Title:

Cache memory simulator RC8000/55.
RC8000 instruction counter.

A/S REGNECENTRALEN

RC SYSTEM LIBRARY: FALKONERALLE 1 DK-2000 COPENHAGEN F

RCSL No: 31-D524

Edition: November 1978

Author: Bodil Larsen

Keywords:

RC8000 cache memory simulation, Escape function.

Abstract:

Simulation of a cache memory on RC8000/55.
Examine hitrate for running programs with different cache and block sizes.

The escape routine in RC8000 is used to perform the simulation.

A list of used instructions in the supervised program is created by the simulation.

24 pages.

CONTENT PAGE

INTRODUCTION.

This report describes the method and results simulating a cache memory on the RC8000 computer.

The study was made by Bo Tveden Jørgensen, Rune Einersen and Bodil Larsen in August 1978.

References.

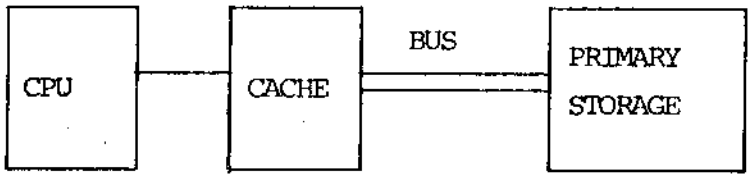
Ref. 1. RC8000 Ref. manual RCSL 31-D383.

1. Cache Memory.

A cache memory is a very fast storage which is set between the primary storage and the CPU. The cache memory is directly connected to the CPU so data is fetched without activating the bus.

The cache memory is used as a buffer between the CPU and the primary storage so when a word is fetched from the primary storage it is stored in the cache.

This is done with the hope that many of the fetched data should be used again. The percentage of reused data is called the hitrate.



Time reduction using cache.

The time for fetching data from the primary storage 900 ns.
Time for fetching data from the cache 200 ns.

Lets consider 100 fetches from the primary store:

Without cache 100 x 900 = 90000 ns.
With cache and 75% hitrate 25x900 + 75x200 = 37500 ns.

As seen, a good hitrate may give a high decrease in fetch time, at the same time the load on the bus is decreased.

This calculation is not fully correct, as a store will go to the primary storage even if it is present in the cache memory.

1.1. Set associative cache.

The set associative cache method is chosen.

The set associative method make a many to one mapping of the primary storage on the cache.

Say primary storage size = m = k x n
cache storage size = n

Primary storage can now be split in the following way:

The elements 0, n, 2n, ..., m-n are all stored in the same cache element if fetched. Similar for the other elements.

This is a cache with single element sets, but a cache may contain sets with several elements.

If the cache contains sets with more elements a primary storage element can be stored in one of the elements in the corresponding set of the cache. A strategy for storing in the different elements of the sets could be either cyclic, random or the longest not used element.

In hardware all elements of a set can be scanned in parallel which make the search fast.

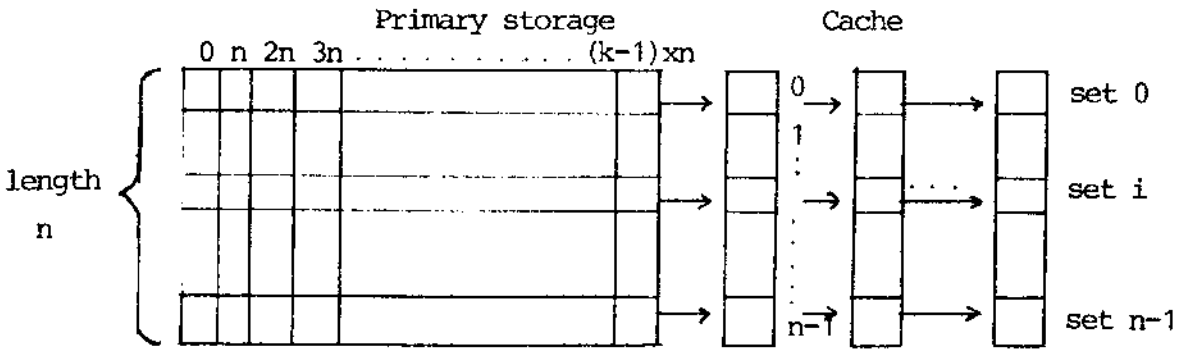


Fig. 1 Set associative cache with set size p

The described method is used for our simulator, but there are other cache strategies e.g. simple cyclical storing.

1.2 Blocks in cache.

Instead of moving one word to the cache at a time a whole block can be transferred so that the sets consists of a number of blocks.

As the program execution is basically sequential, the next instruction will probably be found in the cache. This is not necessarily true in case of data references.

As a whole block should be transferred when one word is referenced, this should be done in one 'operation' on the bus.

In RC8000 it is only possible to monopolize the bus for the time used to transfer two words from primary storage to the CPU. A longer time period may result in dataoverrun on the disc.

Therefore the blocksize was chosen to 2 words.

1.3 Cache strategy.

The chosen strategy is as follows:

A set associative cache is used.

Total cache size of 1 k and 4 k words is simulated.

Block size of 2 and 4 words is simulated.

Number of elements in a set is 1 block.

When a store is made it is stored through to the primary storage; if the address is in the cache, the new data is stored here too. This feature is also called write-through.

The cache listen to the bus. So every transfer to an address in the primary storage will result in the same transfer to the corresponding address in the cache memory. This feature ensures a correct content of the cache even if corresponding words in the primary storage is changed by input from a peripheral device.

2. Simulation.

The cache strategy described in 1.3 is simulated using the escape facility of the RC8000 to survey the program execution. (cf ref. 1)

Three utility programs are used to set or clear the escape function.

A printout program is created to output the collected tables in a readable form.

2.1 Escape routine.

RC8000 escape facility is used to supervise the program execution in order to simulate the cache.

The escape facility is implemented by means of an escape mode, an escape mask and an escape address.

The escape mode tells the monitor if an escape should be performed for this process. The escape mask tells which instructions should be supervised and the escape address tells which routine should be executed when the escape is performed.

The instructions inside the escape routine are executed with escape mode = no. A return to the program is made by the instruction, return escape (RE) which sets the escape mode back to yes.

2.1.1 Function of the escape routine.

The escape routine used for the simulator performs the following:

1. Updates a table, with the first 12 bits of the instruction as, index, (instruction code, working reg, indirect, relative and index register).

   An occurrence of a 12 bit code is counted in this table.

2. Count no. of instructions, loads, stores and indirects.

3. Count instruction misses (instructions not found in cache) load misses and store misses.

2.1.2 Changeable simulation variables.

The escape routine is made as a stand-alone routine incorporated in the utility program translation (cf 2.2)

There is a number of changeable variables in the routine, a recompilation is necessary if a change is wanted.

Variable name	Std. value	Description
F0	1	own cache 1=yes, -1=no If the cache tables are in a separate process use no.
F1	1	Instruction count 1=yes, -1=no.
F2	1	Indirect count 1=yes, -1=no.
F3	1	Load count 1=yes, -1=no.
F4	1	Store count 1=yes, -1=no.
F5	1	Clear cache on jd instruction 1=yes, -1=no. A clear on jd is a simulation of a process change.
F6	9	No. of bits in row index. No. of bits in address part of cache block.
F7	1	No. of block index bits Note F6+F7=number of bits in cache defining standard cachesize to 10 bits = 1024 = 1K

F8	11	No. of bits in standard cache table size
F9	1	Count of cache element destroy by instruc. 1=yes, -1=no.
F10	1	Count of cache element destroy by loads. 1=yes, -1=no.

2.2 Utility programs.

Three utility programs are created to control the escape function.

- setescape
- stopescape
- clearescape

2.2.1 Setescape.

Call:

setescape

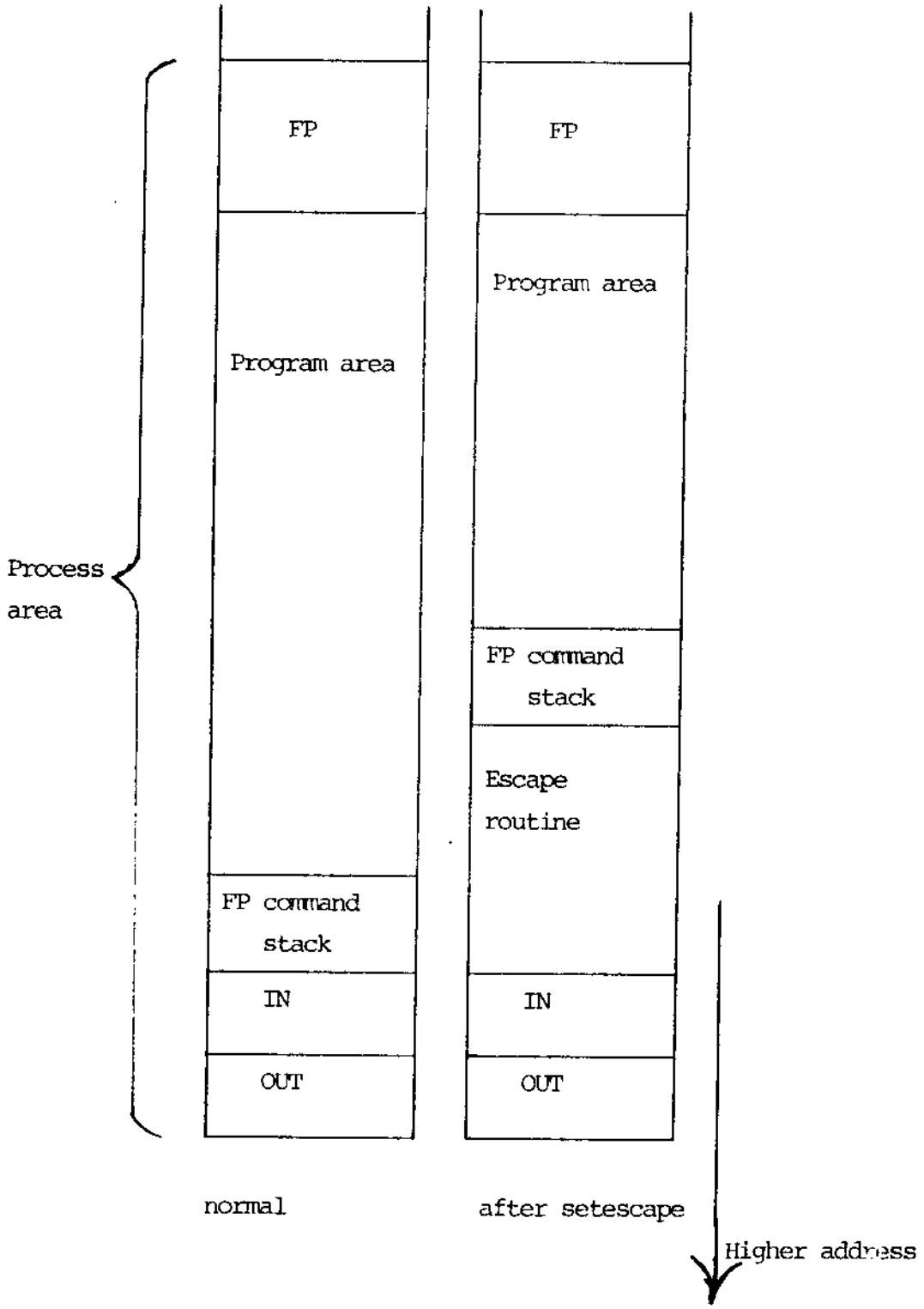
Function:

Moves the program stack to lower addresses to make space for the escape routine, copy the routine to the created space (cf. fig. 2).

Call the monitor procedure set escape to set escape address, escape mode and escape mask.

If a new call of setescape is made with an escape routine of different size, then the new escape routine is stored, else nothing is changed in the stack. In both cases a new call of the monitor procedure set escape is performed.

Fig. 2. Escape routine in FP stack.



2.2.2 Stopescape.

Call:

stopescape

Function:

Clears the escape mask and the escape address.

So the escape function is no longer performed but the escape routine is still in the FP stack.

2.2.3 Clearescape.

call:

clearescape.

Function:

Clears the escape mask and the escape address.

Moves the program stack pointers back (cf 2.2.1 setescape).

2.2.4 Monitor procedure set escape.

Procedure Set Escape, 1

set escape (escape-address, escape-specification)
w0 escape-specification (call)
w1
w2
w3 escape-address (call)

jd 1<11+1
Defines the escape address, escape mode and escape mask of the calling process.

Sets the escape-specification = escape-mode and escape-mask in the exception register (status register).

Escape-specification:

escape-mode <22+ escape-mask <12

escape-mode 1 bit (no. 1) { 0 no escape function / 1 escape function

escape-mask 6 bits (no. 6-11) (cf ref.1.)

The escape address must either be zero or point to an area within the calling process. If zero no escape function is performed.

Setting escape-specification or escape address to zero the escape function is inactive.

PARAMETER ERROR: escape address outside calling process.

2.3 Printout program escprint.

This utility program helps print out the tables created by the simulation.

The program prints the following:

- list of the used changeable variables (cf. 2.1.2)
- table of used instructions printed in number, percent and split into indirect, relative and indexed.
- tabel over x-addressings, split into pure, indirect and relative.
- number of instructions loads, stores and indirects.
- number of instructions, load and store misses.
- number of times an instruction or load destroys something in the cache.
- in percent how many words are averagely used in the cache.
- the total hitrate and the instruction, the load and the store hitrate are printed.

Note the store hitrate is not interesting as we always store through to the primary storage.

See example in chapter 3.

2.4 Systime.

Note if the cache is large (e.g. 4K) and the jd clear facility is used, 'systime' should be leftout from the supervised programs as it may cause an endless loop in the simulation.

3. Example.

A program 'benchprog' is supervised.

3.1 FP program.

The Fp calls to run the supervision could be as follows:

```
head cpu
(setescape          ; set escape routine
benchprog           ; run program
stopescape)         ; clear escape mode

head cpu
escprint            ; print cache tables
clearescape         ; clear escape routine
```

To run this program you need furthermore a file descriptor for the sort file and some input parameters.

The actual FP program looks as follows:

```
HEAD 1 CPU
USORTFILE=SET [illegible] 20.3
(SETESCAPE
BENCHPROG
STOPESCAPE)
[illegible]
EX.1: [illegible] TIL [illegible] AF POSTER,
      [illegible] TEXTEN FORSKYDES / POSITIONER FRA POST TIL POST   *)
HEAD CPU
ESCPRINT

CLEARESCAPE
```

*) Note! This shall actually be one line

3.2 Program listing.

```
BENCHPROG=ALGOL
BENCHMARK PROGRAM FOR GENERATION AND PRINT OF RECORDS.
JW 0.27.01.1977
BEGIN
COMMENT       PARAMETERS ARE READ FROM ZONE IN:
         1. LINE :  <RECORDLENGTH> <NUMBER OF RECORDS> <FUNCTION> <FILE>
         2. LINE :  FUNCTION = 0, GENERATION OF RECORDS:
                    THE 2. LINE CONTAINS A TEXT TO BE USED FOR THE GENE-
                    RATION OF RECORDS. THE TEXT IS USED TO FILL UP 2 *
                    RECORDLENGTH CHARACTERS WHICH ARE USED CYCLICALLY
                    WITH A DISPLACEMENT OF 7 FROM RECORD TO RECORD.
                    FUNCTION > 0, PRINT OF EVERY FUNCTION RECORD:
                    THE LINE CONTAINS MARKER POSITIONS, THE LAST ONE < 0.
;

  INTEGER  FUNCTION, INT, RECLENGTH, RECORDS;
  LONG ARRAY  FILENAME(1:2);

  READ(IN, RECLENGTH, RECORDS, FUNCTION);
  READSTRING(IN, FILENAME, 1);
  REPEATCHAR(IN);

  WRITE(OUT, <:<10>RECLENGTH: :>, RECLENGTH,
             <:     RECORDS: :>, RECORDS,
             <:     FUNCTION: :>, FUNCTION,
             <:     FILENAME: :>, FILENAME,
             <:<10><10>:>);
  INT:= 1;
  BEGIN
ZONE  Z(128*2,2,STDERROR);

    INTEGER  BASE, CLASS, I, LAST_CHAR, NEXT_POS, POS_INX;
    INTEGER ARRAY  POSITION(1:20);
    BOOLEAN ARRAY  TEXT(1:2*RECLENGTH);
    BOOLEAN FIELD  CHAR;

    <* SKIP THE REST OF CURRENT LINE *>
    FOR CLASS:= READCHAR(IN, CHAR) WHILE CLASS < 8 DO;
    IF CHAR <> 25 THEN  CLASS:= 0;
```

```
IF FUNCTION = C THEN
BEGIN
COMMENT     CREATE RECORDS;
  I:= 1;
  OPEN(Z,4, STRING FILENAME(INCREASE(I)), 0);

  LAST_CHAR:= 1; TEXT(1):= FALSE AND 97; <* A *>
  FOR I:= 1 STEP 1 UNTIL 2*RECLENGTH DO
  BEGIN
    IF CLASS < 8 THEN  CLASS:= READCHAR(IN, CHAR);

    IF CLASS < 8 THEN
    BEGIN
      TEXT(I):= FALSE AND CHAR;
      LAST_CHAR:= I;
    END
    ELSE
      TEXT(I):= TEXT((I - 1) MOD LAST_CHAR + 1);
  END  READ THE TEXT FOR THE RECORD GENERATION;

  BASE:= 0;
  FOR I:= 1 STEP 1 UNTIL RECORDS DO
  BEGIN
    OUTREC4(Z, RECLENGTH);
    FOR CHAR:= 1 STEP 1 UNTIL RECLENGTH DO
      Z.CHAR:= TEXT(BASE + CHAR);

    BASE:= BASE + 7; <* USE THE ORIGINAL TEXT CYCLICALLY *>
    IF BASE > RECLENGTH THEN  BASE:= BASE - RECLENGTH;
  END  OUTREC4;

  CLOSE(Z, TRUE);
END  FUNCTION = C, CREATE RECORDS
ELSE
```

```
      BEGIN
      COMMENT     PRINT EVERY FUNCTION RECORD;

        I:= 1;
        OPEN(Z,4, STRING FILENAME(INCREASE(I)), 0);

        <* READ IN MARKING POSITIONS *>
        FOR POS_INX:= 1, POS_INX + 1 WHILE  POSITION(POS_INX - 1) > 0 DO
          READ(IN, POSITION(POS_INX));

        FOR I:= 1 STEP 1 UNTIL RECORDS DO
        BEGIN
          INREC4(Z, RECLENGTH);

          IF (I-1) MOD FUNCTION = 0 THEN
          BEGIN
          COMMENT     PRINT THE RECORD;
            POS_INX:= 1;
            NEXT_POS:= POSITION(POS_INX);

            FOR CHAR:= 1 STEP 1 UNTIL RECLENGTH DO
            BEGIN
              IF CHAR = NEXT_POS THEN
              BEGIN
                WRITE(OUT, <:!:>);
                POS_INX:= POS_INX + 1;
                NEXT_POS:= POSITION(POS_INX);
              END  MARKER POSITION;

              OUTCHAR(OUT, Z_CHAR EXTRACT 12);
            END  FOR CHAR;
            IF CHAR = NEXT_POS THEN  WRITE(OUT, <:!:>);
            OUTCHAR(OUT, 10);
          END  PRINT THE RECORD;
        END  INREC4;
CLOSE(Z,TRUE);
    END  FUNCTION > 0, PRINT;

  END  INNER BLOCK;
END
```

3.3 Escprint output.

Here follows the output from a cache simulation run.

```
*HEAD 1 CPU
BOL  1976.11.16  17.15.10    CPU: 15.01 SEC.
*USORTFIL=SET 400 DISC 5 0 0 20.3
*SETESCAPE
*BENCHPROG

RECLENGTH:  80     RECORDS:  500     FUNCTION:  0     FILENAME: USORTFIL


END     19
*STOPESCAPE
*HEAD CPU
 BOL  1976.11.16  17.17.45    CPU: 165.20 SEC.
*ESCPRINT

ESCPRINT

CACHE:  4096 WORDS
 ROW INDEX LENGTH:  11
 BLOCK INDEX LENGTH (WORDS):  1
OWN CACHE:       YES
INSTR. COUNTS:   YES
INDIR. COUNTS:   YES
LOAD COUNTS:     YES
STORE COUNTS:    YES
JD-CLEAR:        YES
CACHE TABLE:      11
INSTR DESTROY    YES
LOAD DESTROY     YES
```

} Changeable variables

INSTRUKTION		PCT	X0	X1	X2	X3	IX0	IX1	IX2	IX3	RX0	RX1	RX2	RX3	RIX0	RIX1	RIX2	RIX3
EL	156	0	121	3	9	8	0	0	0	0	0	2	1	12	0	0	0	0
HL	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LA	842	0	24	7	88	0	0	0	0	0	723	0	0	0	0	0	0	0
LO	231	0	108	0	6	0	0	0	0	0	117	0	0	0	0	0	0	0
LX	26	0	0	0	0	0	0	0	0	0	26	0	0	0	0	0	0	0
WA	163 924	18	97	7	161412	507	0	0	10	1	1878	0	0	0	12	0	0	0
WS	43 894	5	90	88	752	42129	0	0	0	0	826	0	0	0	9	0	0	0
AM	1 093	0	750	2	0	0	0	8	121	1	0	0	0	0	211	0	0	0
WM	176	0	2	0	3	0	0	0	0	0	171	0	0	0	0	0	0	0
AL	53 370	6	45273	3236	2032	1682	0	0	0	0	1033	4	0	0	110	0	0	0
JL	54 381	6	114	0	8	2517	0	0	0	0	48418	87	20	137	3080	0	0	0
JD	318	0	318	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
XL	23	0	0	0	0	0	0	0	0	0	23	0	0	0	0	0	0	0
ES	95	0	0	0	6	0	0	0	0	0	89	0	0	0	0	0	0	0
EA	970	0	1	0	818	20	0	0	0	0	131	0	0	0	0	0	0	0
ZL	41 595	5	112	40163	122	177	6	0	0	0	206	797	6	6	0	0	0	0
RL	141 881	16	708	1158	128232	3556	12	0	267	0	5634	0	0	0	2314	0	0	0
RS	93 506	10	7	2811	84334	1289	0	2	388	0	4562	0	0	0	113	0	0	0
WD	67	0	1	0	56	0	0	0	0	0	10	0	0	0	0	0	0	0
RX	879	0	11	167	123	500	0	0	8	0	64	0	0	0	6	0	0	0
HS	41 839	5	0	86	11	5	0	0	40161	0	1570	6	0	0	0	0	0	0
XS	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AC	741	0	0	4	0	718	2	0	0	0	17	0	0	0	0	0	0	0
NS	4	0	3	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
AS	40 223	4	40223	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AD	12	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LS	738	0	625	0	0	0	113	0	0	0	0	0	0	0	0	0	0	0
LD	388	0	388	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SH	165 553	18	41511	6	0	738	0	101	120527	1419	513	0	0	0	738	0	0	0
SL	43 240	5	42134	19	6	218	67	22	543	1	60	0	0	0	170	0	0	0
SE	2 399	0	873	0	8	0	300	2	0	1125	47	0	0	0	44	0	0	0
SN	1 752	0	1537	91	0	6	0	1	2	2	0	0	0	0	113	0	0	0
SO	645	0	645	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SZ	3 301	0	2628	0	0	0	0	0	0	300	0	0	0	0	191	170	12	0
FM	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
DL	2 715	0	0	609	1955	23	0	0	6	0	116	0	0	0	6	0	0	0
DS	6 336	1	0	2501	311	6	0	0	0	0	2185	0	0	0	1333	0	0	0
AA	12	0	0	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0

SUM AF INSTRUKTIONER: 907 328
HERAF INDIREKTE: 174 151, 19 PCT
HERAF RELATIVE: 78 130, 9 PCT

	REL	INDIREKTE	RELATIVE	REL OG IND	
X0-ADRESSERING:	178 517	500	68 420	8 450,	28 PCT
X1-ADRESSERING:	50 959	136	696	170,	6 PCT
X2-ADRESSERING:	380 304	162 034	27	12,	60 PCT
X3-ADRESSERING:	54 999	2 849	155	0,	6 PCT
PROGRAMTER:	663 679	165 519	69 498	8 632	
	75	18	8	1	

LOADOPERANDS: 574 567, LOADFACTOR: 65
STOREOPERANDS: 146 901, STOREFACTOR: 16

INST MISS	16 751
LOAD MISS	13 599
STORE MISS	40 224
INSTRUCTIONS	907 323
INDIRECTS	174 152
OPERANDS/LOAD	574 308
OPERANDS/STORE	146 911
INSTR DESTROYS	2 394
LOAD DESTROYS	4 305
EMPTY CACHE	665 522
WORDS PR JOB	1967
USED IN CACHE	4 PCT
HITRATE	96 PCT
INST HITRATE	98 PCT
LOAD HITRATE	98 PCT
STORE HITRATE	66 PCT

END 31
*CLEARESCAPE

4. Results.

4.1 Supervised programs.

As you get a factor 40 on running programs with this simulation, we have only chosen rather small programs to supervise.

We have run the programs with all combinations of the following:

Cache size: 1 K and 4 K.
Block size: 2 words and 4 words
With or without clearing the cache in case of jd instruction (simulation of process change).

The runs:

Algol translation of
BENCHMARK program (the program in chapter 3)
ESCPRINT
Run of programs:
MAXECON run with logfile input
BENCHMARK program (500 records)
ESCPRINT program
SORTBS (500 records)

The result of these runs are shown on fig.3.

The found total hitrate is within in the tabel

Fig 3 Hitrate for simulated cache runs.

program run:

jd-clear / no jd-clear	Block size: 2 words		Block size: 4 words	
	4K	1K	4K	1K
Algol transl. BENCHMARK	96 / 99	94 / 96	97 / 99	96 / 97
Algol transl. ESCPRINT	97 / 100	95 / 97	98 / 100	97 / 98
MAXECON	87 / 94	83 / 85	92 / 96	89 / 91
BENCHMARK 500 records	98 / 100	97 / 98	99 / 100	98 / 99
ESCPRINT	96 / 97	92 / 92	98 / 98	95 / 95
SORT BS 500 records	91 / 94	84 / 86	94 / 96	88 / 90

5. Conclusion.

The supervised runs are not enough to give a full picture of how a cache memory would work.

The following runs should be supervised too:

mathematical-statistical programs
BOSS
Monitor
total system with common cache (cf 2.1.2 F0 variable)

All in all we must say that the given material only is an indicator for how various programs will use a cache memory.

Further possibilities:

. For every instruction type the following instruction is registrated.
A very used sequence could introduce a new instruction.

. Special cache version where only X2/X3 - addressings are stored in the cache (ALGOL/FORTRAN machine).

. It might be interesting to see how much the monitor actually intervenes with the cache parts used by the processes. Another strategy might be to have two cache memories: one for the monitor and one for the unprivileged processes. A further extension could be to have a number of independent cache memories and determin at the time of process creation which cache memory should be used for instance by applying the reminiscent pk-value.

Future:

The made measurements are only tentative real measurements must be carried out on the RC8000/55.

Therefore the RC8000/55 must be equipped with possibility of measuring par example:

. hits and misses in 48 bit counters.

. time lost waiting for transfer of the rest of the block.

If this is done, it is possible to get knowledge of how to construct the best cache memory for the RC8000 system.

Bo Tveden Jørgensen - Rune Einersen - Bodil Larsen.