Mops
Mike's Object-oriented Programming System

Version 2.6
Part IV
Assembler

Mops is an object-oriented programming system, derived from the Neon language developed by Charles Duff and sold by Kriya, Inc. Kriya have discontinued support for Neon, and have released all the source code into the public domain, retaining only the ownership of the name Neon.

Mops implemented by: Michael Hore
Able assistance from: Doug Hoffman
Greg Haverkamp
Xan Gregg

Documentation updated: Version 2.6, June1995
Documentation formatted by: Craig Treleaven

Printing this document
This document is in Microsoft Word Version 5.1 format and uses the fonts Times, Courier, and Helvetica, only. It is formatted using the Laserwriter 7 driver for US Letter paper, portrait orientation, with fractional widths enabled. If you want to print any other way, you will probably need to repaginate and regenerate the table of contents. See below.
Almost every paragraph in this document is formatted using a Word style. Formatting is consistent throughout and can be reformatted in moments this way.

Viewing on-line
Of course, you can read the whole manual on-screen. Word's Find… command can help to locate items of interest. One other technique is useful but not well known. Use the Outline View and click the “2” in the ruler at the top of the screen. Word will then show the chapters and the sub-headings within. Whichever line is at the top of the window in outline view will become the line at the top of the window when you switch back to Normal View. By scrolling in Outline View, you can quickly find the section of interest and position the window for reading in Normal View.

Two-sided printing
As shipped, this document is formattted for 2-sided printing to save paper. If you haven't printed two-sided documents with your printer before, you might want to practise with the first few pages before sending the whole thing. On most printers, you need to use Word's option to print first the odd numbered pages (in the Print… dialog), reload the paper and then print the even numbered pages.

Single-sided printing
If you don't want to bother with two-sided printing, use the Document dialog and make the Gutter margin zero. If you adjust the Left and Right margins so the printable width is still 6.5 inches, the page breaks should stay in the same places. Blank pages may pop out here and there as all chapters start on an odd-numbered page.

A4 Paper
If you select A4 paper in the Page Setup… dialog, the page breaks will change. Regenerate the table of contents, as below. As far as I can tell, the paragraph styles all do the right thing and ajust to the paper width. Well, all except one: the header on odd-numbered pages will extend a quarter inch into the margin because the tab stop is at 6.5 inches. Redefine the Header style to set it to 6.25, if you feel the need.

Table of Contents
Use the Table of Contents… dialog to collect headings from level 1 to level 3 for the Table of Contents. Figure captions have Heading 5 style, but I didn't see a reason to create a table of figures. Similarly, the instructions in Chapter 3 have Heading 4 style and the error codes in the appendix have Heading 6 style.

Chapter 1—Using the Assembler

About this chapter

With the Mops Assembler module you can write colon definitions and method definitions in assembly code and you can also reference Mops data and executable words. This chapter explains how the assembler interfaces with Mops. The various Macintosh models use a Motorola 68000, 68020, 68030 or 68040 microprocessor, and the assembler syntax is based on standard 68000 assembly language. This chapter is not a tutorial and assumes a basic knowledge of 680x0 assembly programming. We recommend you obtain the User's Manual for your particular microprocessor, published by Motorola. Only the 68000 instruction set is supported by the Assembler. The 68020 and later have some additional instructions and addressing modes. If you were to use these, however, your program would not run on earlier Macs which have a 68000 processor (Mac Plus, SE, Classic, and PowerBook 100).

*** WARNING ***

While the Assembler gives you absolute control over the machine, it may give you problems in when (not if) you want to run on Power Macs. Your 680x0 code, assembled with this assembler, will only be able to run under emulation on the Power Mac, which means that later when we have a native Power Mac version of Mops, you would have to make a mixed-mode call to your assembled code. This will be far slower than if you had written the code in high-level Mops to start with. Mops code on the 680x0 is quite fast anyway, so you should really only be using assembly code if you are doing some very low-level machine-specific things. You may have perfectly good reasons for doing this, and if so, fine.

Getting started

Like most assemblers, the Mops Assembler is a two pass assembler. It does not run interactively. Assembly code must be in a source file. The normal Mops "//" load command will load and assemble it.

Recompiling the Assembler

The assembler is a standard Mops module, called AsmMod. It may be recompiled in the usual way, namely

```
compile: asmMod
```

The source file AsmMod.txt, which is compiled by the above command, consists of a series of // commands to load the actual assembler source files. These are located in the folder Asm Source, in the Mops Source folder.

Assembler colon definitions
To write a colon definition with the assembler, just use ":code" for ":" and ";code" for ";". The name of the new word follows the ":code" on the same line.
Example:

```
:code demo1    \ adds two numbers and puts the sum onto the stack
        move.l #4,D0            \ puts 4 into register D0
        add.l   #8,D0                   \ adds 8 to register D0
        move.l D0,-(A6)         \ pushes contents of register D0 onto data
                        \  stack
;code

demo1
. cr
12
```

Assembler method definitions
To write a method definition with the assembler, just use ":mcode" for ":m" and ";mcode" for ";m". The name of the new selector follows the ":mcode" on the same line. In the following example, the "put:" and the "asmput:" of class demo2 are synonymous, as are the "get:" and the "asmget:". You will see that in a method, the base address of the current object is in register A2. See Chapter 2 for more information on addressing modes and Mops's usage of registers.
Example:

```
:class DEMO2  super{ object }

  var int1

:m PUT:   put: int1  ;m

:m GET:   get: int1  ;m

:mcode  ASMGET:
        move.l (A2),-(A6)
;mcode

:mcode  ASMPUT:
        move.l (A6)+,(A2)
;mcode

;class

demo2 test2

put: test2
asmget: test2 . cr
26
34 asmput: test2
get: test2 . cr
34
```

Accessing the dictionary
There is a special syntax in the Mops Assembler that obtains the address of a word in the dictionary, and assembles an An-relative memory reference. An will be A3, A4 or A5 depending on exactly where the addressed word is in the dictionary. The assembler looks after these details. The syntax is "dic[name]". If the word in the dictionary is a Mops object, you can obtain the address of the beginning of the object's ivars with "dicobj[name]". The following example shows this usage, and also how an assembler defined word (demo3) can be accessed in Mops just like any regular definition.

```
var     FUN
0       value   AVALUE
```

```
:code demo3
	move.l #1234,dic[avalue]
	moveq		#10,D0
	move.l D0,dicobj[fun]
;code

: test3
  1  -> avalue   0 put: fun
  demo3
  avalue . cr  get: fun . cr  ;

test3
1234
10
```

The following example shows an array being accessed with dicobj[name]. Five long words of data starting at the absolute address of the array "joe" plus 6 is moved to five registers. The contents of the five registers are then pushed onto the stack. The displacement of 6 is necessary because the first 6 bytes of an array are for record keeping.

```
5 array joe

:code demo4
	lea	dicobj[joe],A0
	movem.l		6(A0),D0-D2/A0-A1
	movem.l		D0-D2,A0-A1-(A6)
;code

: test4
  7 fill: joe
  demo4
  . . . . . cr  ;

test4
7 7 7 7 7
```

Alternatively, the first two lines of demo4 could have been written:

```
lea	6(dicobj[edmund]),A0
movem.l		(A0),D0-D2/A0-A2
```

As you can see from this example, you can add a displacement to a dictionary address. This displacement is incorporated at compile time into the displacement assembled into the instruction.

If you require a PC-relative mode to be generated (which might happen, for example, in an interrupt routine where the A registers aren't set up for Mops when the code executes), then use e.g.

```
MOVE.L		rel[name],D0
```

Note that a memory store or test can't use this mode on a 68000 processor, but can on a 68020 or later.

Executing Mops-defined words

Words defined with ":code" become normal Mops words in every respect. All Mops words are called with a JSR or BSR instruction, as in this example:

```
:code demo5
	move.l #55,-(A6)
	jsr	dic[dup]

: test5  demo5 . . cr  ;

test5
55 55
```

In this example, you could equally well have used

```
bsr	dic[dup]
```

provided dup was not more than 32K away in the dictionary.

As all code called with JSR/BSR, Mops definitions must return with an RTS instruction. However you don't need to put

RTS explicitly at the end of your assembly definitions, since when the Assembler encounters ;code or ;mcode it always assembles an RTS. If, however, you have a routine which returns from the middle, you will have to put RTS there.

Toolbox calls

The Mops Assembler simplifies doing register based toolbox calls. To do a toolbox call, have the needed parameters in the proper locations (registers or the data stack), use "call" as the opcode and the toolbox name as the operand. There are many examples of toolbox calls in the floating point code. Details on toolbox calling can be found in your Mops manual and in Inside Mac. The following example tests two strings to find if they are equal. Inside Mac describes which parameters go into which registers.

```
:code  demo8  \ tests two "addr len" strings for being equal
        move.l (A6)+,D0        \ pop len of first string
        swap D0                \    onto high order word of D0
        movea.l (A6)+,A0       \ pop addr of first string
        or.l (A6)+,D0  \ pop len of 2nd str onto low order word of D0
        movea.l (A6)+,A1       \ pop addr of second string
        exg A6,A7
        call cmpstring \ IF equal THEN returns 0 ELSE 1
        exg A6,A7
        move.l D0,-(A6)        \ push answer onto stack
;code

: test8  " Mops"  " Assembler"  demo8  . cr  ;

test8
1
```

For a stack-based trap, there are a couple of extra points to note. Since we use A6 as the data stack, parameters will normally be there. But the Mac uses A7 as its only stack pointer. The easiest course of action is to precede and follow any stack-based Toolbox call with an exg instruction, thus:

```
exg     A6,A7
call    <someTrap>
exg     A6,A7
```

If the Toolbox call uses A5, which all QuickDraw calls do, then we have an extra problem in that Mops uses A5 to address modules. In these situations you will need Whenever a long word is referenced, all 32 bits are being referenced. With a word sized operand the low order 16 bits are intended. The low order 8 bits are used in a byte reference.

Data Registers

The 8 data registers (D0 - D7) are each 32 bits wide and are primarily used to hold 32 bit (long word) data, 16 bit (word) data, and 8 bit (byte) data. They can also be used for indexing.

Address Registers

The first 7 address registers (A0 - A6) are 32 bits wide and are used to hold addresses although they can also be used for indexing. For addressing references the low order 24 bits are used.

A7 is the stack pointer (SP). When the system is in supervisor mode, it is the supervisor stack pointer, and in user mode it is the user stack pointer. The Macintosh operates in supervisor mode.

Uses of Data and Address Registers

Some of the 68000's registers are used by Mops and the Macintosh for themselves. See Figure 2 for a memory map of the Macintosh while Mops is running.

You may safely manipulate in a ":code" or a ":mcode" definition all D registers, as well as A0 and A1. If you are not in a :mcode definition, A2 is also free. But note that you can't rely on data in ANY register remaining there across calls of your definition. The assignments of each register are:

D0	Free
D1	Free
D2	Free
D3	Free
D4	Free
D5	Free
D6	Free
D7	Free

A0	Free
A1	Free
A2	Address of first ivar in current object in a :mcode
A3	Lobase pointer
A4	Hibase pointer
A5	Module base pointer, or -1 if not in a module
A6	Data stack pointer. Also may be referred to as SP.
A7	Return stack pointer (in supervisor mode)

Figure 1—Registers

Figure 2—Memory Map

Condition Codes

There are five condition codes used by the Mops Assembler and most instructions affect at least one of them. Bits 0 through 4 are the condition codes and they are the only bits on the user byte that are used. See chapter 3 for which instructions affect which condition codes. They are used for various tests for conditional branching and setting bytes. The codes are:

Code	Name	Bit Location	Description
X	eXtend	4	Used for multiprecision computations. If affected then usually set as the C code is set.
N	Negative	3	On if most significant bit of result is on, otherwise off.
Z	Zero	2	On if result is zero, otherwise off.
V	oVerflow	1	On if there is an arithmetic overflow, otherwise off. If on the result is probably wrong.
C	Carry	0	On if a carry is generated by the most significant digit in an addition, or a borrow is generated by the most significant digit in a subtraction, otherwise off.

Interrupt Mask

The interrupt mask is used to disable interrupts at various levels. It occupies bits 8, 9, and 10 of the status register. Interrupt levels range from 1 (001) to 7 (111). Bit 8 is the low bit, 1.e., for interrupt level 1 bit 8 is set and bits 9 and 10 are not set. If the interrupt priority of an interrupt is less than or equal to the interrupt mask, then the interrupt exception is postponed. An interrupt level of 7 (111) will not be postponed by the interrupt mask even if the mask is 7 (111). Problems like loss of power are level 7 (111). A mask of 0 (000) means no interrupts are postponed. 0 (000) is the default for the interrupt mask.

Supervisor Bit

If the supervisor bit is on then the machine is in supervisor mode and if the supervisor bit is off, then it is in user mode. Either mode may be in effect while Mops is running, depending on the actual Mac model and whether virtual memory is in use. This bit (13) affects register A7. A few instructions, which are not usable in user mode, are known as privileged instructions.

Trace Bit

If bit 15 is on then the trace facility is on. After every instruction while bit 15 is on there will be a trap to the currently installed debugger.

Data Addressing Modes

Mode	Operand Syntax
Data Register Direct	Dx
Address Register Direct	Ax
Other Register Direct	CCR, SR, USP, <register list>
Address Register Indirect	(Ax)
Address Register Indirect with PostIncrement	(Ax)+
Address Register Indirect with PreDecrement	-(Ax)
Address Register Indirect with Displacement	d(Ax)
Address Register Indirect with Displacement and Index	d(Ax,Ry)
Program Counter Indirect with Displacement	d(PC)
Program Counter Indirect with Displacement and Index	d(PC,Ry)
Absolute Short Address	#xx
Absolute Long Address	#xxxx
Immediate Data	#<data>
Implicit Reference	NA

Notes:

NA	not applicable

( )	indirect
-( )	predecrement indirect
( )+	postincrement indirect
d	displacement
Ax	address register
Dx	data register
Ry	address register or data register
CCR	user byte of status register
SR	status register
USP	user stack pointer
<register list>	group of registers
PC	program counter
#xx	word sized immediate address
#xxxx	long word sized immediate address
#<data>	immediate data

Table 1—Addressing Modes

Addressing Modes

The 68000 has a rich set of addressing modes. An effective address is the address computed at execution time using the addressing mode. The contents of the effective address are what the operation works on. If the operand has the size of a byte, an address, even or odd, may be accessed. If the operand size is word or long word, only even addresses maybe accessed.

Data Register Direct

The operand is a data register.

```
MOVE D0,D1
```

Address Register Direct

MOVEA D0,A1
Other Register Direct
With the MOVE instruction, the operands can be CCR (user byte of the status register, i.e. condition codes), SR (status register), or USP (user stack pointer while in supervisor mode). See Chapter 3 for more details on the MOVE instruction. One of the two operands of the MOVEM instruction is a list of registers. The registers should be listed in the order: D0 thru D7, A0 thru A7. Note D0/D2 loads D0 and D2; D0-D2 loads D0, D1, and D2. A "-" can only be used to group D registers or A registers but it cannot group D and A registers together.
MOVE D0,SR
MOVEM 4(A0,D1.L),D6-D7/A0-A2
Address Register Indirect
The effective address is the content of the address register.
NEG (A0)
Address Register Indirect with PostIncrement
The effective address is the content of the address register. After the operand is computed the register is incremented by 1, 2, or 4 depending on the operand size. If A7 is used and the operand size is byte then the operand is still byte but the increment is 2. If another address register is used with an operand size of a byte then the increment is 1.
CLR   (A6)+
Address Register Indirect with PreDecrement
Before the operand is computed the address register is decremented by 1, 2, or 4 depending on the operand size. The effective address is the content of the register after decrementation. If A7 is used and the operand size is byte, then the operand is still byte but the decrement is 2. If another address register is used with an operand size of a byte then the decrement is 1.
CLR   -(A6)
Address Register Indirect with Displacement
The effective address is the sum of the content of the address register and the 16 bit two's complement integer. Hex data for all displacements can be specified using a "$".
CLR   4(SP)
CLR $4(SP)
CLR $-4(SP)
Address Register Indirect with Displacement and Index
The effective address is the sum of the content of the address register, the 16 bit two's complement integer, and the index register. The index register can be a data or an address register and it can be a word or a long word in size.
LEA   4(A0,D1.L),A1
Program Counter Indirect with Displacement
The effective address is the sum of the content of the program counter and the 16 bit two's complement integer.
CLR   4(PC)
Program Counter Indirect with Displacement and Index
The effective address is the sum of the content of the program counter, the 16 bit two's complement integer, and the index register. The index register can be a data or an address register and it can be a word or a long word in size.
LEA   4(PC,D1.L),A1
Absolute Short Address
The effective address is specified absolutely. The address can no be larger than 16 bits.
Absolute Long Address
The effective address is specified absolutely. The address is larger than 16 bits.
Immediate Data
The data is specified absolutely. The maximum size depends on the opcode. Hex data can be specified using a "$".
MOVE#6,D0
MOVE#-6,D0
MOVE#$6,D0
MOVE#$-6,D0
Implicit Reference
The operands needed are known by the opcode. No operands are given.
RTS

Chapter 3—Instructions

About this chapter
This chapter explains the machine instructions for the Mops assembler module. The machine instructions are based on the

standard 68000 instruction set syntax. The two significant differences are: 1) you can call the addresses of Mops objects as described in chapter 2, and 2) the default operand size is L (long word). There is a table of all the machine instructions and another of the condition fields for instructions Bcc, DBcc, and Scc. Following the tables are written descriptions of each instruction giving details which the tables do not cover.
Motorola 68000 Instructions

Condition Codes

Opcode
Opcode Description
Operand Size

Operand Syntax
X
N
Z
V
C

ABCD
add decimal with extend
B

Dy,Dx
-(Ay),-(Ax)
A
?
D
?
M

ADD
add binary
B
W
L
<ea>,Dx
Dx,<ea>
A
B
C
F
K

ADDA

add address

W
L
<ea>,Ax
-
-
-
-
-

ADDI
add immediate
B
W
L
#<data>,<ea>
A
B
C
F
K

ADDQ
add quick
B
W
L
#<data>,<ea>
A
B
C
F
K

ADDX
add extended
B
W
L
Dy,Dx
-(Ay),-(Ax)
A
B
C
F
K

AND
AND logical
B
W
L
<ea>,Dx
Dx,<ea>
-

B
C
0
0

ANDI
AND immediate
B
W
L
#<data>,<ea>
-
B
C
0
0

ASL
arithmetic shift left
B
W
L
Dx,Dy   (r=0)
-
B
C
0
0

#<data>,Dy (r<>0)
<ea>
A
B
C
J
P

ASR
arithmetic shift right
B
W
L
Dx,Dy   (r=0)
-
B
C
0
0

#<data>,Dy (r<>0)
<ea>
A
B
C
0
R

Bcc
branch conditionally
B/S
W

<label>
-
-
-
-
-

BCHG
test a bit and change
B

L
Dx,<ea>
#<data>,<ea>
-
-
E
-
-

BCLR
test a bit and clear
B

L
Dx,<ea>
#<data>,<ea>
-
-
E
-
-

BRA
branch always
B/S
W

<label>
-

-
-
-
-

BSET
test a bit and set
B

L
Dx,<ea>
#<data>,<ea>
-
-
E
-
-

BSR
branch to subroutine
B/S
W

<label>
-
-
-
-
-

BTST
test a bit
B

L
Dx,<ea>
#<data>,<ea>
-
-
E
-
-

CHK
check register against bounds

W

<ea>,Dx
-
V
?
?
?

CLR

clear an operand
B
W
L
<ea>
-
0
1
0
0

CMP
arithmetic compare
B
W
L
<ea>,Dx
-
B
C
G
L

CMPA
arithmetic compare address

W
L
<ea>,Ax
-
B
C
G
L

CMPI
compare immediate
B
W
L
#<data>,<ea>
-
B
C
G
L

CMPM
compare memory
B
W
L
(Ay)+,(Ax)+
-
B
C

G
L

DBcc
test condition, decrement and branch

W

Dx,<label>
-
-
-
-
-

DIVS
signed divide

W

<ea>,Dx
-
B
C
H
0

DIVU
unsigned divide

W

<ea>,Dx
-
B
C
H
0

EOR
exclusive OR logical
B
W
L
Dx,<ea>
-
B
C
0
0

EORI
exclusive OR immediate
B
W
L

#<data>,<ea>
-
B
C
0
0

EXG
exchange registers

L
Rx,Ry
-
-
-
-
-

EXT
sign extend

W
L
Dx
-
B
C
0
0

JMP
jump
NA

<ea>
-
-
-
-
-

JSR
jump to subroutine
NA

<ea>
-
-
-
-
-

LEA

load effective address

L
<ea>,Ax
-
-
-
-
-

LINK
link and allocate
NA

Ax,#<displacement>
-
-
-
-
-

LSL
logical shift left
B
W
L
Dx,Dy (r=0)
-
B
C
0
0

#<data>,Dy (r<>0)
<ea>
A
B
C
0
P

LSR
logical shift right
B
W
L
Dx,Dy (r=0)
-
B

C
0
0

#<data>,Dy (r<>0)
<ea>
A
B
C
0
R

MOVE
move data from source to destination
B
W
L
<ea>,<ea>
-
B
C
0
0

MOVE to CCR
move to condition codes

W

<ea>,CCR
S
S
S
S
S

MOVEto SR
move to the status register

W

<ea>,SR
S
S
S
S
S

MOVE from SR
move from the status register

W

SR,<ea>
-
-
-
-
-

MOVE
move user stack pointer

L
USP,Ax
-
-
-
-
-

USP

Ax,USP

MOVEA
move address

W
L
<ea>,Ax
-
-
-
-
-

MOVEM
move multiple registers

W
L
<register list>,<ea>
<ea>,<register list>
-
-
-
-

-

MOVEP
move peripheral data

W
L
Dx,d(Ay)
d(Ay),Dx
-
-
-
-
-

MOVEQ
move quick

L
#<data>,Dx
-
B
C
0
0

MULS
signed multiply

W

<ea>,Dx
-
B
C
0
0

MULU
unsigned multiply

W

<ea>,Dx
-
B
C
0
0

NBCD
negate decimal with extend
B

<ea>
A
?
D
?
N

NEG
two's complement negation
B
W
L
<ea>
A
B
C
I
O

NEGX
negate with extend
B
W
L
<ea>
A
B
C
I
O

NOP
no operation
NA

NA
-
-
-
-
-

NOT
logical complement
B
W
L
<ea>
-
B
C
0
0

OR

inclusive OR logical
B
W
L
<ea>,Dx
Dx,<ea>
-
B
C
0
0

PEA
push effective address

L
<ea>
-
-
-
-
-

RESET
reset external devices
NA

NA
-
-
-
-
-

ROL
rotate without extend left
B
W
L
Dx,Dy (r=0)
-
B
C
0
0

#<data>,Dy (r<>0)
<ea>
-

B
C
0
P

ROR
rotate without extend right
B
W
L
Dx,Dy (r=0)
-
B
C
0
0

#<data>,Dy (r<>0)
<ea>
-
B
C
0
R

ROXL
rotate with extend left
B
W
L
Dx,Dy (r=0)
-
B
C
0
Q

#<data>,Dy (r<>0)
<ea>
A
B
C
0
P

ROXR

rotate with extend right
B
W
L
Dx,Dy (r=0)
-
B
C
0
Q

#<data>,Dy (r<>0)
<ea>
A
B
C
0
R

RTE
return from exception
NA

NA
T
T
T
T
T

RTR
return and restore condition codes
NA

NA
T
T
T
T
T

RTS
return from subroutine
NA

NA
-
-
-

-
-

SBCD
subtract decimal with extend
B

Dy,Dx
-(Ay),-(Ax)
A
?
D
?
N

Scc
set according to condition
B

<ea>
-
-
-
-
-

STOP
stop program execution
NA

#<data>
U
U
U
U
U

SUB
subtract binary
B
W
L
<ea>,Dx
Dx,<ea>
A
B
C
G
L

SUBA
subtract address

W
L
<ea>,Ax
-
-
-
-
-

SUBI
subtract immediate
B
W
L
#<data>,<ea>
A
B
C
G
L

SUBQ
subtract quick
B
W
L
#<data>,<ea>
A
B
C
G
L

SUBX
subtract with extend
B
W
L
Dy,Dx
-(Ay),-(Ax)
A
B
D
G
L

SWAP
swap register halves

W

Dx
-
B
C

0
0

TAS
test and set an operand
B

<ea>
-
B
C
0
0

TRAP
trap
NA

#<vector>
-
-
-
-
-

TRAPV
trap on overflow
NA

NA
-
-
-
-
-

TST
test an operand
B
W
L
<ea>
-
B
C
0
0

UNLK
unlink
NA

Ax

-
-
-
-
-

Notes (other than for condition codes):

B	byte sized operand
W	word sized operand
L	long word operand
S	short branch (byte displacement)
NA	not applicable
<ea>	effective address
#<data>	immediate data (size depends on instruction)
#<vector>	0 - 15
#<displacement>	16 bit two's complement integer
#<register list>	registers to be moved
<label>	user defined label
( )	indirect
-( )	predecrement indirect
( )+	postincrement indirect
d(Ax)	address register with displacement
Ax,Ay	address register

Dx,Dy
data register

Rx,Ry
address register or data register

CCR
condition code byte of status register

SR
status register

USP
user stack pointer

Condition codes:
N
Negative

Z
Zero

V
Overflow

C
Carry

X
Extend

Notes on condition codes:
?
Undefined after operation

-
Unaffected by the operation

1
Set

0
Cleared

A
X <- C

B
N <- Rm

C
Z <- ~Rm * … * ~R0

D
Z <- Z * ~Rm * … * ~R0

E
Z <- ~Rm

F
V <- Sm * Dm * ~Rm + ~Sm * ~Dm * Rm

G
V <- ~Sm * Dm * ~Rm + Sm * ~Dm * Rm

H
V <- Division Overflow

I
V <- Dm * Rm

J
V <- Dm * (~Dm-1 + … + ~Dm-r)
+ ~Dm * (Dm-1 + … + Dm-r)

K
C <- Sm * Dm + ~Rm * Dm + Sm * ~Rm

L
C <- Sm * ~Dm + Rm * ~Dm + Sm * Rm

M
C <- Decimal Carry

N
C <- Decimal Borrow

O
C <- Dm + Rm

P
C <- Dm-r+1

Q
C <- X

R
C <- Dr-1

S
Set according to source operand

T
Set according to contents of word on the stack

U
Set according to immediate operand

V
Set if Dx <0, Clear if Dx > <ea>
otherwise undefined

Notes on notes on condition codes:
Sm
most significant bit of source operand before operation

Dm
most significant bit of destination operand before operation

Rm
most significant bit of result after operation

r
shift amount

n
bit number

Condition fields
(Use for test code cc in Bcc, DBcc, and Scc)
Test Code

Operation

Test to Return True

CC
carry clear
~C

CS
carry set
C

EQ
equal
Z

F
always false
0

GE
greater than or equal
N * V + ~N * ~V

GT
greater than
N * V * ~Z + ~V * ~V * ~Z

HI
high
~C * ~Z

LE
less than or equal
Z + N * ~V + ~N * V

LS
low or same
C + Z

LT
less than
N * ~V + ~N * V

MI
minus
N

NE
not equal
~Z

PL
plus
~N

T
always true
1

VC
no overflow
~V

VS
overflow
V

Figure 3—Shifts and Rotates
Instruction Descriptions
ABCD Add Decimal with Extend
This instruction adds the contents of the two operands and the contents of the X bit together with binary coded decimal arithmetic and places the result into the second operand.
ADD Add Binary
This instruction adds the contents of the two operands together with two's complement binary arithmetic and places the result into the second operand.
ADDA Add Address
This instruction adds the contents of the two operands together with two's complement binary arithmetic and places the result into the second operand.
ADDI Add Immediate
This instruction adds the contents of the two operands together with two's complement binary arithmetic and places the result into the second operand. The immediate data can be up to 32 bits long, depending on the operand size.
ADDQ Add Quick
This instruction adds the contents of the two operands together with two's complement binary arithmetic and places the result into the second operand. The immediate data can be the integers 1 through 8.
ADDX Add Extended
This instruction adds the contents of the two operands and the X bit together with two's complement binary arithmetic and

places the result into the second operand.

AND AND Logical

This instruction performs a bitwise logical AND on the contents of the two operands and places the result into the second operand.

ANDI AND Immediate

This instruction performs a bitwise logical AND on the contents of the two operands and places the result into the second operand. The immediate data can be up to 32 bits long, depending on the operand size. With byte or word operand size the second operand can be the status register. If byte, then only the condition codes are affected. If word, then it is a privileged operation and the whole status register is affected.

ASL Arithmetic Shift Left

If there are two operands, then this instruction arithmetically shifts to the left the contents of the second operand by the amount specified in the first operand. If the first operand is a data register, then the distance shifted is in the right most six bits of the first operand. If the first operand is immediate data, then the immediate data can be the integers 1 to 8. If there is only one operand, then the contents of the operand will be arithmetically shifted to the left only one bit and the operand size is limited to word. Zeros are shifted into the low order bit and the last value shifted out of the high order bit is placed into the C and X bits. See Figure 3. This instruction is identical to LSL.

ASR Arithmetic Shift Right

If there are two operands, then this instruction arithmetically shifts to the right the contents of the second operand by the amount specified in the first operand. If the first operand is a data register, then the distance shifted is in the right most six bits of the first operand. If the first operand is immediate data, then the immediate data can be the integers 1 to 8. If there is only one operand, then the contents of the operand will be arithmetically shifted to the right only one bit and the operand size is limited to word. The high order bit is duplicated with each shift of a bit and the last value shifted out of the low order bit is placed into the C and X bits. See Figure 3.

Bcc Branch Conditionally

This instruction causes the program execution to continue at the user specified label if the condition is met. The condition is specified by the cc which one of the codes in Table 3. Two exceptions are F and T; those conditions codes re not supported in Bcc (BRA can be used).

BCHG Test a Bit and Change

This instruction complements a bit. The bit is in the contents of the second operand and the location within the second operand is specified by the first operand. The second operand can be a data register or a byte in memory. If it is a data register, then any one of the 32 bits in the register can be complemented. The exact bit is specified by the first operand and the bits in the data register are numbered from 1 to 32 and from right to left. If a byte of memory is used, then the bits are numbered from 1 to 8. If the first operand is a data register and the second operand is a byte, then the contents of the data register are modulo 8 for the duration of the instruction.

BCLR Test a Bit and Clear

This instruction clears a bit. The bit is in the contents of the second operand and the location within the second operand is specified by the first operand. The second operand can be a data register or a byte in memory. If it is a data register, then any one of the 32 bits in the register can be cleared. The exact bit is specified by the first operand and the bits in the data register are numbered from 1 to 32 and from right to left. If a byte of memory is used, then the bits are numbered from 1 to 8. If the first operand is a data register and the second operand is a byte, then the contents of the data register are modulo 8 for the duration of the instruction.

BRA Branch Always

This instruction causes the program execution to automatically branch to the user specified label.

BSET Test a Bit and Set

This instruction sets a bit. The bit is in the contents of the second operand and the location within the second operand is specified by the first operand. The second operand can be a data register or a byte in memory. If it is a data register, then any one of the 32 bits in the register can be set. The exact bit is specified by the first operand and the bits in the data register are numbered from 1 to 32 and from right to left. If a byte of memory is used, then the bits are numbered from 1 to 8. If the first operand is a data register and the second operand is a byte, then the contents of the data register are modulo 8 for the duration of the instruction.

BSR Branch to Subroutine

This instruction pushes the contents of the Program Counter (PC) onto the data stack -(A7) and then branches to the user specified label.

BTST Test a Bit

This instruction tests a bit. The bit is in the contents of the second operand and the location within the second operand is specified by the first operand. The second operand can be a data register or a byte in memory. If it is a data register, then any one of the 32 bits in the register can be tested. The exact bit is specified by the first operand and the bits in the data register are numbered from 1 to 32 and from right to left. If a byte of memory is used, then the bits are numbered from 1 to 8. If the first operand is a data register and the second operand is a byte, then the contents of the data register are modulo 8 for the duration of the instruction.

CHK Check Register against Bounds

This instruction checks the lower half of the contents of the second operand and if it is greater than the upper bound (found in the first operand) or less than 0, then the exception processing is initiated and a TRAP is generated. The CHK instruction vector (vector #6) is used for the trap.

CLR Clear an Operand

This instruction clears the contents of the operand.

CMP Compare

This instruction subtracts the contents of the first operand from the contents of the second operand but does not change the contents of either operand. Just condition codes are changed.

CMPA Compare Address

This instruction subtracts the contents of the first operand from the contents of the second operand but does not change the contents of either operand. Just condition codes are changed.

CMPI Compare Immediate

This instruction subtracts the contents of the first operand from the contents of the second operand but does not change the contents of either operand. Just condition codes are changed. The maximum size of the immediate data is determined by the operand size.

CMPM Compare Memory

This instruction subtracts the contents of the first operand from the contents of the second operand but does not change the contents of either operand. Just condition codes are changed.

DBcc Test Condition, Decrement and Branch

This instruction first checks to see if the condition is false. The condition is specified by the cc which is one of the condition codes in Table 3. All 16 condition codes are usable. If the condition is false, then the contents of the data register is decremented by 1. After the decrementation, if the contents of the data register is -1 then the program execution branches to the user specified label.

DIVS Signed Divide

This instruction sign divides the contents of the second operand by the contents of the first operand and places the results into the second operand. The first operand is 16 bits and the second operand is 32 bits. The result is 32 bits with the quotient in the lower word and the remainder in the upper word of the register. If the first operand is a 0, then a TRAP is generated. The Zero Divide vector (vector #5) is used for the TRAP. If there is an overflow, then the operands are unaffected.

DIVU Unsigned Divide

This instruction unsign divides the contents of the second operand by the contents of the first operand and places the results into the second operand. The first operand is 16 bits and the second operand is 32 bits. The result is 32 bits with the quotient in the lower word and the remainder in the upper word of the register. If the first operand is a 0, then a TRAP is generated. The Zero Divide vector (vector #5) is used for the TRAP. If there is an overflow, then the operands are unaffected.

EOR Exclusive OR Logical

This instruction performs a bitwise logical exclusive OR on the contents of the two operands and places the result into the second operand.

EORI Exclusive OR Immediate

This instruction performs a bitwise logical exclusive OR on the contents of the two operands and places the result into the second operand. The immediate data can be up to 32 bits long, depending on the operand size. With byte or word operand size, the second operand can be the status register. If byte, then only the condition codes are affected. If word, then it is a privileged operation and the whole status register is affected.

EXG Exchange registers

This instruction exchanges the contents of two registers. They can be both address registers, both data registers, or an address register and a data register.

EXT Sign Extend

This instruction extends a byte sized number into a word sized number or a word sized number into a long word sized number. If the operand size is word, then bit 7 is copied into bits 8 to 15 and if the operand size is long word, then bit 15 is copied into bits 16 to 31.

JMP Jump

This instruction causes the program execution to automatically branch to the address specified by the contents of the operand.
JSR	Jump to Subroutine
This instruction pushes the contents of the Program Counter (PC) onto the data stack -(A7) and then branches to the address specified by the contents of the operand.
LEA	Load Effective Address
This instruction places the contents of the first operand into the address register.
LINK	Link and Allocate
This instruction pushes the contents of the address register onto the stack. Then the stack pointer is put into the address register and finally the displacement is added to the stack pointer. This is used with UNLK to handle nested subroutine calls.
LSL	Logical Shift Left
If there are two operands, then this instruction logically shifts to the left the contents of the second operand by the amount specified in the first operand. If the first operand is a data register, then the distance shifted is in the right most six bits of the first operand. If the first operand is immediate data, then the immediate data can be the integers 1 to 8. If there is only one operand, then the contents of the operand will be logically shifted to the left only one bit and the operand size is limited to word. Zeros are shifted into the low order bit and the last value shifted out of the high order bit is placed into the C and X bits. See Figure 3. This instruction is identical to ASL.
LSR	Logical Shift Right
If there are two operands, then this instruction logically shifts to the right the contents of the second operand by the amount specified in the first operand. If the first operand is a data register, then the distance shifted is in the right most six bits of the first operand. If the first operand is immediate data, then the immediate data can be the integers 1 to 8. If there is only one operand, then the contents of the operand will be logically shifted to the right only one bit and the operand size is limited to word. Zeros are shifted into the low order bit and the last value shifted out of the high order bit is placed into the C and X bits. See Figure 3.
MOVEMove Data from Source to Destination
This instruction moves the contents of the first operand into the location specified by the second operand.
MOVE to CCR	Move to Condition Codes
This instruction moves the contents of the first operand into the low order byte of the status register. The high order byte of the contents of the first operand is ignored. This is used to set the condition codes. "To CCR" is not part of the opcode.
MOVE to SR	Move to the Status Register
This instruction moves the contents of the first operand into the status register. This is used to set the condition codes and other bits in the status register. This is a privileged instruction. "To SR" is not part of the opcode.
MOVE from SR	Move from the Status Register
This instruction moves the status register into the location specified by the second operand. "From SR" is not part of the opcode.
MOVE USP	Move User Stack Pointer
This instruction moves the user stack pointer into the location specified by the second operand or moves the contents of the first operand into the user stack pointer. This is a privileged instruction. "USP" is not part of the opcode.
MOVEA	Move Address
This instruction moves the contents of the first operand into the address register.
MOVEM	Move Multiple Registers
This instruction moves the contents of more than one register into memory or vice versa. If the operand size is word, then the low order word is moved out of the registers or sign extended words are moved into the registers. With one exception, the order of moving data in or out of memory is: D0 to D7, A0 to A7. The one exception is when predecrement mode is used for the effective address; then the order is A7 to A0, D0 to D7. In predecrement mode, movement can only be from register to memory and in postincrement mode, movement can only be from memory to register.
MOVEP	Move Peripheral Data
This instruction moves bytes in a register to alternating bytes in memory. The transfers start with the high order byte of the register and end with the low order byte. The transferred bytes go onto even addressed memory bytes. If the effective address is even and the operand size is long word, then the resulting memory, starting at the effective address is 31-24 register byte, empty byte, 23-16 register byte, empty byte, 15-8 register byte, empty byte, 7-0 register byte, empty byte. The exact opposite can be done.
MOVEQ	Move Quick
This instruction moves an 8 bit number into a data register.
MULS Signed Multiply

This instruction multiplies the contents of two word sized signed operands and leaves a long word sized signed result in the second operand. The high order word of the second operand is ignored in multiplying and is written over by the result.
MULUUnsigned Multiply
This instruction multiplies the contents of two word sized unsigned operands and leaves a long word sized unsigned result in the second operand. The high order word of the second operand is ignored in multiplying and is written over by the result.
NBCD Negate Decimal with Extend
This instruction negates a binary coded decimal number and uses the X bit to do it. The operation is 0 minus the contents of the operand minus the X bit.
NEG Negate
This instruction negates a two's complement number and does not use the X bit to do it. The operation is 0 minus the contents of the operand.
NEGX Negate with Extend
This instruction negates a two's complement number and uses the X bit to do it. The operation is 0 minus the contents of the operand minus the X bit.
NOP No Operation
This instruction does nothing except increment the program counter by two and take time.
NOT Logical Complement
This instruction performs a bitwise logical complement on the contents of the operand.
OR Inclusive OR Logical
This instruction performs a bitwise logical OR on the contents of the two operands and places the result into the second operand.
ORI Inclusive OR Immediate
This instruction performs a bitwise logical OR on the contents of the two operands and places the result into the second operand. The immediate data can be up to 32 bits long, depending on the operand size. With byte or word operand size, the second operand can be the status register. If byte, then only the condition codes are affected. If word, then it is a privileged operation and the whole status register is affected.
PEA Push Effective Address
This instruction pushes the effective address onto the stack and postdecrements the stack pointer.
RESET Reset External Devices
This instruction resets the external devices. It is a privileged instruction.
ROL Rotate without Extend Left
If there are two operands, then this instruction rotates to the left the contents of the second operand by the amount specified in the first operand. If the first operand is a data register, then the distance shifted is in the right most six bits of the first operand. If the first operand is immediate data, then the immediate data can be the integers 1 to 8. If there is only one operand, then the contents of the operand will be rotated to the left only one bit and the operand size is limited to word. With each rotate of a bit the high order bit is shifted out and into two places: the low order bit and the C bit. See Figure 3.
ROR Rotate without Extend Right
If there are two operands, then this instruction rotates to the right the contents of the second operand by the amount specified in the first operand. If the first operand is a data register, then the distance shifted is in the right most six bits of the first operand. If the first operand is immediate data, then the immediate data can be the integers 1 to 8. If there is only one operand, then the contents of the operand will be rotated to the right only one bit and the operand size is limited to word. With each rotate of a bit the high order bit is shifted out and into two places: the low order bit and the C bit. See Figure 3.
ROXL Rotate with Extend Left
If there are two operands, then this instruction rotates to the left the contents of the second operand by the amount specified in the first operand. If the first operand is a data register, then the distance shifted is in the right most six bits of the first operand. If the first operand is immediate data, then the immediate data can be the integers 1 to 8. If there is only one operand, then the contents of the operand will be rotated to the left only one bit and the operand size is limited to word. With each rotate of a bit the high order bit is shifted out and into three places: the low order bit, the X bit, and the C bit. See Figure 3.
ROXR Rotate with Extend Right
If there are two operands, then this instruction rotates to the right the contents of the second operand by the amount specified in the first operand. If the first operand is a data register, then the distance shifted is in the right most six bits of the first operand. If the first operand is immediate data, then the immediate data can be the integers 1 to 8. If there is only one operand, then the contents of the operand will be rotated to the right only one bit and the operand size is limited to

word. With each rotate of a bit the high order bit is shifted out and into three places: the low order bit, the X bit, and the C bit. See Figure 3.

RTE   Return from Exception

This instruction is performed at the end of exception processing. It replaces the status register and the program counter with the original status register and program counter that are on the supervisor stack. They were put there by TRAP. This is a privileged instruction.

RTR   Return and Restore Condition Codes

This instruction is performed at the end of a subroutine started by BSR or JMP. It replaces the condition codes and the program counter with the original condition codes and program counter that are on the stack. BRA and JMP do no put the condition codes onto the stack. If you want to return with RTR, then immediately after the jump you must push the condition codes onto the stack, i.e., MOVE SR, -(SP).

RTS   Return from Subroutine

This instruction is the normal instruction to use at the end of a subroutine started by BSR or JMP. It replaces the program counter with the original program counter that is on the stack.

SBCD Subtract Decimal with Extend

This instruction subtracts the contents of the first operand and the contents of the X bit from the contents of the second operand with binary coded decimal arithmetic and places the result into the second operand.

Scc   Set According to Condition

This instruction causes the specified byte to be set if the condition is met. The condition is specified by the cc which is one of the codes in Table 3.

STOP Stop Program Execution

This instruction places the immediate data into the status register and stops the microprocessor from executing any more instructions. The immediate data is 16 bits. There are three ways to stop the STOP and restart execution. A trace exception will happen immediately if the trace bit is on. If an interrupt request occurs and it is of higher priority that that of the current processor priority, then an interrupt exception occurs. A reset request will always execute. This is a privileged instruction.

SUB   Subtract Binary

This instruction subtracts the contents of the first operand from the contents of the second operand with two's complement binary arithmetic and places the result into the second operand.

SUBA Subtract Address

This instruction subtracts the contents of the first operand from the contents of the second operand with two's complement binary arithmetic and places the result into the second operand.

SUBI   Subtract Immediate

This instruction subtracts the contents of the first operand from the contents of the second operand with two's complement binary arithmetic and places the result into the second operand. The immediate data can be up to 32 bits long, depending on the operand size.

SUBQ Subtract Quick

This instruction subtracts the contents of the first operand from the contents of the second operand with two's complement binary arithmetic and places the result into the second operand. The immediate data can be the integers 1 through 8.

SUBX Subtract with Extend

This instruction subtracts the contents of the first operand and the contents of the X bit from the contents of the second operand with two's complement binary arithmetic and places the result into the second operand.

SWAP Swap Register Halves

This instruction exchanges the contents of the high word and the contents of the low word in a data register.

TAS   Test and Set an Operand

This instruction sets the high order bit of the contents of the operand to 1. The tests for condition codes are done before the high order bit is set. This instruction can be interrupted during is operation. This operation is useful in synchronizing independent programs running simultaneously.

TRAP Trap

This instruction initiates exception processing. It pushes the contents of the program counter and then the contents of the status register onto the supervisor stack pointer. The address at the TRAP instruction vector is then put in the program counter.

TRAPV   Trap on Overflow

This instruction executes a TRAP if the V bit is on. The Trap instruction vector used is 7.

TST   Test an Operand

This instruction only sets condition codes. The operand is not affected.

UNLK Unlink
This instruction copies the contents of the address register into the stack pointer and then pops the top of the stack into the address register. This is used with LINK to handle nested subroutine calls.

Appendix—Error Messages

About this Appendix

The Mops Assembler provides its own error handler for assembler code errors and can supply error messages for them.

Error in loading AsmCodes	200
There was an I/O error generated by the Macintosh file Manager. The file AsmCode loads during compilation. Check this file. Normally, the user should never change this file.

Bad operand size	202
The operand size at the end of the opcode should be ".L", ".W", or ".B".

Bad operand	203
A faulty operand was used. Some operand modes are illegal with some opcodes.

Bad immediate operand	205
A faulty immediate operand was used. It is most likely a wrong character.

Error in loading Operands	206
There was an I/O error generated by the Macintosh file Manager. The file Operands loads during compilation. Check this file. Normally, the user should never change this file.

Operands do not match	207
For opcodes ABCD and SBCD, only two types of operands are allowed (Dx and -(Ax)). For ABCD, SBCD, ADDX, SUBX< and CMPM both operands must be of the same mode.

Operand not an address register	208
An operand not an address register in the MOVE USP instruction. USP must be one operand and an address operand must the the other operand.

Bad register mask	210
The register list for MOVEM is faulty

Error in first pass	211
The assembler makes two passes over the code. An error was found in the first pass so assembly was aborted before the second pass was started.

Cannot find object or word	216
The object or word looked for by MOPS[objname] could not be found in the dictionary.

Register direct operand needed	219
At least one of the two operands must be a register direct.

Mode mismatch	245
An operand was not of the correct mode for the instruction.

Short absolute address out of range	246
Possibly you should be using a long absolute address.

Byte displacement out of range	247
The displacement field in an indexed mode instruction is only one byte long. You are trying to address a location too far from the base address.

Word displacement out of range	248
You are trying to address a location further than 32K bytes from the base address. You might have to use an extra instruction or instructions to compute the address.

Immediate operand out of range	249
Fairly self-explanatory. Possibly you should be using long immediate mode.

Branch out of range	250
Should be self-explanatory. If you are using a short branch, substitute a long one.

Undefined label	251
You are referring to a label which hasn't been defined anywhere in a label field.

Bad opcode	252
You have specified an opcode which doesn't exist. Possibly a typo. Or it may be a 68020/30/40-specific instruction, which the assembler can't handle yet.