		(3/2n)	(3/2n)	(4/2n)	-	-

n is the number of registers to move

\* is the size of the index register (ix) does not affect the instruction's execution time

## 1.118 Multi-Precision Instruction Execution Times

This table indicates the number of clock periods for the multi-precision instructions. The number of clock periods includes the time to fetch both operands, perform the operations, store the results and read the next instructions. The number of read and write cycles is shown in parenthesis as (r/w).

The headings have the following meanings: Dn = data register operand and M = memory operand.

### Multi-Precision Instruction Execution Times

instruction	size	op Dn,Dn	op M,M
ADDX	byte, word	4(1/0)	18(3/1)
	long	8(1/0)	30(5/2)
CMPM	byte, word	-	12(3/0)
	long	-	20(5/0)
SUBX	byte, word	4(1/0)	18(3/1)
	long	8(1/0)	30(5/2)
ABCD	byte	6(1/0)	18(3/1)



SBCD            byte        6(1/0)      18(3/1)

## 1.119 Miscellaneous Instruction Execution Times

This table indicates the number of clock periods for the following miscellaneous instructions. The number of bus read and write cycles is shown in parenthesis as (r/w). The number of clock periods and plus the number of read and write cycles must be added to those of the effective address calculation where indicated.

### Miscellaneous Instruction Execution Times

instruction	size	register	memory
ANDI to CCR	byte	20(3/0)	-
ANDI to SR	word	20(3/0)	-
CHK	-	10(1/0) +	-
EORI to CCR	byte	20(3/0)	-
EORI to SR	word	20(3/0)	-
ORI to CCR	byte	20(3/0)	-
ORI to SR	word	20(3/0)	-
MOVE from SR	-	6(1/0)	8(1/1)+
MOVE to CCR	-	12(1/0)	12(1/0)+
MOVE to SR	-	12(1/0)	12(1/0)+
EXG	-	6(1/0)	-
EXT	word	4(1/0)	-
	long	4(1/0)	-
LINK	-	16(2/2)	-
MOVE from USP	-	4(1/0)	-
MOVE to USP	-	4(1/0)	-
NOP	-	4(1/0)	-
RESET	-	132(1/0)	-
RTE	-	20(5/0)	-
RTR	-	20(5/0)	-
RTS	-	16(4/0)	-
STOP	-	4(0/0)	-
SWAP	-	4(1/0)	-
TRAPV (No Trap)	-	4(1/0)	-
UNLK	-	12(3/0)	-

+ add effective address calculation time

## 1.120 Move Peripheral Instruction Execution Times

instruction	size	register->memory	memory->register
MOVEP	word	16(2/2)	16(4/0)
	long	24(2/4)	24(6/0)

## 1.121 Exception Processing Execution Times

This table indicates the number of clock periods for exception processing. The number of clock periods includes the time for all stacking, the vector fetch and the fetch of the first two instruction words of the handler routine. The number of bus read and write cycles is shown in parenthesis as (r/w).

### Exception Processing Execution Times

exception	periods
address error	50(4/7)
bus error	50(4/7)
CHK instruction (trap taken)	44(5/3)+
Divide by Zero	42(5/3)
illegal instruction	34(4/3)
interrupt	44(5/3)*
privilege violation	34(4/3)
RESET **	40(6/0)
trace	34(4/3)
TRAP instruction	38(4/3)
TRAPV instruction (trap taken)	34(4/3)

+ add effective address calculation time

\* the interrupt acknowledge cycle is assumed to take four clock periods

\*\* indicates the time from when RESET and HALT are first sampled as negated to when instruction execution starts

## 1.122 ASP68K PROJECT, Sixth Edition

ASP68K PROJECT

Sixth Edition

by Michael Glew  
 mglew@laurel.ocs.mq.edu.au  
 Technophilia BBS +61-2-8073563

January 1994

---

### C O N T R I B U T O R S

---

Erik Bakke, Robert Barton, Bernd Blank, Kasimir Blomstedt, Frans Bouma, David Carson, Nicolas Dade, Aaron Digulla, Irmen de Jong, Andy Duplain, Denis Duplan, Steven Eker, Calle Englund, Alexander Fritsch, Charlie Gibbs, Kurt Haenen, Jon Hudson, Kjetil Jacobsen, Olav Kalgraf, Makoto Kamada,

---

Markku Kolkka, John Lane, Jonathan Mahaffy, Dave Mc Mahan, Lindsay Meek, Walter Misar, Boerge Noest, Gunnar Rxnning, Jay Scott, Olaf Seibert, Peter Simons.

---

## I N T R O D U C T I O N

---

A while back, I was quite interested to find that there was an electronic magazine called "howtocode" that included lots of interesting hints and tips of coding. In the fifth edition, there was a list of optimizations that really got me thinking. "What if there was a proggy that you could put an assembler program through, that would speed it up, taking out all the stupid things output by compilers, and over-tired coders?" 8). I started combing the networks, and came across one such program, called the "SELCO Source Optimizer". It only had four optimizations, so I set to writing my own.

Step one was to collect as many optimization ideas as I could. I posted to Usenet and got an impressive response, and the contributors are listed above. I promised a report on the optimizations received, and here it is. My aim now is to write a program to make these optimizations, and to distribute it. Contributors will receive a copy of the final archive, to thank them for their time and energy. Further contributions will be welcomed, so rather than making changes yourself tell me what you want changed, and I'll distribute it with the next update.

---

## C H A N G E S

---

### 2nd Edition

The second edition incorporated a hell of a lot of corrections. Double copies of some optimizations were incorporated in to just one copy, and a few additions were made. Sorry that the first edition was not sent out to all contributors, but I was a tad busy. 8)

### 3rd Edition

Due to the distribution of the second edition document, many comments were received and a couple of the "optimizations" were found to be incorrect. Analysis of the mul/div optimizations ended in a few modifications for safety. They still save a huge number of clock cycles, so it is better to be safe than sorry.

Also, I have made it so that the number of words of space saved or increased is shown. Space savings are positive, increases are negative. Zero means no change.

### 4th Edition

---

Some minor changes and additions as well as the addition of columns for '030 and '040 CPUs - whole new format was required...

#### 5th Edition

Eric Bakke released his docs on 020+ CPUs and 881/882 FPU's. I have been given permission to use these docs to further the capabilities of asp68k. Thanks Eric... I really would like to get a hold of the 020,030,040 Programmer Reference Cards or manuals, so if anyone has any copies they wanna send me, let me know... Local Motorola Distributers are not too helpful.

#### 6th Edition

Aaron Digulla advised that it would be helpful if the optimizations were sorted somehow. I will sort by the the first letters of the first line of the optimizations. Also a special thanks to Makoto Kamada for his detailed contributions, without such this text would have died long ago..

---

### O P T I M I Z A T I O N S

---

Note:-

m? = memory operand  
 dx = data register  
 ds = data register (scratch)  
 ax = address register  
 rx = either a data or address register  
 #n = immediate operand  
 ??,?1,?2= address label  
 \* = anything  
 .x = any size  
 b<cc> = branch commands

Opt = optimization  
 Notes = notes about where optimization is valid, and misc notes  
 Speed = are clock periods saved? ("Y" = yes  
       "y" = in some cases  
       "N" = no  
       "\*" = increase  
        "-" = cannot be used on this cpu  
        "!" = must be used on this cpu)  
 Size = how many bytes are saved?

---

Opt	Speed	Size
	000 010 020 030 040	

---

* ??* -> * n(pc)*	Y   Y   ?   ?   ?   2
-------------------	-----------------------

---

```

n = ??-pc, n < 32768
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
*0(ax)* -> *(ax)*          | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
add*.x #0,dx -> tst.x dx    | Y | Y | ? | ? | ? | 2/4
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
add.x #n,* -> addq.x #n,*   | Y | Y | ? | ? | ? | 2/4
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
if 1 <= n <= 8
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
add.x #n,* -> subq.x #-n,*  | Y | Y | ? | ? | ? | 2/4
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-8 <= n <= -1
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
add.x #n,ax -> lea n(ax),ax | Y | Y | ? | ? | ? | 0/2
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-32767 <= n <= -9, 9 <= n <= 32767
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
addq.l #n,ax -> addq.w #n,ax | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
addq.l #n,ry -> add.l #(n+m),ry | Y | Y | ? | ? | ? | -2
addq.l #m,ry          | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
addq.x #2,ax -> move.w *,(ax) | Y | Y | ? | Y | ? | 2
move.w *,-(ax)          | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
.x is .w or .l
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
addq.x #4,ax -> move.l *,(ax) | Y | Y | ? | Y | ? | 2
move.l *,-(ax)          | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
.x is .w or .l
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
addq.x #6,ax -> move.w *1,4(ax) | Y | Y | ? | ? | ? | 0
move.w *1,-(ax)   move.l *2,(ax) | | | | |
move.l *2,-(ax)   | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
.x is .w or .l
*1 and *2 do not contain ax
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
addq.x #6,ax -> move.l *1,2(ax) | Y | Y | ? | ? | ? | 0
move.l *1,-(ax)   move.w *2,(ax) | | | | |
move.w *2,-(ax)   | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
.x is .w or .l
*1 and *2 do not contain ax
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
addq.x #8,ax -> move.l *1,4(ax) | Y | Y | ? | ? | ? | 0
move.l *1,-(ax)   move.l *2,(ax) | | | | |
move.l *2,-(ax)   | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
.x is .w or .l
*1 and *2 do not contain ax
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
addq.x #4,sp -> move.l ax,(sp) | Y | Y | ? | Y | ? | 2
pea (ax)          | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

```

.x is .w or .l
ax,ay are not a7(=sp)
-----+-----+-----+-----+-----+-----+-----+-----+
addq.x #6,sp  -> move.w *,4(sp)   | Y | Y | ? | ? | ? | 0
move.w *,-(sp)  move.l ax,(sp)  |   |   |   |   |   |   |
pea (ax)        |   |   |   |   |   |   |
-----+-----+-----+-----+-----+-----+-----+
.x is .w or .l
ax,ay are not a7(=sp)
-----+-----+-----+-----+-----+-----+-----+
addq.x #6,sp  -> move.l ax,2(sp) | Y | Y | ? | ? | ? | 0
pea (ax)      move.w *,(sp)     |   |   |   |   |   |   |
move.w *,-(sp) |   |   |   |   |   |   |
-----+-----+-----+-----+-----+-----+-----+
.x is .w or .l
ax,ay are not a7(=sp)
-----+-----+-----+-----+-----+-----+-----+
addq.x #8,sp  -> move.l *,4(sp)   | Y | Y | ? | ? | ? | 0
move.l *,-(sp)  move.l ax,(sp)  |   |   |   |   |   |   |
pea (ax)        |   |   |   |   |   |   |
-----+-----+-----+-----+-----+-----+-----+
.x is .w or .l
ax,ay are not a7(=sp)
-----+-----+-----+-----+-----+-----+-----+
addq.x #8,sp  -> move.l ax,4(sp)  | Y | Y | ? | ? | ? | 0
pea (ax)      move.l *,(sp)     |   |   |   |   |   |   |
move.l *,-(sp) |   |   |   |   |   |   |
-----+-----+-----+-----+-----+-----+-----+
.x is .w or .l
ax,ay are not a7(=sp)
-----+-----+-----+-----+-----+-----+-----+
addq.x #8,sp -> move.l ax,4(sp)  | Y | Y | ? | ? | ? | 0
pea (ax)      move.l ay,(sp)    |   |   |   |   |   |   |
pea (ay)      |   |   |   |   |   |   |
-----+-----+-----+-----+-----+-----+-----+
.x is .w or .l
ax,ay are not a7(=sp)
-----+-----+-----+-----+-----+-----+-----+
and.l #n,dx -> bclr.l #b,dx     | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+-----+
not(n) = 2^b (only 1 bit off)
-----+-----+-----+-----+-----+-----+-----+
asl.b #2,dy -> add.b dy,dy      | Y | Y | ? | ? | ? | -2
               add.b dy,dy      |   |   |   |   |   |
-----+-----+-----+-----+-----+-----+-----+
asl.b #n,dx -> clr.b dx         | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+-----+-----+
status flags are wrong, n>=8
-----+-----+-----+-----+-----+-----+-----+
asl.l #16,dx -> swap dx         | Y | Y | ? | ? | ? | -2
               clr.w dx         |   |   |   |   |   |
-----+-----+-----+-----+-----+-----+-----+
status flags are wrong
-----+-----+-----+-----+-----+-----+-----+
asl.l #n,dx -> asl.w #(n-16),dx | Y | Y | ? | ? | ? | -4
               swap dx         |   |   |   |   |   |
               clr.w dx         |   |   |   |   |   |

```

```

-----+-----+-----+-----+-----+-----+
status flags are wrong, 16<n<32
-----+-----+-----+-----+-----+-----+
asl.l #n,dx -> moveq #0,dx      | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
status flags are wrong, n>=32
-----+-----+-----+-----+-----+-----+
asl.w #2,dy -> add.w dy,dy      | Y | Y | ? | ? | ? | -2
          add.w dy,dy          | |  | |  | |
-----+-----+-----+-----+-----+
asl.w #n,dx -> clr.w dx        | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
status flags are wrong, n>=16
-----+-----+-----+-----+-----+-----+
asl.x #1,dy -> add.x dy,dy      | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
asr.b #n,dx -> clr.b dx        | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
status flags are wrong, n>=8
-----+-----+-----+-----+-----+-----+
asr.l #16,dx -> swap dx        | Y | Y | ? | ? | ? | -2
          ext.l dx            | |  | |  | |
-----+-----+-----+-----+-----+
status flags are wrong
-----+-----+-----+-----+-----+-----+
asr.l #n,dx -> moveq #0,dx      | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
status flags are wrong, n>=32
-----+-----+-----+-----+-----+-----+
asr.l #n,dx -> swap dx        | Y | Y | ? | ? | ? | -4
          asr.w #(n-16),dx    | |  | |  | |
          ext.l dx            | |  | |  | |
-----+-----+-----+-----+-----+
status flags are wrong, 16<n<32
-----+-----+-----+-----+-----+-----+
asr.w #n,dx -> clr.w dx        | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
status flags are wrong, n>=16
-----+-----+-----+-----+-----+-----+
b<cc>.w ?? -> b<cc>.s ??      | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+
abs(??-pc)<128
-----+-----+-----+-----+-----+-----+
bclr.l #n,dx -> and.w #m,dx    | Y | Y | ? | Y | ? | 0
-----+-----+-----+-----+-----+
0 <= n <= 15, m = 65535-(2^n)
status flags are wrong
-----+-----+-----+-----+-----+-----+
bra ?? -> (nothing)          | Y | Y | Y | ? | ? | 2/4
??  ??                      | |  | |  | |
-----+-----+-----+-----+-----+
remove null branches, but keep the label
-----+-----+-----+-----+-----+-----+
bset.b #7,m? -> tas m?        | y | y | ? | ? | ? | 2
beq ??                      bpl ?? | |  | |  | |
-----+-----+-----+-----+-----+
m? must be address allowing read-modify-write transfer.

```

Status flags are wrong

bset.b #7,m? -> tas m?		y		y		?		?		?		2
bne ??												

m? must be address allowing read-modify-write transfer.

Status flags are wrong

bset.b #7,m? -> tas m?		y		y		?		?		?		2
------------------------	--	---	--	---	--	---	--	---	--	---	--	---

m? must be address allowing read-modify-write transfer.

Status flags are wrong

bset.l #7,dx -> tas dx		Y		Y		?		Y		?		2
beq ??												

status flags are wrong

bset.l #7,dx -> tas dx		Y		Y		?		Y		?		2
bne ??												

status flags are wrong

bset.l #7,dx -> tas dx		Y		Y		?		Y		?		2
------------------------	--	---	--	---	--	---	--	---	--	---	--	---

status flags are wrong

bset.l #n,dx -> or.w #m,dx		Y		Y		?		Y		?		0
----------------------------	--	---	--	---	--	---	--	---	--	---	--	---

0 <= n <= 15, m = 2^n

status flags are wrong

bsr ?? -> bra ??		Y		Y		?		?		?		2
rts												

different stack depth

btst.b #7,m? -> tst.b m?		Y		Y		?		?		?		2
beq ??												

Status flags are wrong. Not valid for Dn, d16(PC), d8(PC,Xn) dest address modes.

btst.b #7,m? -> tst.b m?		Y		Y		?		?		?		2
bne ??												

Status flags are wrong. Not valid for Dn, d16(PC), d8(PC,Xn) dest address modes.

btst.l #7,dx -> tst.b dx		Y		Y		?		Y		?		2
beq ??												

Status flags are wrong.

btst.l #7,dx -> tst.b dx		Y		Y		?		Y		?		2
bne ??												



Status flags are wrong.

btst.l #15,dx -> tst.w dx		Y		Y		?		Y		?		2
beq ??												

Status flags are wrong.

btst.l #15,dx -> tst.w dx		Y		Y		?		Y		?		2
bne ??												

Status flags are wrong.

btst.l #31,dx -> tst.l dx		Y		Y		?		Y		?		2
beq ??												

Status flags are wrong.

btst.l #31,dx -> tst.l dx		Y		Y		?		Y		?		2
bne ??												

status flags are wrong

clr.b mn -> clr.w mn		Y		Y		?		?		?		2/4/6
clr.b mn+1												

best if mn is longword aligned

clr.l dx -> moveq #0,dx		Y		Y		?		?		?		0
-------------------------	--	---	--	---	--	---	--	---	--	---	--	---

clr.w mn -> clr.l mn		Y		Y		?		?		?		2/4/6
clr.w mn+2												

best if mn is longword aligned

clr.x -(ax) -> move.x ds,-(ax)		Y		Y		?		?		?		0
--------------------------------	--	---	--	---	--	---	--	---	--	---	--	---

ds must equal zero

clr.x n(ax,rx) -> move.x ds,n(ax,rx)		Y		Y		?		?		?		0
--------------------------------------	--	---	--	---	--	---	--	---	--	---	--	---

ds must equal zero

cmp.x #0,ax -> move.x ax,ds		Y		Y		?		?		?		2/4
-----------------------------	--	---	--	---	--	---	--	---	--	---	--	-----

move ax to scratch register

cmp.x #0,ax -> tst.x ax		-		-		?		?		?		?
-------------------------	--	---	--	---	--	---	--	---	--	---	--	---

for .w and .l

cmp.x #0,dx -> tst.x dx		Y		Y		?		?		?		2/4
cmp.x #0,m? -> tst.x m?		Y		Y		?		?		?		2/4

may not be legal on some early '000 CPUs

divu.l #n,dx -> lsr.l #m,dx		!		!		?		?		?		4
-----------------------------	--	---	--	---	--	---	--	---	--	---	--	---



```
-----+-----+-----+-----+-----+
lea n(ax),ax -> subq.w #-n,ax      | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+
```

```
if -8 <= n <= -1
```

```
-----+-----+-----+-----+-----+
lsl.b #2,dx -> add.b dy,dx        | Y | Y | ? | ? | ? | -2
      add.b dy,dx      | | | | |
-----+-----+-----+-----+-----+
```

```
-----+-----+-----+-----+-----+
lsl.b #n,dx -> clr.b dx          | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
```

```
status flags are wrong, n>=8
```

```
-----+-----+-----+-----+-----+
lsl.l #16,dx -> swap dx          | Y | Y | ? | ? | ? | -2
      clr.w dx      | | | | |
-----+-----+-----+-----+-----+
```

```
status flags are wrong
```

```
-----+-----+-----+-----+-----+
lsl.l #n,dx -> lsl.w #(n-16),dx  | Y | Y | ? | ? | ? | -4
      swap dx      | | | | |
      clr.w dx     | | | | |
-----+-----+-----+-----+-----+
```

```
status flags are wrong, 16<n<32
```

```
-----+-----+-----+-----+-----+
lsl.l #n,dx -> moveq #0,dx       | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
```

```
status flags are wrong, n>=32
```

```
-----+-----+-----+-----+-----+
lsl.w #2,dx -> add.w dy,dx        | Y | Y | ? | ? | ? | -2
      add.w dy,dx      | | | | |
-----+-----+-----+-----+-----+
```

```
-----+-----+-----+-----+-----+
lsl.w #n,dx -> clr.w dx          | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
```

```
status flags are wrong, n>=16
```

```
-----+-----+-----+-----+-----+
lsl.x #1,dx -> add.x dy,dx        | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
```

```
-----+-----+-----+-----+-----+
lsr.b #n,dx -> clr.b dx          | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
```

```
status flags are wrong, n>=8
```

```
-----+-----+-----+-----+-----+
lsr.l #16,dx -> clr.w dx          | Y | Y | ? | ? | ? | -2
      swap dx      | | | | |
-----+-----+-----+-----+-----+
```

```
status flags are wrong
```

```
-----+-----+-----+-----+-----+
lsr.l #n,dx -> clr.w dx          | Y | Y | ? | ? | ? | -4
      swap dx      | | | | |
      lsr.w #(n-16),dx | | | | |
-----+-----+-----+-----+-----+
```

```
status flags are wrong, 16<n<32
```

```
-----+-----+-----+-----+-----+
lsr.l #n,dx -> moveq #0,dx       | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
```

```
status flags are wrong, n>=32
```

```
-----+-----+-----+-----+-----+
lsr.w #n,dx -> clr.w dx          | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+
```

```

status flags are wrong, n>=16
-----+-----+-----+-----+-----+-----+-----+-----+
move.b #-1,(ax) -> st (ax)      | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+
status flags are wrong
-----+-----+-----+-----+-----+-----+
move.b #-1,(ax)+ -> st (ax)+    | N | N | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+
status flags are wrong
-----+-----+-----+-----+-----+-----+
move.b #-1,-(ax) -> st -(ax)    | N | N | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+
status flags are wrong
-----+-----+-----+-----+-----+-----+
move.b #-1,?? -> st ??         | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+
status flags are wrong
-----+-----+-----+-----+-----+-----+
move.b #-1,dx -> st dx         | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+
status flags are wrong
-----+-----+-----+-----+-----+-----+
move.b #-1,n(ax) -> st n(ax)    | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+
status flags are wrong
-----+-----+-----+-----+-----+-----+
move.b #-1,n(ax,rx) -> st n(ax,rx) | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+
status flags are wrong
-----+-----+-----+-----+-----+-----+
move.b #x,mn -> move.w #xy,mn   | Y | Y | ? | ? | ? | 4/6/8
move.b #y,mn+1                | | | | |
-----+-----+-----+-----+-----+-----+
best if mn is longword aligned
-----+-----+-----+-----+-----+-----+
move.l #n,-(sp) -> pea n.w      | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+
-32767 <= n <= 32767
-----+-----+-----+-----+-----+-----+
move.l #n,ax -> move.w #n,ax    | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+
-32767 <= n <= 32767
-----+-----+-----+-----+-----+-----+
move.l #n,dx -> moveq #-128,dx  | Y | Y | ? | N | * | 2
subq.l #n+128,dx              | | | | |
-----+-----+-----+-----+-----+-----+
-136 <= n <= -129
-----+-----+-----+-----+-----+-----+
move.l #n,dx -> moveq #m,dx     | Y | Y | ? | ? | ? | 2
not.b dx                       | | | | |
-----+-----+-----+-----+-----+-----+
128 <= n <= 255, m = 255-n
-----+-----+-----+-----+-----+-----+
move.l #n,dx -> moveq #m,dx     | Y | Y | ? | ? | ? | 2
not.w dx                       | | | | |
| | | | |
-----+-----+-----+-----+-----+-----+

```

```

65534 <= n <= 65408 or -65409 <= n <= -65536, m = 65535-abs(n)
-----+-----+-----+-----+-----+-----+-----+-----+-----+
move.l #n,dx -> moveq #m,dx      | Y | Y | ? | ? | ? | 2
      swap dx      | | | | | |
                    | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
-8323073 <= n <= -65537 or 4096 <= n <= 8323072, n = m*65536
-----+-----+-----+-----+-----+-----+-----+-----+-----+
move.l #n,dx -> moveq #n,dx      | Y | Y | ? | ? | ? | 4
-----+-----+-----+-----+-----+-----+-----+-----+-----+
if -128 <= n <= 127
-----+-----+-----+-----+-----+-----+-----+-----+-----+
move.l #n,dx -> moveq #y,dx      | * | * | ? | ? | ? | 2
      lsl.l #z,dx  | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
n = y * 2^z
-----+-----+-----+-----+-----+-----+-----+-----+-----+
move.l #n,dx -> moveq #m,dx      | Y | Y | ? | N | ? | 2
      add.b dx,dx  | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
(128 <= n <= 254 or -256 <= n <= -130) and n is even, m = n/2
-----+-----+-----+-----+-----+-----+-----+-----+-----+
move.l #n,dx -> moveq #m,dx      | Y | Y | ? | * | ? | 2
      bchg.l dx,dx  | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
n = -32881 -> m = -113
n = -32849 -> m = -81
n = -32817 -> m = -49
n = -32785 -> m = -17
n = -16498 -> m = -114
n = -16466 -> m = -82
n = -16434 -> m = -50
n = -16402 -> m = -18
n = -8307 -> m = -115
n = -8275 -> m = -83
n = -8243 -> m = -51
n = -8211 -> m = -19
n = -4212 -> m = -116
n = -4180 -> m = -84
n = -4148 -> m = -52
n = -4116 -> m = -20
n = -2165 -> m = -117
n = -2133 -> m = -85
n = -2101 -> m = -53
n = -2069 -> m = -21
n = -1142 -> m = -118
n = -1110 -> m = -86
n = -1078 -> m = -54
n = -1046 -> m = -22
n = -631 -> m = -119
n = -599 -> m = -87
n = -567 -> m = -55
n = -535 -> m = -23
n = -376 -> m = -120
n = -344 -> m = -88
n = -312 -> m = -56
n = -280 -> m = -24

```

n = 264 -> m = 8  
 n = 296 -> m = 40  
 n = 328 -> m = 72  
 n = 360 -> m = 104  
 n = 521 -> m = 9  
 n = 553 -> m = 41  
 n = 585 -> m = 73  
 n = 617 -> m = 105  
 n = 1034 -> m = 10  
 n = 1066 -> m = 42  
 n = 1098 -> m = 74  
 n = 1130 -> m = 106  
 n = 2059 -> m = 11  
 n = 2091 -> m = 43  
 n = 2123 -> m = 75  
 n = 2155 -> m = 107  
 n = 4108 -> m = 12  
 n = 4140 -> m = 44  
 n = 4172 -> m = 76  
 n = 4204 -> m = 108  
 n = 8205 -> m = 13  
 n = 8237 -> m = 45  
 n = 8269 -> m = 77  
 n = 8301 -> m = 109  
 n = 16398 -> m = 14  
 n = 16430 -> m = 46  
 n = 16462 -> m = 78  
 n = 16494 -> m = 110  
 n = 32783 -> m = 15  
 n = 32815 -> m = 47  
 n = 32847 -> m = 79  
 n = 32879 -> m = 111

move.l #n, dx -> moveq #m, dx		N		N		?		*		?		2
bchg.l dx, dx												

n = -2147483617 -> m = 31  
 n = -2147483585 -> m = 63  
 n = -2147483553 -> m = 95  
 n = -2147483521 -> m = 127  
 n = -1073741922 -> m = -98  
 n = -1073741890 -> m = -66  
 n = -1073741858 -> m = -34  
 n = -1073741826 -> m = -2  
 n = -536871011 -> m = -99  
 n = -536870979 -> m = -67  
 n = -536870947 -> m = -35  
 n = -536870915 -> m = -3  
 n = -268435556 -> m = -100  
 n = -268435524 -> m = -68  
 n = -268435492 -> m = -36  
 n = -268435460 -> m = -4  
 n = -134217829 -> m = -101  
 n = -134217797 -> m = -69  
 n = -134217765 -> m = -37  
 n = -134217733 -> m = -5  
 n = -67108966 -> m = -102

n = -67108934 -> m = -70  
n = -67108902 -> m = -38  
n = -67108870 -> m = -6  
n = -33554535 -> m = -103  
n = -33554503 -> m = -71  
n = -33554471 -> m = -39  
n = -33554439 -> m = -7  
n = -16777320 -> m = -104  
n = -16777288 -> m = -72  
n = -16777256 -> m = -40  
n = -16777224 -> m = -8  
n = -8388713 -> m = -105  
n = -8388681 -> m = -73  
n = -8388649 -> m = -41  
n = -8388617 -> m = -9  
n = -4194410 -> m = -106  
n = -4194378 -> m = -74  
n = -4194346 -> m = -42  
n = -4194314 -> m = -10  
n = -2097259 -> m = -107  
n = -2097227 -> m = -75  
n = -2097195 -> m = -43  
n = -2097163 -> m = -11  
n = -1048684 -> m = -108  
n = -1048652 -> m = -76  
n = -1048620 -> m = -44  
n = -1048588 -> m = -12  
n = -524397 -> m = -109  
n = -524365 -> m = -77  
n = -524333 -> m = -45  
n = -524301 -> m = -13  
n = -262254 -> m = -110  
n = -262222 -> m = -78  
n = -262190 -> m = -46  
n = -262158 -> m = -14  
n = -131183 -> m = -111  
n = -131151 -> m = -79  
n = -131119 -> m = -47  
n = -131087 -> m = -15  
n = -65648 -> m = -112  
n = -65616 -> m = -80  
n = -65584 -> m = -48  
n = -65552 -> m = -16  
n = 65552 -> m = 16  
n = 65584 -> m = 48  
n = 65616 -> m = 80  
n = 65648 -> m = 112  
n = 131089 -> m = 17  
n = 131121 -> m = 49  
n = 131153 -> m = 81  
n = 131185 -> m = 113  
n = 262162 -> m = 18  
n = 262194 -> m = 50  
n = 262226 -> m = 82  
n = 262258 -> m = 114  
n = 524307 -> m = 19  
n = 524339 -> m = 51

---

```

n = 524371 -> m = 83
n = 524403 -> m = 115
n = 1048596 -> m = 20
n = 1048628 -> m = 52
n = 1048660 -> m = 84
n = 1048692 -> m = 116
n = 2097173 -> m = 21
n = 2097205 -> m = 53
n = 2097237 -> m = 85
n = 2097269 -> m = 117
n = 4194326 -> m = 22
n = 4194358 -> m = 54
n = 4194390 -> m = 86
n = 4194422 -> m = 118
n = 8388631 -> m = 23
n = 8388663 -> m = 55
n = 8388695 -> m = 87
n = 8388727 -> m = 119
n = 16777240 -> m = 24
n = 16777272 -> m = 56
n = 16777304 -> m = 88
n = 16777336 -> m = 120
n = 33554457 -> m = 25
n = 33554489 -> m = 57
n = 33554521 -> m = 89
n = 33554553 -> m = 121
n = 67108890 -> m = 26
n = 67108922 -> m = 58
n = 67108954 -> m = 90
n = 67108986 -> m = 122
n = 134217755 -> m = 27
n = 134217787 -> m = 59
n = 134217819 -> m = 91
n = 134217851 -> m = 123
n = 268435484 -> m = 28
n = 268435516 -> m = 60
n = 268435548 -> m = 92
n = 268435580 -> m = 124
n = 536870941 -> m = 29
n = 536870973 -> m = 61
n = 536871005 -> m = 93
n = 536871037 -> m = 125
n = 1073741854 -> m = 30
n = 1073741886 -> m = 62
n = 1073741918 -> m = 94
n = 1073741950 -> m = 126
n = 2147483551 -> m = -97
n = 2147483583 -> m = -65
n = 2147483615 -> m = -33
n = 2147483647 -> m = -1

```

```

-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
move.l #n,m? -> moveq #n,ds      | Y | Y | ? | ? | ? | ? | 2
      move.l ds,m?      | | | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

-128 <= n <= 127

```

-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
move.l (ax),ay -> move.x ([ax],n),dz | - | - | ? | ? | ? | ? |

```



```

move.x n(ay), dz          | | | | |
-----+-----+-----+-----+-----+
move.l (ax), ay -> move.x ([ax]), dz | - | - | ? | ? | ? | ?
move.x (ay), dz          | | | | |
-----+-----+-----+-----+
move.l (bd.x, ax), dy ->          | - | - | ? | ? | ? | ?
      move.l bd.x, dy | | | | |
-----+-----+-----+-----+
move.l (n.w, ax), dy ->          | - | - | ? | ? | ? | ?
      move.l n(ax), dy | | | | |
-----+-----+-----+-----+
move.l (sp), (n, sp) -> rtd #n     | - | - | ? | ? | ? | ?
lea (n, sp), sp          | | | | |
rts                      | | | | |
-----+-----+-----+-----+
move.l (sp), 0(dx, sp) -> rtd dx   | - | Y | ? | ? | ? | ? | 6
lea 0(dx, sp), sp       | | | | |
rts                      | | | | |
-----+-----+-----+-----+
move.l 12(ax), 12(ay) -> move.l 6  | - | - | - | - | ? | ?
move.l 8(ax), 8(ay) (ax)+, (ay)+ | | | | |
move.l 4(ax), 4(ay)      | | | | |
move.l (ax)+, (ay)+      | | | | |
-----+-----+-----+-----+
move.l ax, -(sp) -> link ax, #n    | Y | Y | ? | ? | ? | ? | 4
move.l sp, ax            | | | | |
add.w #n, sp             | | | | |
-----+-----+-----+-----+
      -32767 <= n <= 32767
-----+-----+-----+-----+
move.l ax, -(sp) -> pea -n(ax)     | Y | Y | ? | ? | ? | ? | 0/4
sub*.l #n, (sp)          | | | | |
-----+-----+-----+-----+
move.l ax, -(sp) -> pea n(ax)      | Y | Y | ? | ? | ? | ? | 0/4
add*.l #n, (sp)          | | | | |
-----+-----+-----+-----+
move.l ax, az -> lea n(ax.l*4), az | - | - | ? | ? | ? | ?
asl.l #2, az             | | | | |
add.x #n, az             | | | | |
-----+-----+-----+-----+
      az=n+4*ax, -128<=n<=127
-----+-----+-----+-----+
move.l ax, az -> lea n(ax.l*8), az | - | - | ? | ? | ? | ?
asl.l #3, az             | | | | |
add.x #n, az             | | | | |
-----+-----+-----+-----+
      az=n+8*ax, -32767<=n<=32767
-----+-----+-----+-----+
move.l ax, sp -> unlk ax           | Y | Y | ? | ? | ? | ? | 2
move.l (sp)+, ax          | | | | |
-----+-----+-----+-----+
move.l ay, az -> lea n(ax, ay.l*4), az | - | - | ? | ? | ? | ?
asl.l #2, az             | | | | |
add.l ax, az             | | | | |
add.x #n, az             | | | | |
-----+-----+-----+-----+
      az=n+ax+4*ay, -32767<=n<=32767

```

```

-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
move.l ay,az -> lea n(ax,ay.l*8),az | - | - | ? | ? | ? | ?
asl.l #3,az | | | | |
add.l ax,az | | | | |
add.x #n,az | | | | |
-----+-----+-----+-----+-----+-----+
az=n+ax+8*ay, -32767<=n<=32767
-----+-----+-----+-----+-----+-----+
move.w #x,mn -> move.l #xy,mn | Y | Y | ? | ? | ? | 2/4/6
move.w #y,mn+2 | | | | |
-----+-----+-----+-----+-----+-----+
best if mn is longword aligned
-----+-----+-----+-----+-----+-----+
move.x #0,ax -> sub.l ax,ax | Y | Y | ? | ? | ? | 2/4
-----+-----+-----+-----+-----+-----+
move.x #n,ax -> lea n,ax | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+-----+
n <> 0
-----+-----+-----+-----+-----+-----+
move.x (rx,ay),az -> move.x ay,az | Y | Y | ? | ? | ? | 0
add.x rx,az | | | | |
-----+-----+-----+-----+-----+-----+
move.x ax,ay -> lea n(ax),ay | Y | Y | ? | ? | ? | 2/4
add.x #n,ay | | | | |
-----+-----+-----+-----+-----+-----+
-32767 <= n <= 32767
-----+-----+-----+-----+-----+-----+
move.x ax,az -> lea -n(ax,ay),az | Y | Y | ? | ? | ? | 2
sub.x #n,az | | | | |
add.x ay,az | | | | |
-----+-----+-----+-----+-----+-----+
az=n+ax+ay, n<=32767
-----+-----+-----+-----+-----+-----+
move.x ax,az -> lea n(ax,ay),az | Y | Y | ? | ? | ? | 2
add.x #n,az | | | | |
add.x ay,az | | | | |
-----+-----+-----+-----+-----+-----+
az=n+ax+ay, n<=32767
-----+-----+-----+-----+-----+-----+
movem.l (ax)+,registers | * | * | ? | ? | Y | *
-> move.l (ax)+,ry | | | | |
for each reg | | | | |
-----+-----+-----+-----+-----+-----+
movem.w *,dx -> move.w *,dx | Y | Y | ? | ? | ? | 0
ext.l dx | | | | |
-----+-----+-----+-----+-----+-----+
movem.x *,@ -> move.x *,@ | Y | Y | ? | ? | ? | 2
| | | | |
-----+-----+-----+-----+-----+-----+
@ = a single register, not (@=dx & .x=.w)
-----+-----+-----+-----+-----+-----+
movem.x @,* -> move.x @,* | Y | Y | ? | ? | ? | 2
| | | | |
-----+-----+-----+-----+-----+-----+
@ = a single register, status flags are wrong
-----+-----+-----+-----+-----+-----+
moveq #n,az -> lea n(ax,ay.l*2),az | - | - | ? | ? | ? | ?

```

```

add.x ay,az      | | | |
add.x ax,az      | | | |
add.x ay,az      | | | |
-----+-----+-----+-----+-----+-----
az=n+ax+2*ay, -128<=n<=127
-----+-----+-----+-----+-----+-----
mul*.l #1,dx -> (nothing)      | ! | ! | Y | Y | Y | 6
-----+-----+-----+-----+-----+-----
delete
-----+-----+-----+-----+-----+-----
mul*.l #10,dx -> add.l dx,dx    | ! | ! | ? | ? | ? | -2
  move.l dx,ds      | | | |
  asl.l #2,dx       | | | |
  add.l ds,dx       | | | |
-----+-----+-----+-----+-----+-----
mul*.l #12,dx -> asl.l #2,dx   | ! | ! | ? | ? | ? | -2
  move.l dx,ds      | | | |
  add.l dx,dx       | | | |
  add.l ds,dx       | | | |
-----+-----+-----+-----+-----+-----
mul*.l #2,dx -> add.l dx,dx    | ! | ! | ? | ? | ? | 4
-----+-----+-----+-----+-----+-----
mul*.l #3,dx -> move.l dx,ds   | ! | ! | ? | ? | ? | 0
  add.l dx,dx       | | | |
  add.l ds,dx       | | | |
-----+-----+-----+-----+-----+-----
mul*.l #5,dx -> move.l dx,ds   | ! | ! | ? | ? | ? | 0
  asl.l #2,dx       | | | |
  add.l ds,dx       | | | |
-----+-----+-----+-----+-----+-----
mul*.l #6,dx -> add.l dx,dx    | ! | ! | ? | ? | ? | -2
  move.l dx,ds      | | | |
  add.l dx,dx       | | | |
  add.l ds,dx       | | | |
-----+-----+-----+-----+-----+-----
mul*.l #7,dx -> move.l dx,ds   | ! | ! | ? | ? | ? | 0
  asl.l #3,dx       | | | |
  sub.l ds,dx       | | | |
-----+-----+-----+-----+-----+-----
mul*.l #9,dx -> move.l dx,ds   | ! | ! | ? | ? | ? | 0
  asl.l #3,dx       | | | |
  add.l ds,dx       | | | |
-----+-----+-----+-----+-----+-----
mul*.l #n,dx -> moveq #m,ds    | ! | ! | ? | ? | ? | 2
  asl.l ds,dx       | | | |
-----+-----+-----+-----+-----+-----
n is 2^m, 8<m<14
-----+-----+-----+-----+-----+-----
muls.l #0,dx -> moveq #0,dx    | ! | ! | ? | ? | ? | 4
-----+-----+-----+-----+-----+-----
muls.l #n,dx -> asl.l #m,dx   | ! | ! | ? | ? | ? | 4
-----+-----+-----+-----+-----+-----
n is 2^m, 1 <= m <= 8
-----+-----+-----+-----+-----+-----
muls.w #0,dx -> moveq #0,dx    | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----
muls.w #1,dx -> ext.l dx      | Y | Y | ? | ? | ? | 2

```

```

-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
muls.w #10,dx -> ext.l dx      | Y | Y | ? | ? | ? | -6
  add.l dx,dx      | | | | | |
  move.l dx,ds     | | | | | |
  asl.l #2,dx      | | | | | |
  add.l ds,dx      | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
muls.w #11,dx -> ext.l dx      | Y | Y | ? | ? | ? | -8
  move.l dx,ds     | | | | | |
  add.l dx,dx      | | | | | |
  add.l dx,ds      | | | | | |
  asl.l #3,dx      | | | | | |
  add.l ds,dx      | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
muls.w #12,dx -> ext.l dx      | Y | Y | ? | ? | ? | -6
  asl.l #2,dx      | | | | | |
  move.l dx,ds     | | | | | |
  add.l dx,dx      | | | | | |
  add.l ds,dx      | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
muls.w #2,dx -> ext.l dx      | Y | Y | ? | ? | ? | 0
  add.l dx,dx      | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
muls.w #3,dx -> ext.l dx      | Y | Y | ? | ? | ? | -4
  move.l dx,ds     | | | | | |
  add.l dx,dx      | | | | | |
  add.l ds,dx      | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
muls.w #5,dx -> ext.l dx      | Y | Y | ? | ? | ? | -4
  move.l dx,ds     | | | | | |
  asl.l #2,dx      | | | | | |
  add.l ds,dx      | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
muls.w #6,dx -> ext.l dx      | Y | Y | ? | ? | ? | -6
  add.l dx,dx      | | | | | |
  move.l dx,ds     | | | | | |
  add.l ds,dx      | | | | | |
  add.l ds,dx      | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
muls.w #7,dx -> ext.l dx      | Y | Y | ? | ? | ? | -4
  move.l dx,ds     | | | | | |
  asl.l #3,dx      | | | | | |
  sub.l ds,dx      | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
muls.w #9,dx -> ext.l dx      | Y | Y | ? | ? | ? | -4
  move.l dx,ds     | | | | | |
  asl.l #3,dx      | | | | | |
  add.l ds,dx      | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
muls.w #n,dx -> ext.l dx      | Y | Y | ? | ? | ? | 0
  asl.l #m,dx      | | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
n is 2^m, 1 <= m <= 8
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
muls.w #n,dx -> moveq #m,ds    | Y | Y | ? | ? | ? | -2
  ext.l dx         | | | | | |
  asl.l ds,dx      | | | | | |

```

```

-----+-----+-----+-----+-----+-----
n is 2^m, 8<m<14
-----+-----+-----+-----+-----+-----
mul.s.w #n,dx -> swap dx      | Y | Y | ? | ? | ? | -2
  clr.w dx      | |   | |   | |
  asr.l #(16-m),dx | |   | |   | |
-----+-----+-----+-----+-----+-----
n is 2^m, 8 <= m <= 15
-----+-----+-----+-----+-----+-----
mulu.l #0,dx -> moveq #0,dx   | ! | ! | ? | ? | ? | 4
-----+-----+-----+-----+-----+-----
mulu.l #n,dx -> lsl.l #m,dx   | ! | ! | ? | ? | ? | 4
-----+-----+-----+-----+-----+-----
n is 2^m, 1 <= m <= ?
-----+-----+-----+-----+-----+-----
mulu.w #0,dx -> moveq #0,dx   | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----
mulu.w #1,dx -> swap dx      | Y | Y | ? | ? | ? | -2
  clr.w dx      | |   | |   | |
  swap dx       | |   | |   | |
-----+-----+-----+-----+-----+-----
mulu.w #12,dx -> swap dx     | Y | Y | ? | ? | ? | -10
  clr.w dx      | |   | |   | |
  swap dx       | |   | |   | |
  asl.l #2,dx   | |   | |   | |
  move.l dx,ds  | |   | |   | |
  add.l dx,dx   | |   | |   | |
  add.l ds,dx   | |   | |   | |
-----+-----+-----+-----+-----+-----
mulu.w #2,dx -> swap dx      | Y | Y | ? | ? | ? | -4
  clr.w dx      | |   | |   | |
  swap dx       | |   | |   | |
  add.l dx,dx   | |   | |   | |
-----+-----+-----+-----+-----+-----
mulu.w #3,dx -> swap dx      | Y | Y | ? | ? | ? | -8
  clr.w dx      | |   | |   | |
  swap dx       | |   | |   | |
  move.l dx,ds  | |   | |   | |
  add.l dx,dx   | |   | |   | |
  add.l ds,dx   | |   | |   | |
-----+-----+-----+-----+-----+-----
mulu.w #5,dx -> swap dx      | Y | Y | ? | ? | ? | -8
  clr.w dx      | |   | |   | |
  swap dx       | |   | |   | |
  move.l dx,ds  | |   | |   | |
  asl.l #2,dx   | |   | |   | |
  add.l ds,dx   | |   | |   | |
-----+-----+-----+-----+-----+-----
mulu.w #6,dx -> swap dx      | Y | Y | ? | ? | ? | -10
  clr.w dx      | |   | |   | |
  swap dx       | |   | |   | |
  add.l dx,dx   | |   | |   | |
  move.l dx,ds  | |   | |   | |
  add.l ds,dx   | |   | |   | |
  add.l ds,dx   | |   | |   | |
-----+-----+-----+-----+-----+-----
mulu.w #7,dx -> swap dx      | Y | Y | ? | ? | ? | -8

```

```

    clr.w dx      | | | | |
    swap dx      | | | | |
    move.l dx,ds | | | | |
    asl.l #3,dx  | | | | |
    sub.l ds,dx  | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
mulu.w #9,dx -> swap dx      | Y | Y | ? | ? | ? | -8
    clr.w dx      | | | | |
    swap dx      | | | | |
    move.l dx,ds | | | | |
    asl.l #3,dx  | | | | |
    add.l ds,dx  | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
mulu.w #n,dx -> swap dx      | Y | Y | ? | ? | ? | -4
    clr.w dx      | | | | |
    swap dx      | | | | |
    lsl.l #m,dx   | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
n is 2^m, 1 <= m <= 8
-----+-----+-----+-----+-----+-----+-----+-----+-----+
mulu.w #n,dx -> swap dx      | Y | Y | ? | ? | ? | -2
    clr.w dx      | | | | |
    lsr.l #(16-m),dx | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
n is 2^m, 8 <= m <= 15
-----+-----+-----+-----+-----+-----+-----+-----+-----+
neg.x dx      -> add.x dx,dy  | Y | Y | Y | ? | ? | 2
sub.x dx,dy   | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
dx is trashed
-----+-----+-----+-----+-----+-----+-----+-----+-----+
neg.x dx      -> eor.x #n-1,dx | Y | Y | ? | ? | ? | 2
add.x #n,dx   | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
n is 2^m, dx < n
-----+-----+-----+-----+-----+-----+-----+-----+-----+
neg.x dx      -> sub.x dx,dy  | Y | Y | Y | ? | ? | 2
add.x dx,dy   | | | | |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
dx is trashed
-----+-----+-----+-----+-----+-----+-----+-----+-----+
nop -> (nothing) | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+-----+-----+-----+
remove nops
-----+-----+-----+-----+-----+-----+-----+-----+-----+
or.l #n,dx -> bset.l #b,dx   | Y | Y | ? | ? | ? | 2
-----+-----+-----+-----+-----+-----+-----+-----+-----+
n = 2^b (only 1 bit set)
-----+-----+-----+-----+-----+-----+-----+-----+-----+
sub*.x #0,dx -> tst.x dx     | Y | Y | ? | ? | ? | 2/4
-----+-----+-----+-----+-----+-----+-----+-----+-----+
sub.x #n,* -> addq.x #-n,*   | Y | Y | ? | ? | ? | 2/4
-----+-----+-----+-----+-----+-----+-----+-----+-----+
-8 <= n <= -1
-----+-----+-----+-----+-----+-----+-----+-----+-----+
sub.x #n,* -> subq.x #n,*   | Y | Y | ? | ? | ? | 2/4
-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

```

if 1 <= n <= 8
-----+-----+-----+-----+-----+-----+-----+-----+
sub.x #n,ax -> lea -n(ax),ax      | Y | Y | ? | ? | ? | 0/2
-----+-----+-----+-----+-----+-----+-----+
-32767 <= n <= -9, 9 <= n <= 32767
-----+-----+-----+-----+-----+-----+-----+
subq.l #n,ax -> subq.w #n,ax     | Y | Y | ? | ? | ? | 0
-----+-----+-----+-----+-----+-----+-----+
subq.w #1,dx -> db<cc> dx,??     | y | y | ? | ? | ? | -2
b<cc> ?? b<cc> ??                | | | | | |
-----+-----+-----+-----+-----+-----+-----+
if dx=0 then will be slower
-----+-----+-----+-----+-----+-----+-----+
subq.w #1,dx -> dbf dx,??        | Y | Y | ? | ? | ? | -2
bra ?? bra ??                    | | | | | |
-----+-----+-----+-----+-----+-----+-----+
if dx=0 then will be slower
-----+-----+-----+-----+-----+-----+-----+
tst.w dx -> dbra dx,??          | y | y | ? | ? | ? | 2
bne ??                            | | | | | |
-----+-----+-----+-----+-----+-----+-----+
dx will be trashed
-----+-----+-----+-----+-----+-----+-----+

```

---

#### H I N T S   &   T I P S

---

This new section is for stuff that cannot be included in the above tables. This can include pipelining optimizations and other stuff.

020+ Sequential memory accesses can cause pipeline stalls, so try and rearrange code so memory accesses do not immediately follow each other. The same problem occurs if an address register updated in one line is accessed in the next line.

ALL Include small routines as macros, because inline routines will be much faster, and in extreme cases smaller.

ALL If a subroutine is only called from one position, either move it inline, or only use jmp/bra commands.

---

#### C O N C L U S I O N

---

There are the optimizations i've come up with so far. If you could check what i've done, and report any errors, that would make this list better. I only have so much time to spend on this, and many hands make light work. Also, stats (and more optimizations) for 68020+ CPU's would be welcomed. Currently this list is only for simple peephole optimization stuff, but I will hopefully get around to more extensive optimizations. Pipeline optimization is on the way, so look out. Any info on the 68020+ pipelines

---

would be appreciated.

Optimizations with ?question-marks? in the boxes next to them, I do not have the data to check yet.

The latest version of the asp68k archive is available by anonymous ftp from ftp.mq.edu.au in the /home/mglew/ directory or by calling Technophilia BBS on +61 2 807 3563 (or (02) 807 3563 in Australia).

```
=====
EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF
=====
```

## 1.123 Generic Assembler Documentation

```
TO      Specify output filename
WITH    Specify a file that contains a list of options to be
used.  There may only be one WITH file per assembly, and this option may
only be used on the command line.  Such files may be created using the
editor's command for saving an assembler settings file.  Be careful when
using this as any main file that is set via the editor will be included
in the WITH file.
-.      (or QUIER) disable assembly messages
-B      No binary file will be created.
-C      Caseinsensitive labels (OPT NOCASE)
-D      Debug (OPT DEBUG)
-E      allows labels to be set; assignments must be separated
by commas.  Labels will be set as if they were on line 2 of the main
source file.
-H      (or HEADER) specify the pre-assembled header files that
are to be loaded before assembly starts.  Multiple files may be
separated with a comma.
-I      (or INCDIR) specify include directories to be searched
(follow _immediatly_ with path).  These directories will be searched
when the assembler is opening include files.  These should normally be
terminated with a slash.
-L      Amiga@ linkable code (OPT ALINK)
-L6     output Motorola S-records (OPT SREC)
-M      use low memory (slower) assembly mode.  See the section
on integrated options in the previous chapter.  _Not_ the same as OPT M+.
-O      specify output filename.
-P      specify listing filename, defaults to source filename
with extension of .LST
-Q      pasue for key press after assembly.
-S      include a symbol table at the end of listing
-T      specifies tab setting for listing
-V      specify options as if they were specified using OPT on
the second line of the main source file.
-X      use just exported labels in debugging (OPT XDEBUG)
-Z      enable listing on pass 1.  The information in the code
filed may be incorrect but this can be used to find mistakes when
omitting an ENDC (OPT LIST1).  This is provided for backwards
compatibility; OPT TRACEIF can normally be used to find such errors more
```



quickly.

---