as Assembler

Reference Manual

Table of Contents

1. Introduction

The COHERENT™ assemblers are multipass assemblers intended for use as targets by some of the COHERENT compilers and for writing small assembly language subroutines. Since they are not intended to be used for full scale assembly language programming, they lack many of the more elaborate facilities of full fledged assemblers; for example, there are no facilities for conditional compilation or user defined macros. However, they do attempt to optimize span dependent instructions (for example, branches) on machines where this is useful and reasonable.

In general, the COHERENT assemblers use the same syntax for address expressions and machine instructions as the assemblers supplied by the manufacturers of the machines. However, all COHERENT assemblers use the same set of assembly directives to make remembering them easier and to make a greater part of the COHERENT compilers machine independent.

2. Invocation

All assemblers are invoked from the shell as follows:

```
        as [-glx] [-o file] file [file...]
```

By convention, the command **as** always invokes the native assembler for the host machine. Cross assemblers use the same command line syntax, but have a command name constructed by concatenating the name of the target machine and the string **as**. For example, an assembler for the IBM 370 would be called **370as**.

The named files are concatenated and the resulting object code is written to the file specified by the **−o** option, or to file **l.out** if no **−o** option is given. This file is made executable if there are no undefined symbols.

The optional **−g** argument causes all symbols which are undefined at the end of the first pass to be given the type undefined external.

The optional **−l** option causes the assembler to generate a listing. The listing is written to the standard output, where it may be easily redirected into a file or through a pipe to **lpr**.

The optional **−x** argument causes all non global symbols that begin with the character 'L' to be stripped from the symbol table of the object file. This option is used by the C compiler to speed up the loading of C generated object files.

3. Lexical Conventions

Assembler tokens consist of identifiers (also known as symbols or names), constants, and operators.

3.1. Identifiers

An identifier consists of a sequence of alphanumeric characters (including the period '.' and the underscore '_') the first of which may not be numeric. Only the first sixteen characters of the name are significant; the remainder are quietly thrown away. Upper and lower case are considered different. The machine instructions, assembly directives and frequently used built-in symbols are in lower case. On some machines less often used built-in symbols may have upper case names to prevent clashes between them and the programmer's labels.

3.2. Constants

The C compiler and the COHERENT assemblers use the same syntax for constants. A sequence of digits not beginning with a '0' is a decimal constant. A sequence of digits beginning with a '0' is an octal constant. A sequence of digits beginning with a '0x' is a hexadecimal constant. The letters 'a' through 'f' (and 'A' through 'F') have decimal values 10 through 15 respectively.

A character constant consists of a single quote ('′') followed by any ASCII character (except the newline). The constant's value is the ASCII code for the character right justified in the machine word.

The following multicharacter escape sequences are acceptable in a character constant to represent newline and other special characters.

\b	Backspace	(0010)
\f	Form Feed	(0014)
\n	Newline	(0012)
\r	Carriage Return	(0015)
\t	Tab	(0011)
\nnn	Octal value	(0nnn)

3.3. Blanks and Tabs

Blanks and tabs may be used freely between tokens, but may not appear within identifiers. A blank or a tab is required to separate adjacent tokens not otherwise separated.

3.4. Comments

Comments are introduced by a slash '/' and continue until the next newline character. All characters in comments are ignored by the assembler.

4. Location Counters and Program Sections

The assemblers permit the division of programs into a number of program sections. Each section corresponds (roughly) to a functional area of the address space.

There are seven program sections, subdivided into three groups. The instruction group holds the instructions (code) of a program. The data group holds the data (both initialized and uninitialized) of a program. Finally, the symbol table group (consisting of a single program section) contains tables used by high level debuggers. These tables are not read into memory when the program is executed.

The code and data groups each contain three program sections. The shared sections (SHRI, SHRD) hold the portion of the program which may be shared among all users of the program. If possible this section will be loaded write protected. The private sections (PRVI, PRVD) hold the portion of the image that must not be shared, either because it is impure or because there is some complex reason why it must be writable. The third sections (BSSI, BSSD) are like the private sections in that they are impure, but differ in that they are uninitialized, and do not need to occupy any space in the output file. The operating system will allocate the required blocks of cleared (zeroed) memory for the BSS sections when the program is run.

Every program section contains a location counter. This counter is the high water mark of the information (code or data) assembled into the section.

At the beginning of the assembly all location counters are set to zero. At the end of pass two (when the sizes of all program sections are known) the location counters are adjusted so that all program sections occupy a single block of addresses beginning at location 0. Enough information is retained so that this adjustment can be unraveled by the linkage editor; however, this adjustment makes the output of the assembler immediately executable.

The private data actually contains two location counters. The second location counter (STRN) is a special one used by the COHERENT compilers to allocate memory for strings. It appears in the assembler output as an extension of the private data section.

4.1. The Current Location

The special symbol '.' represents the current location counter. Its value is the offset into the current program section of the start of the statement in which it appears. It may be assigned to, with the restrictions that the assignment must not either change the program section or cause the value to decrease.

5. Expressions

An expression is a sequence of symbols representing a value and a program section. Expressions are made up of identifiers, constants, operators and brackets. All binary operators have equal precedence and are executed in a strict left to right order (unless altered by brackets).

5.1. Types

Every expression has a type determined by its operands. The types are:

Undefined	Upon first encounter a symbol is undefined. A symbol may also become undefined if it is assigned the value of an undefined expression. It is an error to assemble an undefined expression in pass 2. Pass 1 allows assembly of undefined expressions, but phase errors may result if undefined expressions are used in certain contexts (i.e., in a .BLKW or .BLKB).
Absolute	An absolute symbol is one defined ultimately from a constant or from the difference of two relocatable values.
Register	Register symbols are used to refer to the registers of the target machine. They are special to allow the assembler to distinguish register addressing from normal memory addressing. The predefined register symbols (if any exist) are different for every machine. The appendix for the specific target machine lists the names and values of any predefined register symbols.
Relocatable	All other user symbols are relocatable symbols in some program section. Each program section is a different relocatable type.

Each keyword in the assembler has a secret type which identifies it internally. However, all of these secret types are converted to absolute in expressions. Thus any keyword may be used in an expression to obtain the basic value of the keyword. The basic value of a machine operation is (usually) the opcode with any operand specific bits set to zero. The basic value of an assembler directive is, in general, uninteresting.

5.2. Operators

The following characters are interpreted as operators in expressions.

'+'	Addition
'-'	Subtraction
'*'	Multiplication
'-'	Unary negation
'~'	Unary complement
'^'	Type transfer
'|'	Segment construction

Expressions may be grouped by means of square brackets ('[' and ']'); parentheses are reserved for use in address mode descriptions.

5.3. Type Propagation

When operands are combined in expressions the resulting type is a function of both the operator and the types of the operands. The '*', '~' and unary '-' operators can only manipulate absolute operands and always yield an absolute result.

The '+' operator allows the addition of two absolute operands (yielding an absolute result) and the addition of an absolute to a relocatable operand (yielding a result with the same type as the relocatable operand).

The binary '-' operator allows two operands of the same type, including relocatable, to be subtracted (yielding an absolute result) and an absolute to be subtracted from a relocatable (yielding a result with the same type as the relocatable operand).

The binary '^' operator yields a result with the value of its left operand and the type of its right operand. It may be used to create expressions (usually intended to be used in an assignment statement) with any desired type.

6. Statements

A program consists of a sequence of statement separated by newlines or by semicolons. There are three kinds of statements: null statements, assignment statements and keyword statements.

6.1. Labels

Any statement may be preceded by any number of labels.

A name label consists of an identifier followed by a colon (':'). The program section and value of the label are set to that of the current location counter. It is an error for the value of a label to change during an assembly. This most often happens when an undefined symbol is used to control a location counter adjustment.

A temporary label consists of a digit ('0' to '9') followed by a colon (':'). Such a label defines temporary symbols of the form 'xf' and 'xb', where 'x' is the digit of the label. References of the form 'xf' refer to the first temporary label 'x:' forward from the reference; those of the form 'xb' refer to the first temporary label 'x:' backward from the reference. Such labels conserve symbol table space in the assembler.

6.2. Null statements

A null statement is just an empty line (which may have labels and be followed by a comment). Null statements are ignored by the assembler. Common examples of null statements are an empty line and a line consisting of only a label.

6.3. Assignment statements

An assignment statement consists of an identifier followed by an equal sign '=' and an expression. The value and program section of the identifier are set to that of the expression. Any symbol defined by an assignment statement may be redefined, either by another assignment statement or by a label.

6.4. Keyword statements

Most assembler statements are keyword statements. All of the machine operations and assembler directives are of this type.

A keyword statement begins with one of the assembler's predefined keywords, followed by any operands required by that keyword.

6.4.1. Section change

These assembler directives change the current program section to the named section:

```
.shri
.prvi
.bssi
.shrd
.prvd
.bssd
.strn
.symt
```

The current location counter is set to the high water mark of code or data in the section.

6.4.2. .byte

The expressions in the comma separated list are truncated to the target machine's byte size (usually eight bits) and are assembled into successive bytes:

```
.byte expression [ , expression ] ...
```

6.4.3. .word

The expressions in the comma separated list are truncated to the target machine's word size (or the shorter of the two integer sizes if the assembler supports long integers) and the resulting data is assembled into successive words:

```
.word expression [ , expression ] ...
```

6.4.4. .long

The expressions in the comma separated list are evaluated and the resulting data is assembled into successive long integers:

```
        .long expression [ , expression ] ...
```

This assembler directive is only implemented on some machines.

6.4.5. .ascii string

The first non-whitespace character after the keyword is taken as a delimiter. Successive characters from the string are assembled into successive bytes until another instance of this delimiter is encountered. It is an error for a newline to be encountered before reaching the final delimiter. The multicharacter escape sequences described in section 3.2 may be used in the string to represent newlines and other special characters.

6.4.6. Alignment Directives

Some target machines have alignment requirements. Such machines will provide a number of alignment directives. The details of the alignment directives for a specific machine may be found in the appendix covering that machine.

6.4.7. .blkb .blkw .blkl

These statements assemble blocks of bytes, words or longs that are filled with zeros:

```
        .blkb expression
        .blkw expression
        .blkl expression
```

The size of the block is expression bytes, words or longs. The .blkl directive is only implemented on some machines.

6.4.8. .globl

The identifiers in the comma separated list are marked as global:

```
        .globl identifier [ , identifier ] ...
```

If they are defined in the current assembly they may be referenced by other object modules; if they are undefined they must be resolved by the linker before execution.

6.4.9. .title string

The argument string appears on the top of every page in the assembly listing. This directive also causes the listing to skip to a new page.

6.4.10. .page

This assembly directive causes the assembly listing to skip to the top of a new page; a form feed character is inserted into the file and the page position is reset so that a new title line is generated.

6.5. Machine Instructions

The syntax of machine instructions and their associated addressing expressions is different on every machine. The details for a specific machine can be found in the appropriate appendix.

In general, the syntax of machine instructions and address expressions is the same as that used by the machine manufacturer. Sometimes this is changed if the syntax is extraordinarily baroque or if the characters chosen by the manufacturer for some part of the address syntax are difficult to enter given the default COHERENT typing conventions.

7. Diagnostics

All errors are reported on the standard error stream as a single character error code tagged with a line number. If there is a symbol associated with the error code (for example, if a symbol is undefined) then the symbol's name is also reported.

If more than one input file appears on the command line error messages are tagged with the name of the source file.

Errors are reported on the listing file in the traditional fashion, as single character error flags at the extreme left side of the listing. If a listing is generated the total number of errors is displayed on the standard error stream at the end of assembly. This is useful since in most cases the listing will be disappearing into a file or down a pipe.

a	Addressing error. Generated by just about any kind of operand/instruction mismatch or address field semantic error.
b	Byte alignment. The offending instruction or assembly directive cannot be assembled at the current location because of some sort of alignment problem. For example, on the PDP-11 (whose instructions must be on even addresses) this error is given if you attempt to assemble any type of instruction on an odd addresses ('.' is odd).
e	Expression syntax. Some kind of syntax error has occured in an expression.
m	Multiple definition. The offending line is involved in the multiple definition of a label. The label name is displayed.
n	Number syntax. Some kind of syntax error has been discovered while reading in a numeric constant.
p	Phase error. The value of a label has changed during the assembly. The name of the troublesome label is displayed.
q	Questionable syntax. The assembler has no idea how to parse the offending line, so it has just given up.
r	Relocation error. An attempt has been made to create or use an expression in a way that the linker cannot resolve.

u	Undefined symbol. A symbol is used but never defined. The undefined symbol's name is displayed.

Appendix A: The Digital Equipment Corporation PDP-11 Assembler

The assembler for the Digital Equipment Corporation's PDP-11 family of computers is quite similar to the PAL-11 and MACRO-11 assemblers. Some details of address syntax have been changed, since the default typing conventions of COHERENT make typing PAL-11 or MACRO-11 address expressions difficult.

A.1 Predefined Symbols

The following symbols are predefined. The type of the symbol is set to 'register'.

r0	000000	r1	000001
r2	000002	r3	000003
r4	000004	r5	000005
sp	000006	pc	000007

A.2 Address Descriptors

The following syntax is used for general source and destination address descriptors.

In the examples the symbol 'r' refers to a register and the symbol 'e' to an expression.

r	Register
(r)	Register deferred
(r)+	Autoincrement
*(r)+	Autoincrement deferred
-(r)	Autodecrement
*-(r)	Autodecrement deferred
e(r)	Index
*e(r)	Index deferred
e	Direct (PC relative)
$e	Immediate (PC autoincrement)

A.3 Assembly Directives

The **.long** and **.blkl** assembler directives are not supported by the PDP-11 assembler.

The **.even** and **.odd** directives are provided to force the current location counter to an even or odd location respectively (by emitting a 0 byte if necessary).

A.4 Machine Instructions

The following machine instructions are defined. The examples illustrate the general syntax of the operands. Combinations that are syntactically valid may be forbidden for semantic reasons.

In the examples 's' and 'd' refer to general source and destination addresses, 'e' refers to an expression and 'r' refers to a register.

There is no facility for generating code for the commercial instruction set (CIS) feature of the new PDP-11 processors.

absd	d	170600
absf	d	170600
adc	d	005500
adcb	d	105500
add	s,d	060000
addd	s,r	172000
addf	s,r	172000
ash	s,r	072000
ashc	s,r	073000
asl	d	006300
aslb	d	106300
asr	d	006200
asrb	d	106200
bcc	e	103000
bcs	e	103400
beq	e	001400
bge	e	002000
bgt	e	003000
bhi	e	101000
bhis	e	103000
bic	s,d	040000
bicb	s,d	140000
bis	s,d	050000
bisb	s,d	150000
bit	s,d	030000
bitb	s,d	130000
ble	e	003400
blo	e	103400
blos	e	101400
blt	e	002400
bmi	e	100400
bne	e	001000
bpl	e	100000
bpt		000003
br	e	000400
bvc	e	102000
bvs	e	102400
cfcc		170000
clc		000241
cln		000250

clr	d	005000
clrb	d	105000
clv		000242
clz		000244
cmp	s,d	020000
cmpb	s,d	120000
cmpd	s,r	173400
cmpf	s,r	173400
com	d	005100
comb	d	105100
dec	d	005300
decb	d	105300
div	s,r	071000
divd	s,r	174400
divf	s,r	174400
emt	e	104000
halt		000000
inc	d	005200
incb	d	105200
iot		000004
jmp	d	000100
jsr	r,d	004000
ldcdf	s,r	177400
ldcfd	s,r	177400
ldcid	s,r	177000
ldcif	s,r	177000
ldcld	s,r	177000
ldclf	s,r	177000
ldd	s,r	172400
ldexp	s,r	176400
ldf	s,r	172400
ldfps	s	170100
mark	e	006400
mfpd	d	106500
mfpi	d	006500
mfps	d	106700
modd	s,r	171400
modf	s,r	171400
mov	s,d	010000
movb	s,d	110000

mtpd	d	106600
mtpi	d	006600
mtps	d	106400
mul	s,r	070000
muld	s,r	171000
mulf	s,r	171000
neg	d	005400
negb	d	105400
negd	d	170700
negf	d	170700
nop		000240
reset		000005
rol	d	006100
rolb	d	106100
ror	d	006000
rorb	d	106000
rti		000002
rts	r	000200
rtt		000006
sbc	d	005600
sbcb	d	105600
sec		000261
sen		000270
setd		170011
setf		170001
seti		170002
setl		170012
sev		000262
sez		000264
sob	r,e	077000
spl	e	000230
stcdf	r,d	176000
stcdi	r,d	175400
stcdl	r,d	175400
stcfd	r,d	176000
stcfi	r,d	175400
stcfl	r,d	175400
std	r,d	174000
stexp	r,d	175000
stf	r,d	174000

stfps	d	170200
stst	d	170300
sub	s,d	160000
subd	s,r	173000
subf	s,r	173000
swab	d	000300
sxt	d	006700
trap	e	104400
tst	d	005700
tstb	d	105700
tstd	d	170500
tstf	d	170500
wait		000001
xor	r,d	074000

Appendix B: The Intel iAPX-86 Assembler

The assembler for the Intel iAPX-86 bears little resemblance to the Intel supplied assembler (ASM-86). Just about everything is different: the assembler directives, the syntax of address expressions, and so on.

B.1 Predefined Symbols

The following symbols are predefined. The type of the symbol is set to agree with the symbol's use.

ax	0000	cx	0001
dx	0002	bx	0003
sp	0004	bp	0005
si	0006	di	0007
al	0000	cl	0001
dl	0002	bl	0003
ah	0004	ch	0005
dh	0006	bh	0007
es	0000	cs	0001
ss	0002	ds	0003

All of these symbols are names for the machine registers. Their types are set to internal values that permit the assembler to determine the actual bit pattern for a machine instruction and to check for incorrect register usage (for example, a byte register used as the destination of a word instruction).

B.2 Address Descriptors

The following syntax is used for general source and destination address descriptors.

In the examples the symbol 'r' refers to a register and the symbol 'e' to an expression.

`r`	Register
`e`	Direct address
`(r)`	Indexing, no displacement
`e(r)`	Indexing with displacement
`(r,r)`	Double indexing, no displacement
`e(r,r)`	Double indexing with displacement
`$e`	Immediate

A direct address is interpreted as either a direct address or a PC relative displacement, depending on the requirements of the instruction.

If an address descriptor indicates an indexing mode and the base expression is of type absolute, the assembler uses the shortest displacement length (zero, one or two bytes) that can hold the expression's value. Relocatable base expressions, whose values cannot be completely determined until the program is loaded, are always assigned two byte displacements.

Any address descriptor may be modified by a segment escape prefix. A segment escape prefix consists of a segment register name followed by a colon ':'. The escape causes the assembler to output a segment override prefix using the specified segment register as an operand.

B.3 Assembly Directives

The **.long** and **.blkl** assembler directives are not supported by the iAPX-86 assembler.

The **.even** and **.odd** directives are provided to force the current location counter to an even or odd location respectively (by emitting a 0 byte if necessary).

B.4 Machine Instructions

The following machine instructions are defined. The examples illustrate the general syntax of the operands. Combinations that are syntactically valid may be forbidden for semantic reasons.

In the examples 'a' refers to a general address, 'd' refers to a direct address, 'e' refers to any expression, 'm' refers to a memory address (not an immediate), 'p' refers to a port address, '$e' refers to an immediate expression, and 'ax', 'al', 'cl' and 'dx' refer to the named registers.

Some machine operations that are handled by special syntax in ASM-86 (such as the 'lock' prefix and the repeat prefixes) are treated as ordinary one byte machine operations. The assembler makes no attempt to prevent the generation of incorrect sequences of these prefix bytes.

Although all of the machine operations haye types and values associated with them, in most cases the values have been chosen to assist the assembler in formatting the machine instructions. Their values should not be used by programs, since they may not be constant across different versions of the assembler.

```
aaa
aad
aam
aas
adcb     r,a
adc      r,a
adcb     a,r
adc      a,r
adcb     a,$e
adc      a,$e
andb     r,a
and      r,a
andb     a,r
and      a,r
andb     a,$e
and      a,$e
call     d
cbw
clc
cld
cli
cmc
cmpb     r,a
cmp      r,a
cmpb     a,r
cmp      a,r
cmpb     a,$e
cmp      a,$e
cmpsb
cmps
cwd
daa
das
dec       a
div       m
esc       a
hlt
icall     a       (Indirect)
idiv      m
ijmp      a       (Indirect)
```

```
imul      m
inb       al,p
in        ax,p
inb       al,dx
in        ax,dx
inc       a
int       e
into
iret
ja        d
jae       d
jb        d
jc        d
jbe       d
jcxz      d
je        d
jg        d
jge       d
jl        d
jle       d
jmp       d
jna       d
jnae      d
jnc       d
jnb       d
jnbe      d
jne       d
jng       d
jnge      d
jnl       d
jnle      d
jno       d
jnp       d
jns       d
jnz       d
jo        d
jp        d
jpe       d
jpo       d
js        d
```

```
jz       d
lahf
lds      r,a
lea      r,a
les      r,a
lock
lodsb
lods
loop     d
loope    d
loopne   d
loopnz   d
loopz    d
movb     r,a
mov      r,a
movb     a,r
mov      a,r
movb     a,$e
mov      a,$e
movsb
movs
mul      m
neg      a
nop
notb     a
not      a
orb      r,a
or       r,a
orb      a,r
or       a,r
orb      a,$e
or       a,$e
outb     p,al
out      p,ax
outb     dx,al
out      dx,ax
pop      m
popf
push     m
pushf
```

```
rclb    a,$1
rclb    a,cl
rcl     a,$1
rcl     a,cl
rcrb    a,$1
rcrb    a,cl
rcr     a,$1
rcr     a,cl
rep
repz
repe
repne
repnz
ret
ret     e
rolb    a,$1
rolb    a,cl
rol     a,$1
rol     a,cl
rorb    a,$1
rorb    a,cl
ror     a,$1
ror     a,cl
sahf
salb    a,$1
salb    a,cl
sal     a,$1
sal     a,cl
shlb    a,$1
shlb    a,cl
shl     a,$1
shl     a,cl
sarb    a,$1
sarb    a,cl
sar     a,$1
sar     a,cl
sbbb    r,a
sbb     r,a
sbbb    a,r
sbb     a,r
```

```
sbbb     a,$e
sbb      a,$e
scasb
scas
shlb     a,$1
shlb     a,cl
shl      a,$1
shl      a,cl
shrb     a,$1
shrb     a,cl
shr      a,$1
shr      a,cl
stc
std
sti
stosb
stos
subb     r,a
sub      r,a
subb     a,r
sub      a,r
subb     a,$e
sub      a,$e
testb    r,a
test     r,a
testb    a,$e
test     a,$e
wait
xcall    d,d      (Intersegment)
xchgb    r,a
xchg     r,a
xicall   a        (Intersegment, indirect)
xijmp    a        (Intersegment, indirect)
xjmp     d,d      (Intersegment)
xlat
xorb     r,a
xor      r,a
xorb     a,r
xor      a,r
xorb     a,$e
```

```
xor       a,$e
xret                (Intersegment)
xret      e         (Intersegment)
```

Appendix C: The Zilog Z-8000 Assembler

The assembler for the Zilog Z-8000 microprocessor uses the same machine opcodes and register names as the assembler supplied by the manufacturer. However, the assembler directives are different, and the structured programming features are not supported.

C.1 Predefined Symbols

The following symbols are predefined. The type of the symbol is set to agree with the symbol's use.

rh0	0000	rq12	000C
r0	0000	rl5	000D
rr0	0000	r13	000D
rq0	0000	rl6	000E
rh1	0001	r14	000E
r1	0001	rr14	000E
rh2	0002	rl7	000F
r2	0002	r15	000F
rr2	0002	un	0008
rh3	0003	z	0006
r3	0003	nz	000E
rh4	0004	c	0007
r4	0004	nc	000F
rr4	0004	pl	000D
rq4	0004	mi	0005
rh5	0005	ne	000E
r5	0005	eq	0006
rh6	0006	ov	0004
r6	0006	nov	000C
rr6	0006	pe	0004
rh7	0007	po	000C
r7	0007	ge	0009
rl0	0008	lt	0001
r8	0008	gt	000A
rr8	0008	le	0002
rq8	0008	uge	000F
rl1	0009	ult	0007
r9	0009	ugt	000B
rl2	000A	ule	0003
r10	000A	NVI	0001
rr10	000A	VI	0002
rl3	000B	C	0080
r11	000B	Z	0040
rl4	000C	S	0020
r12	000C	P	0010
rr12	000C	V	0010
FLAGS	0001	FCW	0002
REFRESH	0003	PSAP	0005
NSP	0007		

Most of these symbols are the register names, and have type 'register'. The assembler makes no attempt to distinguish between byte, word, long and quad registers. It will accept machine instructions that use the wrong type of register in a register field; the bit pattern used in the instruction is the value of the built in symbol.

The names of the flags and of the control registers are in upper case to reduce clashes between them and the programmer's labels.

C.2 Address Descriptors

The following syntax is used for general source and destination address descriptors. The syntax is the same as that used by the Zilog assembler, except that the character '$' is used instead of '#' to specify immediate mode.

In the examples the symbol 'r' refers to any register and the symbol 'e' to any expression.

r	R	Register
@r	IR	Indirect register
(r)	IR	Indirect register
$e	IM	Immediate
e	DA	Direct
e(r)	X	Indexed
r(r)	BX	Based index
r(e)	BA	Based

Note that on Z8002 there is no syntax for the based (BA) addressing mode. This mode is identical to the indexed addressing mode on the unsegmented processor.

The assembler understands most of the quirks of the Z-8000 binary representation and attempts to always do something reasonable. For example, it duplicates the value of an immediate mode (IM) address into both halves of the immediate word on byte instructions.

C.3 Assembly Directives

The Z-8000 assembler supports the **.long** and **.blkl** assembly directives.

The **.even** and **.odd** directives are provided to force the current location counter to an even or odd location respectively (by emitting a 0 byte if necessary).

C.4 Machine Instructions

The following machine instructions are defined. The operands have the illustrated syntax.

In the examples 'r' refers to a register, 'a' refers to general source destination address description, 'cc' refers to a condition code name, 'name' refers to a flag name, 'd' refers to a direct address, 'im' refers to an immediate address and 'n' refers to an absolute expression.

The examples illustrate the general syntax of the operands. Combinations that are syntactically valid may be forbidden for semantic reasons. For example, the instruction "ldk r1,#0100" is syntactically legal, but is not accepted because the immediate is out of the legal range for the instruction.

adc	r,r	B500
adcb	r,r	B400
add	r,a	0100
addb	r,a	0000
addl	r,a	1600
and	r,a	0700
andb	r,a	0600
bit	a,$n	2700
bit	a,r	2700
bitb	a,$n	2600
bitb	a,r	2600
call	a	1F00
calr	d	D000
clr	a	0D08
clrb	a	0C08
com	a	0D00
comb	a	0C00
comflg	name	8D05
cp	r,a	0B00
cpb	r,a	0A00
cpd	(r),(r),r,cc	BB08
cpdb	(r),(r),r,cc	BA08
cpdr	(r),(r),r,cc	BB0C
cpdrb	(r),(r),r,cc	BA0C
cpi	(r),(r),r,cc	BB00
cpib	(r),(r),r,cc	BA00
cpir	(r),(r),r,cc	BB04
cpirb	(r),(r),r,cc	BA04
cpl	r,a	1000
cpsd	(r),(r),r,cc	BB0A
cpsdb	(r),(r),r,cc	BA0A
cpsdr	(r),(r),r,cc	BB0E
cpsdrb	(r),(r),r,cc	BA0E
cpsi	(r),(r),r,cc	BB02
cpsib	(r),(r),r,cc	BA02
cpsir	(r),(r),r,cc	BB06
cpsirb	(r),(r),r,cc	BA06
dab	r	B000
dec	a,$n	2B00
decb	a,$n	2A00

di	name	7C03
div	r,a	1B00
divl	r,a	1A00
djnz	r,d	F080
dbjnz	r,d	F000
ei	name	7C07
ex	r,a	2D00
exb	r,a	2C00
exts	r	B10A
extsb	r	B100
extsl	r	B107
halt		7A00
in	r,(r)	3D00
in	r,d	3D00
inb	r,(r)	3C00
inb	r,d	3C00
inc	a,$n	2900
incb	a,$n	2800
ind	(r),(r),r	3B08
indb	(r),(r),r	3A08
indbr	(r),(r),r	3A08
indr	(r),(r),r	3B08
ini	(r),(r),r	3B00
inib	(r),(r),r	3A00
inirb	(r),(r),r	3A00
inir	(r),(r),r	3B00
iret		7B00
jp	cc,a	1E00
jr	cc,d	E000
ld	a,im	2100
ld	a,r	2100
ld	r,a	2100
lda	r,a	0000
ldar	r,d	0000
ldb	a,im	2000
ldb	a,r	2000
ldb	r,a	2000
ldctl	ctlr,r	7D00
ldctl	r,ctlr	7D00
ldctlb	ctlr,r	8C00

ldctlb	r,ctlr	8C00
ldd	(r),(r),r	BB09
lddb	(r),(r),r	BA09
lddr	(r),(r),r	BB09
lddrb	(r),(r),r	BA09
ldi	(r),(r),r	BB01
ldib	(r),(r),r	BA01
ldir	(r),(r),r	BB01
ldirb	(r),(r),r	BA01
ldk	r,$n	BD00
ldl	a,r	1400
ldl	r,a	1400
ldm	a,r,$n	0000
ldm	r,a,$n	0000
ldps	a	3900
ldr	d,r	3100
ldr	r,d	3100
ldrb	d,r	3000
ldrb	r,d	3000
ldrl	d,r	3500
ldrl	r,d	3500
mbit		7B0A
mreq	r	7B0D
mres		7B09
mset		7B08
mult	r,a	1900
multl	r,a	1800
neg	a	0D02
negb	a	0C02
nop		8D07
or	r,a	0500
orb	r,a	0400
otdr	(r),(r),r	3B0A
otdrb	(r),(r),r	3A0A
otir	(r),(r),r	3B02
otirb	(r),(r),r	3A02
out	(r),r	3F00
out	d,r	3F00
outb	(r),r	3E00
outb	d,r	3E00

outd	(r),(r),r	3B0A
outdb	(r),(r),r	3A0A
outi	(r),(r),r	3B02
outib	(r),(r),r	3A02
pop	a,(r)	1700
popl	a,(r)	1500
push	(r),a	1300
pushl	(r),a	1100
res	a,$n	2300
res	a,r	2300
resb	a,$n	2200
resb	a,r	2200
resflg	name	8D03
ret		9E00
ret	cc	9E00
rl	r,im	B300
rlb	r,im	B200
rlc	r,im	B308
rlcb	r,im	B208
rldb	r,r	BE00
rr	r,im	B304
rrb	r,im	B204
rrc	r,im	B30C
rrcb	r,im	B20C
rrdb	r,r	BC00
sbc	r,r	B700
sbcb	r,r	B600
sc	n	7F00
sda	r,r	330B
sdab	r,r	320B
sdal	r,r	330F
sdl	r,r	3303
sdlb	r,r	3203
sdll	r,r	3307
set	a,$n	2500
set	a,r	2500
setb	a,$n	2400
setb	a,r	2400
setflg	name	8D01
sin	r,d	3B05

sinb	r,d	3A05
sind	(r),(r),r	3B09
sindb	(r),(r),r	3A09
sindr	(r),(r),r	3B09
sindrb	(r),(r),r	3A09
sini	(r),(r),r	3B01
sinib	(r),(r),r	3A01
sinir	(r),(r),r	3B01
sinirb	(r),(r),r	3A01
sla	r,im	B309
slab	r,im	B209
slal	r,im	B30D
sll	r,im	B301
sllb	r,im	B201
slll	r,im	B305
sotdr	(r),(r),r	3B0B
sotdrb	(r),(r),r	3A0B
sotir	(r),(r),r	3B03
sotirb	(r),(r),r	3A03
sout	d,r	3B07
soutb	d,r	3A07
soutd	(r),(r),r	3B0B
soutdb	(r),(r),r	3A0B
souti	(r),(r),r	3B03
soutib	(r),(r),r	3A03
sra	r,im	B309
srab	r,im	B209
sral	r,im	B30D
srl	r,im	B301
srlb	r,im	B201
srll	r,im	B305
sub	r,a	0300
subb	r,a	0200
subl	r,a	1200
sys	n	7F00
tcc	cc,r	AF00
tccb	cc,r	AE00
test	a	0D04
testb	a	0C04
testl	a	1C08

trdb	(r),(r),r	B808
trdrb	(r),(r),r	B80C
trib	(r),(r),r	B800
trirb	(r),(r),r	B804
trtdb	(r),(r),r	B80A
trtdrb	(r),(r),r	B80E
trtib	(r),(r),r	B802
trtirb	(r),(r),r	B806
tset	a	0D06
tsetb	a	0C06
xor	r,a	0900
xorb	r,a	0800

Appendix D: The Motorola MC68000 Assembler

The assembler for the Motorola MC68000 microprocessor uses a subset of the machine opcodes and register names provided by the manufacturer's assembler. The names which are unsupported are in all cases longer synonyms for names which are supported. Assembler directives, statement syntax, and expression syntax are different.

D.1 Predefined Symbols

The following register names are predefined.

usp	00FD	sp	000F
ccr	00FE	pc	0010
sr	00FF	d0.l	0800
d0	0000	d1.l	1800
d1	0001	d2.l	2800
d2	0002	d3.l	3800
d3	0003	d4.l	4800
d4	0004	d5.l	5800
d5	0005	d6.l	6800
d6	0006	d7.l	7800
d7	0007	a0.l	8800
a0	0008	a1.l	9800
a1	0009	a2.l	A800
a2	000A	a3.l	B800
a3	000B	a4.l	C800
a4	000C	a5.l	D800
a5	000D	a6.l	E800
a6	000E	a7.l	F800
a7	000F	sp.l	F800

All of these symbols have type 'register' although none are general purpose registers legal in all contexts. The values assigned to ccr, sr, usp, pc and the '.l' registers are for internal use.

D.2 Address Descriptors

The following syntax is used for general source and destination address descriptors. The syntax is a subset of that used by Motorola assemblers, except that the character '$' is used to specify immediate mode, and that the suffix ':s' appended to an absolute address forces absolute short addressing.

In the examples the symbols 'a', 'd', and 'r' refer to address, data, and any register, and the symbol 'e' refers to any expression.

`d`	Data register direct
`a`	Address register direct
`(a)`	Address register indirect
`(a)+`	Address register postincrement
`-(a)`	Address register predecrement
`e(a)`	Address register displacement
`e(a,r)`	Address register short index
`e(a,r.l)`	Address register long index
`e:s`	Absolute short address
`e`	Absolute long address
`e(pc)`	Program counter displacement
`e(pc,r)`	Program counter short index
`e(pc,r.l)`	Program counter long index
`$e`	Immediate data

Unsupported address constructions are synonyms for constructions listed above.

The addressing modes are classified into four categories which are used in the instruction listings to distinguish allowed addresses.

Data addresses are all addresses except address registers.

Memory addresses are all addresses except data and address registers.

Control addresses are all memory addresses except address register predecrement and address register postincrement.

Alterable addresses are all addresses except program counter displacement, program counter index, and immediate.

Failure to observe category restrictions will generate address errors.

D.3 Assembly Directives

The MC68000 assembler supports the **.long** and **.blkl** assembly directives.

The **.even** and **.odd** directives are provided to force the current location counter to an even or odd location respectively (by emitting a '0' byte if necessary).

D.4 Machine Instructions

The following machine instructions are defined. For the most part they form a subset of the instructions provided by Motorola assemblers which eliminates long synonyms such as 'bsr.l' or 'add.w'. The conditions 'hs' and 'lo' are provided as synonyms for 'cc' and 'cs'. Register list syntax for 'movem' is not supported.

In the examples 'an', 'dn', and 'rn' refer to address, data, and either registers, 'ea' refers to general effective addresses, 'l' refers to direct addresses, 'e' refers to a general expression, and 'n' refers to an absolute expression.

Many syntactically correct instructions may prove to be semantic errors because of restrictions of effective addresses to data, alterable, memory, or control categories. Contrary to appearances, no MC68000 instruction operates on all addressing modes; some modes are always forbidden. These restrictions are noted at the end of each instruction description in the MC68000 User's Manual. In the following listing instructions have been classified according to their allowed addressing modes. Each classification is named by the lexicographically first instruction in the class.

D.4.1 ABCD Type

The following instructions accept only two kinds of operands: data register direct and address register predecrement. The BCD instructions operate on byte size operands only.

```
abcd    dn,dn
abcd    -(an),-(an)

abcd    C100
addx    D140
addx.b  D100
addx.l  D180
sbcd    8100
subx    9140
subx.b  9100
subx.l  9180
```

D.4.2 ADD Type

The following instructions take a data register source to a memory-alterable destination or any source to a data register destination. If the operation size is byte, then address register direct sources are forbidden.

```
add     dn,ea
add     ea,dn

add     D040
add.b   D000
add.l   D080
sub     9040
sub.b   9000
sub.l   9080
```

D.4.3 ADDA Type

The following instructions accept any source effective address. The ‘cmp’ instruction cannot combine byte operations with address register sources.

```
adda     ea,an   D0C0
adda.l   ea,an   D1C0
cmp      ea,dn   B040
cmp.b    ea,dn   B000
cmp.l    ea,dn   B080
cmpa     ea,an   B0C0
cmpa.l   ea,an   B1C0
movea    ea,an   3040
movea.l  ea,an   2040
suba     ea,an   90C0
suba.l   ea,an   91C0
```

D.4.4 ADDI Type

The following instructions require a data-alterable destination effective address. The nbcd, set according to condition, and tas instructions are implicitly byte sized.

addi	$n,ea	0640
addi.b	$n,ea	0600
addi.l	$n,ea	0680
clr	ea	4240
clr.b	ea	4200
clr.l	ea	4280
cmpi	$n,ea	0C40
cmpi.b	$n,ea	0C00
cmpi.l	$n,ea	0C80
eor	dn,ea	B140
eor.b	dn,ea	B100
eor.l	dn,ea	B180
nbcd	ea	4800
neg	ea	4440
neg.b	ea	4400
neg.l	ea	4480
negx	ea	4040
negx.b	ea	4000
negx.l	ea	4080
not	ea	4640
not.b	ea	4600
not.l	ea	4680
scc	ea	54C0
scs	ea	55C0
seq	ea	57C0
sf	ea	51C0
sge	ea	5CC0
sgt	ea	5EC0
shi	ea	52C0
shs	ea	54C0
sle	ea	5FC0
slo	ea	55C0
sls	ea	53C0
slt	ea	5DC0
smi	ea	5BC0
sne	ea	56C0
spl	ea	5AC0
st	ea	50C0
subi	$n,ea	0440
subi.b	$n,ea	0400

```
subi.l  $n,ea   0480
svc     ea      58C0
svs     ea      59C0
tas     ea      4AC0
tst     ea      4A40
tst.b   ea      4A00
tst.l   ea      4A80
```

D.4.5 ADDQ Type

The following instructions take an immediate source operand in the range 1 to 8 and an alterable effective address destination operand. If the operation size is byte, then address register direct destinations are forbidden.

```
addq    $n,ea   5040
addq.b  $n,ea   5000
addq.l  $n,ea   5080
subq    $n,ea   5140
subq.b  $n,ea   5100
subq.l  $n,ea   5180
```

D.4.6 AND Type

The following instructions take two forms: data register direct source to memory-alterable destinations, and data source effective address to a data register direct destination.

```
and     dn,ea
and     ea,dn

and     C040
and.b   C000
and.l   C080
or      8040
or.b    8000
or.l    8080
```

D.4.7 ANDI Type

The following instructions combine an immediate source operand with either a data-alterable effective address destination operand or the status register. The whole status register or only the low byte is selected depending on whether the operation size is word or byte.

```
andi    $n,ea
andi    $n,sr

andi    0240
andi.b  0200
andi.l  0280
eori    0A40
eori.b  0A00
eori.l  0A80
ori     0040
ori.b   0000
ori.l   0080
```

D.4.8 ASL Type

The shift instructions come in three flavors: immediate shift count of data register, data register shift count of data register, and shift by one of a word at a memory-alterable effective address. The memory shift opcode is formed from the opcodes given by setting bits 6 – 7, and by moving bits 3 – 4 to positions 9 – 10.

```
asl     $n,dn
asl     dn,dn
asl     ea

asl     E140
asl.b   E100
asl.l   E180
asr     E040
asr.b   E000
asr.l   E080
lsl     E148
lsl.b   E108
lsl.l   E188
lsr     E048
lsr.b   E008
lsr.l   E088
rol     E158
rol.b   E118
rol.l   E198
ror     E058
ror.b   E018
ror.l   E098
roxl    E150
roxl.b  E110
roxl.l  E190
roxr    E050
roxr.b  E010
roxr.l  E090
```

D.4.9 BCHG Type

The bit instructions take an immediate or data register source operand and a data-alterable destination effective address. The operation size is implicitly long for data register destinations and implicitly byte for other destinations.

```
bchg    $n,ea
bchg    dn,ea

bchg    0140
bclr    0180
bset    01C0
btst    0100
```

D.4.10 CHK Type

The following instructions take a data source effective address and a data register destination. Source and destination are implicitly word sized for chk, muls, and mulu. Source is word sized and destination is long for divs and divu.

```
chk     ea,dn   4180
divs    ea,dn   81C0
divu    ea,dn   80C0
muls    ea,dn   C1C0
mulu    ea,dn   C0C0
```

D.4.11 JMP Type

The following instructions require control effective addresses.

```
jmp     ea      4EC0
jsr     ea      4E80
lea     ea,an   41C0
pea     ea      4840
```

D.4.12 MOVE Type

Move instructions take any source effective address to data-alterable destination effective addresses, but byte moves from address registers are forbidden. When the destination is the condition code or status register the source must be a data effective address and the instruction size is implicitly byte or word respectively. When the status register is the source the destination must be a data-alterable effective address. When the user stack pointer is an operand the other operand is an address register and the instruction size is implicitly long.

```
move    ea,ea   3000
move.b  ea,ea   1000
move.l  ea,ea   2000
move    ea,ccr  44C0
move    ea,sr   46C0
move    sr,ea   40C0
move    an,usp  4E60
move    usp,an  4E68
```

D.4.13 MOVEM Type

Movem instructions take two forms: an immediate register mask source with a control or predecrement destination; or a control or postincrement source with an immediate register mask destination. The bit ordering in register masks is the programmer's responsibility.

```
movem   $n,ea   4880
movem   ea,$n   4C80
movem.l $n,ea   48C0
movem.l ea,$n   4CC0
```

D.4.14 MOVEP Type

The move peripheral instruction uses data register and address register indirect with displacement operands.

```
movep   e(an),dn 0108
movep   dn,e(an) 0188
movep.l e(an),dn 0148
movep.l dn,e(an) 01C8
```

D.4.15 Miscellaneous Instructions

The remaining instructions have operand syntax which is self explanatory.

bcc	l	6400
bcc.s	l	6400
bcs	l	6500
bcs.s	l	6500
beq	l	6700
beq.s	l	6700
bge	l	6C00
bge.s	l	6C00
bgt	l	6E00
bgt.s	l	6E00
bhi	l	6200
bhi.s	l	6200
bhs	l	6400
bhs.s	l	6400
ble	l	6F00
ble.s	l	6F00
blo	l	6500
blo.s	l	6500
bls	l	6300
bls.s	l	6300
blt	l	6D00
blt.s	l	6D00
bmi	l	6B00
bmi.s	l	6B00
bne	l	6600
bne.s	l	6600
bpl	l	6A00
bpl.s	l	6A00
bra	l	6000
bra.s	l	6000
bsr	l	6100
bsr.s	l	6100
bvc	l	6800
bvc.s	l	6800
bvs	l	6900
bvs.s	l	6900
cmpm	(an)+,(an)+	B148
cmpm.b	(an)+,(an)+	B108
cmpm.l	(an)+,(an)+	B188
dbcc	dn,l	54C8

dbcs	dn,l	55C8
dbeq	dn,l	57C8
dbf	dn,l	51C8
dbge	dn,l	5CC8
dbgt	dn,l	5EC8
dbhi	dn,l	52C8
dbhs	dn,l	54C8
dble	dn,l	5FC8
dblo	dn,l	55C8
dbls	dn,l	53C8
dblt	dn,l	5DC8
dbmi	dn,l	5BC8
dbne	dn,l	56C8
dbpl	dn,l	5AC8
dbra	dn,l	50C8
dbt	dn,l	50C8
dbvc	dn,l	58C8
dbvs	dn,l	59C8
exg	rn,rn	C100
ext	dn	4880
ext.l	dn	48C0
link	an,$n	4E50
moveq	$n,dn	7000
nop		4E71
reset		4E70
rte		4E73
rtr		4E77
rts		4E75
stop	$n	4E72
swap	dn	4840
trap	$n	4E40
trapv		4E76
unlk	an	4E58

Index

User Reaction Report

To keep this manual and COHERENT free of bugs and facilitate future improvements, we would appreciate receiving your reactions. Please fill in the appropriate sections below and mail to us. Thank you.

Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614

Name: ______________________________

Company: ______________________________

Address: ______________________________

____

Phone: ______________ Date: ______________

Version and hardware used: ______________________________

Did you find any errors in the manual? ______________________________

____

____

Can you suggest any improvements to the manual? ______________________________

____

____

Did you find any bugs in the software? ______________________________

____

____

Can you suggest improvements or enhancements to the software?

____

____

____

Additional comments: (Please use other side.)