

ksm

RCSL No: 52-AA1039
Edition: April, 1981
Author: Flemming Hermansen
Henning Jakobsen
Volker Raab
Harald Villemoes
Bo Bagger Laursen

Title:

RC3500 ASSEMBLER MANUAL
SLAM5

Keywords:

RC3500 Assembler Manual, RC8000, SLAM5, RC3500, RC3503.

Abstract:

This paper is a description of SLAM5 - the symbolic assembler language for the RC3500/RC3503 computer - designed to operate under RC8000 file processor.

This manual replaces RCSL: 52-AA929.

(54 printed pages).

**Copyright © 1981, A/S Regnecentralen af 1979
RC Computer A/S**

Printed by A/S Regnecentralen af 1979, Copenhagen

Users of this manual are cautioned that the specifications contained herein are subject to change by RC at any time without prior notice. RC is not responsible for typographical or arithmetic errors which may appear in this manual and shall not be responsible for any damages caused by reliance on any of the materials presented.

CONTENTS	PAGE
0. FOREWORD	1
1. GENERAL DESCRIPTION	3
1.1 Language Features	3
1.2 Programming Examples	4
2. BASIC ELEMENTS OF SLAM5	8
2.1 Character Set	8
2.1.1 Syntax	8
2.1.2 Semantics	8
2.1.3 Letters, Spaces, and Horizontal Tabulation .	8
2.2 Directives and Comments	9
2.2.1 Syntax	9
2.2.2 Semantics	9
2.2.3 Input Control	9
2.2.4 Testoutput Control	10
2.2.5 Line Comments	10
2.2.6 Error Messages about "g." and "f."	11
2.3 Numbers	11
2.3.1 Syntax	11
2.3.2 Semantics	11
2.4 Identifiers	12
2.4.1 Syntax	12
2.4.2 Semantics	12
2.4.3 Load Address	13
2.4.4 Defined and Undefined Identifiers	13
2.4.5 Examples	13
2.5 Expressions	13
2.5.1 Syntax	13
2.5.2 Semantics	13
2.5.3 Defined and Undefined Expressions	14
2.5.4 Operators	14
2.5.5 Precedence of Operators	15
2.5.6 Examples	15

CONTENTS (continued)	PAGE
2.6 Instructions	16
2.6.1 Syntax	16
2.6.2 Operation Parts	18
2.6.3 Address Parts	19
2.6.4 RC3500 Mode	19
2.6.5 Format	20
2.7 Datawords	20
2.7.1 Syntax	20
2.7.2 Semantics	20
2.7.3 Examples	21
2.8 Text Strings	21
2.8.1 Syntax	21
2.8.2 Semantics	21
2.8.3 Examples	21
3. STRUCTURE OF SLAM5 PROGRAMS	22
3.1 Datawordmode	22
3.1.1 Syntax	22
3.1.2 Semantics	22
3.2 Labels	23
3.2.1 Syntax	23
3.2.2 Semantics	23
3.2.3 Examples	23
3.3 Assignment Statements	24
3.3.1 Syntax	24
3.3.2 Semantics	24
3.3.3 Load Address Control	24
3.3.4 Examples	24
3.4 Repetition Statements	25
3.4.1 Syntax	25
3.4.2 Semantics	25
3.4.3 Examples	25
3.5 Conditional Assembly	26
3.5.1 Syntax	26
3.5.2 Semantics	26
3.5.3 Examples	26

CONTENTS (continued)	PAGE
3.6 Programs and Blocks	27
3.6.1 Syntax	27
3.6.2 Semantics	27
3.6.3 Declarations	28
3.6.4 Program Comment	28
4. PUNCH16	29
4.1 Program Description	29
4.2 PUNCH16 Program Call	29
4.2.1 Explanation of Terms Used in Call	29
4.3 Error Messages	31
4.4 Normal Messages	31

APPENDICES:

A. OPERATION OF THE ASSEMBLER ON RC8000	33
A.1 Introduction	33
A.2 Size and Speed	33
A.3 Calling the Assembler	33
A.4 Format of Binary Program	34
A.5 Assembly Options	35
A.6 Assembly Messages	40
A.7 Examples of Program Calls	45

This manual is a description of Slam5 - the symbolic language for the RC3500/3503. Slam5 is a pure extension of Slam3.

Chapter 1 contains a general description of the language. Chapter 2 introduces the basic constituents of Slam5, which are: directives, numbers, identifiers, expressions, datawords, and instructions. The overall structure of Slam5 programs is defined in chapter 3. The syntax of the language is described by means of Backus notation.

Chapter 4 contains a description of the Punch16 program which is used to change the output from the assembler so that it can be loaded and executed on the RC3500/RC3503.

The Slam5 language was defined and implemented by the authors. The manual is concluded by descriptions of the operation of the Slam5 assembler.

Appendix A: Operation on RC8000.

GENERAL DESCRIPTION

1

Language Features

1.1

In Slam5 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 parentheses 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, Slam5 has a block structure similar to Algol60: 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, Slam5 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.

Programming Examples

1.2

The following examples illustrate the overall structure of Slam5 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 cpu-time required
;depends on op 2, but
;is about 0.5 ms per 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 op 2 = 1 then
      addcw1 i          ;result := result + op 1

a2:    0              ;op 1 is stored here

a3:    asr w1        1        ;these shifts will
      ror w2          1        ;act as multiplying op 2 with 2
      sub w3i         1        ;count := count - 1
      bgw w3i         0,a1     ;if count > 0 then go to loop
      bsz w2i         1,a4     ;if op 2 < 0 then
      sub w1           a2      ;result := result - op 1

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 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 w3 i 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 cwl 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 other instructions are given:

```
ldr awl in x3+b3
bew cwl in 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 wli-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 examples illustrate the assembly of datawords:

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


BASIC ELEMENTS OF SLAM5

2

Character Set

2.1

Syntax

2.1.1

$\langle \text{assembly character} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{letter} \rangle \mid \langle \text{blind} \rangle \mid \langle \text{sp} \rangle \mid \langle \text{new line} \rangle \mid$
 $\langle \text{special character} \rangle$
 $\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
 $\langle \text{letter} \rangle ::= a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid$
 $s \mid t \mid u \mid v \mid w \mid x \mid y \mid z \mid \text{æ} \mid \phi \mid \text{å}$
 $A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M \mid N \mid O \mid P \mid$
 $Q \mid R \mid S \mid T \mid U \mid V \mid W \mid X \mid Y \mid Z \mid \text{Æ} \mid \Phi \mid \text{Å}$
 $\langle \text{sp} \rangle ::= \langle \text{SP} \rangle \mid \langle \text{HT} \rangle$
 $\langle \text{new line} \rangle ::= \langle \text{NL} \rangle \mid \langle \text{FF} \rangle \mid \langle \text{EM} \rangle$
 $\langle \text{special character} \rangle ::= \& \mid (\mid) \mid * \mid + \mid , \mid - \mid . \mid / \mid : \mid ; \mid < \mid = \mid > \mid ! \mid \#$
 $\langle \text{blind} \rangle ::= \langle \text{NUL} \rangle \mid \langle \text{DEL} \rangle$

Semantics

2.1.2

Slam5 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

Slam5 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 $\langle \text{w-register} \rangle$, $\langle \text{x-register} \rangle$, $\langle \text{letters followed by point} \rangle$).

In all places in Slam5 programs, where there may occur for example w., p. may also occur.

A p. has no effect on the binary code, but during the listing an FF will be printed after the full stop.

If p. appears in the comments, FF is not printed, however.

N.B.: It should be noted that a p. does not refer to the line number in Slam5. The line which comes before and the line which comes after p. are considered as belonging to the same line.

Directives and Comments

2.2

Syntax

2.2.1

<directive> ::= <input control> | <testoutput directive> |
 <global repeat control> | <listing control> |
 m. <line comment> <newline> | <separator>

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

<testoutput control> ::= ϕ . | æ.

<global repeat control> ::= g. <expression> | f.

<listing control> ::= l. | d.

<separator> ::= , | <endline>

<endline> ::= <newline> | ; <line comment> <newline> |

<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 the surrounding system (see app. A.5).

- d. - stops any further listing of the sourcetext, if the list parameter is specified as ON or OFF.
- l. - resumes the listing of the source text, if the list parameter is specified as ON or OFF.

- g. - global repetition begin.

g. will be followed by an arithmetic print-out called the repetition factor. If the repetition factor is larger than 1, the translation of the source text between g. and f. is repeated this many times. The translation of the source text between g. and f. is skipped if the repetition factor is zero or negative.

No change of sourcefile may occur between g. and f. Nesting is not allowed.

- f. - global repetition end.

OBS: global repetition cannot be nested.

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 α . This facility is used to debug the assembler. For further information see section 4.5.

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.

Error Messages about "g." and "f."

2.2.6

"end repetition no: number" is not an error, but is output when the program is listed.

"global repetition" means the expression following "g." is illegal, i.e. negative or greater than repeat max. (see Assembly Options, A.5), or "f." is missing between two "g."s.

Numbers

2.3

Syntax

2.3.1

$\langle \text{number} \rangle ::= \langle \text{unsigned integer} \rangle \mid \langle \text{radix number} \rangle$
 $\langle \text{radix number} \rangle ::= \langle \text{radix} \rangle \langle \text{unsigned integer} \rangle$
 $\langle \text{radix} \rangle ::= \langle \text{unsigned integer} \rangle.$
 $\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle$

Semantics

2.3.2

Slam5 accepts radix numbers and unsigned integers. Unsigned integers are decimal numbers. Radix numbers are expressed as a radix followed by a digitstring, for example 2.110111 (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:

$$\text{number} := \text{radix} * \text{number} + \text{digit}$$

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 8000, 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

$\langle \text{identifier} \rangle ::= \langle \text{declared identifier} \rangle \mid \langle \text{load address} \rangle \mid \langle \text{index identifier} \rangle$

$\langle \text{declared identifier} \rangle ::= \langle \text{identifier letter} \rangle \langle \text{index} \rangle$

$\langle \text{index identifier} \rangle ::= \langle \text{identifier letter} \rangle \# \langle \text{declared identifier} \rangle$

$\langle \text{index} \rangle ::= \langle \text{unsigned integer} \rangle$

$\langle \text{identifier letter} \rangle ::= a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j$

$\langle \text{load address} \rangle ::= k$

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.

An index identifier is an identifier whose index is the value of the declared identifier following the number sign. The declared identifier must be defined.

The number sign will in the optional listing be replaced by the actual index, i.e. the value of the declared identifier.

Example:

The following two programs will generate equal code:

b. a10,b10

a1=7

b7=5

b # a1

e.

b. a10,b10

a1=7

b7=5

b7

e.

The expression b# a1 above will be listed as b7a1.

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 as 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 d25 h123

Expressions

2.5

Syntax

2.5.1

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

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

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

<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 occurring 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 preceding 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 preceding text, the last definition of the identifier must have been an assignment-statement 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 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 permissible 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 parentheses "(" 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

ϕ after a bracket means that each item inside the bracket may occur at most one time.

Syntax

2.6.1

<instruction> ::=

<arithm. instruction> <endline> |
 <branch instruction> <endline> |
 <shift instruction> <endline> |
 <interlevel instruction> <endline> |
 <I/O instruction> <endline> |
 <move instruction> <end line>

<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>

<move instruction> ::=

<move opcode>

$$\langle \text{format } 1/2 \text{ mod.} \rangle ::= \left\{ \begin{array}{l} \langle \text{format} \rangle \\ \langle \text{w-register} \rangle \\ \langle \text{immediate} \rangle \\ \langle \text{next/not} \rangle \\ \langle \text{x-register} \rangle \end{array} \right\} \phi$$

$$\langle \text{interlevel mod.} \rangle ::= \left\{ \begin{array}{l} \langle \text{format} \rangle \\ \langle \text{w-register} \rangle \\ \langle \text{eds} \rangle \\ \langle \text{x-register} \rangle \end{array} \right\} \phi$$

$$\langle \text{shift mod.} \rangle ::= \left\{ \langle \text{w-register} \rangle \right\} \phi$$

$$\langle \text{V/O mod.} \rangle ::= \left\{ \begin{array}{l} \langle \text{w-register} \rangle \\ \langle \text{x-register} \rangle \end{array} \right\} \phi$$



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:

<u>Type of Address Part</u>	<u>Range</u>
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 NB'
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 instructions)	- 32768 32767

RC 3500 Mode

2.6.4

When the assembler is called with the parameter RC3500.yes, it will run in a special mode where the mow instruction is illegal and <format> modification is illegal for interlevel instructions.

Format

2.6.5

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 for 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), then the 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)

Text Strings

2.8

Syntax

2.8.1

<string> ::= <string of characters not containing newline> <newline>

Semantics

2.8.2

- s. - the rest of the line, beginning with the first character which is not a space is interpreted as an ISO-text string. On the basis of Slam5's method of reading numbers, a number can only be given with difficulty. A number is therefore used to give a temporary ISO-character, the comma is implicit, the semicolon, nl, em, ff, cr are all counted as end of line.

Examples

2.8.3

s. 50,48 abc ;<20 abc:>

STRUCTURE OF SLAM5 PROGRAMS

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. In addition, if the load address is odd, the delimiter w. 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

Syntax

3.3.1

$\langle \text{assignment} \rangle ::= \langle \text{identifier} \rangle = \langle \text{expression} \rangle \langle \text{separator} \rangle$

Semantics

3.3.2

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.

Load Address Control

3.3.3

The load address k can be changed explicitly by an assignment statement. This affects 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 Slam5 language.

Examples

3.3.4

```

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 Slam5 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 3.7) 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

$$\langle \text{program} \rangle ::= \langle \text{program comment} \rangle \langle \text{block} \rangle$$

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

$$\langle \text{block} \rangle ::= \text{b.} \langle \text{declarations} \rangle \langle \text{endline} \rangle \langle \text{statements} \rangle \text{e.} \mid \text{b.} \langle \text{endline} \rangle \langle \text{statements} \rangle \text{e.}$$
$$\langle \text{declarations} \rangle ::= \langle \text{declared identifier} \rangle \mid \langle \text{declared identifier} \rangle, \langle \text{declarations} \rangle$$

```

<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 preceded 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.

PUNCH 16

4

Program Description

4.1

PUNCH16 is an RC 8000 utility program which is used in connection with the RC 3500 controller and assembler. The RC 3500 assembler running on an RC 8000 does not produce a loadable and executable RC 3500 object code.

Therefore, the output from the assembler must be changed before it can be loaded and executed on the RC 3500.

PUNCH16 Program Call

4.2

The program call is as follows:

$$\left\{ \begin{array}{l} \text{outfile} = \end{array} \right\}_{0}^{1} \text{punch16} \left\{ \begin{array}{l} \langle s \rangle \langle \text{modetype} \rangle \\ \langle s \rangle \langle \text{inputfile} \rangle \end{array} \right\}_{0}^{\infty}$$

$$\text{where } \langle \text{modetype} \rangle ::= \left\{ \begin{array}{l} \text{mode.} \left\{ \begin{array}{l} 8 \\ \text{auto} \\ \text{boot} \end{array} \right\} \end{array} \right\}$$

Explanation of Terms Used in Call

4.2.1

Outfile is the name for an RC 8000 backing storage file which will contain the loadable version of the RC 3500 program.

Before the Punch16 call is written "<outfile>". This results in the output from Punch16 being placed in "<outfile>" instead of on paper tape.

If "<outfile>" does not exist in the catalogue, Punch16 will set up a temporary area on backing-storage with the name in question.

Punch16 can handle backing-storage files as well as paper tape files. The autoloading formats can be punched with the help of edit or copy. They only contain the characters (decimal values): 0, 64-96, 3 and in conclusion: 25 (together with 0 with respect to the punch. These 0's are not copied, but instead some new ones will be printed).

It should also be noted that the autoloader on RC 3500 ignores the parity bits.

mode denotes the mode used for loading.

8 - binary eight bits information for the RC 3500.

Note: A special load program must be inserted (by hand) if mode 8 is to be used.

auto - used for autoloading by means of autoloading prom.

Format: Each 8 bit byte contains 4 data bits and 4 control bits. Each RC 3500 word consists of 4 groups of 4 data bits while the control part is skipped during reading.

The format is used to make the data appear as ASCII characters.

boot - used to load the RC 3500 via an FPA channel from the RC 8000. The output contains binary information for the RC 3500, enclosed within control information for the RC 8000 program autoloading which performs the loading of the RC 3500 together with the autoloading prom and an RC 3500 bootloader, called fpabootload.

The binary version of the bootloader, called fpabootbin, which is produced by the assembler, must be used as the first inputfile.

<inputfile> is an RC 8000 backing-storage file which has been output by the RC 3500 assembler.

Error Messages

4.3

params - wrong parameters in call.

connect file <inputfileno> - the inputfile could not be found.

Normal Messages

4.4

Each time a new inputfile is connected the following message appears on the primary output medium:

sourcefile : <inputfile name>
segm : <segment number>
bytes : <byte number>

The segment number and bytes describe approximately the current position in the outputfile.

Introduction

A.1

This appendix describes the features and the operation of the Slam5 assembler. The assembler is programmed in Algol 6 on the RC 8000 computer and must be run on this computer.

The assembler takes its input from 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 Algol8, the speed depends very much on the process size. The number of locations used for the stack is independent of the Slam5 program.

Calling the Assembler

A.3

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

<result> Slam5 <parameter list>

The <result> can be empty or 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>.on	
or	<assembly option>.off	
or	<assembly option>	$\left\{ \begin{array}{l} \text{.<number>} \\ 1 \end{array} \right\}^*$

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 in the Slam5 call, Slam5 translates the program correctly, but the assembler does not output binary code. The size of the backing store file is not decreased by the assembler.

If the output file does not exist in the catalogue, Slam5 will establish a temporary area on backing storage with the name in question.

Format of Binary Program

A.4

Each RC 3500 machine word is stored in an RC 8000 machine word. As the RC 8000 word length is 24 bits, the RC 3500 word is written in bit positions 8 through 23, and bits 0 through 7 contain a flag indicating status of the datum (0 = normal datum, 1 = loader directive, 255 = EOF).

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 Algol8 procedure outrec6 with a share length of 128 reals.

Although the assembler can generate output on every document, the program Punch16 should be used to generate loadable and executable RC 3500 object code, because output generated by the assembler cannot be read by the RC 3500. See section 4, Punch16, for further explanation.

Assembly Options

A.5

The following is a list of assembly modes that can be chosen by explicit parameters to Slam5:

Assembly options followed by ".yes" or ".no" give the initial values of the options.

Assembly options followed by ".on" or ".off" are valid for all following files, until another appearance of the option followed by ".on" or ".off".

The options:

type, list, message, formatc, condlist, test

may be followed by ".on", ".off", ".yes", and ".no", whereas the options:

names, warning, print, blocks, xref, rc3500

may only be followed by ".yes" and ".no".

The options:

$$\text{test } \left\{ .\langle \text{number} \rangle \right\}_1^*,$$

repeat.<number>, errors.<number>

are valid for all following files until redefined by another appearance of the option.

type.yes	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.
type.no	
list.yes	Defines whether the assembler should list the source text (including the "typed" input) on the current output medium.
list.no	
xref.yes	Defines whether the assembler should print a cross reference listing of definition and usage of all names after the listing.
xref.no	
rc3500.yes	Defines whether the assembler should recognize the full RC3503 instruction set or only the RC 3500 part of it.
rc3500.no	
names.yes	Defines whether the assembler should print the blocks with the identifier values on the current output medium.
names.no	
message.yes	Defines whether the delimiter m. should cause the assembler to print comments on the current output medium.
message.no	

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 the purpose of debugging a Slam5 program the formatc option has the advantage that you can correct the instruction manually without having to assemble the program.
condlist.yes condlist.no	Defines whether the directives l. and d. should be allowed to switch the listing on and off. If condlist is true, then the "list" option defines whether the assembler should start in the listmode or not. Also, if condlist is true, listing of source text that is skipped using c- or g-directives will be suppressed.
blocks.yes	Prints out during pass 2 a list of all blocks with block head and block finish. The print-out is an "interpretation" of the names.yes print-out.
blocks.no	Nothing, initialization values.
warning.yes warning.no	Defines whether the assembler should print warning messages on the current output medium.

test.yes

test.no

test

$\left\{ \begin{array}{l} \text{test.no} \\ \text{test} \end{array} \right\} \left\{ \begin{array}{l} \text{.<number>} \\ \text{.<number>} \end{array} \right\} \left\{ \begin{array}{l} * \\ 1 \end{array} \right\}$

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 $\left\{ \begin{array}{l} \text{.<number>} \\ \text{.<number>} \end{array} \right\} \left\{ \begin{array}{l} * \\ 1 \end{array} \right\}$ sets the bit "1 shift number" equal to 1.

When the assembler reads the delimiter $\phi.$, the normal mask is exchanged for the test mask; when it reads the delimiter α , 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. causes 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 occurrences of the delimiters $\phi.$ and α .

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 occurrences of the delimiters `ø.` and `æ.`
- 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 programm 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 This 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 14 This bit is irrelevant in the test mask. In the normal mask it is set by the parameter `blocks.yes`.
- 1 shift 13 This bit is irrelevant in the test mask. In the normal mask it is set by the parameter `rc3500.yes`.
- 1 shift 12 This bit is irrelevant in the test mask. In the normal mask it is set by the parameter `xref.yes`.
- 1 shift 0 This bit causes the assembler to print internal control information during the assembly.

repeat.<number>	With this option the maximum number for the expression following "g" may be redefined.
errors.<number>	With this option the maximum numbers of errors accepted, before assembly is terminated, may be changed to the number specified.

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

```

type.no
list.no
xref.no
rc3500.no
names.no
message.yes
warning.yes
test.no
print.no
formatc.no
condlist.no
block.no
repeat.20
errors.100

```

Assembly Messages

A.6

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 <*l*>. When counting line numbers, lines only containing invisual characters are not counted.

The following are normal control messages:

<l> message <message comment>

The delimiter m. causes printing of a message comment.

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

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

If the Slam5 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.

<l> <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.

<l> 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 parentheses 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.

xref area

Hard error at the xref work area.

connect file <sourceno>

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

too many errors

Appears if the number of errors is greater than the maximum number of errors.

Assembly is terminated immediately.

Examples of Program Calls

A.7

The following string of RC 8000 file process programs will generate a paper tape which the RC 3500 could be autoloaded with.

```
rc3500bin=slam5 rc3500text
tpe=punch16 mode.auto rc3500bin
```

Explanation:

rc3500text	is the file containing the RC 3500 program text.
rc3500bin	is the name of the temporary file used to hold the output from the Slam5 assembler.
tpe	is the name of the iso-even-parity paper tape.

The RC 3500 front-end could be loaded from an RC 8000 using the normal fpa-line connection between the RC 3500 and the RC 8000. This kind of loading is called bootloading.

```
rc3500bin=slam5 rc3500text
rc3500boot=punch16 mode.boot fpabootbin rc3500bin
main3500=autoload rc3500boot
```

Explanation:

rc3500bin	and rc3500text have the same meaning as described above.
rc3500boot	is the name of a temporary file containing the bootload version of the RC 3500 program.
mode.boot	is an option to the Punch16 program indicating that a bootload version of the binary program is wanted.
fpabootbin	is a binary program to be used when the RC 3500 should be bootloaded. The program is standard like Slam5 and Punch16.

main3500 is the name of the RC 8000 external process communicating with the RC 3500.

autoload is the name of an RC 8000 program capable of loading another computer regarded as a front-end computer by the RC 8000.

RETURN LETTER

Title: RC3500 Assembler Manual, SLAM5

RCSL No.: 52-AA1039

A/S Regnecentralen af 1979/RC Computer A/S maintains a continual effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback, your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability:

Do you find errors in this manual? If so, specify by page.

How can this manual be improved?

Other comments?

Name: _____ Title: _____

Company: _____

Address: _____

Date: _____

Thank you

..... Fold here

..... Do not tear - Fold here and staple

Affix
postage
here

 **REGNECENTRALEN**
af 1979

Information Department
Lautrupbjerg 1
DK-2750 Ballerup
Denmark