H S-M 20.12.73.

RCSL: 061

pp: 1 : 45

Author: Flemming Hermansen/
Henning Jakobsen

Edited: August 1972

RC 3500
ASSEMBLER MANUAL

Keywords: RC 3500 Assembler Manual
RC 4000
CDC Scope 3.4

Abstract: This paper is a description of SLAM 2 - the symbolic assembler language for the RC 3500 computer - designed to operate under RC 4000 file processor. A version for the CDC SCOPE 3.4 is also included.

A/S REGNECENTRALEN
Guldsmedegade 3,
DK-8000 Aarhus C.

CONTENTS

Section	page
1. GENERAL DESCRIPTION	1-1
1.1. Language features	1-1
1.2. Programming examples	1-2
2. BASIC ELEMENTS OF SLAM	2-1
2.1. Character set	2-1
2.2. Directives and Comments	2-1
2.3. Numbers	2-2
2.4. Identifiers	2-3
2.5. Expressions	2-4
2.6. Instructions	2-7
2.7. Datawords	2-10
3. STRUCTURE OF SLAM PROGRAMS	3-1
3.1. Datawordmode	3-1
3.2. Labels	3-2
3.3. Assignment Statements	3-3
3.4. Repetition Statements	3-4
3.5. Conditional Assembly	3-5
3.6. Programs and Blocks	3-6
A. APPENDIX A	
OPERATION OF THE ASSEMBLER ON RC 4000	A-1
A.1. Introduction	A-1
A.2. Size and Speed	A-1
A.3. Calling the Assembler	A-2
A.4. Format of Binary Program	A-3
A.5. Assembly Options	A-4
A.6. Assembly messages	A-8

CONTENTS (Continued)

Section		page
B.	APPENDIX B	
	OPERATION OF THE ASSEMBLER ON CYBER 73	B-1
	B.1. Introduction	B-1
	B.2. Changes to Chapter 2.	B-2
	OPERATION OF THE ASSEMBLER	B-4
	B.3 Introduction	B-4
	B.4. Size and Speed	B-5
	B.5. Calling the Assembler	B-6
	B.6. Format of pseudo binary programs	B-7
	B.7. Assembly Options	B-8

pp. 1 : 45

This manual is a description of SLAM 2 - the symbolic language for the RC 3500. Slam 2 is a pure extension of Slam 1. The extension concerns the use of undefined expressions in assignment statements.

Chapter 1 contains a general description of the language. Chapter 2 introduces the basic constituents of Slam, which are: directives, numbers, identifiers, expressions, datawords, and instructions. The overall strcuture of Slam programs is defined in Chapter 3. The syntax of the language is described by means of Backus notation.

The Slam language was defined and implemented by the authors. The manual is concluded by descriptions of the operation of the Slam assembler.

Appendix A. Operation on RC 4000

Appendix B. Operation on CYBER 73

Appendix C. Operation on IBM 370

1-1

Language features.

In Slam the contents of storage locations are described as instructions or datawords. Instructions and datawords can refer to other storage locations by means of symbolic addresses called identifiers.

An identifier can also refer to the value of an expression defined during assembly.

Assembly expressions are written in conventional arithmetic notation using parantheses and the operators + - * / < (logical left shift) > (logical right shift) a. (logical and) o. (logical or). The operands are integers and identifiers. In expressions, which are used in instructions or datawords, the identifiers need only to be defined somewhere. In other expressions all identifiers must be defined by previous use as labels or in assignment statements.

With respect to the scope and usage of identifiers, Slam has a block structure similar to Algol 60: identifiers must be declared in a block head before they are used. They can only be used within the block in which they are declared and in blocks enclosed by the former.

The outermost block, which contains all other blocks, delimits the entire program. To facilitate the handling of programs, that exist in several versions, Slam includes conditional assembly in the following form: a piece of code can be headed by an expression, which is evaluated by the assembler. If the result is positive or zero, assembly will continue after the expression. A negative result, however, will cause the assembler to skip the following statements up to a given delimiter.

The following examples illustrate the overall structure of Slam programs.
Let us first consider the following piece of code:

```
b.a 10;multiply
;this procedure multiplies w1
;and w2 and places the result
;in the same registers
;with the least significant bit
;in w2 position 15 and the most
;significant bit in w1 pos. 0. The operands
;may be negative, but the absolute
;value of w1 must not exceed
;16384 because of overflow
;the cpa-time required
;depends on op 2, but
;is about o.5 ms pr. call

;       call                return
;w1     operand 1           result
;w2     operand 2           result
;w3     return address      0

w.
c.(a90>3a.1)-1              ;if testmode then
a0:     blw w1 i - 16384,a0;test range of
        bgw w1 i 16384,a0 ;operand 1

z.
        sti w3 i            a5  ;save return address
        sti w1 i            a2  ;store op 1
        ldr w1 i            0   ;result:=0
        ldr w3 i            15  ;count:=15
                                ;loop:
```

```
a1:   bsz w2i      1,a3 ;if next bit of op2=1 then
      addcwl i          ;result:=result+op1

a2:   0                 ;op1 is stored here
a3:   asr w1       1    ;these shifts will
      ror w2       1    ;act as multiplying op2 with 2
      sub w3i      1    ;count:=count-1
      bgw w3i      0,a1 ;if count>0 then go to loop

      bsz  w2i     1,a4 ;if op2<0 then
      sub w1       a2   ;result:=result-op1

a4:   asr w1       1    ;shift the result
      ror w2       1    ;into position 15 of w2
      ldr cw0 i         ;return

a5:   0
```

e.; multiply

The text preceded by semicolons are comments ignored by the assembler.

The example shows a block of code enclosed by begin and end delimiters b. and e. The block begin is followed by a declaration a10 indicating that the programmer can use the - the names a0,a1,a2,---a10 as local identifiers within the block.

The following lines each describe a single machine instruction or data-word. The delimiter w. means that they are assembled as words (in contrast to bytes).

An instruction like the following:

```
sti w3i a5
```

consists of a mnemonic operation code (sti meaning store indirect),

a modification part (w3 meaning working register 3, and i meaning immediate addressing), and an address part (a5 referring to the word labelled a5:). As the above instruction exists in several formats, the assembler chooses a suitable format. In the following instruction:

```
add cw1 i
```

the programmer has explicitly chosen format c. Furthermore the address part is missing, because the programmer wants the instruction to take its address from the succeeding location.

In the following examples of other instructions are given:

```
ldr a w1 i n X3+b3
bew c w1 i n X2+c17,b31
lsr w1      c2+1
ild w2 s    X3 31
rbb w3      X2 d1
```

Conditional assembly is illustrated by the statement

```
c.(a90>3 a.1)-1
a0:     blw w1i-16384,a0
        bgw wli 16384,a0
z.
```

The value of the expression (a90>3 a.1)-1 determines whether the two following instructions are assembled or skipped. It is assumed that the assembly condition has been defined previously in the program by assignment to a90:

```
a90=1<3 ;testmode:=true
```

or

```
a90=0   ;testmode:=false
```

The next example illustrates the assembly of datawords:

```
h.f2:8.203,115,c15 ;byte values
w.f3:c16x3,1,8.3003 ;word values
```

Byte values are prefixed by the delimiter h. The first value 8.203 is an octal number with the digits 203. It is followed by a decimal number 115. The last byte will get the value of the identifier, which must be defined elsewhere. Instructions and data can be labelled by identifiers followed by a colon, for example:

f3:

Finally, we give the skeleton of a program consisting of several blocks in order to illustrate the nesting of blocks:

```
b.---                          ;level 0, global block
    b.---                      ;level 1
        b.---                  ;level 2
        e.
        b.---                  ;level 2
        e.
    e.
    b.---                      ;level 1
        b.---                  ;level 2
            b.---              ;level 3
            e.
        e.
    e.
e.
```

Character Set 2.1

Syntax 2.1.1

<assembly character>::=<digit>!<letter>!<blind>!<sp>!<new line>
<special character>

<digit>::=0!1!2!3!4!5!6!7!8!9

<letter>33=a!b!c!d!e!f!g!h!i!j!k!l!m!n!o!p!q!r!s!t!u!v!x!y!z!æ!ø!å
A!B!C!D!E!F!G!H!I!J!K!L!M!N!O!P!Q!R!S!T!U!V!W!X!Y!Z!Æ!Ø!Å

<sp>::=<SP>!<HT>

<new line>::=<NL>!<FF>!<EM>

<special character>::=&!(!)*!+!,!-!.!/!:!;!<!=!>!<exclamation sign>

<blind>::=<NUL>!<DEL>

Semantics 2.1.2

Slam accepts the subset of the ISO 7-bit character set, as described in section 2.1.1. All other characters are illegal.

Letters, Spaces, and Horizontal Tabulation. 2.1.3.

Slam does not distinguish between small letters and capital letters. Spaces and Horizontal tabulation are treated as blind characters; however, they can be used to separate items. As these characters are used to separate basic items, they must not occur within the basic items (i.e. operation parts, numbers, radix numbers, identifiers <w-register>,<x-register>,<letters followed by point>)

Directives and Comments 2.2

Syntax 2.2.1

<directive>::=<input control>!<testoutput directive>!
m.<line comment><newline>!<separator>

<input control>::= t. ! n.

<testoutput control>::= ø. !æ.

<separator>::= ,!<endline>

<endline>::= <new line>!;<line comment> <new line>!

<line comment::=<any string not containing a newline>

Semantics. 2.2.2.

Directives have no effect on the binary code produced, but serve to control the assembly.

Input control. 2.2.3

The delimiters t. and n. permit switching between two alternative input mediums during assembly. The exact functions of the input control depend on surrounding system (see app. A.5).

Testoutput Control. 2.2.4

The delimiter ϕ. causes the assembler to print internal testoutput during assembly; it can be turned off again by means of the delimiter œ. This facility is used to debug the assembler. For further information see ~~section 4.5.~~ appendix A5.

Line comments. 2.2.5

Comments can be included in the program following the delimiter m. or a semicolon. This causes everything up to a new line to be ignored. Comments following the delimiter m. are messages which are displayed on the output medium during assembly.

Numbers 2.3

Syntax 2.3.1

<number>::=<unsigned integer>!<radix number>

<radix number>::=<radix> <unsigned integer>

<radix>::=<unsigned integer>.

<unsigned integer>::=<digit>!<unsigned integer> <digit>

Semantics. 2.3.2

Slam accepts radix numbers and unsigned integer. Unsigned integers are decimal numbers. Radix numbers are expressed as a radix followed by a digitstring, for example 2.110111 (a binary number) or 8.4357 (an octal number).

The radix must not be greater than 9. The number is converted digit by digit as follows:

number:=radix*number+digit

in fact, the present version of the assembler reads the digit string as a decimal number and converts it afterwards. This method restricts the digit string according to the capacity of RC 4000, which is about six digits. Greater numbers must be written using the right-shift operator or multiplication operator, for example: 2.100<6+2.101100 or 3.201x27x27+3.201010.

Identifiers. 2.4

Syntax. 2.4.1

<identifier>::=<declared identifier>!<load address>

<declared identifier>::=<identifier letter> <index>

<index>::= <unsigned integer>

<identifier letter>::=a!b!c!d!e!f!g!h!i!j

<load address>::=k

Semantics. 2.4.2

An identifier is a symbolic name given either to a storage location in the object program or to the value of an expression evaluated by the assembler. The index must be in the range 0 through 127.

Load Address. 2.4.3

The letter k is a reserved identifier for the load address of the current byte or word. K is initialized to zero before assembly and increased concurrently with the production of the assembled program.

Defined and undefined identifiers. 2.4.4

A declared identifier can in a given place in the program-text either be defined or undefined.

When the identifier is declared it will be in the state: "undefined". When the identifier is used by a label or is defined by a defined expression, it turns to the state "defined". When it is defined by an undefined expression it turns to the state "undefined".

The load address will always be defined.

Examples. 2.4.5

```
k       a0      d 25      h 123
```

Expressions. 2.5

Syntax. 2.5.1

<expression>::=<sign> <term>!<expression> <operator> <term>

<term>::=<operand>!(<expression>)

<operator>::=+!-!*!/!<!>!a!o!&!<exclamation sign>

<sign>::=<empty>!+!-

<operand>::=<unsigned integer>!<radixnumber>!<identifier>

Semantics. 2.5.2

An expression specifies the computation of an integer value during assembly. All operands and intermediate results are evaluated as 16 bit integers.

Defined and undefined expressions. 2.5.3

If all occuring identifiers in an expression are defined, then the expression is said to be defined. Otherwise it is said to be undefined. If the expression is used in k-assignment-, conditional-, or repeat statements, then the expression must be defined. In all other cases undefined expressions are allowed, presupposed that the undefined identifiers are defined somewhere, according to the following rules for each undefined identifier in the expression.

1. If the identifier has never occurred as labels and never been defined by assignment-statements in the preceeding text, the value of the identifier will be the one, which it gets by the first succeeding use as label or definition by assignment statement. In case of an assignment-statement, the expression in the assignment statement must not be undefined.

2. If the identifier is undefined and has indeed been defined in the preceeding text, the last definition of the identifier must have been an assignment-statemnet with an undefined expression. The value of the identifier will in this case be the value of this expression.

The explanation of the above rules of use of undefined identifiers is the one, that the assembler for the language should be able to be constructed as a single two-pass assembler.

Operators. 2.5.4

The arithmetic operators +-*/ have their conventional meanings of addition, subtraction, multiplication, and division. The shift operators "<" and ">" shift the left-hand operand logically left and right the number of bit positions specified by the right-hand operand. The Boolean operators a. and o. form the and and or combinations of the two operands bit by bit. Instead of a. and o. It is allowed to use & and !.

Precedence of Operators. 2.5.5

The sequence of operation in evaluating an expression is from left to right subject to the following rules of precedence among the operators:

first:	< >
second:	* /
third:	+ -
fourth:	a. &
fifth:	o. !

This can be overruled, however, by the use of parantheses "(" and ")".

Examples. 2.5.6

```
1
-a5
k+128
2.1011<12+6<8+g4
((-c35-c0)/5)<3 o. 6.323
```

Instructions.

2.6.1 Syntax.

<instruction>::=

<arithm. instruction> <endline>!

<branch instruction> <endline>!

<shift instruction> <endline>!

<interlevel instruction> <endline>!

<I/O instruction> <endline>

<arithm. instruction>::=

<arithm. opcode> <format 1/2 mod.> <address part>

<branch instruction>::=

<branch opcode> <format 1/2 mod.> <branch address part>

<shift instruction>::=

<shift opcode> <shift mod.> <address part>

<interlevel instruction>::=

<interlevel opcode> <interlevel mod.> <address part>

<I/O instruction>::=

<I/O opcode> <I/O mod.> <address part>

<format 1/2 mod.>::= { <format> | <w-register> | <immediate> | <next/not> | <x-register> } 5 … 0

<interlevel mod.>::= { <w-register> | <eds> | <x-register> } 3 … 0

<shift mod.>::=<w-register>!<empty>

<I/O mod.>::= { <w-register> | <x-register> } 2 … 0

<format>::=a!b!c

<w-register>::= w0 ! w1 ! w2 ! w3

<x-register>::= x0 ! x1 ! x2 ! x3

<immediate>::= i

<next/not>::= n

<eds>::= e ! d ! s

<branch address part>::=<address part>,<address part>

<address part>::=<expression>

<arithm. opcode>::=add!and!lce!ldb!ldc!ldi!ldr!lor!
rsw!stb!sti!str!sub!xor!xsr!

<branch opcode>::=beb!bew!bgb!bgw!blb!blw!bop!bsz!

<shift opcode>::=asr!lsl!lsr!rol!ror

<interlevel opcode>::=ild!ist

<I/O opcode>::=rbb!rbw!rsc!rwc!wbb!wbw!wcc!wwc

Operation parts. 2.6.2

The operation part of an instruction is assembled in accordance with the reference manual. The modifications may follow in any order, but each modification must only occur once. A missing modification is assembled as a zero in the corresponding part of the instruction.

Address parts. 2.6.3

The value of the address part must not be so great that it cannot fit into the instruction format, i.e. it must fulfil the following conditions:

Type of address part	Range
zero bit address part (format b)	0 : 0
four bit address part (shift instructions)	0 : 15
five bit address part (interlevel/I/O instructions)	0 : 31
seven bit address part (format a)	- 64 : 63
eight bit byte (see section 2.7 and 3.1)	0 : 255
sixteen bit address part (format c and second address part of branch instructions)	- 32768 32767

If the format is not specified, the assembler chooses a format according to the following rule:

If the address part does not contain any undefined identifier, the expression is evaluated, and the value determines the format, as format a is preferred to format b, and format b to format c. Otherwise format c is chosen. An undefined identifier is an identifier, to which no value has been assigned by the previous source text.

In order to assemble only the first word of an instruction, the following special conventions have been introduced:

If format c is specified explicitly for arithmetic and branch instructions, the address part and branch address part may be omitted. If format a or b are specified explicitly for branch instructions, the last expression of the branch address may be omitted. For example:

```
ldr c
bew a     3,
bew b     0,
bew c
```

This convention is because of convenience, however, not shown in the Backus notation in section 2.6.1.

Datawords. 2.7

Syntax. 2.7.1

<dataword> : :=<expression> <separator>

Semantics. 2.7.2

The expression may contain undefined identifiers.

The expression is evaluated by the assembler and stored in the assembled program.

If halfwordmode is selected (see section 3.1), thenthe expression must be in the range 0 : 255. Otherwise it must be in the range - 32768 : 32767.

Examples. 2.7.3

```
c17 * 3 - (c19 + 3)
```

Datawordmode. 3.1

Syntax. 3.1.1

<datawordmode selector>::=h. ! w.

Semantics. 3.1.2

The datawordmode is set to wordmode by the delimiter w. and to halfwordmode by the delimiter h. If the load address is odd, the delimiter w. further has the effect of loading a byte value zero and increasing the load address accordingly. In halfwordmode instructions are not allowed.

Labels. 3.2

Syntax. 3.2.1

<label>::=<declared identifier>:

Semantics. 3.2.2.

Labels are used as symbolic addresses of storage location. The identifier in a label gets the current load address as its value. If the identifier had been defined by previous labels or assignment, the assembler furthermore checks that its value corresponds to the current load address.

Examples. 3.2.3

```
a3:          e22:          j0:
```

Assignment Statements.

3.3

3.3.1

Syntax.

<assignment>::=<identifier>=<expression> <separator>

3.3.2

Semantics.

An assignment statement assigns the value of an expression to an identifier. A defined identifier can be redefined by an assignment. Declared identifiers may be defined by undefined expressions, but the load address (k) can only be defined by defined expressions.

3.3.3

Load Address Control.

The load address k can be changed explicitly by an assignment statement. This effects the values of subsequent identifiers used as labels.

In wordmode the expression must be even. If it is odd, the assembler will increase it by one before assigning it to the load address.

The binary code produced does not contain explicit information about the change of load address to be used when the program is actually loaded and executed. The conventions for program loading in a given system is outside the scope of the Slam language.

3.3.4

Examples.

```
k = 8
k = k - 150
c6 = b6 - (g32 + 4)
i4 = 2.101 < 12 + 12
```

Repetition Statements. 3.4

Syntax. 3.4.1

<repetition statement>::=r.<expression> <separator>

Semantics. 3.4.2

The expression must not contain any undefined identifiers. The expression is evaluated by the assembler. If the result is greater than zero, the assembler will repeat the last byte or word <expression>-1 times. In wordmode the word is repeated, and in halfwordmode the byte is repeated (see section 3.1).

Examples. 3.4.3

```
r.30
r. c7 * c8 + c9
```

Conditional assembly. 3.5

Syntax. 3.5.1

<condition head>::=c.<expression> <separator>

<condition tail>::=z.

Semantics. 3.5.2

The expression must not contain any undefined identifiers.

Conditional assembly of a piece of Slam code is specified by the delimiter c. followed by an expression that is evaluated by the assembler. If the result is greater than or equal to zero, assembly continues after the expression. A negative result indicates that the following statements up to the delimiter z. are to be skipped.

Assembly conditions can be nested, but do not necessarily follow the block structure (see section 3.6) of the program, i.e. a sequence such as:

```
c. --- b. --- z. --- e.
```

is accepted.

Examples. 3.5.3

```
a1=2,c. (a90>12a.1)-1,a1=3,z.
    c.-1 ;comment skip until z in the next line
    z.
```

Programs and Blocks. 3.6

Syntax. 3.6.1

<program>::=<program comment> <block>

<program comment>::=<any string not containing b.>

<block>::b.<declarations> <endline> <statements>e.!b. <endline> <statements>e.

<declarations>::=<declared identifier>!<declared identifier>,<declarations>

<statements>::=<empty>!

<statements> statements>!

<instruction>!<dataword>!

<directive>!<label>!

<assignment>!

<dataword mode selector>!

<repetition statement>!

<condition head>!

<condition tail>!

<block>

Semantics. 3.6.2

A block is a sequence of statements enclosed by the delimiter pair b. and e. Blocks can be nested. The block concept is introduced to control the scope of identifiers in the following way: first, all identifiers must be declared in a block head before being used; second, identifiers can only be used within the block in which they are declared and in blocks enclosed by the former; third, if the same identifier is declared twice in several nested blocks, it will function as several identifiers, each of which can only be used within its own block.

in halfwordmode (see section 3.1) instructions are not allowed. This is, however, not shown by the Backus notation.

Declarations. 3.6.3

at each block entry the assembler reserves a table to contain the values of identifiers declared in that block. A declaration consists of one of the letters a through j followed by an unsigned integer. This permits the programmer to use any identifier beginning with the selected letter followed by an integer less than or equal to the limit specified. For example, the declaration:

b.;c27,a9

initializes a block in which the identifiers c0 through c27 and a0 through a9 can be used. A block end e. deletes all local identifier values from the internal table.

Program Comment. 3.6.4

The input control directives t. and n. as well as line comments preceeded by a semicolon are allowed before the first block head of the program; all other characters are, however, treated as a comment to the program.

APPENDIX A

OPERATION OF THE ASSEMBLER ON RC 4000

A

Introduction.

A.1

This appendix describes the features and the operation of the Slam assembler. The assembler is programmed in Algol 6 of the RC 4000 computer and must be run on this computer.

The assembler takes its input fron one or more text files described in the catalog on the backing store; it outputs a binary program on a single file. The assembler is written as a two-pass assembler and uses therefore an internal working area on the backing store for the internal output. Further it uses an internal area holding information about the blocks and identifiers.

Size and Speed.

A.2

As the program is written in Algol 6, the speed highly depends on the process size. The number of locations used for the stack is independent of the Slam program. The speed has empirically been found to about

characters per second in a 30 K bytes process and to about characters per second in a 20 K process. The minimum process size, in which the assembler can run is about K bytes.

Calling the Assembler.

The assembler is called by the file processor, and the call has the following format:

<result>slam<parameter list>

The <result> can be empty of of the form:

<file>=

The <parameter list> can be empty or contain one or more parameters of the form:

<source file>

or <assembly option>.yes

or <assembly option>.no

or <assembly option> .<number>

The assembler can input the source program from one or more text files given in the parameter list: when the first source file is exhausted, input is taken from the next source file, and so on. A source file is identified by a name referring to a catalog entry, which describes an input document and its initial position.

If no source file is specified, input is taken from the current input medium; if this medium is emptied, assembly is terminated.

The assembler can output the object program on a single result file. If no result file is specified, the assembler does not output binary code. The size of the backing store file is not changed by the assembler.

N.B.: As the compiler is written in Algol 6, it should be noticed that if the first parameter in the call is a name not followed by a point, this file will be used both as current input medium and as the first source file in the list. If especially the first parameter is trf, the system will load the first tape for current input. Then the assembler will skip this tape in order to read from the first input medium which is also trf.

Format of Binary Program.

Each RC 3500 machine word is stored in an RC 4000 machine word. As the RC 4000 word length is 24 bits, the RC 3500 word is written in bit positions 8 through 23, and bits 0 through 7 are filled with zeroes.

After the last assembled word the assembler places a word containing ones in each bit position in order to signal the end of the binary program. The file containing the binary program is written by the Algol 6 procedure outrec6 with a share length of 128 reals.

Although the assembler can generate output on every document, the program punch 16 should be used to generate paper tapes, because paper tapes generated by the assembler cannot be read by the RC 3500. See the description of punch 16 for further explanation.

Assembly Options.

The following is a list of assembly modes that can be chosen by explicit parameters to Slam:

type.yes type.no	Defines whether the delimiter t. should cause the assembler to take input from the current input medium. The delimiter n. returns the assembler to the normal source file given in the parameter list.
list.yes list.no	Defines whether the assembler should list the source text (including the "typed" input) on the current output medium.
names.yes names.no	Defines whether the assembler should print the blocks with the identifier values on the current output medium.
message.yes message.no	Defines whether the delimiter m. should cause the assembler to print comments on the current output medium.
print.yes print.no	Defines whether the assembler should print the binary program. If yes, each word of the program is printed on current output medium in octal form during pass 2 of the assembler.
formatc.yes formatc.no	Defines whether the assembler should choose a suitable format or if it should use format c for all arithmetic and branch instructions. For debugging purposes of a Slam program the formatc option has the advantage that you can correct the instruction manually without having to assemble the program.

warning.yes warning.no	Defines whether the assembler should print warning messages on the current output medium.
stack.<number>	Defines the length counted in segments of the internal backing store area holding the information of blocks and names.
work.<number>	Defines the length counted in segments of the backing store area holding the internal output.
test.yes test.no test *.<number> 1	Defines the bit configuration in the test mask. test.yes sets all bits equal to one; test.no sets all bits equal to zero; test *.<number> 1 sets the bit "1 shift number" equal to 1; When the assembler reads the delimiter ϕ., the normal mask is exchanged for the test mask; when it reads the delimiter œ, it switches back to the normal mask.

The meaning of the bits in the masks is explained in the following table:

1 shift 23	The assembler prints pass-output on current output medium.
1 shift 22	The assembler lists the program. In the normal mask this bit is set by the assembler option list.yes.
1 shift 21	The delimiter t. cuases the assembler to take its input from the current input medium. The assembly option type.yes sets this bit both in the normal mask and in the test mask.

1 shift 20	This bit is irrelevant in the test mask. In the normal mask it is set by the assembly option names.yes and causes the assembler to print the blocks and the names. This printing is made independently of the occurences of the delimiters ϕ. and œ.
1 shift 19	This bit causes the assembler to print message comments on the current output medium. In the normal mask this bit is set by the assembly option message.yes.
1 shift 18	This bit is irrelevant in the test mask. In the normal mask it is set by the assembly option warning.yes and causes the assembler to print warning messages on the current output medium. This printing is executed independently of the occurences of the delimiters ϕ. and œ.
1 shift 17	This bit is irrelevant in the test mask. In the normal mask it is set to one, if a result file is specified in the program call, otherwise it is set to zero; it is further set, if the program discovers an error on the result file during pass 2. The bit prevents the assembly from generating output during pass 2.
1 shift 16	This bit is irrelevant in the test mask. In the normal mask it is set by the assembly option print.yes and causes the assembler to print the binary program.

1 shift 15	The bit causes the assembler to use format c for the arithmetic and branch instructions, for which no format is specified. In the normal mask it is set by the assembly option formatc.yes.
1 shift 0	This bit causes the assembler to print internal control information during the assembly.

If no assembly options are defined in the parameter list, the following initial values are used:

type.no
list.no
names.no
message.yes
warning.yes
stack.50
work.50
test.no
print.no
formatc.no

During assembly, certain control and error messages will be printed on the current output medium. Most of these include the line number, the load address k, and the octal value of k; in the following these numbers are denoted by <I>. When counting line numbers, lines only containing invisual characters are not counted.

The following are normal control messages:

<I>message<message comment>

The delimiter m. causes printing of a message comment.

<I> <block level>begin<name of the block>

The assembly option names.yes causes the assembler to print the name of the block.

If the Slam program contains a comment in the line, in which the block head (i.e. the b.) is placed, then this comment is the name of the block. Otherwise the name of the block is empty.

<I> <block level>end<name of the block>

<id list>

The assembly option names.yes causes the assembler to print the name of the block, and the last values assigned to the identifiers by <identifier>: or <identifier>=<expression>, when the corresponding e. is met.

<I>prog end,a:<no a>,b:<no b>,c:<no c>

The assembler has terminated the assembly normally and returns to the file processor. This message will occur, even if the assembler has not generated output because of errors at the output file.

The numbers <no a>, <no b>, and <no c> mention the number of instructions corresponding to the three formats. If formatc.yes is specified, the numbers give the number of instructions that could fit into these formats.

During assembly the following warnings can appear:

<I> <value>illegal

Input of an illegal character. The character is interpreted as a space. In listings it will appear as the sign "?".

<I>value

An expression results in a value which is too large to be held in the address part of an instruction or in a byte. The assembler will only use the last bits of the expression, whereas the other bits are ignored.

<I> <repetitions>repetition

A repetition expression with a value less than 1.

During assembly error messages are printed in the following cases:

<I>syntax

Illegal structure of delimiters and operands. This error message is the only one that causes the assembler to skip. The assembler skips everything up to the next expected terminator, which can be comma, nl, ff, em, or semicolon.

<I>missing i after a-format

The modifier i is missing in an a-format, where the <x-register> equals zero. The assembler simulates that it has read an i-modifier.

<I> <identifier>declaration

An identifier is declared twice in the same block head. The assembler uses the identifier which has the larger index.

<I> <identifier>index

An identifier has an index greater than 127. The assembler uses the index modulo 128.

<I> <identifier>undeclared

An identifier is used without being declared.

<I> <identifier>definition

An identifier that has been defined elsewhere by a label or an assignment is redefined by a label, which changes the value of the identifier.

<I> <identifier>undefined

An undefined identifier is used in an expression violating the rules as given in section 2.5.3.

end source<number>ends missing

the source list is exhausted, or the assembler has read an EM character on the current input medium, before it has met the correct number of e. The assembler will generate the number of missing e.

condition counter<missing z.>

The program misses some z. These are generated by the assembler. The end source error message will always follow this message.

program too big

The binary output exceeds the capacity of the result file. Assembly continues without output.

The following situations cause immediate termination of assembly:

create area

The assembler fails in creating the internal stack area or work area.

no text

A source file contains a character with a value > 127.

stack

An expression uses too many parantheses within each other. This alarm message can never occur, if the expression does not exceed 100 character positions.

no program

The last source file is emptied before the assembler reads a b followed by a point.

params

The assembler is called by an improper file processor call.

work area

Hard error at the working area.

stack area

Hard error at the stack area.

output document

Hard error at the output document.

connect file <sourceno>

A file cannot be connected. <sourceno> equal 0 means the output document, and <sourceno> greater than 0 means a source file.

APPENDIX B

OPERATION OF THE ASSEMBLER ON CYBER 73 B

Introduction. B.1

This appendix describes the features and the operation of the SLAM Fortran version.

First the differences from chapters 2, next operation of the assembler.

Changes to chapter 2. B.2

Syntax. B.2.1.1

<sp>::= <blank>

<HT> does not exist

<newline>::= <end-of-card>!<end-of-line>

These codes are generated internally by the system.

Semantics. B.2.1.2

Slam accepts the CDC 63-character set, as described in section B.2.1.1. and CDC Manuals appendix A.

Only the first 72 characters on a card or line is processed.

Positions beyond 72 may be used for sequencing.

Display code	Character	Display code	Character
53	illegal	70	Ø
60	illegal	71	Å
61	illegal	72	<
62	!	73	>
64	illegal	74	illegal
65	illegal	75	illegal
66	Æ	76	illegal
67	&	77	¡

<input control> has no meaning. B.2.2.3

Use COPY to merge files before assembly.

Semantics. B.2.3.2

Slam accepts radix numbers and unsigned integers.

Unsigned integers are decimal numbers.

Radix numbers are expressed as a radix followed by a digitstring for example 2.11011 or 8.4357.

The radix must not be greater than 9.

The number is converted digit by digit as follows:

number:= radix * number + digit

The CDC FORTRAN limits integer values to 14 decimal digits. In a RC 3500 the range is - 32768 to 32767.

Introduction. B.3

This chapter decribes the features and the operation of the Slam assembler. The assembler is programmed in FORTRAN EXTENDED for the CYBER 73 and may be run on this computer under SCOPE 3.4.

The assembler takes its input from a textfile created by EDITOR, an UPDATE COMPILE file or direct from INPUT. It outputs a binary program on a sequentiel file and the listings on the OUTPUT file.

The assembler is written as a two-pass assembler and uses therefore a sequentiel file as a working area for the internal output. Further it uses a random file holding information about the blocks and identifiers.

Size and Speed.

The program resides in the permant file SLAMCY,ID = HJ, MR = 1.

Filelength _____ pru's.

The program needs CM 36000 to be executed.

The speed has empiricially been found to about _____ characters per CP second or _____ lines per CP second.

Calling the assembler.

First fetch the assembler by

ATTACH (SLAM, SLAMCY, ID=HJ, MR=1)

The call has the following format

SLAM (<source file>,<input>,<object file>,<output>)

<source file>	contains the program text.
SOURCE	is default lfn name.
<input>	contains the parameter list to the assembler.
INPUT	is defaul t lfn name.
<object file>	is the lfn of the generated pseudo binary code.
OBJECT	is default lfn name.
<output>	the file to which listings are directed.
OUTPUT	is default lfn name.

The assembler can only input the source-program from one file. Merging of files into one file should be done by EDITOR, UPDATE, or COPY.

Format of pseudo binary programs. B.6

SCOPE files. B.6.1

Each CYBER 60 bit word contains 3 RC 3500 machine words.

Each 20 bit part consists of 4 zeroes and 16 bits which is the RC 3500 word

59 0

0000<word 1>0000<word 2>0000<word 3>

After the last assembled RC 3500 word the assembler places 1, 2, or 3 dummy words each containing 20 ones to signal end of program.

The <object file> is written by unformatted WRITE with a length of 127 CYBER words.

TTY papertape for autoload. B.6.2

The program in the permanent file

SLAMCY, ID=HJ, CY=3, MR=1.

should be used on a TTY papertape puncher.

The format is described in (AUTOLOAD of RC 3500).

Papertape punch at central site. B.6.3

Use the program in SLAMCY, ID=HJ, CY=6, MR=1

to punch papertapes at the central site. Format as for B.6.2.

Remote autoload. B.6.4

This feature is not implemented yet.

The following is only the differences from A.5.

stack.<number>	has no effect
work.<number>	has no effect
type.yes	has no effect
type.no	has no effect
end	signals end of parameter list.

The parametres are given to SLAM via the <input file>
(2. parameter in the call of SLAM).
One parameter per card or line.
Last parameter must be END in coulumb 1-3.