TEKNISK SERVICE, Glostrup, 27th September, 1971. PEH/BECA.

Dear Sirs,

Please find enclosed a new set of testprograms which extends the number of hardware tests for the RC 4000.

The set consists of

CPU-test	1 description	51-VB1172
	3 paper tapes	31-T15
		31-T16
		31-T17
	1 test cable	type 27/50
Loader 2	2 descriptions	31-A29
		31-D14
	2 paper tapes	31-T19
		31-T20
Disc test	1 description	31-D17
	1 paper tape	31-T30

It is strongly recommended that you run the testprograms when convenient, so you get familiar to them before an error situation arises.

Kind regards,
A/S REGNECENTRALEN

Per Hansen

RCSL: 51-VB1172
Author: Allan Giese
Edited: March 1971

RC 4000 DIAGNOSTIC PROGRAM

FOR CENTRAL PROCESSOR

PROGRAM DESCRIPTION

KEY: Technical Manual, Fault Detection and Isolation

Abstract: This manual describes the loading of the diagnostic program and explains how error messages are used for fault detection and location.

A/S REGNECENTRALEN
Falkoneralle 1
DK 2000 Copenhagen F

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

0. INTRODUCTION

This section explains the program loading, the operator commands and the messages issued by the system. An introduction to the documentation for each subprogram is also given.

0.1. Automatic Program Loading.

The program loading proceeds as follows:

1. Set the computer into Reset Mode. This state is signalled by a red light in the indicator for Reset Mode.
2. Insert the program LOADER FOR CPU TESTPROGRAM into the RC 2000 paper tape reader and press the RESET button belonging to the reader.
3. Press only once the AUTOLOAD key whereafter the computer reads and executes the Loader, and the computer turns then into the Reset state.
4. Insert the program CPU TESTPROGRAM into the paper tape reader and press the RESET button belonging to the reader.
5. Activate the START key and the Loader takes care of loading of the testprogram. After loading the computer turns into the reset state and register W3 contains the address of the first free location after the testprogram.
   If the operator wants to verify the loading of the CPU TESTPROGRAM, which should normally be the case, he continues with step 6, otherwise step 6 is skipped.
6. Insert the program CHECK LOADING OF CPU TESTPROGRAM into the reader and press the AUTOLOAD key. The program writes either che
   checksum ok or checksum incorrect
   depending on whether the testprogram is successfully loaded or not. Before proceeding to the next step the RESET key must be activated.
7. The CPU TESTPROGRAM is started by the START key and the program responds by printing
   test end
   number of runs =
   The operator commands may now be used confer Section 0.5.

FILE PROCESSOR UTILITY PROGRAM: CHECKIO

General

Checkio may supervise all actions on a particular document. That is performed in the following way: Assume that checkio is executed in a job process called dev. Then all messages sent to dev are passed on by checkio to the document supervised. The answer from the document is passed back to the original sender process, which seems to be handling the document in the normal way.

Checkio can easily print all messages and answers as they are passed on, and you can later find out about parity errors from the document, rereading, etc.

Call

The process dev must be created with a protection register which enables it to modify the contents of all other processes.

Checkio must be called with one parameter specifying the name of the document to be supervised. The messages and answers are printed on the current output of dev. A console communication to start dev may look like this:

```
to s
new dev size 5000 pr 1 2 3 4 5 6 7 pk 1 run

to dev
o lp            ; print on the line printer
checkio t6      ; check the document t6
```

You will then see nothing from dev until some messages are sent to it, for instance from another job started like this:

```
to s
new sl run

to sl
s=set mto dev   ; edit to the 'magnetic tape' dev,
s=edit tre      ; which acts as the magnetic tape t6.
```

Dev will now start listing the communication on the line printer. When you have finished using dev for supervising of t6, you can proceed like this:

0.2. Manual Loading from Paper Tape Reader.

The manual loading is used in cases where the CPU TESTPROGRAM cannot be loaded by the automatic way or the error reactions from the program are suspicious-looking. The reasons for the manual start-up procedure to be superior to the automatic are that fewer parts of the computer are necessary to operate satisfactory for loading and that the testprogram is initiated in a different way.

The loading proceeds as follows:

1. Set the computer into Reset Mode. This state is signalled by a red light in the indicator for Reset Mode.
2. Insert the program CPU TESTPROGRAM into the paper tape reader and press the RESET button belonging to the reader.
3. Insert the Slang program below into the w-registers.

```
w0:     aw      x3+0     ; numeric code  0
w1:     al  w3  x3+2     ;   -       -   11
w2:     jl          0    ;   -       -   13
w3:                 8
```

   Set the Protection Register, PR:= b1111 1111
   and the Function Register, FR(5):= 0;

4. Set the Micro Address Register, MAR:= x16y4 and press the Single Micro Instruction pushbutton. The paper tape is read by

   setting the Instruction Counter, IC:= 0;
   setting MAR:= x4y0;
   activation of the pushbutton MAR COMPUTER CONTROLLED;
   activation of the pushbutton CONTINUE.

5. The CPU TESTPROGRAM is initiated by setting

   IC con 0:= 128+64+ 16+8+4
   MAR:= x4y0; EX:= 0;
   w0:= 49; w1:= 50; w2:= 52; w3:= 56

   after activation of the pushbuttons

   MAR COMPUTER CONTROLLED
   CONTINUE (SINGLE INSTRUCTION repeatly if this is wanted)

   the testprogram writes if no errors are found

   12481248/3Y1248
   00 000000 000000 000000 000000
   test end
   number of runs=

   The operator commands may now be used confer Section 0.5.

0.3. Manual Loading from High-Speed Device.

The manual loading explained in the previous section requires that quite a number of micro commands and jump conditions, as well as parts of the arithmetic unit do function properly. In order to surmount this obstacle the program may be loaded from a high-speed device, for example a magnetic tape, a drum, or a disc. The program must, of course, have been stored on the storage medium in question at a time where the computer was faultless or by another computer. The requirements for high-speed loading are limited to error-free operation of the high-speed data channel the part of the store controller that handles high-speed data transfer, and the necessary I/O instructions for initialization of data transfer. If, for example, the CPU TESTPROGRAM is stored as the first block on an RC 747 or RC 749 magnetic tape station then the loading proceeds as follows:

1. Set the computer into Reset Mode. This state is signalled by a red light in the indicator for Reset Mode. Set the Protection Register, PR:= b1111 1111.
2. Execute the instruction (inserted in the W-registers)

   ```
   io  wx  dev no < 6 + 5   ; where wx = last storage addr, e.g.
                            ; last addr of core store
   ```

3. Execute the instruction

   ```
   io  wx  dev no < 6 + 13  ; where wx = first storage addr = 8
   ```

   The CPU TESTPROGRAM is now loaded into core store from location 8 and upwards.
4. Execute the instruction

   ```
   io  wx  dev no < 6 + 0   ; wx = status word
   ```

5. Execute the instruction

   ```
   iio  wx  dev no < 6 + 4  ; wx = block size
   ```

   The block size is determined by executing the above mentioned steps when the computer is error-free.
6. The CPU TESTPROGRAM is initiated as explained in step 5 of Section 0.2.

0.4. Test of Interruption System.

A complete test of the interruption system requires a possibility that enables the testprogram to set bits in the interrupt register. This is implemented by removal of one of the external I/O bus cables plus the wired interrupt plug 1021 and replaces them with a cable of type 27. In other words one of the following connections is made

plug 1062 - cable type 27 - plug 1021
plug 1061 - cable type 27 - plug 1021.

By this arrangement the ones in the effective address of an I/O-instruction set the corresponding bits in the interrupt register to one. Since the operators console typewriter (device no. 2) is connected to an internal I/O bus it is still possible for the program to communicate over the typewriter while the external bus is used for interruption test.
In cases where communication to the operator takes place via the external I/O bus (i.e.: device no. 2 is not installed) the interruption system cannot be completely tested.

0.5. Normal Operator Commands.

After activation of the CPU TESTPROGRAM no matter which method is used, the program arrives at the start point where it writes

test end
number of runs =

The operator now types the number of times he wants the program to be executed and terminates with a New Line (NL) character. This character must have the ISO value 10. The program then asks whether the interruption system should be tested or not by writing

interrupt/no interrupt ?

Typing an i signifies test interruption system, an n no test. The prerequisites for this test are stated in Section 0.4. The testprogram is then started and it writes at regular intervals

run no. < run no. >

where < run no. > is the run number to be executed next.
The interval equals number of runs/10.
After execution of all runs the program returns to the start point and a test may be specified anew.
If for one reason or another the operator wants to terminate the test, he only has to activate the RESET and START keys taken in that order; this forces the program to the start point.

0.6. Sequence and Interrupt Errors.

The diagnostic program consists of 87 independent test programs. Figure 0.1 shows a layout of the core and the succession in which the test programs are executed.

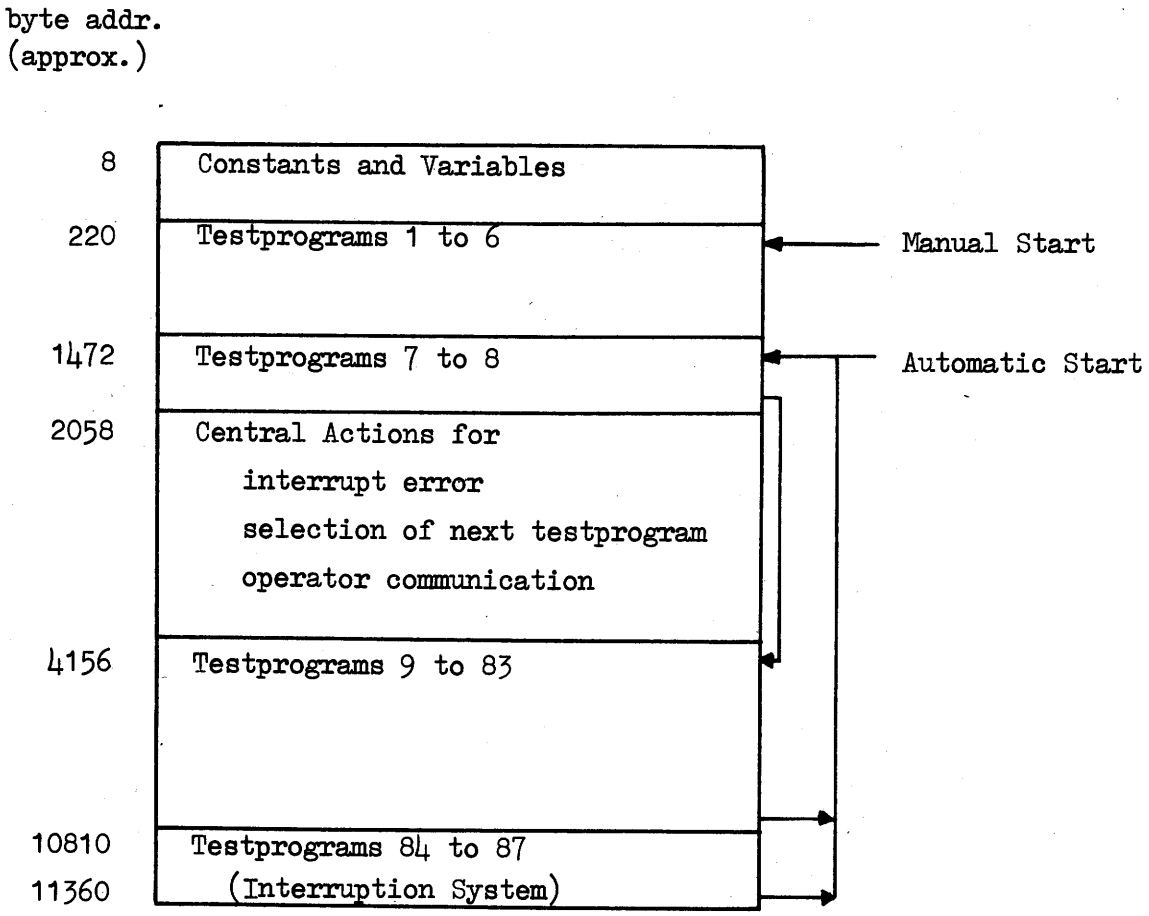


Figure 0.1 Core Layout

The last two instructions of a testprogram causes normally a loading of register w1 with the start address of the next testprogram followed by a jump to the central action. In order to check that the chronological order of testprograms is kept, the central action compares the contents of w1 against the same start address but obtained from a table stored in the central action. If the two addresses match, the diagnostic program continues with the next testprogram, otherwise the following message is printed

sequence error < name of testprogram >
repeat/continue ?

The operator may now type r for repeat, i.e. the same testprogram is executed once more, or c for continue, i.e. the next testprogram as defined by the table lookup, is executed.

If interrupts occur when no interrupts are expected the following alarm is given

interrupt error: < name of interrupted testprogram >
return addr.: < address of interrupted testprogram; core loc. 10 >
itr. no. × 2: < interrupt number ; core loc 8 >
repeat/continue ?

and the operator may choose to repeat (r) or continue (c).

0.7. Error Messages from Testprograms.

If one of the testprograms unveils an error the following error message is normally printed

< section number > < name of testprogram >
test no. < number >
received < received bitpattern >
expected < expected bitpattern >

Information about the nature of the error is found in the section equal to < section number >. Consult Section 0.8. if no section number is printed. Each section is often divided into four parts.

1. Test Purpose:
   Explains how the test is performed; the used test data are also given.
2. Error Reaction:
   Shows the exact layout of the error message and indicates in some instances the most likely error.

3. Complete Test of:

Lists the micro orders and jump conditions which are systematically tested. The corresponding logic diagrams and circuits are given. For example:

< ARU064:ARU066 - ARU068, 174, 175 >

delimits the error to logic diagrams ARU064, ARU065, ARU066, plus more specifically to logic diagram ARU068 boards 174 and 175.

4. Partly Test of:

Lists the micro orders and jump conditions which must operate satisfactory for successful test but which on the other hand has not been tested themselves.

0.8. List of Error Messages with no Section Number.

The following error messages are not preceeded by a section number

Error Message	Program Name	Section
<letter>: <24-bit pattern>	integer adder	7
<number>: <24-bit pattern>	relative addressing and jump	8
0-ext: <24-bit pattern>	sb(12) extension	9
1-ext: <24-bit pattern>	sb(12) extension	9
<program name>		consult list of contents sections 10 to 22

0.9. Alteration of Device No. for Typewriter.

The source program writes the error messages on device no. 2, the operators console typewriter. In case error messages should be printed on another IBM selectric typewriter, a Teletype or an Olivetti typewriter, the device number should be altered in the original source text. This is easily done by a global editing command where

2 < 6 is replaced by device no. < 6

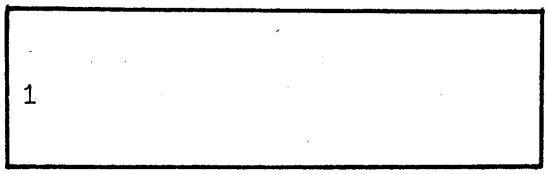
It should be noted that all new line commands in the original source text are 13, 10, 13.

Note, by altering the device no. it might not be possible to test the interruption system, confer Section 0.4.

1. WRITE THE DIGIT 1 FROM W0

The aim of this test is to write the digit 1 on the console typewriter, and the test is a link in the exploration of the output possibilities for the program. The number 49 ISO-value for digit 1 is already manually inserted in register W0.

Correct Test Output:



Error in Test Output:

1) Output is 0, 2, ..., ?

If, for example, the output is 3, it is most likely that the micro order BUS:= W(fr) has selected both register W0 and W1. In general, we have:

Output	Selected Registers	
0	none	
2	W1	NB
3	W0,W1	
4	W2	NB
5	W2,W0	
6	W2,W1	
7	W2,W1,W0	
8	W3	NB
9	W3,W0	
:	W3,W1	
;	W3,W1,W0	
<	W3,W2	
=	W3,W2,W0	
>	W3,W2,W1	
?	W3,W2,W1,W0	

NB The error may also be due to an interchange of bit positions.

2) No output at all, or the output is different from 1)

Check the io instruction by executing it Single Micro Instruction by Single Micro Instruction. Remember that the error may be due to the input/-output data transfer IO BUS:= SB.

Complete Test of:

MMode:= PROTECT for PROTECT = 1 <ARU105,421>
IO Phase A; IO Phase B; IO Timing <LCI003,LCI011,LCI012,LCI020,LCI026>
Test IO for Disconnected = 0 and Busy = 0; <ARU101,ARU102>
BR(23):= SB(23) for SB(23) = 1
SB(16:23):= W(fr)(16:23) for fr = 0 and WO = 49
EX(22,23) < > 0 [0]; <MPC006,120-ARU102,382>
MMode [1]; <MPC009,123-ARU105,421>
BR(23) [1]; <MPC008,122-ARU076,185>

Partly Test of:

Read Instruction

2. WRITE THE DIGITS 2, 4, AND 8 FROM W1, W2, AND W3, RESPECTIVELY

Continue to explore the communication possibilities by writing the digits 2, 4, and 8. The numbers 50, 52, and 56 (ISO-values for digits 2, 4, and 8) are already manually inserted in registers W1, W2, and W3, respectively.

Correct Test Output:

```
1248
```

Error in Test Output:

1) Output is 1, 2, ..., ?

Confer 1) in WRITE THE DIGIT 1 FROM W0.

2) Output sequence is 1222...22

In this case the sx instruction does not skip when the output device has cleared its Busy signal.

3) Output sequence is 1 < characters at random >

If the sx instruction always skip, it follows that the program does not wait until the typewriter clears its Busy signal - but transmits test outputs and error messages at the speed of the program. Since the typewriter cannot cope with this speed only those characters are printed which happen to be transmitted to the typewriter controller when its Busy signal is 0.

4) No output at all, or the output is different from 1), 2), and 3)

Confer 2) in WRITE THE DIGIT 1 FROM W0

Complete Test of:

Test IO for Disconnected = 0 and Busy = 1; <ARU101,ARU102>

SB(16:23):= W(fr)(16:23) for fr = 1,2,3 and

W(1) = 50, W(2) = 52, W(3) = 56

EX(22,23):= 0 where EX(23) is first set equal to 1; <ARU102,440>

AR:= 22extOconEX for EX = 0,1

AR:= AR ∧ SBa for AR = 0,1 and SB = 3

EX(22,23) < > 0 [1] for EX(22,23) = 1; <MPC006,120-ARU102:382>

MMode ∨ -,PROTECT [1] for MMode = 1 and PROTECT = 1;

<MPC010,124-ARU105,245>

Partly Test of:

Read Instruction; SB(0:11):= 12extSB(12) for SB(12) = 0

Read Data; AR:= SBa; IC:= AR(5:22)

if AR = 0 then IC:= IC+1

FR(6) ∨ FR(7) [0]

3. WRITE THE DIGITS 1, 2, 4, AND 8 FROM W0, W1, W2, AND W3, RESPECTIVELY BY MEANS OF AL INSTRUCTIONS

The numbers 49, 50, 52, and 56 are first loaded by the program into the registers, whereafter their contents are printed out as in the two previous tests.

Correct Test Output:

```
12481248
```

Error in Test Output:

An erroneous output pattern signifies most likely an error in one of the micro orders BUS(0:23):= SB or W(fr)(12:23):= BUS(12:23).

An error may, of course, also be due to one of the partly tested micro orders. The jump condition FR(6) $\vee$ FR(7) is incorrect if the only error is that 1 is replaced by the letter ø. More specific, the instruction

```
250:  jl  w0  a13
```

stores erroneously the link address 252 in register w0.

Complete Test of:

W(fr)(16:23):= SB(16:23) for fr = 0, and SB(16:23) = 49
for fr = 1, and SB(16:23) = 50
for fr = 2, and SB(16:23) = 52
for fr = 3, and SB(16:23) = 56

FR(6) $\vee$ FR(7) [0]; <MPC004,118-ARU070,092>

Partly Test of:

Read Instruction; SB(0:11):= 12extSB(12) for SB(12) = 0

Read Instruction; AR:= SBa; IC:= AR(5:22);

if AR = 0 then IC:= IC+1

4. WRITE THE CHARACTERS /, 3, AND Y FROM WO BY MEANS OF AL INSTRUCTIONS

If this test, which checks the input/output datapaths, and the previous tests, all work satisfactory, the program is capable of writing any message on the typewriter. Register WO is loaded successively by the following patterns:

	ISO-value	Character
WO(0:16):= 0; WO(17:23):= 0101111	47	/
WO(0:16):= 0; WO(17:23):= 0110011	51	3
WO(0:16):= 0; WO(17:23):= 1011001	89	Y

Correct Test Output:

```
12481248/3Y
```

Error in Test Output:
The datapaths conveying the output information are all checked by these patterns, and any interchange of bit positions will be disclosed.

Complete Test of:

W(fr)(16:23):= SB(16:23) for fr = 0
SB(16:23):= W(fr)(16:23) for fr = 0

Partly Test of:

Read Instruction; SB(0:11):= 12extSB(12) for SB(12) = 0
Read Data; AR:= SBa; IC:= AR(5:22);
if AR = 0 then IC:= IC+1

5. WRITE THE DIGITS 1, 2, 4, AND 8 FROM WO BY MEANS OF LO INSTRUCTIONS

This method of constructing the above-mentioned digits is employed in the test REGISTER W1 AND CORE STORE DATA PATHS in order to specify different errors.

Correct Test Output:

12481248/3Y1248

Error in Test Output:
No new micro orders or jump conditions are tested, and errors originate consequently from the partly tested items.

Complete Test of:
No new micro orders or jump conditions are tested.

Partly Test of:

Read Instruction; SB(0:11):= 12extSB(12) for SB(12) = 0
Read Data; AR:= SBa; IC:= AR(5:22);
if AR = 0 then IC:= IC+1

6. REGISTER W1 AND CORE STORE DATA PATH

A number of commonly used micro orders are tested by this scheme.
First, register W1 is loaded from core store with a testpattern and the contents of W1 are verified. Second, the contents of W1 are stored in a fixed location in core store from where it is reloaded into W1 and the testpattern is checked once more.
The 26 testpatterns are for:

Test no.	1	000000 000000 000000 000000
	2	111111 111111 111111 111111
	3	100000 000000 000000 000000
	4	010000 000000 000000 000000
	5	001000 000000 000000 000000
	6	000100 000000 000000 000000
	7	000010 000000 000000 000000
	8	000001 000000 000000 000000
	9	000000 100000 000000 000000
	10	000000 010000 000000 000000
	11	000000 001000 000000 000000
	12	000000 000100 000000 000000
	13	000000 000010 000000 000000
	14	000000 000001 000000 000000
	15	000000 000000 100000 000000
	16	000000 000000 010000 000000
	17	000000 000000 001000 000000
	18	000000 000000 000100 000000
	19	000000 000000 000010 000000
	20	000000 000000 000001 000000

```
21      000000 000000 000000 100000
22      000000 000000 000000 010000
23      000000 000000 000000 001000
24      000000 000000 000000 000100
25      000000 000000 000000 000010
26      000000 000000 000000 000001
```

The test works in details as described by the Slang program below

```
z1:  testpattern
z2:  inverted testpattern
z3:  1
z4:  2
z6:  4
z8:  8
b6:  current testpattern

                              ; Load Test:
     al  w0       48          ;   w0:= 48; comment ISO character 0;
     rl  w1       z1          ;   w1:= testpattern;
     sz  w1 (     z2          ;   if ors (inverted testpattern ∧ w1)
     lo  w0       z3          ;     then w0:= w0+1; comment Error Type I;
     so  w1 (     z1          ;   if ors (testpattern ∧ -,w1)
     lo  w0       z4          ;     then w0:= w0+2; comment Error Type II;
                              ; Store Test:
     rs  w1       b6          ;   ST(b6):= w1;
     rl  w1       b6          ;   w1:= ST(b6);
     sz  w1 (     z2          ;   if ors (inverted testpattern ∧ w1)
     lo  w0       z6          ;     then w0:= w0+4; comment Error Type III;
     so  w1 (     z1          ;   if ors (testpattern ∧ -,w1)
     lo  w0       z8          ;     then w0:= w0+8; comment Error Type IV;
z9:  io  w0       2<6+3       ;   write(w0);
     sx           3
     jl           z9          ;   goto next testpattern;
```

Correct Test Output:

```
12481248/3Y1248
00 000000 000000 000000 000000
```

Error in Test Output:

Each character in the last output line explains the result of a specific testpattern. For example, the 5th character signifies the outcome of the 5th testpattern.

The interpretation of the test output appears in this table:

Test Output Character	Error Type IV	III	II	I
0	no error			
1				x
2			x	
3			x	x
4		x		
5		x		x
6		x	x	
7		x	x	x
8	x			
9	x			x
:	x		x	
;	x		x	x
<	x	x		
=	x	x		x
>	x	x	x	
?	x	x	x	x

Error Type I

Instruction	Micro Orders
rl w1 z1	SB:= testpattern from storage
	W(1):= SB
sz w1 (z2	SB:= inverted testpattern from storage
	AR:= Wa(1)
	AR:= AR ∧ SBa
	if AR = 0 then IC:= IC+1

Error Type II

Instruction	Micro Orders
rl w1 z1	See Error Type I
so w1 (z1	SB:= testpattern from storage
	AR:= SBa
	SB:= W(1)
	AR:= AR ∧ -,SB
	if AR = 0 then IC:= IC+1

Error Type III

Instruction	Micro Orders
rs w1 y3	AR:= Wa(1)
	SB:= AR(0:23)
	storage(y3):= SB
rl w1 y3	See Error Type I
sz w1 (z2	See Error Type I

Error Type IV

Instruction	Micro Orders
rs w1 y3	See Error Type III
rl w1 y3	See Error Type I
so w1 (z1	See Error Type II

Complete Test of:

```
Read Data  for SB:= STdata(0:23)
  comment data path from core store to SB;
  <STC006-STC010:STC013-STC019:STC020-ARU084:ARU086-ARU090,363>
Read Split for STdata(0:23):= SB
  comment data path from SB to core store;
  <STC006,499,505-STC010:STC013-STC017:STC018>
BUS(0:23):= SB; <ARU002:ARU025-ARU090,237>
W(fr)(0:11):= BUS(0:11)   for fr = 1; <ARU052,285,292-ARU058,385>
W(fr)(12:23):= BUS(12:23) for fr = 1; <ARU053,304,313-ARU058,385>
BUS(0:23):= W(fr)         for fr = 1; <ARU002:ARU025-ARU058,281>
AR(-1:23):= if MC(11) then BUS(-1:23) else BUS(0,0:23)
 for MC(11) = 0 but except for AR(-1); <ARU071:ARU072-ARU081,311
                                                    -ARU082,190>
Adder:= b 1011 000; Adder:= b 0111 000; <ARU098:ARU100>
BUS(0:23):= AND(0:23)
  for 1∧0 = 0 and 0∧1 = 0; <ARU093:ARU095>
BUS(-1:23):= AR(-1:23);    <ARU001:ARU025-ARU082,237>
SB(0:11):= BUS(0:11);      <ARU084-ARU085-ARU090-ARU091>
SB(12:23):= BUS(12:23);    <ARU086-ARU090-ARU091>
```

Partly Test of:

```
Read Instruction; SB(0:11):= 12extSB(12) for SB(12) = 0
Read Data; Read Split; Split Write
IC:= AR(5:22)
if AR = 0 then IC:= IC+1
```

7. INTEGER ADDER

The adder and carry circuitry for integer addition and subtraction is tested by means of the testpatterns shown below. The first 12 tests (identified by small letters) are concerned with addition and the next 12 (identified by capital letters) with subtraction. The main purpose of these 24 tests is a complete verification of the actual adder network, but the tests do not assure that, for example, bit(0) and bit(2) have not been interchanged when connected to the arithmetic bus. This type of error is solved by the additional tests u, v, x, y, and z.

```
Test a:  AR:= 00000000 00000000 00000000
         SB:= 00000000 00000000 00000000
    AR + SB:= 00000000 00000000 00000000

Test b:  AR:= 00000000 00000000 00000000
         SB:= 11111111 11111111 11111111
    AR + SB:= 11111111 11111111 11111111

Test c:  AR:= 11111111 11111111 11111111
         SB:= 00000000 00000000 00000000
    AR + SB:= 11111111 11111111 11111111

Test d:  AR:= 01010101 01010101 01010101
         SB:= 01010101 01010101 01010101
    AR + SB:= 10101010 10101010 10101010

Test e:  AR:= 00000001 00000000 00000001
         SB:= 11111111 11111110 11111111
    AR + SB:= 00000000 11111111 00000000

Test f:  AR:= 00000010 00000001 00000010
         SB:= 11111110 11111101 11111110
    AR + SB:= 00000000 11111111 00000000
```

```
Test g:  AR:= 00000100 00000011 00000100
         SB:= 11111100 11111011 11111100
    AR + SB:= 00000000 11111111 00000000

Test h:  AR:= 00001000 00000111 00001000
         SB:= 11111000 11110111 11111000
    AR + SB:= 00000000 11111111 00000000

Test j:  AR:= 00010000 00001111 00010000
         SB:= 11110000 11101111 11110000
    AR + SB:= 00000000 11111111 00000000

Test k:  AR:= 00100000 00011111 00100000
         SB:= 11100000 11011111 11100000
    AR + SB:= 00000000 11111111 00000000

Test l:  AR:= 01000000 00111111 01000000
         SB:= 11000000 10111111 11000000
    AR + SB:= 00000000 11111111 00000000

Test m:  AR:= 10000000 01111111 10000000
         SB:= 10000000 01111111 10000000
    AR + SB:= 00000000 11111111 00000000

Test A:  AR:= 00000000 00000000 00000000
         SB:= 00000000 00000000 00000000
    AR - SB:= 00000000 00000000 00000000

Test B:  AR:= 11111111 11111111 11111111
         SB:= 00000000 00000000 00000000
    AR - SB:= 11111111 11111111 11111111

Test C:  AR:= 11111111 11111111 11111111
         SB:= 11111111 11111111 11111111
    AR - SB:= 00000000 00000000 00000000

Test D:  AR:= 10101010 10101010 10101010
         SB:= 01010101 01010101 01010101
    AR - SB:= 01010101 01010101 01010101
```

```
Test E:  AR:= 00000000 00000001 00000000
         SB:= 00000001 00000000 00000001
    AR - SB:= 11111111 00000000 11111111

Test F:  AR:= 00000001 00000010 00000001
         SB:= 00000010 00000001 00000010
    AR - SB:= 11111111 00000000 11111111

Test G:  AR:= 00000011 00000100 00000011
         SB:= 00000100 00000011 00000100
    AR - SB:= 11111111 00000000 11111111

Test H:  AR:= 00000111 00001000 00000111
         SB:= 00001000 00000111 00001000
    AR - SB:= 11111111 00000000 11111111

Test J:  AR:= 00001111 00010000 00001111
         SB:= 00010000 00001111 00010000
    AR - SB:= 11111111 00000000 11111111

Test K:  AR:= 00011111 00100000 00011111
         SB:= 00100000 00011111 00100000
    AR - SB:= 11111111 00000000 11111111

Test L:  AR:= 00111111 01000000 00111111
         SB:= 01000000 00111111 01000000
    AR - SB:= 11111111 00000000 11111111

Test M:  AR:= 01111111 10000000 01111111
         SB:= 10000000 01111111 10000000
    AR - SB:= 11111111 00000000 11111111

Test u:  AR:= 00000000 00001111 11111111
         SB:= 00000000 00000000 00000000
    AR + SB:= 00000000 00001111 11111111

Test v:  AR:= 00000011 11110000 00111111
         SB:= 00000000 00000000 00000000
    AR + SB:= 00000011 11110000 00111111
```

```
Test x:  AR:= 00011100 01110001 11000111
         SB:= 00000000 00000000 00000000
    AR + SB:= 00011100 01110001 11000111

Test y:  AR:= 00100100 10010010 01001001
         SB:= 00000000 00000000 00000000
    AR + SB:= 00100100 10010010 01001001

Test z:  AR:= 01001001 00100100 10010010
         SB:= 00000000 00000000 00000000
    AR + SB:= 01001001 00100100 10010010
```

Error Reaction:

If a test fails, the test identification letter and the result of the test shall be printed out. For example:

```
G:  11110111 00000000 11111111
```

shows that Test G has failed and, by comparing with the correct result, bit(4) is spotted to produce the error.

The way in which the erroneous binary result is printed out is based on a shift instruction which has not yet been tested.

Complete Test of:

Adder:= b 1011 000; Adder:= b 0111 100 <ARU098:ARU100>

BUT(-1:23):= SUM(-1:23) except for SUM(-1) <ARU001:ARU025 -ARU098,234>

Partly Test of:

Read Instruction; SB(0:11):= 12extSB(12) for SB(12) = 0

Read Data; IC:= AR(5:22)

if AR = 0 then IC:= IC+1

The list does not include the micro orders used in possible error messages.

8. RELATIVE ADDRESSING AND JUMP

Transfer of addresses from BUS to the instruction counter, IC, and vice versa is checked for addresses in the interval 6 <= address < 32K bytes. Likewise, these addresses are used to check the address path from IC and SB to the address register in core store.
The test generates the following piece of code starting at location addr.

```
addr:   al  w1    0
        al. w1   -2          ;   w1:= addr + 2 - 2;
        se  w1    addr       ;   if w1 < > addr
        jl        error      ;     then write contents of w1
        jl        next test  ;     else goto next test
```

When the program is set up, a jump is made into location addr. Addr is for the various tests as follows:

```
Test no. 1     00000000 01011101 11000000
         2     00000000 01100101 00100110
         3     00000000 01110010 10010100
         4     00000000 00111100 00111000
```

Error Reaction:
If a test fails, the test no. and the contents of register W1 shall be printed out. For example:

```
        3:    00000000 00110010  10010000
```

The way in which the contents of W1 is printed out is based on a shift instruction which has not been tested.

Complete Test of:

IC:= BUS(5:22) for 0 <= BUS(5:22)con0 < 32K bytes;
<ARU060:ARU062-ARU063,171,175>

BUS(0:23):= 5ext0conICcon0 for 0 <= ICcon0 < 32K bytes;
<ARU016:ARU024-ARU029:ARU030-ARU063,237>

if -,FR(8) then AR(-1:23):= BUS(0,0:23); <ARU080,255-ARU081,255>

Read Instruction for STaddr:= IC(6:22) 4 <= ICcon0 < 32K bytes
comment core store address selection from IC;
<STC004-STC008:STC009-STC016>

Read Split and Read Data for STaddr:= SB(6:22) 4 <= SB < 32K bytes
comment core store address selection from SB;
<STC004-STC008:STC009-STC016>

Partly Test of:

SB(0:11):= 12extSB(12) for SB(12) = 0.1

if AR = 0 then IC:= IC + 1

The list does not include the micro orders used in possible error messages.

9. SB(12) EXTENSION

The micro order SB(0:11):= 12extSB(12), used to convert a 12-bit integer to a 24-bit integer, is checked by means of the bl instruction.

Error Reaction:
The two types of error messages for zero- and one-extensions are:

0-ext: <received result of 12extSB(12)conSB(12:23)>

1-ext: <received result of 12extSB(12)conSB(12:23)>

where SB before extension was equal to

SB = b 111111111111 011111111111 for 0-ext

and SB = b 000000000000 100000000000 for 1-ext

If the incorrect results are -1 (for 0-ext) and 0 (for 1-ext), the left halfword has properly been selected which signifies that the jump condition HA(23) is s-a-0.
The way in which received results are printed out is based on a shift instruction which has not been tested.

Complete Test of:

SB(0:11):= 12extSB(12) <ARU084-ARU085-ARU090,363-ARU091,374>

HA(23) [1] <MPC007,121-STC003,445>

Partly Test of:

if AR = 0 then IC:= IC+1

The list does not include the micro orders used in possible error messages.

10. SKIP ON AR(-1) = 1

The micro order, if AR(-1) = 1 then IC:= IC+1, is verified by the aid of the sl instruction and the following testpatterns.

Testpattern in AR(-1:23):

Test no. 1	0	000000	000000	000000	000000
2	0	000000	000000	000000	000001
3	1	100000	000000	000000	000000

Error Reaction:

If the program does not skip as expected, the test number will be typed out, for example:

```
skip on ar(-1) = 1
test no. 2
```

An error is most likely due to the decoding network for register AR.

Complete Test of:

```
if AR(-1) = 1 then IC:= IC+1 <ARU063,168,176>
AR(-1:23):= if MC(11) then BUS(-1:23) else BUS(0,0:23)
  for MC(11) = 0 and AR(-1); <ARU001-ARU071,158-ARU081,311
                                                -ARU082,190>
BUS(-1:23):= SUM(-1:23) for SUM(-1) <ARU001-ARU098,234>
```

11. SKIP ON AR < > 0

The micro order, if AR < > 0 then IC:= IC+1, is verified by the aid of the sn instruction and the following testpatterns.

Testpattern in AR(-1:23):

Test no. 1	0	000000	000000	000000 000000
2	0	100000	000000	000000 000000
3	0	010000	000000	000000 000000
4	0	001000	000000	000000 000000
5	0	000100	000000	000000 000000
6	0	000010	000000	000000 000000
7	0	000001	000000	000000 000000
8	0	000000	100000	000000 000000
9	0	000000	010000	000000 000000
10	0	000000	001000	000000 000000
11	0	000000	000100	000000 000000
12	0	000000	000010	000000 000000
13	0	000000	000001	000000 000000
14	0	000000	000000	100000 000000
15	0	000000	000000	010000 000000
16	0	000000	000000	001000 000000
17	0	000000	000000	000100 000000
18	0	000000	000000	000010 000000
19	0	000000	000000	000001 000000
20	0	000000	000000	000000 100000
21	0	000000	000000	000000 010000
22	0	000000	000000	000000 001000
23	0	000000	000000	000000 000100
24	0	000000	000000	000000 000010
25	0	000000	000000	000000 000001

It should be noted that AR(-1) = 1 and AR(0:23) = 0 can never occur by subtracting two 24-bit numbers.

Error Reaction:

If the program does not skip as expected, the test number will be typed out, for example:

```
skip on ar < > 0
test no.  9
test no. 17
```

An error is most likely due to the decoding network for register AR.

Complete Test of:

```
if AR < > 0 then IC:= IC+1; <ARU063,168,176>
AR(-1:23):= if MC(11) then BUS(-1:23) else BUS(0,0:23)
  for MC(11) = 1; <ARU001-ARU071,158-ARU081,311-ARU082,190>
```

12. SKIP ON AR = 0

The micro order, if AR = 0 then IC:= IC+1, is verified by the aid of the se instruction and the following testpatterns.

Testpattern in AR(-1:23):

Test no. 1 0 000000 000000 000000 000000

2 1 100000 000000 000000 000000

Error Reaction:

If the program does not skip as expected, the test number will be typed out, for example:

skip on ar = 0

test no. 2

An error is most likely due to the decoding network for register AR.

Complete Test of:

if AR = 0 then IC:= IC+1; <ARU063,168,176>

13. SKIP ON AR > 0

The micro order, if AR > 0 then IC:= IC+1, is verified by the aid of the sh instruction and following testpatterns.

Testpattern in AR(-1:23):

Test no. 1	0 000000	000000	000000
2	1 100000	000000	000000
3	0 000000	000000	000001

Error Reaction:

If the program does not skip as expected, the test number will be typed out, for example:

```
skip on ar > 0
test no. 1
```

An error is most likely due to the decoding network for register AR.

Complete Test of:

```
if AR > 0 then IC:= IC+1; <ARU063,168,176>
```

14. INDEX X1

The index address mode for index X1 is tested where

	W0 = b 000000 000000	000000 000001
index X1 =	W1 = b 111111 000000	100000 000000
	W2 = b 000000 000000	000000 000100
	W3 = b 000000 000000	000000 001000

and the displacement d = b 000000 000000 011111 000000

Error Reaction:

If the calculated effective address, d + X1, differs from the expected one, the program issues the following message

index x1
received <24-bit received result>
expected 111111 000000 111111 000000

If either none, W0, W2, or W3 is selected instead of W1, the received results are:

received result	selected index register
000000 000000 011111 000000	none
000000 000000 011111 000001	W0
000000 000000 011111 000100	W2
000000 000000 011111 001000	W3

If more than one index register is selected, the received result shall be equal to the logical sum of these registers.

If bit 11 becomes 1, it is probably because the instruction counter is added to the displacement, instead of the index register; confer the second micro order under Complete Test.

The way in which the received result is printed out is based on a shift instruction which has not yet been tested. The received result is properly the received one if the expected result is printed correctly.

Complete Test of:

BUS(0:23):= W(index) for index = 1; <ARU002:ARU025-ARU058,281>

if -,FR(8) then AR(-1:23):= BUS(0,0:23); <ARU080-255-ARU081,255>

Partly Test of:

The list does not include the micro orders used in possible error messages.

15. REGISTER W2

Register W2 is checked by means of the following testpatterns:

Test no.				
1	000000	000000	000000	000000
2	100000	000000	000000	000000
3	010000	000000	000000	000000
4	001000	000000	000000	000000
5	000100	000000	000000	000000
6	000010	000000	000000	000000
7	000001	000000	000000	000000
8	000000	100000	000000	000000
9	000000	010000	000000	000000
10	000000	001000	000000	000000
11	000000	000100	000000	000000
12	000000	000010	000000	000000
13	000000	000001	000000	000000
14	000000	000000	100000	000000
15	000000	000000	010000	000000
16	000000	000000	001000	000000
17	000000	000000	000100	000000
18	000000	000000	000010	000000
19	000000	000000	000001	000000
20	000000	000000	000000	100000
21	000000	000000	000000	010000
22	000000	000000	000000	001000
23	000000	000000	000000	000100
24	000000	000000	000000	000010
25	000000	000000	000000	000001

Error Reaction:

If the test fails, the programs issue the following message:

register w2

received <24-bit received result>

expected <24-bit expected result>

The way in which the received result is printed out is based on a shift instruction which has not yet been tested. The received result is properly the received one if the expected result is printed correctly.

Complete Test of:

W(fr)(0:11):= BUS(0:11) for fr = 2; <ARU054,286,294-ARU059,436>

W(fr)(12:23):= BUS(12:23) for fr = 2; <ARU055,307,316-ARU059,436>

BUS(0:23):= W(fr) for fr = 2; <ARU002:ARU025-ARU059,91>

Partly Test of:

The list does not include the micro orders used in possible error messages.

16. REGISTER W3

Register W3 is checked by means of the following testpatterns:

Test no. 1	000000	000000	000000	000000
2	100000	000000	000000	000000
3	010000	000000	000000	000000
4	001000	000000	000000	000000
5	000100	000000	000000	000000
6	000010	000000	000000	000000
7	000001	000000	000000	000000
8	000000	100000	000000	000000
9	000000	010000	000000	000000
10	000000	001000	000000	000000
11	000000	000100	000000	000000
12	000000	000010	000000	000000
13	000000	000001	000000	000000
14	000000	000000	100000	000000
15	000000	000000	010000	000000
16	000000	000000	001000	000000
17	000000	000000	000100	000000
18	000000	000000	000010	000000
19	000000	000000	000001	000000
20	000000	000000	000000	100000
21	000000	000000	000000	010000
22	000000	000000	000000	001000
23	000000	000000	000000	000100
24	000000	000000	000000	000010
25	000000	000000	000000	000001

The way in which the received result is printed out is based on a shift instruction which has not yet been tested. The received result is properly the received one if the expected result is printed correctly.

Error Reaction:

If the test fails, the programs issue the following message:

register w3

received <24-bit received result>

expected <24-bit expected result>

Complete Test of:

W(fr)(0:11):= BUS(0:11) for fr = 3; <ARU056,286,294-ARU059,436>

W(fr)(12:23):= BUS(12:23) for fr = 3; <ARU057,307,316-ARU059,436>

BUS(0:23):= W(fr) for fr = 3; <ARU002:ARU025-ARU059,091>

Partly Test of:

The list does not include the micro orders used in possible error messages.

17. REGISTER WO

Register WO is checked by means of the following testpatterns:

Test no. 1	000000	000000	000000	000000
2	100000	000000	000000	000000
3	010000	000000	000000	000000
4	001000	000000	000000	000000
5	000100	000000	000000	000000
6	000010	000000	000000	000000
7	000001	000000	000000	000000
8	000000	100000	000000	000000
9	000000	010000	000000	000000
10	000000	001000	000000	000000
11	000000	000100	000000	000000
12	000000	000010	000000	000000
13	000000	000001	000000	000000
14	000000	000000	100000	000000
15	000000	000000	010000	000000
16	000000	000000	001000	000000
17	000000	000000	000100	000000
18	000000	000000	000010	000000
19	000000	000000	000001	000000
20	000000	000000	000000	100000
21	000000	000000	000000	010000
22	000000	000000	000000	001000
23	000000	000000	000000	000100
24	000000	000000	000000	000010
25	000000	000000	000000	000001

Error Reaction:

If the test fails, the programs issue the following message:

register w0
received <24-bit received result>
expected <24-bit expected result>

The way in which the received result is printed out is based on a shift instruction which has not yet been tested. The received result is properly the received one if the expected result is printed correctly.

Complete Test of:

W(fr)(0:11):= BUS(0:11) for fr = 0; <ARU050,285,292-ARU058,385>
W(fr)(12:23):= BUS(12:23) for fr = 0; <ARU051,304,313-ARU058,385>
BUS(0:23):= W(fr) for fr = 0; <ARU002:ARU025-ARU058,281>

Partly Test of:

The list does not include the micro orders used in possible error messages.

18. INDEX X2

The index address mode for index X2 is tested where

	W0 = b 000000 000000	000000 000001
	W1 = b 000000 000000	000000 000010
index X2 =	W2 = b 111111 000000	100000 000000
	W3 = b 000000 000000	000000 001000
and the displacement	d = b 000000 000000	011111 000000

Error Reaction:
If the calculated effective address, d + X2, differs from the expected one, the program issues the following message

index x2
received <24-bit received result>
expected 111111 000000 111111 000000

If either none, W0, W1, or W3 is selected instead of W2, the received results are:

received result		selected index register
000000 000000	011111 000000	none
000000 000000	011111 000001	W0
000000 000000	011111 000010	W1
000000 000000	011111 001000	W3

If more than one index register is selected, the received result shall be equal to the logical sum of these registers.
The way in which the received result is printed out is based on a shift instruction which has not yet been tested. The received result is properly the received one if the expected result is printed correctly.

Complete Test of:

BUS(0:23):= W(index) for index = 2; <ARU002:ARU025-ARU059,091>

Partly Test of:

The list does not include the micro orders used in possible error messages.

19. INDEX X3

The index address mode for index X3 is tested where

W0 = b 000000 000000 000000 000001
W1 = b 000000 000000 000000 000010
W2 = b 000000 000000 000000 000100
index X3 = W3 = b 111111 000000 100000 000000
and the displacement d = b 000000 000000 011111 000000

Error Reaction:
If the calculated effective address, d + X3, differs from the expected one, the program issues the following message

index x3
received <24-bit received result>
expected 111111 000000 111111 000000

If either none, W0, W1, or W2 is selected instead of W3, the received results are:

received result	selected index register
000000 000000 011111 000000	none
000000 000000 011111 000001	W0
000000 000000 011111 000010	W1
000000 000000 011111 000100	W2

If more than one index register is selected, the received result shall be equal to the logical sum of these registers.
The way in which the received result is printed out is based on a shift instruction which has not yet been tested. The received result is properly the received one if the expected result is printed correctly.

Complete Test of:

BUS(0:23):= W(index) for index = 3; <ARU002:ARU025-ARU059,091>

Partly Test of:

The list does not include the micro orders used in possible error messages.

20. LOGICAL LEFT SHIFT SINGLE

The left shift mechanisms for register AR (used by single shift instructions) are checked by means of the testpatterns in the table below. The register is only shifted 1 position for each testpattern.

Testpattern in AR(0:23)conBR(0):

Test no.					
1	100000	000000	000000	000000	0
2	010000	000000	000000	000000	0
3	001000	000000	000000	000000	0
4	000100	000000	000000	000000	0
5	000010	000000	000000	000000	0
6	000001	000000	000000	000000	0
7	000000	100000	000000	000000	0
8	000000	010000	000000	000000	0
9	000000	001000	000000	000000	0
10	000000	000100	000000	000000	0
11	000000	000010	000000	000000	0
12	000000	000001	000000	000000	0
13	000000	000000	100000	000000	0
14	000000	000000	010000	000000	0
15	000000	000000	001000	000000	0
16	000000	000000	000100	000000	0
17	000000	000000	000010	000000	0
18	000000	000000	000001	000000	0
19	000000	000000	000000	100000	0
20	000000	000000	000000	010000	0
21	000000	000000	000000	001000	0
22	000000	000000	000000	000100	0
23	000000	000000	000000	000010	0
24	000000	000000	000000	000001	0

Error Reaction:

If the program finds an error, the test number will be typed out, for example:

```
logical left shift single
test no. 20
```

If all tests fail, it may be due to the jump conditions

SB(0), SB < > 0, SB ≥ 64, SC < > 0

Complete Test of:

```
SC:= BUS(11:23) for BUS(11:23) = 1; <ARU064:ARU066-ARU068,174,175>
SC:= SC-1 for SC = 1; <ARU068,174,089>
BR:= BUS(0:23) for BUS(0) = 0; <ARU075,153-ARU083,169,190>
Adder:= b 1111 000; <ARU098:ARU100>
lshl ARconBR     Only shifts in AR are tested; <ARU071:ARU072,
                                                ARU080:ARU083>
SB(0)    [0]            ; <MPC009,123-ARU084,283>
SB < > 0 [1] for SB = 1; <MPC010,124-ARU089>
SB ≥ 64  [0] for SB = 1; <MPC003,117-ARU089>
```

Partly Test of:

```
SC < > 0 [0]            ; <MPC010,124-ARU067>
```

21. W PRE

The micro orders which transfer from BUS to W(pre) and vice versa are tested separately for the 4 working registers W0, W1, W2, and W3 by means of the ld instruction (only one shift).
The testpatterns loaded into these registers appear from the scheme below

contents of	W0	W1	W2	W3
for the test of W0	p	2	4	8
for the test of W1	1	p	4	8
for the test of W2	1	2	p	8
for the test of W3	1	2	4	p

where p:= b 000000 000000 100000 000000

Error Reaction:
The error message is

w <number> pre
received <24-bit received result>
expected 000000 000001 000000 000000

If either W0, W1, W2, or W3 is selected instead of the appropriate one, the received results are

received result	selected W(pre) register
000000 000000 000000 000010	W0
000000 000000 000000 000100	W1
000000 000000 000000 001000	W2
000000 000000 000000 010000	W3

If more than one W(pre) is selected the received result should be equal to the logical sum of these registers.
The way in which the received result is printed out is based upon a shift instruction which is only partly tested. The received result is properly the received one if the expected result is printed correctly.

Complete Test of:

```
BUS(0:23):= W(pre)   ;   <ARU058:ARU059>
W(pre):= BUS(0:23)   ;   <ARU058:ARU059>
SC < > 0   [0]       ;   <MPC010,124 - ARU067>
```

Partly Test of:

The list does not include the micro orders used in possible error messages.

22. LOGICAL LEFT SHIFT DOUBLE

The left shift mechanisms for register BR (used by double shift instructions) are checked by means of the testpatterns in the table below. The register is only shifted 1 position for each testpattern.

Testpattern in AR(23)conBR(0:23):

Test no. 1	0	100000	000000	000000	000000
2	0	010000	000000	000000	000000
3	0	001000	000000	000000	000000
4	0	000100	000000	000000	000000
5	0	000010	000000	000000	000000
6	0	000001	000000	000000	000000
7	0	000000	100000	000000	000000
8	0	000000	010000	000000	000000
9	0	000000	001000	000000	000000
10	0	000000	000100	000000	000000
11	0	000000	000010	000000	000000
12	0	000000	000001	000000	000000
13	0	000000	000000	100000	000000
14	0	000000	000000	010000	000000
15	0	000000	000000	001000	000000
16	0	000000	000000	000100	000000
17	0	000000	000000	000010	000000
18	0	000000	000000	000001	000000
19	0	000000	000000	000000	100000
20	0	000000	000000	000000	010000
21	0	000000	000000	000000	001000
22	0	000000	000000	000000	000100
23	0	000000	000000	000000	000010
24	0	000000	000000	000000	000001

Error Reaction:

If the program finds an error, the test number will be typed out, for example:

```
logical left shift double
test no. 13
```

Complete Test of:

```
BR:= BUS(0:23); <ARU075:ARU076-ARU083,169,190>
BUS(0:23):= BR; <ARU002:ARU025-ARU083,234>
lshl ARconBR; <ARU075:ARU076,ARU080:ARU083>
```

23. JUMP

The first two tests check the jump condition, FR(6) $\vee$ FR(7), for s-a-0. The third test inspects the micro order, BUS(-1:23):= AR(-1:22)con0. This micro order has only effect when a jl instruction, having an odd effective address, is followed by an instruction where relative mode is employed. An example:

```
            jl.     addr+1
addr:       al. w1  0
```

If the micro order functions, W1 = addr (bit 23 is 0), if not, W1 = addr +1.

Error Reaction:

```
jump
test no.      <number>
received      <24-bit received return address>
expected      <24-bit expected return address>
```

Test no. 1 fails: The received result is then -1 and no link is stored because FR(7) is s-a-0 in the jump condition, FR(6) $\vee$ FR(7).
Test no. 2 fails: The received result is then -1 and no link is stored because FR(6) is s-a-0 in the jump condition, FR(6) $\vee$ FR(7).
Test no. 3 fails: Bit 23 is 1 in the received result; confer the above example.

Complete Test of:

```
BUS(-1:23):= AR(-1:22)con0; <ARU082,237>
FR(6) ∨ FR(7)  [1]        ; <MPC004,118-ARU070,092>
```

24. JUMP CONDITIONS FOR FR

Jump conditions derived from the function register, FR, are all tested by executing a number of instructions with different address modifications. A number of these jump conditions are tested beforehand, but not in a systematic manner.

	Instruction	FR(0:5)	FR(8:11)
Test no. 1	rl w0	010100	0000
2	rl w0 x1	010100	0001
3	rl w0 x2	010100	0010
4	rl w0 (	010100	0100
5	rl. w0	010100	1000
6	al w0	001011	0000
7	al w0 x1	001011	0001
8	al w0 x2	001011	0010
9	al w0 (	001011	0100
10	al. w0	001011	1000
11	al w0	001011	0000
12	ac w0	100001	0000
13	ac w0 x1	100001	0001
14	ac w0 x2	100001	0010
15	ac w0 (	100001	0100
16	ac. w0	100001	1000

Error Reaction:

jump conditions for fr
test no. <number>
received <24-bit received result>
expected <24-bit expected result>

The expected result is 111110 001111 011111 111000 for tests 1 to 5 and
000000 000000 011111 111111 for tests 6 to 16.

Complete Test of:

FR(n) ∧ ~,Modif for n = 0,1,2,3,4,5 [0] [1]; <MPC005:MPC010>

FR(n) for n = 0,1,2,3,4,5,8,9 [0] [1]; <MPC002:MPC003,MPC005:MPC010>

FR(10) ∨ FR(11) [0] [1]; <MPC011-ARU070,92>

25. REGISTER FR

The function register, FR, is normally loaded directly from store. However, for instructions that follow immediately after a jump instruction, the FR register is loaded from the arithmetic bus. This is tested for the following instructions:

		Instruction	FR(0:5)	FR(6:11)
Test no.	1	al w0 (x3	001011	000111
	2	rl w1 x1	010100	010001
	3	al w3 (x1	001011	110101
	4	ac w0 (	100001	000100
	5	wa. w0 x1	000111	001001

Error Reaction:

register fr
test no. <number>
received <24-bit received result>
expected 111110 001111 011111 111000

Complete Test of:

FR:= BUS(0:11); <ARU069-ARU070,174,92,27>

26. LOGICAL RIGHT SHIFT

The logical right shift mechanisms for registers AR and BR are checked by means of the testpatterns in the table below. The registers are only shifted 1 position for each testpattern.

Testpattern in AR(-1:23)conBR(0:23):

Test no. 1	1	100000	000000	000000	000000	000000	000000	000000	000000
2	0	010000	000000	000000	000000	000000	000000	000000	000000
3	0	001000	000000	000000	000000	000000	000000	000000	000000
4	0	000100	000000	000000	000000	000000	000000	000000	000000
5	0	000010	000000	000000	000000	000000	000000	000000	000000
6	0	000001	000000	000000	000000	000000	000000	000000	000000
7	0	000000	100000	000000	000000	000000	000000	000000	000000
8	0	000000	010000	000000	000000	000000	000000	000000	000000
9	0	000000	001000	000000	000000	000000	000000	000000	000000
10	0	000000	000100	000000	000000	000000	000000	000000	000000
11	0	000000	000010	000000	000000	000000	000000	000000	000000
12	0	000000	000001	000000	000000	000000	000000	000000	000000
13	0	000000	000000	100000	000000	000000	000000	000000	000000
14	0	000000	000000	010000	000000	000000	000000	000000	000000
15	0	000000	000000	001000	000000	000000	000000	000000	000000
16	0	000000	000000	000100	000000	000000	000000	000000	000000
17	0	000000	000000	000010	000000	000000	000000	000000	000000
18	0	000000	000000	000001	000000	000000	000000	000000	000000
19	0	000000	000000	000000	100000	000000	000000	000000	000000
20	0	000000	000000	000000	010000	000000	000000	000000	000000
21	0	000000	000000	000000	001000	000000	000000	000000	000000
22	0	000000	000000	000000	000100	000000	000000	000000	000000
23	0	000000	000000	000000	000010	000000	000000	000000	000000
24	0	000000	000000	000000	000001	000000	000000	000000	000000

25	0	000000	000000	000000	000000	100000	000000	000000	000000
26	0	000000	000000	000000	000000	010000	000000	000000	000000
27	0	000000	000000	000000	000000	001000	000000	000000	000000
28	0	000000	000000	000000	000000	000100	000000	000000	000000
29	0	000000	000000	000000	000000	000010	000000	000000	000000
30	0	000000	000000	000000	000000	000001	000000	000000	000000
31	0	000000	000000	000000	000000	000000	100000	000000	000000
32	0	000000	000000	000000	000000	000000	010000	000000	000000
33	0	000000	000000	000000	000000	000000	001000	000000	000000
34	0	000000	000000	000000	000000	000000	000100	000000	000000
35	0	000000	000000	000000	000000	000000	000010	000000	000000
36	0	000000	000000	000000	000000	000000	000001	000000	000000
37	0	000000	000000	000000	000000	000000	000000	100000	000000
38	0	000000	000000	000000	000000	000000	000000	010000	000000
39	0	000000	000000	000000	000000	000000	000000	001000	000000
40	0	000000	000000	000000	000000	000000	000000	000100	000000
41	0	000000	000000	000000	000000	000000	000000	000010	000000
42	0	000000	000000	000000	000000	000000	000000	000001	000000
43	0	000000	000000	000000	000000	000000	000000	000000	100000
44	0	000000	000000	000000	000000	000000	000000	000000	010000
45	0	000000	000000	000000	000000	000000	000000	000000	001000
46	0	000000	000000	000000	000000	000000	000000	000000	000100
47	0	000000	000000	000000	000000	000000	000000	000000	000010
48	0	000000	000000	000000	000000	000000	000000	000000	000001

Error Reaction:

logical right shift
test no. <number>
received <48-bit received result>
expected <48-bit expected result>

Complete Test of:

SC:= BUS(11:23) for BUS(11:23) = -1; <ARU064:ARU066-ARU068,174,175>

SC:= SC + 1 for SC = -1; <ARU068,174,089>

lshr ARconBR; <ARU075:ARU076-ARU080:ARU083,ARU098,234>

SB(0) [1] ; <MPC009,123-ARU084,283>

SB < > 0 [1] for SB = -1; <MPC010,124-ARU089>

Partly Test of:

SB $\leq$ - 65 [0] for SB = -1; <MPC002,116-ARU089,373,370,298>

27. ARITHMETIC RIGHT SHIFT

Arithmetic right shifts are similar to logical right shifts; hence two test-patterns are sufficient for verification. The testpatterns are only shifted 1 position.

Testpattern in AR(-1:23):

Test no. 1	0	010000 000000	000000 000000
2	1	100000 000000	000000 000000

Error Reaction:

arithmetic right shift
test no. <number>
received <24-bit received result>
expected <24-bit expected result>

Complete Test of:

ashr ARconBR except for AR(-1); <ARU080:ARU082>

Partly Test of:

SB $\leq$ -65 [0] for SB = -1; <MPC002,116-ARU089,373,370,298>

28. MULTIPLE LEFT SHIFTS

Multiple left shifts are tested for various number of shifts. In this way different micro orders and jump conditions are checked. The original 48-bit contents of the double register before it is shifted are

100000 000000 000000 000000 000000 000000 000000 000001

The number of shifts for each test is listed below. Number of left shifts expressed in binary form for

Test no.	1	000000	000000	000000
	2	000000	000000	000001
	3	000000	000000	000010
	4	000000	000000	000100
	5	000000	000000	001000
	6	000000	000000	010000
	7	000000	000000	100000
	8	000000	000000	000001 000000
	9	000000	000000	000010 000000
	10	000000	000000	000100 000000
	11	000000	000000	001000 000000
	12	000000	000000	010000 000000
	13	000000	000000	100000 000000
	14	000000	000001	000000 000000
	15	000000	000010	000000 000000
	16	000000	000100	000000 000000
	17	000000	001000	000000 000000
	18	000000	010000	000000 000000
	19	000000	100000	000000 000000
	20	000001	000000	000000 000000
	21	000010	000000	000000 000000
	22	000100	000000	000000 000000
	23	001000	000000	000000 000000
	24	010000	000000	000000 000000

25	000000 001000	000000 000001
26	000000 010000	000000 000001

Error Reaction:

multiple left shifts
test no. <number>
received <48-bit received result>
expected <48-bit expected result>

The expected results are for each test:

Test no. 1	100000 000000 000000 000000	000000 000000 000000 000001
2	000000 000000 000000 000000	000000 000000 000000 000010
3	000000 000000 000000 000000	000000 000000 000000 000100
4	000000 000000 000000 000000	000000 000000 000000 010000
5	000000 000000 000000 000000	000000 000000 000100 000000
6	000000 000000 000000 000000	000000 010000 000000 000000
7	000000 000000 000100 000000	000000 000000 000000 000000
8-26	000000 000000 000000 000000	000000 000000 000000 000000

If the results of tests 25 and 26 show that only 1 left shift has been performed, the error is due to the decoding for SB $\geq$ 64.

Complete Test of:

SC:= BUS(11:23)
for BUS = 1,2,4,8,16,32; <ARU064:ARU066-ARU068,174,175>
BUS(0:23):= 48; <ARU031,088-ARU030,308-ARU020:ARU021>
SC:= SC-1 for SC = 1,2,3,...,48; <ARU068,174,89>
SB < > 0 [0] [1]; <MPC003,117-ARU089>
SC < > 0 [1] for SC = 1,2,3,...,48; <MPC010,124-ARU067>

29. MULTIPLE ARITHMETIC RIGHT SHIFTS

Arithmetic right shifts are tested for various number of shifts. In this way different micro orders and jump conditions are checked. The original 48-bit contents of the double register before it is shifted are

Test no.

1-13	100000 000000 000000 000000 000000 000000 000000 000001
14	011111 111111 111111 111111 111111 111111 111111 111111

Number of arithmetic right shifts for

Test no. 1	111111 111111 111111 111111	(- 1)
2	111111 111111 111111 010010	(- 46)
3	111111 111101 111111 111111	
4	111111 111011 111111 111111	
5	111111 110111 111111 111111	
6	111111 101111 111111 111111	
7	111111 011111 111111 111111	
8	111110 111111 111111 111111	
9	111101 111111 111111 111111	
10	111011 111111 111111 111111	
11	110111 111111 111111 111111	
12	101111 111111 111111 111111	
13	100000 000000 000000 000000	(- 2××23)
14	101111 111111 111111 111111	

Error Reaction:

multiple arithmetic right shifts
test no. <number>
received <48-bit received result>
expected <48-bit expected result>

The expected results for each test are:

Test no. 1	110000	000000	000000	000000	000000	000000	000000	000000
2	111111	111111	111111	111111	111111	111111	111111	111110
3-13	111111	111111	111111	111111	111111	111111	111111	111111
14	000000	000000	000000	000000	000000	000000	000000	000000

If only test 1 and properly 2 delivers the result -1, it is most likely that the jump condition, SB $\leq$ -65, is s-a-1. If test 2 does not shift the correct number of times, the SC counter fails to count up. If tests 3 to 14 deliver a result which is only shifted 1 position, the jump condition, SB $\leq$ -65, is s-a-0. Test 14 checks the AR(-1) jump condition for zero value. An interrupt may occur due to test 13 if the jump condition, SB = 0, gives a zero result for SB = -2××23.

Complete Test of:

BUS(0:23):= -1; <ARU029:ARU030,ARU031,218,232,88,157>

SC:= SC + 1 for SC = -1,-2,...,-46; <ARU068,174,089>

SB < > 0 for SB = -2××23; <MPC010,124-ARU089>

SB $\leq$ -65 [0] [1]; <MPC002,116-ARU089,373,370,298>

AR(-1) [0] [1]; <MPC007,121-ARU071,290>

30. NORMALIZE

This test checks the normalization of different numbers. The numbers to be normalized and their expected exponents are listed below.

	Number to be normalized	Expected exponent
Test no. 1	111111 111111 111111 111111	111111 101001 (-23)
2	000000 000000 000000 000000	100000 000000 (-2048)
3	100000 000000 000000 000000	000000 000000 (0)
4	010000 000000 000000 000000	000000 000000 (0)
5	001000 000000 000000 000000	111111 111111 (-1)
6	000100 000000 000000 000000	111111 111110 (-2)
7	000010 000000 000000 000000	111111 111101 (-3)
8	000001 000000 000000 000000	111111 111100 (-4)
9	000000 100000 000000 000000	111111 111011 (-5)
10	000000 010000 000000 000000	111111 111010 (-6)
11	000000 001000 000000 000000	111111 111001 (-7)
12	000000 000100 000000 000000	111111 111000 (-8)
13	000000 000010 000000 000000	111111 110111 (-9)
14	000000 000001 000000 000000	111111 110110 (-10)
15	000000 000000 100000 000000	111111 110101 (-11)
16	000000 000000 010000 000000	111111 110100 (-12)
17	000000 000000 001000 000000	111111 110011 (-13)
18	000000 000000 000100 000000	111111 110010 (-14)
19	000000 000000 000010 000000	111111 110001 (-15)
20	000000 000000 000001 000000	111111 110000 (-16)
21	000000 000000 000000 100000	111111 101111 (-17)
22	000000 000000 000000 010000	111111 101110 (-18)
23	000000 000000 000000 001000	111111 101101 (-19)
24	000000 000000 000000 000100	111111 101100 (-20)
25	000000 000000 000000 000010	111111 101011 (-21)
26	000000 000000 000000 000001	111111 101010 (-22)

Error Reaction:

normalize

test no. <number>

received <24-bit received exponent con
24-bit normalized received result>

expected <24-bit expected exponent con
24-bit normalized expected result>

Complete Test of:

SC:= BUS(11:23) for BUS(11:23) = 0; <ARU064:ARU066-ARU068,174,175>

SC:= SC-1 for SC = 0,-1,-2,...,-22; <ARU068,174,89>

BUS(12:23):= SC(12:23)
for SC = 0,-1,-2,...,-23; <ARU014:ARU025-ARU068,237>

AR < > 0 [1]; <MPC008,122-ARU078>

AR(0) = AR(1) [0] [1]; <MPC004,118-ARU078,222>

AR(1) = AR(2) [0] [1]; <MPC003,117-ARU078,222>

Partly Test of:

BUS(0:11):= 0 ; BUS(12:23):= -2048

AR < > 0 [0] <MPC008,122-ARU078>

31. SB DIAGONAL READ-OUT

The micro order, BUS(0:23):= 12ext0conSB(0:11), is checked by means of the bz instruction. The following testpatterns are used:

Test no. 1	100000	000000	000000	000000
2	010000	000000	000000	000000
3	001000	000000	000000	000000
4	000100	000000	000000	000000
5	000010	000000	000000	000000
6	000001	000000	000000	000000
7	000000	100000	000000	000000
8	000000	010000	000000	000000
9	000000	001000	000000	000000
10	000000	000100	000000	000000
11	000000	000010	000000	000000
12	000000	000001	000000	000000

Error Reaction:

sb diagonal read-out
test no. <number>
received <24-bit received result>
expected <24-bit expected result>

If the results for all tests are zero, the jump condition, HA(23), is s-a-1.

Complete Test of:

BUS(0:23):= 12ext0conSB(0:11); <ARU002:ARU025-ARU090,091>
HA(23) [0]; <MPC007,121-STC003,445>

32. SB(0) EXTENSION

The micro order, SB(0:11):= 12extSB(0), is checked by means of the bl instruction. Two testpatterns are used, namely for

Test no. 1 011111 111111 111111 111111
2 100000 000000 000000 000000

Error Reaction:

sb(0) extension
test no. <number>
received <24-bit received result>
expected <24-bit expected result>

If the received result is 0 for test 1 and -1 for test 2, a race condition exists between the two micro orders, SB(0:11):= 12extSB(0) and SB(12:23):= SB(0:11).

Complete Test of:

SB(0:11):= 12extSB(0); <ARU084:ARU085-ARU090:ARU091>

33. SB DIAGONAL READ-IN

The micro order, SB(0:11):= BUS(12:23), is checked by means of the hs <even addr> instruction. The following testpatterns are used:

Test no. 1	000000 000000	100000 000000
2	000000 000000	010000 000000
3	000000 000000	001000 000000
4	000000 000000	000100 000000
5	000000 000000	000010 000000
6	000000 000000	000001 000000
7	000000 000000	000000 100000
8	000000 000000	000000 010000
9	000000 000000	000000 001000
10	000000 000000	000000 000100
11	000000 000000	000000 000010
12	000000 000000	000000 000001

Error Reaction:

sb diagonal read-in
test no. <number>
received <24-bit received result>
expected <24-bit expected result>

Complete Test of:

SB(0:11):= BUS(12:23); <ARU084:ARU085-ARU090,363,026>

34. LOGICAL AND

The micro order, BUS(0:23):= AND(0:23), has previously been used (and tested), namely in the test REGISTER W1 AND CORE STORE DATA PATHS. In this case, however, the resultant logical product was all zeroes. Now we are interested in the cases where the logical product has one bit equal to one and the rest equal to zero. In other words, we check the result of testpattern ∧ testpattern for the following cases:

Test no. 1	100000	000000	000000	000000
2	010000	000000	000000	000000
3	001000	000000	000000	000000
4	000100	000000	000000	000000
5	000010	000000	000000	000000
6	000001	000000	000000	000000
7	000000	100000	000000	000000
8	000000	010000	000000	000000
9	000000	001000	000000	000000
10	000000	000100	000000	000000
11	000000	000010	000000	000000
12	000000	000001	000000	000000
13	000000	000000	100000	000000
14	000000	000000	010000	000000
15	000000	000000	001000	000000
16	000000	000000	000100	000000
17	000000	000000	000010	000000
18	000000	000000	000001	000000
19	000000	000000	000000	100000
20	000000	000000	000000	010000
21	000000	000000	000000	001000
22	000000	000000	000000	000100
23	000000	000000	000000	000010
24	000000	000000	000000	000001

Error Reaction:

logical and
test no. <number>
received <24-bit received result>
expected <24-bit expected result>

Complete Test of:

BUS(0:23):= AND(0:23); <ARU093:ARU095>

35. REGISTER EX

Register EX is checked for loading and storing by means of the following testpatterns in EX(21:23):

Test no.	1	100
	2	010
	3	001

Error Reaction:

If the test fails due to a transmission error from BUS to EX or vice versa, (the two types of error cannot be distinguished) the error message is:

```
register ex
test no.     <number>
received     <24-bit received result>
expected     <24-bit expected result>
```

Complete Test of:

EX(21:23):= BUS(21:23); <ARU101,435,028-ARU102,440>

BUS(0:23):= 21extOconEX; <ARU030:ARU031-ARU101,435,028>

36. PROTECTION KEY

The protection key of a word in core store is tested by the following test-patterns in PK(0:2):

Test no. 1 100
2 010
3 001

A testpattern is first applied to the register whereafter it is read out a-gain and finally the two bitpatterns are compared.

Error Reaction:

protection key
test no. <number>
received <24-bit received result>
expected <21ext0 con3-bit expected PK>

Complete Test of:

PK:= BUS(21:23); <ARU104,317-ARU105,432,253,187,26>
BUS(0:23):= 21ext0conPK; <ARU030:ARU031-ARU105,187,26>
Split Write for STdata(24:27):= PK
comment data path from PK to core store;
<STC014-STC018>
Read Data for PK:= STdata(24:27)
comment data path from core store to PK;
<STC014-STC020-ARU104,317-ARU105,385>

37. REGISTER PR

The protection register is tested by the following testpatterns:

Test no.	1	10000000
	2	01000000
	3	00100000
	4	00010000
	5	00001000
	6	00000100
	7	00000010
	8	00000001

A testpattern is first applied to the register whereafter it is read out again and finally the two bitpatterns are compared. Note PR(0) is permanently equal to one. Prior to the execution of this test all its instructions are supplied with a key equal to zero.

Error Reaction:

register pr
test no. <number>
received <24-bit received result>
expected <16ext0con1con expected PR(1:7)>

Complete Test of:

PR(1:7):= BUS(17:23); <ARU103-ARU105,237,187,25>
BUS(0:23):= 16ext0conPR; <ARU018:ARU025-ARU105,187,27>

38. SKIP ON PROTECT

The micro order, if -,PROTECT then IC:= IC + 1, is verified by the sp instruction. PROTECT is a decoding network whose value depends on PK and PR. The equation is PROTECT:= PR(PK) = 1. The operation of the network is checked by the following testpatterns:

	PK(0:2)	PR(16:23)	PROTECT
Test no. 1	000	10 000000	1
2	001	11 000000	1
3	010	10 100000	1
4	011	10 010000	1
5	100	10 001000	1
6	101	10 000100	1
7	110	10 000010	1
8	111	10 000001	1
9	001	10 010100	0
10	010	10 010010	0
11	011	11 100001	0
12	100	10 000110	0
13	101	11 001001	0
14	110	10 101001	0
15	111	10 010110	0

Prior to the execution of this test all its instructions are supplied with a protection key equal to zero.

Error Reaction:

If the program does not skip as expected, the test number will be typed out

```
skip on protect
test no.     <number>
received     000000 000000  000000 000000
expected     000000 000000  000000 000000
```

An error is most likely due to the decoding network.

Complete Test of:

if -,PROTECT then IC:= IC + 1; <ARU104,157,241,249,253
-ARU063,168,176>

39. READ FROM W0

The datapath from register W0 to SB, which is controlled by the store controller (STC), is activated when the effective address of store and load instructions equals 0. The datapath is tested by means of the instruction, rl w0 0, where the following testpatterns are employed

W1:= -1 for test no. 1; 2 for test nos. 2 to 25

W2:= -1 for test no. 1; 4 for test nos. 2 to 25

W3:= -1 for test no. 1; 8 for test nos. 2 to 25

and W0 equals for

Test no.	1	000000	000000	000000 000000
	2	100000	000000	000000 000000
	3	010000	000000	000000 000000
	4	001000	000000	000000 000000
	5	000100	000000	000000 000000
	6	000010	000000	000000 000000
	7	000001	000000	000000 000000
	8	000000	100000	000000 000000
	9	000000	010000	000000 000000
	10	000000	001000	000000 000000
	11	000000	000100	000000 000000
	12	000000	000010	000000 000000
	13	000000	000001	000000 000000
	14	000000	000000	100000 000000
	15	000000	000000	010000 000000
	16	000000	000000	001000 000000
	17	000000	000000	000100 000000
	18	000000	000000	000010 000000
	19	000000	000000	000001 000000

20	000000	000000	000000 100000
21	000000	000000	000000 010000
22	000000	000000	000000 001000
23	000000	000000	000000 000100
24	000000	000000	000000 000010
25	000000	000000	000000 000001

Error Reaction:

read from w0
test no. <number>
received <24-bit received result>
expected <24-bit expected result>

1. Received result:= W1, W2, or W3.
   An error in the register selection network of the store controller is discovered.
2. Received result:= constant different from W1, W2, and W3.
   Core store address 0 has properly been selected instead of W0 due to an error in the bistable Addr ST. This situation may lead to a core store error.

Complete Test of:

Read Data for STBUS(0:23):= W0;
comment datapath from W0 to STBUS;
<STC003,445-STC007,499,506-STC010:STC013>

40. READ FROM W1

The datapath from register W1 to SB, which is controlled by the store controller (STC), is activated when the effective address of store and load instructions equals 2. The datapath is tested by means of the instruction,
rl w1 2, where the following testpatterns are employed

W0:= -1 for test no. 1; 1 for test nos. 2 to 25
W2:= -1 for test no. 1; 4 for test nos. 2 to 25
W3:= -1 for test no. 1; 8 for test nos. 2 to 25

and W1 equals for

Test no.	1	000000	000000	000000 000000
	2	100000	000000	000000 000000
	3	010000	000000	000000 000000
	4	001000	000000	000000 000000
	5	000100	000000	000000 000000
	6	000010	000000	000000 000000
	7	000001	000000	000000 000000
	8	000000	100000	000000 000000
	9	000000	010000	000000 000000
	10	000000	001000	000000 000000
	11	000000	000100	000000 000000
	12	000000	000010	000000 000000
	13	000000	000001	000000 000000
	14	000000	000000	100000 000000
	15	000000	000000	010000 000000
	16	000000	000000	001000 000000
	17	000000	000000	000100 000000
	18	000000	000000	000010 000000
	19	000000	000000	000001 000000

20	000000	000000	000000	100000
21	000000	000000	000000	010000
22	000000	000000	000000	001000
23	000000	000000	000000	000100
24	000000	000000	000000	000010
25	000000	000000	000000	000001

Error Reaction:

read from w1
test no. <number>
received <24-bit received result>
expected <24-bit expected result>

1. Received result:= W0, W2, or W3.
   An error in the register selection network of the store controller is discovered.
2. Received result:= constant different from W0, W2, and W3.
   Core store address 2 has properly been selected instead of W1 due to an error in the bistable Addr ST. This situation may lead to a core store error.

Complete Test of:

Read Data for STBUS(0:23):= W1;
comment datapath from W1 to STBUS;
<STC003,445-STC007,499,506-STC010:STC013>

41. READ FROM W2

The datapath from register W2 to SB, which is controlled by the store controller (STC), is activated when the effective address of store and load instructions equals 4. The datapath is tested by means of the instruction, rl w2 4, where the following testpatterns are employed

W0:= -1 for test no. 1; 1 for test nos. 2 to 25

W1:= -1 for test no. 1; 2 for test nos. 2 to 25

W3:= -1 for test no. 1; 8 for test nos. 2 to 25

and W2 equals for

Test no.	1	000000	000000	000000 000000
	2	100000	000000	000000 000000
	3	010000	000000	000000 000000
	4	001000	000000	000000 000000
	5	000100	000000	000000 000000
	6	000010	000000	000000 000000
	7	000001	000000	000000 000000
	8	000000	100000	000000 000000
	9	000000	010000	000000 000000
	10	000000	001000	000000 000000
	11	000000	000100	000000 000000
	12	000000	000010	000000 000000
	13	000000	000001	000000 000000
	14	000000	000000	100000 000000
	15	000000	000000	010000 000000
	16	000000	000000	001000 000000
	17	000000	000000	000100 000000
	18	000000	000000	000010 000000
	19	000000	000000	000001 000000

20	000000	000000	000000 100000
21	000000	000000	000000 010000
22	000000	000000	000000 001000
23	000000	000000	000000 000100
24	000000	000000	000000 000010
25	000000	000000	000000 000001

Error Reaction:

read from w2
test no. <number>
received <24-bit received result>
expected <24-bit expected result>

1. Received result:= W0, W1, or W3.
   An error in the register selection network of the store controller is discovered.
2. Received result:= constant different from W0, W1, and W3.
   Core store address 4 has properly been selected instead of W2 due to an error in the bistable Addr ST. This situation may lead to a core store error.

Complete Test of:

Read Data for STBUS(0:23):= W2;
comment datapath from W2 to STBUS;
<STC003,445-STC007,499,506-STC010:STC013>

42. READ FROM W3

The datapath from register W3 to SB, which is controlled by the store controller (STC), is activated when the effective address of store and load instructions equals 5. The datapath is tested by means of the instruction, rl w3 6, where the following testpatterns are employed

W0:= -1 for test no. 1; 1 for test nos. 2 to 25

W1:= -1 for test no. 1; 2 for test nos. 2 to 25

W2:= -1 for test no. 1; 4 for test nos. 2 to 25

and W3 equals for

Test no.	1	000000	000000	000000 000000
	2	100000	000000	000000 000000
	3	010000	000000	000000 000000
	4	001000	000000	000000 000000
	5	000100	000000	000000 000000
	6	000010	000000	000000 000000
	7	000001	000000	000000 000000
	8	000000	100000	000000 000000
	9	000000	010000	000000 000000
	10	000000	001000	000000 000000
	11	000000	000100	000000 000000
	12	000000	000010	000000 000000
	13	000000	000001	000000 000000
	14	000000	000000	100000 000000
	15	000000	000000	010000 000000
	16	000000	000000	001000 000000
	17	000000	000000	000100 000000
	18	000000	000000	000010 000000
	19	000000	000000	000001 000000

20	000000	000000	000000	100000
21	000000	000000	000000	010000
22	000000	000000	000000	001000
23	000000	000000	000000	000100
24	000000	000000	000000	000010
25	000000	000000	000000	000001

Error Reaction:

read from w3
test no. <number>
received <24-bit received result>
expected <24-bit expected result>

1. Received result:= W0, W1, or W2.
   An error in the register selection network of the store controller is discovered.
2. Received result:= constant different from W0, W1, and W2.
   Core store address 6 has properly been selected instead of W3 due to an error in the bistable Addr ST. This situation may lead to a core store error.

Complete Test of:

Read Data for STBUS(0:23):= W3;
comment datapath from W3 to STBUS;
<STC003,445-STC007,499,506-STC010:STC013>

43. READ DOUBLE

The datapath from BR to the address register of the core store is checked by means of dl instructions. The contents of BR for the individual tests are for:

Test no. 1	000000 000101	110111 000000	
2	000000 000110	010100 100110	
3	000000 000111	001010 010100	
4	000000 000011	110000 111000	
5	000000 000000	000000 000010	(selection of W1)
6	000000 000000	000000 000100	(selection of W2)
7	111111 111111	111111 111111	(selection of W3)

The addresses of the first four tests are identical to the addresses used in RELATIVE ADDRESSING AND JUMP.

Error Reaction:

read double
test no. <number>
received <24-bit received result>
expected 111110 001111 011111 111000

The test no. defines the address which has not been interpreted correctly. The received result has no meaning.

Complete Test of:

Read Data Double for STaddr:= BR(6:22) 4 <= SB < 32K bytes
and HA(21:22):= BR(21:22)
comment address selection from BR;
<STC003-STC008:STC009>

44. INSTRUCTIONS IN W

Instructions are loaded into the w registers W0, W1, and W2, namely

```
W0:   al  w0     0
W1:   rl  w0     b5+22
W2:   jl     x3  0
W3:   contains return address
```

followed by a jump to W0.

Error Reaction:

```
instructions in w
test no.      1
received      <24-bit received W0>
expected      111110 001111  011111 111000
```

Complete Test of:

```
ICaddrST   for  0 <= ICcon0 < 8 bytes;
<STC008,297-ARU049>
```

45. WRITE INTO W REGISTERS

The rs instruction with an effective address of 0, 2, 4, or 6 invokes the store controller (STC) for data transfer from SB to the appropriate W registers. Four tests are necessary, namely

```
Test no. 1     rs  w0     6   ;   transfer from SB to W3
         2     rs  w1     4   ;   transfer from SB to W2
         3     rs  w2     2   ;   transfer from SB to W1
         4     rs  w3     0   ;   transfer from SB to W0
```

Before each test, the contents of W0, W1, W2, and W3 are set to 1, 2, 4, and 8, respectively. The contents of W registers before and after the execution are seen from the table below.

	test no. 1		test no. 2		test no. 3		test no. 4	
	before	after	before	after	before	after	before	after
W0	1	1	1	1	1	1	1	8
W1	2	2	2	2	2	4	2	2
W2	4	4	4	2	4	4	4	4
W3	8	1	8	8	8	8	8	8

Error Reaction:

```
write into w registers
test no.      <number>
received      <24-bit received result>
expected      <24-bit expected result>
```

More than one error message occurs for a particular test number if the store controller writes into more than one W register.

Complete Test of:

```
Split Write   for BUS(0:23):= SB; W(fr):= BUS(0:23)
  comment datapath from SB to W(fr);
<STC007,438,444-ARU059-ARU090,237>
```

46. REGISTER PB0

PB0 is the protection key belonging to register W0. PB0 itself and its incoming and outgoing datapaths are tested. The datapath, PBUS(0:2), from PK to PB0 is activated by the instruction

```
ks    0
```

whereas data transfer from PB0 to PK takes place via STBUS(24:26) supervised by the store controller. The latter transfer is activated for

```
kl    0
```

The testpatterns for PB0 are:

```
Test no. 1    100
         2    010
         3    001
```

Error Reaction:

```
pb0
test no.     <number>
received     <21ext0 con received PB0>
expected     <21ext0 con expected PB0>
```

Complete Test of:

```
Split Write  for PB0:= PK; <ARU105,187-ARU106,187,253-ARU107,319
                                                    -ARU109,382>
Read Data    for PK:= PB0; <STC014,383-ARU104,317>
```

47. REGISTER PB1

PB1 is the protection key belonging to register W1. PB1 itself and its incoming and outgoing datapaths are tested. The datapath, PBUS(0:2), from PK to PB1 is activated by the instruction

```
ks    2
```

whereas data transfer from PB1 to PK takes place via STBUS(24:26) supervised by the store controller. The latter transfer is activated for

```
kl    2
```

The testpatterns for PB1 are:

Test no.	1	100
	2	010
	3	001

Error Reaction:

```
pb1
test no.      <number>
received      <21ext0 con received PB1>
expected      <21ext0 con expected PB1>
```

Complete Test of:

```
Split Write   for PB1:= PK; <ARU105,187-ARU106,187,253-ARU107,319
                                                    -ARU109,382>
Read Data     for PK:= PB1; <STC014,383-ARU104,317>
```

48. REGISTER PB2

PB2 is the protection key belonging to register W2. PB2 itself and its incoming and outgoing datapaths are tested. The datapath, PBUS(0:2), from PK to PB2 is activated by the instruction

```
ks    4
```

whereas data transfer from PB2 to PK takes place via STBUS(24:26) supervised by the store controller. The latter transfer is activated for

```
kl    4
```

The testpatterns for PB2 are:

Test no.	1	100
	2	010
	3	001

Error Reaction:

```
pb2
test no.      <number>
received      <21ext0 con received PB2>
expected      <21ext0 con expected PB2>
```

Complete Test of:

```
Split Write   for PB2:= PK; <ARU105,187-ARU106,187,253-ARU108,320
                                                    -ARU109,382>
Read Data     for PK:= PB2; <STC014,383-ARU104,317>
```

49. REGISTER PB3

PB3 is the protection key belonging to register W3. PB3 itself and its incoming and outgoing datapaths are tested. The datapath, PBUS(0:2), from PK to PB3 is activated by the instruction

```
ks    6
```

whereas data transfer from PB3 to PK takes place via STBUS(24:26) supervised by the store controller. The latter transfer is activated for

```
kl    6
```

The testpatterns for PB3 are:

```
Test no. 1    100
         2    010
         3    001
```

Error Reaction:

```
pb3
test no.     <number>
received     <21ext0 con received PB3>
expected     <21ext0 con expected PB3>
```

Complete Test of:

```
Split Write  for PB3:= PK; <ARU105,187-ARU106,187,253-ARU108,320
                                                    -ARU109,382>
Read Data    for PK:= PB3; <STC014,383-ARU104,317>
```

50. REGISTER AE

Data transfer to and from the 12 most significant bits of register AE is checked by means of the fa instruction. The two exponents of the operands are chosen in such a way that the exponent difference (SC) is > = 38. This implies that no addition takes place for which reason the expected result equals the contents of the operand stored in WOconW1. In details, the test works as follows:

```
        fa  w1     addr
```

where

```
        WO: 101111 000111  111000 000001
        W1: depends on test number
    addr-2: 101111 000000  000000 000000
      addr: 111111 100000  000000 000000
```

The testpatterns equal for

```
Test no. 1     000000 000000  000000 111111
         2     100000 000000  000000 111111
         3     010000 000000  000000 111111
         4     001000 000000  000000 111111
         5     000100 000000  000000 111111
         6     000010 000000  000000 111111
         7     000001 000000  000000 111111

         8     000000 100000  000000 111111
         9     000000 010000  000000 111111
        10     000000 001000  000000 111111
        11     000000 000100  000000 111111
        12     000000 000010  000000 111111
        13     000000 000001  000000 111111
```

Error Reaction:

register ae
test no. <number>
received <48-bit received result>
expected <48-bit expected result>

If the not tested jump conditions, SC > -38 ∧ SC < 38 and SC(11), do not operate satisfactorily, the following results can be expected:

SC>-38∧SC<38	SC(11)	Received Result
0	0	The jump conditions are correct and the received result should be the correct one provided the new micro orders are executed correctly.
0	1	101111 000000 000000 000000 111111 100000 000000 000000
1	0	101111 000111 111000 00000x xxxxxx xxxxxx 000000 111111
1	1	101111 000000 000000 000000 111111 011111 000000 000000

Complete Test of:

AE:= BUS(0:11)con0con0 except for AE(12,13); <ARU073:ARU074-ARU082,236>
BUS(0:11):= if EX(21) = 0 then AE(0:11)
else AR(0:9,9,9) for EX(21) = 0;
<ARU029:ARU030-ARU082,190>
SC(11) [0] ; <MPC009,123-ARU064,166>

Partly Test of:

BUS(0:11):= 12extW(1)(12) for W(1)(12) = 0; <ARU002:ARU013
-ARU060,281,213>
SE:= SB(0:11)con0con0, SB(0:11):= 12extSB(12) only the last micro order has influence on the final result.
SC > -38 ∧ SC < 38 [0] for SC = 63; <MPC008,122-ARU067>

51. REGISTER SE

Data transfer to and from the 12 most significant bits of register SE is checked by means of the fa instruction. The two exponents of the operands are chosen in such a way that the exponent difference (SC) is < = -38. This implies that no addition takes place for which reason the expected result equals the contents of the operand stored in addr-2 and addr. In details, the test works as follows:

```
        fa  w1     addr
```

where

```
        W0: 101111 000000  000000 000000
        W1: 111111 100000  000000 000000
    addr-2: 101111 000111  111000 000001
      addr: depends on test number
```

The testpatterns equal for

Test no. 1	000000	000000	111111
2	100000	000000	111111
3	010000	000000	111111
4	001000	000000	111111
5	000100	000000	111111
6	000010	000000	111111
7	000001	000000	111111
8	000000	100000	111111
9	000000	010000	111111
10	000000	001000	111111
11	000000	000100	111111
12	000000	000010	111111
13	000000	000001	111111

Error Reaction:

register se
test no. <number>
received <48-bit received result>
expected <48-bit expected result>

If the not tested jump conditions, SC > -38 ∧ SC < 38 and SC(11), do not operate satisfactorily, the following results can be expected:

SC>-38∧SC<38	SC(11)	Received Result
0	0	101111 000000 000000 000000 111111 100000 000000 000000
0	1	The jump conditions are correct and the received result should be the correct one provided the new micro orders are executed correctly.
1	0	101111 000000 000000 000000 111111 011111 000000 000000
1	1	101111 000111 111000 00000x xxxxxx xxxxxx 000000 111111

Complete Test of:

SE:= SB(0:11)con0con0, SB(0:11):= 12extSB(12) except for SE(12,13);
<ARU084:ARU085-ARU087:ARU088-ARU090,363,370-ARU091,432,370,374>
BUS(0:11):= SE(0:11); <ARU002:ARU013-ARU091,212>
SC(11) [1] ; <MPC009,123-ARU064,166>

Partly Test of:

BUS(0:11):= 12extW(1)(12) for W(1)(12) = 0; <ARU002:ARU013
-ARU060,281,213>
SC > -38 ∧ SC < 38 [0] for SC = -63; <MPC008,122-ARU067>

52. FLOATING ADDER

The adder and carry circuitry for floating arithmetic is investigated in this test. Since integer arithmetic is already known to operate, we shall here confine our investigations to the mantissae bits 12 to 35. The testpatterns for each test are listed below (all operand exponents are chosen to be zero). Tests 21 and 22 are included with the sole purpose of testing for interchanged bit positions.

```
Test no. 1   AE:= 00000000  0000  00
             SE:= 00000000  0000  00
       AE + SE:= 00000000  0000  00

Test no. 2   AE:= 00000000  0000  00
             SE:= 11111111  1111  00
       AE + SE:= 11111111  1111  00

Test no. 3   AE:= 11111111  1111  00
             SE:= 00000000  0000  00
       AE + SE:= 11111111  1111  00

Test no. 4   AE:= 01010101  0101  00
             SE:= 01010101  0101  00
       AE + SE:= 10101010  1010  00

Test no. 5   AE:= 00000000  1000  00
             SE:= 11111110  1000  00
       AE + SE:= 11111111  0000  00

Test no. 6   AE:= 00000001  1000  00
             SE:= 11111101  1000  00
       AE + SE:= 11111111  0000  00

Test no. 7   AE:= 00000011  1000  00
             SE:= 11111011  1000  00
       AE + SE:= 11111111  0000  00
```

```
Test no. 8    AE:= 00000111  1000  00
              SE:= 11110111  1000  00
         AE + SE:= 11111111  0000  00

Test no. 9    AE:= 00001111  0001  00
              SE:= 11101111  1111  00
         AE + SE:= 11111111  0000  00

Test no. 10   AE:= 00011111  0010  00
              SE:= 11011111  1110  00
         AE + SE:= 11111111  0000  00

Test no. 11   AE:= 00111111  0100  00
              SE:= 10111111  1100  00
         AE + SE:= 11111111  0000  00

Test no. 12   AE:= 01111111  1000  00
              SE:= 01111111  1000  00
         AE + SE:= 11111111  0000  00

Test no. 13   AE:= 00000000  0000  00
              SE:= 00000000  0000  00
         AE - SE:= 00000000  0000  00

Test no. 14   AE:= 11111111  1111  00
              SE:= 00000000  0000  00
         AE - SE:= 11111111  1111  00

Test no. 15   AE:= 11111111  1111  00
              SE:= 11111111  1111  00
         AE - SE:= 00000000  0000  00

Test no. 16   AE:= 10101010  1010  00
              SE:= 01010101  0101  00
         AE - SE:= 01010101  0101  00

Test no. 17   AE:= 00000001  0111  00
              SE:= 00000000  1000  00
         AE - SE:= 00000000  1111  00
```

```
Test no. 18  AE:= 00000010  0111  00
             SE:= 00000001  1000  00
          AE - SE:= 00000000  1111  00

Test no. 19  AE:= 00000100  0111  00
             SE:= 00000011  1000  00
          AE - SE:= 00000000  1111  00

Test no. 20  AE:= 00001000  0111  00
             SE:= 00000111  1000  00
          AE - SE:= 00000000  1111  00

Test no. 21  AE:= 00010000  0000  00
             SE:= 00001111  0001  00
          AE - SE:= 00000000  1111  00

Test no. 22  AE:= 00100000  0001  00
             SE:= 00011111  0010  00
          AE - SE:= 00000000  1111  00

Test no. 23  AE:= 01000000  0011  00
             SE:= 00111111  0100  00
          AE - SE:= 00000000  1111  00

Test no. 24  AE:= 10000000  0111  00
             SE:= 01111111  1000  00
          AE - SE:= 00000000  1111  00

Test no. 25  AE:= 00000000  0000  00
             SE:= 00001111  0000  00
          AE + SE:= 00001111  0000  00

Test no. 26  AE:= 00000000  0000  00
             SE:= 00110011  0011  00
          AE + SE:= 00110011  0011  00
```

Error Reaction:

floating adder
test no. <number>
received <received 36-bit mantissa con 12-bit exponent>
expected <expected 36-bit mantissa con 12-bit exponent>

Received result is:

1. equal to one of the two matissae for all tests.
   This signifies that the jump condition, SC > -38 ∧ SC < 38, does not become 1 for SC = 0. Hence the microprogram believes that the exponent difference of the two floating numbers is greater than |38|, which explains the result.
2. equal to half of the expected results for all tests.
   In this case AR(-1) = AR(0) is 0 instead of 1 and, as this condition signifies overflow, the calculated sum or difference shall be shifted arithmetically one position towards left.
3. equal to the expected result + 1 added to the least significant bit of the mantissa for all tests.
   The jump condition Round is simply 1, i.e. rounding has taken place.

Complete Test of:

Adder:= b 1010 000 ; Adder:= b 0101 001; <ARU098:ARU100>
AF(-1:37):= if -,MC(10) then SUM(-1:37) else SUM(-1:35)con0con0
for MC(10) = 0, SUM(37) is not tested;
<ARU073:ARU074-ARU081,311,355-ARU082,236,232>
SC > -38 ∧ SC < 38 [1] for SC = 0; <MPC008,122-ARU067>
AR(-1) = AR(0) [1] for AR(-1,0) = 0; <MPC002,116-ARU078,222>
Round [0] for AF(-1:1,35:37):= b 001 000 or
b 001 100;
<MPC125,011-ARU079,213,365>

Partly Test of:

BUS(0:11):= 12extW(1)(12) for W(1)(12) = 0; <ARU002:ARU013
-ARU060,281,213>

53. W(12) EXTENSION

The extension micro order, 12extW(p)(12) for p = 0,1,2,3, is tested by adding a floating point number, f, to itself.

```
Test nos. 1 and 2:     fa  w0     0   ; test of W0(12) extension
          3 and 4:     fa  w1     2   ; test of w1(12) extension
          5 and 6:     fa  w2     4   ; test of W2(12) extension
          7 and 8:     fa  w3     6   ; test of W3(12) extension
```

For odd test numbers

f = 001000 000000 000000 000000 000000 000000 100000 000000

and for even test numbers

f = 001000 000000 000000 000000 111111 111111 011111 111111

Error Reaction:

w(12) extension
test no. <number>
received <received 36-bit mantissa con 12-bit exponent>
expected <expected 36-bit mantissa con 12-bit exponent>

Complete Test of:

BUS(0:11):= 12extW(fr)(12); <ARU002:ARU013-ARU060,281,213,231>

54. REGISTER SC

Data transfer to and from SC is investigated. The two floating point numbers have equal exponent for all tests. In details, the test works as follows:

fa w1 addr

where

W0: 010000 000000 000000 000000
W1: depends on test number
addr-2: 000000 000000 000000 000000
addr: depends on test number

Testpatterns equal for

Test no. 1	111111	111111	011111 111111
2	000000	000000	100000 000000
3	000000	000000	010000 000000
4	000000	000000	001000 000000
5	000000	000000	000100 000000
6	000000	000000	000010 000000
7	000000	000000	000001 000000
8	000000	000000	000000 100000
9	000000	000000	000000 010000
10	000000	000000	000000 001000
11	000000	000000	000000 000100
12	000000	000000	000000 000010
13	000000	000000	000000 000001

Error Reaction:

register se
test no. <number>
received <received 36-bit mantissa con 12-bit exponent>
expected <expected 36-bit mantissa con 12-bit exponent>

Received result has error in

1. exponent part.
   The corresponding bit in register SC is not able to either receive from BUS or deliver to BUS the expected exponent.
2. mantissa part.
   This signifies properly that the exponent difference, which should be equal to zero, has been calculated incorrectly; properly due to an error in the extension micro order.

Complete Test of:

SC:= BUS(11:23) except for SC(11); <ARU064:ARU066-ARU068,174,175>

BUS(0:23):= 12extOconSC(12:23); <ARU002:ARU025-ARU068,237>

55. CONSTANT 23

Constant 23 is checked by converting an integer to a floating point number. The test is:

```
ci  w0     0
```

where

```
W0: 010000 000000  000000 000000
```

and this leads to a floating point number having an exponent equal to 23.

Error Reaction:

```
constant 23
test no.      1
received      <12 least significant mantissa bits con received expo-
               nent>
expected      000000 000000  000000 010111
```

Complete Test of:

```
BUS(0:23):= 23; <ARU031,157,88>
```

56. ARITHMETIC RIGHT SHIFT IN AF

Since arithmetic shifts in AR are tested already, we only have to concentrate on shifts in AE and, of course, the special case where AR(23) is shifted into AE(0). Bits AF(-1,35:37) are not included in the test, but are tested separately. In details, the test works as follows:

```
        fa  w1      addr
```

where

```
        W0: depends on test number
        W1: depends on test number
    addr-2: 010000 000000  000000 000000
      addr: 000000 000000  000000 000001
```

The testpatterns for W0conW1 equal for

Test no.	1	000000	000000	000000	000001	000000	000000	000000	000000
	2	000000	000000	000000	000000	100000	000000	000000	000000
	3	000000	000000	000000	000000	010000	000000	000000	000000
	4	000000	000000	000000	000000	001000	000000	000000	000000
	5	000000	000000	000000	000000	000100	000000	000000	000000
	6	000000	000000	000000	000000	000010	000000	000000	000000
	7	000000	000000	000000	000000	000001	000000	000000	000000
	8	000000	000000	000000	000000	000000	100000	000000	000000
	9	000000	000000	000000	000000	000000	010000	000000	000000
	10	000000	000000	000000	000000	000000	001000	000000	000000
	11	000000	000000	000000	000000	000000	000100	000000	000000
	12	000000	000000	000000	000000	000000	000010	000000	000000

The mantissa in W0conW1(0:11) is the testpattern to be shifted one shift in AF.

Error Reaction:

```
        arithmetic right shift in af
        test no.      <number>
        received      <48-bit received result>
        expected      <48-bit expected result>
```

1. If the received result equals

   010000 000000 000000 000000 000000 000000 000000 000000

   i.e. the operand in store for all tests then the jump condition SC > -38 ∧ SC < 38 is s-a-0 for SC = -1.
2. Other errors are due to the micro order ashr AF.

Complete Test of:

ashrAF except for AC(-1,35:37); <ARU073:ARU074 -ARU080:082>

SC > -38 ∧ SC < 38 [1] for SC = -1; <MPC008,122-ARU067>

57. ARITHMETIC RIGHT SHIFT IN SF

All bits in SF, except for SF(35:37), are tested for arithmetic right shifts. In details, the test works as follows:

fa w1 addr

where

W0: 110000 000000 000000 000000 (test no. 1)
W1: 000000 000000 000000 000001
addr-2: depends on test number
addr: depends on test number

W0: 100000 000000 000000 000000 (test nos. 2 to 35)
W1: 000000 000000 000000 000001
addr-2: depends on test number
addr: depends on test number

The testpatterns for addr-2 and addr equal for

Test no. 1	100000 000000 000000 000000	000000 000000 000000 000000
2	010000 000000 000000 000000	000000 000000 000000 000000
3	001000 000000 000000 000000	000000 000000 000000 000000
4	000100 000000 000000 000000	000000 000000 000000 000000
5	000010 000000 000000 000000	000000 000000 000000 000000
6	000001 000000 000000 000000	000000 000000 000000 000000
7	000000 100000 000000 000000	000000 000000 000000 000000
8	000000 010000 000000 000000	000000 000000 000000 000000
9	000000 001000 000000 000000	000000 000000 000000 000000
10	000000 000100 000000 000000	000000 000000 000000 000000
11	000000 000010 000000 000000	000000 000000 000000 000000
12	000000 000001 000000 000000	000000 000000 000000 000000

13	000000	000000	100000	000000	000000	000000	000000	000000
14	000000	000000	010000	000000	000000	000000	000000	000000
15	000000	000000	001000	000000	000000	000000	000000	000000
16	000000	000000	000100	000000	000000	000000	000000	000000
17	000000	000000	000010	000000	000000	000000	000000	000000
18	000000	000000	000001	000000	000000	000000	000000	000000
19	000000	000000	000000	100000	000000	000000	000000	000000
20	000000	000000	000000	010000	000000	000000	000000	000000
21	000000	000000	000000	001000	000000	000000	000000	000000
22	000000	000000	000000	000100	000000	000000	000000	000000
23	000000	000000	000000	000010	000000	000000	000000	000000
24	000000	000000	000000	000001	000000	000000	000000	000000
25	000000	000000	000000	000000	100000	000000	000000	000000
26	000000	000000	000000	000000	010000	000000	000000	000000
27	000000	000000	000000	000000	001000	000000	000000	000000
28	000000	000000	000000	000000	000100	000000	000000	000000
29	000000	000000	000000	000000	000010	000000	000000	000000
30	000000	000000	000000	000000	000001	000000	000000	000000
31	000000	000000	000000	000000	000000	100000	000000	000000
32	000000	000000	000000	000000	000000	010000	000000	000000
33	000000	000000	000000	000000	000000	001000	000000	000000
34	000000	000000	000000	000000	000000	000100	000000	000000
35	000000	000000	000000	000000	000000	000010	000000	000000

The mantissa stored in addr-2 and addr is the testpattern to be shifted in SF.

Error Reaction:

arithmetic right shift in sf
test no. <number>
received <48-bit received result>
expected <48-bit expected result>

1. If the received result equals for test no. 1

   110000 000000 000000 000000 000000 000000 000000 000001

   and for tests nos. 2 to 35

   100000 000000 000000 000000 000000 000000 000000 000001

   then the jump condition SC > -38 ∧ SC < 38 is s-a-0 for SC = 1.
2. The last bit in the mantissa equals 1 signifies that rounding is always executed implying that Round is s-a-1.
3. Other errors are due to the micro order ashrSF.

Complete Test of:

ashrSF except for SF(35:37; <ARU084:ARU087-ARU091,26,374,370,432>

SC > -38 ∧ SC < 38 [1] for SC = 1; <MPC008,122-ARU067>

AR(-1) = AR(0) [1] for AR(-1,0) = b11; <MPC002,116-ARU078,222>

Round [0] for AF(-1:1,35:37):= b 110000 or b 110100;

<MPC125,011-ARU079,213,365>

58. AF < > 0

The jump condition, AF < > 0, is tested except for AF(36,37). The test works by converting (cf instruction) the following floating point numbers to integer numbers. The floating point number to be tested is stored in W0conW1

Test no. 1	000000 000000 000000 000001	000000 000000 000000 111111
2	000000 000000 000000 000000	100000 000000 000000 111111
3	000000 000000 000000 000000	010000 000000 000000 111111
4	000000 000000 000000 000000	001000 000000 000000 111111
5	000000 000000 000000 000000	000100 000000 000000 111111
6	000000 000000 000000 000000	000010 000000 000000 111111
7	000000 000000 000000 000000	000001 000000 000000 111111
8	000000 000000 000000 000000	000000 100000 000000 111111
9	000000 000000 000000 000000	000000 010000 000000 111111
10	000000 000000 000000 000000	000000 001000 000000 111111
11	000000 000000 000000 000000	000000 000100 000000 111111
12	000000 000000 000000 000000	000000 000010 000000 111111
13	000000 000000 000000 000000	000000 000001 000000 111111
14	000000 000000 000000 000000	000000 000000 000000 111111

The floating point numbers in tests 1 to 13 are too big to be represented by integer numbers, for which reason no conversion takes place and the contents of W0conW1 are unaltered. On the contrary, the zero mantissa in test 14 is converted to an integer zero, i.e. W1:= 0. With the chosen floating point numbers the conversion is executed if, and only if, AF = 0. In other words

if AF < > 0 then W0 and W1 are unaltered;
if AF = 0 then W0 is unaltered and W1:= 0;
actually W1:= W0 where W0 is 0

Error Reaction:

af < > 0

test no. <number>

received <24-bit received result in W1>

expected <24-bit expected result in W1>

1. Received result is W0 for some tests.
   If this happens for a test, the jump condition AF < > 0 is s-a-0 for the corresponding mantissa.
2. Only test 14 fails.
   The jump condition AF < > 0 is s-a-1.
3. All tests except test 14 fail.
   The jump condition AF < > 0 is either s-a-0 for all bitpatterns or the micro order BUS(0:23):= 23 does not generate the number 23 but some other number.

Complete Test of:

AF < > 0 [0] [1] except for AF(36,37);

<MPC008,122-ARU079,300-ARU078,356,365>

59. LOGICAL LEFT SHIFT IN AF

Since logical left shifts in AR are tested already, we only have to concentrate upon shifts in AE and, of course, the special case where AE(0) is shifted into AR(23). Bits AF(35:37) are not included in the test. In details, the test works as follows:

fa w1 addr

where

WO: depends on test number

W1: depends on test number

addr-2: 001000 000000 000000 000000

addr: 000000 000000 000000 000001

The testpatterns for WOconW1 equal for

Test no.	1	000000	000000	000000	000000	100000	000000	000000	000001
	2	000000	000000	000000	000000	010000	000000	000000	000001
	3	000000	000000	000000	000000	001000	000000	000000	000001
	4	000000	000000	000000	000000	000100	000000	000000	000001
	5	000000	000000	000000	000000	000010	000000	000000	000001
	6	000000	000000	000000	000000	000001	000000	000000	000001
	7	000000	000000	000000	000000	000000	100000	000000	000001
	8	000000	000000	000000	000000	000000	010000	000000	000001
	9	000000	000000	000000	000000	000000	001000	000000	000001
	10	000000	000000	000000	000000	000000	000100	000000	000001
	11	000000	000000	000000	000000	000000	000010	000000	000001
	12	000000	000000	000000	000000	000000	000001	000000	000001

The mantissa in WOconW1 is the testpattern to be shifted one shift in AF.

Error Reaction:

logical left shift in af
test no. <number>
received <48-bit received result>
expected <48-bit expected result>

1. Received mantissa = expected mantissa + 2**-35.
   The jump condition Round is s-a-1.
2. Other errors are due to the micro order lshl AF.

Complete Test of:

lshl AF except AF(36,37); <ARU080:ARU082>
Round [0] for AF(-1:1,35:37):= b 000000 or b 000100;
<MPC125,011-ARU079,213,365>

60. BITS (35,36,37)

The floating point instructions (floating add and subtract) use micro orders and jump conditions which depend on bits 35,36, and 37 of register AF and SF. In order to facilitate the error detection, we have for each test number listed the corresponding (mantissae and exponents) plus the micro orders and jump conditions which have not been tested previously.

	(-1:0) (1:4)	(33:37)	exp	Test of micro orders
Test no. 1				
AF:	0 0 . 1 0 0 0	0 0 1 0 0	0	
SF:	0 0 . 0 0 0 0	0 0 0 0 0	1	
1ashr AF:	0 0 . 0 1 0 0	0 0 0 1 0	0	ashr AF
AF:= AF+SF:	0 0 . 0 1 0 0	0 0 0 1 0	1	adder:= b 1010000
Result:=				
1lshl AF:	0 0 . 1 0 0 0	0 0 1	0	lshl AF
Test no. 2				
AF:	0 0 . 1 0 0 0	0 0 1 0 0	0	
SF:	0 0 . 0 0 0 0	0 0 0 0 0	2	
2ashr AF:	0 0 . 0 0 1 0	0 0 0 0 1	2	ashr AF
AF:= AF+SF:	0 0 . 0 0 1 0	0 0 0 0 1	2	adder:= b 1010000
Result:=				
2lshl AF:	0 0 . 1 0 0 0	0 0 1	0	lshl AF
Test no. 3				
AF:	0 0 . 1 0 0 0	0 0 1 0 0	0	
SF:	0 0 . 0 0 0 0	0 0 0 0 0	3	
3ashr AF:	0 0 . 0 0 0 1	0 0 0 0 0	3	ashr AF
AF:= AF+SF:	0 0 . 0 0 0 1	0 0 0 0 0	3	
Result:=				
3lshl AF:	0 0 . 1 0 0 0	0 0 0	0	lshl AF

	(-1:0)	(1:4)	(33:37)	exp	Test of micro orders
Test no. 4					
AF:	0 0 .	0 0 0 0	0 0 0 0 0	1	
SF:	0 0 .	1 0 0 0	0 0 1 0 0	0	
1ashr SF:	0 0 .	0 1 0 0	0 0 0 1 0	1	ashr SF
AF:= AF+SF:	0 0 .	0 1 0 0	0 0 0 1 0	1	adder:= b 1010000
Result:=					
1lshl AF:	0 0 .	1 0 0 0	0 0 1	0	lshl AF
Test no. 5					
AF:	0 0 .	0 0 0 0	0 0 0 0 0	2	
SF:	0 0 .	1 0 0 0	0 0 1 0 0	0	
2ashr SF:	0 0 .	0 0 1 0	0 0 0 0 1	2	ashr SF
AF:= AF+SF:	0 0 .	0 0 1 0	0 0 0 0 1	2	adder:= b 1010000
Result:=					
2lshl AF:	0 0 .	1 0 0 0	0 0 1	0	lshl AF
Test no. 6					
AF:	0 0 .	0 0 0 0	0 0 0 0 0	3	
SF:	0 0 .	1 0 0 0	0 0 1 0 0	0	
3ashr SF:	0 0 .	0 0 0 1	0 0 0 0 0	3	ashr SF
AF:= AF+SF:	0 0 .	0 0 0 1	0 0 0 0 0	3	
Result:=					
3lshl AF:	0 0 .	1 0 0 0	0 0 0	0	lshl AF
Test no. 7					
AF:	0 0 .	0 0 0 0	0 0 0 0 0	2	
SF:	0 0 .	1 0 0 0	0 1 1 0 0	0	
2ashr SF:	0 0 .	0 0 1 0	0 0 0 1 1	2	ashr SF
AF:= AF+SF:	0 0 .	0 0 1 0	0 0 0 1 1	2	adder:= b 1010000
Result:=					
2lshl AF:	0 0 .	1 0 0 0	0 1 1	0	lshl AF

	(-1:0)	(1:4)	(33:37)	exp	Test of micro orders
Test no. 8					
AF:	0 0 . 1 0 0 0		1 1 0 0	0	
SF:	0 0 . 0 0 0 0		0 0 0 0	2	SE(0:13):= BUS(0:11)
					con0con0 for SE(12,13)=b10
2ashr AF:	0 0 . 0 0 1 0		0 0 0 1 1	2	ashr AF
AF:= AF+SF:	0 0 . 0 0 1 0		0 0 0 1 1	2	adder:= b 1010000
Result:=					
2lshl AF:	0 0 . 1 0 0		0 1 1	0	lshl AF
Test no. 9					
AF:	0 0 . 0 0 0 0		0 0 0 0 0	2	
SF:	1 1 . 0 0 0 0		0 0 1 0 0	0	
2ashr SF:	1 1 . 1 1 0 0		0 0 0 0 1	2	ashr SF
AF:= AF-SF:	0 0 . 0 0 1 1		1 1 1 1 1	2	adder:= b 0101001
Result:=					
2lshl AF:	0 0 . 1 1 1 1		1 1 1	0	lshl AF
Test no. 10					
AF:	0 0 . 0 0 0 0		0 0 0 0 0	2	
SF:	1 1 . 0 0 0 0		0 1 1 0 0	0	
2ashr SF:	1 1 . 1 1 0 0		0 0 0 1 1	2	ashr SF
AF:= AF-SF:	0 0 . 0 0 1 1		1 1 1 0 1	2	adder:= b 0101001
Result:=					
2lshl AF:	0 0 . 1 1 1 1		1 0 1	0	lshl AF
Test no. 11					
AF:	0 0 . 0 0 0 0		0 0 0 0 0	2	
SF:	1 1 . 0 0 0 0		0 1 0 0 0	0	
2ashr SF:	1 1 . 1 1 0 0		0 0 0 1 0	2	ashr SF
AF:= AF-SF:	0 0 . 0 0 1 1		1 1 1 1 0	2	adder:= b 0101001
Result:=					
2lshl AF:	0 0 . 1 1 1 1		1 1 0	0	lshl AF

	(-1:0)	(1:4)	(33:37)	exp	Test of micro orders
Test no. 12					
AF:	0	0 . 1 0 0 0	0 1 1 0 0	0	
SF:	0	0 . 0 0 0 0	0 0 0 0 0	2	SE(0:13):= BUS(0:11)
					con0con0 for SE(12,13)=b10
2ashr AF:	0	0 . 0 0 1 0	0 0 1 1	2	ashr SF
AF:= AF-SF:	0	0 . 0 0 1 0	0 0 0 1 1	2	adder:= b 10101001
Result:=					
2lshl AF:	0	0 . 1 0 0 0	0 1 1	0	lshl AF
Test no. 13					
AF:	0	0 . 0 0 0 0	0 0 0 0 0	1	AF < > 0 for AF = 2
SF:	0	0 . 0 0 0 0	0 0 1 0 0	0	
1ashr SF:	0	0 . 0 0 0 0	0 0 0 1 0	1	ashr SF
AF:= AF+SF:	0	0 . 0 0 0 0	0 0 0 1 0	1	adder:= b 1010000
Result:=					
35lshl AF:	0	0 . 1 0 0 0	0 0 0	-34	lshl AF
Test no. 14					
AF:	0	0 . 0 0 0 0	0 0 0 0 0	2	AF < > 0 for AF = 1
SF:	0	0 . 0 0 0 0	0 0 1 0 0	0	
2ashr SF:	0	0 . 0 0 0 0	0 0 0 0 1	2	ashr SF
AF:= AF+SF:	0	0 . 0 0 0 0	0 0 0 0 1	2	adder:= b 1010000
Result:=					
36lshl AF:	0	0 . 1 0 0 0	0 0 0	-34	lshl AF
Test no. 15					
AF:	-	- . - - - -	- - - 1 1		this is done by the cf in-
					struction.
AF:	0	0 . 0 0 0 0	0 0 0 0 0	2	AE(0:13):= BUS(0:11)
SF:	0	0 . 0 0 1 0	0 0 0 0 0	2	con0con0 for AE(12,13)=b11
AF:= AF+SF:	0	0 . 0 0 1 0	0 0 0 0 0	2	adder:= b 1010000
Result:=					
2lshl AF:	0	0 . 1 0 0 0	0 0 0	0	lshl AF

Error Reaction:

bits (35,36,37)
test no. <number>
received <48-bit received result>
expected <48-bit expected result>

Complete Test of:

```
AF(-1:37):= if -,MC(10) then SUM(-1:37) else SUM(-1:35)con0con0
                        for MC(10) = 0 and SUM(37); <ARU074,367
                                                    -ARU082,232>
ashr AF for AF(35:37); <ARU074>
lshl AF for AF(35:37); <ARU074>
ashr SF for SF(35:37); <ARU088>
AE:= BUS(0:11)con0con0  for AE(12,13); <ARU074,367-ARU082,236>
SE:= BUS(0:11)con0con0  for SE(12,13); <ARU088-ARU091,432,370>
Adder:= b 1010000       for bits(35:37)
Adder:= b 0101001       for bits(35:37)
AF < > 0  [1]           for AF(36,37); <ARU078,356>
SC > -38 ∧ SC < 38 [1]  for SC = -3,-2,2,3; <MPC008,122-ARU067>
```

Partly Test of:

Round [0]; <MPC125,011-ARU079,213,365>

61. ROUNDING

Rounding of mantissae for floating point numbers involves a set of micro orders and jump conditions. We have therefore for each test number listed the appropriate items under test.

	(-1:0)	(1:4)	(33:37)	exp	Test of micro orders
Test no. 1					
AF:	0	0 . 1 0 0 0	0 0 0 0 0	-1	
SF:	0	0 . 1 0 0 0	0 1 0 0 0	-1	
AF:= AF+SF:	0	1 . 0 0 0 0	0 1 0 0 0	-1	AR(-1)=AR(0) [0];Round [1]
Result:=					
1ashr AF:	0	0 . 1 0 0 0	0 0 1	0	ashr AF for AF(-1) = 0
Test no. 2					
AF:	0	0 . 0 1 0 0	0 0 0 0 0	1	
SF:	0	0 . 0 0 0 0	0 1 1 0 0	0	
1ashr SF:	0	0 . 0 0 0 0	0 0 1 1 0	1	
AF:= AF+SF:	0	0 . 0 1 0 0	0 0 1 1 0	1	Round [0]
Result:=					
1lshl AF:	0	0 . 1 0 0 0	0 1 1	0	
Test no. 3					
AF:	1	1 . 1 0 0 0	0 0 0 0 0	1	
SF:	0	0 . 0 0 0 0	0 1 1 0 0	0	
1ashr SF:	0	0 . 0 0 0 0	0 1 1 0	1	
AF:= AF+SF:	1	1 . 1 0 0 0	0 0 1 1 0	1	Round [0]
Result:=					
1lshl AF:	1	1 . 0 0 0 0	0 1 1	0	

	(-1:0)	(1:4)	(33:37)	exp	Test of micro orders
Test no. 4					
AF:	1 1	0 0 0 0	0 0 0 0 0	0	
SF:	0 0	0 0 0 0	1 0 1 0 0	-2	
2ashr SF:	0 0	0 0 0 0	0 0 1 0 1	0	
AF:= AF+SF:	1 1	0 0 0 0	0 0 1 0 1	0	Round [0]
Result	1 1	0 0 0 0	0 0 1		
Test no. 5					
AF:	0 0	1 0 0 0	0 0 0 0 0	0	
SF:	0 0	0 0 0 0	1 0 1 0 0	-2	
2ashr SF:	0 0	0 0 0 0	0 0 1 0 1	0	
AF:= AF+SF:	0 0	1 0 0 0	0 0 1 0 1	0	Round [0]
Result	0 0	1 0 0 0	0 0 1	0	
Test no. 6					
AF:	1 1	0 0 0 0	0 0 0 0 0	-1	
SF:	1 1	0 0 0 0	0 1 1 0 0	-3	
2ashr SF:	1 1	1 1 0 0	0 0 0 1 1	-1	AR(-1) = AR(0) [0];
AF:= AF+SF:	1 0	1 1 0 0	0 0 0 1 1	-1	Round [0]
Result:=					
1ashr AF:	1 1	0 1 1 0	0 0 0	0	ashr AF for AF(-1) = 1
Test no. 7					
AF:	0 0	1 0 0 0	0 0 1 0 0	-1	
SF:	0 0	1 0 0 0	0 0 0 0 0	-1	
AF:= AF+SF:	0 1	0 0 0 0	0 0 1 0 0	-1	AR(-1) = AR(0) [0] Round [1]
Result:=					
1ashr AF+4:	0 0	1 0 0 0	0 0 1	0	Adder:= b 1111010 (Carry 36)

	(-1:0)	(1:4)	(33:37)	exp	Test of micro orders
Test no. 8					
AF:	1	1 . 0 0 0 0	0 0 0 0 0	-1	
SF:	1	1 . 0 0 0 0	0 0 1 0 0	-1	
AF:= AF+SF:	1	0 . 0 0 0 0	0 0 1 0 0	-1	AR(-1) = AR(0) [0]; Round [1]
Result:=					
1ashr AF+4:	1	1 . 0 0 0 0	0 0 1	0	ashr AF for AF(-1) = 1; Adder:= b 1111010 (Carry 36)
Test no. 9					
AF:	1	1 . 0 1 1 1	1 1 1 0 0	1	
SF:	0	0 . 0 0 0 0	0 1 1 0 0	-1	
2ashr SF:	0	0 . 0 0 0 0	0 0 0 1 1	1	
AF:= AF+SF:	1	1 . 0 1 1 1	1 1 1 1 1	1	Round [1]
AF:= AF+4					
AF(36:37) := 0:	1	1 . 1 0 0 0	0 0 0 0 0	1	Adder:= b 1111010 (Carry 36) AF(36):= 0
Result:=					
1shl AF:	1	1 . 0 0 0 0	0 0 0	0	
Test no. 10					
AF:	0	0 . 1 0 0 0	0 0 0 0 0	0	
SF:	0	0 . 0 0 0 0	0 0 1 0 0	-1	
1ashr SF:	0	0 . 0 0 0 0	0 0 0 1 0	0	
AF:= AF+SF:	0	0 . 1 0 0 0	0 0 0 1 0	0	Round [1]
Result:=					
AF+4:	0	0 . 1 0 0 0	0 0 1	0	Adder:= b 1111010
Test no. 11					
AF:	0	0 . 0 1 0 0	0 0 0 0 0	1	
SF:	0	0 . 0 0 0 0	0 0 1 0 0	-1	
2ashr SF:	0	0 . 0 0 0 0	0 0 0 0 1	1	
AF:= AF+SF:	0	0 . 0 1 0 0	0 0 0 0 1	1	Round [1]
Result:=					
1shl AF+4:	0	0 . 1 0 0 0	0 0 1	0	Adder:= b 1111010

	(-1:0)	(1:4)	(33:37)	exp	Test of micro orders
Test no. 12					
AF:	1 1	1 0 0 0	0 0 0 0 0	1	
SF:	0 0	0 0 0 0	0 0 1 0 0	-1	
2ashr SF:	0 0	0 0 0 0	0 0 0 0 1	1	
AF:= AF+SF:	1 1	1 0 0 0	0 0 0 0 1	1	Round [1]
Result:=					
lshl AF+4:	1 1	0 0 0 0	0 0 1	0	Adder:= b 1111010
Test no. 13					
AF:	0 0	1 1 1 1	1 1 1 0 0	-2	
SF:	1 1	0 0 0 0	0 0 0 0 0	-2	
SUM(-2:37):=					
AF-SF:	001	1 1 1 1	1 1 1 0 0	-2	AR(-1) = AR(0) [0]; Round [1]
SUM:=SUM+4:	010	0 0 0 0	0 0 0 0 0	-2	ashr AF
Result:=					
2ashr SUM:	0 0	1 0 0 0	0 0 0	0	Adder:= b 1111010 (Carry 36)

Error Reaction:

rounding
test no. <number>
received <36-bit mantissa con 12-bit exponent>
expected <36-bit mantissa con 12-bit exponent>

Complete Test of:

AF(-1:37):= if -,MC(10) then SUM(-1:37) else SUM(-1:35)con0con0 for MC(10) = 1; <ARU074,367-ARU082,232>
Round [0] [1]; <MPC125,011-ARU079,213,365>
AR(-1) = AR(0) [0]; <MPC002,116-ARU078,222>

62. SC COUNT UP

The sequential counter, SC, is tested for counting up. Since the exponent of a floating point number is kept in SC after addition, SC is counted up by 1 if the sum of the mantissa leads to mantissa overflow. This test utilizes

```
  0 0 . 1 0 0 - - - 0   exp
  0 0 . 1 0 0 - - - 0   exp
+ 0 1 . 0 0 0 - - - 0   exp
  0 0 . 1 0 0 - - - 0   exp + 1   (SC:= SC + 1)
```

where the testpatterns for SC(12:23) are

Test no. 1	111111	111110
2	111111	111101
3	111111	111011
4	111111	110111
5	111111	101111
6	111111	011111
7	111110	111111
8	111101	111111
9	111011	111111
10	110111	111111
11	101111	111111
12	011111	111111
13	111111	111111
14	000000	000000
15	001001	001000
16	000000	000111
17	000000	111111
18	000111	111111

19	011011	011011
20	000000	010111
21	000010	111111
22	010111	111111
23	000000	000001
24	000000	001111
25	000001	111111
26	001111	111111

Error Reaction:

sc count up
test no. <number>
received <12ext0 con 12-bit received result>
expected <12ext0 con 12-bit expected result>

Complete Test of:

SC:= SC + 1 for SC(12:23); <ARU068,174,089>

63. SC COUNT DOWN

The sequential counter, SC, is tested for counting down. Since the exponent of a floating point number is kept in SC after addition, SC is counted down by 1 if the sum of the mantissa leads to a result which could be normalized by only one left shift. This test utilizes

```
  0 0 . 0 0 1 0 - - - 0    exp
  0 0 . 0 0 1 0 - - - 0    exp
+ 0 0 . 0 1 0 0 - - - 0    exp
  0 0 . 1 0 0 0 - - - 0    exp - 1   (SC:= SC - 1)
```

where the testpatterns for SC(12:23)are

Test no. 1	000000	000001
2	000000	000010
3	000000	000100
4	000000	001000
5	000000	010000
6	000000	100000
7	000001	000000
8	000010	000000
9	000100	000000
10	001000	000000
11	010000	000000
12	100000	000000
13	000000	000000
14	111111	111111
15	110110	110111
16	111111	111000
17	111111	000000
18	111000	000000

19	100100	100100
20	111111	101000
21	111101	000000
22	101000	000000
23	111111	111110
24	111111	110000
25	111110	000000
26	110000	000000

Error Reaction:

sc count down
test no. <number>
received <12ext0 con 12-bit received result>
expected <12ext0 con 12-bit expected result>

Complete Test of:

SC:= SC - 1 for SC(12:23); <ARU068,174,89>

64. SC < > 0, SC > -38 ∧ SC < 38

The two jump conditions, SC < > 0 and SC > -38 ∧ SC < 38, are tested for various testpatterns by means of floating point additions.

Tests nos. 1 to 10:
For these tests the floating point numbers (f1 and f2) are chosen in such a way that the exponent difference, exp1-exp2 >= 38.

```
          f1:= 0.00000 000000 000111 000000  000000 000000 011111 111111
          f2:= 0.00000 000000 000000 000001  000000 000000 exp2
Result:=f1+f2:= 0.00000 000000 000111 000000  000000 000000 011111 111111
```

The testpatterns for exp1-exp2 equal for

Test no. 1	0 000001 000000	(64)
2	0 000010 000000	(128)
3	0 000100 000000	(256)
4	0 001000 000000	(512)
5	0 010000 000000	(1024)
6	0 100000 000000	(2048)
7	0 000000 110000	(48)
8	0 000000 101000	(40)
9	0 000000 100100	(38)
10	0 111111 111111	(4095)

Tests nos. 11 to 20:
For these tests the floating point numbers (f1 and f2) are chosen in such a way that the exponent difference, exp1-exp2 <= -38.

```
          f1:= 0.00000 000000 000000 000001  000000 000000  exp1
          f2:= 0.00000 000000 000111 000000  000000 000000 011111 111111
Result:=f1+f2:= 0.00000 000000 000111 000000  000000 000000 011111 111111
```

The testpatterns for exp1-exp2 equal for

Test no.	11	1 011111 111111	(-2049)
	12	1 101111 111111	(-1025)
	13	1 110111 111111	(-513)
	14	1 111011 111111	(-257)
	15	1 111101 111111	(-129)
	16	1 111110 111111	(-65)
	17	1 111111 001111	(-49)
	18	1 111111 010111	(-41)
	19	1 111111 011001	(-39)
	20	1 111111 011010	(-38)

Tests nos. 21 to 23:

For these tests the floating point numbers (f1 and f2) are chosen in such a way that the exponent difference, 0 < exp1-exp2 < 38.

f1:= 0.00000 000000 000111 000000 000000 000000 000000 001110

f2:= 0.00000 000000 000000 000001 000000 000000 exp2

Result:=f1+f2:= 0.11100 000000 000000 000000 000000 000000 000000 000000

The testpatterns for exp1-exp2 equal for

Test no.	21	0 000000 100100	(36)
	22	0 000000 100011	(35)
	23	0 000000 011000	(24)

Tests nos. 24 to 26:

For these tests the floating point numbers (f1 and f2) are chosen in such a way that the exponent difference, -38 < exp1-exp2 < 0.

f1:= 000000 000000 000001 000000 000000 000000 exp1

f2:= 000000 000000 000111 000000 000000 000000 000000 001110

Result:=f1+f2:=0.11100 000000 000000 000000 000000 000000 000000 000000

The testpatterns for exp1-exp2 equal for

Test no.	24	1 111111 110000	(-16)
	25	1 111111 011100	(-36)
	26	1 111111 011011	(-37)

Error Reaction:

sc < > 0, sc > -38 ∧ sc < 38

test no. <number>

received <36-bit mantissa con 12-bit exponent>

expected <36-bit mantissa con 12-bit exponent>

Complete Test of:

SC < > 0 [1]; <MPC010,124-ARU067>

SC > -38 ∧ SC < 38 [0] [1]; <MPC008,122-ARU067>

65. LOW PRECISION

Floating point arithmetic can be executed in two modes, viz. high- and low precision. In the low precision mode, the mantissa bits 34 and 35 are set equal to the mantissa bit 33. This is tested by two tests where, if high precision were selected, the resultant mantissa would be

Test no. 1 010000 000000 000000 000000 000000 000100
2 010000 000000 000000 000000 111111 111011

Error Reaction:

low precision
test no. <number>
received <36-bit mantissa con 12-bit exponent>
expected <36-bit mantissa con 12-bit exponent>

Complete Test of:

BUS(0:23):= if EX(21) = 0 then AE(0:11)con12ext0
else AE(0:9,9,9)con12ext0 for EX(21) = 1
; <ARU029:ARU030-ARU082,190>

66. CONSTANT -2048

If a floating point operation results in a zero result the exponent is set equal to the least conceivable exponent which is -2048. Hence the test is verified by the instruction

```
fs  w0     0
```

where

```
W0:= 111111 111111  011111 111111
```

Error Reaction:

```
constant -2048
test no. 1
received     <24-bit received result>
expected     000000 000000  10000 000000
```

Complete Test of:

BUS(0:11):= 0, BUS(12:23):= -2048; <ARU014>

67. EX(22:23):= 0

Bits 22 and 23 of the exception register indicate overflow and carry. The two bits are reset when the micro order, EX(22:23):= 0, is activated. EX(21:23) is set to all ones (xl instruction) before the micro order is tested.

Error Reaction:

ex(22:23):= 0
test no. 1
received <21extOconEX(21:23)>
expected 000000 000000 000000 000100

Complete Test of:

EX(22:23):= 0; <ARU102,140>

68. TEST SHIFT IN EX

Overflow in left shifts for arithmetic shift instructions is indicated by a 1 in EX(22). The testpattern

000100 000000 000000 000000

is shifted left 2 times for test no. 1 and 4 times for test no. 2.

Error Reaction:

test shift in ex
test no. <number>
received <21ext0con received EX>
expected <21ext0con expected EX>

Complete Test of:

Test Shift for register EX; <ARU078,365-ARU101,245,434,435,28
-ARU102,440>

69. TEST EXP IN EX

Floating point exponent overflow is indicated in EX(22). Four tests, two of which imply no overflow, are used for verification.
The testpatterns for exp:= SC equal for

Test no.	1	0 011111 111111	no overflow
	2	1 100000 000000	no overflow
	3	0 100000 000000	overflow
	4	1 011111 111111	overflow

Error Reaction:

test exp in ex
test no. <number>
received <21ext0con received EX>
expected <21ext0con expected EX>

Complete Test of:

Test Exp for register EX; <ARU067-ARU101,434,435,28>
SC:= SC-1 for SC(11); <ARU068,174,89>
SC:= SC+1 for SC(11); <ARU068,174,089>

70. TEST INTEGER IN EX

Overflow and Carry are indicated in EX(22:23). The micro order is justified by integer additions with the following integer numbers.

```
Test no. 1      00.111---111
                00.000     0
              + 00.111     1      Overflow = 0     Carry = 0

Test no. 2      11.000---000
                00.000   000
              + 11.000     0      Overflow = 0     Carry = 0

Test no. 3      00.111---111
                00.000   001
              + 01.000   000      Overflow = 1     Carry = 0

Test no. 4      11.000---000
                11.111---111
              + 10.111---111      Overflow = 1     Carry = 1
```

Error Reaction:

```
        test integer in ex
        test no.      <number>
        received      <21ext0 con received EX>
        expected      <21ext0 con expected EX>
```

Complete Test of:

Test Integer for register EX; <ARU098,222,162-ARU101,245,434,438,28
-ARU102,440>

71. CARRY(0)

The jump condition, Carry(0), is tested by means of the aa instructions in two tests having the following testpatterns

Test no. 1

```
  000000 000000 000000 000000  000000 000000 000000 000000
  000000 000000 000000 000000  000000 000000 000000 000000
+ 000000 000000 000000 000000  000000 000000 000000 000000
  Carry(0) = 0
```

Test no. 2

```
  000000 000000 000000 000000  100000 000000 000000 000000
  000000 000000 000000 000000  100000 000000 000000 000000
+ 000000 000000 000000 000001  000000 000000 000000 000000
  Carry(0) = 1
```

Error Reaction:

```
carry(0)
test no.     <number>
received     <48-bit received result>
expected     <48-bit expected result>
```

Complete Test of:

Carry(0) [0] [1]; <MPC002,116-ARU093,284>

72. BR(22), BR(23)

The two above-mentioned jump conditions are tested by the program sequence

```
        al  w0     0
        wm  w1     addr
```

where for

Test no. 1

```
           addr: 111111 111111 111111 111111
             W1: 111111 111111 111111 111111
Result:= WOconW1: 000000 000000 000000 000000  000000 000000 000000 000001
                  BR(22) = 1, BR(23) = 1
```

Test no. 2

```
           addr: 010101 010101 010101 010101
             W1: 000000 000000 000000 000001
Result:= WOconW1: 000000 000000 000000 000000  010101 010101 010101 010101
                  BR(22) = 0, BR(23) = 1
```

Test no. 3

```
           addr: 000000 000000 000000 000001
             W1: 101010 101010 101010 101010
Result:= WOconW1: 111111 111111 111111 111111  101010 101010 101010 101010
                  BR(23) = 0
```

Error Reaction:

```
        br(22), br(23)
        test no.     <number>
        received     <48-bit received result>
        expected     <48-bit expected result>
```

1. Error in test no. 1
   signifies that the jump condition BR(22) is s-a-0.
2. Error in test no. 2
   signifies that the jump condition BR(22) is s-a-1.
3. Error in test no. 3
   signifies that the jump condition BR(23) is s-a-1.

Complete Test of:

BR(22) [0] [1]; <MPC008,122-ARU076,185>
BR(23) [0] ; <MPC008,122-ARU076,185>

73. BE(0)

Bit 0 of register BE is checked as well as the corresponding jump condition BE(0) by the instruction

```
cf  w0    23
```

where for

```
Test no. 1    W3:= 000000 000000  000000 000101
              W0:= 000000 000000  000000 000000
    Result:= W0:= 000000 000000  000000 000101

Test no. 2    W3:= 000000 000000  000000 000101
              W0:= 100000 000000  000000 000000
    Result:= W0:= 000000 000000  000000 000110
```

Error Reaction:

```
be(0)
test no.      <number>
received      <24-bit received result>
expected      <24-bit expected result>
```

Tests 1 and 2 check BE(0) for the logical values 0 and 1, respectively.

Complete Test of:

```
BE(0):= BUS(0); <ARU077,155-ARU083,169,376,27>
BE(0)  [0] [1]; <MPC011,125-ARU077,155>
```

74. INTEGER DIVISION

The two micro orders Test WD Sign and Divide Integer plus the jump condition BR(1) = BR(2) (signifies division overflow) are necessary prerequisities for the success of an integer division. The test works by executing the instruction

```
        wd  w1      addr
```

where for

Test no. 1

```
        WOconW1: 011111 111111 111111 111111  111111 111111 111111 111111
           addr:                              000000 000000 000000 000000
Result:= WOconW1: 011111 111111 111111 111111  111111 111111 111111 111111
```

The dividend remains in WOconW1, since the division leads to overflow.

Test no. 2

```
        WOconW1: 011111 111111 111111 111111  111111 111111 111111 111111
           addr:                              111111 111111 111111 111111
Result:= WOconW1: 011111 111111 111111 111111  111111 111111 111111 111111
```

The dividend remains in WOconW1, since the division leads to overflow.

Test no. 3

```
        WOconW1: 000000 000000 000000 111111  101010 101010 101010 101010
           addr:                              000000 000001 000000 000000
Result:= WOconW1: 000000 000000 101010 101010  000000 111111 101010 101010
```

Test no. 4

```
        WOconW1: 111111 111111 111111 000000  010101 010101 010101 010110
           addr:                              111111 111111 000000 000000
Result:= WOconW1: 111111 111111 010101 010110  000000 111111 101010 101010
```

Error Reaction:

integer division

test no. <number>

received <48-bit received result>

expected <48-bit expected result>

Only the two first quotient bits are of interest in tests 1 and 2 since these bits, when they are alike, signify an overflow condition.

Complete Test of:

Test WD Sign; <ARU099:ARU100>

Divide Integer; <ARU076,376-ARU080:ARU081-ARU083-ARU098>

BR(1) = BR(2) [0] [1]; <MPC003,117-ARU079,222>

75. CONSTANT 1

A constant 1 is sometimes used in integer division for correcting the least significant bit of the quotient. This is tested by executing the instruction

```
        wd  w1     addr
```

where

```
        WOconW1: 000000 000000 000000 000000  000000 000000 000000 100111
           addr:                              000000 000000 000000 100000
Result:= WOconW1: 000000 000000 000000 000111  000000 000000 000000 000001
```

Error Reaction:

```
        constant 1
        test no. 1
        received     <48-bit received result>
        expected     000000 000000 000000 000111  000000 000000 000000 000001
```

1. W1 = 0 The micro order BUS = 1 has not been activated.
2. W1 < > 0, 1 The micro order does not transmit a 1 but a number which equals W1.

   WO is not influenced by the micro order.

Complete Test of:

```
        BUS(0:23):= 1; <ARU031,157>
```

76. AR < > 0

The jump condition has already been tested for AR having a value different from zero and therefore this test investigates the zero condition. This is done by executing the instruction

```
        wd  w1      addr
```

where

```
        WOconW1: 111111 111111 111111 111111  111111 111111 111110 100000
           addr:                              000000 000000 000000 100000
Result:= WOconW1: 000000 000000 000000 000000  111111 111111 111111 111101
```

Error Reaction:

```
        ar < > 0
        test no. 1
        received     <48-bit received result>
        expected     000000 000000 000000 000000 111111 111111 111111 111101
```

If the jump cannot attain the zero value, then the received result equals

```
        111111 111111 111111 100000  111111 111111 111111 111110
```

Complete Test of:

AR < > 0 [0]; <MPC008,122-ARU078>

77. FLOATING MULTIPLICATION

A number of micro orders and jump conditions are used only in floating point multiplication, for which reason it is not possible to test them separately. The tests work by executing the instruction

fm w1 addr

where

WOconW1: depends on test number

addr: 010000 000000 000000 000000 000000 000000 000000 100101

The testpatterns for WOconW1 equal for

Test no.	1	000000	000000	000000	000000	000000	000000	000000	000000
	2	000000	000000	000000	000000	000000	000001	000000	000000
	3	000000	000000	000000	000000	000000	000010	000000	000000
	4	000000	000000	000000	000000	000000	000100	000000	000000
	5	000000	000000	000000	000000	000000	001000	000000	000000
	6	000000	000000	000000	000000	000000	010000	000000	000000
	7	000000	000000	000000	000000	000000	100000	000000	000000
	8	000000	000000	000000	000000	000001	000000	000000	000000
	9	000000	000000	000000	000000	000010	000000	000000	000000
	10	000000	000000	000000	000000	000100	000000	000000	000000
	11	000000	000000	000000	000000	001000	000000	000000	000000
	12	000000	000000	000000	000000	010000	000000	000000	000000
	13	000000	000000	000000	000000	100000	000000	000000	000000
	14	000000	000000	000000	000001	000000	000000	000000	000000

Error Reaction:

floating multiplication
test no. <number>
received <48-bit received result>
expected <48-bit expected result>

Complete Test of:

BUS(0:23):= 35; <ARU031,88,157>

BE(0:11):= BUS(0:11); <ARU077>

lshr BF; <ARU075:ARU077-ARU081,311,190,236-ARU083,169,212>

BE(10) [0] [1]; <MPC005,119-ARU077>

BE(11) [0] [1]; <MPC005,119-ARU077>

78. FLOATING DIVISION

A number of micro orders and jump conditions are used in floating point division for which reason it is not possible to test them separately. The test works by executing the instruction

```
fd  w1     addr
```

The testpatterns for addr equal

Tests nos. 1 to 13

	010000 000000 000000 000000 000000 000000 000000 000001	(1)
Test no. 14	100000 000000 000000 000000 000000 000000 000000 000000	(-1)

The testpatterns for W0conW1 equal

Test no. 1	010000 000000 000000 000001	000000 000000 000000 000000
2	010000 000000 000000 000000	100000 000000 000000 000000
3	010000 000000 000000 000000	010000 000000 000000 000000
4	010000 000000 000000 000000	001000 000000 000000 000000
5	010000 000000 000000 000000	000100 000000 000000 000000
6	010000 000000 000000 000000	000010 000000 000000 000000
7	010000 000000 000000 000000	000001 000000 000000 000000
8	010000 000000 000000 000000	000000 100000 000000 000000
9	010000 000000 000000 000000	000000 010000 000000 000000
10	010000 000000 000000 000000	000000 001000 000000 000000
11	010000 000000 000000 000000	000000 000100 000000 000000
12	010000 000000 000000 000000	000000 000010 000000 000000
13	010000 000000 000000 000000	000000 000001 000000 000000
14	010101 010101 010101 010101	010101 010101 000000 000000

Error Reaction:

```
floating division
test no.      <number>
received      <48-bit received result>
expected      <48-bit expected result>
```

Tests 1 to 13 check the data transfer BUS(0:11):= BE(0:11) for BE(0:11) = W1(0:11).

Complete Test of:

Test FD Sign; <ARU099:ARU100>

Divide Floating; <ARU076-ARU080:ARU081-ARU083-ARU098>

BUS(0:11):= BE; <ARU002:ARU013-ARU083,212,27>

FDsub [0] [1]; <MPC006,120-ARU099,158>

79. CONSTANT 12

The micro order, BUS:= 12, is used by the interruption service to fetch the Service Address which is kept in location 12. The interruption service is invoked by an illegal instruction.

Error Reaction:

sequence error constant 12

If the micro order generates another value than 12 on the buslines, then the test program shall not return to the expected point in the program, because an erroneous service address is fetched. This implies that the test is not terminated properly and this is normally disclosed by the test sequence superviser. If, however, the fetched service address implies that an interrupt is generated anew, a closed program loop may be expected in which case test-program messages fail to appear on the typewriter.

Complete Test of:

BUS(0:23):= 12; <ARU031,88,157>

80. ADDRESS OVERFLOW

The store controller, STC, invokes an interrupt if the specified storage address exceeds the storage capacity. This is tested by

```
rl  w0     addr
```

where

```
addr = 111111 111111  111111 111111
```

Prior to the execution of this instruction,

```
W0 = 111110 001111  011111 111000
```

Error Reaction:

```
address overflow
test no. 1
received     <24-bit received W0>
expected     111110 001111  011111 111000
```

Complete Test of:

```
Read Data  for  SB = -1   (Fixed Address = 1);
<MPC002:MPC011-MPC027,34-STC002-STC003,445,447-STC006,447-ARU049>
```

81. PROTECTION

The effect of the protection system is tested for the following conditions:

Test no. 1 Execute a privileged instruction in task mode. This should invoke the interruption system.

Test no. 2 Execute an instruction in task mode followed by a protected instruction. This should invoke the interruption system.

Test no. 3 Store a word in a non-protected location by means of an instruction in task mode. This is a valid instruction.

Test no. 4 Store a word in a protected location by means of an instruction in task mode. This should invoke the interruption system.

Error Reaction:

protection
test no. <number>
received <24-bit received result>
expected <24-bit expected result>

The result for tests 1 and 2 is the return address from the interruption routine. For tests 3 and 4, the results are the contents of the location under test.

Complete Test of:

Read Instruction for Test no. 1; <MPC027-STC006,444>
Read Split for illegal storing; <STC007,444>
MMode:= PROTECT for PROTECT = 0; <ARU105,421>
MMode [0]; <MPC009,123-ARU105,421>
MMode ∨ -,PROTECT [0] [1]; <MPC010,124-ARU105,245>

82. REGISTER IM

Register IM is checked for loading and storing by means of the following testpatterns:

Test no.	1	100000	000000	000000 000000
	2	010000	000000	000000 000000
	3	001000	000000	000000 000000
	4	000100	000000	000000 000000
	5	000010	000000	000000 000000
	6	000001	000000	000000 000000
	7	000000	100000	000000 000000
	8	000000	010000	000000 000000
	9	000000	001000	000000 000000
	10	000000	000100	000000 000000
	11	000000	000010	000000 000000
	12	000000	000001	000000 000000
	13	000000	000000	100000 000000
	14	000000	000000	010000 000000
	15	000000	000000	001000 000000
	16	000000	000000	000100 000000
	17	000000	000000	000010 000000
	18	000000	000000	000001 000000
	19	000000	000000	000000 100000
	20	000000	000000	000000 010000
	21	000000	000000	000000 001000
	22	000000	000000	000000 000100
	23	000000	000000	000000 000010
	24	000000	000000	000000 000001

Error Reaction:

register im

test no. <number>

received <24-bit received result>

expected <24-bit expected result>

An error is due to an incorrect transmission to register IM or vice versa. Interrupts may occur if the interruption system cannot be disabled.

Complete Test of:

IM(1:23):= SB(1:23), IM(0) is permanently equal to 1;
<ITR001:ITR006-ITR010,412,345-LCI001:LCI002-ARU028,25>

BUS(0:23):= IM; <ARU026:ARU027-ARU028,218,25>

83. INTERRUPT ENABLE

The micro order, ITRenable:= FR(5) is verified for its ability to suppress interrupts and allow interrupts to break the normal program sequence.
The micro order is tested by means of jd and jl instructions.

Error Reaction:

```
interrupt enable
test no.     <number>
received     000000 000000  000000 000000
expected     000000 000000  000000 000000
```

Error in test no. 1 signifies an error for ITRenable:= 0.
Error in test no. 2 signifies an error for ITRenable:= 1 or that the micro order Test Integer does not produce an interrupt.
Confer also interrupt request.
The received and expected 24 bits have no signification.

Complete Test of:

ITRenable:= FR(5); <ARU102,189,245,28>
Test Integer; comment IR(1):= 1; <ARU101,435-ITR001,400>

84. REGISTER IR

The purpose of this test is to check the interrupt register for

1. activation of individual interrupt bits; IR(n):= 1
2. storing of IR register; BUS:= IR
3. clearing of individual interrupt bits; IR(n):= 0

Activation of external interrupts (IR(3:23)) are simulated by connecting the I/O BUS plug to the interrupt plug 1021. IR(0) is not included in this test. In details, the program works as described below for each test number, and the testpatterns are for

Test no. 1	010000	000000	000000	000000
2	001000	000000	000000	000000
3	000100	000000	000000	000000
4	000010	000000	000000	000000
5	000001	000000	000000	000000
6	000000	100000	000000	000000
7	000000	010000	000000	000000
8	000000	001000	000000	000000
9	000000	000100	000000	000000
10	000000	000010	000000	000000
11	000000	000001	000000	000000
12	000000	000000	100000	000000
13	000000	000000	010000	000000
14	000000	000000	001000	000000
15	000000	000000	000100	000000
16	000000	000000	000010	000000
17	000000	000000	000001	000000

```
        18      000000 000000  000000 100000
        19      000000 000000  000000 010000
        20      000000 000000  000000 001000
        21      000000 000000  000000 000100
        22      000000 000000  000000 000010
        23      000000 000000  000000 000001
```

Program description for test:

```
for n = 1 step 1 until 23 do
begin
  IR:= 0; IR(n):= 1; comment set interrupt bit n;
  WO:= IR;
  if WO(n) = 0 then begin error message; specification:= set end;
  IR:= 0; IR(n):= 1;
  IR(n):= 0; comment clear interrupt bit n;
  WO:= IR;
  if WO < > 0 then begin error message; specification:= clear
end;
```

Error Reaction:

```
        register ir
        test no.      <number>
        received      <24-bit received result>
        expected      <24-bit expected result>
        <specification>
where   <specification>::= set | clear
```

Complete Test of:

```
        IR:= IR ∧ -,SB; <ITR001:ITR006-ITR010,412,345,211>
        BUS(0:23):= IR; <ARU026:ARU027-ARU028,218,25>
        Test Shift; comment IR(1):= 1; <ARU101,435>
        Test Exp; comment IR(2):= 1; <ARU101,435-ITR001,401>
```

85. INTERRUPT REQUEST

Interrupt request is a decoding network whose value depends on IR and IM. The equation is

INTERRUPT REQUEST:= ors(IR ∧ IM).

If ITRenable = 1, then the running program is interrupted if and only if INTERRUPT REQUEST = 1. This assertion shall be proved by the following test-patterns:

Test no. 1	IR:= 0				; IM:= -1
2	IR:=	011111	111111	111111	111111	; IM:= 0
3	IR:= IM:=	010000	000000	000000	000000	
4	IR:= IM:=	001000	000000	000000	000000	
5	IR:= IM:=	000100	000000	000000	000000	
6	IR:= IM:=	000010	000000	000000	000000	
7	IR:= IM:=	000001	000000	000000	000000	
8	IR:= IM:=	000000	100000	000000	000000	
9	IR:= IM:=	000000	010000	000000	000000	
10	IR:= IM:=	000000	001000	000000	000000	
11	IR:= IM:=	000000	000100	000000	000000	
12	IR:= IM:=	000000	000010	000000	000000	
13	IR:= IM:=	000000	000001	000000	000000	
14	IR:= IM:=	000000	000000	100000	000000	
15	IR:= IM:=	000000	000000	010000	000000	
16	IR:= IM:=	000000	000000	001000	000000	
17	IR:= IM:=	000000	000000	000100	000000	
18	IR:= IM:=	000000	000000	000010	000000	
19	IR:= IM:=	000000	000000	000001	000000	
20	IR:= IM:=	000000	000000	000000	100000	
21	IR:= IM:=	000000	000000	000000	010000	
22	IR:= IM:=	000000	000000	000000	001000	
23	IR:= IM:=	000000	000000	000000	000100	
24	IR:= IM:=	000000	000000	000000	000010	
25	IR:= IM:=	000000	000000	000000	000001	

Error Reaction:

interrupt request
test no. <number>
received 000000 000000 000000 000000
expected 000000 000000 000000 000000

Interrupt Request should be 0 for tests 1 and 2, otherwise 1. The received and expected 24 bits have no signification.

Complete Test of:

INTERRUPT REQUEST ∧ ITRenable; <ITR007,211,345,399>

86. INTERRUPT NUMBER

The decoding network that selects the interrupt with the highest priority and also generates the interrupt number to be stored in location 8 is tested.
Testpattern for IR(0:23) ∧ IM(0:23):

		Expected interrupt address
Test no. 1	100000 000000 000000 100000	0
2	110000 000100 000000 000000	0
3	011000 000000 001000 000000	2
4	001001 000000 000000 000000	4
5	000100 000000 000000 000001	6
6	000011 000000 010000 000000	8
7	000001 100000 000000 100000	10
8	000000 100100 000000 000000	12
9	000000 010000 000000 000000	14
10	000000 001100 000001 000000	16
11	000000 000110 000000 000010	18
12	000000 000010 010000 000000	20
13	000000 000001 000000 000000	22
14	000000 000000 110000 000100	24
15	000000 000000 011000 000000	26
16	000000 000000 001001 000000	28
17	000000 000000 000100 000000	30
18	000000 000000 000011 000000	32
19	000000 000000 000001 100000	34
20	000000 000000 000000 100100	36
21	000000 000000 000000 010000	38
22	000000 000000 000000 001100	40
23	000000 000000 000000 000110	42
24	000000 000000 000000 000010	44
25	000000 000000 000000 000001	46
26	000010 000000 000000 000010	8
27	000000 001000 000000 000000	16
28	000000 000000 100000 000000	24
29	000000 000000 000010 000000	32
30	000000 000000 000000 001000	40

Error Reaction:

interrupt number

test no. <number>

received <24-bit interrupt number>

expected <24-bit interrupt number>

If only tests 1 and 2 fail it is most likely that the micro order IR(0):= 1 does not set interrupt bit(0) to 1. Other errors originate from the decoding network itself. Confer this test with the test CLEAR ANSWERED INTERRUPT.

Complete Test of:

IR(0):= 1; <ITR001>

BUS(0:23):= 18ext0 con ITRnumber(18:22) con0;

<ITR007:ITR009-ARU028,157,27-ARU030:AR0031>

87. CLEAR ANSWERED INTERRUPT

Whenever an interrupt is answered, i.e. when the running program is interrupted, the corresponding interrupt bit is turned off. The goal of this test is to verify that the correct bit and only this bit is cleared. Register IR is for test 1 equal to all ones and for tests 2 to 24

IR:= 011111 111111 111111 111111.

The mask register IM equals for

Test no.	1	100000	000000	000000 000000
	2	010000	000000	000000 000000
	3	001000	000000	000000 000000
	4	000100	000000	000000 000000
	5	000010	000000	000000 000000
	6	000001	000000	000000 000000
	7	000000	100000	000000 000000
	8	000000	010000	000000 000000
	9	000000	001000	000000 000000
	10	000000	000100	000000 000000
	11	000000	000010	000000 000000
	12	000000	000001	000000 000000
	13	000000	000000	100000 000000
	14	000000	000000	010000 000000
	15	000000	000000	001000 000000
	16	000000	000000	000100 000000
	17	000000	000000	000010 000000
	18	000000	000000	000001 000000
	19	000000	000000	000000 100000
	20	000000	000000	000000 010000
	21	000000	000000	000000 001000
	22	000000	000000	000000 000100
	23	000000	000000	000000 000010
	24	000000	000000	000000 000001

Error Reaction:

clear answered interrupt

test no. <number>

received <24-bit interrupt register>

expected <24-bit interrupt register>

Complete Test of:

IR(ITR number):= 0; <ITR001:ITR006-ITR008,397,339-ITR010,412>

RCSL: 31-A29
Author: H. Kold Mikkelsen
Edited: July 1971

RC 4000

TEST OF PERIPHERAL DEVICES

OPERATOR's MANUAL

KEY: RC 4000, Test of Peripheral Devices, Operator's Manual

A/S REGNECENTRALEN
Falkoneralle 1
DK 2000 Copenhagen F

CONTENTS:

Preface:

This manual is an substitution of RCSL: 51-VB473 by Jørgen Lindballe.
All test programs designed for the previous loader will work in loader 2.

LOADER, PROCEDURES AND INTERRUPTION

The loader 2 (which reads and stores procedures and testprograms) and 9 procedures (used by the programs for output on and input from the operator's typewriter, for output on a specified output device, for reservation of buffer area and for administration of the test) exist in the binary version on one paper tape.

This paper tape is read from the RC 2000 Paper Tape Reader (dev. No. 0) when the operator activates the RESET and then the AUTOLOAD push-button.

The loader first writes:

channel no. of operator key =
dev. no. of operator's typewriter =

and the operator types the interrupt channel No. of the operator key and the device No. of the typewriter he wants to use. (These numbers may be altered later by activating the RESET and then the START push-button after which the loader writes the above-mentioned questions again).

The loader now writes:

loader 2
input from dev.no.:

and then the operator writes the number of the device, from which the binary testprograms shall be read, followed by <NL>. Consequently he must write 0, if input is wanted from the paper tape reader. If input is wanted from a magnetic tape station (dev. No. > 0), the loader asks:

file no.

and the operator writes the number of the file in which the testprograms are placed. These filenumbers appear from this table:

RC NO.	KIND OF PERIPHERAL DEVICE	FILE NO.
2000	Paper Tape Reader	
150	Paper Tape Punch	
315	Typewriter	
610	Lineprinter, Data Products	
333	Lineprinter, Anelex	
4191	Incremental Plotter	
707	Magnetic Tape Station, 7 tracks	
709	Magnetic Tape Station, 9 tracks	
4415	Drum	
4314	Disc	
	Interval Timer	
	Teletypewriter	
	Display	

When all the testprograms (not more than 15) for the kind of device are stored, the loader writes:

<name of the kind of peripheral device>
device no. =

After this the operator types the device number (or the device numbers, as it is possible to write up to 3 device numbers separated by <comma>; e.g. teletypewriter). When the program hereafter writes:

channel no. =

the operator types the interrupt channel number of the device (or the channel numbers, as it is possible to write up to 3 channel numbers separated by <comma>; e.g. teletypewriter).
Now the program asks for an output device:

output dev. no. =

and the operator types an integer >0.
Hint: programs which uses the output dev. may work as move programs by specifying the cpu-timer (dev. 3) as output device.

Next the program writes:

testprogram:

and the operator may type a or b or c or ... and in this way select the first, the second, the third, ... testprogram. If he types <NL> the following directory of the stored testprograms is written:

a <description of 1st testprogram>
b <description of 2nd testprogram>
c <description of 3rd testprogram>
.
.
.
<description of the last testprogram>
testprogram:

Having selected the testprogram the operator to the question:

number of runs =

must write the number of times the program is wanted to be executed. This number must be chosen so that

$$1 \leq \text{number of runs} \leq 8\ 388\ 607$$

Before the execution of some runs it writes:

run no. <run no.>

namely before the execution of

1st, 2nd, ..., 9th run, if 1 ≤ No. of runs ≤ 9
1st, 11th, 21st, ..., 91st run, if 10 ≤ No. of runs ≤ 99
1st, 101st, 201st, ..., 901st run, if 100 ≤ No. of runs ≤ 999
etc.

Having executed the specified number of runs, the program writes:

test end

and now it is possible to select a new testprogram.

Using the loader some erroneous situations may occur:

If 'end of tape' appears in the paper tape reader or if 'tape mark' appears on magnetic tape when the loader reads testprograms before the first testprogram has been stored, it writes:

mount paper tape

end of file

respectively. If the operator types <NL> after the first message, the loader continues to read.

The error messages:

parity error in <program description>

and (when loading from the paper tape reader):

checksum error in <program description>

means parity error and checksum error in the program being loaded.

If bit 0, 2, 3, 4, 5 or 6 is set in the statusword when the programs are loaded from tape, the loader writes:

status = <bit 0-9>

and the programs must be loaded again.

It is always possible to break the execution of a testprogram by activating the operator key. After this the interrupt sequence writes:

select

Then the operator may type t, o, d, l or c after which the typewriter continues to write the below-mentioned underlined texts:

testprogram:

The operator may select a new testprogram for the same device number.

output dev. no. =

A new output dev. number may be selected.

device no. =

The operator may select a new device number (and after this as usual a new interrupt channel number) for the same kind of peripheral device.

loader 2

Now a new set of testprograms may be loaded, and in this way a new kind of peripheral device may be selected.

core store contents

The contents of some part of the core store may be written on the output device. When the program writes:

first word addr. =

last word addr. =

the operator specifies the part of the core store; it must be mentioned that for the testprograms which use input-output buffer, the addresses of the first and the last bufferword are written on the typewriter immediately before the execution of the first run.

When the program writes:

mode =

the operator types t, d, b or i after which the typewriter continues to write the below-mentioned underlined texts:

text

The bitpatterns are written as a text.

decimal

The bitpatterns are interpreted as integers (negative, zero or positive) and written in decimal.
The program waits for input from the console used for the print layout:

<answer> ::= <integer> ! <empty>

where 0 <= <integer> <= 24 and
<empty> ::= <integer> = 24

For <integer> = 0 the program converts the integer to 24.
If <integer> <0 or <integer>>24 the program asks for a new input.
Now the binary word is divided into a number of blocks from the left side, containing the specified number of bits and is printed as separate decimal numbers. If the division 24/<integer> not comes right, the rest word is printed as a binary number, e.g.

decimal 10

will cause the leftmost 20 bits to be separated into blocks of 10 bits and the rest of 4 bits is printed as a binary number.

binary

The bitpatterns are interpreted as positive integers and they are written in binary.
The program waits for input from the console:

<answer> ::= <integer> ! <empty>

where

0<= <integer> <= 24 and
<empty> ::= <integer> = 24
For <integer> = 0 the program converts it to 24.

The binary word is now printed in blocks of <integer> separated by a <space>. The dividing is made from the left.

instruction

The bitpatterns are written as machine instructions including the mnemonic functioncodes.
In case of hardware or software error interrupt No. 0 may occur. This involves an error message:

interrupt no. 0

from the interrupt-sequence. In this case not only the testprograms but even the loader should be stored again.

By activating the RESET and then the START push-button, a jump to the loader is executed, and a new operator-key and -typewriter may be selected.

On the next page is shown some messages to and from the operator during a test.

The testprograms for high-speed devices are so designed that they propose the start address of the input-output buffer by writing:

fbw = <address of first free word>

and waits for input. If the operator types <NL>, he accepts the start address; if he types a slash, the programs ask:

fbw =

and he must input another start address. This address must be within a free part of the core store, i.e. 1) an address lower than the loader 2 start address (but not less than 24) or 2) an address higher than the test-program's top address. If this condition is not fulfiled the program asks for a new input. Contition 1) is only significant when using the relocatable loader with start address greater than 0. After input from the console the address of the last buffer word is calculated and written.
If the last buffer word (lbw) is calculated to be without the free part of core store some error messages will occur.

Messages to and from the operator:

```
loader  2
input from dev. no.: 0
rc teletypewriter
device no.  = 16,17,18
channel no. = 22,23
output dev. no. = 5
testprogram:

a   1.2 read (echo)
b   1.3 write key - board
c   1.4 timer
d   2.1 sequence

testprogram: b
number of runs = 1

run no.       1
terminal disconnected


select testprogram: c
number of runs = 10
time,expected:3000-6000 msec
time,measured:
run no.       1


select loader  2
input from dev. no.: 0
rc 315 typewriter
device no.  = 2
channel no. = 7
output dev. no. = 2
testprogram: d
number of runs = 1

sequence =
abcdefghijklmn

run no.       1
abcdefghijklmn
test end

testprogram:

select device no.  = 10
channel no. = 8
output dev. no. = 5
testprogram: b
number of runs = 2

run no.       1
run no.       2
test end

testprogram:
```

THE RELOCATABLE LOADER 2

The relocatable loader consists of the above-mentioned loader and procedures, however so designed that the relocatable loader and the testprograms may be stored everywhere within the available core store if the operator before activating the AUTOLAD push-button puts the start address into w3. This start address must be chosen that

$$0 \leq w3 \leq \text{length of core store} - (\text{length of relocatable loader} + \text{length of testprograms})$$

All lengths are measured in No. of bytes. The length of the relocatable loader is 2980 bytes.

When w3 = 0, the loader and the testprograms are stored as usually (see chapter 1.1 page 5).

RCSL: 31-D14
Author: J. Lindballe,
H. Kold Mikkelsen
Edited: July 1971

TEST OF PERIPHERAL DEVICES

MAIN CHARACTERISTICS

KEY: RC 4000, Hardware Testprogram, Reference Manual

ABSTRACT: This paper describes the features of the loader 2. Further more some of the standard facilities of different testprograms are described.

A/S REGNECENTRALEN
Falkoneralle 1
DK 2000 Copenhagen F

CONTENTS:

Chapter 1: MAIN CHARACTERISTICS

PREFACE:

This paper is an extension of 51-VB431 by Jørgen Lindballe.
All testprograms designed for the previous loader will equally well run in loader 2.

1.1. INTRODUCTION

For each of the belowmentioned RC 4000 peripheral devices are made a number of test programs. The purpose of some of these is to check the peripheral device in question (checking programs), while the purpose of others is to help the operator to localize errors if these occur (motion programs):

Kind of Peripheral Devices	Number of Programs
Paper Tape Reader	2
Paper Tape Punch	4
Typewriter	4
Lineprinter, Data Product	3
Lineprinter, Anelex	2
Plotter	2
Magnetic Tape Station, 7 tracks	4
Magnetic Tape Station, 9 tracks	5
Drum	4
Disc	5
Interval Timer	1
Teletypewriter	4
Display	2

Besides the test programs are programmed partly a set of standard procedures used of the programs mainly for output on and input from the operator's typewriter and partly a loader program, the aim of which is to place the test programs of a given kind of peripheral devices and the mentioned procedures in the core store, because the test programs are used independent of the RC 4000 monitor and operative system (because it is a demand to these programs that they are possible to test any kind of peripheral devices within a core store of minimum size: 4096 words).

The loader 2, the test programs as well as the procedures, are written in SLANG.

A test by means of these programs demands perfectly operating:

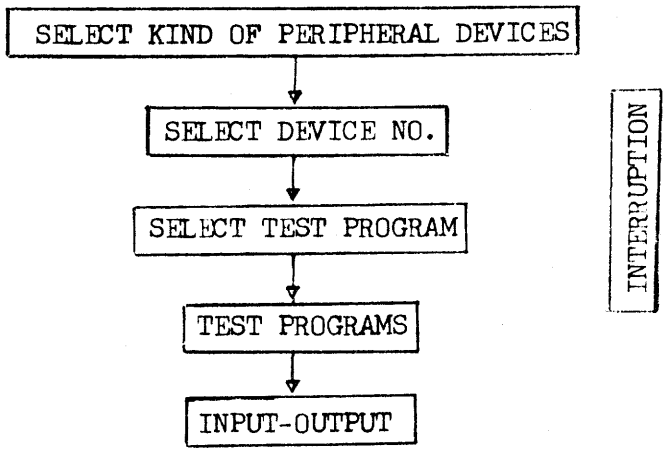
1. a central unit including
2. a core store not less than 4096 words.
3. a paper tape reader (device No. 0).
4. a typewriter (device No. 2).

Furthermore it should be desirable that

5. an operatorkey (interrupt channel No. 3), and
6. a magnetic tape station (7 or 9 tracks)

are available.

The abovementioned programs are linked together in accordance with this hierarchy:



so that, by the aid of the loader, all the test programs of a given kind of peripheral devices (say magnetic tape station or typewriter) can be placed in the core store, and after this the operator may select the device number and then the test program he wants to be executed. It is always possible to break the execution of a test program by pushing the operatorkey; after this he may select for the same device number a new test program, or for the same kind of peripheral devices he may select another device number, or he may load the test programs of a new kind of peripheral devices. (Finally, after such an operator termination, it is possible to have the contents of a part of the core store typed out).

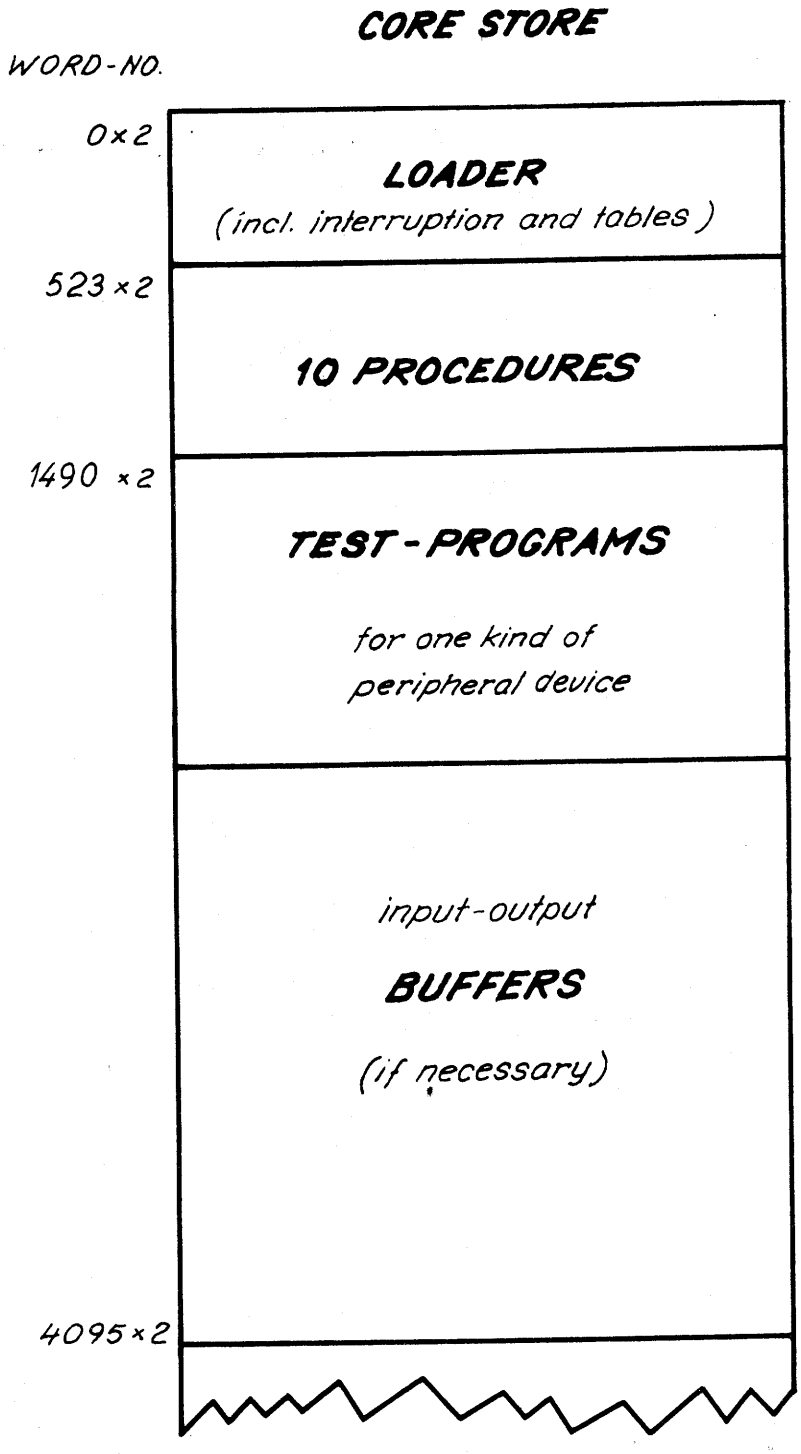
The loader 2 and the procedures are found in a binary version punched on a paper tape which may be read by the paper tape reader after activating the autoload pushbutton.

The test programs are found in a binary version both punched on paper tape (so that all the test programs for a given kind of peripheral devices are collected on one tape) and written on magnetic tape (so that all the test programs for a given kind of peripheral devices are collected in one file, and in such a way that each program forms a block).

Loader, procedures, and test programs are loaded as shown on the next page. It is noticed that only the first 4096 words are necessary for a test.

Before the detailed description of this complex of programs, it must be mentioned that everywhere in this report, the output on the typewriter from the programs is underlined so that it is not mistaken for the operator input.

Finally it is a rule that the operator, when he has carried out what a program has asked him to do, he must type in the character <NL>.



CORE STORE LAYOUT

1.2. THE LOADER 2.

When the binary tape, containing the loader and the procedures, has been placed in the paper tape reader (device No. 0), and the operator has activated the RESET and then the AUTOLOAD pushbutton, the loader is read into the core store by means of autoload word instructions in a 'bootstrap' as shown on the next page.

Next the loader, which occupies 523 words, is executed in a way explained on the following pages.

First the loader is initialized whereby words Nos. 16, 18, 20, and 22 are filled with the start addresses of 4 tables, which in this way are made available for all the programs which later on are placed in the core store. These tables are gradually filled with the following data:

TABLE 1 (startaddress in word 16):

word 1: The address of the first free word. This address is placed by the loader when the last test program has been loaded. It is used by the 7th procedure when a buffer area is reserved.

word 2: 2×the number of test programs (incl. the name (chapter 4)). This number is placed by the loader when the last test program has been loaded. It is used when the 9th procedure delivers the directory (a description of the test programs stored (chapter 1.3)).

word 3: Last word address of the core store + 2. This address is stored by the loader when initialized, and it is used by the 7th procedure.

word 4: When a test program, which tests the interrupt signal from the peripheral device, is initialized, it stores in this word the start address of its own interrupt sequence. Then, in case of interrupt signal from the peripheral device, a return jump from the loaders interrupt sequence to this start address is performed.

word 5: Device No. of operator's typewriter < 6. This device No. is stored by the loader when initialized, and it is used by the 1st and the 4th procedure.

word 6: Output dev. No. < 6 and is stored by the loader when initialized, and is used by the 1st and the 4th procedure.

The Loader Bootstrap

The Paper Tape

```
k

 0  aw                    2
 2  aw                    4
 4  jl                    0
 6  aw                 a401   (=x+0)
 8  aw          x1        4
10  aw                 a402   (=x+2)
12  al      w1  x1        2
14  aw                 a403   (=x+4)
16  jl.                   -4
18  jl      w1         a401   (=x+0)
20                        0

 y a300:


x+0  a401:                 0
x+2  a402:                 0
x+4  a403:                 0
x+6                        0
x+8                        0
x+10                       0
x+12 jl.     w1         a300.  (=y-x-12)
```

The Core Store

```
a

 0  aw                    2
 2  aw                    4   x+0 etc.
 4  jl                    0
 6
 8                        0


y-12  a300:                  ; initialize


x-6                       0
x-4                       0
x-2                       0
x+0   aw          x1      4  jl. w1 y-x-12
x+2   al    w1    x1      2
x+4   jl.                -4
```

TABLE 2 (startaddress in word 18):

word 1-10: Contain the addresses of the entry points of the 10 procedures. These addresses are stored by the loader successively when the procedures are stored. They are used by all of the programs.

TABLE 3 (startaddress in word 20):

word 1-16: Contain the addresses of the entry points of the name (chapter 1.4) and the test programs. These addresses are stored by the loader successively when the programs are stored. In connection with the administration of a test they are used by the 9th procedure. (chapter 1.3).

TABLE 4 (startaddress in word 22):

word 1,3,5: Device number(s) < 6
word 2,4,6: Interrupt channel number(s) ≠(-1)

When a test program is initialized, it fetches from here the device number(s), and if it tests the interrupt signal then the interrupt channel number(s) too.

When the loader has been initialized, it reads (IO-instructions) from the paper tape reader (device No. 0) the 9 procedures, and now it is able to communicate with the operator's typewriter.

The loader first writes:

channel no. of operator-key =

dev. no. of operator's typewriter =

and the operator types the interrupt channel No. of the operator key and the device No. of the typewriter he wants to use. (These numbers may be altered later by activating the RESET and then the START push-button after which the loader writes the above-mentioned questions again).

When after the message and question:

loader 2

input from device no.:

the operator has specified whether further inputs are wanted from the paper tape reader (device No. = 0) or from magnetic tape station (device No. > 0) (in the last case the typewriter writes:

file no.

and the operator must specify the file No. (chapter 1.4)), the test programs are loaded.

The loader now writes:

<name of the peripheral device>

After the questions:

device no.=

channel no.=

output dev. no.=

it reads the device number(s), the interrupt channel number(s) and the wanted output device number (> 0), it jumps with IM(operator-key) = 1, IR = 0 and interrupt enabled to the 9th procedure ('directory', chapter 1.3).

In case of interrupt No. 0 or when the operator-key is activated or in case of interrupt signal from the peripheral device (if the test program tests interrupt) a jump is performed to the loader's interrupt sequence, the start address of which is placed by the loader in word No. 12. A detailed description of this sequence is given in chapter 1.5.

The following rules apply to every program (a procedure or a test program) which is read and stored by the loader:

1. It is stored with the protection key = 0 so that every test is performed in the monitor mode.

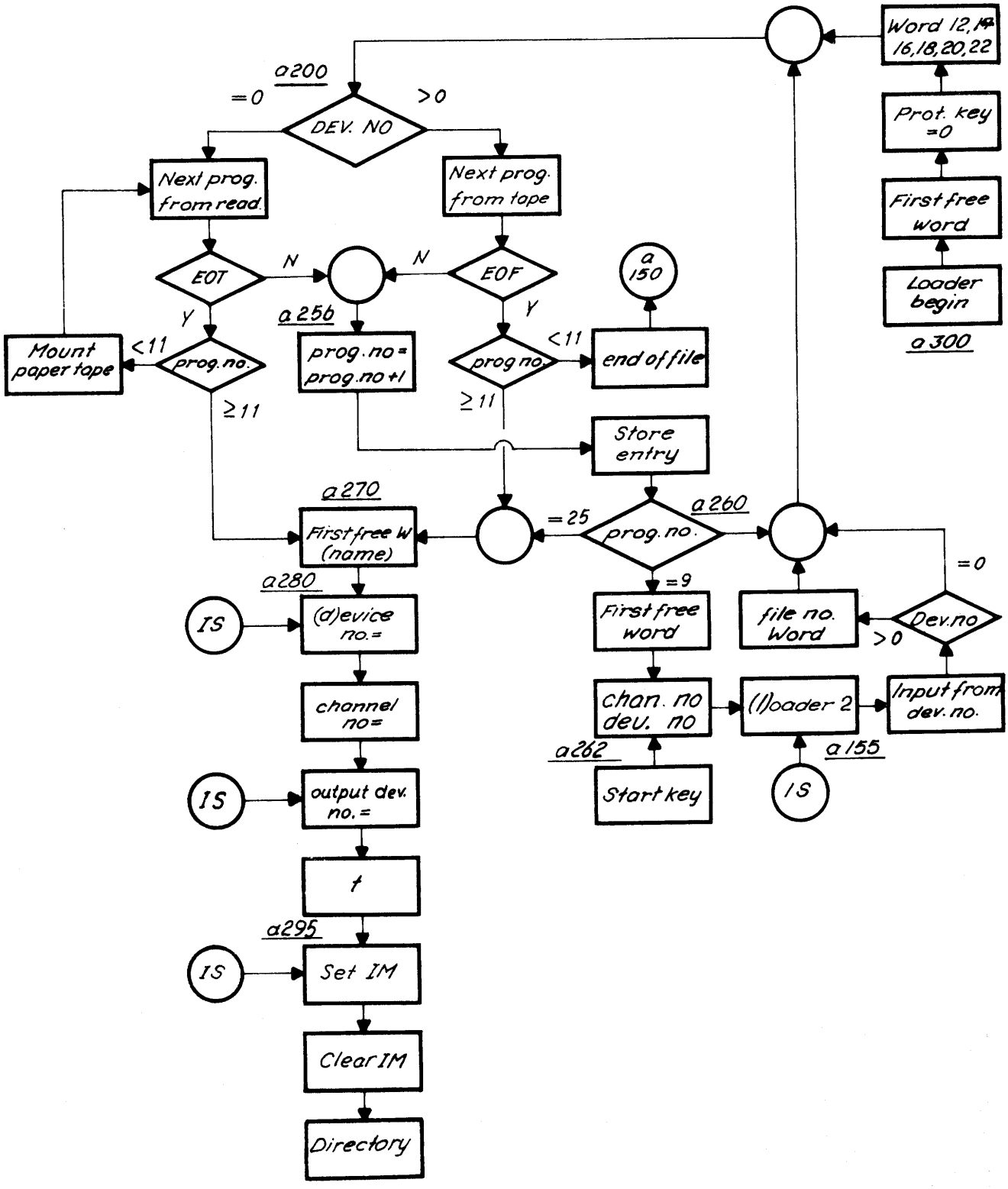
2. The parity and (when punched on paper tape) the check sum are examined, and in case of error, the following messages are given:

   parity error in <name of program>

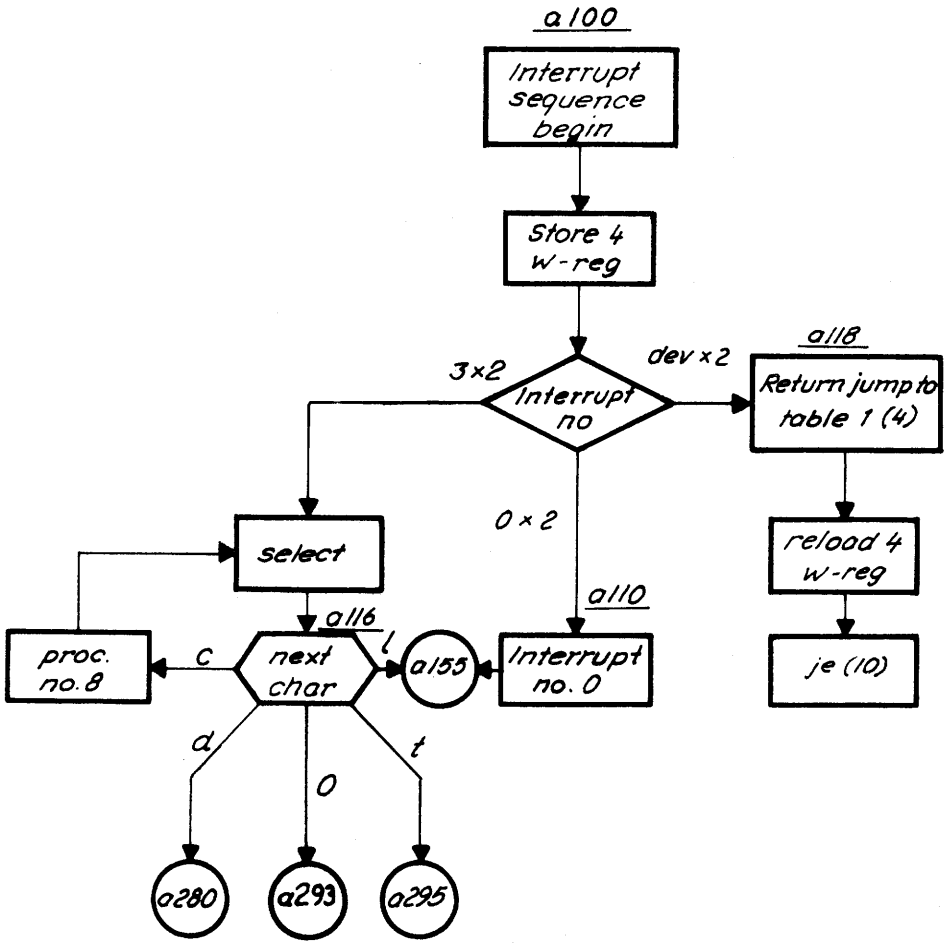
   check sum error in <name of program>

3. The first words of a program must contain a text string finished with the character <0> and giving a description of the program.

4. Entry point of the program must be the first word after this textstring.


Word 12,14,16,18,20,22
Prot. key =0
First free word
Loader begin
a 300
a200
=0
DEV. NO
>0
Next prog. from read
Next prog. from tape
EOT
N
EOF
N
a 150
Y
Y
a 256
Mount paper tape
<11
prog. no.
prog. no = prog. no +1
prog no.
<11
end of file
≥11
≥11
Store entry
a 270
First free W (name)
=25
prog. no.
a 260
=9
=0
a280
IS
(d)evice no.=
First free word
file no. Word
Dev.no
>0
channel no=
chan. no dev. no
(l)oader 2
Input from dev. no.
a 262
a 155
IS
Start key
IS
output dev. no. =
t
a295
IS
Set IM
Clear IM
Directory

THE LOADER 2



THE INTERRUPT SEQUENCE

1.3. THE PROCEDURES

The first programs which are read (by means of the IO-instruction) and stored by the loader are the belowmentioned 10 procedures which in the binary version are punched on the same paper tape as the binary loader:

1. write a text
2. write a decimal number
3. write a binary number
4. read one character from the typewriter.
5. read a decimal number from the typewriter.
6. read a binary number from the typewriter.
7. reserve buffer area.
8. write the contents of a part of the core store.
9. administrate the test.
10. compare 2 binary words, write the result.

Each of these procedures, together occupying 967 words, are described in detail one by one on the following pages.

1st procedure: write a text

This procedure writes a text; the text, which may consist of an arbitrary number of characters, must be finished with the character <0>.

input	output
(w0) = + or - text start address	(w0) = undefined
(w2) = return address	(w2) = undefined

It may be called in this way:

```
al.  w0    <text start>.
am           (18)
jl   w2      (+0)
```

if output on the operators typewriter.

```
ac.  w0    <text start>.
am           (18)
jl   w2      (+0)
```

if output on the spec. output device.

2nd procedure: write a decimal number

This procedure writes in decimal a 24-bit integer. The integer may be negative, zero, or positive.

input	output
(w0) = + or - address of integer	(w0) = undefined
(w2) = return address	(w2) = undefined

It may be called in this way:

```
al.  w0     <int. addr.>.
am          (18)
jl   w2     (+2)
```

if output on the operators typewriter.

```
ac.  w0     <int. addr.>.
am          (18)
jl   w2     (+2)
```

if output on the spec. output device.

3rd procedure: write a binary number part 1

This procedure writes the leftmost bits of a word.

input	output
(w0) = + or - word addr.	(w0) = undefined
(w1) = No. of bits	(w1) = undefined
(w2) = return address	(w2) = undefined

It may be called in this way:

```
al.  w0    <word addr.>.
al   w1    <No. of bits>
am         (18)
jl   w2    (+4)
```

if output on the operators typewriter.

```
ac.  w0    <word addr.>.
al   w1    <No. of bits>
am         (18)
jl   w2    (+4)
```

if output on the spec. output device.

4th procedure: read one character

This procedure reads one character from the operator's typewriter.

input	output
(w2) = return address	(w2) = status and char.

It may be called in this way:

```
am          (18)
jl   w2     (+6)
```

<SP> is treated as a blind character.

If a parity error occurs, the character is replaced by a slash.

5th procedure: read a decimal number

This procedure reads a decimal integer typed on the operator's typewriter. The integer, which may be negative, zero, or positive, must be followed by a terminator (that is an arbitrary character which is not a digit or a space).

input	output
(w0) = undefined	(w0) = integer
(w2) = return address	(w2) = terminator

It may be called in this way:

```
am          (18)
jl   w2     (+8)
sn   w2      10
sh   w0       0
jl.         -8
```

if the call demands an integer greater than 0 and a terminator equal to <NL>.

6th procedure: read a binary number

This procedure reads a positive binary integer typed on the operator's typewriter. The integer must be followed by a terminator (that is an arbitrary character which is not a 0 or a 1 or a space).

input	output
(w0) = undefined	(w0) = integer
(w2) = return address	(w2) = terminator

It may be called in this way:

```
am          (18)
jl   w2     (+10)
se   w2      10
jl.          -6
```

if the call demands a terminator equal to <NL>.

7th procedure: reserve buffer area

This procedure reserves a part of the core store and it writes on the operator's typewriter:

fbw = <address of first buffer word>

lbw = <address of last buffer word>

input	output
(w0) = No. of words wanted	(w0) = No. of words
(w2) = return address	(w2) = start address

If it was not possible to reserve the wanted number of words within the available core store, the output value of both w0 and w2 is 0.

The procedure may be called in this way:

```
al   w0     <No. of words>
am             (18)
jl   w2        (+12)
sh   w2          0
jl.
```

If the input value of w0 is equal to -(No. of words wanted) the procedure waits for input after having written <first buffer word>; if the operator inputs a character different from <NL> (for example /), it writes

fbw =

and waits for another start address. This address must be within the free part of the core store, i.e.

1) lower than the loader start address but not less than 24 (only significant when using the relocatable loader)
2) higher than the last address of the test programs.

If <fbw> is outside the free core the loader will request for a new <fbw>. After input from the operator it calculates and writes <last buffer word>.

8th procedure: write the contents of a part of the core store

This debug procedure is able to write on the operator's typewriter the contents of a specified part of the core store (from word No. 8) inone of four modes: text, decimal, binary, or machine instructions.

After an operator termination it is called when the operator types c; the core store area is specified when the procedure writes:

first word addr. =

last word addr. =

respectively, and the mode is selected when the operator types t, d, b, or i, respectively.
In case of d(ecimal) and b(inary) the program waits for input of an integer which specifies the print lay-out:

<answer>	lay-out
(new line)	24 bits
0 < <integer> <= 24	the word is divided into <integer> No. of bits from the left and printed. If d(ecimal) and if 24 mod <integer> ≠ 0 then print the rest word as a binary number.
<integer> = 0	24 bits

The procedure may be called in this way:

```
am          (18)
jl  w2      (+14)
```

9th procedure: administrate the test ('directory')

Immediately after the selection of the device No. (and the interrupt channel No.) for the peripheral device and the output device, a jump from the loader to 'directory' is performed. In this procedure the test program and the number of runs are selected, and furthermore the procedure is able to write on the operator's typewriter a description of the stored test programs.

After the question:

test program:

the operator may type a letter: a, b, c, ... and in this way select the 1st, 2nd, 3rd, ... test program, or he may type <NL>, after which the procedure writes the following directory:

a <description of the 1st test program>

b <description of the 2nd test program>

c <description of the 3rd test program>

.
.
.

<description of the last test program>

These descriptions are fetched from the first words of each program (chapter 1.4.).

When the operator has answered the question:

number of runs =

the test program is called 0th, 1st, 2nd, ..., last time. During call No. 0 the test program is initialized. At each call the return address is placed in w2, while

w1 (22) = last call
w1 (23) = 0th call

involving that run No. 0 and last run may be selected in this way:

```
sz   w1      1
jl            ; run No. 0 (initiate)
```

and

```
sz   w1      2
jl            ; last run (finish).
```

Before some runs 'directory' writes:

run no. <run No.>

that is before

1st, 2nd,..., 9th run, if 1 ≤ No. of runs ≤ 9
1st, 11th, 21st,..., 91st run, if 10 ≤ No. of runs ≤ 99
1st, 101st, 201st,..., 901st run, if 100 ≤ No. of runs ≤ 999

etc., so that a test is always introduced with the message

run no. 1

and so that a message is sent each time such a number of runs are executed that:

this number = the greatest 10-power which is less or equal the specified number of runs.

Having performed the wanted number of runs, the procedure writes:

test end

after which a new test program may be selected.

If the testporgram wants to finish the test before <No. of runs> are exceeded, it may return to the directory with return address:= return address +2. I.e. if w2 contains the return address the <test end> may be executed in the following way:

```
jl      x2 + 2
```

10th procedure: write a binary number part 2

This procedure compares two binary words and writes some of the leftmost bits in the following way: the two words are called 'received pattern' (rec.) and 'expected pattern' (exp.) respectively; the two words are compared bit by bit and if they are equal the value is written else one of the two letters ø or x is written:

ø if the bitvalue in rec. is 0 (a wrong zero),

x if the bitvalue in rec. is 1 (a wrong one).

input	output
(w1) = + or - table address	(w1) = undefined
(w2) = return address	(w2) = unchanged

It may be called in this way:

```
al.  w1    <table addr.>.
am            (18)
jl   w2      (+18)
```

if output on the operators typewriter.

```
ac.  w1    <table addr.>.
am            (18)
jl   w2      (+18)
```

it output on the spec. output device.

```
<table address> + 0:  <blocksize> <12 + <No. of bits>
                + 2:  bit pattern <received>
                + 4:  bit pattern <expected>
```

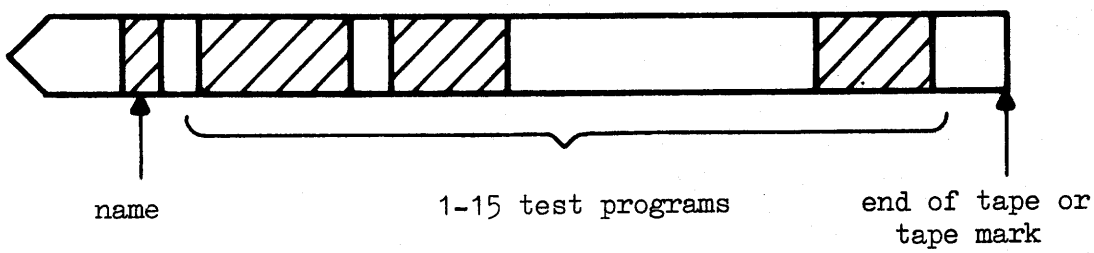
where <blocksize> denotes the number of bits to be printed before a <space>; if <blocksize> <=0 or <blocksize> > = <No. of bits> no <space>'s are printed. <No. of bits> denotes the total number of bits to be printed counted from left to right.

1.4. TEST PROGRAMS.

For each kind of peripheral devices mentioned below is made a set of test programs:

	File No.
RC 2000 Paper Tape Reader	1
RC 150 Paper Tape Punch	2
RC 315 Typewriter	3
RC 610 Lineprinter, Data Products	etc.
RC 333 Lineprinter, Anelex	
RC 4191 Plotter	
RC 707 Magnetic Tape Station, 7 tracks	
RC 709 Magnetic Tape Station, 9 tracks	
RC 4415 Drum	
RC 4314 Disc	
Interval Timer	
Teletypewriter	
DPC401 Display	

The test programs in the binary version exist both on paper tapes (so that all the programs for one kind of peripheral devices are punched on one paper tape) and on 7- or 9-track magnetic tape (so that each program forms one block, and so that all the programs for one kind of peripheral devices form one file. In both cases the parity is odd.



This drawing shows a paper tape or a file on magnetic tape containing the binary test programs for one kind of peripheral devices.

For each program (test program or procedure), which is read and stored by the loader, the following rules apply:

1. The first 15 words (that is the first 45 ISO-characters) must contain a text. This text is used by the loader in the message in case of parity error and check sum error, and it is used by the 9th procedure when writing the directory.

2. The 16th word must be the entry point.

3. The program must be finished with a check sum when punched on paper tape.

The paper tape/the file first contains a 15-word program containing the name of the kind of peripheral devices, for example:

<:rc 707 magnetic tape station, 7 tracks<0> :>

This is the text which is written by the loader immediately after the test programs are stored.

After the name follows a number of test programs in arbitrary succession; if the number exceeds 15, only the first 15 are loaded.

At the jump from 'directory' to a test program w2 contains the return address and

if 0th call then w1(23) = 1
if last call then w1(22) = 1

(the other bits are all 0) so that the test program may initiate and finish the test.

The test programs are divided into two groups:

Test programs:

1. Checking programs for critical check of the device.
2. Motion programs for uncritical use of the device.

Within each group for a given kind of peripheral devices the programs are successively numbered: 1.1, 1.2, ..., and 2.1, 2.2, ...

By means of the checking programs the complete, critical test of a peripheral device is performed. If the device does not react in the expected manner, messages mentioned in chapter 3 are given. The test of

sense, control, read, write
exception register
interrupt signals
status
data

is included in these programs. It is a principle that whatever happens, the test is going on. For example, the absence of an interrupt signal or even a disconnected device causes a message to the operator, but the test continues; but the operator may break the run by activating the operatorkey.

The exception register and the interrupt signal are tested as shown on the next page and as explained below:

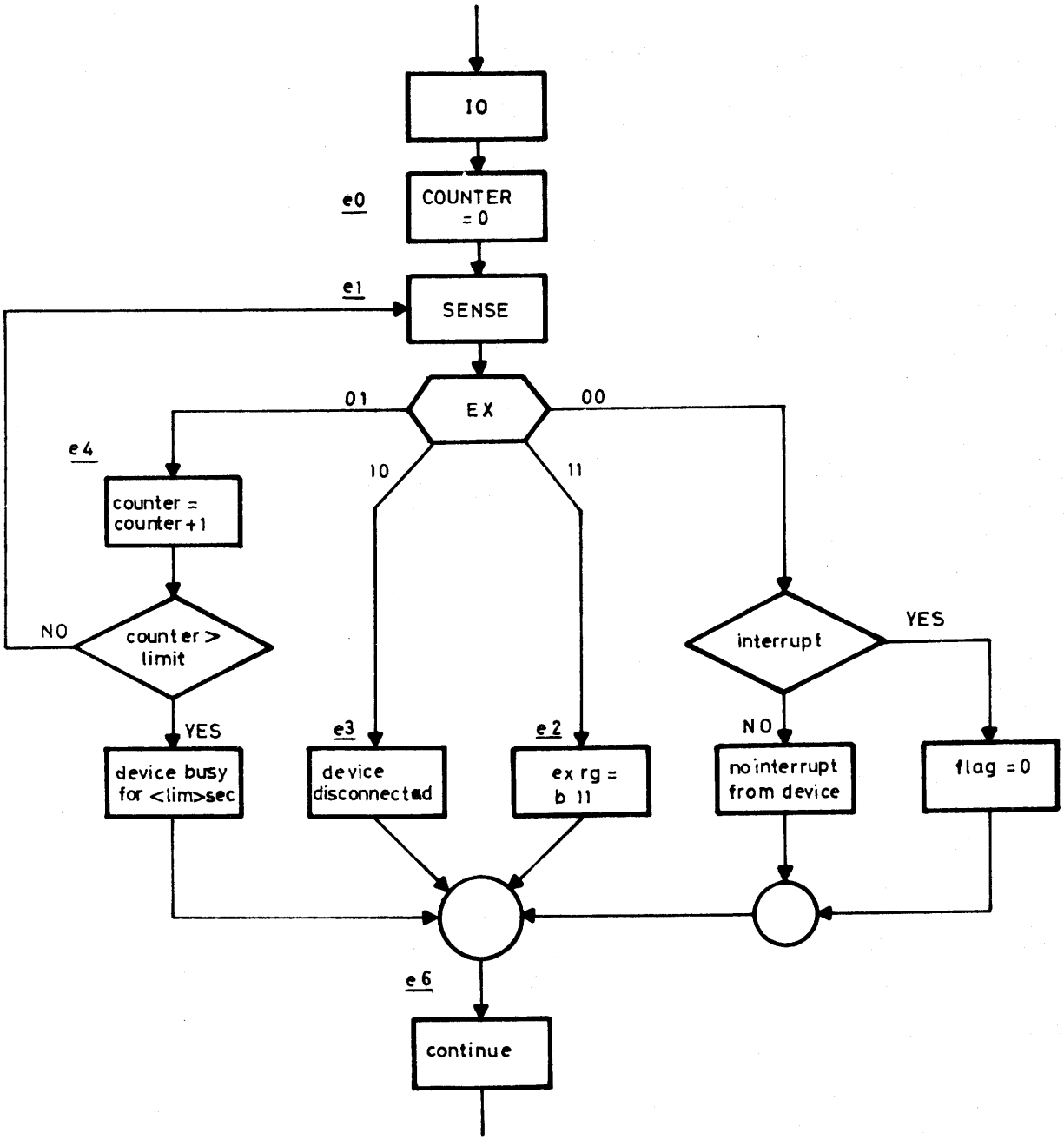
EX = 00: The device is available and, if it has sent an interrupt signal, the test continues; contrary this message is written (and the test continues):

no interrupt from device

EX = 01: The device is busy either because the transmission has not yet finished (especially because the device is in the local state) or because of a hardware error. If the device remains busy for a time depending on the kind of device, the program writes:

device busy for <time> sec.

and the test continues.


IO
e0
COUNTER = 0
e1
SENSE
EX
01
00
10
11
e4
counter = counter+1
NO
counter > limit
YES
device busy for <lim>sec
e3
device disconnected
e2
ex rg = b 11
interrupt
YES
NO
no interrupt from device
flag = 0
e6
continue

EX = 10: The device is disconnected either because of an operator oversight or because of a hardware error. After the message:

device disconnected

the test continues.

EX = 11: This is a hardware error which involves the message:

exception reg. = b11

After this the test continues.

Motion programs are commonly used when checking programs have shown an error. These programs use the peripheral devices in an uncritical way, that is they do not apply interrupt signals, the status word is not examined, and they hardly ever send error messages to the operator. In this way and by selecting a great number of runs it is possible to encircle the error, for example by oscilloscope measurements.

Each kind of peripheral device may be tested by means of a core store of minimum size that is 4096 words, but for high-speed devices it is possible to place the input-output buffer anywhere in the free part of the available core store.

The testprograms for high-speed devices are so designed that they propose the buffer start address by writing:

fbw = <address of first free word>

and wait for input. If the operator types <NL>, he accepts the start address; if he types a slash, the programs write:

fbw =

and he must input another start address. After input from the operator the address of the last buffer word is calculated and written. (ref. 7th procedure: reserve buffer area).

1.5. INTERRUPTION.

Interruption (that is interrupt signals, the interrupt register (IR), the interrupt mask (IM), and interrupt enabled/disabled) is applied in the following way:

When the loader is stored and executed and when the interrupt sequence is executed, interruption is disabled so that only interrupt No. 0 causes interruption. At all other times, that is during the execution of test programs and procedures, interrupt is enabled.

At the jump from the loader to directory the interrupt register is cleared, so that old signals from the operator key or other peripheral devices should not cause interruption. During the execution of directory IM(0) = IM(operator key) = 1, involving that only interrupt No. 0 or the use of the operator key causes interruption.

At the jump to a test program applying interrupt signals from the peripheral device furthermore IM (interrupt channel No.) is set to 1. (This mask is constructed by the test program when initiated; at the same time the test program stores the start address of its own subinterrupt sequence in word No. 4 of table 1 in the loader). So for these test programs interrupt signals from the peripheral device cause interruption. At the return jump from test programs the interrupt mask of directory is reloaded.

The interrupt sequence is placed inside the loader. Its start address is stored in word No. 12 by the loader when initiated. It is shown in the flow chart in chapter 1.2, and is now further described.

At the jump to the interrupt sequence, interrupt is disabled. Such a jump is performed in the following situations:

1. Interrupt No. 0 which may always occur. If the interrupt sequence and the first procedure are not destroyed, this message is written:

   interrupt no. 0

   If this occurs (due to a hardware- or software-error), the loader should be stored again.

2. By activating the operator key during the execution of a test program, directory or one of the other procedures. The interrupt sequence writes:

select

after which the operator may type t, o, d, l, or c. The typewriter now continues to write one of the following underlined texts:

test program:

A jump to directory is performed, and the operator may now select a new test program for the same device No.

output dev. No.=

and the operator may select a new output device number > 0. Hint: when using the cpu-timer as output dev. No. even checking programs may be used as motion porgrams. This can only be used for testprograms newer than medio 1971.

device no.

A jump to the statement in the loader where the device number(-s) is(are) selected. In this way the operator may select a new device No. (and after this a new interrupt channel No.) for the same kind of peripheral device

loader 2

A set of test programs for a new kind of peripheral device may be loaded.

core store contents

A jump to procedure No. 8 (chapter 1.3) is performed.

3. At interrupt signals from the peripheral device during the execution of a test program which applies interrupt. In this case a return jump is performed to the subinterrupt sequence of the test program with interrupt still disabled. w2 contains the return address. The start address of this sequence is stored by the test program when initiated in word No. 4 of table 1 in the loader. The contents of the 4 W-registers and the exception-register are stored by the loader's interrupt sequence, and the registers are reloaded just before a jump with interrupt enabled is performed to the broken test program.

1.6. BITPATTERNS.

In some test programs (for example RC 2000 1.2 and RC 150 1.2) a bitpattern consisting of 304 8-bit characters is generated. The decimal values of these characters are shown in the table on the next page.

The character set has the following proporties:

1. Except for 8 zeroes and 8 ones it contains (one or more times) all characters which are composed by 8 bits.

2. During the generation of the characters row by row each of the 8 bitpositions is activated in a very irregular way.

3. The programming of the generation is rather simple.

1	2	4	8	16	32	64	128
3	6	12	24	48	96	192	129
5	10	20	40	80	160	65	130
9	18	36	72	144	33	66	132
17	34	68	136	17	34	68	136
7	14	28	56	112	224	193	131
13	26	52	104	208	161	67	134
11	22	44	88	176	97	194	133
25	50	100	200	145	35	70	140
19	38	76	152	49	98	196	137
21	42	84	168	81	162	69	138
41	82	164	73	146	37	74	148
15	30	60	120	240	225	195	135
29	58	116	232	209	163	71	142
27	54	108	216	177	99	198	141
53	106	212	169	83	166	77	154
51	102	204	153	51	102	204	153
45	90	180	105	210	165	75	150
85	170	85	170	85	170	85	170
170	85	170	85	170	85	170	85
210	165	75	150	45	90	180	105
204	153	51	102	204	153	51	102
202	149	43	86	172	89	178	101
228	201	147	39	78	156	57	114
226	197	139	23	46	92	184	113
240	225	195	135	15	30	60	120
214	173	91	182	109	218	181	107
234	213	171	87	174	93	186	117
236	217	179	103	206	157	59	118
230	205	155	55	110	220	185	115
244	233	211	167	79	158	61	122
242	229	203	151	47	94	188	121
248	241	227	199	143	31	62	124
238	221	187	119	238	221	187	119
246	237	219	183	111	222	189	123
250	245	235	215	175	95	190	125
252	249	243	231	207	159	63	126
254	253	251	247	239	223	191	127

The characters are generated by cyclic shifts and complementation of the 19 8-bit characters shown below.

These characters consist of 1, 2, 3, or 4 ones. After 8 cyclic shifts

$$19 \times 8 = 152 \text{ characters}$$

are obtained, among which 6 + 4 + 4 = 14 characters are doublets (from the patterns marked with ‡ and ‡‡), i.e.

$$152 - 14 = 138 \text{ different characters.}$$

By complementation of each of these

$$138 \times 2 = 276 \text{ characters}$$

are achieved, among which 8 + 4 + 8 + 2 = 22 characters are doublets (from the patterns marked with ‡‡‡), so the result is

$$276 - 22 = 254 \text{ different characters}$$

i.e. the characters 1 to 254. It is noticed that the 2 characters 0 and 255 are not included.

1	00000001			
2	00000011			
3	00000101			
4	00001001			
5	00010001	‡‡)		
6	00000111			
7	00001101			
8	00001011			
9	00011001			
10	00010011			
11	00010101			
12	00101001			
13	00001111		‡‡‡)	
14	00011101			
15	00011011			
16	00110101			
17	00110011	‡‡)	‡‡‡)	
18	00101101		‡‡‡)	
19	01010101	‡)		

‡) the pattern is repeated after 2 cyclic shifts.

‡‡) the pattern is repeated after 4 cyclic shifts.

‡‡‡) the pattern is repeated after complementation and cyclic shifts.

In the test program RC 707 1.1 the generation of 126 bitpatterns is based upon the shift and complementation of the following 9 7-bit characters:

1	0000001
2	0000011
3	0000101
4	0001001
5	0000111
6	0001101
7	0001011
8	0011001
9	0010101

After 7 cyclic shifts

$$7 \times 9 = 63 \text{ different characters}$$

are optained.

After complementation of each of these

$$63 \times 2 = 126 \text{ different characters}$$

are achieved, i.e. the characters 1 to 126. The 2 characters 0 and 127 are not included.

In the test program RC 709 1.1 and 1.4 the generation of 522 bitpatterns is based upon the shift and complementation of the following 29 9-bit characters:

1	000000001
2	000000011
3	000000101
4	000001001
5	000010001
6	000000111
7	000001101
8	000001011
9	000011001
10	000010011
11	000010101
12	000110001
13	000101001
14	000100101
15	001001001 *)
16	000001111
17	000011101
18	000011011
19	000010111
20	000111001
21	000110011
22	000100111
23	000110101
24	000101101
25	000101011
26	001101001
27	001011001
28	001100101
29	001010101

*) the pattern is repeated after 3 cyclic shifts.

After 9 cyclic shifts

$$29 \times 9 = 261 \text{ characters}$$

are optained, among which 6 are doublets, i.e.

$$261 - 6 = 255 \text{ different characters.}$$

By complementation of each of these

$$255 \times 2 = 510 \text{ different characters}$$

are achieved, i.e. the characters 1 to 510. It is noticed that 2 characters 0 and 511 are not included.

1.7. THE RELOCATABLE LOADER_2

The relocatable loader 2 consists of the abovementioned loader and procedures, however so designed that the relocatable loader and the testprograms may be stored everywhere within the available core store, if the operator before activating the AUTOLOAD push-button puts the start address into w3. This start address must be so chosen that

$$0 \leq w3 \leq \text{length of core store} - (\text{length of relocatable loader} + \text{length of testprograms})$$

All lengths are measured in No. of bytes. The length of the relocatable loader is 2980 bytes.

When w3 = 0, the loader and the testprograms are stored as usual (see Chapter 1.1, page 5).

RCSL : 44 RT 1083
Author : Per Hansen
Edited : June 1975.

TEST OF RC 4818 DISC .

Key : RC 4000, Hardware testprogram, RC 4818.

Contents :

General information.

0.1 Sense Commands and status.

The sense command

```
IO      W      dev < 6 + 0
```

yields the status 0 when the controller is ready and connected .
The meaning of the statusbits is :

Bit no.	Meaning.
0	Intervention
1	Parity
2	Timer
3	Data overrun
8	Disc in local
9	Pack unsafe
10	Synchronication error
11	Heads moving
12 : 15	Command register
16 : 23	Unit selected

The sense command

```
IO      W      dev < 6 + 32
```

yields the status 32 when the controller is connected. Note that the transfer is independent on the ready/busy state. The controller simulates ready with regard to this command even when it is busy. The statusbit have the following meaning :

0 : 14	CR (0 : 14), control register
15	Wrong 2nd index
16	Address error
17	Drop out
18	Seek error
19	Pack unsafe
20 : 23	Unit register

0.2 Control Commands

Control Command Modifications

The control commands are used to specify and initiate cylinder selections and data block transfers. The disc controller accepts 12 modifications of the control command:

5 transfer first	<cyl,head,sector>
9 transfer size	<number of segments>
13 input data	<first storage address>
17 output data	<first storage address>
21 input address	<first storage address>
25 output address	<first storage address>
29 select disc	<disc number>
33 return to zero	<irrelevant>
37 transfer forward	<abs (cylinder difference)>
41 transfer reverse	<abs (cylinder difference)>
45 set mode	<mode>
61 master clear	<irrelevant>

The integers denote the values of bits 18-23 in the effective address of the input/output instruction. The parameters in the brackets < and > denote the contents of the working register selected by the input/output instruction.

The parameters are interpreted as follows:

< disc number >	modulo 16
< abs (cylinder difference) >	modulo 512
< cylinder, head, sector >	modulo 512, 32, 16
< number of segments >	modulo 512
< first storage address >	modulo 262144
< mode >	modulo 16

0.3 Interrupt number specification.

When the loader asks for interruptnumber, the data interrupt number and then the head interrupt number must be specified.
The data interrupt is connected to the latter, the head interrupt to the former channel (the one nearest the CPU on the bus) of the controller.

0.4 Selected output device.

The loader's question

output dev. no.

should be answered with the number of the printer, if available, otherwise the typewriter. On this device all error messages will appear.
If however, the messages are undesired, they may be suppressed by specifying the timer (device no. 3) as output device.

Testprogram description.

1.3 Segment addresses.

Purpose.

The purpose of this program is to perform and test the storing of segment addresses on the disc. Furthermore the exception register, the interrupt signals and the statuswords are tested.

This program is also used (together with program 1.4) to initialize a virgin disckit before put into operation.

Method.

The program writes or reads a sequence of addressmarks on selected tracks. It is possible to skew the addressing from cylinder to cylinder to match the time which the heads need for changing cylinder. The number of addressmarks written is restricted to full tracks since it is not possible to write addressmark on part of a track.

Caution

When addressmarks have been written, the contents of the datasegments are not initialized and must be overwritten by means of e.g. program 1.4 before the kit is put into operation

Initiation.

Typing of numbers.

Numbers typed to this program are interpreted as radix numbers, that is the number base can be specified in front of the number separated with a point. if no radix is specified the number is interpreted as a

decimal number.

Example : the number 2.1100000000

equals the number : 10.768 or just : 768

This feature is useful when specifying parameters with the purpose of loading special bitpatterns and powers of two.

First the program asks the following questions, and the operator answers as shown :

select disc =	($0 \leq n \leq 7$)
first segm =	($0 \leq n \leq 73079$)
no. of segm =	($9 \leq n \leq 73070$) or m(ax)

When m is specified the maximum number of segments is used. Current versions of the program however, use 36540 as max value.
The answers to question 2 and 3 must be numbers which are integer multiples of 9. Otherwise the number is rounded off and a correction message is typed on the console.

Now the program types

fbw = 9690 (if normal loader2 is used).

which may be acknowledged with a < NL >.
If any other character is typed, the first buffer word address must be typed by the operator. The program only reserves 9 words independent on the number of segments under test.
Selection of various values of the fbw throughout the corestore is used to check the highspeed channel addressing circuits.

When the program asks :

no. of shifts per cyl. =

the operator types a number which normally is 3. Any other may be used, but 3 shifts will be optimum during normal operation.

The question :

write or read ?

must be answered y(es) or n(o) and the test enters run no. 1.
If write is selected, the write operation automatically will be followed by a read operation on the segments specified.
Before any read operation the buffer is cleared.

Error messages.

The first run is initiated by issuing a masterclear command followed by a select disc command. Now a sense operation is performed, and it is checked that one and only one bit is sat in unitselected (bit 17 : 23) in statusword 0. If an error is detected a message like this

run 1 , segm. 0 , unit 0 0 0 0 0 0 0 0

will be output.
Further, status 0 is examined, and in case of error both status 0 and status 32 is typed out. To save space, wrong bits are indicated with ϕ or x , where ϕ means a 1 mutilated into 0 and x means a 0 mutilated into 1 (ϕ and x symbolizes 0 and 1 with a slash indicating that they are wrong).

Example :

run 1, segm., 0,select-status 0 : 00000x000000 0$\phi\phi\phi$00000000
run 1, segm., 0,select-status 32: 000001000000 000000000000

In the same way move-head, read and write operations are checked and in case of statuserror the message is identified in one of these ways :

move - status

read - status

write - status

After a read operation a dataerror may occur, in this case a message identifying runno., segment no. dataerror and core address is output.

Example :

runno. 10, segm. 8, addr. ϕϕϕ011xx00ϕ0 1x11x1000ϕ00 core addr 4224

Note :Before the read operation is initiated the buffer is cleared. This means that a message of 0's and ϕ's only may be due to the addressmarks not being received or received in a wrong place in the core store.

After any senseoperation the exception register is examined, this may cause the message :

*** dev. disconnected

Now a masterclear command is sent to the controller and the registers are initialized anew. When (if) the controller turns connected the message :

dev. connected

is output.

Time consuming commands (datatransfers and head moves) are checked for maximum duration. The messages and limits (milliseconds) are :

*** max busy (> 5000 mS)
*** max head move (> 500 mS)

The interrupt signals are checked for arriving at the correct time. If no interrupt is received when the controller goes ready or head move is finished, the messages :

no {head / data} int.

is output.

If an interrupt is received during busy or while heads are moving it will cause this message :

ill. busy {head / data} int.

An unexpected interrupt (e.g. datainterrupt after headmove or vice versa) is indicated with

ill. ready {head / data} int.

If the same interrupt is received more times the message

multi {head / data} int.

is output.

Any of these messages are accompanied by the usual identification of runnumber and segmentnumber.

All errormessages will be output on the outputdevice specified to the loader.

1.4 Read and write.

Purpose.

This program is the main test for the disc. It is intended for use when other tests are running ok and a long-term reliability testing is desired.

After any operation the exception register, interrupt and statusword are examined.

The program is designed in a way which makes it easy for the operator to :

1. test the addressing of the disc.
2. test the addressing of the core store.
3. perform a critical test of the data chain, including write modulator, read amplifier, head selection and read detector.
4. perform a critical test at the surface of the magnetic disc pack.
5. trace the origin of any error.

Overview of operation.

The operation of the program allows the user considerable flexibility, as described below. But a typical run might be as follows. The program first writes test data into an area of the disc specified by the operator. Then it reads this data back and checks that it has been read correctly. It does not immediately report errors discovered in the readoperation. but keeps a record of errorsto be printed out later.After the read operation is completed or a hard error is met, the program prints out a table showing in which segments data errors and status errors have occured. Then it prints out a list of precisely what data errors occured, giving segment number , word number, expected value and received value for each data error. Because the list of data errors

can easily become unmanageably long, the user may specify a limit on how many dataerrors are to be reported for each segment. The test data used may be any of three kinds : (1) constant plus word number , (2) cyclic concatenated constants, (3) cyclic lippel code. These test data kinds are described in detail further on.

There are three addressing modes available for both the write operation and the read operation. The segments may be read or written : (1) forward, from first segment to last, (2) reversed, from last to first, (3) alternating, shifting back and forth between high and low ends of the area. The alternating address mode makes a really brutal test of disc-head motion, but it is of course much slower than the other addressing modes.

The write operation may be omitted if it is not desired. The read operation and its accompanying error printout may be performed from 0 to 8 times per run as specified by the user. This means that a dispack may be tested as follows :
The discpack is mounted on one disc drive, data is written on it, but not read back, the disc pack is moved to another drive, the data is read back and checked.

Initiation.

First the program asks :

mode =

and the operator may answer one of these :

c (hange)
i (nitiate)
r (un)
t (able)

If run is selected, the test proceeds using a set of parameters which are sat to a standard value unless changed by means of initiate.
The contents of these parameters (15 items) may be output on the selected output device by means of the table command.
By means of change or initiate one or more parameters may be changed.
If change is used the parameter changes will be considered temporary, and the parameters will be reset when the runs specified have been executed. If the initiate command is used, the new parameters will remain until changed again by initiate or until the test is re-input.
When the change, initiate or table command has been executed, the program returns to the mode = situation.
When change or initiate has been selected, the operator must type one or more numbers (separated by commas) specifying the parameters to be altered, or he may type

a (ll)

The parameters and answers are :

(1)	write oper.?	y (es) or n (o).
(2)	no. of read oper. per run :	$0 \leq n \leq 8$
(3)	first segm. for test =	$0 \leq n \leq 73079$
(4)	no. of segm. for test =	$1 \leq n \leq 73080$

(5)	blocksize writing =	(amount of segments, max. no. depends on core size)
(6)	double buf. writing ?	y (es) or n (o)
(7)	addr. mode writing :	f (orward), r (everse) or a (lternating)
(8)	blocksize reading =	(like param. no. 5)
(9)	double buf. reading ?	y (es) or n (o)
(10)	addr. mode reading :	(like param. no. 7)
(11)	test data kind =	1,2 or 3 (see below)
(12)	max. no. of messages of word level per segm. :	$0 \leq n \leq 256$
(13)	disc unit :	$0 \leq n \leq 7$

If test data kind 1 is selected, the program continues,

constant plus wordnumber.

constant = (any number, either a decimal integer or with radix as described in 1.3).

Or if kind = 2, then the program continues,

cyclic concatenated constants.

constants = (from 1 to 16 numbers, separated by commas and terminated by a new line).

Or if kind = 3 the program continues,

cyclic Lippelcode.

window width = ($3 \leq$ windowwidth ≤ 11)

If the test data kind is 2 or 3, the program furthermore asks for :

number of shifts per run = (| shifts | ≤ length of standard cycle in bits).

which determines the number of bits the testdata are cycled for each run.

The three kinds of testdata are described further on.

During setting of the parameters the program checks for consistency e.g. if writemode = no attempt to reserve write buffers is rejected.

When the question

fbw =

has been answered the program reserves the space needed in the core store. If the space is too little the program automaticly switches back to redefinition of the buffersizes etc.

When the test is loaded, the parameters are initialized to the following values :

(1)	write oper. ?	yes
(2)	no. of read oper. per run :	1
(3)	first segm. for test =	0
(4)	no. of segm. for test =	36540
(5)	block size writing =	4
(6)	double buf. writing :	yes
(7)	addr. mode writing :	alternating
(8)	block size reading =	5
(9)	double buf. reading ?	yes

(10)	addr. mode reading :	alternating
(11)	test data kind =	3
	cyclic lippel code	
	window width =	5
	no. of shifts per run =	48
(12)	max. no. of messages on	
	word level per segm. :	16
(13)	disc unit =	0

Notice that no. of segm, for test (4) after loading is 36540 corresponding to 9 megawords.

Test.

With the above initiation out of the way, the program runs as follows.

If the write operation was requested (question 1 above), the selected type of testdata is written in the selected area of the disc (question 3 and 4). This is done with the selected blocksize (question 5) except possibly for a short block in the top of the area. The sequence in which writing is performed is either forward, reverse or alternating (question 7) as described later.

Then as many read operation as specified (question 2) are performed. Each read operation is as follows :
The whole area is read in the blocksize specified (question 8), except for a possible short block at the top of the area. The sequence is as specified (question 10). Each block read is checked against what is expected. Any errors found, along with any status errors in reading, are recorded in a table in the free space in core of the program. Then when the read operation is completed a list of errors is printed out as described later. If a hard error is met, it will be reported immediately, the error table will be output and the read operation terminated, that is the test starts from the beginning. Hard error is either busy, disconnected, timer status or synchronization error status.

After all the read operations have been performed if the test data kind is 2 or 3, the standard cycle is rotated left by the amount specified (question 11, third parameter) in preparation for the next run.

Timing.

The following table gives the times in minutes and seconds for reads and writes of all 73080 segments of the disc. No errors occurred in the trials.

	Single-buffered		Doubble-buffered	
	writing	reading	writing	reading
forward	9 : 30	9 : 35	6 : 25	6 : 35
reverse	9 : 00	9 : 00	5 : 50	6 : 10
alternating	14 : 00	15 : 30	10 : 00	10 : 00

All operations were with blocksize equal to ten.

Test data kinds.

If the test data kind (question 11 above) is 1 the contents of each word written is a specified constant (the question constant =) plus the " wordnumber " of that word in the disc. The "wordnumber " is

256 x segment number + wordnumber in segment

where " wordnumber " in segemnt runs from 0 to 255.

If the test data is 3, the data is again a sequence of constants repeated throughout the disc. But this time the sequence is a lippel code generated with the specified window width (question 11, third parameter).

See W.W.Peterson, Error-Correcting Codes, John Wiley and Sons, 1962 for a description of Lippel Codes.
Briefly, this is a sequence of (2 ** window width) bits with the property that, if it is considered cyclic, then every possible sequence of bits (window-with) - long is a subsequence of it.

The sequence of words used is a concatenation of three lippel sequences generated with the specified window width. This concatenation of three is just to get a sequence of bits whose length is divisible by 24.

Addressing modes.

The three addressing modes available are forward, reverse , and alternating. The diagram on the following page shows the order of writing for each of these addressing modes.

In order to describe these addressing modes precisely in words, we first need to define some concepts. To read or write an area on the disc at a given blocksize, the number of blocks is

$$n = (s + b - 1) // b$$

where

s = number of segments in area
b = block size

One of these blocks may be a short block, with size

$$k = s \bmod b < b$$

In this program the n io operations are divided into n blocks, with the short block, if any, at the top of the area (highest segment addresses). This is true regardsless of addressing mode.

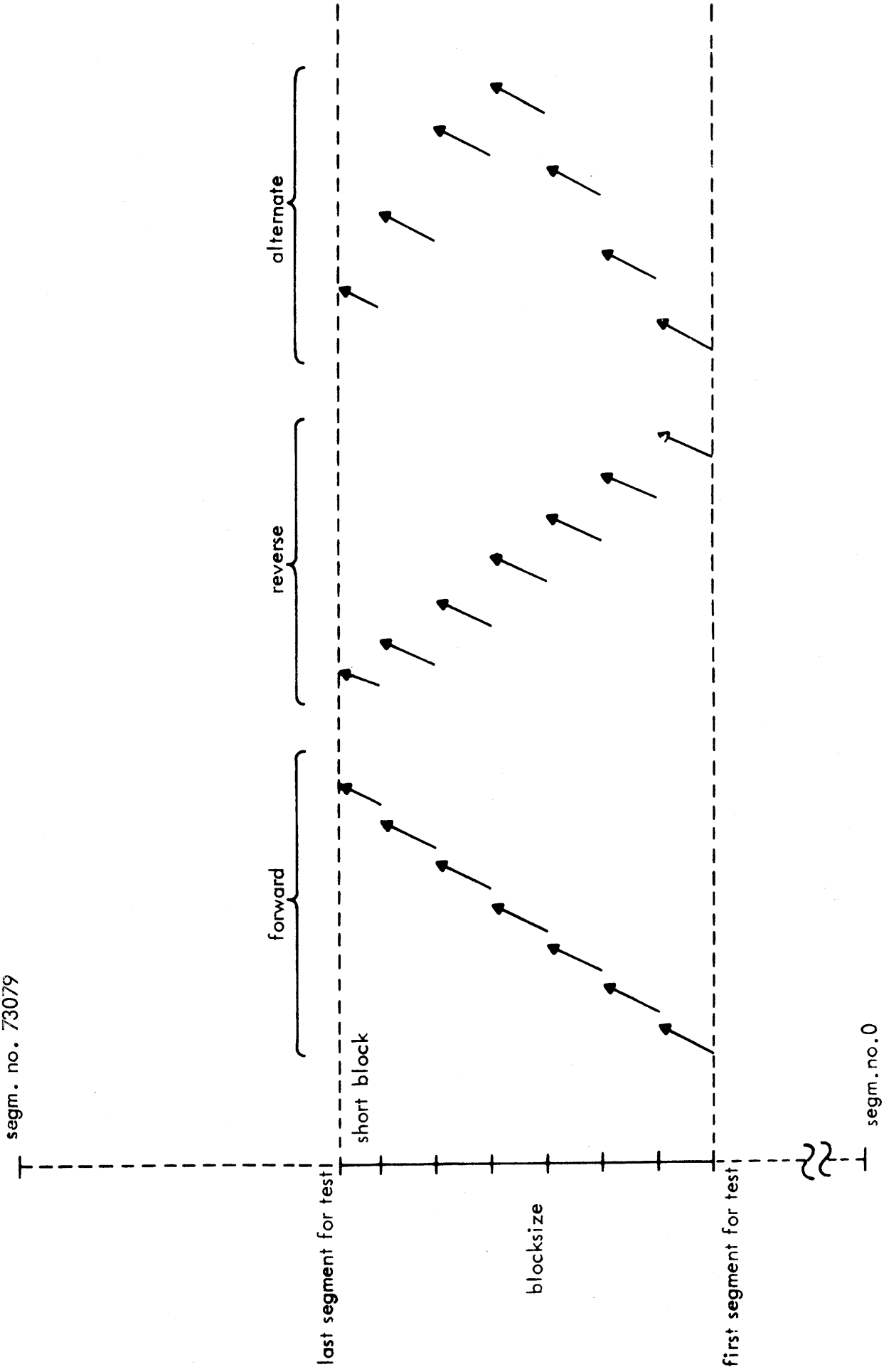
In the forward addressing mode, these blocks are written or read starting from the block at the lowest address and continued upward.

In the reverse addressing mode, the blocks are written or read starting from the block at the highest address (that is, the one which may be a short block), and continuing downward.

In the alternating addressing mode, the first block written or read is the one at the lowest address in the area (the one which may be short). The third is the next-lowest block, etc, as shown in the diagram.

Errormessages, statuserror and dataerrors

All errormessages will be printed on the selected outputdevice (as specified to the loader). When statuserrors or dataerrors are reported,


segm. no. 73079
forward
reverse
alternate
short block
last segment for test
blocksize
first segment for test
segm. no. 0

bad bits are indicated by ϕ or x, when ϕ means a 1 mutilated into a 0 and x means a 0 mutilated into a 1. (ϕ and x symbolizes a 0 and a 1 with a slash indicating that they are wrong).

If statuserrors are detected after a writeoperation or a head move operation or a hard statuserror (timer or sync. error, bit 2 or bit 10) after a read operation a message giving run number, segment number and statuswords is output.

example :

run 1, segm. 90 *** status 0	0000000000x0 001110000000
run 1, segm. 90*** status 32	100000000000 000001000000

A hard status after read will cause termination of the current read operations.

If one or more errors are detected during the input from the disc, a scheme (an example of this is shown on page 25 is delivered on the selected output device. One line is printed for each quarter of a cylinder (5 heads) containing at least one erroneous segment. For every segment the program distinguishes between statuserror (that is bit 0 : 11 in status 0 unequal to zero) and dataerror (that is the received contents of one or more words of the segment do not equal the expected (output-) contents.

In case of statuserror without dataerror a letter , which identifyes the bad status is printed. If a dataerror is detected a digit corresponding to the letter is printed (which is 0 if no statuserror). Errorfree segments in the printout are identifyed by a - (minus).

The first kind of statuserror detected is identifyed by a (or 1), the next by b (or 2) up to i (or 9). After the segment table follows a list of the bad statuswords corresponding to the letters. For example :

status error	data error	output
no	no	-
no	yes	0
yes	no	a
yes	yes	1

a) status 0 : x00000000000 001100000000

stauts 32 : 100000000000 000000000000

During the read operation the errors detected are stored in the free area. If this area runs full the message :

error space exhausted

is output and the error list printing is initiated. After the list of statuserror the dataerrors will be listed in the format :

run 2, segm. 1, data 00φφφxx0 φ1xxφ1φ1 00xxxφx0 wn.1

For each wrong segment the bad words are listed until the limit as specified in parameter 12 (max. no. of messages on word level per segm.) has been reached. This number may be sat to zero in order to suppress the dataerror printing.

Notice, that the input buffer is cleared before each read operation. This means that an error message like this :

000φφφ0φ 0000φ0φφ φφφ000φ0

(which means that the received data were 0) may be caused by an addressing error in the controller or HCI causing the block loaded to a wrong place inside (or outside) the corestore.

Errormessages, timeout and interrupterrors.

When a time consuming operation, i.e. a data operation or a headmove has been initiated, it is checked that the controller goes busy. If not, the message

dev. not busy after io operation

is output and the test proceeds.

```
run no.        1

run          1,read operation no.        1

run          1,        3 lines of segm. error follow:

                     head no.     0          1          2          3          4
      cyl.        segment no.    0-8       9-17      18-26      27-35      36-44
        20     3600    -3644 aaaaaaaaa aaaaaaaaa aaaaaaaaa aaaaaaaaa aaaaaaaaa

                     head no.     5          6          7          8          9
      cyl.        segment no.    0-8       9-17      18-26      27-35      36-44
        20     3645    -3689 aaaaaaaaa aaaaaaaaa aaaaaaaaa aaaaaaaaa aaaaaaaaa

                     head no.    15         16         17         18         19
      cyl.        segment no.    0-8       9-17      18-26      27-35      36-44
        19     3555    -3599 --------- ------aaa aaaaaaaaa aaaaaaaaa aaaaaaaaa

table of status errors follow:

a) status0:  0x0000000000 001110000000
   status32: 100000000000 000000000000


run          1,read operation no.        2

run          1,read operation no.        3
run          1,segm.    4540,ill. ready head int.

run          1,        4 lines of segm. error follow:

                     head no.     0          1          2          3          4
      cyl.        segment no.    0-8       9-17      18-26      27-35      36-44
        25     4500    -4544 --------- --------- --------- --------- ----00000

                     head no.     5          6          7          8          9
      cyl.        segment no.    0-8       9-17      18-26      27-35      36-44
        25     4545    -4589 00000---- --------- --------- --------- ---------
        29     5265    -5309 --------- --------- -------aa aaaaaaaaa aaaaaaaaa

                     head no.    10         11         12         13         14
      cyl.        segment no.    0-8       9-17      18-26      27-35      36-44
        29     5310    -5354 aaaaaaaaa a-------- --------- --------- ---------

table of status errors follow:

a) status0:  x00000000000 001110000000
   status32: 100000000000 000000000000


run          1,        20 word level error messages follow.


run          1
select testprogram: [illegible]
number of runs = [illegible]
 1,[illegible] and write

[illegible]
[illegible] =   [illegible]
[illegible] =   [illegible]

run no.        1

run          1,read operation no.        1
```

```
run       2,read operation no.       1

run       2,error space exhausted

run       2,      11 lines of segm. error follow:

                head no.     0          1          2          3          4
   cyl.      segment no.    0-8        9-17       18-26      27-35      36-44
     0        0     -44 000000000 000000000 000000000 000000000 000000000
     1      180    -224 111111111 111111111 111111111 111111111 111100000
     2      360    -404 000000000 000000000 000000000 000000000 000000000

                head no.     5          6          7          8          9
   cyl.      segment no.    0-8        9-17       18-26      27-35      36-44
     0       45     -89 000000000 000000000 000000000 000000000 000000000
     1      225    -269 000000000 000000000 000000000 000000000 000000000
     2      405    -449 000000000 000000000 000000000 000000000 000000000

                head no.    10         11         12         13         14
   cyl.      segment no.    0-8        9-17       18-26      27-35      36-44
     0       90     -134 000000000 000000000 000000000 000000000 000000000
     1      270    -314 111111111 111111111 111111111 111000000 000000000
     2      450    -494 000000000 000000000 000000000 000000000 00-------

                head no.    15         16         17         18         19
   cyl.      segment no.    0-8        9-17       18-26      27-35      36-44
     0      135    -179 000000000 000000000 000000000 000000001 111111111
     1      315    -359 000000000 000000000 000000000 000000000 000000000

 table of status errors follow:

 a) status0:  0x0000000000 001110000000
    status32: 100000000000 000000000000


 run       2,     975 word level error messages follow.

 run       2,segm.       0,data 00øøøxø0 ø1xøø1ø1 00xxxøx0  wn.       1
 run       2,segm.       0,data x1øxx1x1 00øøøxø0 ø1xøø1ø1  wn.       2
 run       2,segm.       1,data 00øøøxø0 ø1xøø1ø1 00xxxøx0  wn.       1
 run       2,segm.       1,data x1øxx1x1 00øøøxø0 ø
 select testprogram: b
 number of runs = 9
  1.4 read and write

 mode: run
 fbw =   14376
 lbw =   24614

 run no.       1

 run       1,read operation no.       1

 run       1,      3 lines of segm. error follow:

                head no.     0          1          2          3          4
   cyl.      segment no.    0-8        9-17       18-26      27-35      36-44
    20      3600   -3644 aaaaaaaaa aaaaaaaaa aaaaaaaaa aaaaaaaaa aaaaaaaaa

                head no.     5          6          7          8          9
   cyl.      segment no.    0-8        9-17       18-26      27-35      36-44
    20      3645   -3689 aaaaaaaaa aaaaaaaaa aaaaaaaaa aaaaaaaaa aaaaaaaaa
```

After any sense operations the exception register is examined. This may cause one of the messages :

*** exc. = b.11

*** dev. disconnected

*** max busy (> 5000 mS)

*** max head move (> 500 mS)

The 'max' messages refer to the times in the parenthesises. If this situation appears, the program issues a master clear command and initiate the disc. This may (in case of disconnected) cause the disc to go connected again, annouced by this message :

dev. connected

The interrupt signals are checked for arriving at the correct time. If no interrupt is received when the controller goes ready or head move is finished, on of the messages :

no { head / data } int.

is output.

If an interrupt is received during busy or while heads are moving it will cause this message :

ill. busy { head / data } int.

An unexpected interrupt (e.g. data interrupt after head move or vice versa) is indicated with :

ill. ready { head / data } int.

If the same interrupt is received more times the message :

multi {head / data} int.

is output.

Any of these messages are accompanied by the usual identification of runnumber and segmentnumber .

All errormessages will be output on the outputdevice specified to the loader.

Special information.

When testing certain discdrives, the message :

ill. busy head int.

may be output in the beginning of a read or write operation. This is due to an otherwise harmless designerror in the discdrive.

2.1 Read segments.

Purpose.

This program performs input from one or more segments in an uncritical way, that is without examining the status word or the interrupt signals.

Initiation.

First the program writes :

select disc :

and the operator types the unit number of the disc to be tested. Next the program writes :

first segment = ($0 \leq n \leq 73079$)

and

no. of segments = ($1 \leq n \leq 73080$)

and the operator answers with a number in the range shown in the parenthesis . The program checks the range and automatically cuts the size if too big.

Now the program asks :

blocksize =

and the operator must type a number between 1 and 512. It is checked that the block is inside the free area of the core store.

Method

Before the first run a master clear command and a return to zero command is issued.

Next the segments are read in this way :

(1) the heads are moved if necessary.
(2) the input buffer is cleared.
(3) the block transfer is initiated.

Although the data transferred are not checked, they will be available by the loader's core print feature. Thus, according to (2), if the print shows empty cells, this may be because the block has been transferred to a wrong place or not transferred at all.

Error messages

The only thing checked is the busy and disconnected state. After any operation the device is sensed. This may cause one of these messages to be output :

*** exc = b.11

*** dev. disconnected

*** max. busy (> 5000 mS)

*** max head move (> 500 mS)

The 'max' messages refer to the times in the paranthesis. In case of disconnected the program issues a master clear command and

the run is terminated. In case of busy the program proceeds. If the device happens to go connected again, the message :

dev. connected

will be output.

All the messages mentioned will be accompanied by an identification of run number, and segment or cylinder number.

2.2 Write segments.

Purpose

This program performs writing of specified information on some segments in an uncritical way, that is without examining the status word or the interrupt signals.

Initiation

First the program writes :

select disc :

and the operator types the unit number of the disc to be tested.
Next the program writes :

first segment = ($0 \leq n \leq 73079$)

and

no. of segments = ($1 \leq n \leq 73080$)

and the operator answers with a number in the range shown in the parenthesis. The program checks the range and automatically cuts down the size if necessary.

Now the programs asks :

blocksize =

and the operator must type a number between 1 and 512. It is checked that the block lies inside the free area of the core store.

Finally the program asks :

contents =

and the operator types (either in decimal or with radix as described in 1.3 above) from 1 up to maximum of 16 integers separated by > comma > and terminated by < NL >. These integers are filled into the output buffer as shown below (in this example the integers are : -1, 0, 2) :

word no.	output buffer
0	-1
1	0
2	2
3	-1
4	0
.	.
.	.
.	.
254	2
255	-1
0	0
1	2
.	.
.	.
.	.
255	0

If the operator types wrong,the current line may be regretted ty typing ' å '.

Method and error messages

are similar to those of program 2.1.except that writing instead of reading is performed.

2.3 Read segment addresses

Purpose

This program performs input from one or more segment addresses in an uncritical way, that is without examining the statusword or the interrupt signals.

Initiation

First the program writes :

select disc :

and the operator types the number of the disc to be tested. Next the program asks :

first segment = ($0 \leq n \leq 73079$)

and

no. of segment = ($1 \leq n \leq 73080$)

and the operator will answer by typing a number inside the range shown in the parenthesis. The numbers will be checked for being integer multipla of nine. If not, they are automatically cut, and the new value is given on the typewriter.
The blocksize is always set to 9 segments .

Method.

Before the first run a master clear command and a return to zero command is issued. Next the segments are read in this way :

(1.) the heads are moved if necessary.
(2.) the input buffer is cleared.
(3.) the block transfer is started.

Although the address marks transferred are not checked, they will be available by means of the loader's core print feature. Thus , according to (2), if the print shows empty cells, this may be because the block has been transferred to a wrong place or not transferred at all.

Error messages.

The only thing checked is the busy and disconnected state. After any operation the device is sensed. This may cause one of these messages to be output :

*** ecx. = b.11

*** dev. disconnected

*** max. busy (> 5000 mS)

*** max. head move (> 500 mS)

The ' max ' messages refer to the time in the parenthesis.
In case of disconnected the program issues a master clear command and the run is terminated. In case of busy the program proceeds.
If the device happens to go connected again, the message :

dev.connected

will be output.
All the messages mentioned will be accompanied by an identification of runnumber and segment- or cylinder number.

2.4 Write segment addresses

Purpose

This program writes specified information on 9 segments under same head. The writing is performed in an uncritical way that is, without examining the status words and the interrupt signals.

Initiation

First the program writes :

select disc :

and the operator types the unit of the disc to be tested.
Next the program writes :

first segment = ($0 \leq n \leq 73079$)

and

number of segments = ($1 \leq n \leq 73080$)

and the operator will answer by typing a number inside the range shown in the parenthesis. The numbers will be checked for being integer multiple of nine. If not, they are automatically cut, and the new value is given on the typewriter.

The blocksize is always set to 9 segments.
Finally the program asks :

contents =

and the operator types (either in decimal or with radix as described in 1.3 above) from 1 up to maximum of 8 integers separated by < comma > and terminated by < NL >.
These integers are filled into the output buffer as shown below. (In this example the integers are -1, 0, 2) :

word no.	output buffer
0	-1
1	0
2	2
3	-1
4	0
5	2
6	-1
7	0
8	2

If the operator types a wrong number, the current line may be regretted by typing ' å '.

Method and errormessages

are similar to those of program 2.3 except that writing instead of reading is performed.

2.5 Move heads

Purpose

This program perform a sequence of head movings to specified cylinders.

Initiation

The program asks the following questions, and the operator must answer as indicated to the right :

select disc =	($0 \leq$ unit ≤ 7)
cyl. sequence =	(up to 16 decimals or radix number separated by commas and terminated by < NL >).

Errormessages

After each move operation the statusword is examined to see if bit 10 (= synchronization error) is equal to zero. If not, the message :

*** sync. error

will be output. As neither read nor write has been attempted, sync. error in this case is identical to the signal ' seek error ' from the disc drive.

Further , the exception register is examined, and if disconnected, the message :

*** dev. disconnected

will be output. This will cause the program to issue a master clear and the run will be terminated. If the device goes connected again the message :

dev. connected

will appear. Hard errors during head move may give rise to one of these messages :

*** exc. = b.11

*** max. busy (> 5000 mS)

*** max. head move (> 500 mS)

Notice, that the controller must not go busy during head move operations.

All error messages are accompanied by an identification of run number and cylinder number.