Table of Contents

Chapter	Name	Page
1	Using the Assembler	1
2	Registers and Addressing Modes	7
3	Instructions	15
Appendix	Error Messages	33

Chapter 1
Using the Assembler

About this Chapter
With the Yerk Assembler module you can write colon definitions and method definitions in assembly code and you can also reference Yerk data and executable words. This chapter explains how the assembler interfaces with Yerk. The Macintosh uses a Motorola 68000 microprocessor and the assembler syntax is based on standard 68000 assembly language. This chapter is not a tutorial and assumes a basic knowledge of 68000 assembly programming. Among the good books on 68000 assembly are *MC68000 16-bit Microprocessor User's Manual* published by Motorola and *68000 Assembly Language Programming* by Gerry Kane, Doug Hawkins & Lance Leventhal and published by Osborne/McGraw-Hill.

Getting Started
Included on your diskette is the assembler module (ASMMOD.BIN), two text files - AsmCodes and Operands, assembler source files, floating point source files, and some demonstration files.

To use the assembler put ASMMOD.BIN, AsmCodes, and Operands with your other modules. The text files AsmCodes and Operands are used by the assembler during compilation and need to be present while assembling. The assembler and its two text files are not needed at run time. The Yerk Assembler may not be distributed with sealed applications. On the first use of the assembler after a dictionary comes up, there will be a longer than usual wait because AsmCodes and Operands will be loading.

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

Due to size limitations you cannot use the Yerk Assembler on a 128K Macintosh.

Recompiling the Assembler
The assembler comes compiled but if you wish to recompile it then move the assembler source files, Lbug, String, 2dArray, and Dictionary onto the working disk and bring up yerk.com or yerkFP.com and enter:

```
module asmmod
```

Recompiling Floating Point

With the assembler you can recompile the floating point of Yerk. You need the then floating point source files and FrontEnd to do it. To recompile floating point bring up yerk.com and enter:

```
// fp.ld
```

then do a "Save dictionary As…" with a new dictionary name.

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. This example and the other examples give the code from the demo file and the screen commands after the file is loaded.

Example: (from file Demo1)

```
: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,-(SP)   \ pushes contents of register D0 onto data stack
;code

0->demo1
1->. 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:". See Chapter 2 for more information on addressing modes and Yerk's usage of registers.

Example: (from file Demo2)

```
class demo2 <super object
  var int1

  :M put: put: int1 ;M
  :M get: get: int1 ;M

  :MCODE asmget:
     move.l   0(A3,A2.l),-(A7)
  ;MCODE

  :MCODE asmput:
     move.l   (A7)+,0(A3,A2.l)
  ;MCODE
;class
```

```
demo2 test2

0->26 put: test2
0->asmget: test2 . cr
26
0->34 asmput: test2
0->get: test2 . cr
34
```

Calling Yerk objects

There is a special operand in the Yerk Assembler that finds the relative address of an object. Its syntax is "YERK[objname]". The following example shows how a Yerk defined object can be accessed by the assembler. The relative address of the object (fun) is placed in D7. It is a convention, but not required, to use D7 for relative addresses of objects. The address is relative to the base of Yerk which is in A3. In the example the absolute address of the var "fun" is D7 plus A3. Addressing in the assembler is absolute while in Yerk addressing is relative to A3. This example also shows how an assembler defined word (demo3) can be accessed in Yerk just like a colon definition.

Example: (from file Demo3)

```
var fun

:code demo3         \ put a 10 in the Yerk var fun
      move.l   YERK[fun],D7
      move.l   #10,0(A3,D7.l)
;code

: test3
  0 put: fun
  demo3
  get: fun . cr
;

0->test3
10
```

The following example shows an array being accessed with YERK[objname]. Five long words of data starting at the absolute address of the array "edmund" plus 4 is moved to five registers. The contents of the five registers are then pushed onto the stack. The displacement of 4 is necessary because the first 4 bytes of an array are for record keeping.

Example: (from file Demo4)

```
5 array edmund

:code demo4
      move.l   YERK[edmund],D7
      movem  4(A3,D7.l),d0-d1/a0-a2
      move.l   d0,-(sp)
```

```
        move.l   d1,-(sp)
        move.l   a0,-(sp)
        move.l   a1,-(sp)
        move.l   a2,-(sp)
;code

: test4
  7 fill: edmund
  demo4
```

```
  . . . . . cr
;

0->test4
7 7 7 7 7
```

Executing Yerk defined Words

Words defined with ":code" become Yerk primitives, i.e. words that contains executable code, rather than addresses of other words. Words in the nucleus are also Yerk primitives. To determine whether a word is a primitive do the following:

'code *word* colcode =

A true is returned for a secondary (colon definition) and a false otherwise.

The next example shows the calling of a primitive. This is done by executing a "JMP" to the absolute address of the word. After the called word (dup) executes, it will return to the word (test5) that called the ":code" word.

Example of calling primitive: (from file Demo5)

```
:code demo5
  move     #55,-(A7)
  move     Yerk[dup],D7
  jmp      0(A3,D7.L)
;code

: test5
  demo5
  . . cr
;

0->test5
55 55
```

The next example shows the calling of a secondary. After the called word (not) executes, it will return to the word (test6) that called the ":code".

Example of calling a secondary word: (from file Demo6)

```
:code demo6        \ places a number on the stack and "not"'s it
  move     #66,-(A7)
```

```
  move     Yerk[not],D7
  move     0(A3,D7.L),D6
  jmp      0(A3,D6.L)
;code

: test6
  demo6
  . cr
;
```

```
0->test6
0
```

To write assembler subroutines that can be called by other assembler routines, terminate them with an "RTS" instruction and call them with a "JSR". Note that assembler subroutines terminated by a "RTS" will not work if called by a regular colon definition. Floating point source file fltMem has many examples of this.

Example of calling an assembler defined word: (from file Demo7)

```
:code demi
  move.l     #2,D0
  rts
;code

:code demo7
  move.l     #3,-(SP)
  move.l     YERK[demi],D7
  jsr        0(A3,D7.l)
  move.l     D0,-(SP)
  move.l     #1,-(SP)
;code

: test7
  demo7
  . . . cr
;

0->test7
1 2 3
```

Toolbox calls

The Yerk 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 Yerk 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.

Example: (from file Demo8)

```
:code demo8         \ tests two "addr len" strings for being equal
  move.l      (SP)+,D0          \ pop len of first string
  swap D0     \     onto high order word of D0
  movea.l     (SP)+,A0          \ pop addr of first string
  adda.l      A3,A0 \ make address absolute
  or.l        (SP)+,D0          \ pop len of second string onto low order word of D0
```

```
  movea.l   (SP)+,A1        \ pop addr of second string
  adda.l    A3,A1 \ make address absolute
  call      cmpstring       \ IF equal THEN returns 0 ELSE 1
  move.l    D0,-(SP)        \ push answer onto stack
;code

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

0->test8
1
```

Syntax

The Yerk Assembler uses a standard syntax. In Yerk parentheses denote comments, but not in the assembler. The assembler supports "\" and ";" as the start of a comment.

A label may not have the same spelling as an opcode and an opcode can start in the first column if there is no label. Labels do not have to start in column one. After the opcode you may define the operand size as byte, word, or long word.

Example:
```
MOVE.B
MOVE.W
MOVE.L
```

The Yerk Assembler's default is long word because Yerk's stacks have long word sized elements. Most 68000 assemblers use word as the default.

The Yerk Assembler is case insensitive.

Chapter 2
Registers and Addressing Modes

About this Chapter
This chapter describes the registers which you can work with and the addressing modes that can be used on operands.

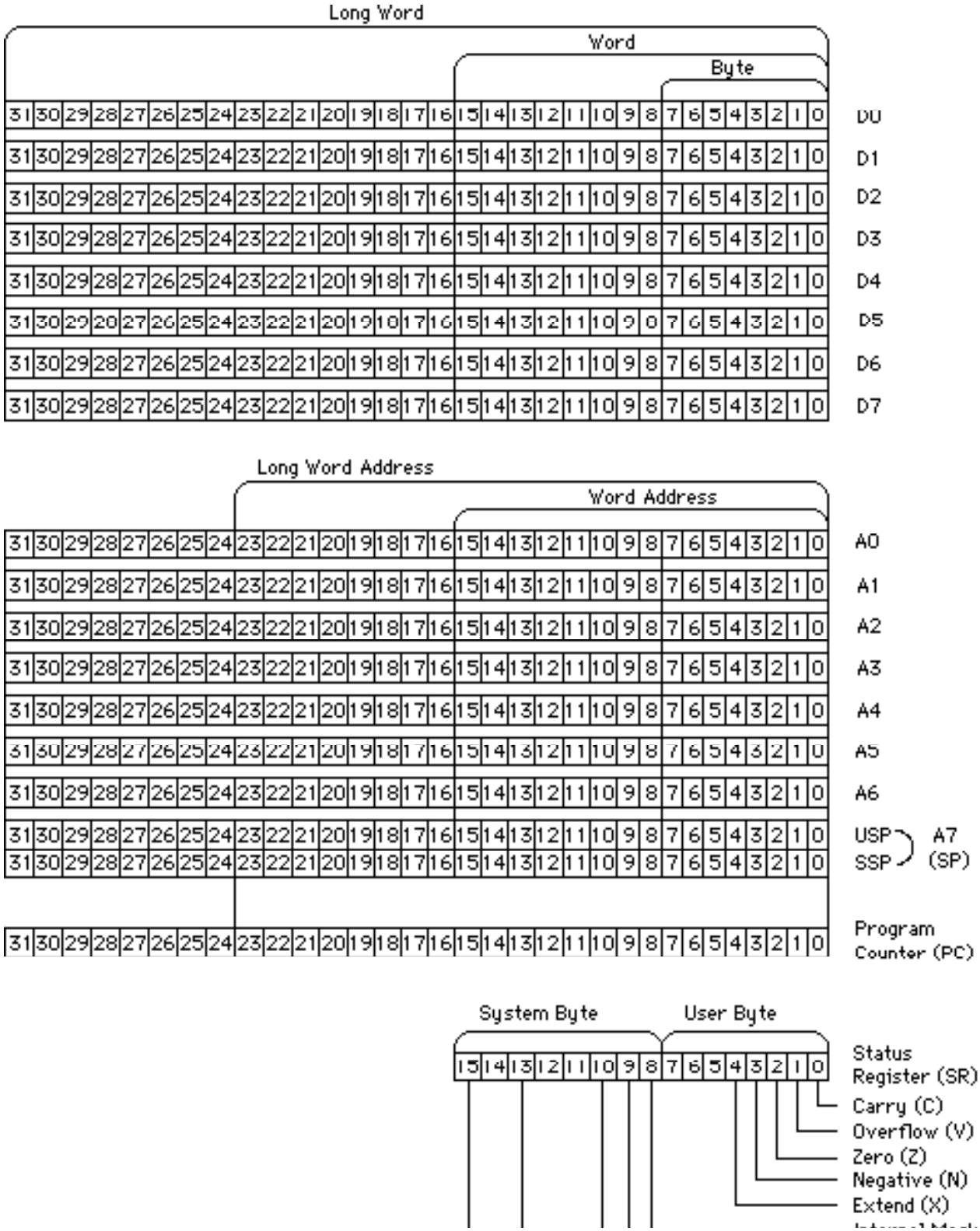
Registers
There are 18 registers that you should concern yourself with: the 8 data registers, the 8 address registers, the program counter and the status register. They are each 32 bits in size with the exception of the status register, which is 16 bits. They are numbered with 0 as the right most bit. See Figure 1 for a chart of all 18 registers. 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 Yerk and the Macintosh for themselves. See Figure 2 for a memory map of the Macintosh while Yerk is running.


Long Word
Word
Byte
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
D0
D1
D2
D3
D4
D5
D6
D7
Long Word Address
Word Address
A0
A1
A2
A3
A4
A5
A6
USP
SSP
A7
(SP)
Program
Counter (PC)
System Byte
User Byte
Status
Register (SR)
Carry (C)
Overflow (V)
Zero (Z)
Negative (N)
Extend (X)

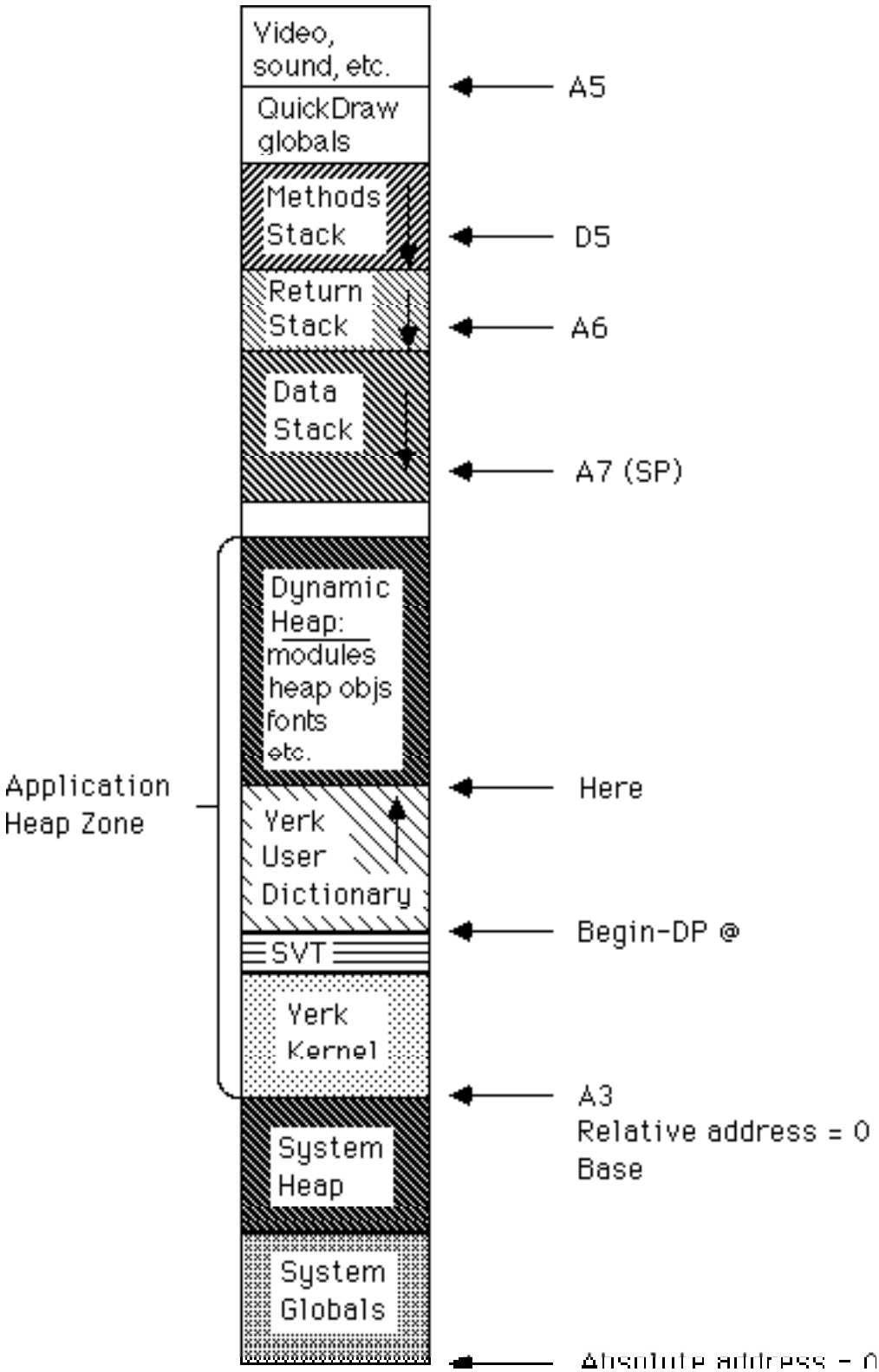

Video,
sound, etc.
QuickDraw
globals
Methods
Stack
Return
Stack
Data
Stack
Dynamic
Heap:
modules
heap objs
fonts
etc.
Yerk
User
Dictionary
SVT
Yerk
Kernel
System
Heap
System
Globals
A5
D5
A6
A7 (SP)
Here
Begin-DP @
A3
Relative address = 0
Base
Application
Heap Zone

Figure 2 - Memory Map

You may safely manipulate in a ":code" or a ":mcode" definition most registers (A0, A1, A2, D0, D1, D2, D3, D4, D6, D7) without danger. However, the data in these registers may change if your program exits the definition. A2 is set to the address of the fist ivar of the given object at the beginning of a ":mcode" definition. This is just for convenience; you can use A2 for other purposes if you wish. The assignments of each register are:

A0	Free
A1	Free
A2	Free (address of first ivar in an object at the start of a ":mcode")
A3	Kernel base pointer
A4	Instruction pointer
A5	Pointer to base of QuickDraw Globals
A6	Return stack pointer
A7	Data stack pointer (in supervisor mode)
D0	Free
D1	Free
D2	Free
D3	Free
D4	Free
D5	Methods stack pointer
D6	Free
D7	Free

Condition Codes

There are five condition codes used by the Yerk 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 Yerk is in supervisor mode and if the supervisor bit is off, then Yerk is in user mode. The default is for the bit to be on. 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 dump of registers.

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
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.

```
Example:    MOVE    D0,D1
```

Address Register Direct

```
Example:    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.

```
Examples:   MOVE    D0,SR
            MOVEM 4(A3,D7.L(,D6-D7/A0-A2
```

Address Register Indirect
The effective address is the content of the address register.

```
Example:    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.

```
Example:    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.

Example: `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 "$".

```
Examples:   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.

```
Example:    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.

```
Example:    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.

```
Example:    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 "$".

```
Examples:   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.

Example: RTS

Chapter 3
Instructions

About this Chapter

This chapter explains the machine instructions for the Yerk 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 Yerk 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.

ABCD	add decimal with extend D				B ? M -(Ay),-(Ax)		Dy,Dx	A	?			
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) #<data>,Dy (r<>0) <ea>		-	B	C	0	0	
							A	B	C	J	P	
ASR	arithmetic shift right	B	W	L	Dx,Dy (r=0) #<data>,Dy (r<>0) <ea>		-	B	C	0	0	
							A	B	C	0	R	
Bcc	branch conditionally	B	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	W		<label>		-	-	-	-	-	
BSET	test a bit and set	B		L	Dx,<ea> #<data>,<ea>		-	-	E	-	-	
BSR	branch to subroutine	B	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 #<data>,Dy <ea>	(r=0) (r<>0)	- A	B B	C C	0 0	0 P
LSR	logical shift right	B	W	L	Dx,Dy #<data>,Dy <ea>	(r=0) (r<>0)	- A	B B	C C	0 0	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
MOVE to 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 USP	move user stack pointer			L	USP,Ax 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
ORI	inclusive OR immediate	B	W	L	#<data>,<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) #<data>,Dy (r<>0) <ea>	- -	B B	C C	0 0	0 P
ROR	rotate without extend right	B	W	L	Dx,Dy (r=0) #<data>,Dy (r<>0) <ea>	- -	B B	C C	0 0	0 R
ROXL	rotate with extend left	B	W	L	Dx,Dy (r=0) #<data>,Dy (r<>0) <ea>	- A	B B	C C	0 0	Q P

ROXR	rotate with extend right	B	W	L	Dx,Dy #<data>,Dy <ea>	(r=0) (r<>0)	-	B	C	0	Q 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
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

Table 3

Shifts and Rotates

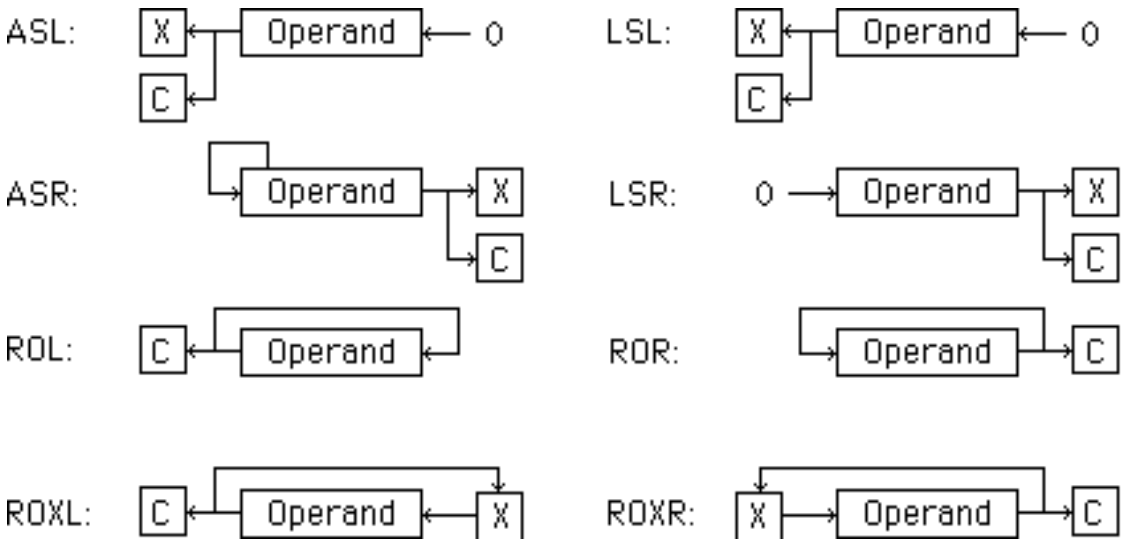

ASL:
X
Operand
0
C
LSL:
X
Operand
0
C
ASR:
Operand
X
C
LSR:
0
Operand
X
C
ROL:
C
Operand
ROR:
Operand
C
ROXL:
C
Operand
X
ROXR:
X
Operand
C

Figure 3

Machine Instructions 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.

MOVE **Move 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.

MULU **Unsigned 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 Yerk 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 YERK[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.