

```

title[memA];
top level;
* December 19, 1977 8:11 AM
%
```

ABOUT SOURCE FILE MAINTENANCE

The sources live on <dlsource>memAllSource.dm and <dlsource>diagnosticSubrs.dm. By loading those files the user acquires all the files necessary to assemble a new version of memAll:

```

memA.cm // assemble and place memAll.mc
memA.mc // control file
```

see "insert" commands below for the subroutine files used by this program

```

memA.midas // set up midas context
memAtoIfs.cm // make new:
[ifs]<dlsource>memASource.dm,<d1>memAll.dm
memA.files // list of memAll files put in memASource.dm
preamble.mc // preamble code that that precedes memAll
postamble.mc // postamble code that that follows memAll
kernelalu.mc // defines alu ops for this diagnostic
```

To make a new version of memA:

```

ftp maxc load <dlsource>memASource.dm, load <dlsource>diagnosticSubrs.dm
<make your changes in memA.mc>
@memA.cm // assemble and place the microcode
@memATOIFS.cm // move the sources, .mb file to maxc
```

%

%

```

February 1, 1980 8:52 PM
add restartDiagnostic label and code.
```

```

June 25, 1979 2:10 PM
resetIfu inside initMemTest.
```

%

afterSubrs:

restartDiagnostic:

```

t_1c;
stkp_t;
```

begin:

```

call[initMemTest];
* now tpc[17] _ 7777C. midas puts a breakpoint in this address. anywakeups
* are probably errors -- unless specifically managed by the code
* call[aHaltTask17];
```

```

call[disableConditionalTask];
call[beginCtest];
```

afterCtest:

```

call[initMemTest];
call[beginXtest];
```

afterXtest:

```

call[initMemTest];
call[beginDtest];
```

afterDtest:

```

call[initMemTest];
call[beginStest];
```

afterStest:

%

```

call[initMemTest];
insert[memPipeAndFaultA]
call[initMemTest];
goto[beginFIOTest];
```

afterFIOTest:

%

afterTest:

```

call[initMemTest];
branch[done];
```

initMemTest: subroutine;

```

RBASE _ rbase[defaultRegion];
rscr _link;
top level;
t _ r0 _ t-t, ifuReset;
```

```
rml _ cml;  
r1 _ t + 1;  
rhigh1 _ 100000C;  
r01 _ not(r10);  
t _ (r0)+1;  
stkp _ t;  
ProcSRN_r0;  
returnUsing[rscr];
```

* November 29, 1977 10:31 AM

* CODE for midas debugging

top level;

set[dbgTbls,100];

11: branch[11],

at[dbgTbls,0];

12: noop,

at[dbgTbls,1];

branch[12],

at[dbgTbls,2];

13: noop,

at[dbgTbls,3];

noop,

at[dbgTbls,4];

branch[13],

at[dbgTbls,5];

14: noop,

at[dbgTbls,6];

noop,

at[dbgTbls,7];

noop,

at[dbgTbls,10];

branch[14],

at[dbgTbls,11];

15: noop,

at[dbgTbls,12];

noop,

at[dbgTbls,13];

noop,

at[dbgTbls,14];

noop,

at[dbgTbls,15];

branch[15],

at[dbgTbls,16];

16: noop,

at[dbgTbls,17];

noop,

at[dbgTbls,20];

noop,

at[dbgTbls,21];

noop,

at[dbgTbls,22];

noop,

at[dbgTbls,23];

branch[16],

at[dbgTbls,24];

branch[begin];

END;

```

title[MemMisc];
top level;
* May 29, 1981 10:44 AM
%
```

ABOUT SOURCE FILE MAINTENANCE

The sources live on <doradoSource>memMiscSource.dm and <doradoSource>diagnosticSubrs.dm. By loading those files the user acquires all the files necessary to assemble a new version of memAll:

```

memMisc.cm           // assemble and place memAll.mc
memMisc.mc           // control file
```

see "insert" commands below for the subroutine files used by this program

```

memMisc.midas        // set up midas context
memMiscToIfs.cm      // make new: [ivy]<doradoSource>memASource.dm,<ddorado>memAll.dm
memA.files            // list of memAll files put in memASource.dm
preamble.mc          // preamble code that that precedes memAll
postamble.mc         // postamble code that that follows memAll
kernelalu.mc         // defines alu ops for this diagnostic
memAtoMaxc.cm        // temporary expedient only for use when IFS down
```

To make a new version of memA:

```

ftp maxc load <dlsource>memASource.dm, load <dlsource>diagnosticSubrs.dm
<make your changes in memA.mc>
@memMisc.cm           // assemble and place the microcode
@memAtoIVY.cm         // move the sources, .mb file to maxc
```

June 23, 1981 9:20 AM

```
Call aProcShifter
```

June 17, 1981 8:55 AM

```
Add call to aMapTest
```

May 29, 1981 10:44 AM

```
Add restartDiagnostic label.
```

%

afterSubrs:

restartDiagnostic:

```
t_1c;
StkP_ t;
```

begin:

```
call[initMemTest];
call[aProcShifter];
call[aMapTest];
call[initMemTest];
call[disableConditionalTask];
goto[fiotest];
```

afterFiotest:

```
call[initMemTest];
call[iMemState];           * fioTest may clobber sTestFlags register
call[aPipeTestCtrl];
```

afterTest:

```
call[initMemTest];
branch[done];
```

initMemTest: subroutine;

```
RBASE _ rbase[defaultRegion];
rscr _link;
top level;
t _ r0 _ t-t;
rml _ cml;
r1 _ t + 1;
rhigh1 _ 100000C;
r01 _ not(r10);
t _ (r0)+1;
stkp _ t;
ProcSRN_r0;
returnUsing[rscr];
```

* November 29, 1977 10:31 AM

* CODE for midas debugging

top level;

set[dbgTbls,100];

11: branch[11],

at[dbgTbls,0];

12: noop,

at[dbgTbls,1];

branch[12],

at[dbgTbls,2];

13: noop,

at[dbgTbls,3];

noop,

at[dbgTbls,4];

branch[13],

at[dbgTbls,5];

14: noop,

at[dbgTbls,6];

noop,

at[dbgTbls,7];

noop,

at[dbgTbls,10];

branch[14],

at[dbgTbls,11];

15: noop,

at[dbgTbls,12];

noop,

at[dbgTbls,13];

noop,

at[dbgTbls,14];

noop,

at[dbgTbls,15];

branch[15],

at[dbgTbls,16];

16: noop,

at[dbgTbls,17];

noop,

at[dbgTbls,20];

noop,

at[dbgTbls,21];

noop,

at[dbgTbls,22];

noop,

at[dbgTbls,23];

branch[16],

at[dbgTbls,24];

branch[begin];

END;

```
title[memAfio];
top level;
```

```
beginFIOTest:
```

```
* September 19, 1978 4:17 PM
```

```
*****
```

ROUTINE	DESCRIPTION
fioTest	Entry point for fio test
fioTestCode	Code that runs in test task
fioStest	Code that tests storage
fioTesterReset	Reset the fio jig, clobbers rmx7
setFIOTester	Set fio jig for t = task num, rscr = subtask
fioSetFIOTester	Set fio jig for current subtask, task
iaSubTask	Initialize subtask variable
aNextSubTask	Return the next subtask value
aGetSubTask	Return current subtask value
aGetSTva	Returns effective address for a subtask
fioIMem	Initialize Storage as required for fio test
fioStorage	Write identity into storage (mem[va] _ va)

```
*****
```

```
*****
```

```
May 16, 1979 1:49 PM
```

Another incompatibility: since rstk is wider, the old assumptions about which bits are unchanged by subtasking are incorrect.

```
May 16, 1979 11:51 AM
```

Fix more model0/modell incompatibility problems: numerous mem subroutines save their return on the stack & must not be called from non-emulator tasks.

```
May 15, 1979 4:03 PM
```

Fix model0/modell incompatibility: stack no longer shared w/ RM means that the initialization code can't write into RM via stkp manipulations.

```
May 4, 1979 2:27 PM
```

Construct modell version of code from model0 version.

```
*****
```

```
*****
```

here's how it works:

TIOA 300 stores what the device believes is its task number and its subtask number.

Top 4 bits task, next 2 subtask, next two magic (see below)

When an IOfetch or IOstore is done by the task and subtask stored in TIOA 300,

The device does the right thing BUT ONLY TWICE, i.e., it is two munches big.

The microcode should first set TIOA 300 to the appropriate task and subtask number,

then do (in that task and subtask) TWO IOfetches, then two IOstores, and see if

what it got back is the same as what it sent. All subtasks should be verified.

Bits 06 and 07 of TIOA 300 should be ZERO for now. They are wire-ored

to the parity bits of the FIN bus to inject ST parity errors.

Reminders about the Testing

The emulator can have subtasks, too! First, for every task, make sure

that membase and rbase get the appropriate bits or'd into them by

the current subtask:

(things to do

1) choose addresses other than 0..37B to make sure the subtask bits arrive early enough for base register arithmetic

2) check all the bits on the fast io bus

3) don't forget to check rstk bits

4) worry about what portion of the test to run during task switching and holds. verify w/ dough that taskswitching and holds will be ok.

5) remember, for subtasks, RBase[2:3] _ RBase[2:3] OR Subtask[0:1].

6) subtask[0:1] are or'd w/ memBase[2:3] during memory references.

7) don't forget to exit with subtask = 0

8) Initialize the Base registers such that when the membase is zero and the subtask bits get or'd into

membase, the resulting base register used by the processor will have a unique value in it that enables the test to make sure the subtask bits really got or'd in. In particular, we initialize as follows: BR[i]_ i*1000B. This results in a situation where "vm 0" for subtask i should REALLY be vm (i * 10000B). This fact relates to which bits get or'd into memBase.

* June 5, 1981 4:16 PM

```

FioControl: TYPE = MACHINE DEPENDENT RECORD[
  Task: TaskN,
  subTask: SubTaskN
];
TaskN: CARDINAL[0..17];
subTaskN: CARDINAL[0..3];

FOR task IN [0..17] DO
  FOR subTask IN subTaskN DO
    iMem[];
    NOTIFY[task, @fioTestCode]; --awaken task to xqt the test
    ENDLLOOP; -- subTask loop
  ENDLLOOP; -- Task loop;
ioReset[];
OUTPUT _ fioTester;
END;
fioTestCode: PROCEDURE =
BEGIN
  FIOtestRbase[Task, subTask];
  FOR i IN Va DO mem[i] _ i;
  clearCacheFlags[];
  setFIOtester[task, subTask];
  IOFetch _ 0;
  IOFetch _ 20B;
  IOStore _ 0;
  IOStore _ 20B;
  ioReset[]; -- could substitute setFIOtester[0,0];

  FOR i IN [0..37B] DO
    expectAddr _ i + BITSHIFT[subTask, subTaskMemShift];
    IF mem[expect] #NOT(expectAddr)THEN ERROR;
  ENDLLOOP;

  FOR i IN [0..37B] DO
    expect _ i + BITSHIFT[subtask, subTaskMemShift]
    mem[expectAddr] _ i;
  ENDLLOOP;
  setFIOtester[task, subTask];
  IOFetch _ 0;
  IOFetch _ 20B;
  IOStore _ 0C;
  IOStore _ 20B;
  ioReset[];

  FOR i IN [0..37] DO
    expectAddr _ i + BITSHIFT[subTask, subTaskMemShift];
    IF mem[expectAddr] #NOT(expectAddr)THEN ERROR;
  ENDLLOOP;
END

FIOtestRbase: PROCEDURE[Task: TaskN, subTask:subTaskN] =
BEGIN
-- subTask.0 gets or'd into RBASE.3 while subTask.1 gets or'd
-- into subtask.1. If RBASE = 0, rstk = 0 then
-- subtask 0 refs effective RBASE[0]
-- subtask 1 refs effective RBASE[1]
-- subtask 2 refs effective RBASE[2]
-- subtask 3 refs effective RBASE[3]
-- To test the fio tester we must set the first 20B RM locations to known
-- values then, reference and store from RM while under the influence of
-- the task/subtask mechanism of the fastio tester.

ioReset[];
FOR xRM in [0..77B] DO RM[xRM]_ xRM; ENDLLOOP;

setFIOtester[task, subTask];
T _ R0;
R0 _ 100C;
ioReset[];

```



```
expect_ LSHIFT[subTask, 4];  
IF t # expect THEN ERROR;  
RBASE_ subTask;  
t_ r0;  
RBASE_ RBASE[defaultRegion];  
%*+++++
```

* June 9, 1981 10:19 AM

```
mc[fioWaitC, 60]
rvrel[rmx11, 11];      rvrel[rmx12, 12];      rvrel[rmx13, 13];      rvrel[rmx14, 14];
rvrel[rmx15, 15];      rvrel[rmx16, 16];      rvrel[rmx17, 17];
```

fioTest:

```
call[fioTesterReset]; * perform full init incase we're running
```

```
call[getMemState];
```

```
t AND (memState.FIOtest); * see if our test is enabled
```

```
branch[FIOtestDone, ALU=0];
```

```
noop;
```

```
call[EcOn]; * remove this when nolonger debugging fio jig
```

```
call[iSboard]; * this test from Midas & interrupting at
```

```
* arbitrary places.
```

```
call[initBrs];
```

fioIBrL: * init the brs for all the membases so that

```
call[nextBr]; * the BR for memBase i contains i left shifted
```

```
skpif[ALU#0]; * by fio.subTaskBrShift (= i * 1000C)
```

```
branch[fioIBrXit]; * Note: subtask 0 ors no bits into memBase,
```

```
noop; * (placement 'cause of branch)
```

```
call[setMbase]; * subTask 1 or's 2 into memBase, subTask 2
```

```
rscr2 _ t; * or's 4 into memBase and subTask3 or's
```

```
noop; * 6 into memBase. Thus, for subTask 3, va = 0
```

```
rscr2 _ lsh[rscr2, fio.subTaskBrShift]; * will really reference 6000B since
```

```
call[setBr, rscr _ t-t; * we init base register 6 to contain 6000B.
```

```
branch[fioIBrL];
```

fioIBrXit:

```
call[iapTestTask];
```

* June 9, 1981 10:04 AM

%*****

This is the main loop. Each time the test passes through here, it increments the current test task. Ie., it proceeds to test the next task.

%*****

```

    call[fioIMem];
fioTaskL:
    call[apNextTestTask]; * top of task loop
    skipif[ALU#0];
    branch[fioTaskXit];
    noop;

    call[iaSubTask];

```

%*****

This is the sub-loop. Each time the test passes through here, it increments the current test Sub-Task. Ie., it proceeds to test the next sub-task for the current task.

%*****

```

fioSubTaskL:
    call[aNextSubTask]; * top of subtask loop
    skipif[ALU#0];
    branch[fioSubTaskXit];
    noop; * for placement

```

* initialize first 20B RM locations to be 0..20B BEWARE! This code clobbers RM locations used by various subroutines! Use subroutines that change RBASE or store into different regions with care.

* Restore first 64K of storage.
 call[fioRestoreMem];

* Awaken the task we're currently testing

```

    rscr _ fioTestLoc0C;
    rscr _ (rscr) + (fioTestLoc1C);
    call[apGetTestTask]; * return t = next task to run = apNextTaskX
    taskingOff;
    subroutine;
    link _ rscr;
    top level;
    LdTPC _ t; * TPC[apNextTaskX] _ fioTestLoc
    taskingOn;
    call[notifyTask]; * awaken the code that really does the test
fioEmuWait:
    noop; * give it time to awaken;
    branch[fioSubTaskL];

```

```

fioSubTaskXit:
    branch[fioTaskL];

```

```

fioTaskXit:
    branch[afterFIOtest];

```

* June 5, 1981 4:27 PM

 This code runs as different tasks and exercises the fio test jig.

The basic idea is to background RM[0..77] w/ [0..77B] ie., each rm location will contain its address. THEN proceed as follows:

Set fio jig to select current subtask
 Set RBASE to 0
 (Now, rm references are under influence of subtask logic)
 t_ rmx0 (rstk=0, rbase = current value or'd w/ subtask)
 rmx0_ 100c;
 Reset fio test jig (get rid of subtask influence)

Check that t contains correct value for rmx0
 Check that correct rmx0 contains 100C.

fioRmTest:

```

set[xtask, 1];
top level;
taskingOn, at[fioTestLoc];
call[fioTesterReset];           * do it early for paranoia
RBASE _ rbase[defaultRegion];
call[setMbase], t_ a0;         * use BR0

call[fioInitRM], t_a0;         * background RM Region 0
call[fioInitRM], t_1c;         * background RM Region 1
call[fioInitRM], t_2c;         * background RM Region 2
call[fioInitRM], t_3c;         * background RM Region 3

* Begin by checking the oring of bits into RBASE, RSTK

call[fioSetFIOtester];         * Automatically sets tester w/ current
* task, subtask value. When this subroutine returns, the subtask logic is enabled! BEWARE!!!

RBASE _ 0s;

t_ rmx0;                         * read rmx0 and write it while under influence of
tester
rmx0 _ 100c;
call[fioTesterReset];         * clear out the tester

rscr _ t;                         * rscr = value from RM during influence of subtask
call[aGetSubTask];
q_t;
RBASE_t;                         * save SubTask in Q
t_ rmx0, RBASE_ rbase[defaultRegion];

rscr2_ t;                         * rscr2 = "rmx0" for the rm region

t_ q;
t_ lsh[t, 4];
t # (rscr);
skpif[ALU=0];

fioRmRMX0Err:
error;

(rscr2)#(100c);
skpif[ALU=0];
fioRmRMX0Err2:
error;

```

* value we read when performing t_rmx0
 * doesn't match what we expect. Subtask
 * logic didn't work????

* rscr2=value in RMX0 that should have
 * been written w/ 100c while under the
 * the influence of the subtask logic

* June 9, 1981 9:40 AM

fiOStest

Test the use of IOFetch, IOStore. This test depends upon the proper initialization of the base registers and of storage. For a given subtask, specific bits will be or'd into memBase, and that causes known base registers to be used during storage references. (The Default base register is 0.) Each base register that corresponds to a subtask has the unique value, subTaskNumber*10000B. Thus a reference to vm 0 in subtask 0 will reference virtual location 0; however, a reference to vm 0 in subtask 2 will reference vm 20000, etc. This is how the test can check that fetches and stores from a given subtask behave properly.

There are 3 tests. The first test simply checks that IOFetch_0, IOFetch_20, IOStore_0, IOStore_20 leaves storage correct. The second test writes into "subtask location 0" and "subtask location 20". Then it performs the two IOFetch, IOStore operations. Storage must be correct at the end of this operation (it must have the new, dirty values). The third, more complex test performs the two IOFetch operations (which will need to take munches from the cache because they will be dirty), writes into "subtask location 0" and then performs two IOStores. The two IO stores should overwrite the data written into the cache by the processor.

fiOStest:

```
call[aHaltTask17];          * cause task 17 to hit brkpt if it runs
call[fiOSetFIOTester];
```

This is the first test. It merely performs two IOFetches followed by a delay and two IOStores. The program checks the data of both munches.

fiOSdoIOI:

```
rscr _ 20c;                * beware of subtask influence on RBASE!
call[fiOIOFetch2],t_a0;    * IOFetch & store the first two munches of
* memory, then check that storage is correct.
```

```
t _ a0;
IOStore _ t;
IOStore _ rscr;
```

```
FETCH _ rscr;
B_MD;
call[fiOTesterReset];     * wait for last fastio operation to complete
                          * before we change our fastio device!
```

* now we check that storage is correct

```
t _ 37c;
cnt _ t;
call[iSvaCtrl];          * two munches
```

fiOScheck1L:

```
noop;
call[nextSva];           * returns va in t
skipif[ALU#0];
branch[fiOScheck1Xit];
noop;
call[aGetSTva], sva _ t; * (placement 'cause of branch)
                          * t = current, subtask relative va; rtns t = subtask's
```

real va

```
FETCH _ T;
rscr _ MD;
t # (rscr);
skipif[ALU=0];          * compare addr w/ value
```

fiOSerr1:

```
clobbered word in location t
error;
loopUntil[CNT=0&-1, fiOScheck1L]; * current subtask should have referenced & not
                                      * mem[t] should be t
```

fiOScheck1Xit:

```
call[aChkPipeFlt];
skipif[alu=0];
```

fiOSerr1b:

```
error;
                          * there was some sort of error left in the pipe!
                          * examine the pipe using midas
```

This is the second test. It checks to see that IOFetch gets dirty munches from the cache. It checks only the dirty words in the munch.

```
call[aGetSTva], t _ a0;    * in subtask relative locations 0 and 20 store
rscr _ not(t);            * NOT( their contents)
```

```

STORE_t, DBuf _ rscr;

t _ t + (20c);
rscr _ not(t);
STORE _ t, DBuf _ rscr;

call[fioSetFIOTester];
B _ MD;
fioSdoIO2:
rscr_20c;
call[fioIOFetch2], t_a0;
* still modified.

t _ a0;
IOstore _ t;
IOstore _ rscr;

FETCH _ rscr;
B_MD;
call[fioTesterReset];

call[aGetSTva], t _ a0;
rscr2 _ (FETCH _ t);
rscr _ (MD);
t _ not(rscr2);
t # (rscr);
skpif[ALU=0];
fioSerr2a:
error;
STORE_rscr2, DBuf_rscr2;

rscr2 _ t _ (rscr2) + (20c);
FETCH _ t;
rscr _ (MD);
t _ not(t);
t # (rscr);
skpif[alu=0];
fioSerr2b:
error;

STORE_rscr2, DBuf_rscr2;
call[aChkPipeFlt];
skpif[alu=0];
fioSerr2c:
error;

```

* fetch and store the first two munches of
* hold as required for stores to complete

* **beware of subtask influence on RBASE !**
* memory. check that our modified words are

* wait for last fastio op to finish before we
* modify our fastio device !

* save correct address in rscr2

* should have read not(address)

* fastio device should have gotten the dirty
* munch that was in the cache.
* rscr2 = va, rscr = MD, t = expected value
* fix up our dirty & different location

* check the next munch. save addr in rscr2

* should have read not(address)

* rscr2 = va, rscr=MD, t = expected value

* fix up our dirty & different location

* there was some sort of error left in the pipe!
* examine the pipe using midas

This is the third test. Note that munches w/ va0, va20 are dirty! This was caused by fioSdoIO2. The IOfetches below will take data from the cache. The longWait that occurs after the two IOfetches prevents the processor from writing into the munch for Va 0 before the fio jig has fetched the data. Remember it takes a while for the IOfetch to run to completion!

```

fioSdoIO3:
call[fioSetFIOTester];
B _ MD;
rscr_20c;
call[fioIOFetch2],t_a0;

RBASE _ 0s;
t _ cml;
rmx7 _ t-t;
STORE_rmx7, DBuf _ t;

IOstore _ rmx7;
t _ 20c;
IOstore _ t;
RBASE _ rbase[defaultRegion];
FETCH _ t;
B_MD;
call[FIOTesterReset];

```

* dirty a munch after fio tester has fetched it.
* hold as required for stores to complete
* **beware of subtask influence on RBASE !**

* set word "zero" to -1

* make word (subtask relative) 0 dirty

* wait for last fastio op to finish before we
* modify our fastio device !

```
    call[aGetSTva], t_r0;
    FETCH _ t;
    rscr _ MD;
    t # (rscr);
    skipif[ALU=0];
fioSerr3:
    error;
value,
* t = address referenced.
    call[aChkPipeFlt];
    skipif[alu=0];
fioSerr3b:
    error;

fioStestBlock:
    block;

* see if mem[i] = i
* dirty data in cache interfered w/ storing
* proper data in fioTester. rscr = MD, t = expected
* there was some sort of error left in the pipe!
* examine the pipe using midas
* RETURN TO EMULATOR!
```

* June 9, 1981 9:11 AM

fioTesterReset
setFIOtester
fioSetFIOtester
iaSubTask
aNextSubTask
aGetSTva

fioTesterReset Use rmx17 and reset (zero) FIO tester
setFIOtester Use rmx17, t, rscr & set FIO tester
fioSetFIOtester Set FIO tester according to current task, subtask
iaSubTask Initialize aSubTaskX
aNextSubTask Return next value of aSubTaskX
aGetSubTask Return current value of aSubTaskX
aGetSTva Return "real" va seen by current subtask

fioTesterReset: subroutine; * use last rm location in this region

* CLOBBERS RMX17!!!!

```

rmx17 _ fioTestAddrC;
TIOA _ rmx17;
rmx17 _ (rmx17) - (rmx17); * wait 1 instruction after resetting device
OUTPUT _ rmx17; * before returning -- so device has time to
return, RBASE_ rbase[defaultRegion]; * stop oring bits into membase, rstk, & rbase!
```

setFIOtester: subroutine;

*

```

q _ link; * enter w/ t= task num, rscr = subtask num, both
top level; * right justified. CLOBBERS T, RSCR, RSCR2
t _ lsh[t, 14]; * left justify 4 bit task field (20B-4)
rscr _ lsh[rscr, 12]; * position 2 bit subtask next to task field
t _ t or (rscr);
rscr _ fioTestAddrC;
TIOA _ rscr;
OUTPUT _ t; * t = task, subtask control for fio tester
subroutine;
link _ q;
return;
```

fioSetFIOtester: subroutine;

```

q _ link; * beware subtask RM addressing influence
top level;
call[aGetSubTask];
call[apGetTestTask], rscr _ t;
```

* expand setFIOtester because once we've started subtasking, we can't access any return links we store into an rm location.

```

t _ lsh[t, 14]; * left justify 4 bit task field (20B-4)
rscr _ lsh[rscr, 12]; * position 2 bit subtask next to task field
t _ t or (rscr);
rscr _ fioTestAddrC;
TIOA _ rscr;
OUTPUT _ t; * t = task, subtask control for fio tester
subroutine;
link _ q;
return;
```

iaSubTask: subroutine;

```

t _ cml;
return, aSubTaskX _ t;
```

aNextSubTask: subroutine;

```

saveReturn[aRlink];
RBASE _ rbase[aSubTaskX];
t _ aSubTaskX _ (aSubTaskX) + 1;
RBASE _ rbase[defaultRegion];
rscr _ t - (maxSubTaskC);
returnAndBranch[Alink, rscr];
```

aGetSubTask: subroutine;

```

RBASE _ rbase[aSubTaskX];
return, t _ aSubTaskX, RBASE _ rbase[defaultRegion];
```



```
aGetSTva: subroutine;
subtask number, construct the va (and return it in T) as seen by the subTask, also given fioTest's
initialization of the base registers.
    saveReturnAndT[Arlink, rscr2];
    call[aGetSubTask];
    noop;
    t _ lsh[t, fio.subTaskMemShift];
    t _ t OR (rscr2);
    returnUsing[Arlink];

* given t = va & aGetSubTask returns the current
* save the va in rscr2
* get the current subTask number into T
* wait for rm to settle down
* position subtask number properly
* construct the proper va
```

* May 4, 1979 2:30 PM

```

fioIMem: subroutine;
  saveReturn[fioIMemRtn];
  call[setMbase], t_r0;
  call[fioIStorage];
  call[fioIStorage];
  call[clearCacheFlags];
  waitOnMapBusy[rscr2];
  call[sUseDefaultMcr];

  rscr2 _ t-t;
  call[setBr], rscr _ t-t;
  t _ not(FaultInfo[]);
  returnUsing[fioIMemRtn];

```

* do it twice to make sure clearing cache flags
 * doesn't leave dirty munches in cache
 * clear cache for first test

* clear out any pipe faults

* June 9, 1981 9:40 AM

fioIStorage

Initialize memory to contain "self". Ie., mem[i] = i. This routine may be called twice in a row to force all the data munches to storage (required if, for example, clearCacheFalgs gets called immediately after calling this routine).

```

fioIStorage: subroutine;
  saveReturn[fioIStorageRtn];
  call[iSvaCtrl];

```

fioIMemL:

```

  call[nextSva];
  skipif[alu#0];
  branch[fioIMemXit];
  branch[fioImemL], STORE _ t, DBuf _ t;

```

fioIMemXit:

```

  returnUsing[fioIStorageRtn];

```

fioRestoreMem: subroutine;

```

  t_ 177777C;
  cnt_t;
  loopUntil[CNT=0&-1,], t_ (Store_t)-1, DBuf_t;
  return;

```

fioIOFetch2: subroutine;

```

  rscr2_ link;
  top level;
  IOFetch_t;
  IOFetch_rscr;
  t_ FioWaitC;
  call[longWait];
  returnUsing[rscr2];

```

* IOFetch_t, IOFetch_rscr, wait a while

FIOtestDone:

```

  goto[afterFIOtest];

```

fioInitRM: subroutine;

* enter w/ t= RM region.

* This subroutine initializes an RM region w/ its address. IE. RM 22 has 22, RM 3 has 3, etc.

```

  RBASE_t;
  t_ lsh[t, 4];

```

four bits wide).

* t=rbase.

* Now t=high bits of RM address (remember that rstk is

```

  rmx0 _ t; rmx1 _ t+1; rmx2 _ t+(2c); rmx3 _ t+(3c);
  rmx4 _ t+(4c); rmx5 _ t+(5c); rmx6 _ t+(6c); rmx7 _ t+(7c);
  rmx10 _ t+(10c); rmx11 _ t+(11c);
  rmx12 _ t+(12c); rmx13 _ t+(13c);
  rmx14 _ t+(14c); rmx15 _ t+(15c);
  rmx16 _ t+(16c); rmx17 _ t+(17c);

```

```

  return, RBASE_ rbase[defaultRegion];

```

```
title[memChaosS];
```

```
%
```

Table of Contents by order of occurrence

```
sChaosTest          main program for chaos test
setChaosCode       patch vartious locations in IM
chaosSimInit       Start up the task simulator
startChaos         Start up a pass of entire chaos test
checkError         return ALU#0 if (loopOnError) and (errorOccured)
chaosError         location that handles chaos errors.  when at breakpoint, t = "error number"
iChaosScopeLoop   remembers that an error has occurred if loopOnError is true.
```

```
%
```

```
%
```

```
September 14, 1979  3:40 PM
```

```
    move sChaosChkAddr into memSubrsCkchaos to help with micro STORAGE FULL problem.  move
isChaosMem and iSavedHoldValue, too.
```

```
June 25, 1979  11:33 PM
```

```
    fix incomplete edit to the loop control of sChaosTest (sChaosTop).
```

```
June 25, 1979  10:48 PM
```

```
    Cause startChaos to use getChaosTaskFreq, fix old, old bug in chaosError (called getSubrScr
instead of getsSubrScr), add comments.
```

```
June 25, 1979  2:34 PM
```

```
    Make chaostest into subroutine called from memRWS.mc
```

```
June 25, 1979  8:34 AM
```

```
    Change placement locations to accommodate ifu entry points.
```

```
April 18, 1979  11:36 AM
```

```
    Move iChaosCache, iChaosCode into memChaosSubrs.mc
```

```
April 18, 1979  10:06 AM
```

```
    Remove sMainChaosRtn, perform _FaultInfo[] before temporarily enabling the conditional task.
```

```
February 28, 1979  11:07 AM
```

```
    Move actual test code into memChaosSubrs.mc
```

```
January 16, 1979  3:18 PM
```

```
    Fix code init so that the random addresses selected occur in sChaosRow0, sChaosRow1.  Rename
constants for clarity.  Reset mcr after cache init so that it no longer restricts the cache to one row
only.
```

```
January 13, 1979  12:23 AM
```

```
    Shorten maximum interval before Task simulator starts up.  Max longWait delay set to 37B.
```

```
%
```

```
top level;
```

* January 16, 1979 3:22 PM

%

CHAOS
Mixed Timing and Memory-Op Test

Goal: Test memory by generating random memory operations that are delayed by random amounts of time. The memory operations will be executed by two different tasks to increase the storage traffic and occurrence of odious boundary conditions.

Generating the Environment and Memory-Ops

The test scans two different rows of the cache (8 munches). Random numbers determine which row, which munch, which word will be referenced. This requires one bit of row address, four bits of word address (in the munch), two bits of munch address (the "addressing" of munches is encoded in the address bits outside the cache), and one bit of shadow address. The shadow address bit provides an extra collection of virtual addresses that map into the same two rows of the cache and consequently prevent the cache from containing all the munches of interest.

Cache Row0 and Row1 are the two cache rows tested. The address bits that encode (*columnNumber lshift skipCacheShift*) select columns zero through three respectively. The shadow bit, encoded by *4 lshift skipCacheShift*, provides enough address space to overflow the cache.

Chaos uses the following structure:

General Initialization

Background storage with known values (*mem[va] _ va*)

Run Chaos test

Use the random number generator to select the munches that will be present in the cache. Randomly set *cflags.Dirty*.

Randomly select one of the four possible test patterns for task 0 and for task 17. A test pattern is a pair of references separated by a delay (eg., *fetch, delay, store, or store, delay, store*).

Randomly select both the addresses referenced by task 0 and task 17, and the delay between those references.

Run the test.

Check that the interesting values are correct

Begin again

Random Selection of Munches in the Cache

The test must know which addresses to make present in each column of the first two rows of the cache, and it needs the value for the cache flag dirty bit.

Actually, the test needs only two random bits for cache initialization:

one bit selects *cflags.dirty* (or not)

The other bit selects the shadow bit.

For code initialization four fields are used in each random number:

wordX which selects which word in the munch

rowX which selects which of the two possible rows will be selected

colX which selects which column will be addressed

delay which selects the parameter to *longWait* which the code invokes after the first memory reference during the test.

%

* June 25, 1979 11:34 PM

%

codeRand

These constants and shifts describe the format applied to random numbers to decide which addresses the chaos test code will reference and to decide how long the delay will be after the first memory reference.

%

```

mc[codeRand.store, 40];
mc[codeRand.wordX, 17];
mc[codeRand.rowX, 20];
set[codeRand.colSize, 2];          * mc[codeRand.colX, 300];
set[codeRand.colShift, 6];
set[codeRand.delaySize, 5];       * mc[codeRand.delay, 17400]
set[codeRand.delayShift, 10];

mc[addr.shadowBit, lshift[4,skipCacheShift]]; * va bit for shadow address

* addrRand is a two bit nibble selected from the current two lsb of a random number.
* the test uses addrRand to determine how to initialize each column of the two rows
* in the cache.
mc[addrRand.dirty, 1]; * mask bit for address template.
mc[addrRand.shadow, 2]; * mask bit for address template

mc[instr.jcnLongJumpC, 0];
mc[instr.ffMask, 177400];

set[instr.ffShift, 10];
mc[instr.jcn4thru7C, 17];
set[instr.notJcn4Thru7Shift, 4]; * rt justify portion of an IM address not encoded by
set[instr.notJcn4Thru7Size, 10]; * the jcn[4:7] bits. also define size constant -- for ldf

set[chaosTestBase, 7000]; mc[chaosTestBaseC, chaosTestBase];          **duplicated in
memSubrsChaos.mc+++++
set[chaosDelayBase, 7400]; mc[chaosDelayBaseC, chaosDelayBase];      **duplicated in
memSubrsChaos.mc+++++

set[chaosStage0Loc, 5400]; mc[chaosStage0LocC, chaosStage0Loc];
set[chaosStageSimLoc, 6000]; mc[chaosStageSimLocC, chaosStageSimLoc];
set[chaosSimInitLoc, 6440];
    mc[chaosSimInitLoc0C, and[chaosSimInitLoc, 7400]];
    mc[chaosSimInitLoc1C, and[chaosSimInitLoc, 377]];

sChaosTest:
    pushReturn[];
    checkMtest[memFlags.sChaos, sChaosDone];
    t _ flags.taskSim; * don't bother to execute unless
    call[checkFlags]; * task simulating is enabled
    skipif[alu#0];
    branch[sChaosDone]; * no tasking, no chaos
    noop;

    call[isChaosMem]; * init memory so that mem[va] = va.
    call[isChaosX];

sChaosTop:
    call[nextChaosX]; * see if its time for another test
    skipif[alu#0];
    branch[sChaosTopXit]; * check all sva for consistency, then exit
    noop;

    call[isSavedHoldValue]; * perform task simulator init
    call[saveRandState];

chaosScopeLoop:
    call[iChaosCache]; * select cache addresses and flags
    noop; * for placement.
    call[isChaosCode]; * init RM for chaos test, WRITE delay constant

sChaosStart:
    call[startChaos]; * run the test with both tasks running
    call[checkError]; * see if an scope loop error occured in the past
    skipif[alu=0];
    branch[ichaosScopeLoop]; * perform scope loop

```

```
branch[sChaosTop];
```

```
sChaosTopXit:
```

```
call[sChaosChkAdrs];
```

```
sChaosDone:
```

```
call[disableConditionalTask];
```

```
returnP[];
```

* January 8, 1979 11:00 PM

%

Set Chaos Code

Patch the delay constant and staging locations in the chaos test as required by the current random number values. When the test executes, a branch must be made to the proper location in IM to perform the pair of memory ops and the intervening delay. **setChaosCode** initializes the jump to that test and it initializes the B-mux constant that provides the delay between the two operations.

Enter with T = 0 or 4 (task 0 or task sim, an offset into some tables), rscr2 = the last random number used, and sChaosRandV the current random number in use. The concatenation of the two store bits from these two pseudo random numbers provides an index into a table of IM locations. These locations point to the starting point of the proper test in the chaosTestBase table, and point to the address of the B-mux constant in the chaosDelayBase table. The contents of these tables are actual instructions, so addressing them is a matter of adding the constructed constant to a base value. The offset in T address the simTask locations. For example:

ChaosTestBase + 000	beginning of fetch-fetch test
+ 001	beginning of fetch-store test
+ 010	beginning of store-fetch test
+ 011	beginning of store-store test
+ 111	beginning of fetch-fetch (sim task)
+ 101	beginning of fetch-store (sim task)
+ 110	beginning of store-fetch (sim task)
+ 111	beginning of store-store (sim task)

setChaosCode: PROCEDURE[taskOffset, oldRandomV, currentRandomV] =

```

BEGIN
  index _ IF oldRandomV.store THEN 2 ELSE 0;
  index _ index +IF currentRandomV.store THEN 1 ELSE 0;
  index _ index + offset;

  gotoAddr _ index + ChaosTestBase;
  gotoInstrRightHalf _ 120B;          -- jc = 1, jn[0:1] = 01
  gotoInstrRightHalf _ gotoInstrRightHalf + (gotoLoc AND 17B);  -- jn[2:5]
  gotoLoc _ [ (gotoLoc AND (NOT 17B) -- isolate remaining bits--) LSHIFT 4];  -- position for
  FF constant
  gotoInstrRightHalf _ gotoInstrRightHalf + gotoLoc;

  IF index = 0 THEN IMRH[chaosStage0Loc] _ gotoInstrRightHalf
  ELSE IMRH[chaosStageSimLoc] _ gotoInstrRightHalf;

  ffConstant _ currentRandomV AND ffConstantMask;  -- isolate delay field
  ffConstant _ ffConstant LSHIFT ffConstantShift;  -- position for micro instruction

  codeLoc _ ChaosDelayTable + offset;
  instr _ IMRH[codeLoc];
  instr _ instr AND (not ffConstantMask);  -- isolate rest of instruction
  instr _ instr OR ffConstant;  -- add our new ff constant
  IMRH[codeLoc] _ ffConstant;
END;

```

%

* January 16, 1979 3:48 PM

```
setChaosCode: subroutine; * T = task offset (0=> task 0, 4 =>simTask)
* rscr2 = old random number, sChaosRandV = current random number
  saveReturnAndT[sRlink, sSubrScr];
```

```
  (rscr2) AND (codeRand.store); * see if store bit is set
  skipif[alu=0], t_ t-t;
  t_ 2c;
  (sChaosRandV) AND (codeRand.store); * see if store bit is set
  skipif[alu=0], rscr_ t; * copy current value into rscr
  rscr_ t + 1; * rscr = index

  call[getSsubrScr]; * restore task offset into T
  t_ t + (rscr); * t_ offset + index
  sSubrScr_ t; * save complete index in sSubrScr
```

setChaosCodeJump:

```
%
  The format for a long jump is that jcn[4:7] contains the low 4 bits of the target address, and
  FF[0:7] contains the remaining 8 bits of the target address. In the code below, instr.jcnLongJumpC is
  a constant that properly sets the other bits of jcn.
```

```
%
  rscr_ (instr.jcnLongJumpC); * encode long jump
  t_ rscr2_ t + (chaosTestBaseC); * generate address to jump to
  t_ t AND (instr.jcn4thru7C); * add low order bits of address
  rscr_ (rscr) + t;
```

```
%
  Now finish constructing the long jump to the target address. First right justify the remaining
  bits of the address (ie., those bits not encoded in jcn[4:7]), then left shift those bits into the FF
  field of the microinstruction.
```

```
%
  rscr2_ ldf[rscr2, instr.notJcn4Thru7size, instr.notJcn4Thru7Shift];
  rscr2_ lsh[rscr2, instr.ffShift];
  t_ rscr2;
  rscr_ (rscr) + t; * this is right half of the micro instruction

  call[getSsubrScr]; * choose between task0 stage location and
  t_ t AND (4C); * simulator stage location
  skipif[alu=0], t_ chaosStage0LocC;
  t_ chaosStageSimLocC;

  call[putIMRH]; * Write the GOTO[properTest]
```

setChaosCodeConstant:

```
  call[getSsubrScr]; * get address of instruction whose B-mux
  t_ t + (chaosDelayBaseC); * constant (FF) must be initialized
  call[getIMRH];

  rscr2_ not(instr.ffMask); * remove the current FF bits
  t_ t and (rscr2);
  rscr_ t;

  t_ ldf[sChaosRandV, codeRand.delaySize, codeRand.delayShift]; * rt. justify delay
  t_ lsh[t, instr.ffShift]; * position delay for ff constant
  rscr_ t OR (rscr); * rscr = new instruction

  call[getSsubrScr];
  t_ t + (ChaosDelayBaseC);
  call[putIMRH]; * write the delay constant
  returnUsing[sRlink];
```


* April 24, 1978 3:01 PM

%

Chaos Simulator Initialization Code

Set Q to zero upon entry (the sim code sets q to 1 when it is done). ChaosSimStageit has been initialized with a long branch to the proper test sequence by **setChaosCode**. Task 0 maintains responsibility for checking the consistency of the results.

This code does not "return" to caller because **chaosSimStageIt** has been patched with a branch to test code that explicitly (via goto) returns to the chaos checking code.

%

```

top level;
chaosSimInit:
    t _ q,                                at[chaosSimInitLoc]; * hold value passed in Q

    RBASE _ rbase[defaultRegion];
    q _ r0;

    rscr _ t-t;                            * zero the current membase
    call[setBR], rscr2 _ t-t;

    RBASE _ rbase[rm10];                  * initialize rbase and branch to the test
    top level;                            * allow pc to get off page
    noop;

chaosSimStageit:
    noop,                                  at[chaosStageSimLoc]; * patched by setChaosCode !!
    error;                                 * should never get here
% should branch to code that looks like,
    hold _ t, block;
    <test code sequence>
    branch[chaosSimXit];
%
    set[xtask,1];
chaosSimXit:
    t _ 0c;
    t _ t + 1, hold&tasksim _ t;
    noop;                                  * 1st instr after hold&tasksim_ doesnt count
    q _ t, block;                          * should never get here
    branch[.], breakpoint;
    set[xtask,0];

```

* June 25, 1979 10:49 PM

%

Start Chaos Code

Start up the chaos simulator and proceed with the current memory test. This code checks that all the relevant memory locations are correct.

%

```

knowRbase[defaultRegion];
startChaos: subroutine;
  pushReturn[];

  call[setMCR], t _ r0;

  rscr _ t-t;          * zero the current membase
  call[setBR], rscr2 _ t-t;

  call[getChaosTaskFreq];      * construct hold value
  q_t;

  taskingOn;
  t _ (chaosSimInitLoc0C);
  t _ t + (chaosSimInitLoc1C);
  subroutine;
  link _ t;
  top level;
  ldTPC _ simTaskLevelC;
  notify[simTaskLevel];
  noop;
  RBASE _ rbase[rm00];
chaos0Stageit:
  goto[.+1],          at[chaosStage0Loc];
  error;

chaosAfterChk0:          * arrive here via explicit goto from task0 test.
* Perform an exhaustive test of RM and storage values
  t _ q;          * see if sim task is done
  branch[., alu=0], t _ q;

  t _ rm01;
  t _ t # (rm00);
  skipif[alu=0];
sChaosErr01:          * rm00 # rm01
  branch[chaosError], t _ 1c;

  fetch _ rm00;
  t _ md;
  rm01 _ t;
  t _ t # (rm00);
  skipif[alu=0];
sChaosErr02:          * mem[rm00] # rm00
  branch[chaosError], t _ 2c;          * rm01 = MD

  t _ rm03;
  t _ t # (rm02);
  skipif[alu=0];
sChaosErr03:          * rm02 # rm03
  branch[chaosError], t _ 3c;

  FETCH _ rm02;
  rm03 _ MD;
  t _ rm03;
  t _ t # (rm02);
  skipif[alu=0];
sChaosErr04:          * mem[rm02] # rm02
  branch[chaosError], t _ 4c;          * rm03 = MD

* TEST Chaos SimTask registers and locations
  t _ rm11;
  t _ t # (rm10);
  skipif[alu=0];
sChaosErr11:          * rm10 # rm11

```

```
branch[chaosError], t _ 5c;

fetch _ rm10;
t _ md;
rm01 _ t;
t _ t # (rm10);
skpif[alu=0];
sChaosErr12:                                * mem[rm10] # rm10
branch[chaosError], t _ 6c;                  * rm11 = MD, t = bad bits

t _ rm13;
t _ t # (rm12);
skpif[alu=0];
sChaosErr13:                                * rm02 # rm13
branch[chaosError], t _ 7c;

FETCH _ rm12;
rm13 _ MD;
t _ rm13;
t _ t # (rm12);
skpif[alu=0];
sChaosErr14:                                * mem[rm12] # rm12
branch[chaosError], t _ 10c;                 * rm13 = MD
returnP[];
```

* April 27, 1978 3:19 PM

%

Chaos Error: breakpoint or scope loop

%

```

    knowRbase[defaultRegion];
checkError: subroutine;                * return alu#0 if (loopOnError) AND (error occurred)
    saveReturn[Srlink];                  * see if we care
    t _ memState.loopOnError;
    call[checkMemState];
    skipif[alu#0];                        * presume no errors
    branch[checkErrorRtn], t _ r0;        * don't loop so return w/ alu=0
    t _ memState.loopErrorOccured;
    call[checkMemState];
    skipif[alu=0], t _ r0;
    t _ (r0) + 1;
checkErrorRtn:
    returnAndBranch[Srlink, t];

    top level;
chaosError:
    RBASE _ rbase[defaultRegion];
    sSubrScr _ t;                          * remember error value

    t _ (memState.loopOnError);
    call[checkMemState];
    skipif[alu=0];
    branch[chaosErr1];                    * do the scope loop

    call[getsSubrScr];                    * get the error number into t. See code after
startChaos
    error;                                  * to decode significance of error number.

chaosErr1:
    call[loopErrorOccured];                * now fall thru to iChaosScopeLoop

iChaosScopeLoop:
    call[loopErrorOccured];                * remember that err occurred for future ref.
    call[restoreRandState];
    branch[chaosScopeLoop];

    set[xtask,0];
    knowRbase[defaultRegion];

```

```

%
June 23, 1981 8:59 AM
  Add memState.aProcShift
June 15, 1981 11:29 AM
  Remove TaskContinueLoc from MemA stuff; make Fiotest default false, add aMapTest
June 9, 1981 10:14 AM
  Provide for default initialization of memState (lh of memFlags word)
June 2, 1981 10:22 AM
  Removing more old defns associated w/ memMisc: memPipeAndFaultA.
June 1, 1981 2:23 PM
  Remove old defns associated with memMisc: memPipeAndFaultA
February 9, 1981 9:55 AM
  Move defn. of xPageXLo, xEndPageLo, xPageXHi, xEndPageHi to rbase where they fit.
February 5, 1981 5:59 PM
  Rename brx to cBrx to accommodate dllang.
August 13, 1979 5:55 PM
  Fix discrepancy between number of pseudorandom number patterns for cache data test and for storage
  test.
August 13, 1979 4:48 PM
  Add patterns 3-4 for cache data test.
June 28, 1979 11:13 AM
  Change memFlags location, again, to accommodate ifu entry point locations.
June 26, 1979 4:39 PM
  Add SBrHi and SbrLo
June 25, 1979 8:26 AM
  Change memSimInitLoc to accommodate changes in postamble locations due to changes for ifu entry
  point locations.
June 17, 1979 4:51 PM
  Change value of memFlagsLoc to accommodate ifu entry point locations.
May 7, 1979 2:13 PM
  Change config definitions to accommodate Rev C's encoding for icType and the encoding for
  mapDirtyb, MapParity
January 25, 1979 5:38 PM
  Add pipe2.nFaultsSize.
January 25, 1979 11:06 AM
  Remove simTaskLevel definition since taskSimulator now awakens 12B.
January 16, 1979 2:22 PM
  add memState.mcrVictim, memState.mcrVictimShift (real sMCRvictim lives in memstate) -- code may
  change sMCRvictim at will, yet the value used by the tests a whole won't be clobbered.
January 13, 1979 12:47 AM
  add sChaosRow0, sChaosRow1
January 12, 1979 4:59 PM
  remove sGetConfigRtn, sCountModulesRtn, sGetICtypeRtn. add xMapBase. fix sPat3EndC
January 11, 1979 9:57 AM
  add memFlags.sFlush
%

TITLE[memDefs];
%
  January 25, 1979 5:39 PM

%
* MCR values (not Bmux constants)
  set[mcr.mcrVshift, 13]; * left shift "column" for position in mcr
  set[mcr.mcrNVshift, 11]; * left shift "column" for position in mcr

* MCR bmux constants

  mc[mcr.dPipeVa,b0]; * read load pipeVA w/ cache victim
  mc[mcr.fdMiss,b1]; * force dirty miss references ==>store victim
  mc[mcr.useMcrV,b2]; * use mcrV as victim and mcrNV as next vicitm in cache misses
  mc[mcr.disBr, b7]; * disable base registers
  mc[mcr.disCF, b8]; * disable cache flags
  mc[mcr.disHold, b9]; * disable hold
  mc[mcr.noRef, b10]; * don't make memory reference
  mc[mcr.noRefHold, b9,b10]; * disable hold and memory references
  mc[mcr.WMiss, b13]; * wakeup fault task after every miss
  mc[mcr.noSEwake, b14]; * no single error wakeups
  mc[mcr.noWake, b15]; * never awaken fault task

```

* PIPE2 B mux constants and Values

```

mc[pipe2.refType, b0,b1];          * type of reference
set[pipe2.refTypeShift, 16];
mc[pipe2.subtask, b2,b3];          * subtask that made reference
set[pipe2.subTaskShift, 14];
mc[pipe2.task, b4,b5,b6,b7];       * task that made reference
set[pipe2.taskShift, 10];
mc[pipe2.emuFault, b8];            * emulator made a fault
set[pipe2.emuFaultShift, 7];
mc[pipe2.nFaults, b9,b10,b11];     * nFaults-1 at time pipe read
set[pipe2.nFaultsSize, 4];
set[pipe2.nFaultsShift, 4];
mc[pipe2.faultSrn, b12,b13,b14,b15]; * SRN of first fault

```

* PIPE3 (_Map) B mux constants

```

mc[pipe3.realPageMask,177777];

```

* PIPE4 (_Errors) B mux constants and Values

```

mc[pipe4.ref, b0];
mc[pipe4.Mfault, b1];
mc[pipe4.wProtect, b2];
mc[pipe4.dirty, b3];
set[pipe4.dirtyShift, 14]
mc[pipe4.memErr, b4];
mc[pipe4.ecFault, b5];
mc[pipe4.quadWordMask,1400];
set[pipe4.quadWordShift, 10];
mc[pipe4.syndrome,377];
mc[pipe4.syndromeWdX, b12,b13,b14];
set[pipe4.syndromeWdXShift, 1];

mc[pipe4.sexChange0, b1,b4,b5];
mc[pipe4.sexChange1, b12,b13,b14];

```

* PIPE5 B mux Constants and values

```

mc[pipe5.colBit1,b7];             * bit 0 of column field
mc[pipe5.colBit0,b6];             * bit 0 (left to right) of column field

set[pipe5.colShift, 10];          * amt to shift "col" to align w/ PIPE5 field

mc[pipe5.MbufBusy,b0];            * map buf busy
mc[pipe5.flushStore, b1];
mc[pipe5.tag, b2];                * cache tag bit
mc[pipe5.store, b4];              * store' bit
mc[pipe5.ifuRef, b5];             * ifu reference
mc[pipe5.dirty, b8];              * cache entry is dirty
mc[pipe5.vacant, b9];             * cache entry is vacant
mc[pipe5.wProtect, b10];          * cache entry write protected
mc[pipe5.beingLoaded, b11];       * cache entry being loaded
mc[pipe5.flagsMask, 360];         * mask to isolate cache flags

```

* FaultINFO B mux constants and Values

```

mc[faultInfo.Asrn, b4,b5,b6,b7];  * current Asrn used by hardware
set[faultInfo.AsrnShift, 10];
mc[faultInfo.emuFault, b8];       * emulator made a fault
set[faultInfo.emuFaultShift,7];
mc[faultInfo.nFaults, b9,b10,b11]; * nFaults at time pipe read
set[faultInfo.nFaultsShift, 4];
mc[faultInfo.faultSRN, b12,b13,b14,b15]; * SRN of first fault

mc[faultInfo.faultMask, 377];     * B8 thru B15

```

* **CFLAGS constants and values**

```

mc[cflags.dirty, b8];
mc[cflags.vacant, b9];
mc[cflags.wProtect, b10];
mc[cflags.beingLoaded, b11];
mc[cflags.mask, b8,b9,b10,b11];
set[cflags.lshift, 4];

```

* May 7, 1979 2:14 PM **MEMORY CONFIGURATION definitions**

```

set[config.modSZ, 4]; mc[config.modSZC, config.modSZ];
set[config.modPOS, 4];
set[config.icTypeSZ, 2];
set[config.icTypePOS, 2];
mc[config.m0, b8];
mc[config.m1, b9];
mc[config.m2, b10];
mc[config.m3, b11];
mc[config.mapDirtyb, b14];
mc[config.mapParity, b15];
mc[config.Asrn, b4,b5,b6,b7]; * current Asrn value used by hardware
set[config.AsrnShift, 10];

```

* June 23, 1981 9:02 AM **memFLAGS bit definitions: control which tests execute**

```

set[memFlagsLoc, 310]; mc[memFlagsLocC, memFlagsLoc];

mc[memFlags.cPipeVA, b0]; * check pipe va bits
mc[memFlags.cBR, b1]; * check base register bits
mc[memFlags.cAmem, b2]; * check cache Amemory bits
mc[memFlags.cComprs, b3]; * check cache comparators
mc[memFlags.cFlags, b4]; * check cache addressing mechanism
mc[memFlags.cAddr, b5]; * check cache addressing mechanism
mc[memFlags.Cboard, b0, b1, b2, b3, b4, b5];

mc[memFlags.mRW, b6]; * check Map read/write
mc[memFlags.mAddr, b7]; * check map addressing
mc[memFlags.Xboard, b6,b7];

mc[memFlags.dRW, b8]; * check cache Data bits (read/write)
mc[memFlags.dAddr, b9]; * check cache Data addressing
mc[memFlags.dHold, b10]; * check HOLD logic
mc[memFlags.Dboard, b8,b9,b10];

mc[memFlags.sRW, b11]; * check Storage bits (read/write)
mc[memFlags.sFlush, b12]; * check Flush_
mc[memFlags.sAddr, b13]; * check storage addressing mechanism
mc[memFlags.sChaos, b14]; * multi task, multi op interference test
mc[memFlags.Sboard, b11,b12,b13, b14];

mc[memFlags.aPipeAndFault, b15]; * test pipe and fault mechanism
mc[memFlags.Aboard, b15];

mc[memFlags.default0, memFlags.Cboard, memFlags.Xboard];
mc[memFlags.default1, memFlags.Dboard, memFlags.Sboard];
set[defaultMemFlags, ADD[ memFlags.default0!, memFlags.default1! ] ];

```

* **MEMSTATE definitions: these bits live in LEFT HALF of memFlags word**

```

mc[memState.usingOneColumn, b0]; * use only one column of cache
mc[memState.mcrVictim, b1,b2,b3]; * column to use if usingOneColumn
set[memState.mcrVictimShift, 14]; * position a column number to mcrVictim
mc[memState.useTestSyn, b15]; * turn on error correction
mc[memState.loopOnError, b14]; * loop on occurrence of errors
mc[memState.loopErrorOccured, b13]; * at least one error has occured
mc[memState.noWake, b12]; * value of mcr.noWake bit
mc[memState.PipeTest, b11];
mc[memState.FIOTest, b10];
mc[memState.aMapTest, b7];
mc[memState.aProcShift, b6];

```

```
mc[memState.noErrs, b9];          * don't look for storage errors

set[defaultMemFlagsLeft, ADD[memState.aMapTest!, memState.PipeTest!,
memState.aProcShift!]];

* Set defaultFlagsP to notify the task simulation mechanism not to run unless
* flags.conditionalOK is true.

sp[defaultFlagsP, flags.conditionalP];
```


* June 26, 1979 4:38 PM

* MEMORY oriented MACROS, and Memory RM definitions

```
m[waitOnMapBusy, ilc[(#1 _ pipe5)]
    ilc[(#1 _ (#1) and (pipe5.MbufBusy))]
    ilc[(dblbranch[.+1, .-2, alu=0])]
    ];

m[shortMemWait,
    ilc[(#1 _ cnt)]
    ilc[(t _ 12c)]
    ilc[(call[longWait])]
    ilc[(cnt _ #1)]
    ]; * save cnt, call[longWait],t_12c; restore cnt
```

* Check to see if memory test is enabled
 * eg., checkMtest[memFlags.cAmem, cAmemDone];

```
m[checkMtest,
    ilc[(t _ #1)]
    ilc[(call[checkMemFlags])]
    ilc[(dblbranch[.+1, .+2, alu=0])]
    ilc[(branch[#2])]
    ilc[(noop)]
    ];
```

* RM definitions FOR CACHE tests: MemC

```
RM[col, IP[RHIGH1]];
* use va from cachedata = r01
rm[flagsV, IP[RM1]];
```

* RM definitions for CACHE DATA tests: MemD

```
* USE "col" from CACHE TEST definition
RM[va, IP[r01]];
RM[CData, IP[RM1]];
```

* RM definitions FOR MAP tests: MemX

```
RM[Mrow, IP[RHIGH1]];
RM[Mcol, IP[R01]]1;
RM[MpatHi, IP[R10]];
RM[MpatLow, IP[RM1]];
RM[mcrBits, IP[R1]];
```

* RM definitions for STORAGE tests: MemS

```
RM[sva, IP[va]];
RM[sval, IP[va]];
* use "col" from memC definitions
RM[sChaosRandV, IP[RM1]]; * hold current random number for chaos test
RM[Sva2, IP[r10]];
RM[sExpected, IP[r10]]; * pattern we expected back from memory
RM[Swait, IP[RM1]];
RM[Smcr1, IP[stackPAddr]];
RM[Smcr2, IP[rscr3]];
RM[SBrHi, IP[stackPTopBits]];
RM[SBrLo, IP[klink]];
RM[stsyn, IP[rscr4]];
RM[Sdata1, IP[hack0]];
RM[Sdata2, IP[hack1]];
RM[Sxor, IP[hack2]];
```

* RM definitions for All of memory

```
RM[aUseSrn, IP[R01]];
* presume sva
```


* MEMC Definitions

* February 5, 1981 5:59 PM

Cache Subroutines: Loop control

```
rmRegion[rmForLoops];
knowRbase[rmForLoops];
```

```

    rv[cBrx,0];          * base register index
    rv[patX,0];         * pattern index
    rv[rowx,0];        * row index
    rv[colx,0];        * column index
    rv[col2x,0];       * column index
    rv[curPattern,0];  * current pattern
    rv[subrScr,0];     * subroutines' scratch reg
    rv[rlink,0];       * return link
    rv[flagsVal,0];    * current flags value
    rv[cva, 0];        * copy of va
    rv[cflagsv, 0];    * copy of a flags value
    rv[cBrCacheABits, 0]; * copy of "CacheA bits in BR" value
*   rv[cXX,0];         * 4 unused rm locations

*   mc[BrHiEndC, 1000]; * maximum value+1 for br hi
    mc[BrHiEndC, 400]; * maximum value+1 for br hi
    mc[brEndC, 40];    * maximum br index +1
    mc[colEndC, 4];    * maximum column index +1
    mc[rowEndC,100];   * maximum row index +1
    mc[pat1EndC, 17];  * [0..pat1EndC)
    mc[pat2EndC, 36];  * [pat1EndC..pat2EndC)
    mc[pat3EndC, 37];  * [pat2EndC..pat3EndC)
    mc[lastPatC, 37];  * no more patterns
    mc[flagsEndC,400]; * flagsV_20B,flagsV+20B UNTIL 400B
    * flags are 4 bits wide, left shifted from the right by 4

    set[cacheShift, 4];
    set[nBitsInRow, 6];
*   set[nBitsInCache, 17]; * use fewer bits whie we wait for alu chips
*   set[CABitsInPipe0, 11];
*   mc[CABitsInPipe0Mask, 777];
    set[nBitsInCache, 16];
    set[CABitsInPipe0, 10];
    mc[CABitsInPipe0Mask, 377];
    set[CABitsInPipe1, 6];
    mc[CABitsInPipe1Mask, 176000];
    set[CABitsInPipe1Shift, 12];
    set[skipCacheShift, add[cacheShift,nBitsInRow] ];
*   mc[cacheRowMask, 1760]; * mask to isolate the cache bits in a va
    mc[cacheRowMask0, AND[1760,177400] ];
    mc[cacheRowMask1, AND[1760,377] ];
*   mc[CABitsMaskC, 77777]; * mask to isolate the CacheA bits in a
*   mc[CABitsMaskC, 37777]; * mask to isolate the CacheA bits in a pattern
```

* August 13, 1979 5:54 PM
* Cache Data Board:

rmRegion[rmForMemDloops];
knowRbase[rmForMemDloops];

rv[cMunchVa,0];
rv[cMunchEnd,0];
rv[cdVa,0];
rv[cdVaEnd,0];
rv[curCDpattern,1];
rv[CDpatx,1];
rv[DsubrScr,0];
rv[Drlink,0];
rv[pcFlags,0];
rv[pcHi8,0];
* rv[cdXX,0];

mc[CDpat1EndC,20];
mc[CDpat2EndC,40];
mc[CDpat3EndC,60];
mc[CDpat4EndC,64];
mc[CDlastPatC,64];
mc[cdMaxVa,1shift[rowEndC!,6]];
* by the four columns

* MEMD

Loop Control Mechanisms

* current cache munch address
* last + 20B munch address
* current va
* last va +1
* current pattern
* current pattern index
* subroutines' scratch reg
* return link
* kludge place to keep presetCache
* local variables!
* 6 unused registers

* 6 = 4 addr bits in munch + 2 addr bits implied

* MEMS

```

* January 13, 1979 12:46 AM STORAGE TEST SUBROUTINES: LOOP CONTROL MECHANISMS

rmRegion[rmForStoreLoops];
knowRbase[rmForStoreLoops];
  rv[SpatX,0]; * pattern index
  rv[CURSPattern,0]; * current pattern value
  rv[svaX,0]; * low 16 bits of VA
  rv[sVaHiX,0]; * hi 8 bits of va
  rv[sMaxBrHi, 0]; * max BRHI+1
  rv[sNmodules,0]; * num storage modules
  rv[sSubrScr,0]; * scratch register
  rv[Srlink, 0]; * keeps return link for loop control guys
  rv[sTestFlags,0]; * shadow copy of memState kept in RM for speed.
rv[xPageXLo, 0]; * low 16 bits of current page number
rv[xPageXHi, 0]; * hi 16 bits of current page number
rv[xEndPageLo,0]; * low bits of current last page+1
rv[xEndPageHi,0]; * hi 16 bits of current last page+1
rv[xChipEndRasCas,0]; * last ras/cas addr+1 in map ics
* rv[sXX,0]; * 2 unused registers: MemX uses 5 registers

rmRegion[rm2ForStoreLoops];
knowRbase[rm2ForStoreLoops];

  rv[sVaHiOld, 0]; * used by diagnostics NOT by loop mechanism
  rv[sVaXold, 0]; * used by diagnostics NOT by loop mechanism
  rv[sMCRvictim, 7]; * cache column to use during storage tests IF
  * dirty misses and storage read/writes are to be encouraged by using MCR to force
  * the same column as victim. IF sMCRvictim >3 THEN setMCR[defaultMCR] ELSE
  * setMCR[ makeUseMcrV[sMCRvictim]]
  rv[simScr0, 0];
  rv[simScr1, 0];
  rv[simScr2, 0];
  rv[sChaosX,0]; * current index in Chaos test
  rv[sSavedHoldValue, 0]; * currend hold/task sim value
  rv[pingPongRtn, 0];
  rv[iapFltTskRtn, 0]; * formerly sGetICtypeRtn, but that's no longer needed
  rv[incSvaRtn, 0];
  rv[zeroMemoryRtn, 0];
  rv[saSetMcrRtn, 0];
  rv[sChaosRow0, 0];
  rv[sChaosRow1, 1];

* rv[sXX,0]; * One unused register.

  rm[saOrMcrNoWakeRtn, IP[pingPongRtn ]];

* NOTE: The Chaos test shares RM locations
* with the Cache Data subroutine RM locations. Exercise Caution!! Chaos must not call
* subroutines that execute in the context of the cache Data tests!!!

  rm[rm00, ip[cMunchVa]]; * RM for chaos, task 0
  rm[rm01, ip[cMunchEnd]];
  rm[rm02, ip[cdVa]];
  rm[rm03, ip[cdVaEnd]];

  rm[rm10,ip[curCDpattern]]; * RM for chaos, simulator task
  rm[rm11, ip[CDpatX]];
  rm[rm12, ip[dSubrScr]];
  rm[rm13, ip[dRlink]];

* LOOP and other Constants

  mc[Spat1EndC, 20]; * [0..pat1EndC), cycled 1
  mc[Spat2EndC, 40]; * [pat1EndC..pat2EndC), cycled 0
  mc[Spat3EndC, 64]; * [pat2EndC..pat3EndC), cycled va
  mc[Spat4EndC, 100]; * [pat2EndC..pat3EndC), random
  mc[SlastPatC, 100]; * no more patterns
  mc[SchaosEndC, 10000]; * number of Chaos iterations

```

* MEMA

* June 15, 1981 11:29 AM

%

The mema rm definitions must not occur as the last set of definitions: the fio test clobbers RM IN [0..17B) and the last two rm regions defined may well have those values.

%

```

rmRegion[rmForAsubrs];
knowRbase[rmForAsubrs];
    rv[aTestTaskX, 0];
    rv[aNfaultsX, 0];
    rv[aSrnX,0];
    rv[aNfaultsX2, 0];
    rv[aMakingFaults,0];
    rv[aSubrScr, 0];
    rv[aRlink, 0];
*    rv[aXX,0]; * 8 unused registers
knowRbase[defaultRegion];
*    rm[iapFltTaskRtn, IP[sgetICTypeRtn] ]; * steal rm from midas subroutine
* now iapFltTskRtn is defined directly in one of memSubrSS's regions
    rm[apSaveT0TpcRtn, IP[simScr0] ]; * more RM stealing
    rm[apSavedT0Tpc, IP[simScr1] ];
    rm[apSetTpcRtn, IP[simScr2] ];

set[memSimInitLoc, 5000];          mc[memSimInitLocC, memSimInitLoc];
mc[simInitLocC, memSimInitLoc];  * define the IM location that setHold should know about!

mc[fioTestLoc0C, 377];
mc[fioTestLoc1c, 3000];
set[fioTestLoc, add[fioTestLoc0C!, fioTestLoc1c!]];

mc[fioTestAddrC, 140000];          * Address of fioTestCode (xqts for each task)
set[fio.subTaskBrShift, 11];      * shift memBase index to correspond to bits in BR
set[fio.subTaskMemShift, 12];    * shift subTask index to corresponds to bits in BR

rm[aSubTaskX, IP[aNfaultsX] ];
rm[fioIStorageRtn, IP[aSrnX]];
rm[fioIMemRtn, IP[iapFltTskRtn] ];

mc[maxSubTaskC, 4]; * subTasks IN [0..maxSubTaskC)

```

```

* February 10, 1981 9:55 AM
rmRegion[rmForMapLoops];
knowRbase[rmForMapLoops];

rv[MpatX,0];
rv[Mrowx,0];
rv[Mcolx,0];
rv[curMPatHi,0];
rv[curMPatLow,1];
rv[curMWait,1];
rv[MsubrScr,0];
rv[Mrlink,0];

rv[MpageX,0];
rv[MwriteVal, 0];
rv[Maddr1, 0];
rv[Mwait, 24];
rv[Maddr2, 1];
rv[Mread1, 125252];
rv[Mread2, 125252];
rv[xMapBase, 0];
* rv[xXX,0];

*rmRegion[rmForStoreLoops];
*knowRbase[rmForStoreLoops];
*rv[xPageXLo, 0];
*rv[xPageXHi, 0];
*rv[xPageEndLo,0];
*rv[xPageEndHi,0];
*rv[xChipEndRasCas,0];
*rmRegion[defaultRegion];
*knowRbase[defaultRegion];

mc[MwaitEndC,60000];
mc[MwaitIncrC, 20000];

mc[Mpat1EndC, 21];
mc[Mpat2EndC, 22];
mc[MlastPatC, 22];

set[nBitsInPage, 10]

knowRbase[defaultRegion];

```

* MEMX

MAP TEST SUBROUTINES: LOOP CONTROL MECHANISMS

```

* pattern index
* row index
* column index
* current high 2 bits of map pattern
* current low 16 bits of map pattern
* current wait between write and check
* subroutines' scratch reg
* return link

```

```

* page number
* registers for xBoardLoop stuff

```

```

* first real page of storage
* 1 unused register.

```

* SEE MEMS section!

```

* low 16 bits of current page number
* hi 16 bits of current page number
* low bits of current last page+1
* hi 16 bits of current last page+1
* last ras/cas addr+1 in map ics

```

```

* [0..pat1EndC)
* [pat1EndC..pat2EndC)
* no more patterns

```

```
title[MemDesperateA];
top level;
```

```
%
```

```
June 17, 1981 9:29 AM
  Create this file from old contents of memSubrsS.mc
```

A FEW RULES

Subroutines clobber rscr, rscr2 and T unless otherwise specified at both point of call and in subroutine description. Subroutines return single values in T, and they accept single values in T. Two parameter subroutines accept their values in rscr and rscr2.
Global values for S board

Abbreviations used herein

i	Init
M	Map
col	Column
S	Storage
a	All
p	Pipe
ap	AllPipe (test pipe, use all of memory system)
nextFoo	increments foo loop variable, returns with fast branch
condition to check for done w/ loop	
getFoo	returns current value of loop variable, foo
getFooRtn	subroutine that returns in T the saved value of foo's
return link	
iFooCtrl	initialize foo loop control

```
%
```



```

* May 20, 1981 5:18 PM
msc[maxMunchX, 17];
** commented out!
sPingPong:
* rscr = pattern to use
* move a munch back and forth from storage.

    saveReturnAndT[pingPongRtn, SvaXold];
    t _ rscr;
    curSpattern _ t;

    rscr _ t _ t-t;
    call[presetCache], rscr2 _ t;

    call[initColCtrl];
sppSetupL:
* row w/ values = their va

    call[nextCol];
    skipif[alu#0];
    branch[sppWriteMunch];
    col _ t;

    call[getSvaXold];
    call[cRowForVa];
    rscr _ col;
    call[cVaForCrowCol];

    cnt _ maxMunchX;
    store _ t, t _ (DBuf _ t);
    loopuntil[cnt=0&-1, .-1], t _ t+1;

    call[getSvaXold];
    rscr _ (cflags.vacant);
    call[setCflags];

    branch[sppSetupL];

sppWriteMunch:
* with not(va)

    call[getSvaXold];
    sva _ t;
    call[colForVa];
    call[makeUseMcrv];
    call[setMCR];

    cnt _ maxMunchX;
    rscr _ not(sva);
    t_sva;
sppWriteML:
    store_t, DBuf _ rscr;
    t _ t+1;
    loopuntil[cnt=0&-1, sppWriteML], rscr _ not(t);

```

```

* 17s
REMOVE first * to really comment it out
* enter w/ T = virtual address to test

* preset the cache to point to zero

* background the words of each munch in this

* get next column or exit loop

* refetch saved va
* find out which row this va references

* t= row, rscr = column. returns a va

* background this munch. t = va
* cache[va] _ va
* loop, va _ va+1

* T _ va
* rscr _ flags value
* uses col = column.

* pcSetCflags sets cache flags

* now write each word of the target munch

* refetch saved va

* cache[va] _ not(va)
* va _ va + 1
* rscr _ not(va)

```

* May 20, 1981 5:24 PM

sppBounce:

* In an infinite loop do the following:
 * read a different va into the selected column. This will flush the test-munch to storage.
 * read the test munch and test the va. this brings it back into the cache.

```
t _ al;
cnt _ t;
call[getCurSpattern];
rscr2 _ t;
t _ sva;
store _ t, DBuf _ rscr2;
```

* keep curSpattern in rscr2
 * write curSpattern into SVA

sppRL:

```
t _ (sva) # (b0);
STORE _ t, DBuf _ t;
FETCH _ sva;
t _ MD;
t _ t # (rscr2);
skipif[alu=0];
error;
loopuntil[cnt=0&-1, sppRL], t_al;* t used below.
```

* force a different munch into this cache column
 * read and check sva

* check MD w/ curSpattern (its in rscr2)

* contents of sva # curSpattern

sppInfiniteL:

```
branch[sppRL], cnt _ t;
```

sppRtn:

```
returnUsing[pingPongRtn];
```

*** commented out!

REMOVE first * to really comment it out

* June 25, 1979 10:36 AM
%

Dirty Write Loop

This is a MIDAS subroutine to help debug the memory. Begin execution at "dirtyWriteLoop". When a breakpoint occurs, the map has been initialized. At that point the user should place a virtual address in the register sva, and then proceed.

The code sets the cache Amemory for the appropriate row to four different values that are sva+2000, sva+10000B, etc. The code also sets the cflags to vacant. When this is done, a store is performed followed by a long wait. At the end of the wait the test loops to the point where it sets the cache Amemory again.

```

dirtyWriteLoop: PROCEDURE[sva: Virtual Address] =
  BEGIN
    presetMap[];
    victimColumn _ 0;
    breakpoint;                                -- at this point, user sets sva appropriately (may
    change victim column, too)
    WHILE true DO
      col _ 0;
      FOR CNT _ -3, CNT _ CNT + 1, UNTIL CNT >0 DO
        sva _ sva + 2000B;
        setCAMem[sva, col];
        setCflags[sva, col, Vacant];
        col _ col+1;
      ENDOLOOP;
      sva _ sva - 4000B;
      setBR[0,0];
      setMCR[useMcrv, mcrV=victimColumn];
      MD _ 0, STORE _ sva;
      ENDOLOOP;
    END;

    %

    ** commented out!                          REMOVE first * to really comment it out
dirtyWriteLoop: top level;
    call[presetMap];
    col2X _ t-t;                                * init victim column
    breakpoint;                                  * SET sva HERE

dwMainLoop:
    cnt _ 3s, col _ t-t;                        * init cnt and col

idwL: * initialize the Amemory and the Cflags
    sva _ (sva) + (2000C);
    call[setCAMem];                              * set Mmemory to sva
    rscr _ cflags.vacant;                         * set Cflags to vacant
    call[setCflags], t _ sva;
    loopUntil[cnt=0&-1, idwL], col _ (col)+1;    * choose next column and loop

    sva _ (sva) - (4000C);                        * reset sva to original value
    rscr _ t-t;
    call[setBR], rscr2 _ t-t;                     * zero base registers
    call[getCol2];
    call[makeUseMcrV];                             * set mcrv to select col2x
    t _ t or (mcr.noWake);
    call[setMCR], ;                               * set mcr
    noop;                                          * noop for patches if you want one

dwStore:
    DBuf _ t, STORE _ sva;                       * HERES the store!!
    t _ 62C;
    call[longWait];
    branch[dwMainLoop];

    ** commented out!                          REMOVE first * to really comment it out

```

* July 1, 1979 3:43 PM

%

Storage checkout code.

sval = address 1	default = 0
sva2 = address 2	default = 2000
swait = long wait value	default = 100
smcr1 = mcr value	default = noWake+UseMCRV
smcr2 = mcr value	+ VNV=Column3
sdata1 = pattern to utilize	default = 0
sdata2 = pattern to utilize	default = 0
sxor = xor pattern	default = 177777
stsyn = test syndrome value	default = 0
rscr = scratch value	default = ?
rscr2 = scratch value	default = ?
br0 = base register 0	default = ?
SBrHi = value for BrHi	default = 0
SBrLo = value for BrLow	default = 0

This code is useful for checking out storage boards. It consists of a series of initialization instructions followed by an infinite loop that writes the munch specified by sval then the munch specified by sva2. There is a second infinite loop which reads sval followed by sva2. That loop will not execute unless the operator changes the instruction at **scoF** into a noop. Doing this will enable the second loop after the code has written the two munches specified by sval and sva2.

%

top level;

doScheckOut:

```

RBASE _ rbase[defaultRegion];
t _ lc;
stkp _ t;
call[iSboard]; * initialize the entire memory system.

```

```

sval _ t-t;
sva2 _ 2000c;
swait _ 100c;
smcr1 _ 0c; * leave from for patching
smcr1 _ (smcr1) or (mcr.noWake);
call[scoAddMcrVNV], t _ smcr1;
smcr1 _ t;
smcr2 _ t;
sdata1 _ t-t;
sdata2 _ t-t;
sxor _ 177777c;
stsyn _ t-t; * test syn
memBase _ 0s;
sBrHi _ 0c; * default br = 0
sBrLo _ t _ 0c;
rscr _ SBrHi;
call[setBR], rscr2 _ sBrLo;

```

sCheckOut:

```

t _ (smcr1) or (mcr.disHold);
call[setMCR];
t _ stsyn;
store _ r0, DBuf _ t;
loadTestSyndrome[t];
call[longWait], t _ swait;

```

scoInit: noop, breakpoint;

```

call[xVacateCacheRow], t _ sval;
t _ mcr.noWake; * set mcr.noWake (and reenable BRs)
call[setMCR]; * so that loading BRs can actually happen.

```

```

rscr _ SBrHi; * restore BR, which xVacateCacheRow
rscr2 _ SBrLo; * clobbered.
call[setBR];

```

scoS:

```

t _ cml; * scope trigger if you wish
TIOA _ t, rscr _ a0;
TIOA _ rscr;
call[setMCR], t _ smcr1; * initialize 1st munch

```

```
rscr _ sdata1;
call[scoLoadMunch], t _ sval;

call[setMCR], t _ smcr2;
noop;
scoFlush1:
fetch _ sva2;; * 1st munch to storage, fetch 2nd munch
call[longWait], t _ swait;
call[setMCR], t _ smcr1; * initialize 2nd munch
rscr _ sdata2;
call[scoLoadMunch], t _ sva2;

call[setMCR], t _ smcr2;
noop;
scoFlush2:
fetch _ sval;; * 2nd munch to storage, fetch 1st munch
call[longWait], t _ swait;

scoF: branch[sCos]; * noop for read transports

scoL:
t _ cm1; * scope trigger if you wish
TIOA _ t, rscr _ a0;
TIOA _ rscr;
fetch _ sval; * fetch first munch
call[longWait], t _ swait;
call[xGetPipe4]; * get pipe4 value into t
rscr _ pipe4.memErr;
rscr _ (rscr) or (pipe4.ecFault);
t and (rscr); * mask all but error indicators
skpif[ALU=0];
breakpoint;
scoGo:
noop;
fetch _ sva2; * fetch second munch
call[longWait], t _ swait;
noop;
branch[scoL]; * loop for read only transports
```

* June 25, 1979 10:26 AM

%

Subroutines for the storage checkout code.

%

scoLoadMunch: subroutine; * enter w/ t = va, rscr = pattern
rscr2 _ t;
t _ rscr;

scoLML:
DBuf _ t, rscr2 _ (Store _ rscr2) + 1;
t _ t # (sxor);
(rscr2) and (17c); * see if done w/ munch
loopUntil[ALU=0, scoLML];
return;

scoAddMcrVNV: subroutine; * enter w/ t = initial mcr value. add:
* mcr.useMcrV, mcr.useMcrNV, set victim, next victim columns to 3.
* Return mcr value in t, clobber rscr.

t _ t or (mcr.useMcrV);
rscr _ 3c; * victim = column 3
rscr _ lsh[rscr, mcr.mcrVshift];
t _ t or (rscr);
rscr _ 3c; * next victim = column 3
rscr _ lsh[rscr, mcr.mcrNVshift];
return, t _ t or (rscr);

```
title[MemMapA];
top level;
```

```
%
```

```
June 15, 1981 10:50 AM
Create this file
```

A FEW RULES

Subroutines clobber rscr, rscr2 and T unless otherwise specified at both point of call and in subroutine description. Subroutines return single values in T, and they accept single values in T. Two parameter subroutines accept their values in rscr and rscr2.
Global values for S board

Abbreviations used herein

i	Init
M	Map
col	Column
S	Storage
a	All
p	Pipe
ap	AllPipe (test pipe, use all of memory system)
nextFoo	increments foo loop variable, returns with fast branch
condition to check for done w/ loop	
getFoo	returns current value of loop variable, foo
getFooRtn	subroutine that returns in T the saved value of foo's
return link	
iFooCtrl	initialize foo loop control

```
%
```

* June 16, 1981 4:18 PM

%

Memory Map test using all of memory system

aMapTest: PROCEDURE = -- This test checks the map reference bit and that various faults cause wakeups. The only way to set the ref bit in the map is to make a memory reference!

```

BEGIN
  IF ~memState.aMapTest THEN RETURN;
  FOR x IN [0..lastMapPage] DO
    oldM_ readMap[x];
    setMap[oldM];           -- setting map will clear ref bit
    t_ readMap[x];
    IF t.ref=1 THEN ERROR;  -- ref bit non zero
    xSetBRforPage[x];      -- set BR to ref page x
    Prefetch[0];           -- prefetch will set ref bit
    t_ readMap[x];
    IF ~t.ref THEN ERROR;  -- ref bit should be one

    t.wp_ TRUE;            -- set write protect bit
    writeMap[x, t];
    xSetBRforPage[x];      -- setup BR again
    aMakingFaults_TRUE;   -- tell fault task we're making faults
    faultInfo_0;           -- task 17 will set this w/ hardware value
    Store_0, DBuf_ 0;     -- this should cause a write protect fault
    IF faultInfo.nFaults#0 THEN ERROR;  -- remember 0 means 1 fault...
    -- set ProcSrn to address the pipe entry
    ProcSrn_ faultInfo.firstSrn;  -- with the error in it
    IF ~pipe.wp THEN ERROR;  -- wp should be set in pipe
    ProcSrn_0;             -- correct ProcSrn

    t.wp_ t.dirty_ TRUE;  -- now test that touching vacant map entries
    writeMap[x,t];        -- causes the right sort of error
    faultInfo_0;         -- task 17 will set this w/ hardware value
    Fetch_ 0;
    t_ MD;
    aMakingFaults_ FALSE;
    ProcSrn_ faultInfo.firstSrn;
    IF ~pipe.pageFault THEN ERROR;
    ProcSrn_0;
    setMap[x,oldM];
  ENDFOR;

```

END

%

* June 16, 1981 4:18 PM
%

aMapTest

This test checks parts of the map that require all the boards of the memory system.

Check the ref bit (only set when a page is actually referenced)
Check that wp (write protect) bit causes page faults

RSCR3= original contents of current map entry
RSCR4= current page in Map
%

aMapTest:

```

pushReturn[];
call[disableConditionalTask];          * don't run HOLD simulator
call[iSboard];                          * init Sboard in case its not happened already
call[getMemState];
t AND (memState.aMapTest);             * see if our test is enabled
branch[aMapTestXit, ALU=0];
noop;

call[iapFltTask];
call[iMapPageCtrl];

```

aMapL:

```

* rscr3=map entry we read              * This code keeps rscr4=current page,

call[nextMPage];                        * save current map page in rscr4
skpif[alu#0], rscr4_t;
branch[aMapXitL];
noop;

call[xReadMapPage];                     * save current map entry in RSCR3
rscr3_ t;

call[xSetBRforPage], t_ rscr4;          * make current BR point to current map page
rscr2_ rscr3;                           * rscr2=mapBuf data=real page
call[writeMap], rscr_a0;                 * rscr= tioa bits

call[xReadMapPage], t_ rscr4;           * rtns w/ t= real page num, rscr= pipe4
(rscr)and(pipe4.ref);                   * after setting map, ref bit should be zero
skpif[ALU=0];                            * ref bit still set!

aMapErr0:
error;

call[xSetBrForPage], t_ rscr4;
t_a0;
PreFetch_t;
call[longWait], t_ 100c;
call[xReadMapPage], t_ rscr4;
(rscr)AND(pipe4.ref);
skpif[ALU#0];                            * preFetch should cause page to be referenced

aMapErr1:
error;                                   * ref bit (rscr=pipe4) not set!
noop;                                    * for placement

call[xSetBRforPage], t_rscr4;
rscr2_ rscr3;
call[writeMap], rscr_ a0;
call[xReadMapPage], t_ rscr4;           * now we see if we zeroed the ref bit
(rscr) and (pipe4.ref);
skpif[ALU=0];

aMapErr2:
error;                                    * we didn't succeed in clearing the ref bit
                                           * for page in rscr4.

call[xSetBRforPage], t_rscr4;
call[aMapClearMunch],t_ a0;             * must clear our target munch from cache
rscr2_ rscr3;
call[writeMap], rscr_ 2c;               * set write protect bit
t_1c;
aMakingFaults_t;
t_a0;

```

```

Store_t, DBuf_t;
call[longWait], t_100c;
aMakingFaults_a0;
t_Q;
t AND (FaultInfo.nFaults);
skpif[ALU=0];
aMapErr3:
error;

t_Q;
t_ t and (FaultInfo.faultSRN);
ProcSRN_t;
t_a0;
ProcSRN_t;
t_ not(Pipe4');
t AND (Pipe4.wProtect);
skpif[ALU#0];
aMapErr4:
error;

t AND (Pipe4.Mfault);
skpif[ALU=0];
aMapErr5:
error;
noop;

call[xSetBRforPage], t_rscr4;
call[aMapClearMunch],t_ a0;
rscr2_ rscr4;
call[writeMap], rscr_ 3c;
t_ 1c;
aMakingFaults_ t;
t_ a0;
Fetch_t;
Pd_ (Md);
aMakingFaults_ a0;

t_Q;
t_ t AND (FaultInfo.faultSRN);
ProcSRN_ t;
t_ not(Pipe4');
rscr_a0;
ProcSRN_rscr;
t AND (Pipe4.wProtect);
skpif[ALU#0];
aMapErr6:
error;

t AND (Pipe4.Mfault);
skpif[ALU=0];
aMapErr7:
error;

t AND (Pipe4.memErr);
skpif[alu#0];
aMapErr8:
error;
noop;

call[xSetBRforPage], t_rscr4;
call[aMapClearMunch],t_ a0;
rscr2_ rscr3;
call[writeMap], rscr_a0;
branch[aMapL];
aMapXitL:
noop;

aMapTestXit:
returnP[];

aMapClearMunch: subroutine;

```

* fault task put FaultInfo in Q
* nFaults=0 => 1 fault occurred.
* should have one fault.
* no fault or too many faults

* write protect should be set!

* should set mapTrouble when writing
* into a write protected. (low true val)
* for placement

* must clear our target munch from cache
* marking it dirty and write protected makes it vacant

* fault task left q = FaultInfo
* wp should still be on

* map trouble should be (low) true

* NOT a memory error...its a page fault
* memErr is low true
* for placement

* must clear our target munch from cache
* map to old value
* try the next Srn

* performs Flush_t

```
Pd_Md;  
Flush_t;  
return, Pd_Md;
```

```
* so that the map's flags get used
```

```
title[memPipeAndFaultA];
top level;
```

```
%
June 17, 1981 8:20 AM
    add iapMap to make it easy to cause faults -- they occurred by chance before.
June 16, 1981 5:22 PM
    Change apMakeNFIts to avoid reusing same address and yet to stay in same cache row.
June 16, 1981 4:07 PM
    Change faulttask to wait one more cycle after B_FaultInfo before blocking.
June 15, 1981 11:28 AM
    Change faultTasks to save FaultInfo in Q. Remove faultTaskContinueLoc.
June 1, 1981 3:07 PM
    More fixes to pipe for model 1.
May 29, 1981 11:12 AM
    Remove comments around various pieces of code -- they were commented-out
    to accommodate MicroD placement. This is first step in rejuvenating this test.
May 4, 1979 5:11 PM
    Change code to use WriteMap instead of defunct xWriteMap, and change it to use sUseDefaultMcr
    instead of defunct saSetMcr.
May 4, 1979 3:40 PM
    Convert to model 1 from model 0 sources.
```

```
%
%
```

A FEW RULES

Subroutines clobber rscr, rscr2 and T unless otherwise specified at both point of call and in subroutine description. Subroutines return single values in T, and they accept single values in T. Two parameter subroutines accept their values in rscr and rscr2.
Global values for S board

Abbreviations used herein

i	Init
M	Map
col	Column
s	Storage
a	All
p	Pipe
ap	AllPipe (test pipe, use all of memory system)
nextFoo	increments foo loop variable, returns with fast branch
getFoo	condition to check for done w/ loop
getFooRtn	returns current value of loop variable, foo
return link	subroutine that returns in T the saved value of foo's
iFooCtrl	initialize foo loop control

```
%
```

* June 19, 1978 5:59 PM

%

Memory Pipe And Fault Test

```

testPipeAndFault: PROCEDURE =
  BEGIN
    initFaultTask[];
    FOR testTask IN [1..14] DO
      FOR srn IN [ 1..15] DO
        setUpMemory[];          -- mem[0..17] in cache, cache row1 dirty, mem[20..37] absent.
        TPC[testTask] _ @pipeTaskCode;
        NOTIFY[testTask, srn];
      ENDOLOOP;
    ENDOLOOP;
    disableFaultTask[];
  END
pipeTaskCode: PROCEDURE[taskX, srn: CARDINAL]; =
  BEGIN
    --check RefType: 0 =>undefined, 1 =>storage read, 2 => storage write, 3 => map or non-storage
    op
    -- check task, subtask, nFaults, faultSRN

    setRBASE[defaultRegion];
    taskingOn[];
    aSetAsrnForWrite[srn];      * since this code makes faults it must xqt as taskX for faultTask
    to do right thing
    t_ FaultInfo[]; * this clears the faultInfo data in the pipe
    SETBR[0,0];
    t _ mem[0];      -- ref 0
    aIncAsrn[];
    mem[1] _ 1;      -- ref 1
    aIncAsrn[];
    t _ 20c;         -- ref 2: miss & force storage write (see setUpMemory)
    FETCH _ t;
    t _ MD;
    useSrn _ srn;
    FOR i IN [0..2] DO
      setPROCSRN[useSRN];
      myPipe2 _ PIPE2[];
      refType _ IF i = 2 THEN 2 ELSE 1;
      IF myPipe2.ref # refType THEN SIGNAL[pipe2.refTypeErr, myPipe2, refType];
      IF myPipe2.subTask#0 THEN SIGNAL[pipe2.SubTaskErr, myPipe2, 0];
      IF myPipe2.task # taskX THEN SIGNAL[pipe2.taskErr, myPipe2, taskX];
      IF myPipe2.emuFault THEN SIGNAL[pipe2.emuFaultErr, myPipe2, 0];
      myPipe4 _ PIPE4[];
      IF myPipe4.ref #1 THEN SIGNAL[pipe4.refErr, 1];
      myFaultInfo _ FAULTINFO[];
      IF myFaultInfo.emuFault THEN SIGNAL[faultInfo.emuFaultErr, myPipe, 0];
      IF myFaultInfo.nFaults # 7 THEN SIGNAL[faultInfo.nFaultsErr, myPipe, i];
      useSRN _ incSRN[useSrn];
    ENDOLOOP;
    FOR nFlts IN [1..6] DO
      aSetAsrnForWrite[srn];
      useSrn _ srn;
      makeNfaults[nFlts];
      FOR i IN [1..nFlts] DO
        setProcSRN[useSrn];
        myPipe2 _ Pipe2;
        IF myPipe2.emulatorFault THEN SIGNAL[pipe2.emuFaultErr, myPipe2,0];
        IF myPipe2.nFaults # nflts THEN SIGNAL[pipe2.nFaultsErr, myPipe2, i];
        IF myPipe2.FaultSRN #srn THEN SIGNAL[pipe2.FaultSrnErr, myPipe2, useSRN];
        myPipe_ FaultInfo[];
        IF myFaultInfo.emuFault THEN SIGNAL[faultInfo.emuFaultErr, myPipe, 0];
        IF myFaultInfo.nFaults # nflts THEN SIGNAL[faultInfo.nFaultsErr, myPipe, i];
        IF myFaultInfo.FaultSRN #srn THEN SIGNAL[faultInfo.faultSrnErr, myPipe, useSRN];
        useSrn _ incSRN[useSrn];
      ENDOLOOP;
    
```

```

        ENDLLOOP;
RETURN;
END;
initFaultTask: PROCEDURE =
BEGIN
noWakeOff[];           -- default = allow memory wakeups
saSetMcr[getSMcrVictim[] ] ;           -- set mcr for diagnostic
TPC[faultTaskX] _ @faultCode;
NOTIFY[faultTaskX];
RETURN;
faultCode:           -- This code executes when a memory fault occurs
taskingOn[];
setRBASE[rbaseOf[makingFaults]];
DO
        BLOCK;           -- wait here for next fault to occur
        IF aMakingFaults = 1 THEN
                BEGIN
                TPC[testTask] _ @makeNfaultsContinue
                END;
        ELSE IF aMakingFaults = 2 THEN
                BEGIN
                TPC[testTask] _ @aSetAsrnContinue;
                END;
        ELSE ERROR[]; -- awakened unexpectedly
        ENDLLOOP;
END;
disableFaultTask: PROCEDURE=
BEGIN
faultInfo _ FAULTINFO[];           -- clear out waiting wakeups
noWakeOn[]
saSetMcr[getSMcrVictim[] ] ;
TPC[faultTaskX] _ 7776B;
END;
makeNfaults: PROCEDURE[n] =
BEGIN
setBR[377B, 0];
aMakingFaults _ 1;           -- global value visible to fault handler
FOR i IN [1..n] DO
        t _ mem[i];
makeNfaultsContinue:           -- fault task will force us to continue here
        ENDLLOOP;
makingFaults _ FALSE;
END;
setUpMemory: PROCEDURE=
BEGIN
-- This procedure forces mem[0..17B] into the cache, and backgrounds row 1 of the cache w/
munches that don't include mem[20B..37B]. This enables the test to make a reference that is
guaranteed to cause a miss and a dirty write. Recall that 20B addresses the first row of the
cache and that increments of 1000B will not change the row address.

FOR i IN [0..17B] DO mem[i] _ i; ENDLLOOP;
FOR i _ 10020B, i + 1000B UNTIL 20000B DO
        mem[i] _ i;
        ENDLLOOP;
END;
incSrn: PROCEDURE[oldSrn] RETURNS[newSrn] =
BEGIN
newSrn _ oldSrn + 1;
IF newSrn > 15 THEN newSrn _ 1;
RETURN;
aSetAsrnForWrite: PROCEDURE[srn] =
BEGIN
aMakingFaults _ 2;
setBR[-1,0];
UNTIL config.Asrn = srn DO
        FETCH _ 0;
        T _ MD;
        aSetAsrnContinue;
        ENDLLOOP;
aMakingFaults _ 0;
setBR[0,0];
END;

```

```
aIncAsrn: PROCEDURE=  
BEGIN  
  xWriteMap[0,0];          -- write the map w/ virtual page 0 mapped  
  -- onto real page 0 (default state for diagnostics). Map writes cause Asrn to increment.  
  
  IF aMakingFaults # 0 THEN setBR[377B,0] ELSE setBR[0,0];  -- a side effect of xWriteMap is to  
  clobber the current BR, so it must be regenerated.  
  
END;
```

```
%
```

* June 9, 1981 10:22 AM

%

Pipe test

This code runs in different tasks and causes various sorts of memory references: Three normal memory references and a series of references that cause faults. The point of the test is to fill the pipe with different sorts of entries and to make sure that the pipe data reporting logic really works. The test has two phases. In the first phase three normal references occur and the data in the pipe gets checked. In the second phase a series of faults occur and then all the entries in the pipe are checked.

The "main program" for this test iterates through the various tasks and the various srn values. It sets up the pc for the correct task and causes it to run. The code at **apPipeTest** actually makes the references and checks the pipe.

%

apPipeTestCtrl:

```

pushReturn[];
call[disableConditionalTask];          * don't run HOLD simulator
call[iSboard];                        * init Sboard in case its not happened already
call[getMemState];
t AND (memState.PipeTest);            * see if our test is enabled
branch[apTestCtrlXit, ALU=0];
noop;

call[iapMap];                          * setup map for making page faults conveniently
call[iapFltTask];
call[iapTestTask];

```

apTestCtrlL:

```

call[apNextTestTask];
skpif[alu#0];
branch[apTestCtrlXit];
noop;
call[iapSrn];

```

apCtrlL2:

```

call[apNextSrn];
skpif[alu#0];
branch[apTestCtrlL];
noop;
call[apGetTestGo];
rscr_link;
call[apGetTestTask];
subroutine;
link _ rscr;
top level;
LdTPC _ t;
call[notifyTask];
noop;                                * give it time to awaken
branch[apCtrlL2];                    * try the next Srn

```

apTestCtrlXit:

```

returnP[];

```

apGetTestGo: subroutine;

```

coreturn;
branch[apTest];

```


* June 4, 1981 2:07 PM

```

apTest:
    set[xtask, 1];
    top level;

    RBASE _ rbase[defaultRegion];
    taskingOn;
    call[setMbase], t_a0;
    rscr _ a0;
    call[setBR], rscr2 _ a0;
    call[iapMemory];

    call[apGetSrn];
    call[aSetAsrnForWrite];
    have the proper value, it must execute at the aTestTaskX level.

    noop;
    t _ FaultInfo'[];
    call[setMcr], t_mcr.noSeWake;

    rscr _ a0;
    call[setBR], rscr2_ a0;

    t _ a0;
    FETCH _ t;
    rscr _ MD, t _ t+1;

    call[apGetSrn];
    aUseSrn_ProcSRN_t;

    rscr_ a0;
    rscr2_ 1c;
apTest1:
    call[apChkPipe2];

    * This code runs at various task levels.
    * init task specific things
    * use BR 0
    * set BR 0 to zero
    * init cache in case it has changed
    * Since this routine makes faults to cause Asrn to
    * for MicroD
    * clear out faultInfo
    * set current BR to zero
    * ref 0
    * no faults
    * reftype=1
    * rscr=numFaults=7=none, rscr2=ref type=1

    * check procSrn for correctness, keep FaultInfo in rscr

    rscr_ not(FaultInfo');
apTest2:
    call[apChkProcSrn], rscr2_ aUseSrn;
    call[apChkAsrn], rscr2_ aUseSrn;
    call[apChkFaults], rscr2_ a0;

    t_ a0;
    ProcSRN_ t;
    t _ 20c;
    FETCH _ t;
    rscr_ MD;

    rscr_a0;
    rscr2_1c;
    ProcSRN_ aUseSrn;

    call[apChkPipe2];
    rscr_ not(FaultInfo');
apTest3:
    call[apChkProcSrn], rscr2_ aUseSrn;
    call[apIncSrn], t_ aUseSrn;
    call[apIncSrn];

    * no faults
    * ref 2: cause dirty miss (see iapMemory)
    * ref type=1=read missing munch
    * must increment aSrn twice
    * since dirty miss writes two entries
apTest4:
    call[apChkAsrn], rscr2_ t;
    call[apChkFaults], rscr2_ a0;

    call[apIncSrn], t_ aUseSrn;
    ProcSRN_ t;
    rscr_ a0;
    rscr2_ 2c;
    call[apChkPipe2];

    ProcSRN_r0;
apTest5:

```

```

* June 4, 1981 12:59 PM
* Now cause a series of faults and check the pipe
* CHECK OTHER PIPE VALUES, TOO!

    noop;                                * for placement.
    call[setMcr], t_mcr.noWake;

    cnt_10s;                              * clear-out row 0 so that page faults
    t_1000c;                              * won't get any dirty victims in the cache
    loopUntil[CNT=0&-1, .], t_ (Fetch_t)+t; * thereby avoid problems computing
* expected value of ASRN.

    call[iapNFlts];

apFltL:
    call[apNextFltX];
    skipif[alu#0], t_a0;
    branch[apFltXitL];
    ProcSRN_ t;

* cause nFaults to appear in the pipe beginning
    call[apGetSrn];                       * begin faulting at proper place in pipe
    call[aSetAsrnForWrite];               * IE. firstFaultSrn= this value

    call[apGetFltX];                       * make nFaultsX errors
    call[apMakeNFlts];

    noop;                                * for MicroD
    call[apGetSrn];                       * point to right place in pipe to read
    aUseSrn_ ProcSrn_ t;

* Check FaultInfo values. Remember that _FaultInfo resets nFaults to 7

    rscr_not(FaultInfo');                 * check code needs rscr=faultInfo
    rscr2_ aUseSrn;

apFltsP1:
    call[apChkProcSrn];
    call[apGetFltX];
    t_ t-1;
    cnt_t, call[apGetSrn];

    noop;                                * for placement.
    call[apIncSrn];                       * proper Asrn=(starting Asrn+nFlts)
    loopUntil[CNT=0&-1, .-2];

apFltsP2:
    call[apChkAsrn], rscr2_ t;

    call[apGetFltX];
    call[apGetSrn], rscr2_t;             * t_ firstFaultSRN

apFltsP3:
    call[apChkFaults], t_ aUseSrn;

    call[iapNflts2];                     * FOR i IN [1..nFaults] check EACH pipe entry

apFltChkL:
    noop;                                * here for placement problems
    call[apNextFlt2];
    skipif[alu#0];
    branch[apFltL];                     * try outer, nFlts, loop again

    procSRN _ aUseSrn;
    rscr_a0;                             * nfaults
    rscr2_ 1c;                           * refType=read

apFltsP4:
    call[apChkPipe2], t_ aUseSrn;

    call[apIncSrn], t _ aUseSrn;
    aUseSrn _ t;
    branch[apFltChkL];

apFltXitL:
    call[setMcr], t_mcr.noSeWake;
    ProcSRN_ t;                          * reset ProcSrn to zero
    branch[.], block;                   * done w/ current test. allow emulator to run.

```

```
set[xtask, 0];
```

* June 3, 1981 5:46 PM

%

This subroutine checks to see if the contents of Pipe2 are correct:
 numFaults, firstFaultSRN, refType, subTask (must be 0), task (must
 be current task), EmuFault (must be 0)

ENTRY:

```
rscr=numFaults we expect
rscr2=refType we expect
t=firstFaultSRN (if numFaults #7)
```

on ERROR:

Arlink points to the caller of this subroutine

USE:

T, rscr, rscr2, rscr3, Arlink, Q

%

apChkPipe2: subroutine;
 rscr3_t;
 SaveReturn[Arlink];
 t_ not(Pipe2');

* rscr=numFaults, rscr2=ref type, t=1st FaultSRN
 * save firstFaultSRN for a while

* Check nFaults for correctness

```
t_t and (pipe2.nFaults);
rscr_ (rscr)-1;
rscr_ (rscr)AND (7c);
rscr_ lsh[rscr, pipe2.nFaultsShift];
t # (rscr);
skpif[ALU=0];
```

* mem system keeps nFaults-1 in pipe
 * so we decrement and mask to do same

apChkP2Err1:
 error;

* nFaults in Pipe2 does not match expected
 * value. See caller of subroutine.

* IF nFaults=7 (same as mask value) then we don't check firstFaultSRN

```
t # (pipe2.nFaults);
branch[apChkP2Cont, ALU=0], t_ not(Pipe2');
t_ t and (pipe2.faultSrn);
t # (rscr3);
skpif[ALU=0];
```

* rscr3=expected value of firstFaultSRN

apChkP2Err2:
 error;
 t_not(Pipe2');

* firstFaultSRN does not match expected
 * value. See caller of subroutine.

apChkP2Cont:
 rscr3_ t;

* copy Pipe2 into rscr3.

* check task=current task

```
call[apGetTestTask];
rscr_ lsh[t, pipe2.taskShift];
t_ (rscr3) and (pipe2.task);
(rscr)#t;
skpif[ALU=0];
```

* we're done w/ numFaults so rscr is free

apChkP2Err3:
 error;

* Expected task (rscr, shifted) is not
 * same as one in Pipe2 (t). See caller of subroutine.

* Check subtask=0;

```
t_ (rscr3) AND (pipe2.subTask);
skpif[ALU=0];
```

apChkP2Err4:
 error;

* Subtask should always be zero in
 * memPipeAndFaultA. See caller of subroutine.

* Check refType= rscr2;

```
t_ (rscr3) AND (pipe2.refType);
rscr2_ lsh[rscr2, pipe2.refTypeShift];
t#(rscr2);
skpif[ALU=0];
```

apChkP2Err5:

* ref type (rscr2, shifted) does not match

```
error;                                * value in Pipe2.

* Check EmuFault=0

    t_ (rscr3) AND (pipe2.emuFault);
    skipif[ALU=0];
apChkP2Err6:                        * Emulator fault bit is true, should be false.
    error;                              * see caller of subroutine.

    returnUsing[Arlink];
```

* June 3, 1981 5:46 PM

%

Check FaultInfo

These routines check the values in faultInfo. ALL OF THEM ASSUME rscr=FaultInfo. Examine subroutine caller to determine how error was invoked.

apChkProcSrn checks that rscr2=expected ProcSrn= faultInfo.procSrn

apChkAsrn checks that rscr2=expected Asrn=faultInfo.asrn

apChkFaults checks

rscr2=numFaults=faultInfo.numFaults

t=firstFaultSRN=faultInfo.firstFaultSRN (if numFaults#7)

emuFault is 0

%

mc[faultInfo.ProcSrn, b0,b1,b2,b3];

set[faultInfo.ProcSrnShift, 14];

apChkProcSrn: subroutine;

saveReturn[Arlink];

rscr2_ lsh[rscr2, faultInfo.ProcSrnShift];

t_ (rscr) AND (faultInfo.ProcSRN);

t # (rscr2);

skpif[ALU=0];

apChkProcSrnErr:

error;

returnUsing[Arlink];

* ENTER: rscr=FaultInfo, rscr2=expected ProcSrn

* ProcSRN should be same as expected val.
* see caller of subroutine.

apChkAsrn: subroutine;

saveReturn[Arlink];

rscr2_ lsh[rscr2, faultInfo.asrnShift];

t_ (rscr) and (faultInfo.asrn);

t#(rscr2);

skpif[ALU=0];

apChkAsrnErr:

error;

returnUsing[Arlink];

* rscr=faultInfo, rscr2=expected Asrn

* expected ASrn (rscr2, shifted) not same
* as one in FaultInfo (t). See caller of subroutine;

apChkFaults: subroutine;

rscr3_t;

saveReturn[Arlink];

rscr2_ (rscr2)-1;

rscr2_ (rscr2) and (7c);

rscr2_ lsh[rscr2, FaultInfo.nFaultsShift];

t_ (rscr) AND (faultInfo.nFaults);

t # (rscr2);

skpif[ALU=0];

apChkFaultsErr1:

error;

subroutine.

(rscr2) # (faultInfo.nFaults);
branch[apChkFaultsCont, ALU=0];

t_ (rscr) AND (faultInfo.faultSrn);

t # (rscr3);

skpif[ALU=0];

apChkFaultsErr2:

error;

noop;

apChkFaultsCont:

t_ (rscr) and (faultInfo.emuFault);

skpif[ALU=0];

apChkFaultsErr3:

error;

returnUsing[Arlink];

* rscr=faultInfo, rscr2=numFaults, t=
* SRN for first fault. NOTE: if numFaults
* is 7 (means no faults) then don't check
* firstFaultSRN
* Memsystem uses nFaults-1, so we do
* the same thing!

* expected num faults (rscr2, shifted)
* doesn't match faultInfo (t). See caller of

* if nfaults=mask value, then don't
* check value of firstFaultSRN.

* expected first fault srn doesn't match
* fault info. See caller of subroutine.
* for placement limitation

* don't expect emulator faults

*August 24, 1978 3:18 PM

%

iapMemory

Force the muncch containing mem[0:17B] into the cache and remove the mucnch containing mem[20B:37B] from the cache. Remember that increments of 1000B do not change the row address in the cache for a memory location.

%

iapMemory: subroutine;

saveReturn[Arlink];

cnt _ 17s, t_a0;

iapMemL1:

loopUntil[CNT=0&-1, .], t _ (Store_t)+1, DBuf_ t;

cnt _ 10s;

t _ 20c;

noop;

*** for MicroD**

t _ t + (1000c);

iapMemL2:

STORE _ t, DBuf _ t;

loopUntil[CNT=0&-1, iapMemL2], t _ t + (1000c);

returnUsing[Arlink];

* June 16, 1981 4:07 PM
%

iapFltTask

This subroutine initializes the fault task for the memory pipe and fault diagnostics. The fault task code examines makingFaults, an RM flag set by the diagnostics to determine what to do if there's been a fault.

IF makingFaults is true, tpc[testTask]_@testTaskContinueLoc; otherwise, there's been an error. The fault task uses the aMakingFaults Rbase.

%

```

iapFltTask: subroutine;
  saveReturn[iapFltTskRtn];
  t _ FaultInfo'[];
  call[setMcr],t_mcr.noSeWake;
  call[getFaultTaskLoc];
  subroutine;
  t _ 17c;
  top level;
  LdTPC _ t;
  noop;
  call[notifyTask];
  noop;
  returnUsing[iapFltTskRtn];

  * clear any waiting wakeups
  * set mcr for wakeups

  * link _ code loc from getFaultTaskLoc
  * the fault task is task 17

  * for microD
  * t = task to notify;
  * wait to assure it has run

getFaultTaskLoc: subroutine;
  coreturn;
  branch[faultTask];

  set[xtask, 1];
  top level;

faultTask:
  RBASE _ rbase[aMakingFaults];
  taskingOn;

apFltTaskL:
  block;
  (aMakingFaults)-1;
  branch[apFltTaskL, alu=0];

  error;

  * see if apMakeNfaults caused this fault

  * AWAKENED WHEN aMakingFaults NOT valid !!!!

apFltTask1:
  t _ not(FaultInfo');
  Q_t;
  branch[apFltTaskL];

  * come here if apMakeNfaults
  * block won't work until we do this

  * wait extra cycle before blocking

  set[xtask, 0];
  knowRbase[defaultRegion];

```

* June 17, 1981 8:49 AM

%

apMakeNFIts

Make a series of faults in the pipe. Enter with T = the number of faults to make. ENTER W/
MCR.noWake!!! Once the fault task wakes up because of a memory wakeup, it cannot go to sleep again
without performing _FaultInfo,
and THAT will reset nFaults which screws up the diagnostics.

%

mc[largerThanCacheC, 4000]; * this value should be larger than the cache

apMakeNFIts: subroutine;

saveReturnAndT[Arlink, AsubrScr];

rscr_ t_ 1c;

call[setBR], rscr2_ t;

* set BR to 1,,1 to cause faults on our refs

aMakingFaults _ t;

* aMakingFaults _ 1, to notify fault task

* that our faults are ok.

call[getAsubrScr];

t _ t - 1;

* remove one 'cause we test at end of loop

cnt _ t;

apmakeNFItsL:

FETCH _ t;

* this reference causes a fault

rscr _ MD;

t_ t+ (largerThanCacheC);,

loopUntil[CNT=0&-1, apMakeNFItsL];

rscr_a0;

call[setBR], rscr2_a0;

* reset BR to usual value

aMakingFaults _ a0;

* disallow further faults

returnUsing[Arlink];

iapMap: subroutine;

pushReturn[];

* This emulator subroutine sets up the map so that

pages w/ BRHI=1 are vacant. This makes is easy to guarantee that apMakeNFIts will cause faults

rscr2_a0, cnt_ 17s;

iapMapL:

rscr_1c;

call[setBR];

* rscr=brHi, rscr2=brLO

rscr_3c;

* wp+dirty means vacant

call[writeMap];

* leaves rscr2 as it is

loopUntil[CNT=0&-1, iapMapL], rscr2_ (rscr2)+(largerThanCacheC);

returnP[];

```

* June 2, 1981 10:18 AM
%
                                loop controls, misc subroutines
%
mc[maxTestTaskC, 16];
iapTestTask: subroutine;
    return, aTestTaskX _a0;

apNextTestTask: subroutine;
    saveReturn[Arlink];
    RBASE _ rbase[aTestTaskX];
    t _ aTestTaskX _ (aTestTaskX) + 1, RBASE _ rbase[defaultRegion];
    noop;                                * for MicroD
    rscr _ t - (maxTestTaskC);
    returnAndBranch[Arlink, rscr];
apGetTestTask: subroutine;
    RBASE _ rbase[aTestTaskX];
    return, t _ aTestTaskX, RBASE _ rbase[defaultRegion];

mc[maxNfaultsC, 7];                                * nFaults IN [0..maxNfaultsC)

iapNFlts: subroutine;
    return, aNfaultsX _ a0;
apNextFltX: subroutine;
    saveReturn[Arlink];
    RBASE _ rbase[aNfaultsX];
    t _ aNfaultsX _ (aNfaultsX)+1, RBASE _ rbase[defaultRegion];
    noop;                                * for MicroD
    rscr _ t - (maxNfaultsC);
    returnAndBranch[Arlink, rscr];
apGetFltX: subroutine;
    RBASE _ rbase[aNfaultsX];
    return, t _ aNfaultsX, RBASE _ rbase[defaultRegion];

iapNFlts2: subroutine;
    return, aNfaultsX2 _ a0;
apNextFlt2: subroutine;
    saveReturn[Arlink];
    RBASE _ rbase[aNfaultsX2];
    t _ aNfaultsX;
    rscr _ t+1;
    t _ aNfaultsX2 _ (aNfaultsX2)+1, RBASE _ rbase[defaultRegion];
    rscr _ t - (rscr);
    returnAndBranch[Arlink, rscr];

apGetFlt2: subroutine;
    RBASE _ rbase[aNfaultsX2];
    return, t _ aNfaultsX2, RBASE _ rbase[defaultRegion];

```

```

* June 3, 1981 11:52 AM
mc[maxSrnC, 20];
set[xtask, 1];
* Asrn IN [1..20)
aSetAsrnForWrite: subroutine;
saveReturnAndT[Arlink, AsubrScr];
t_ lshift[300,10]C;
TIOA_t, t_a0;

aSetAsrnL:
IOFetch_t;
rscr _ not(Config');
rscr _ (rscr) and (config.Asrn);
call[getAsubrScr];
* isolate and rt justify current asrn value
noop;
* for MicroD
rscr _ rsh[rscr, config.AsrnShift];
(rscr) - t;
loopUntil[alu=0, aSetAsrnL], t_a0;
* gets incremented before it gets used again

returnUsing[Arlink];

* June 2, 1981 10:19 AM
aIncAsrn: subroutine;
* cause Asrn to increment. Accomplish this
* by writing the map. The subroutine that writes the map also clobbers BR so this routine must
regenerate BR as well.
t_ lshift[300,10]C;
TIOA_t;
IOFetch_t;
RETURN;

iapSrn: subroutine;
t _ (r0)+1;
return, aSrnX _ t;

apNextSrn: subroutine;
saveReturn[Arlink];
RBASE _ rbase[aSrnX];
t _ aSrnX _ (aSrnX) + 1;
RBASE _ rbase[defaultRegion];
rscr _ t - (maxSrnC);
returnAndBranch[Arlink, rscr];

apGetSrn: subroutine;
RBASE _ rbase[aSrnX];
return, t _ aSrnX, RBASE _ rbase[defaultRegion];

apIncSrn: subroutine; * Increment an Srn value. wraparound to 2
t _ t+1; * when new srn>17B. Note: enter w/ T =
t-(20c); * current srn value
skipif[alu#0];
t _ 2c;
return;

apDecSrn: subroutine; * decrement an Srn value. wraparound to 17
t _ t-1; * when new srn=1. Note: enter w/ T =
Pd_t-1; * current srn value
skipif[alu#0];
t _ 17c;
return;

```

```

* August 27, 1978 7:51 PM

* apSaveT0Tpc
*   Save tpc[0] in apSavedT0Tpc so that a non emulator task can briefly
*   "run" as the emulator (so it can perform map_ operations).

* apRestoreT0Tpc
*   Restore tpc[0] that is stored in apSavedT0Tpc

* apSetTpc
*   t = task, rscr = location

apSaveT0Tpc: subroutine;
  saveReturn[apSaveT0TpcRtn];
  zeroHold[rscr2];
  RdTpc _ r0;
  t _ link;
  apSavedT0Tpc _ t; * we've stashed tpc[0] into apSavedT0Tpc
  call[resetHold];
  returnUsing[apSaveT0TpcRtn];

apRestoreT0Tpc: subroutine;
  saveReturn[apSaveT0TpcRtn];
  noop; * for MicroD

  RBASE _ rbase[apSavedT0Tpc];
  t _ apSavedT0Tpc, RBASE _ rbase[defaultRegion];
  rscr _ t;
  call[apSetTpc], t _ r0;
  noop; * for MicroD

  returnUsing[apSaveT0TpcRtn];

apSetTpc: subroutine; * t = task number, rscr = IM location
  saveReturnAndT[apSetTpcRtn, rscr2];
  noop; * for MicroD

  zeroHold[t];
  t _ rscr2;
  subroutine;
  link _ rscr;
  top level;
  LdTpc _ t;
  noop; * for MicroD
  call[resetHold];

  returnUsing[apSetTpcRtn];

```

```
title[MemProcA];
top level;
```

```
%
```

```
June 17, 1981 11:08 AM
    Create this file
```

A FEW RULES

Subroutines clobber rscr, rscr2 and T unless otherwise specified at both point of call and in subroutine description. Subroutines return single values in T, and they accept single values in T. Two parameter subroutines accept their values in rscr and rscr2.
Global values for S board

Abbreviations used herein

i	Init
M	Map
col	Column
S	Storage
a	All
p	Pipe
ap	AllPipe (test pipe, use all of memory system)
nextFoo	increments foo loop variable, returns with fast branch
condition to check for done w/ loop	
getFoo	returns current value of loop variable, foo
getFooRtn	subroutine that returns in T the saved value of foo's
return link	
iFooCtrl	initialize foo loop control

```
%
```

* June 17, 1981 11:08 AM

%

Test MD path in shifter/masker

The Shifter/masker allows the processor to deposit an arbitrary sized field into a word coming from MD. This test checks that these things all work. We assume that kernel's shifter test has already run. That test assures that the shifter works properly.

This test checks that the MD data paths associated with the shifter work

%

* June 17, 1981 11:16 AM

%

aProcShifter

This test checks parts of the map that require all the boards of the memory system.

Check the ref bit (only set when a page is actually referenced)

Check that wp (write protect) bit causes page faults

RSCR3= original contents of current map entry

RSCR4= current page in Map

%

aProcShifter:

```

pushReturn[];
call[disableConditionalTask];          * don't run HOLD simulator
call[iSboard];                         * init Sboard in case its not happened already
call[getMemState];
t AND (memState.aProcShift);          * see if our test is enabled
branch[aProcShifterXit, ALU=0];
noop;

```

```

rscr_a0;
call[setBR], rscr2_ a0;
Store_rscr, DBuf_rscr, t_ a1;
Store_ls, DBuf_ t;
Fetch_ 0s;                             * get md=zero

```

```

t_ DpF[t,1,0,Md];                       * isolate 1 bit (b15) in t and add to Md
t # (1C);
skipif[ALU=0];

```

aProcShiftErr1:

```

error;                                  * we expect all zeros because
                                          * Md is zero and the field was 0 length

```

```

t_ a1;
t_ DpF[t, 1,17, Md];                   * isolate 1 bit (b0) in t and add to Md
t # (100000C);
skipif[ALU=0];

```

aProcShiftErr2:

```

error;

```

```

Fetch_ls, t_ a0;
t_ DpF[t, 1,0, Md];                     * isolate one zero bit (b15) and add to Md
(t) # (177776C);
skipif[ALU=0];                          * we expect all ones because

```

aProcShiftErr3:

```

error;                                  * Md is all ones and the field was 0 length

```

```

t_ DpF[t, 1,17, Md];                   * isolate one zero bit (b0) and add to Md
(t) # (77777C);
skipif[ALU=0];                          * we expect all ones because

```

aProcShiftErr4:

```

error;                                  * Md is all ones and the field was 0 length
noop;

```

aProcShifterXit:

```

returnP[];

```

```

title[memrwC];
top level;
%
February 5, 1981 5:57 PM
    Add toplevel decl to singleStep to satisfy micro.
September 19, 1979 6:23 PM
    Move beginCTest to beginning of file, call disableConditionalTask from beginCTest, add
table of contents, remove surplus code from cacheComprTest. Add lost mods that constructed and used
ccColWrite (to save space & simplify the logic) -- changed cacheAddr and cacheCompr.
%
%*+++++
                TABLE OF CONTENTS, by order of Occurence
beginCTest  Control subroutine that calls the tests in thi file
cPipeVA:    Test the low 16 bits of the pipe
cBRrwTest: Test all the bits of the base registers
cacheAddr: Test the Cache A-Memory
cacheCompr: Test the cache comparators
ccColWrite: Subroutine to write Cache A.
cFlagsTest: Test cache flags memory
cacheAddrTest: Cache A addressing test
setCAAF:   Set current cache location and cache flags to all ones
resadCurrentCFlags: Read "current" value of CFlags
readOldCflags:     Read "old" value of CFlags
readCurrentCAMem:  Read current Cache Amemory
readOldCAMem:     Read "old" value of CacheA memory
%*+++++

```


* September 19, 1979 7:14 PM

```
beginCtest: top level;
  pushReturn[];
  call[disableConditionalTask];
  call[cPipeVa];
  call[cBRrwTest];
  call[cacheAddr];
  call[cacheCompr];
  call[cFlagsTest];
  call[cacheAddrTest];
  returnP[];
```

* FIRST TEST IN MEMRWC

```
* test pivpe va
* test base registers
```

* February 5, 1981 5:57 PM

```

set[xtask,1];
set[sslocC,210]; mc[ssloc,sslocC];
singleStep:
    taskingon;
    t_mcr.disBr;
    t_t+(mcr.disCF);
    loadmcr[t,t];
    t_ssloc;
    link _ t;
    toplevel;
    ldtpc _ r1;
    notify[1];
    noop;
    noop;
    branch[emulMem];
ssHere:
    rbase _ 17s, at[sslocC];
    breakpoint,t_r1;
* set t to zero when done
memAsel:
*   map _ r01;
    set[xtask, 0];
    store_ID;
    set[xtask, 1];
    store _ t;
    store _ r01;
memAsel1:
    preFetch_r01;
    set[xtask, 0];
    fetch _ ID;
    set[xtask, 1];
    fetch _ t;
    fetch _ r01;

memAsel2:
*   ioStore_t;
    ioStore_r01;

memAsel3:
    set[xtask, 0];
    dummyRef_r01;
    Fetch_ID;
    set[xtask, 1];
    ioFetch_t;
    ioFetch_r01;

memAselNotOK:
    store_r01, t_1c;
    fetch_r01, t_1c;
    t_t;
    skipif[alu#0];
tsklquit:
    goto[.],block;
    branch[memAsel];

* EMULATOR ASEL CHECKOUT COMES HERE
set[xtask,0];
emulMem:
    map _ r01;
    Flush_r01;
    branch[emulMem];
afterSingleStep:

```

* NON EMULATOR!!!

* INIT SINGLE STEP CODE

* turn off base registers, cache flags

* init our rbase

* begin single step. t controls blocking

* ASEL=0, FF[0:1]=0. FF OK (emulator)

* ASEL=0, FF[0:1]=1. FF OK

* ASEL=0, FF[0:1]=2. FF OK

* ASEL=0, FF[0:1]=3. FF OK

* ASEL=1, FF[0:1]=0. FF OK

* ASEL=1, FF[0:1]=1. FF OK

* ASEL=1, FF[0:1]=2. FF OK

* ASEL=1, FF[0:1]=3. FF OK

* ASEL=2, FF[0:1]=2. FF OK (emulator)

* ASEL=2, FF[0:1]=3. FF OK

* ASEL=2, FF[0:1]=0. FF OK

* ASEL=2, FF[0:1]=1. FF OK

* ASEL=2, FF[0:1]=2. FF OK

* ASEL=2, FF[0:1]=3. FF OK

* ASEL=0, FF not OK

* ASEL=1, FF not OK

* test to see if we should block

* ASEL=0, FF[0:1]=0. FF OK

* ASEL=2, FF[0:1]=0. FF OK

* September 13, 1979 4:37 PM

%

CHECK PIPE VA BITS!!!

First disable base registers and init cnt to allow 20B iterations.
Then perform a dummyRef once for every bit on marMux (1, 2, 4, 10, etc). Note that the value stored with dummyRef is kept in rscr, the value read from pipel is kept in rscr2. This code need not be single stepped.

%

cPipeVA:

```

pushReturn[];
checkMtest[memFlags.cPipeVA, cPipeVaDone];
cnt _ 17s;                * loop control: once for each bit
rscr _ r1;                * RSCR begins w/ 1
T_mcr.disBr;
T_T+(mcr.disCf);
t _ t OR (mcr.noRef);
call[setMCR];            * DISABLE BASE REGISTERS

```

cPipeVAL:

```

dummyRef _ rscr;        * RSCR goes to pipe
noop;
t_pipel;
rscr2_t;                * rscr2 gets pipeVA
t_t#(rscr);            * t _ pipeVA xor RSCR
skpif[alu=0];

```

cPipeVAerr:

```

ERROR;                * t = bad bits, rscr = value we wrote into
                    * pipe, rscr2 = value from pipe

loopuntil[cnt=0&-1, cPipeVAL], rscr_(rscr)+(rscr);    * shift left test value by one

```

cPipeVaDone:

```

returnP[];

```

* March 17, 1978 8:32 AM

%

CHECK BASE REGISTER BITS

```

FOR BR IN [0..31] DO -- once for every base register
  FOR brhi _ 1,brhi+brhi UNTIL brhi = BrHiEndC DO -- once for each bit
    FOR brlo _ 1, brlo+brlo UNTIL brlo = 0 DO -- once for each bit
      writeBR[br, brhi, brlo];
      IF pipe1 # brlo THEN ERROR;
      if pipe0.va # brhi THEN ERROR;
    ENDLLOOP; ENDLLOOP;
  ENDLLOOP;
%

cBRrwTest:
  pushReturn[];
  checkMtest[memFlags.cBR, cBRdone];
  Q_r0; * Q = base register being tested
  t_(37c);
  cnt _ t; * cnt controls Q-loop
  t_rscr _ r1; * rscr = brHI
  rscr2 _ t; * rscr2 = brLO
  t_mcr.discf;
  t _ t OR (mcr.noRef);
  call[setMCR]; * reenable base registers

brL:
  noop;
  call[setMbase],t_q;
  noop;

brWriteHiL: * come here from incrementing brHi=rscr
  branch[brWriteLoL]; * instruction for placement

brWriteLoL: * come here from incrementing brLo=rscr2
  branch[brl2];

brl2:
  CALL[setBR];
  dummyref _ r0; * rscr =>brhi, rscr2 => brlo
  noop; * force pipe entry
  t _ pipe1; * pipe1 = VA[8:23] = low 16 bits
  t _ t #(rscr2);
  skipif[alu=0];

cbrLow16ER: * rscr2=value written into BrLo, t=bad bits
  error; * q = current base register being tested

  t_pipe0;
  t_t and(CABitsInPipe0Mask);
  t_t#(rscr);
  skipif[alu=0];

cbrHi8ER: * rscr = value written into brhi. t = bad
  error; * bits that came back.
  * q = current base register being tested

brLoopCtrl:
  t_(rscr2)+(rscr2); * increment and check brlo
  loopuntil[alu=0, brWriteLoL],rscr2 _ t;

  rscr2 _ r1; * reset brlo, increment and check brhi
  t_(rscr)+(rscr);
  t#(BrHiEndC);
  loopuntil[alu=0,brWriteHiL],rscr_t;

  rscr _ r1; * reset brhi, increment and check BR
  t_(r1)+(q);
  loopuntil[cnt=0&-1, brL],q_t;

cBRdone:
  returnP[];

```

%

Test the cache address memory.

Proceed as follows:

For each pattern, write the entire cache address memory with known values. Then for each row and for each column write the memory again and check to see that the correct entry appeared in the pipe.

Load MCR as follows:

dPipeVa_Vic, FDmiss, useMcrV, DisCF, DisHOLD

mcrVec: ARRAY [0..3] OF [

[FDmiss, useMcrV, McrV:0, DisCF, DisHold],

[FDmiss, useMcrV, McrV:1, DisCF, DisHold],

[FDmiss, useMcrV, McrV:2, DisCF, DisHold],

[FDmiss, useMcrV, McrV:3, DisCF, DisHold]

];

FOR pat IN Npats DO

FOR row IN Row DO

FOR col IN Column DO

checkPattern _ getCPattern[pat,row,col];

va_0;

va.cacheBits _ row; -- select current row

LOADMCR[mcrVec[col] OR noRef];

setBrCacheABits[checkPatternj];

DBuf_0,STORE_va;

ENDLOOP;

ENDLOOP;

FOR row IN Row DO

FOR col IN Column DO

LOADMCR[mcrVec[col] OR dPipeVa_Vic OR noRef];

otherPattern _ NOT[getCPattern[pat,row,col]];

setBrCacheABits[otherPattern];

-- now perform the check

va_0; va.cacheBits _ row;

DBuf_0,STORE_va;

NOOP; --?????

pipe0 _ PIPE0[]; pipe1 _ PIPE1[];

vaBits _ getCaBitsFromPipe[];

pat _ not(OtherPattern);

pat _ pat AND (CABitsMaskC);

IF pipe1.cacheBits # row THEN ERROR;

ENDLOOP;

ENDLOOP: -- end of row check loop

ENDLOOP: -- end of pattern loop

%

* September 19, 1979 7:03 PM

INIT cache address code and data

cacheAddr:

pushReturn[];
checkMtest[memFlags.cAmem, caTestDone];
call[initCPatterns];

* initialize pattern loop control

caPatternL:

call[nextCPattern];
branch[caTestDone,alu=0];
noop;

* THIS IS MAIN, OUTER LOOP

* for placement

caRowInit:

call[initRowCtrl];

* initialize row loop control

caRowL:

call[nextRow];
branch[noRow,alu=0];
Q _ t;
call[initColCtrl];

* increment row value. rtn'd in T

* nextRow sets up this test

* save current row in Q

* initialize column loop control

caColL:

call[nextCol];
branch[noColumn,alu=0];
col _ T;
call[ccColWrite], t_a0;
branch[caColL];

* increment column value. rtn'd in T

* nextCol sets up this test

* save current col in col

* write current cache entry

noColumn:

branch[caRowL];

noRow:

* March 13, 1978 9:11 AM
loop

TEST THE CACHE DATA: we're inside several layers of

caMemTest:

call[initRowCtrl];

* initialize row loop control

caRowTestL:

call[nextRow];
branch[noRowTest,alu=0];
Q _ t;
call[initColCtrl];

* increment row value. rtn'd in T
* nextRow sets up this test
* save current row in Q
* initialize column loop control

caColTestL:

call[nextCol];
branch[noColumnTest,alu=0];
col _ T;

* increment column value. rtn'd in T
* nextCol sets up this test
* save current col in col

t_(mcr.dPipeVa);
t _ t OR (mcr.noRef);
call[mcrForCol];
call[setBrCacheABits], t_r0;

* set up mcr for current column
* use 0: dPipeVa causes data to be or'd.

call[vaForRow], t _ q;
va _ t;
DBuf_r0,STORE _ t;

* TEST proper cache entry

call[getCPattern];
t _ t and (CABitsMaskC);
call[getPipeCacheABits],rscr2_t;
rscr _ t;

* test 15 bit pattern
* isolate exactly the bits we believe in
* against pipe info. rscr2 is preserved!
* remember pipe hi 15 bits

%*

membase+va = the address we referenced. The value in va selects the cache row, and the value in
membase is the value used to test cacheA. Q=current row, col = current column (both in the cache).

%*

t_t#(rscr2);
skpif[alu=0];

* see if any bits are different
* t=bad bits, rscr = hi 15 bits from pipe,
* rscr2=expected pattern (rscr, rscr2 right
* justified). Q=current row, col=current column

caBadVaBits:

error;

call[chkPipeRow],t_q;
skpif[alu=0];

* check "row" bits of PIPE1

caBadRow:

error;

* Q=current row. For some reason we didn't
* read the row in the pipe we wanted

branch[caColTestL];

noColumnTest:

branch[caRowTestL];

noRowTest:

branch[caPatternL];

caTestDone: noop;

returnP[];

* September 30, 1978 6:24 PM
%

TEST CACHE COMPARATORS

```

FOR br IN [0..31] DO
  SETMEMBASE[br];
  FOR pat IN Npats DO
    FOR row IN Row DO
      FOR col IN Col DO
        -- check comparators using data from each column in this row
        FOR otherCol IN Col DO -- init current row
          pattern _ getCPattern[pat, col, row];
          IF col # otherCol THEN pattern _ NOT[pattern];
          va _ 0;
          va.cacheBits _ row;
          setBrCacheABits[pattern];
          setMcrForCol[otherCol, noRef];
          DBuf_0,STORE_va;
          ENDLLOOP;
        -- set up to generate test reference
        pattern _ getCPattern[pat, col, row];
        va _ 0;
        va.cacheBits _ row;
        setBrCacheABits[pattern];
        set mcr appropriately;
        DBuf_0, STORE_va;
        noop?;
        -- test the results of the reference
        pipe5 _ pipe5[];
        IF pipe5.col # col THEN ERROR;
        ENDLLOOP; -- column loop
      ENDLLOOP; -- row loop
    ENDLLOOP; -- pattern loop
  ENDLLOOP; -- base registers loop
ENDLOOP;
%
```


* December 7, 1977 4:31 PM

%

INITIALIZE THE CODE THAT TESTS THE CACHE COMPARATORS

Use same loop control subroutines as the cache address test code

%

cacheCompr:

pushReturn[];
checkMtest[memFlags.cComprs, ccTestDone];
call[initBrs];

* initialize base registers loop
* THIS IS MAIN, OUTER LOOP

ccBrL:

call[nextBr];
skpif[alu#0];
branch[ccTestDone];
noop;
call[setMbase];
call[initCPatterns];

* change membase
* initialize pattern loop control

ccPatternL:

call[nextCPattern];
skpif[alu#0];
branch[ccNoPats];
noop;

* for placement

ccRowInit:

call[initRowCtrl];

* initialize row loop control

ccRowL:

call[nextRow];
branch[ccNoRow,alu=0];
Q _ t;

* increment row value. rtn'd in T
* nextRow sets up this test
* save current row in Q

* September 19, 1979 7:03 PM
layers of loop

TEST THE CACHE COMPARATORS: we're inside several

cacheComprTest:

call[initColCtrl];

* initialize column loop control

ccColL:

call[initCol2Ctrl];

ccCol2L:

call[nextCol2];
branch[ccNoCol2,alu=0];
col _ T;
call[ccColWrite],t_al;
branch[ccCol2L];

* this loop inits the background values
* (background = NOT(pattern))
* nextCol sets up this test
* save current col in col
* write current col,row w/ not(current pattern)

ccNoCol2:

call[nextCol];
branch[ccNoCol,alu=0];
col _ t;

%

now that the current row has been properly backgrounded by the "col2" loop, force a known value into this row in the current column, then reference that value and see if the comparators find it.

%

call[ccColWrite], t_a0;

* write current col,row w/ current pattern

t_(mcr.disCF);
t_t+(mcr.noRefHold);
t _ t+(mcr.noWake);
call[setMcr];
call[getCPattern];
call[setBrCacheABits];

* set mcr to disable cache faults,
* disable hold

* knows about Q, currnet pat
* getCPattern returned value in T

call[vaForRow], t _ q;
va _ t;
DBuf_r0,STORE _ t;

* TEST proper cache entry

membase+va = the address we referenced. The value in va selects the cache row, and the value in membase is the value used to test the comparators. Q=current row, col = current column (both in the cache). Only one column contains the value implicit in the current membase. A hit in a different column means that the comparators failed.

noop;
rscr _ pipe5[];
call[chkPipe5col],t_col;
skipif[alu=0];

* see if matched proper column

ccBadCol:

error;

* T=bad bits, rscr=pipe5, col=column
* expected

branch[ccColL];

ccNoCol:

%

Add code here to make sure that misses occur when then should

%

branch[ccRowL];

ccNoRow:

branch[ccPatternL];

ccNoPats:

branch[ccBrL];

ccTestDone:

returnP[];

* September 19, 1979 6:57 PM

ccColWrite: % Write current (col) column with current pattern in
the row specified by Q. We xor (rscr) with the current pattern. This produces the current pattern
when rscr=0 and, the complement of the current pattern when rscr=-1, and (probably) a bug otherwise.
%

```
pushReturnAndT[];
t_mcr.fdmis;
t_t OR (mcr.noRef);
call[mcrForCol];
defaulted by mcrForCol

call[getCPattern];
call[setBrCacheAbits], t_(stack&-1)#t;

call[vaForRow], t_q;
DBuf_r0, Store_t;
returnP[];
```

* set up mcr for current column, other flags are

* knows about Q, SHC, current pattern
* get CPattern returned value in T

* write proper cache entry

* November 30, 1977 10:04 AM

%

TEST CACHE FLAGS: TREAT THEM LIKE A MEMORY

For each column and row entry in the cache, test all the possible cache flag values. The enigmatic method for reading and writing the cache flags reflects the fact that the flags weren't designed to be read and written like a memory.

Basically, write the cache flags as follows:

Turn on fdMiss, disHold, enable the flags and the base registers. Use useMcrV to select the current column. PRESUME the cflags value has been left shifted by four to align it with the proper position in the cache.

Let va = vaForRow[row] - flagsValue, and write the low 16 bits of the current base register with va. Now write the value as follows:

```
dummyREF _ flagsValue;
CFLAGS _ flagsValue;
```

To read the flags, the current contents of the A memory must be known and there must not be two hits in the cach when the reference occurs:

Disable hold, enable the flags, use UseMcrv to force a different column for the victim. Perform,

```
dummyREF _ va; pipe5 _ pipe5[];
```

where the pipe5[] occurs in the next instruction and va[0:14] are known to be in the A memory of the cache. Check that col = the column that should have matched and not the column that was chosen as victim.

FOR flagsV _ 0, flagsV+20B UNTIL 1000B DO -- generate vals properly positioned

FOR row IN Row DO

useFlags _ flagsV;

FOR col IN Col DO

-- first, initialize the current row so that subsequent efforts to read the cache flags that have been written will not result in two hits in the cache

```
setBr[0];
va _ vaForRow[row];
setMcrForCol[fdMiss, disHold, col2];
va _ va+1000B;
DBuf_0,STORE_va;
setMcrForCol[fdMiss, disHold, useMcrv, mcrv=col2];
```

-- now write a different flags pattern into the current column so that each column of the row will have a different value, a value not same as the one being tested

```
setBr[va-useFlags];
dummyRef _ useFlags;
CFLAGS _ useFlags;
useFlags _ useFlags+minFlagVal;
IF useFlags > maxFlags THEN useFlags _ minFlagVal;
ENDLOOP;
```

-- now check it

useFlags _ flagsV;

FOR col IN Col DO

```
setMcr[disHold, useMcrv, mcrv:~col];
va _ vaForRow[row] + col *1000B;
setBR[va];
dummyRef _ 0;
pipe5 _ pipe5;
IF pipe5.col#col THEN ERROR;
fval _ pipe5 AND cFlagsMask;
IF (fval # useFlags) # 0 THEN ERROR;
useFlags _ useFlags+minFlagVal;
IF useFlags > maxFlags THEN useFlags _ minFlagVal;
ENDLOOP; -- end col loop
```

ENDLOOP;

-- end row loop

ENDLOOP;

-- end flagsV loop

%

* March 14, 1978 11:16 AM

cFlagsTest:

```
pushReturn[];
checkMtest[memFlags.cFlags, cfTestDone];
call[setMbase],t_r0;
call[initFlagsCtrl];
```

cfL:

```
call[nextCFlag];
branch[cfTestDone,alu=0];
flagsV _ t;
call[initRowCtrl];
```

cfRowL:

```
call[getCflag];
flagsV _ t;
call[nextRow];
skpif[alu#0];
branch[cf1];
q_t;
```

cfColInitL:

```
call[initColCtrl];
```

cfColWL:

```
call[nextCol];
branch[cfBeginCheck, alu=0],col_t;
rscr _ col;
call[cVaForCrowCol], t _ q;
va _ t;
call[setCAmem];
rscr _ flagsv;
call[setCFlags], t _ va;
call[incFlagsV];
branch[cfColWL];
```

cfBeginCheck:

```
call[initColCtrl];
call[getCflag];
flagsV _ t;
```

cfColReadL:

```
call[nextCol];
skpif[alu#0], col_t;
branch[cfRowL];
```

```
rscr _ col;
call[cVaForCrowCol], t _ q;
va _ t;
rscr2 _ col;
call[readCflags], rscr _ t;
```

membase+va = the address we referenced. The value in va selects the cache row, and the value in membase gets us to the correct column in the row (because the cache row has been written with unique addresses for each column). Q=current row, col = current column (both in the cache). FlagsV contains the value we wrote into the cache flags for the current row and column.

```
rscr _ t;
call[chkPipe5Col], t _ col;
skpif[alu=0];
```

cfBadCol:

```
error;
```

```
call[chkCflags],t_flagsV;
skpif[alu=0];
```

cfBadFlags:

```
error;
```

```
call[incFlagsV];
branch[cfColReadL];
```

incFlagsV: subroutine;

```
FlagsV _ (FlagsV) + (cflags.beingLoaded);
```

```
* INIT cache flags read/write test
* read and write cache flags
```

```
* MEMBASE _ 0
```

```
* MAIN outer LOOP
```

```
* top of row loop
* restore flagsV since cfColReadL
* clobbers it
* check for next row
```

```
* try next set of flags
* keep row in Q
```

```
* top of column loop
```

```
* top of row write loop
```

```
* get unique va for this row/col
```

```
* va = addr, col = column
```

```
* set the flags for this col/row
```

```
* get unique flags for each column
```

```
* init column loop control
```

```
* reset flagsV to current value
```

```
* top of column read/check loop
```

```
* no more cols, try next row
```

```
* get unique va for this row/col
```

```
* save the va
```

```
* returns w/ pipe5 in T
```

```
* expects rscr = pipe5, t = col
```

```
* bad column. bad bits in T, pipe5 in rscr
```

```
* expected val in col
```

```
* check for proper cflags in rscr
```

```
* t = bad bits, flagsV = expected bits,
```

```
* rscr = pipe5
```

```
* adjust for unique flagsV for each col
```

```
* better be least signif. bit!
```

```
FlagsV _ (FlagsV) AND (cflags.mask);
skipif[alu#0];
FlagsV _ (cflags.beingLoaded);
return;
top level;
cfTestDone:
returnP[];
```

* February 21, 1978 6:53 PM

%

Cache Addressing Test

This test checks that the addressing mechanism (as opposed to the bits that hold cache address values) works. The algorithm works as follows:

Zero the amemory and the cache

Ascend thru the amemory and cach flags: check that the current value is zero and then set it to -1. If the current value is not zero, an earlier store clobbered this entry. In this case perform the "find UP" check test.

Zero the amemory and the cache flags

Descend thru the amemory and cache flags: check that the current value is zero and then set it to -1. If the current value is not zero, an earlier store clobbered this entry. In this case perform the "findDOWN" check test

Otherwise, the addressing works.

Find UP: zero the amemory and the cache flags.

Ascend thru the amemory and cache flags: before setting the current location to -1 check that the earlier clobbered location is still zero. If it is not zero, the previous store clobbered that location.

FindDown: same as findUP except descend thru the amemory and cache flags

ZeroCacheAndFlags[];

FOR row IN Row DO

FOR col IN Col DO

IF cache[row,col] #0 THEN findErrUp[row,col];

IF cacheFlags[row,col] #0 THEN findErrUP[row, col];

cache[row,col] _ -1;

cacheFlags[row,col] _ -1;

ENDLOOP;

ENDLOOP:

zeroCacheAndFlags[];

FOR Row DESCENDING IN Row DO

FOR col IN Col DO

IF cache[row,col] #0 THEN findErrUp[row,col];

IF cacheFlags[row,col] #0 THEN findErrUP[row, col];

cache[row,col] _ -1;

cacheFlags[row,col] _ -1;

ENDLOOP;

ENDLOOP:

-- These are subroutines used in the "addressing test" shown above.

findErrUp: PROCEDURE[r: Row, c: Col] =

BEGIN

zeroCacheAndFlags[];

FOR row IN Row DO

FOR col IN Col DO

IF cache[r,c] #0 THEN Signal CacheAddrUp[r,c, row-1, col-1];

IF cacheFlags[r,c] #0 THEN Signal CacheAddrUp[r,c,row-1, col-1];

cache[row,col] _ -1;

cacheFlags[row,col] _ -1;

ENDLOOP:

ENDLOOP:

END:

findErrDown: PROCEDURE[r: Row, c: Col] =

BEGIN

zeroCacheAndFlags[];

FOR row DESCENDING IN Row DO

FOR col IN Col DO

IF cache[r,c] #0 THEN Signal CacheAddrUp[r,c, row-1, col-1];

IF cacheFlags[r,c] #0 THEN Signal CacheAddrUp[r,c,row-1, col-1];

cache[row,col] _ -1;

cacheFlags[row,col] _ -1;

ENDLOOP:

ENDLOOP:

END:

%

* March 14, 1978 11:17 AM

cacheAddrTest:

```
pushReturn[];
checkMtest[memFlags.cAddr, catDone];
call[zeroCacheAndFlags];
call[initRowCtrl];
```

catRowUpL:

```
call[nextRow];           * see if done going up
skpif[alu#0];
branch[catUpXit];       * time to descend thru addresses
q _ t;                 * KEEP ROW IN Q
```

```
call[initColCtrl];
```

catColUpL:

```
call[nextCol];          * check if done w/ cols
skpif[alu#0];
branch[catRowUpL];     * try next row
```

```
col _ t;
call[readCurrentCflags];
skpif[alu=0];
branch[catFindUpF];    * flags were clobbered
call[readCurrentCAMem];
skpif[alu=0];
branch[catFindUpAd];  * amem clobbered
```

```
call[setCAAF];        * set amem and flags to -1
branch[catColUpL];
```

catUpXit:

* ascending stores did not causes a **detectable** addressing error. Try the same approach w/ descending addresses.

```
call[zeroCacheAndFlags];
call[initRowDown];
```

catRowDownL:

```
call[nextRowDown];
skpif[alu#0];
branch[catDone];
q _ t;                 * KEEP ROW IN Q
```

```
call[initColCtrl];
```

catColDownL:

```
call[nextCol];
skpif[alu#0];
branch[catRowDownL];
col _ t;
call[readCurrentCflags];
skpif[alu=0];
branch[catFindDownF]; * flags were clobbered
call[readCurrentCAMem];
skpif[alu=0];
branch[catFindDownAd]; * amem clobbered
```

```
call[setCAAF];        * set amem and flags to -1
```

```
branch[catRowDownL];
```


* December 13, 1978 9:57 AM

set[catPackShift, 3];
the column (all in a "packed" va).

* left shift the row value this much to make room for

catFindDownF:

catFindDownAD:

va _ q;
t _ col;
va _ lsh[va, catPackShift];
va _ (va) OR t;

* PACK row,,col INTO VA

call[zeroCacheAndFlags];
call[initRowDown];

catRowFindDL:

call[nextRowDown];
skpif[alu#0];
branch[catDone];
q_t;

* KEEP ROW IN Q

call[initColCtrl];

catColFindDL:

call[nextCol];
skpif[alu#0];
branch[catRowFindDL];
col _ t;

%*

These two errors indicate that *the last time we wrote the cache A memory or the flags* there was an addressing error. Unfortunately this means the reader must determine what the address was "last time". Q = current row and col = current column. **Subtract** one from the column number to determine the last address. If the column number goes negative, then the last column was 3 and the last row was current row +1 (this loop scans "down" the possible cache addresses). The packed va, contained in "va" may be decoded as follows: The 3 least significant bits are the column number of the address that got clobbered and the remaining bits are the row number. Hence, if va = 132, the clobbered address is row 13 column 2.

%*

call[readOldCflags];
skpif[alu=0];

catDownCFerr:

* va = packed, old, clobbered row,column
error;

* q=current row, col = current column,
* flags were clobbered

call[readOldCAmem];
skpif[alu=0];

catDownAmemErr:

* va = packed, old, clobbered row,column
error;

* q=current row, col = current column,
* amem clobbered

call[setCAAF];

* set amem and flags to -1

branch[catRowFindDL];

* December 13, 1978

catFindUpF:

catFindUpAD:

va _ q;

* PACK row,,col INTO VA

t _ col;

va _ lsh[va, catPackShift];

* VA = row,,col of clobbered location.

va _ (va) OR t;

call[zeroCacheAndFlags];

call[initRowCtrl];

catRowFindUpL:

call[nextRow];

skpif[alu#0];

branch[catDone];

q_t;

* KEEP ROW IN Q

call[initColCtrl];

catColFindUpL:

call[nextCol];

skpif[alu#0];

branch[catRowFindUpL];

col _ t;

These two errors indicate that *the last time we wrote the cache A memory or the flags* there was an addressing error. Unfortunately this means the reader must determine what the address was "last time". Q = current row and col = current column. **Subtract** one from the column number to determine the last address. If the column number goes negative, then the last column was 3 and the last row was current row -1 (this loop scans "up" the possible cache addresses). The packed va, contained in "va" may be decoded as follows: The 3 least significant bits are the column number of the address that got clobbered and the remaining bits are the row number. Hence, if va = 132, the clobbered address is row 13 column 2.

call[readOldCflags];

skpif[alu=0];

catUpCFerr:

* q=current row, col = current column,

* va = packed, old, clobbered row,column
error;

* flags were clobbered

call[readOldCAMem];

skpif[alu=0];

catUpAddrErr:

* q=current row, col = current column,

* va = packed, old, clobbered row,column
error;

* amem clobbered

call[setCAAF];

* set amem and flags to -1

branch[catRowFindUpL];

* December 5, 1978 12:18 PM

setCAAF: subroutine;
ones

```

pushReturn[];
t _ q;
call[vaForRow];
rscr _ t;
t _ col;
rscr2 _ t;
t _ rscr;
rscr _ cml;
call[putCAmem];
t _ q;
call[vaForRow];
rscr _ t;
t _ col;
rscr2 _ t;
t _ rscr;
rscr _ (cflags.mask);
call[putCFmem];
returnP[];

```

* set current cache location and cache flags to all

* t = va, rscr = brHi15, rscr2 = col

* t = va, rscr = flags, rscr2 = col

readCurrentCFlags: subroutine;

```

pushReturn[];
t _ q;
call[vaForRow];
rscr _ t;
rscr2 _ col;
call[readCflags];
t _ t AND (cflags.mask);
returnPAndBranch[t];

```

* rscr = va, rscr2 = col

* isolate flags from other pipe5 stuff

readOldCFlags: subroutine;

```

pushReturn[];
t _ rsh[va, catPackShift];
call[vaForRow];
rscr _ t;
t _ (va) AND (3c);
call[readCflags], rscr2 _ t;
t _ t AND (cflags.mask);
returnPAndBranch[t];

```

* isolate col

* rscr = va, rscr2 = col

* isolate flags from other pipe5 stuff

readCurrentCAmem: subroutine;

Therefore, read the memory w/ the value we expect to find!

```

pushReturn[];
t _ (mcr.dPipeVa);
t _ t OR (mcr.noRef);
call[mcrForCol];
call[setBrCacheABits], t _ r0;
call[vaForRow], t _ q;
rscr2 _ t;
DBuf _ r0, STORE _ t;
shortMemWait[rscr];
call[getPipeCacheABits];
rscr _ t;
t _ rscr2;
DBuf _ r0, STORE _ t;
shortMemWait[rscr2];
t _ rscr;
returnPAndBranch[t];

```

* the cache address memory requires destructive read.

* save va since shortMemWait clobbers t

* TAG bits now screwed

* does not clobber rscr2

* save hi 15 bits of va in rscr

* restore va

* fix TAG bits

readOldCAmem: subroutine;

Therefore, read the memory w/ the value we expect to find!

```

pushReturn[];
t _ q;
rscr _ va;
va _ lsh[t, catPackShift];
t _ (col) AND (3c);
va _ (va) OR t;
t _ (rscr) AND (3c);
col _ t;
t _ rsh[rscr, catPackShift];
q _ t;

```

* the cache address memory requires destructive read.

* SWAP "va state"

* va = packed copy of "current" state

* now, q = old row, col = old col

```

t _ (mcr.dPipeVa);
t _ t OR (mcr.noRef);
call[mcrForCol];
call[setBrCacheABits], t _ r0;
call[vaForRow], t _ q;
rscr2 _ t;
DBuf _ r0, STORE _ t;
shortMemWait[rscr];
call[getPipeCacheABits];
rscr _ t;
t _ rscr2;
DBuf _ r0, STORE _ t;
shortMemWait[rscr2];

t _ q;
noop;
t _ lsh[t, catPackShift];
t _ t OR (col);
rscr2 _ t;

t _ (va) and (3c);
col _ t;
va _ rsh[va, catPackShift];
q _ va;
t _ rscr2;
va _ rscr2;
t _ rscr;
returnPAndBranch[t];
CATdone: noop;
returnP[];

```

* save va since shortMemWait clobbers t
* TAG bits now screwed

* does not clobber rscr2
* save hi 15 bits of va in rscr
* restore va
* fix TAG bits

* rscr2 = original value of packed va

* "current" col restored

* "current" row restored

* "old" packed va restored
* return w/ Amem in T

```
title[memRwd];
top level;
%
```

A FEW RULES

Subroutines clobber rscr, rscr2 and T unless otherwise specified at both point of call and in subroutine description. Subroutines return single values in T, and they accept single values in T. Two parameter subroutines accept their values in rscr and rscr2. Subroutines may use the global variables Mrow, Mcol, etc.

Global values for D board

Abbreviations used herein

i	Init
M	Map
col	Column
C	Cache
D	cacheData
nextFoo	increments foo loop variable, returns with fast branch
condition to check for done w/ loop	
getFoo	returns current value of loop variable, foo
getFooRtn	subroutine that returns in T the saved value of foo's
return link	
iFooCtrl	initialize foo loop control

```
%
%
```

May 26, 1981 9:50 AM

Invert the order of GallPat test and HoldTest-- hold test sets mcr.

May 21, 1981 3:47 PM

Add GallPat test for cache data.

September 19, 1979 7:16 PM

Add call to disableConditionalTask to beginDtest.

September 13, 1979 4:46 PM

Change control structure so that a single subroutine invokes each dboard test. Last previous mod on 28Dec78.

```
%
```

* May 22, 1981 10:54 AM

beginDtest:

```
pushReturn[];  
call[disableConditionalTask];  
call[cDataTest1];  
call[cdaTest];  
call[cdGallPat];  
call[cdHoldTest];  
returnP[];
```

%

To test the D-board cache data by reading and writing it:

```

TESTSYNDROME _ 0; -- initialize the D-board
--For the first set of tests, disable the cache flags and the base registers
test1MCR _ [ disHold];
SetMCR[test1MCR];
FOR va _ 0, va _ va + 20b UNTIL va >= 1000B DO
  col _ colFromVa[va];
  SetMcr[test1MCR OR useMcrV OR (mcrV:col)];
  store _ va, MD _ 0;
  ENDLLOOP; -- end of initialization sequence

FOR patX IN PatX DO -- begin testing current va
  FOR va IN [0..4000) DO
    pattern _ getPat[patX, va];
    md _ pattern, store _ va;
    ENDLLOOP; -- end of data writing loop
  FOR va IN [0..4000) DO
    pattern _ getPat[patX, va];
    FETCH _ va;
    x1 _ MDI;
    x2 _ MD;
    IF x1 #x2 THEN ERROR;
    IF x1 # pattern THEN ERROR;
    ENDLLOOP; -- end of check loop
  ENDLLOOP; -- end of pattern loop

SetCFlags: PROCEDURE[baseRegs, va, cacheFlags, column] =
  BEGIN
    realBR _ make24BitBR[baseRegs];
    mcrValue _ [useMcrV, mcrv: col, noRef, disHold];
    SetBR[realBR-cacheFlags];
    dummyRef _ va;
    CFLAGS _ cacheFlags;
  END;

```

%

* March 20, 1978 1:59 PM

CDDataTest1:

```

pushReturn[];
call[iDboard];
checkMtest[memFlags.dRW, cdXit];
t_rscr_(rscr)-(rscr);          * BRHI = va = 0
call[presetCache],rscr2 _ t;  * cache flags=0
call[setMCR], t _ r0;

```

cdlPatI:

```
call[iCDpatCtrl];
```

cdlPatL:

```

call[nextCDpat];
skipif[alu#0];
branch[cdXit];
noop;

```

cdlwVaI:

```
call[iCDvaCtrl], t _ r0;          * init cache beginning w/ va=0
```

cdlwVaL:

```

call[nextVa];                    * sets va directly
skipif[alu#0];                  * go read what we've written
branch[cdlrVa];
noop;
call[getCDpat];                 * knows about va
DBuf _ t, store _ va;
branch[cdlwVaL];

```

cdlrVa:

```
call[iCDvaCtrl], t _ r0;          * init cache beginning w/ va=0
```

cdlrVaL:

```

call[nextVa];                    * top of read/check loop
skipif[alu#0];
branch[cdlPatL];

fetch _ va;
noop;
t _ (r0) + (md);                * removed: + (r0); for a while
CData _ md;                      * try both immediate and regular md

```

```

%*-----
Perform two tests on cache data. First (cdlrerr1), make sure the values from MD and from MDI are the
same. Second (cdlrerr2), make sure the value we got from the cache is the same as the one we wrote.
In both cases, CData contains the value from the cache, and va is the current cache address.
%*-----

```

```

t#(CData);                       * T contains cache data from MD,
skipif[alu=0];                   * CData contains cache data from MDI
cdlrerr1:                       * va is the address we referenced.
error;                            * two reads of mem data don't mach

call[getCDpat];                   * put current pattern in rscr
rscr _ t;
t _ t # (CData);                 * compare current pattern, cache data
skipif[alu=0];                   * t = bad bits, CData = val from cache
cdlRerr2:                         * va is the address we referenced.
error;                            * and rscr = expected data
branch[cdlrVaL];

```

cdXit:

```
returnP[];
```


* January 12, 1978 10:20 AM

%

Exhaustive test of Cache Addressing

Perform exhaustive test of cache addressing mechanism:

```

zeroCache[];
FOR va IN [0..cdMaxVa) DO                                -- catch bad stores ahead of va
  IF cache[va] # 0 THEN FindErrUP[];
  cache[va] _ -1;
  ENDOLOOP;
zeroCache[];
FOR va DECREASING IN [0..cdMaxVa) DO                    -- catch bad stores below va
  IF cache[va] # 0 THEN FindErrDown[];
  cach[va] _ -1;
  ENDOLOOP;
FindErrUP: PROCEDURE[va] =
  BEGIN
  zeroCache[];
  FOR va2 IN [0..cdMaxVa) DO
    IF cache[va2] # 0 THEN ERROR;                        -- store at va2-1 clobbered va
    cache[va2] _ -1;
    ENDOLOOP;
  addrGhosts _ addrGhosts + 1;
  END;
FindErrDown: PROCEDURE[va] =
  BEGIN
  zeroCache[];
  FOR va2 DECREASING IN [0..cdMaxVa) DO
    IF cache[va2] # 0 THEN ERROR;                        -- store at va2-1 clobbered va
    cache[va2] _ -1;
    ENDOLOOP;
  END;
%
* December 28, 1978 1:47 PM
cdaTest:                                             * cache data addressing test
  pushReturn[];
  checkMtest[memFlags.dAddr, cdaTestXit];
  call[zeroCache0];                                     * zero cache. Begin at va=0
  call[iCDvaCtrl], t_r0;

cdaUpL:                                             * FOR va IN [0..cdMaxVa) DO
  call[nextVa];
  skipif[alu#0],t_cml;
  branch[cdaUpXit];
  * try decreasing va now

  fetch _ va;                                          * rscr = last va
  CData _ md;
  CData _ CData;
  skipif[alu=0];
  branch[cdaUpErr];
  * see which store caused the problem

  branch[cdaUpL], store_va, DBuf_t;                  * cache[va] _ -1

cdaUpXit:                                           * try the same test w/ decreasing va
  call[zeroCache0];
  call[iCDvaCtrl], t_r0;

cdaDownL:                                           * FOR va DECREASING IN [0..cdMaxVa) DO
  call[nextVa];
  skipif[alu#0];
  branch[cdaTestXit];
  * nextVa writes directly into va

  t _ cdMaxVa;
  t _ t-1;
  va _ t-(va);
  * cdMaxVa = last address + 1
  * max valid address into t
  * make va Decrease!
  fetch _ va;
  CData _ md;
  CData _ CData;
  skipif[alu=0];
  branch[cdaDownErr];
  * see which write caused the failure

  t _ cml;
  branch[cdaDownL],DBuf_t, store_va;                  * cache[va]_-1

```



```

cdaUpErr:
* We know that a store in the interval [0..va) has clobbered location va. Loop again
* and check location va after each store. KEEP (at entry) Va IN R1

    r1 _ va;
    call[zeroCache0];
    call[iCdVaCtrl], t_r0;

cdaUpErrL:
    call[nextVa];
    skipif[alu#0];
    branch[cdaUpNoFind];

    fetch _ r1;
    CData _ md;
    CData _ CData;
    skipif[alu=0];
cdaUpError:
    error;

    t_va;
    t-(r1);
    skipif[alu#0];
    branch[cdaUpErrL];

    t _ cml;
    branch[cdaUpErrL], store_va, DBuf_t;

cdaUpNoFind:
    error;

cdaDownErr:
* We know that a store in the interval [0..va) has clobbered location va. Loop again
* and check location va after each store.

    r1 _ va;
    call[zeroCache0];

    call[iCDvaCtrl], t_r0;
cdaDownErrL:
    call[nextVa];
    skipif[alu#0];
    branch[cdaDownNoFind];

    t _ cdMaxVa;
    t_t-1;
    va _ t-(va);

    fetch _ r1;
    CData _ md;
    CData _ CData;
    skipif[alu=0];
cdaDownError:
    error;

    t _ rscr _ (va);

    t-(r1);
    skipif[alu#0];
    branch[cdaDownErrL];

    t _ cml;
    branch[cdaDownErrL], DBuf_t, store_va;

cdaDownNoFind:
    error;

cdaTestXit:
    r1 _ 1c;
    returnP[];

```

* va-1 = last address used for a store.

* TEMPORARY EXPEDIENT

* FOR va2 IN [0..cdMaxVa) DO

* CData = bad value

* r1 = clobbered address.

* va-1 = addr whose store clobbered r1 addr

* see if we are about to write the location

* that was clobbered earlier. If so, skip it

* --its a transient problem. Continue testing

* since we might find another problem

* cache[va2] _ -1

* This suggests a transient error.

* va-1=last address used for a store.

* KEEP clobbered Addr in R1!!

* FOR va2 DECREASING IN [0..cdMaxVa) DO

* make va DECREASING from last

* valid address

* CData = bad value, r1 = clobbered address

* va+1 = addr whose store clobbered r1 addr

* remember this va for next time

* see if we are about to write the location

* that was clobbered earlier. If so, skip it

* --its a transient problem. Continue testing

* since we might find another problem

* cache[va2] _ -1;

* This suggests a transient error.

* just in case we continued from Midas

* December 27, 1978 11:00 AM

%

TEST HOLD, MD, MDI INTERACTIONS

Fetch known values under different timing circumstances to make sure the memory system returns the correct value. These tests try to retrieve a known value w/ zero, one or two noops between fetch_ and _md. Both md and mdi are tested. The test calls cdHoldReset to force a zero into md.

%

```

cdHoldTest:                                * init some values
  pushReturn[];
  checkMtest[memFlags.dHold, afterCDhold];
  t _ mcr.noWake;
  call[setMCR];
  DBuf _ r0, STORE _ r0;                       * cacheData[0] _ 0
  rscr _ t _ cml;
  DBuf _ t, store _ r1;                         * cacheData[1] _ 177777b

  call[cdHoldReset];
  t _ fetch _ r1;
  t _ md;                                       * zero noops
  t _ t # (rscr);
  skipif[alu=0];

cdHoldErr0:                                * t=bad bits, rscr = expected value, r1=addr
  error;

  call[cdHoldReset];
  t _ fetch _ r1;
  noop;                                       * one noop
  t _ md;
  t _ t # (rscr);
  skipif[alu=0];

cdHoldErr1:                                * t=bad bits, rscr = expected value, r1=addr
  error;

  call[cdHoldReset];
  t _ fetch _ r1;
  noop;                                       * two noops
  noop;
  t _ md;
  t _ t # (rscr);
  skipif[alu=0];

cdHoldErr2:                                * t=bad bits, rscr = expected value, r1=addr
  error;

  call[cdHoldReset];
  t _ fetch _ r1;
  t _ (md) + (r0);                             * zero noops, use mdi
  t _ t # (rscr);
  skipif[alu=0];

cdHoldMdiErr0:                             * t=bad bits, rscr = expected value, r1=addr
  error;

  call[cdHoldReset];
  t _ fetch _ r1;
  noop;                                       * one noop
  t _ (md) + (r0);                             * use mdi
  t _ t # (rscr);
  skipif[alu=0];

cdHoldMdiErr1:                             * t=bad bits, rscr = expected value, r1=addr
  error;

  call[cdHoldReset];
  t _ fetch _ r1;
  noop;                                       * two noops
  noop;
  t _ (md) + (r0);                             * use mdi
  t _ t # (rscr);
  skipif[alu=0];

cdHoldMdiErr2:                             * t=bad bits, rscr = expected value, r1=addr
  error;
branch[afterCDhold];

```

```
*      This subroutine forces a zero into md.
cdHoldReset:
  subroutine;
  fetch _ r0;
  t _ md;
  return;
  top level;
afterCDhold:
  returnP[];
```

* May 26, 1981 10:31 AM

%

cdGallPat:

Traditional gallpat test: zero memory then write each cell of the memory.
After each write, read all of the memory to see if any other cell has been effected by the write

%

cdGallPat:

```
pushReturn[];
call[cdDoGallPat], t_a0;          * background with zeros
call[cdDoGallPat], t_a1;          * background with ones
returnP[];
```

cdDoGallPat:

```
pushReturnAndT[];
call[setCache0], t_ Stack;
```

```
call[iCdVaCtrl], t_a0;          * Start va at zero
```

cdGallPatL:

```
call[nextVa];                   * Top of MAIN LOOP
skpif[alu#0];
branch[cdGallPatXit];
t_ cdMaxVa;
call[cdCheckVaRange], rscr_a0;  * cache[0..cdMaxVa) should be =stack
```

cdGallPat2:

```
t_ not(stack);
Pd_ (Store_ va)-1, DBuf_ t;
branch[cdNoChk, ALU<0];
rscr_ a0;                        * placement requires instr here
call[cdCheckVaRange], t_ (va)-1; * check the interval [0..va)
rscr_ (va)+1;
call[cdCheckVaRange], t_ cdMaxVa; * check the interval (va..cdMaxVa)
noop;                             * for placement.
```

cdNoChk:

```
Fetch_va;
t_ not(stack);
t#(Md);
skpif[ALU=0], t_ (va)+1;
```

cdGallPErr2:

```
error;                            * can't find the value (not(stack)) we
```

```
error;                            * just wrote!
```

cdGall3:

```
t_va;
Store_t, DBuf_ stack;            * restore it to original value
```

```
t_SUB[cdMaxVa!, 1]C;
```

cdGall4L:

```
Fetch_t;
(md)#(stack);
skpif[ALU=0], t_ t-1;
```

cdGallErr3:

```
error;                            * storing Stack into va clobbered location
branch[cdGall4L, ALU>=0];        * at t+1
```

```
branch[cdGallPatL];
```

cdGallPatXit:

```
pReturnP[];
```

* May 26, 1981 10:11 AM

```

cdCheckVaRange: subroutine;                                * Enter w/ t=lastva+1, rscr=1st va, stack=expected
value.                                                         *
                                                                * Keep expected value in Q
                                                                * Save return link
    Q_ Stack&+1;
    Stack_ link;
    top level;
    (rscr)-t;
    branch[.+2, alu<0];
    branch[cdCheckVaXit], link_ stack&-1;
    noop;                                                       * for placement

cdChkVaRangeL:
    rscr_ (Fetch_rscr)+1;                                       * fetch current value and increment va
    (Md)#(Q);
    skipif[alu=0], (rscr)#t;

cdChkErr:
    error;                                                       * cache[RSCR+1] is not same as Q
                                                                * see return link on STACK to find
*where this problem has occurred. This subroutine is called from cdGallpat.

    loopWhile[ALU#0, cdChkVaRangeL];
    subroutine;
    link_ stack&-1;
cdCheckVaXit:
    return;

```

```

%*****
May 19, 1981 11:23 AM
  Change addressing test to take a parameter=value to background memory with.
  Then change the test to run once w/ 0, once w/ 177777B.
February 6, 1981 10:34 AM
  Change main r/w test to use prefetch.
September 19, 1979 10:39 PM
  Cause sFlushTest, sMiscTest to do _MD to wait for any activity to finish that should finish before
  manipulating the cache flags.
September 13, 1979 4:52 PM
  Make beginStest into a subroutine that invokes all the other storage board tests.
September 7, 1979 6:23 PM
  Change, again, the order of the tests so that the sDtest happens before the addressing test.
June 28, 1979 8:36 AM
  Fix bug in svarpipe2resume -- resumes at wrong label -- change name to sVaPipeResume, too.
June 27, 1979 6:52 PM
  Fix bug in sDbufMdTest (wrong sex on skip), make sLongFetch reset BRs when done.
June 27, 1979 11:46 AM
  Fix placement problem with afterStest, iLongFetchMem.
June 27, 1979 11:23 AM
  Add sLongFetchTest.
June 25, 1979 3:02 PM
  Make each test in memRWS into a subroutine; add sMiscTest.
June 8, 1979 10:12 AM
  Save and restore random number generator seed in sDataTest -- to aid patching the code.
April 18, 1979 3:18 PM
  Change branch at end of sAddrTest to branch to sDtest rather than sChaosTest (this caused memA
  to skip the storage data test!).
April 18, 1979 9:36 AM
  Remove conditionalTasks from Flush test because it reads CFlags and this leaves memory in state
  that can't handle references from arbitrary task wakeups.
April 17, 1979 10:41 PM
  Bracket the flush test w/ calls to enableConditionalTask, disableConditionalTask.
April 12, 1979 8:53 AM
  Move the storage addressing test to be first in the sequence of storage tests.
April 11, 1979 3:47 PM
  Added breakpoint at end of read/check loop so operator can tell when the diagnostic has made one
  complete pass over the memory storage.
January 16, 1979 3:55 PM
  Invoke sRestoreMcrVictim at end of sFlushTest -- to allow entire cache to be used if it is
  enabled.
January 15, 1979 5:43 PM
  fix sFlushTest to guarantee no hits in a column not selected by sMCRvictim
January 15, 1979 10:55 AM
  more comments about patching, force microD to cause instrs to occur sequentially in IM.
January 12, 1979 11:55 PM
  comments that describe how to avoid conditional tasking during sDtest's write loop
January 11, 1979 9:15 AM
  add Flush_test, fix missing "sVa_t" in addressing test's find error sections, add flush_ to
  addressing test.
%*****

```

```

title[memrws];
top level;
* September 7, 1979 6:23 PM
beginStest:
  pushReturn[];
  checkMtest[memFlags.sBoard, doneStest]; * skip everything if required
  call[disableConditionalTask]; * prevent task simulator from running
  call[iSboard]; * find out mem configuration, size, etc

  call[sDtest]; * data test
  call[sAddrTest]; * addressing test
  call[sFlushTest]; * flush_test
  call[sMiscTest]; * miscellaneous tests
  call[sChaosTest]; * chaos (random operations) test
doneStest:
  returnP[]; * done
%
```

A FEW RULES

Subroutines clobber rscr, rscr2 and T unless otherwise specified at both point of call and in

subroutine description. Subroutines return single values in T, and they accept single values in T. Two parameter subroutines accept their values in rscr and rscr2.
Global values for S board

Abbreviations used herein

i	Init
M	Map
col	Column
S	Storage
nextFoo	increments foo loop variable, returns with fast branch
condition to check for done w/ loop	
getFoo	returns current value of loop variable, foo
getFooRtn	subroutine that returns in T the saved value of foo's
return link	
iFooCtrl	initialize foo loop control

%

* May 19, 1981 11:20 AM

Storage Addressing Test

NOTE: The Addressing test has been rewritten so that it runs once backgrounding memory w/ 0. Then it runs again and backgrounds memory w/ 177777B. The test was changed to accommodate this new subroutine parameter (the background value). Otherwise, the algorithms are the same.

Background memory w/ 0 then write -1 into ascending addresses. Before the write, check that the word that is about to be written is still zero. Suppose it is non zero. Then there was an addressing error. Remember the address of the non zero word, background memory with zeros again and proceed writing -1s. This time, check the previously clobbered address each time before writing the -1. This approach will catch the reference that clobbers the known location. The same algorithm can be applied for descending addresses, mutatis mutandi, to finish a complete check of the addressing logic.

sAddrCheck: PROCEDURE=

BEGIN

zeroMem: PROCEDURE

BEGIN

FOR munch _ 0, munch _ munch+20B UNTIL maxMunch DO

FOR i IN [0..20B) DO

mem[munch+i] _ 0; -- microcode implementation will be fast!

ENDLOOP:

ENDLOOP:

END;

findErrUp: PROCEDURE [clobbered: VA] =

BEGIN

zeroMem[];

FOR i IN VA DO

IF mem[clobbered]#0 THEN SIGNAL ErrUp[i-1, clobbered];

mem[i] _ -1;

ENDLOOP:

SIGNAL IntermittentErrUp[clobbered];

END;

findErrDown: PROCEDURE [clobbered: VA] =

BEGIN

zeroMem[];

FOR i DECREASING IN VA DO

IF mem[clobbered] #0 THEN SIGNAL ErrDown[i+1, clobbered];

mem[i] _ -1;

ENDLOOP;

SIGNAL IntermittentErrDown[clobbered];

END;

-- this is the program

zeroMem[];

FOR i IN VA DO

IF mem[i] # 0 THEN FindErrUp[i];

mem[i] _ -1;

ENDLOOP;

zeroMem[];

FOR i DECREASING IN VA DO

IF mem[i] # 0 THEN FindErrDown[i];

mem[i] _ -1;

ENDLOOP;

END;

* May 19, 1981 1:00 PM

sAddrTest:

```
pushReturn[];
checkMtest[memFlags.sAddr, sAddrTestDone];
t_a0; * for MicroD
call[sDoAddressTest], t_a0; * address test w/ background = 0
t_al; * for MicroD
call[sDoAddressTest], t_al; * address test w/ background = 1
```

sAddrTestDone:

```
pReturnP[];
```

sDoAddressTest: * This address test backgrounds memory with the value we were passed in T. Keep that value on the stack.

```
pushReturnAndT[];
call[enableConditionalTask];
call[setMemory], t_stack;
call[iSvaCtrl];
```

sAddrUpL:

```
call[nextSva];
branch[sAddrLxit, alu=0], Sva _ t;

fetch _ t, t_stack;
rscr _ md;
(rscr) # t; * is memory same value as we stored there?
skipif[alu=0], t _ not(stack);
branch[sAddrUpFindErr];

branch[sAddrUpL], DBuf _ t, store _ Sva;
```

sAddrLxit:

```
noop;
call[setMemory], t_stack;
call[iSvaDownCtrl];
```

sAddrDownL:

```
call[nextSvaDown];
branch[sDoAddrTestDone, alu=0], sva _ t;

fetch _ t, t_stack;
rscr _ md;
(rscr)#t; * is memory same value as we stored there?
skipif[alu=0], t _ not(stack);
branch[sAddrDownFindErr];

branch[sAddrDownL], DBuf _ t, store _ Sva;
```

* May 19, 1981 11:19 AM

sAddrUpFindErr

REMEMBER: Stack contains the background value for this test.

This test tries to isolate the reference that clobbered the location (denote it the *target location*) that was detected by the **sAddrUpL** loop. The test proceeds as before except that as it ascends memory it continually checks to see if the *target location* has been clobbered yet. Use **Flush_** to force the target munched out of the cache.

sAddrUpFindErr:

```

call[saveMemAddr];          * save addr of clobbered location
call[setMemory], t_stack;  * returns w/ rscr = brHi, rscr2 = low 16
call[fetchMemAddr];
t _ rscr2;
call[setBR], rscr2 _ t-t;
fetch _ t, rscr3_t;
t_stack;
rscr2 _ md;
(rscr2)#t;
skpif[alu=0];              * error implies setMemory didn't work

```

sAddrUpErr1:

```

error;                      * rscr3 = low 16 bits of addr, rscr = hi 8 bits, rscr2
= MD value

```

saueFlush1:

```

Flush _ rscr3;              * remove target munched from cache

```

```

call[iSvaCtrl];

```

sAddrUpFindL:

```

call[nextSva];
branch[sAddrUpNoFind, alu=0], sVA _ t;

```

```

noop;
call[restoreBrHi];          * we clobbered it to check clobbered mem location
t _ not(stack);            * perform next reference before we check for clobbered

```

data

```

store _ Sva, DBuf _ t;

```

```

call[fetchMemAddr];        * returns rscr= brhi, rscr2 = low 16 bits of va
t _ rscr2;

```

```

call[setBR], rscr2 _ t-t;  * save low 16 bits of addr in rscr3
fetch _ t, rscr3_t;
t _ md;

```

```

(t)#(stack);
skpif[alu=0];              * error => last reference clobbered the location at

```

sAddrUpErr2:

```

error;                      * rscr,,rscr3. bad bits are in rscr2. Last reference

```

at sVaHi,,sVaX

saufeFlush2:

```

Flush _ rscr3;              * remove target munched from cache

```

```

branch[sAddrUpFindL];

```

sAddrUpNoFind:

```

error;                      * intermittent error;

```

* May 19, 1981 11:18 AM

sAddrDownFindErr

REMEMBER: Stack contains the background value for this test.

This test tries to isolate the reference that clobbered the location (denote it the *target location*) that was detected by the **sAddrDownL** loop. The test proceeds as before except that as it descends memory it continually checks to see if the *target location* has been clobbered yet. Use **Flush_** to force the target munch out of the cache.

sAddrDownFindErr:

```

    call[saveMemAddr];           * save address of clobbered location
    call[setMemory], t_ stack;
    call[fetchMemAddr];         * return w/ rscr = brhi, rscr2 = low 16 bits
    t_ rscr2;
    call[setBR], rscr2 _ t-t;    * make sure target word is zero
    fetch _ t, rscr3_t;
    t_ md;
    (t)#(stack);
    skipif[alu=0];              * failure implies setMemory didn't work

```

sAddrDownErr1:

```

    error;                       * rscr = brHI, rscr3 = low 16 bits of va, rscr2 = md

```

sAdfeFlush1:

```

    Flush _ rscr3;               * remove target munch from cache

```

```

    call[iSvaDownCtrl];

```

sAddrDownFindL:

```

    rscr2 _ Sva;
    call[nextSvaDown];
    branch[sAddrDownNoFind, alu=0], sVA _ t;

```

```

    noop;
    call[restoreBrHi];           * we clobbered it to check clobbered mem location
    t_ not(stack);
    store_sva, DBuf _ t;

```

```

    call[fetchMemAddr];         * returns w/ rscr = brhi, rscr2 = low a6 bits
    t_ rscr2;
    call[setBR], rscr2 _ t-t;
    fetch _ t, rscr3_t;
    t_ md;
    (t)#(stack);
    skipif[alu=0];

```

sAddrDownErr2:

```

    error;                       * ref at sVaHi,,sVaX clobbered rscr,,rscr3

```

sAdfeFlush2:

```

    Flush _ rscr3;               * remove target munch from cache

```

```

    branch[sAddrDownFindL];

```

sAddrDownNoFind:

```

    error;                       * intermittent error;

```

sDoAddrTestDone:

```

    call[disableConditionalTask];
    pReturnP[];

```

* February 6, 1981 10:52 AM

Storage Data Test

```

FOR patX IN PatX DO
  FOR va IN VA DO
    mem[va] _ getPattern[patX, va];
  ENLOOP:
  FOR munch IN [MunchVa] DO
    FOR i IN [0..17] DO
      va _ munch+va;
      got _ mem[va];
      expect _ getPattern[patX, va];
      IF ~sChkNoErrs[] THEN      -- sChkNoErrs turns OFF checking
        T _ expect xor got;
        IF T # 0 THEN SIGNAL memErr[va, expect, got];
      ENLOOP;
      -- check the pipe after we've read the entire munch
      IF ~ sChkNoErrs[] THEN      -- sChkNoErrs turns OFF checking
        IF aChkPipeFlt[] THEN
          BEGIN;
          myPipe4 _ PIPE4[];
          numlBits _ CountlBits[myPipe4.syndrome];
          IF numlBits = 1 THEN SIGNAL checkBitFailure[myPipe4];
          IF myPipe4.syndrome = 0 AND (numlBits AND 1) = 0 THEN
            SIGNAL doubleError[myPipe4];
          synWdX _ myPipe4.syndromeWdX;
          IF (synWdX=3) OR (synWdX=5) OR (synWdX=6) THEN
            SIGNAL singleError[myPipe4];
          SIGNAL unknownError[myPipe2];
          END;
        ENLOOP;
      ENLOOP;
    ENLOOP;
  ENLOOP;

```

%
%

sNoErrsOn() is a midas subroutine that will cause the sdTest to avoid ALL error checking

sNoErrsOff() is a midas subroutine that reenables sdTest error checking

sdNoDataErrs is the location of the branch that skips error checking within the data check of sdTest. Patch out this branch to allow data checking and to disallow pipe checking when sNoErrsOn is in effect. Notice the note below.

sdNoPipeErrs is the location of a branch that skips error checking of the pipe within the read loop of the sdTest. Patch out this branch to allow pipe error checking when sNoErrsON is in effect. Notice the note above.

sdOffCTask is the location of a call to enable conditional task that occurs just before the write loop portion of sdTest. Patch out this call to disallow the memory task simulator's memory references during the sdTest. Notice the note below.

sdOnCtTask is the location of a noop that may be patched into a call on the **enableConditionalTask** subroutine to cause the memory task simulator to run during the checking portion of the sdTest. Notice the note above.

sVaTryNextPat is the location of the branch that causes the diagnostic to write a new pattern into memory after the current pattern has been checked. Patch this location to into a **branch[sdBeginCheck]** to force the diagnostic into an infinite loop that always checks the current pattern.

sdtest:

```

pushReturn[];
checkMtest[memFlags.sRW, sdTestDone];

```

sdOffCTask: * Comment-OUT this instruction to avoid conditional tasking during
call[enableConditionalTask]; * the sdTest write loop.

```

call[iSPatCtrl];

```

sdpatL:

```

call[nextSPat];          * top of pattern loop
skpif[alu#0];
branch[sDtestDone];
noop;

```

```
    call[saveRandState];
    call[iSmunchCtrl];
sVaWMunchL:                                * top of write loop; get here once/munch.
    call[nextSmunch];
    skipif[alu#0], Sva _ t;
    branch[sVaWxit];                          * no more vas. Now Read it

    t_ t+(20C);
    rscr_ (PreFetch_ t)-t;                    * speed-up the test; rscr_0
    cnt_ 17s;

sVaWL:                                     * For this munch: mem[Sva]_curPattern
    noop;                                     * for placement.
    call[getSPat], t _ Sva;
    loopuntil[cnt=0&-1, sVaWL], DBuf _ t, Sva_(Store _ Sva)+1;
    branch[sVaWMunchL];
sVaWxit:
```

* February 6, 1981 10:51 AM

```

sdOnCTask:
    call[justReturn];
    * Patch this location into a call on
    * enableConditionalTask, if desired.

sdBeginCheck:
    call[iSmunchCtrl];
    call[restoreRandState];
    B _ FaultInfo'[];

sVaRmunchL:
    call[nextSmunch];
    skipif[alu#0], Sva _ t;
    * top of munch loop

sVaTryNextPat:
    branch[sDpatL], breakpoint;

    t_t+(20C);
    rscr_ (PreFetch_ t)-t;
    cnt _ 17s;
    * speed-up the test; rscr_0
    * inner loop to zero current munch
    * top of word read/check loop

sVaRL:
    FETCH _ sva;
    call[getSPat], t _ Sva;
    sExpected _ t;
    rscr2 _ MD;
    * copy current pattern into sExpected
    * (avoid bypass)

    call[sChkNoErrs];
    skipif[ALU=0];

sdNoDataErrs: * Patch out this branch to allow data error checking during sdTest when
    branch[sVaResume];
    * sNoErrsOn is in effect.

    t _ sExpected;
    t _ t # (rscr2);
    skipif[alu=0];
    * retrieve sExpected into t
    * t _ cur t xor md
    * error => bad data from memory. Sva = addr
    * sExpected = expected value
    * t = bad bits, rscr2 = MD

sVaRErr:
    error;

sVaRResume:
    noop;
    loopuntil[cnt=0&-1, svaRL], sva _ (sva) +1;
    * placement for microD

* now check pipe 2
    noop;
    call[sChkNoErrs];
    skipif[ALU=0];
    * placement for microD

sdNoPipeErrs: * Patch out this branch to allow pipe error checking in sdTest when
    branch[sVaRmunchL];
    * sNoErrsOn is in effect.
    noop;

    call[aChkPipeFlt];
    skipif[alu#0];
    branch[sVaRmunchL];
    * no errors; try next munch

    * Pipe checking code on next page.

```


* June 28, 1979 8:38 AM

%*****

nFaults#7 This means that a check bit failed or there was a double error or there's some unknown thing occurring. "Strang thing" = not check error and not double error.

NOTE: sva points to one beyond the "current" munch since we incremented it at the same time we performed the loop control test. First we decrement Sva.

%*****

```

sva _ (sva)-1;          * now sva points to last word in current munch
call[xGetPipe4];
rscr2 _ t;
cnt _ 7s;              * count the number of '1' bits in syndrome.
t _ t-t;

```

sVaNsynBitsL:

```

skipif[r even], rscr2;
t _ t+1;
rscr2 _ rsh[rscr2, 1];
loopUntil[cnt=0&-1, sVaNsynBitsL];

```

```

t-(1c);
skipif[ALU#0], rscr _ t;

```

sVaChkBitErr:

```

error;
noop;
call[xGetPipe4];

```

```

skipif[r even], rscr;
branch[sVaChkSingleErr];
rscr _ (t) AND (pipe4.syndrome);
skipif[alu#0];
branch[sVaChkSingleErr];

```

sVaDblErr:

```

error;

```

sVaChkSingleErr:

```

t _ t AND (pipe4.syndromeWdX);
noop;
t _ rsh[t, pipe4.syndromeWdXShift];
t-(3c);
skipif[ALU#0];
branch[sVaSingleErr];
t - (5c);
skipif[ALU#0];
branch[sVaSingleErr];
t - (6c);
skipif[ALU#0];
branch[sVaSingleErr];
t - (7c);
skipif[ALU=0];
branch[sVaUnknown];

```

sVaSingleErr:

```

error;

```

sVaUnknown:

```

error;

```

sVaPipeResume:

```

branch[sVaRmunchL];

```

sDtestDone:

```

call[disableConditionalTask];
returnP[];

```

* move numlBits into rscr for even/odd check
* numlBits ==>check bit error. See midas for
* PERRS 20, PVA 20, etc
* for placement

* double errs only w/ even parity

* check for nonzero syndrome

* syndrome # 0 and syndrome parity even
* means there was a double error

* put encoded word index in t

* corresponds to word zero

* error in word zero of quadword

* corresponds to word one

* error in word one of quadword

* corresponds to word two

* error in word two of quadword

* corresponds to word three

* double error. see PVA 20, PERRS 20
* using midas. t = encoded word index

* diagnostic is confused. Use Midas to
* examine the pipe to see what happened.

* April 17, 1979 10:41 PM

Flush_

Test the Flush_ function of the memory system. Read the cache to determine if Flush_ really worked. The test forces the memory system to use only one column during the checking phase; this makes it easier to determine what is happening. This test disables conditional tasking because it reads CFLAGS.

For each column:

```

    Set CacheA to all zeros (works on all columns)
    Set All CFLAGS to vacant (works on all columns)
    Zero the memory (works on current column, only)
    For each munch (works on current column, only)
        Dirty the munch
        Flush that va
        Read the cache for the appropriate row to make sure the munch is vacant
        Reread the munch to assure the dirty data is really there

```

sFlushTest:

```

    pushReturn[];
    checkMtest[memFlags.sFlush, afterSflush];
    call[disableConditionalTask];
    call[initCol2Ctrl];

```

* manipulating subroutines we call clobber "col".

sFlushColl:

```

    call[nextCol2];
    skipif[ALU#0];
    branch[sFlushColXit];
    T_MD;

    call[ZeroCacheAndFlags];
    call[clearCacheFlags];

```

```

    call[getCol2];
    sMCRvictim _ t;
    call[sUseDefaultMcr];

```

```

    call[zeroMemory];

```

```

    call[iSmunchCtrl];

```

sFlushMunchL:

```

    call[nextSmunch];
    skipif[ALU#0], sVa _ t;
    branch[sFlushMunchXit];
    noop;

```

* readCflags, which we call below, clobbers Mcr, so we must reset Mcr each time thru the loop.

```

    t_cml;
    STORE_sVa, DBuf _ t;
    B_MD;
    Flush_sVa;

```

```

    t _ 100c;
    call[longWait];
    rscr _ sVa;
    call[getCol2];
    call[readCflags], rscr2 _ t;

```

```

    t _ t and (pipe5.flagsMask);
    rscr _ t # (cflags.vacant);
    skipif[ALU=0];

```

sFlushErr1:

```

    error;

```

```

    call[sUseDefaultMcr];
    FETCH _ sVa;
    t _ MD;
    sExpected _ cml;
    t # (sExpected);
    skipif[ALU=0];

```

sFlushErr2:

```

    error;
    * sExpected = value we expected

```

```
branch[sFlushMunchL];  
  
sFlushMunchXit:  
  branch[sFlushColL];  
sFlushColXit:  
afterSflush:  
  call[disableConditionalTask];  
  call[sRestoreMcrVictim];  
  returnP[];
```

* September 19, 1979 10:41 PM

sMiscTest

Miscellaneous memory tests:
 make sure Dbuf_ doesn't clobber MD
 test the bits of MD*

sMiscTest:

```

pushReturn[];
T_MD;                * force Hold, if required.
call[sDbufMdTest];
noop;                * for placement
call[sLongFetchTest];
returnP[];

```

* June 27, 1979 6:56 PM

Check that MD is valid after writing into Dbuf, and check that we can read the value stored into DBuf. The approach is to write one known value into memory then to fetch a different, known value from memory. Then check that MD and DBuf are as expected.

sDbufMdTest:

```

pushReturn[];
t _ a0;
store _ t, DBuf _ t;    * initialize location 0 w/ 0
fetch _ t;
rscr _ MD;              * fill MD with 0s
rscr2 _ cml;
store _ t, DBuf _ rscr2; * fill DBuf with -1
rscr2 _ MD;
PD _ (rscr2);
skpif[ALU=0];          * see if storing into dbuf seems

```

sDbufMdErr0: * to clobber MD.
 error; * Expected 0 in MD, got rscr2.

```

t _ DBuf;
t # (cml);
skpif[ALU=0];

```

sDbufMdErr1: * expected -1 in Dbuf,
 error; * t = value read from DBuf.

```

t _ a0;
fetch _ t;
rscr _ md;
Store _ t, DBuf _ 0c;  * this should fetch the -1 we stored earlier
rscr2 _ MD;            * Write 0 into Dbuf
(rscr2) # (cml);
skpif[ALU=0];

```

sDbufMdErr2: * see if storing all zeros into DBuf
 error; * clobbered the bits of MD
 * expected all one bits, got rscr2.

```

t _ PD _ DBuf;
skpif[ALU=0];

```

sDbufMdErr3: * expected 0 in Dbuf,
 error; * t = value read from DBuf.
 returnP[];

* June 27, 1979 11:39 AM

sLongFetchTest:

```

pushReturn[];
%*****
Test long fetch: (B[4:15],Mar[0:15]) + BR = actual 28 bit va used by memory system.
We'll initialize the 1st 65 K words so that mem[va] = va. Then set BrHi to various values and perform
long fetches to cause the values to wrap around into the 1st 65K. Check both the data and pipeVa.
%*****
    rscr _ a0;
    call[setBR], rscr2 _ a0;
    call[iSlongFetchMem];
    rscr _ (r0)+1;
    call[setBr], rscr2 _ a0;
    t _ 7777C;
    LongFetch _ rscr2, B _ t;
    rscr _ MD;
    PD _ rscr;
    skipif[ALU=0];
sLongFetchErr0:
    error;
    t _ pipe0;
    t _ t and (7777C);
    skipif[ALU=0];
sLongFetchErr1:
    error;
    t _ PD _ pipe1;
    skipif[ALU=0];
sLongFetchErr2:
    error;
    rscr _ 2c;
    call[setBR], rscr2 _ a0;
    t _ (and[7776,177400]C);
    t _ t or (and[7776,377]C);
    rscr _ 34c;
    LongFetch _ rscr, B _ t;
    rscr2 _ MD;
    (rscr2) # (34c);
    skipif[ALU=0];
sLongFetchErr3:
    error;
    t _ pipe0;
    t _ t and (7777C);
    skipif[ALU=0];
sLongFetchErr4:
    error;
    t _ pipe1;
    t # (34C);
    skipif[ALU=0];
sLongFetchErr5:
    error;
    rscr _ a0;
    call[setBR], rscr2 _ a0;
    returnP[];

iSlongFetchMem:
pushReturn[];
t _ rscr _ a0;
iSlongFetchMemL:
cnt _ 17s;
prefetch _ rscr;
loopUntil[CNT=0&-1, .], t _ (Store_t)+1, DBuf _ t;
loopUntil[ALU=0, iSlongFetchMemL], rscr _ t + (40c);
returnP[];

```

* t = long fetch bits
* va = 7777,,0 + 1,,0 = 0

* should have wrapped around to 0
* pipe 0 shows va, rscr = MD, expected 0.

* VA high bits should be zero
* t = actual value of bits

* VA low bits should be zero
* t = actual bits

* BR = 2,,0

* t = 7776 = long fetch bits
* arbitrary word to fetch in 1st 65K
* va = 7776,,34 + 2,,0 = 0

* Expected to pick up word at location 34
* Expected value = 34B, got rscr2

* Br hi bits should be zero!
* got value in t

* expected BrLo bits = 34B
* got value in t.

```
title[mapReadWriteX];
%*****
                TABLE of Contents, by Order of Occurence
beginXtest          main subroutine that calls the tests in this file
mapSingleStep     single step code for midas debugging
mapRWtest         Read and write the map
mapAddrTest       Map addressing test
mapRWtest2        Map read write test that checks timing margins -- does not execute unless
operator patches code. It takes more than 20 minutes to execute.
%*****

%
May 14, 1981  4:49 PM
    skip xboard tests if memFlags.xBoard not true (avoid problemes w/ xGetMapIcs)
September 19, 1979  7:16 PM
    Move beginXtest to beginning of listing, add call to disableConditionalTask.
September 14, 1979  9:54 AM
    Add table of contents, fix typo in beginXtest.
September 13, 1979  4:41 PM
    add code that calls each x-board test as a subroutine.
January 25, 1979  4:52 PM
    Add check for pipe2.nfaults =7.
January 25, 1979  3:41 PM
    Add checking to see if pipe4.mFault is set dukring mapRWtest.
%
```

```

* May 14, 1981 2:05 PM
beginXtest: subroutine;
  pushReturn[];
  call[disableConditionalTask];
  checkMtest[memFlags.xBoard, endXtest];
  call[xGetMapICs];           * init xChipRasCas, etc.
  call[mapRWtest];
  call[mapAddrTest];
  call[mapRWtest2];          % this test doesn't really do anything unless operator
                             patches the code at the entry to this subroutine. This test checks the timing margins of the map
                             chips and requires about 20 minutes or more to execute.
  %
endXTest:
  returnP[];

top level;
* December 20, 1977 3:16 PM
%
```

A FEW RULES

```

Mrow IN [0..mRowEndC)
Mcol IN [0..mColEndC)
```

Subroutines clobber rscr, rscr2 and T unless otherwise specified at both point of call and in subroutine description. Subroutines return single values in T, and they accept single values in T. Two parameter subroutines accept their values in rscr and rscr2. Subroutines may use the global variables Mrow, Mcol, etc.

Global values for X board

Mrow	map row
Mcol	map column
mcrBits	mcr bits for map manipulation
Mpat	current pattern value

Abbreviations used herein

i	Init
M	Map
X	for X board, error prefix
col	Column
nextFoo	increments foo loop variable, returns with fast branch
condition to check for done w/ loop	
getFoo	returns current value of loop variable, foo
iFooCtrl	initialize foo loop control

%

* December 14, 1978 12:04 PM

mapSinglStep:

mSSfetch:

```
fetch _ r0;
noop;
branch[.];
```

* you get what you ask for

mSSfetch2:

```
fetch _ r0;
noop;
branch[.,cnt#0&-1];
branch[mSSfetch2];
```

* September 13, 1979 4:42 PM

%

Read and Write the Map

%

```

mapRWtest:                                * read and write the map
    pushReturn[];
    checkMtest[memFlags.mRW, mapRWdone];
    call[clearCacheFlags];
    call[iMpatCtrl];
mapMainL:                                * MAIN, OUTER LOOP
    call[nextMpat];
    skipif[alu#0];
    branch[mapRWdone];
    noop;

    call[iMcolCtrl];
mapRWcolL:                                * init column control
    call[nextMcol];
    skipif[alu#0];
    branch[mapMainL];
    Mcol _ t;

    call[resetMap];
    call[iMrowCtrl];
mapRWrowL:                                * the INNER WRITE loop. most complete in < 2 ms
    call[nextMrow];
    skipif[alu#0];
    branch[mapRWchk];
    noop;
    call[getMpat];
    call[setBRforMap];

    call[xGetNumFlts];
    t # (7c);
    skipif[ALU=0];
XrwPipeFlts1Err:
    error;

    rscr _ MpatHi;
    rscr2 _ MpatLow;
    call[writeMap];

    branch[mapRWrowL];

```

%NOTE: we can't check the ref bit explicitly!
 It can't be set to one.

Test must be written to set it implicitly and then to check the results

%

* February 10, 1981 9:54 AM
 * now that we have written the map, lets check it!

```

mapRWchk:
  call[iMrowCtrl];
mapRWchkL:
  call[nextMrow];
  skipif[alu#0];
  branch[mapRWcolL];
  noop;
  call[getMpat];
  call[readMap];
  Q _ rscr2;
  t _ rscr;
  t _ t # (MpatLow);
  skipif[alu=0];
XrwPipe3Er:
  error;

  t _ ldf[rscr2, 2, pipe4.dirtyShift];
  t#(MpatHi);
  skipif[ALU=0];
XrwPipe4Er:
  error;
  * true bits inverted.

  t _ (rscr2) and (pipe4.mFault);
  skipif[ALU=0];
XrwPipeMfltErr:
  error;
  * rscr3 = value wrote, rscr2 = pipe4, MrowX, McolX = map row and column,
  * current BR shows VA used to read the map.

  call[xGetNumFlts];
  t # (7c);
  skipif[ALU=0];
XrwPipeFlt2Err:
  error;
  * T = num faults; see above error for other register info.

  branch[mapRWchkL];
mapRWDone:
  *this is a one time only check to see if reading the map might clobber the same map entry! this is
  interesting because reading is implemented by using the write logic and turning off the write signal
  at the last minute.

  Mrow_ t_ 10c;
  call[setBRforMap], Mcol_t;
  rscr_ al;
  call[writeMap], rscr2_ al;
  call[readMap];
  (rscr)#(177777C);
  skipif[alu=0];
xRWreadBad:
  error;
  call[readMap];
  (rscr)#(cml);
  skipif[alu=0];
xRWreadBad2:
  error;
  noop;
  returnP[];

```

* INNER CHECK loop
 * this subr SETS Mrow itself!

* rtn w/ rscr=pipe3, rscr2=pipe4
 * copy pipe 4 for future reference.
 * remember pipe3 in rscr for a while
 * compare pipe3 w/ expected value
 * bad bits in T, expected val in Mpat

* map value in rscr. See pipe for VA

* rt justify the dirty, wprotect bits from
 * the copy of Pipe4 that is in rscr2.
 * T= wprotect,dirty from pipe4, MpatHi
 * is value of those bits as we wrote them.
 * rscr2 contains a copy of pipe4, w/ low

* see if map fault bit is set.

* pipe4.mFault is set. Suggests map parity
 * error (tho, data read back was good)

* Pipe2.nfaults is set. the current map
 * read caused the fault to occur.

* write map w/ row=col=10B

* write it w/ all ones

* rscr= map value. should be -1

* see if last time we read map it
 * clobbered this location

* rscr= map value. should be -1

* September 13, 1979 4:43 PM

%

mapAddrTest Test the addressing mechanism of the map. This test follows the same pattern as the addressing tests for C, D, and S boards:

(Ascent Test) Zero the memory, then ascend the memory as follows:

- 1) If the current location is non zero, there was an addressing error (a previous store to an "earlier" location clobbered the "current" location). Invoke the routine that finds exactly which store that caused this error.
- 2) Otherwise, set the current location to all ones.
- 3) Increment the current address; if all addresses have been tested proceed to the descent addressing test, otherwise proceed to step 1.

(Descent test) Zero the memory, then proceed as in the ascent test, except we descend through the memory (decrement) rather than ascend (increment).

(Ascent error routine) Denote the "error address" as the clobbered address discovered in the ascent test. The ascent error routine proceeds by zeroing the memory, then ascending thru the memory as follows:

- 1) If the error address is nonzero, the last store which the ascent error routine made clobbered that location (storing into the "current address -1" clobbered the "error address" that was detected by the ascent test.
- 2) Otherwise, set the current location to all ones.
- 3) Increment the current address; if all addresses have been tested, there must have been an intermittent error, otherwise proceed to step 1.

The descent error routine is similar to the ascent error in the same way.

%

mapAddrTest:

```
pushReturn[];
checkMtest[memFlags.mAddr, xDownXit];      * see if this test enabled.
call[clearCacheFlags];                      * just incase being loaded is set
call[xZeroMap];
call[iMapPageCtrl];
```

xUpL:

```
call[nextMpage];                             * rtns w/ t = next page, fast br condition
skpif[ALU#0];                                * tells if we're done w/ loop.
branch[xUpXit];
noop;                                         * for placement

call[xReadMapPage];                           * current address should still be zero
PD _ t;
skpif[ALU=0];
branch[xErrUp];                               * a previous store clobbered current addr

call[getMpage];                              * get current addr into t again
call[xWriteMapPage], rscr _ cml;             * set current addr to all ones
branch[xUpL];
```

xUpXit:

```
call[xZeroMap];
call[iMpageDownCtrl];
```

xDownL:

```
call[nextMpageDown];                         * rtns w/ t = next page, fast br condition
skpif[ALU#0];                                * tells if we're done w/ loop.
branch[xDownXit];
noop;                                         * for placement

call[xReadMapPage];                           * current address should still be zero
PD _ t;
skpif[ALU=0];
branch[xErrDown];                            * a previous store clobbered current addr

call[getMpage];                              * get current addr into t again
call[xWriteMapPage], rscr _ cml;             * set current addr to all ones
branch[xDownL];
```

xDownXit:

```
returnP[];
top level;
```

* December 25, 1978 11:09 AM

```

xErrUp:
in the map. Now we proceed to determine whgich store clobbers that location.
    call[getMpage];
    stack&+1 _ t;
    call[xZeroMap];
    call[iMapPageCtrl];
    call[xReadMapPage], t _ r0;
    PD _ t;
    skipif[ALU=0];
xUpErr1:
xZeroMap
    error;
    noop;

xErrUpL: * Top of loop that ascends memory looking for the store that clobbers the error
    call[nextMpage];
    skipif[ALU#0];
    branch[xErrUpIntermittent];
    noop;
    call[xReadMapPage], t _ stack;
    PD _ t;
    skipif[ALU=0];
xErrUp2: * the previous map write clobbered the map entry at "stack"
    error;

    call[getMpage];
    (stack) # t;
    skipif[ALU=0];
    call[xWriteMapPage], rscr _ cml;
    * the descent test discovered that error address was clobbered by a store we've
    * already performed. Sigh. Keep going to see what happens.
    noop;
    branch[xErrUpL];
xErrUpIntermittent: * Can't find error ascent test found (entry at "stack" gets clobbered).
    error;

xErrDown:
in the map. Now we proceed to determine which store clobbers that location.
    call[getMpage];
    stack&+1 _ t;
    call[xZeroMap];
    call[iMPageDownCtrl];

xErrDownL:
    call[nextMpageDown];
    skipif[ALU#0];
    branch[xErrDownIntermittent];
    noop;

    call[xReadMapPage], t _ stack;
    PD _ t;
    skipif[ALU=0];
xDownErr2: * The previous write (MpageX+1) clobbered the error address (stack).
    error;

    call[getMpage];
    t # (stack);
    skipif[ALU=0];
intermittent failue since
    * the descent test discovered that error address was clobbered by a store we've
    * already performed. Sigh.
    call[xWriteMapPage], rscr _ cml;
    branch[xErrDownL];

xErrDownIntermittent: * Can't find error descent test found (entry at "stack" clobbered).
    error;

```

* The ascent test has discovered a clobbered location

* stack = map page that got clobbered.

* clear the map

* loop init

* make sure we succeeded in zeroing the

* first location of the map

* first entry in map is non zero. this suggests that

* didn't work

* for placement

* location (held in stack).

* intermittent error if no more map pages

* for placement

* see if "error location" clobbered yet.

* set "current location" to all ones, except

* don't write into the error address.

* Skip if an intermittent failue since

* (WRITE map)

* for placement

* Can't find error ascent test found (entry at "stack" gets clobbered).

* The descent test has discovered a clobbered location

* stack = map page that clobbered

* top of loop that descends the map memory

* intermittent error if no more map pages

* for placement

* see if "error location" clobbered yet.

* don't write into the error address.

* if we've come this far, there must have been an

* WRITE map

* Can't find error descent test found (entry at "stack" clobbered).

* December 28, 1977 8:59 AM
%

```

                                Test MAP TIMING ATTRIBUTES
FOR cycles _0,cycles _ cycles+20000B UNTIL cycles >=60000B DO
  FOR patx IN PatX DO
    FOR hi2BRbits IN[0..3] DO
      FOR row IN Row DO
        FOR col IN Column DO
          pat _ getPattern[patX, row, col];
          testMap[row,col,hi2BRbits,pat, cycles];
        ENDLLOOP;
      ENDLLOOP;
    ENDLLOOP;
  ENDLLOOP;
ENDLLOOP;

```

Note: This test checks every column for a particular row before it proceeds to next row.

%
* December 14, 1978 12:04 PM

```

mapRWtest2:
  pushReturn[];
  branch[map2RWdone]; * must patch this test explicitly. It takes over 20 minutes to run.
  call[iMWaitCtrl];
  call[resetMap];
map2CyclesL: * MAIN, OUTER LOOP: Cycles to wait
  call[nextMwait];
  skipif[alu#0];
  branch[map2RWdone]; * wait loop done. try next hi order bit
  noop; * throw away result. we'll get it again.

  call[iMpatCtrl];
map2PatternL: * LOOP: Pattern Control
  call[nextMpat];
  skipif[alu#0];
  branch[map2CyclesL]; * have used all patterns. try next wait value
  noop;

  call[iMrowCtrl];
map2RWrowL: * LOOP: Map Row
  call[nextMrow]; * this subr SETS Mrow itself!
  skipif[alu#0];
  branch[map2PatternL]; * rows done. try next high order Va bit value
  noop;

  call[iMcolCtrl]; * init column control
map2RWcolL:
  call[nextMcol]; * returns w/ fast br indicating exit condition
  skipif[alu#0];
  branch[map2RWrowL]; * column loop done; try next row
  Mcol _ t;
  call[getMpat];
  call[getMwait];
  rscr _ t; * assumes rscr = cycles to wait.
  rscr2 _ r0; * dont call resetMap
  call[testMap]; * test current <row,col,pattern,wait>
  branch[map2RWcolL];
map2RWdone:
  returnP[];

```

%

June 9, 1981 11:15 AM
Add iMemState to init sTestReg from IM.

May 14, 1981 4:33 PM
add rlinkRet, etc. to reduce size of add/remove/only ??board tests

June 27, 1979 7:06 PM
Make orMemFlags, andMemFlags, and xorMemFlags global to save IM space.

May 7, 1979 4:04 PM
Fix register clobber bug in memory simulator task.

April 19, 1979 12:57 PM
Initialize memory simulator task's membase, baseregisters.

April 17, 1979 10:43 PM
Cause memSim code to notice that taskSimulation is disabled (make holdSim independent of task sim).

January 17, 1979 9:35 AM
sInterference test calls sUseDefaultMcr (saSetMcr now defunct).

January 13, 1979 12:39 PM
diddles to get mema to place

January 13, 1979 2:45 AM
comment-out "breakpoint" in midas subroutines: microD can't place memA

January 13, 1979 12:50 AM
Add comments describing how to patch out stores in memory task simulator.

%

```

title[memSubrsA];
top level;
branch[memSubrsADone];

getASubrScr: subroutine;
    RBASE _ rbase[aSubrScr];
    return, t _ aSubrScr, RBASE _ rbase[defaultRegion];
getAMakingFaults: subroutine;
    RBASE _ rbase[aMakingFaults];
    return, t _ aMakingFaults, RBASE _ rbase[defaultRegion];
rlinkRet:
    returnUsing[rlink];

```

* May 7, 1979 4:04 PM

%

Task Simulator Code for Memory Storage Diagnostic

This code runs when the Dorado task simulator awakens task 17. Its purpose is to cause multi-task memory activity during the memory storage test. The storage diagnostics continually cause misses, and this code is set up to fetch & store from a roving pointer (increment can be patched to zero). There is a delay loop to prevent the diagnostics from spending too long in hold in the task simulation code.

The place where the delay register is initialized and where the address is incremented is shown in bold. These values can be patched using Midas; changing the delay constant will modify the amount of time the memory task spends being held by its fetches, and changing the address increment may modify the row selected by the memory task during its fetches. Note that only one row is touched by the memory task with an increment of cdMaxVa.

simScr0 The address that we fetch

simScr1 The data obtained from MD when this task woke up

simScr2 Used as a scratch register to remember the current hold vaule, also loaded with a loop constant for the memSimL1 loop. The memSimL1 loop prevents task 17 from using all the machine by doing references that always hold. The smaller the constant (see memSimSetDelay), the more frequently task 17 makes memory references when it runs.

memSimStore The address to patch into a noop to remove STOREs from the memory task simulator.

memSimSetDelay The address to patch to change the delay constant for the task simulator.

memSimIncAddr The address to patch to change the increment applied to the current address.

%

```

top level;
set[xtask, 1];                                * non emulator code
memSimInit:
t _ q;                                         at[memSimInitLoc];
hold&tasksim _ t;                             * zero hold if that is what is happening
t _ t and (sim.taskMask);                     * do nothing if not tasking (allow hold sim)
skpif[alu#0];                                 * see if we are really supposed to run
block;                                         * no, just turning off hold
RBASE _ rbase[defaultRegion];
call[setMbase],t_12C;                         * set memBase, BR for this task
t _ rscr _ t-t;
call[setBR], rscr2_t;
RBASE _ rbase[simScr0];
t _ q;                                         * restore t = holdValue
branch[memSimFetchNextTime], FETCH _ simScr0;

memSimL1:                                    * top of delay loop
simScr2 _ (simScr2) - 1;                       * we'll make memory references only after
skpif[alu>=0];                                 * waiting a while. This enables the relatively
branch[memSimFetchNextTime];                  * slow memory diagnostics to make progress.
hold&tasksim _ t;                             * otherwise, most real time is spent w/ memory
noop;                                         * task being held.
branch[memSimL1], block;                       * Still waiting.

memSimFetchNextTime:
hold&tasksim _ t;
fetch _ simScr0;                              * remember 1st instr. after hold_ doesn't count
block;

memSimFetchMD:
simScr1 _ MD;                                  * fetch the memory data waiting for us
simScr2 _ t;                                  * save t, our hold value, for future use

FETCH _ simScr0;                              * fetch it again in preparation for storing it
t _ MD;                                       * get data for our store

memSimStore:
STORE _ simScr0, DBuf _ t;                    * store the data we just read

t _ simScr2;                                  * restore hold value in T

memSimSetDelay:
simScr2 _ 25c;                             * reset the delay counter

hold&tasksim _ t;                             * remember 1st instr. after hold_ doesn't count
noop;

memSimIncAddr:
branch[memSimL1], simScr0 _ (simScr0) + (cdMaxVa), block; * selects

```



```
knowRbase[defaultRegion];  
set[xtask, 0];
```

* a different addr without changing the row!

* emulator code

* May 14, 1981 4:29 PM

orMemFlagsAndExit:

```
call[orMemFlags];
branch[rlinkRet];
```

andMemFlagsAndExit:

```
call[andMemFlags];
branch[rlinkRet];
```

addCboardTest: subroutine;

```
saveReturn[rlink];
branch[orMemFlagsAndExit], t _ memFlags.Cboard;
```

removeCboardTest: subroutine;

```
saveReturn[rlink];
branch[andMemFlagsAndExit], t _ not(memFlags.Cboard);
```

onlyCboardTest: subroutine;

```
saveReturn[rlink];
call[andMemFlags], t _ t - t;          * remove all other tests
branch[orMemFlagsAndExit], t _ memFlags.Cboard; * add Cboard test
```

addXboardTest: subroutine;

```
saveReturn[rlink];
branch[orMemFlagsAndExit], t _ memFlags.Xboard;
```

removeXboardTest: subroutine;

```
saveReturn[rlink];
branch[andMemFlagsAndExit], t _ not(memFlags.Xboard);
```

onlyXboardTest: subroutine;

```
saveReturn[rlink];
call[andMemFlags], t _ t - t;          * remove all other tests
branch[orMemFlagsAndExit], t _ memFlags.Xboard; * add Xboard test
```

addDboardTest: subroutine;

```
saveReturn[rlink];
branch[orMemFlagsAndExit], t _ memFlags.Dboard;
```

removeDboardTest: subroutine;

```
saveReturn[rlink];
branch[andMemFlagsAndExit], t _ not(memFlags.Dboard);
```

onlyDboardTest: subroutine;

```
saveReturn[rlink];
call[andMemFlags], t _ t - t;          * remove all other tests
branch[orMemFlagsAndExit], t _ memFlags.Dboard; * add Dboard test
```

addSboardTest: subroutine;

```
saveReturn[rlink];
branch[orMemFlagsAndExit], t _ memFlags.Sboard;
```

removeSboardTest: subroutine;

```
saveReturn[rlink];
branch[andMemFlagsAndExit], t _ not(memFlags.Sboard);
```

onlySboardTest: subroutine;

```
saveReturn[rlink];
call[andMemFlags], t _ t - t;          * remove all other tests
branch[orMemFlagsAndExit], t _ memFlags.Sboard; * add Sboard test
```

addAboardTest: subroutine;

```
saveReturn[rlink];
branch[orMemFlagsAndExit], t _ memFlags.Aboard;
```

removeAboardTest: subroutine;

```
saveReturn[rlink];
branch[andMemFlagsAndExit], t _ not(memFlags.Aboard);
```

onlyAboardTest: subroutine;

```
saveReturn[rlink];
call[andMemFlags], t _ t - t;          * remove all other tests
branch[orMemFlagsAndExit], t _ memFlags.Aboard; * add Aboard test
```

addAllTests: subroutine;

```
saveReturn[rlink];
t _ memFlags.default0;
t _ t + (memFlags.default1);
branch[orMemFlagsAndExit];
```

removeAllTests: subroutine;

```
saveReturn[rlink];
```

```

branch[andMemFlagsAndExit], t _ t-t;

orMemFlags: subroutine;                                * save the bits to xor
  sSubrScr _ t, global;
  t _ link;
  top level;
  Srlink _ t;
  t _ memFlagsLocC;                                    * get current flag bits
  call[getIMRH];
  call[getsSubrScr], rscr _ t;                          * re-fetch bits to xor
  rscr _ t OR (rscr);                                    * xor the bits
  t _ memFlagsLocC;
  call[putIMRH];                                        * save new bit values
  returnUsing[Srlink];

andMemFlags: subroutine;                                * save the bits to xor
  sSubrScr _ t, global;
  t _ link;
  top level;
  Srlink _ t;
  t _ memFlagsLocC;                                    * get current flag bits
  call[getIMRH];
  call[getsSubrScr], rscr _ t;                          * re-fetch bits to xor
  rscr _ t AND (rscr);                                    * xor the bits
  t _ memFlagsLocC;
  call[putIMRH];                                        * save new bit values
  returnUsing[Srlink];

xorMemFlags: subroutine;                                * save the bits to xor
  sSubrScr _ t, global;
  t _ link;
  top level;
  Srlink _ t;
  t _ memFlagsLocC;                                    * get current flag bits
  call[getIMRH];
  call[getsSubrScr], rscr _ t;                          * re-fetch bits to xor
  rscr _ t # (rscr);                                    * xor the bits
  t _ memFlagsLocC;
  call[putIMRH];                                        * save new bit values
  returnUsing[Srlink];

```



```

saveReturn[Arlink];

call[getMemState];
rscr _ not(memState.noWake);
call[putMemState], t _ t AND (rscr);      * remove the bit
returnUsing[Arlink];

* October 1, 1978  4:16 PM
* turn on memstate.FIOtest. FIOtest only runs if this bit is true.

FIOtestOn: subroutine;                    * set memState.loopOnError
saveReturn[Arlink];

call[getMemState];
t _ t OR (memState.FIOtest);              * OR in the bit
call[putMemState];
RBASE _ rbase[Arlink];
link _ Arlink;
breakpoint, RBASE _ rbase[defaultRegion];

* October 2, 1978  8:25 AM
* turn off memstate.FIOtest. FIOtest only runs if this bit is true.

FIOtestOff: subroutine;                  * set memState.loopOnError
saveReturn[Arlink];

call[getMemState];
rscr _ not(memState.FIOtest);
call[putMemState], t _ t AND (rscr);      * remove the bit
RBASE _ rbase[Arlink];
link _ Arlink;
breakpoint, RBASE _ rbase[defaultRegion];

* August 19, 1978  2:25 AM
* compose value for mcr:
* enter with t = mcr bits
* exit w/ t = mcr bits OR (correct value for mcr.noWake)
* the correct value of mcr.noWake is the current value of memState.noWake

saOrMcrWake: subroutine;
saveReturnAndT[sRlink, sSubrScr];         * note: using sRlink, sSubrScr
call[checkMemState], t _ memState.noWake;
skpif[alu=0], rscr _ t-t;
rscr _ mcr.noWake;
call[getsSubrScr];
t _ t OR (rscr);
returnUsing[sRlink];

* April 26, 1978  12:55 PM
* turn off loopOnError: for those tests that are so implemented, this bit would cause the
* test to enter a loop if an error occurred, the loop reexecutes the sequence that caused
* the error.

loopOnErrorOff: subroutine;              * set memState.loopOnError
saveReturn[Arlink];

call[getMemState];
rscr _ not (memState.loopOnError);
t _ t AND (rscr);                          * remove in the bit
call[putMemState];
returnUsing[Arlink];

* April 26, 1978  1:46 PM
loopErrorOccured: subroutine;           * SET memState.loopErrorOccured
saveReturn[Arlink];                        * this bit remembers an error so that the
t _ memFlagsLocC;                          * appropriate test will continue to loop even
call[getIMLH];                              * tho the error may be intermittent
t _ t OR (memState.loopErrorOccured);
call[putMemState];
returnUsing[Arlink];

* January 5, 1979  12:07 AM
checkMemState: subroutine;
saveReturnAndT[Arlink, ASubrScr];

```



```

rscr2 _ MD; * rscr2 _ mem[sva]
t # (rscr2);
skpif[alu=0];
sInterfereR1Err:
breakpoint; * mem[sva] # (sva+j)
loopuntil[cnt=0&-1, sInterfereRL1], sva _ (sva) + 1;

IsInterfereR2:
t _ q;
t _ t + (cdMaxVa); * increment unique to its second value
q _ t;
sva _ t-t;
cnt _ 17s; * once for each word in the munch

sInterfereRL2:
t _ (sva) + (q); * t _ sva + unique
FETCH _ t;
rscr2 _ MD; * rscr2 _ mem[sva]
t # (rscr2);
skpif[alu=0];

sInterfereR2Err:
breakpoint; * mem[sva] # (sva+j)
loopuntil[cnt=0&-1, sInterfereRL2], sva _ (sva) + 1;

t _ q;
t _ t + (cdMaxVa); * new value for unique
rscr _ 7c; * construct a mask that affects only
rscr _ lsh[rscr, skipCacheShift]; * cacheA bits in address:
t _ t AND (rscr); * don't let it change to much! This facilitates
q _ t; * more "collisions" in memory task
sva _ t-t;
branch[IsInterfereW1];

```


* September 25, 1978 5:14 PM

%

aHaltTask17

Set tpc[17]_7777C. We depend upon midas keeping a breakpoint in IM at this location. Thus if task 17 gets awakened, it will immediatel cause the machine to halt with a breakpoint.

Clobber rscr, rscr2, t

%

aHaltTask17: subroutine;

```
rscr2 _ link;
top level;
t _ 17c;
rscr _ 7777C;
subroutine;
link _ rscr;
top level;
LdTpc _ t;
taskingOn;
subroutine;
link _ rscr2;
return;
```

* October 3, 1978 5:13 PM

aChkPipeFlt: subroutine;

```
saveReturn[Arlink];
t_ rscr2 _ not(Pipe2'[]);
rscr _ pipe2.nFaults;
t _ t AND (rscr);
t _ t # (rscr);
branch[aChkPipeFltXit, ALU=0], rscr _ t;
rscr2 _ (rscr2) AND (pipe2.faultSrn);
PROCSRN _ rscr2;
rscr _ t;
```

* check pipe2 to see if there's been a fault
* IF NOT return w/ alu=0 fast branch condition

* IF faults THEN procsrnrn _ firstFault, return
* with alu#0 fast branch condition

* nFaults=nFaults mask ==> no faults

* t has (nFaults#mask) which must be #0 to be here

achkPipeFltXit:

```
returnAndBranch[Arlink, rscr];
top level;
```

memSubrsADone: branch[afterSubrs];

**Table of Contents by
Order of Occurrence**

TEST	DESCRIPTION
initBrs	Initialize base register loop control
initCPatterns	Initialize cache pattern loop control
initRowCtrl:	Init cache row loop control
initRowDown	Init cache row Descending loop control
initColCtrl	Init cache column loop control
initCol2Ctrl	Init second cache column loop control
initFlagsCtrl	Init cflags pattern loop control
getCva	Return current value of cva
getSubrScr	Return current value of subrScr
getcBrCacheABits	return current value of cBrCacheABits
nextBr	Return value of next BR
nextCol	Return value of next column
nextCol2	Return value of next column2 (ther's two loop variables, col & col2).
getCol2	Return current value of column2
nextRow	Return value of next cache row
nextRowDown	Return next value of row, descending loop control
getCrow	Return current value for cache row
nextCFlag	Return next value for CFlags
getCflag	Return current value of flagsVal
nextCPattern	Generate next cache pattern. Return ALU=0 when no more
getCPattern	Generate current pattern based on currnet pattern index (changed by nextCPattern)
mcrForCol	Set mcr for current column, extra flags passed in t
vaForRow	Return in T a va that matches row in T upon entry
cRowForVa	Return cache row in T for va in T upon entry
cVaForCrowCol	Return va that matches row in T, col in rscr.
makeUseMcrV	Construct mcr value and forces useMcrV for column in T
setBrCacheABits	Set hi 15 bits of memory base register
setBr	BrLo _ rscr2, BrHi _ rscr15 bits of memory base register
setBr	BrLo _ rscr2, BrHi _ rscr
setMCR	mcr _ t
longWait	Wait number of cycles in T where T = 0 or T>3. Count doesn't include call and
return overhead	
checkMemFlags	Subroutine that returns w/ ALU#0 if any memflags in T are set
setMbase	Set current memBase to T
clearCacheFlags	Set entire cache to vacant
setCAMem	Set cacheA memory w/ va = addr to write, col = column select
setCflags	Set cache flags for T = va, rscr = pattern, col = column
putCFmem	Set cache flags t = va, rscr = flags, rscr2 = column
putCAMem	Set CacheA memory: t = va, rscr = brhi15, rscr2 = column
zeroCacheAndFlags	Zero cacheAmem and zero cache flags
readCflags	read flags for rscr = va, rscr2 = column
getPipeCacheABits	return h 15 bits of pipe VA in T
chkPipeRow	Enter w/ t = row, check that pipel.row = t (ALU#0 if false)
chkPipe5Col	RTN ALU#0 if pipe5.col # T
chkCflags	RTN T #0 if (rscr.cflags) # t
setRegs0	(rscr,rscr2,t) _ (RM 0, RM 1, RM 2) midas subroutine
setRegs1	(rscr,rscr2,t) _ (RM 3, RM 4, RM 5) midas subroutine
setRegs2	(rscr,rscr2,t) _ (RM 6, RM 7, RM 10) midas subroutine

%	
June 9, 1981 10:35 AM	
	Accommodate default values for memState (left half of memFlags)
May 21, 1981 4:32 PM	
	Fix setMbase to use bmux rather than bdispatch
February 5, 1981 6:02 PM	
	Remove forceRbase because dilang now accommodates our need. Change brx to cBrX to
	accommodate change in memdefs required by changes to dilang.
	Trying to find obscure problem, possibly related to some debugging instructions' not being
Held. add '_MD' to zeroCacheAndFlags	
July 1, 1979 2:39 PM	
	Add table of contents, cause cRowForVa to use ldf rather than two constants & clobber
rscr2.	
May 29, 1979 11:07 AM	
	Modify clearCacheFlags to use putCFmem.
%	

```

title[memSubrsC];
top level;
subroutine;
* February 5, 1981 6:07 PM

initBrs:
    t _ cml;
    return, cBrX _ t;

initCPatterns:
    RBASE_ rbase[patX];
    t_patX _ cml;
    curPattern _ t;
    return, RBASE_ rbase[defaultRegion];

initRowCtrl:
    t _ cml;
    return, rowx _ t;

initRowDown:
    t _ rowEndC;
    return, rowx _ t;

initColCtrl:
    t _ cml;
    return, colx _ t;

initCol2Ctrl:
    t _ cml;
    return, col2x _ t;

initFlagsCtrl:
    RBASE_ rbase[flagsVal];
    flagsVal _ (flagsVal)-(flagsVal);
    flagsVal _ (flagsVal)-(20c);          * first val returned will be zero
    return, RBASE_ rbase[defaultRegion];

getCva: subroutine;
    RBASE _ rbase[cVa];
    return, t _ cVa, RBASE _ rbase[defaultRegion];

getSubrScr: subroutine;
    RBASE _ rbase[subrScr];
    return, t _ subrScr, RBASE _ rbase[defaultRegion];

getCBrCacheABits: subroutine;
    RBASE _ rbase[cBrCacheAbits];
    return, t _ cBrCacheAbits, RBASE _ rbase[defaultRegion];

nextBr:                                * compute mem base reg; return it in T
* January 1, 1979 3:04 PM
    RBASE_ rbase[cBrX];                    * compute end of base regs as branch condition
    t_cBrX_(cBrX)+1;
    RBASE_ rbase[defaultRegion];
    return,t-(brEndC);

nextCol:                                * compute next column; return it in T
                                        * compute end of columns as branch condition
    RBASE_ rbase[colx];
    t_colx_(colx)+1;
    RBASE_ rbase[defaultRegion];
    return,t-(colEndC);

nextCol2:                                * compute next column; return it in T
                                        * compute end of columns as branch condition
    RBASE_ rbase[col2x];
    t_col2x_(col2x)+1;
    RBASE_ rbase[defaultRegion];
    return,t-(colEndC);

getCol2:                                * return current row in T
    RBASE_ rbase[col2x];
    return, t_col2x, RBASE _ rbase[defaultRegion];

nextRow:                                * compute next row; return it in T
                                        * compute end of row as branch condition
    RBASE_ rbase[rowx];

```

```
    t_rowx_(rowx)+1;
    RBASE_ rbase[defaultRegion];
    return,t-(rowEndC);
nextRowDown:                                * compute next row(descending); return it in T
    RBASE_ rbase[rowx];                        * compute end of row as branch condition
    t_rowx_(rowx)-1;
    RBASE_ rbase[defaultRegion];
    return,t-(cml);

getCrow:                                    * return current row in T
    RBASE_ rbase[rowx];
    return, t_rowx, RBASE _ rbase[defaultRegion];

nextCFlag:                                  * return next cache flag value in T
    RBASE _ rbase[flagsVal];                  * return w/ fast branch condition
    t _ flagsVal _ (flagsVal)+(20C);
    RBASE _ rbase[defaultRegion];
    return,t-(flagsEndC);

getCflag:                                   * return current row in T
    RBASE_ rbase[flagsVal];
    return, t_flagsVal, RBASE _ rbase[defaultRegion];
```

* December 26, 1978 11:16 AM

%

Cache Pattern loop control: There are three sets of patterns. The first is a left cycled, single one bit. The second is a left cycled, single zero bit, and the third is a composite number = (current row lshift 3)+ current column. This routine returns an ALU result upon exit whose zero result means there are no more valid patterns. To obtain the actual pattern, the diagnostics use the subroutine getCpattern.

nextCPattern: PROCEDURE RETURNS[morePatterns: BOOLEAN] =

BEGIN

patX _ patX + 1;

SELECT patX FROM

 IN [0..Pat1EndC) => curPattern _ doPat1[patX, curPattern];

 IN [Pat1EndC..Pat2EndC) => curPattern _ doPat2[patX, curPattern];

 IN [Pat2EndC..Pat3EndC) => curPattern _ doPat3[patX, curPattern];

 ENDCASE => NULL;

morePatterns _ patX >= LastPatC

END

%

nextCPattern:

 RBASE_ rbase[patx];

 pushReturn[];

 top level;

 t_patx_(patx)+1;

 t-(pat1EndC);

 branch[nxtPat2, alu>=0],t_t;

* compute next pattern index; return in t

* compute end of patterns as branch condition

* save return link

* keep patX in T

* see if IN [0..pat1EndC)

* goto[nextPat2, ~IN [0..pat1EndC)]

* IN [0..pat1EndC)

 skpif[alu#0];

 skip, curPattern _ 1c;

 curPattern_(curPattern)+(curPattern);

 branch[nxtPatXit];

* see if just initialized

* patX=0 ==> begin with one

nxtPat2:

 t-(pat2EndC);

 branch[nxtPat3, alu>=0];

* see if IN[pat1EndC..pat2EndC)

* goto[nextPat3, ~IN[pat1EndC..pat2EndC)]

* IN [pat1EndC..pat2EndC)

 curPattern _ lcy[curPattern, curPattern, 1];

 t - (pat1EndC);

 skpif[ALU#0];

 curPattern _ not(b15);

 branch[nxtPatXit];

* If first time thru pattern 2, init

* curPattern for left cycled zero pattern.

nxtPat3:

 t - (pat3EndC);

 branch[nxtPat4, alu>=0];

 curPattern _ (curPattern)-(curPattern);

 branch[nxtPatXit];

* are we IN [pat3EndC..lastPat) ??

* Zero it since getCPattern constructs

* each value separately.

nxtPat4:

 branch[nxtPatXit];

* add code here for another pattern.

nxtPatXit:

 link _ stack&-1;

 subroutine;

 RBASE_ rbase[defaultRegion];

 return,t-(lastPatC);

* restore return link, fix up rbase,

* return w/ proper branch condition


```

* December 7, 1978 3:04 PM
mcrForCol: subroutine; * set mcr for current column
% extra falgs to set are passed in T.
This code uses an array of four mcr values that is stored in IM at caMcrLoc..caMcrLoc+3. There is an
mcr value for each column, plus bits for useMcrV, disCF, and disHold.
%

    rscr _ mcr.useMcrV;
    rscr _ (rscr) OR (mcr.disCF);
    rscr _ (rscr) OR (mcr.disHold);
    rscr _ (rscr) OR (mcr.noWake);
    rscr _ (rscr) or t; * Or in the extra bits
    t _ lsh[col, mcr.mcrVshift]; * position the column for mcr
    t _ t or (rscr);
    noop; * assure there's 8 cycles from time of
    loadMcr[t,t]; * last memory reference
    return;

vaForRow: * construct va that matches row in T
* clobber T and rscr2

    t _ lsh[t, cacheShift];
    rscr2 _ cacheRowMask0;
    rscr2 _ (rscr2) OR (cacheRowMask1);
    return, t_tAND(rscr2); * isolate bits of row
cRowForVa: * t = va. return the cache row. CLOBBER t, rscr2
    return, t _ ldf[t, nBitsInRow, cacheShift];

cVaForCrowCol: * t = row, rscr = column. RETURN va
* CLOBBER T, RSCR, RSCR2

    t _ lsh[t, cacheShift];
    rscr _ lsh[rscr, skipCacheShift];

    return, t _ t or (rscr);

makeUseMcrV: * construct an mcr value
* that specifies mcr.useMcrV, and selects contents of T as column for victim
* clobber T and rscr2

    t_tAND(3c); * just for paranoia
    t_lsh[t, mcr.mcrVshift];
    return, t_t OR (mcr.useMcrV); * rtn w/ t= mcr value

setBrCacheABits: * set hi 15 bits of memory base reg
* Enter w/ T = 15 bit value to use. clobber T, rscr, and rscr2

    rscr _ rsh[t, CABitsInPipe1];
    rscr _ (rscr) and (CABitsInPipe0Mask);
    rscr2 _ lsh[t, CABitsInPipe1Shift];
    rscr2 _ (rscr2) and (CABitsInPipe1Mask);
    BrHi _ rscr;
    return, BrLo _ rscr2;

setBR: * rscr2 = brLO, rscr = brHI
    brlo _ rscr2;
    brhi _ rscr;
    return;

setMCR:
noop; noop; noop; noop; *NOTE: Hardware constraints require minimum
noop; noop; noop; * of 8 cycles between memops and setting mcr.

    loadmcr[t,t]; * set MCR w/ T
    noop; * wait for MCR to settle down
    return;

* December 4, 1978 5:38 PM

longWait: subroutine; * wait (T + 2) cycles where T >3, not counting call or
return! SPECIAL KLUDGE: IF T=0, return almost immediately (ie. total delay = 2 cycles + call + return)
PD_t; * CLOBBER T
loopUntil[ALU=0, .], t _ t-1;

```


* May 21, 1981 4:32 PM

%

Set MemBase to value in t.

%

subroutine;

setMbase:

* t = value to use

MemBase_t, RETURN;

* May 29, 1979 11:07 AM
%

Clear Cache Flags

```
clearCacheFlags: PROCEDURE =
  BEGIN
    FOR rowX IN CacheRowX DO
      t _ getVaForRow[rowX] - cflags.vacant;
      FOR col IN Column DO
        makeUniqueAmemEntryAtRowCol[rowX, col];
        setBR[0,t];
        mcrV _ MakeUseMcrV[col];
        mcrV _ BitOR[mcrV, mcr.fMiss, mcr.disHold];
        setMcr[mcrV];
        dummyRef _ cflags.vacant;
        CFLAGS _ cflags.vacant;
      ENDLLOOP;
    ENDLLOOP;
  END;
```

This subroutine causes the entire cache to be vacant. It clobbers MCR, BR
%

```
clearCacheFlags:
  pushReturn[];

  call[initRowCtrl];
  clrCacheFrowL: * loop for all the rows of the cache
    call[nextRow];
    skipif[alu#0];
    branch[clrCacheXit];

    noop; * for placement

    call[initColCtrl];
    clrCacheFcolL: * loop for the columns in this row
      call[nextCol];
      skipif[alu#0];
      branch[clrCacheFrowL];

      col _ t; * col is the current column

      call[getCrow], rscr _ t; * return w/ t = rowX
      call[cVaForCrowCol]; * t = row, rscr = col, return t = va
      call[setCAmem], va _ t; * enter w/ t = va, col = column
      rscr2 _ col;
      rscr _ cflags.vacant;
      call[putCFmem], t _ va;
      branch[clrCacheFcolL];

  clrCacheXit:
    returnP[];
```

* December 4, 1978 5:41 PM

```

setCAmem:                                     * yet another way to set Amemory
* ENTER w/ va = addr to write, col = column to select
* CLOBBER T, rscr, rscr2
  pushReturn[];
  t _ col;
  rscr2 _ t;
  t _ va;
  call[putCAmem], rscr _ t-t;                   * call w/ t = va, rscr = brHil5, rscr2 =col
  returnP[];

setCflags:                                     * T = va, rscr =pattern, col = column
  pushReturnAndT[];
  t _ stack;
  call[putCFmem], rscr2 _ col;
  pReturnP[];

putCFmem: subroutine;                          * t = va, rscr=flags, rscr2 = col
  cva _ t;
  pushReturn[];
  t _ rscr;                                     * save flags value
  cflagsV _ t;
  call[makeUseMcrV], t _ rscr2;
  t _ t OR (mcr.fdmMiss);
  t _ t OR (mcr.noRefHold);
  call[setMCR];
  call[getCva];
  rscr2 _ t;                                    * rscr2 _ cva
  t _ not(rscr);                               * t _ cflags
  t _ t and (cflags.mask);                    * invert only the cflags bits
  rscr2 _ (rscr2)-t;                          * rscr2 _ cva - flags
  call[setBR], rscr _ t-t;
  dummyRef _ t;
  CFLAGS _ t;
  returnP[];

putCAmem: subroutine;                          * t = va, rscr = brhil5, rscr2 = col;
  cva _ t;
  pushReturn[];
  t _ rscr;
  cBrCacheAbits _ t;

  t _ col;
  subrScr _ t;

  col _ rscr2;
  t _ (mcr.fdmMiss);
  t _ t OR (mcr.noRef);
  call[mcrForCol];                             * no extra flags, use rscr2 as victim
  call[getcBrCacheAbits];
  call[setBrCacheAbits];
  call[getCva];
  rscr2 _ t;
  DBuf _ r0, STORE _ T;
  shortMemWait[rscr];
  t _ rscr2;                                   * restore t; shortMemWait used it
  DBuf _ r0, STORE _ T;
  shortMemWait[rscr];

  call[getSubrScr];
  col _ t;
  returnP[];

```

* September 19, 1979 10:52 PM

```

zeroCacheAndFlags: subroutine;
* CLOBBER RSCR, RSCR2, T
  pushReturn[];
  t_MD;
  call[initRowCtrl];
zcafRowL:
  call[nextRow];
  skipif[alu#0];
  branch[zcafRtn];
  q _ t;

  call[initColCtrl];
zcafColl:
  call[nextCol];
  skipif[alu#0];
  branch[zcafRowL];
  col _ t;
  t _ q;
  call[vaForRow];

  rscr _ t;
  t _ col;
  rscr2 _ t;
  t _ rscr;
  call[putCAmem], rscr _ t-t;

  call[vaForRow], t _ q;
  call[setCflags], rscr _ t-t;
  branch[zcafColl];

zCafRtn:
  returnP[];

```

* zero cache Amem and cache flags

* see if this makes obscure problems go away.

* save row in Q

* 'round the register merry go round

* t = va, rscr = brHil5, rscr2 = col

* t = va, rscr = pattern, col = column

```

* December 7, 1978 3:11 PM                    PIPE reading routines
readCflags:                                * rscr = va, rscr2 = column
* clobber T, rscr, rscr2. Return all the bits from PIPE5 in T

    pushReturn[];
    top level;
* Set MCR for appropriate column, fdMiss, dPipeVA, disHold (use disBR for convenience)
    call[makeUseMcrV],t _ rscr2;            * setup mcr. RSCR ~clobbered !
    t _ t OR (mcr.fdMiss);
    t _ t OR (mcr.dPipeVa);
    t _ t OR (mcr.disBR);                    * convenience: needn't load BR!
    t_tOR (mcr.noRefHold);                 * mcr.noRef + mcr.disHold
    call[setMCR];

    dummyRef _ rscr;
    noop;
    t _ PIPE5;                                * now read and remember flags
%
notice that we don't invert the bits for cflags. To write a set of flags F, the processor must first
invert F (F'), and then proceed. However, when reading the flags the bit
values have their "true" sex.
%
    returnP[];                                * return w/ flag bits in T

getPipeCacheABits: subroutine;            * return hi 15 bits of pipe VA in T.
* leave rscr2 untouched.
* clobber T, rscr

    t_pipe0;
    t _ t and (CABitsInPipe0Mask);
    rscr _ pipel;
    rscr _ (rscr) AND (CABitsInPipe1Mask);
    t_lsh[t, CABitsInPipe1];                * hi bits of VA from pipe 0
    rscr_rsh[rscr, CABitsInPipe1Shift];    * isolate mid bits of VA from pipe 1
    t_t OR (rscr);                            * add them together and mask
    return, t_t AND (CABitsMaskC);         * return: t= hi15 bits of VA rt. justified

```

* December 7, 1978 5:17 PM

Cache RESULT TESTING subroutines

chkPipeRow: subroutine;
* clobber T, rscr, rscr2

* enter with T = row. check that pipe1.row = t

```
rscr _ (cacheRowMask0);
rscr _ (rscr) OR (cacheRowMask1);
rscr2 _ PIPE1;
t _ lsh[t, cacheShift];
T _ (rscr2) # (t);
return, T _ (rscr) AND (t);
```

* construct row bits mask
* NOW CHECK ROW=CACHE BITS. t=ROW

* use row bits only

chkPipe5Col: subroutine;

* T=expected col, rscr =PIPE5 value

* return w/ T=bad bits, rscr= PIPE5 val. return w/ fast branch condition

* CLOBBER rscr2,t

```
t _ t and (3c);
t _ lsh[t, pipe5.colShift];
t_t#(rscr);
rscr2 _ pipe5.colBit0;
rscr2 _ (rscr2) + (pipe5.colBit1);
return, t_t and (rscr2);
```

* mask for paranoia

* position column field

* construct mask to isolate col field

* return w/ fast branch condition

chkCflags: subroutine;

* T = expected flags, rscr = PIPE5 value

* return w/ T=bad bits, rscr = PIPE5 val. return w/ fast branch condition

* clobber rscr2, t

```
t_t#(rscr);
rscr2 _ pipe5.flagsMask;
return, t_t and (pipe5.flagsMask);
```

* isolate different bits

* return w/ fast branch condition

```
* February 5, 1981 6:02 PM
% subroutines for midas level degbugging
```

```
This code sets rscr, rscr2, and T from RM0 thru RM10
%
```

```
top level;
call[setRegs0]; * these calls assure that the named
call[setRegs1]; * subroutines really get loaded onto call
call[setRegs2]; * locations.
noop; * sigh. here for placement.
```

```
setRegs0: subroutine;
RBASE _ 0s;
t _ rmx0;
rscr _ t;
t _ rmx1;
rscr2 _ t;
return, t _ rmx2, RBASE _ rbase[defaultRegion];
```

```
setRegs1: subroutine;
RBASE _ 0s;
t _ rmx3;
rscr _ t;
t _ rmx4;
rscr2 _ t;
return, t _ rmx5, RBASE _ rbase[defaultRegion];
```

```
setRegs2: subroutine;
RBASE _ 0s;
t _ rmx6;
rscr _ t;
t _ rmx7;
rscr2 _ t;
return, t _ rmx10, RBASE _ rbase[defaultRegion];
```

```
data[(memFlags: lh[defaultMemFlagsLeft], rh[defaultMemFlags], at[memFlagsLoc])];
```

title[memSubrsChaos];

table of contents by order of occurrence

Name	Description
iChaosCache	Initialize the 4 columns of the 2 rows chaos uses.
sChaosChkAddr	subroutine to verify validity of all of vm after chaos test is done
isChaosMem	initialize mem[va] _ va
iSavedHoldValue	save current holdValue (postamble uses it to drive task/hold simulator)
getSchaosRow0Addr	return current value of sChaosRow0 shifted into an address
getSchaosRow1Addr	return current value of sChaosRow1 shifted into an address
sShadowBitToVa	set addr.shadowBit in sVa if the current two bit nibble in our random number indicates that this munch should have the shadow bit set.
CFlagsFromTemplate	set cflags.dirty for the value for the current munch's CFLAGS if the current two bit nibble in our random number indicates this munch should be dirty.
iSchaosCode	Initialize the random rm values, patch IM as required by pseudo random number generator.
sChaosAddrFromRandV	Generate an address for "row0" or for "row1" based upon current sChaosRandV
getChaosTaskFreq	Return 7 bit random number to put into task portion of hold&tasksim.
get1Rand	subroutine to get a random number
chaosTsk<i>Tst<j>	i=0 means task 0, i=1 means simulator task, j IN [0..3]. This is the code that gets patched based on pseudorandom numbers. It's execution is the chaos test.

%
 September 14, 1979 3:37 PM
 Make some of the code shorter to help with micro's storage full problems. Add sChaosChkAddr.
 Add isChaosMem and iSavedHoldValue, too.
 June 25, 1979 11:09 PM
 Add get1Rand because mema won't place due to the way "getRandom[]" works.
 June 25, 1979 10:32 PM
 add getChaosTaskFrq.
 June 25, 1979 8:33 AM
 Change placement declarations to accommodate ifu entry points.
 April 18, 1979 11:44 AM
 Moved iChaosCache, iChaosCode into here from memChaosS.mc
 %

* This code duplicated in memChaosS.mc!

```
mc[codeRand.store, 40];
mc[codeRand.wordX, 17];
mc[codeRand.rowX, 20];
set[codeRand.colSize, 2];          * mc[codeRand.colX, 300];
set[codeRand.colShift, 6];
set[codeRand.delaySize, 5];       * mc[codeRand.delay, 17400]
set[codeRand.delayShift, 10];
```

```
mc[addr.shadowBit, lshift[4,skipCacheShift]]; * va bit for shadow address
```

* addrRand is a two bit nibble selected from the current two lsb of a random number.
 * the test uses addrRand to determine how to initialize each column of the two rows
 * in the cache.

```
mc[addrRand.dirty, 1];           * mask bit for address template.
mc[addrRand.shadow, 2];         * mask bit for address template
```


* January 16, 1979 7:26 PM

%

iChaosCache

This code initializes all four columns of two rows of the cache. Eight two-bit nibbles in a random number determine for each column-row munch whether that munch is dirty and whether that munch belongs to the "shadow" address. Remember that eight addresses (munch addresses) are not enough to cause cache misses for two rows worth of data. Consequently there is an extra "shadow" bit that may set in the cache address, and that bit provides enough address space to cause misses in the cache.

Obtain the two row addresses from **sChaosRow0** and **sChaosRow1**. These are row numbers, not addresses!

%

iChaosCache: subroutine;

```
pushReturn[];
call[getSchaosRow0Addr];
call[xVacateCacheRow];          * clear cache "row 0"
call[getSchaosRow1Addr];
call[xVacateCacheRow];          * clear cache "row 1"
```

```
call[get1Rand];
sChaosRandV _ t;
```

```
t _ cml;          * quit row loop when we see -1
stack+1 _ t;     * push the two row values onto the stack
call[getSchaosRow0Addr];
stack+1 _ t;
call[getSchaosRow1Addr];
stack+1 _ t;
```

isChaosCRowL:

```
col _ cml;          * top of loop that inits row zero, then row
cnt _ 3s;          * addr.rowl
```

isChaosColL:

```
col _ (col) + 1;   * move to next column, select it as the current victim
rscr _ t-t;
call[setBR],rscr2_a0;
call[makeUseMcrv],t _ col;
call[setMCR];      * init BR to zero each time putCFmem clobbers it
                  * and perform a reference that will force the
                  * desired address into the cache.
```

```
call[sShadowBitToVa];
t _ lsh[col, skipCacheShift];
sVa _ (sVa) OR t;  * returns after va _ shadowBit
t _ stack;        * into the munch selection position in the va
sVa _ t or (sVa); * get row address.
FETCH _ sVa;      * OR the current row into the va
```

```
call[cFlagsFromTemplate];
rscr2 _ col;      * get cflags value into RSCR based upon
t _ MD;          * sChaosRandV, then write Cflags into cache.
call[putCFmem], t _ sVa;
t _ 100c;        * don't forget to synchronize w/ memory
call[longWait];  * t = va, rscr2 = column, rscr = "cflags"
                  * wait for cache to settle down
```

```
sChaosRandV _ rsh[sChaosRandV, 2];
loopUntil[cnt=0&-1, isChaosColL]; * move on to next column...
```

```
stkp-1;
t _ stack;
t # (cml);
branch[isChaosCacheXit, ALU=0];
branch[isChaosCRowL]; * quit when we see -1
```

isChaosCacheXit:

```
noop;          * for placement
call[sRestoreMcrVictim];
pReturnP[];    * pop the -1 off the stack before returning
```

* June 25, 1979 2:38 PM

%

sChaosChkAdrrs

This code runs at the end of the chaos test. It reads vm to see that mem[va] = va for all the location in the memory. If this isn't true, some earlier action of the chaos test clobbered the location that now has the incorrect value. Due to the time consumption of this check it does not run at the end of each chaos "run".

%

sChaosChkAdrrs: * scan all of vm to see that its still correct
 pushReturn[];
 call[iSvaCtrl];

sChaosChkAdrrsL:

call[nextSva];
 skipif[alu#0], rscr2 _ t; * remember sva in rscr2
 branch[sChaosChkAdrrsXit]; * everything was ok
 FETCH _ t;
 rscr _ MD;
 t _ t # (rscr); * compare sva w/ mem[sva]
 skipif[alu=0];

sChaosAdrrsErr:

error; * t = bad bits, rscr = md, rscr2 = expected
 branch[sChaosChkAdrrsL]; * value from memory
 * try next value

sChaosChkAdrrsXit:

returnP[];

* June 25, 1979 2:40 PM

* init memory so that mem[va] = va

iSChaosMem:

pushReturn[];
 call[iSvaCtrl];

iSChaosL:

call[nextSva]; * background memory w/ "self"
 skipif[alu#0];
 branch[sChaosFlush];
 branch[iSChaosL], store _ t, DBuf _ t; * mem[va] _ va;

sChaosFlush:

rscr _ 20000c; * there are dirty munches left in the cache
 * cycle thru the cache to make sure we force

sChaosFlushL:

STORE _ rscr, DBuf _ rscr; * dirty munches to storage
 rscr _ (rscr) -1;
 loopuntil[alu<0, sChaosFlushL];
 noop;
 returnP[];

* January 13, 1979 12:29 AM

* Init sSavedHoldValue which we use for starting the Hold simulator.

* Prevent the maximum delay from being longer than 200B - 150B

iSavedHoldValue: subroutine;

pushReturn[]; * fire-up task simulator long enough to
 * save the hold value in sSavedHoldValue
 mc[taskFreqLocC, 302]; * UGH! Yes, this is a DUPLICATE definition.
 t _ taskFreqLocC; * It copies the one in postamble.mc
 call[getIMRH];
 t - (150c);
 branch[ishOK, ALU>=0];

* the current value of holdFreq is too small. set it to our minimum = 150

rscr _ 150c;
 t _ taskFreqLocC;
 call[putIMRH];
 noop;

ishOK:

B_FaultInfo[];
 call[enableConditionalTask]; * remove any lurking memroy errors
 * HACK. FORCE postamble to make
 t _ holdValueLocC; * a new value for hold simulator.
 call[getIMRH]; * Then we remember its value for
 * future use -- and we diable the
 sSavedHoldValue _ t;
 call[disableConditionalTask]; * to prevent task sim's memory refs

```
* from screwing up the cache.  
  returnP[];
```

* January 13, 1979 2:32 AM

%

getSchaosRow0Addr return current value of sChaosRow0 shifted into an address

getSchaosRow1Addr return current value of sChaosRow1 shifted into an address

sShadowBitToVa set addr.shadowBit in sVa if the current two bit nibble in our random number indicates that this munch should have the shadow bit set.

CFlagsFromTemplate set cflags.dirty for the value for the current munch's CFLAGS if the current two bit nibble in our random number indicates this munch should be dirty.

%

subroutine;

getSchaosRow0Addr:

```
RBASE _ rbase[sChaosRow0];
t _ sChaosRow0, RBASE _ rbase[defaultRegion];
return, t_ lsh[t, cacheShift];
```

getSchaosRow1Addr:

```
RBASE _ rbase[sChaosRow1];
t _ sChaosRow1, RBASE _ rbase[defaultRegion];
return, t_ lsh[t, cacheShift];
```

sShadowBitToVa:

```
(sChaosRandV) AND (addrRand.shadow); * see if shadow bit is set in template
skipif[alu=0], sva _ t-t;
sva _ (sva) OR (addr.shadowBit); * add shadow bit to va
return;
```

CFlagsFromTemplate:

```
noop; * set rscr _ cflags based upon template dirty bit
(sChaosRandV) AND (addrRand.dirty);
skipif[alu=0], rscr _ t-t;
rscr _ cflags.dirty;
return;
```

* January 16, 1979 4:47 PM

%

iSChaosCode

Initialize the chaos code. The code immediately below initializes the eight registers rm00-rm03 and rml0-rml3. The emulator level code uses the first group of four and the simulator task uses the last group of four.

Call the subroutine **setChaosCode** twice, once for the emulator code and once for the simulator task code. That subroutine initializes the FF constants and long jumps used by the chaos code. Notice this requires writing valid instructions into IM.

%

iSchaosCode: subroutine;
pushReturn[];

* TASK 0 RM and Code Init

```

call[get1Rand];
sChaosRandV _ t;
sSubrScr _ t;
call[sChaosAddrFromRandV];
rm00 _ t;
rm01 _ t;

call[get1Rand];
sChaosRandV _ t;
call[sChaosAddrFromRandV];
rm02 _ t;
rm03 _ t;
call[getsSubrScr];
rscr2 _ t;
call[setChaosCode], t _ r0;

```

* save currend random value
* init rm for first memop of task 0

* init rm for second memop of task 0

* set-up pointer to proper code, init delay

* Sim Task RM and Code Init

```

call[get1Rand];
sChaosRandV _ t;
sSubrScr _ t;
call[sChaosAddrFromRandV];
rm10 _ t;
rm11 _ t;
rscr2 _ sChaosRandV;

call[get1Rand];
sChaosRandV _ t;
call[sChaosAddrFromRandV];
RM12 _ t;
rm13 _ t;
call[getsSubrScr];
rscr2 _ t;
call[setChaosCode], t _ 4c;
returnP[];

```

* save currend random value
* init rm for first memop of task 17

* init rm for second memop of task 17

* set-up pointer to proper code, init delay

sChaosAddrFromRandV:

```

pushReturn[];
(sChaosRandV) and (codeRand.rowX);
branch[scaRow0, ALU=0];

```

* this addr should use "row0"
* this addr should use "row1"

scaRow1:

```

noop;
call[getSchaosRow1Addr];

```

scaMakeAddr:

```

stack+1 _ t;
t _ (sChaosRandV) and (codeRand.wordX);
stack _ t or (stack);
t _ ldf[sChaosRandV, codeRand.colSize, codeRand.colShift];
t _ lsh[t, skipCacheShift];
t _ t or (stack&-1);

```

* remember row address in stack
* OR row, word bits
* or column into the addr, pop stack

```

returnP[];

```

scaRow0: top level;
call[getSchaosRow0Addr];
branch[scaMakeAddr];

* current address should come from "row0"

* June 25, 1979 10:42 PM

getChaosTaskFreq:

pushReturn[];

gctfL:

noop;

* for placement

call[get1Rand];

t _ t and (177C);

t _ t - (77C);

* don't want to wait more than 77 cycles

loopUntil[ALU>=0, gctfL];

t _ lsh[t, sim.taskShift];

returnP[];

* June 25, 1979 11:08 PM

get1Rand:

pushReturn[];

noop;

* maximize placement advantage

getRandom[];

returnP[];

%

CHAOS Task Code

%

```

set[xtask, 1];
chaosTsk1Tst0:
  hold&tasksim _ t,
  noop;
  block;

  FETCH _ rm10;
  t _ 0c,
  call[longWait];
  rm11 _ MD;
  FETCH _ rm12;
  rm13 _ MD;

  branch[chaosSimXit];

  at[chaosTestBase, 4]; * awakened by the simulator
  * 1st instr after hold&tasksim_ doesnt count

  at[chaosDelayBase,4]; * this gets patched each time
  * rm11 _ mem[rm10]
  * rm13 _ mem[rm12]
  * proceed to check task 17 results

chaosTsk1Tst1:
  hold&tasksim _ t,
  noop;
  block;

  FETCH _ rm10;
  t _ 0c,
  call[longWait];
  rm11 _ MD;
  STORE _ rm12, DBuf _ rm12;

  branch[chaosSimXit];

  at[chaosTestBase, 5]; * awakened by the simulator
  * 1st instr after hold&tasksim_ doesnt count

  at[chaosDelayBase,5]; * gets patched each time
  * rm11 _ mem[rm10]
  * mem[rm12] _ rm12
  * proceed to check task 17 results

chaosTsk1Tst2:
  hold&tasksim _ t,
  noop;
  block;

  STORE _ rm10, DBuf _ rm10;
  t _ 0c,
  call[longWait];
  FETCH _ rm12;
  rm13 _ MD;

  branch[chaosSimXit];

  at[chaosTestBase,6]; * awakened by the simulator
  * 1st instr after hold_ doesnt count

  * mem[rm10] _ rm10
  at[chaosDelayBase,6]; * gets patched each time

  * proceed to check task 17 results

chaosTsk1Tst3:
  * STORE, Delay, FETCH

  hold&tasksim _ t,
  noop;
  block;

  STORE _ rm10, DBuf _ rm10;
  t _ 0c,
  call[longWait];
  STORE _ rm12, DBuf _ rm12;

  branch[chaosSimXit];

  at[chaosTestBase,7]; * awakened by the simulator
  * 1st instr after hold_ doesnt count

  * mem[rm10] _ rm10
  at[chaosDelayBase,7]; * gets patched each time
  * mem[rm12] _ rm12
  * proceed to check task 17 results

```



```

title[memSubrsD];
top level;
%*+++++
Routine                               Description
iMunchCtrl:                          Initialize cache munch counter
iCDvaCtrl:                            Initialize cache va control
iCDpatCtrl:                          Initialize cache data pattern control
nextVa:                                Return next cache va in "va"
nextMunch:                            Return next munch addr in "va"
nextCDpat:                            Return next cache data pattern
getCDpat:                             Return current cache data pattern
iDboard:                              Initialize the D-board
cdRead:                               Read a word from va = t, result in rscr2
cdWrite:                              Write a word w/ t = va, rscr = data
colForVa:                             Return t = column for a va (not general purpose)
cacheAforVa:                          Return "midas" cache addr for a va in t
presetCache:                          Set all cache flags cache addr
pcSetCflags:                          Set CFlags; private subr for presetCache
pcSetAmemory:                         Set cacheA, private subr for presetCache
zeroCache0:                           For va IN cacheAddrs DO cache[va]_0
setCache0:                             FOR va IN cacheAddrs DO cach[va]_t
pcGetHi8B:                             Return current pcHi8
pcGetCflags:                          Return current pcFlags
%*+++++
%
May 21, 1981 11:46 AM
    Add setCache0 -- a modification of zeroCache0
August 13, 1979 6:06 PM
    Cause iCDpatCtrl to initialize memory storage patterns, too.
August 13, 1979 5:23 PM
    Old getCDpat, nextCDpat assumed RBASE=rmForMemDLoops, new implementation uses RBASE =
defaultregion; fix the discrepancy.
August 13, 1979 3:28 PM
    Add new patterns for cacheData testing.
May 4, 1979 5:14 PM
    Remove branch[afterDtest].
%

SUBROUTINE;

knowRbase[rmForMemDLoops];

iMunchCtrl:
    RBASE _ rbase[cMunchVa];
    cMunchVa _ t - (20c);
    t _ (t)+(cdMaxVa);
    return, cMunchEnd _ t, rbase _ rbase[defaultRegion];

iCDvaCtrl:                                * init next va control. T = begin addr.
* va IN [beginAddr..beginAddr+4000B)

    RBASE _ rbase[cdVa];
    cdVa _ t - 1;                                * initial value -1
    t _ (t)+(cdMaxVa);
    cdVaEnd _ t;                                * last value +1
    return, cdVaEnd _ t, rbase _ rbase[defaultRegion];

iCDpatCtrl:
    t_cml;
    CDpatx _ t;
    SpatX _ t;
    curSPattern _ t;
    return, curCDpattern _ t;

```

* December 22, 1977 10:20 AM

nextVa:

```
RBASE _ rbase[cdVa];
t _ cdVa _ (cdVa)+1, ;
va _ t;
t _ cdVaEnd;
RBASE _ rbase[defaultRegion];
return, (va)-(t);
```

* WRITE NEXTVA DIRECTLY INTO "VA"

* write it into caller's rbase

nextMunch:

```
rbase _ rbase[cMunchVa];
t _ cMunchVa _ (cMunchVa) + (20C);
va _ t;
t _ cMunchEnd;
RBASE _ rbase[defaultRegion];
return, (va)-(t);
```

* WRITE next munch DIRECTLY INTO VA

* ugh. return w/ fast branch condition

* August 13, 1979 5:26 PM

```

nextCDpat:                                * compute next pattern index; return in t
%
This routine increments CDpatx, the pattern control variable that determines which pattern the routine
getCDPat will generate. For simple patterns like cycled 1 and cycled 0, nextCDpat also generates the
value for the current pattern. The routine getCDPat computes the values for the more complicated
patterns.
pattern1 IN [0..pat1EndC)                    = cycled 1
pattern2 IN [CDpat1EndC..CDpat2EndC)        = cycled 0
pattern3 IN [CDpat2EndC..CDpat3EndC)        = cycled va
pattern4 IN [CDpat3EndC..CDpat4EndC)        = pseudo random numbers
%

        RBASE_ rbase[CDpatx];                * compute end of patterns as branch condition
        Drlink _ link;                       * save return link
        top level;
        t_CDpatx_(CDpatx)+1;
        RBASE _ rbase[defaultRegion];        * keep it simple.
        t-(CDpat2EndC);                      * see if IN [0..pat1EndC)
        branch[nxtCDPat3, alu>=0];          * goto[nextPat3, ~IN [0..pat2EndC]]

* IN Pattern1 or Pattern2. We'll use nextSpat to do the work.
        call[nextSpat];
        branch[nxtCDPatXit];

nxtCDPat3:                                * This pattern also gets generated
% by the nextSpat code. However, we're not interested in the full and glorious alternatives of
pattern 3 (it is a left cycled va which gets left cycled 28 times).
%
        t - (CDpat3EndC);                    * if we're done with the local
        branch[nxtCDPat4, ALU>=0];          * version of pattern 3, try next one
        noop;
        call[nextSpat];
        branch[nxtCDPatXit];

nxtCDPat4:                                * this pattern consists of pseudo randoms
% that are generated by nextSpat, getSpat. So far, this subroutine has "secretely" called nextSpat
and everything worked. Unfortunately there more pattern3 values for the nextSpat than nextCDpat
really needs. Consequently, here we notice if we've just begun nextCDpat's pattern4. In that case,
we set nextSpat's SpatX to the proper value to begin pattern 4.
%
        t - (CDpat4EndC);                    * see if we've just entered pattern4
        skipif[ALU#0], t _ Spat3EndC;
        SpatX _ t;                           * diddle nextSpat's index. ugly.
        noop;                                * for placement
        call[nextSpat];
        branch[nxtCDPatXit];

nxtCDPatXit:
        subroutine;
        RBASE _ rbase[Drlink];
        link _ Drlink;                       * restore return link, fix up rbase,
        t _ CDpatX, RBASE _ rbase[defaultRegion];
        return,t-(CDlastPatC);              * return w/ proper branch condition

```

* August 13, 1979 5:25 PM

CACHE DATA pattern generator

getCDPat:

* compute next pattern; return it in T

%

This code actually does much of it's work by calling getSpat, the storage pattern generator. It was easier (and less microcode) to do it this way. However, nextSpat's pattern3 left cycles a 28 bit virtual address and uses that cycled value as a pattern. Naturally the cache test is not interested in doing things that way, consequently we depend upon **nextCDpat** to change nextSpat's value in SpatX.

%

```

RBASE_ rbase[CDpatx];
Drlink _ link;
top level;
t _ CDpatX, RBASE _ rbase[defaultRegion];
t-(CDpat4EndC);          * see which pattern we are using
branch[getCDP5,alu>=0];
noop;                    * for placement
call[getSpat];
branch[getCDPxit];

```

getCDP5:

* add code for pattern 5 here

branch[getCDPxit];

getCDPxit:

```

RBASE _ rbase[Drlink];
link _ Drlink;
subroutine;
return, RBASE_ rbase[defaultRegion];

```

* December 12, 1978 11:25 AM

iDboard:

```
pushReturn[];
t _ FaultInfo';
call[setMbase], t _ r0;
rscr _ t-t;
call[setBR], rscr2 _ t-t;
t _ r0;
call[setMCR];
```

```
returnP[];
```

* init the dBoard

* clear out waiting errors
* use membase 0, br=0

* reset mcr to allow everything

* December 21, 1978 8:46 PM

```

top level;                                * assure that midas subroutines really work
noop;                                     * for placement
call[cdRead]; call[cdWrite]; call[cacheAforVa];
noop;
subroutine;

cdRead:                                * T = va. This subr to be used from Midas
fetch _ t;
noop;
rscr2 _ mdr;
return;                                   * make it possible to return

cdWrite:                               * T = va, rscr = data
store _ t, DBuf _ rscr;
noop;
return;                                   * make it possible to return

colForVa:                             * given a va, return its cache column
* T = 16 bit va. The 4 lsb pertain to munch, then skip the bits in the row then use 2 bits

return, t _ ldf[t,2,add[nBitsInRow, cacheShift]]; * extract 2 bits for column

cacheAforVa:                           * T = va. Return the midas style "cache" addr
* for va. This subr to be used from Midas
pushReturn[];
rscr _ cacheRowMask0;                    * make mask = size of cache row bits
rscr _ (rscr) OR (cacheRowMask1);
rscr _ (rscr) OR (3c);                   * add mask for bits addressed by column
t _ ldf[t, nBitsInRow, cacheShift];     * isolate cache row bits
t _ t AND (rscr);                       * isolate cache addr whose size is
* nBitsInRow + 2 (2 bits for the four columns)
returnP[];

presetCache:                           * t = hi8, rscr = low 16, rscr2 = CFlags
pcHi8 _ t;
pushReturn[];
va _ rscr;                               * save va
t _ rscr2;
pcFlags _ t;                             * save Cache flags

call[iMunchCtrl], t _ va;

presetL:
call[nextMunch];                         * SETS VA DIRECTLY!
skpif[alu#0];                            * see if more to do
branch[presetTag];
noop;

call[pcSetCflags];
call[pcSetAmemory];

branch[presetL];                         * loop again

presetTag:
t _ (va) - (100C);                       * we know this address is in the cache
call[resetTag];                          * now cause the tag to get reset

returnP[];

pcSetCflags:                          * private subroutine to presetCache
t _ link;                                * inits CFLAGS for "va" to contain value
Drlink _ t;                              * of pcFlags. This subr MUST NOT CALL
top level;                               * another subr that stores link in Drlink!

call[colForVa], t_va;                    * Set mcr to select appropriate row, column
call[makeUseMcrV];                       * for this va
t _ t OR (mcr.fdmMiss);
t _ t OR (mcr.noRefHold);
call[setMCR];

call[pcGetCFlags];
t _ t # (cflags.mask);                   * invert only the cflags bits

```

```

t_(va) - t; * compute value for BR
rscr2 _ t;
call[setBR], rscr _ (rscr)-(rscr); * BRLO = va-cflags, BRHI = 0

call[pcGetCFlags]; * now set the cache flags
t _ t # (cFlags.mask); * invert only the cflags bits
dummyRef _ t;
CFLAGS _ t;

subroutine;
returnUsing[Drlink];

pcSetAmemory: * private subroutine to presetCache
t _ link; * inits cache A memory to "contain" specific
Drlink _ t; * va. Sets BRHI to value of pHi8! This subr
top level; * MUST NOT CALL another subr that stores
* link in Drlink!

call[colForVa],t_va; * set MCR for current column.
col _ t;
t _ mcr.fdMiss;
t _ t OR (mcr.noRef);
call[mcrForCol];

call[pcGetHi8];
rscr _ t;
call[setBR], rscr2 _ (rscr2) - (rscr2); * init BRHI = pcGetHi8

t _ r0; * perform the reference
STORE _ va, DBuf _ t;

subroutine;
returnUsing[Drlink];

%
Zero the cache. presume it has been init'd for va IN [0..3777B]
Clobber T (that all!)
%
* May 21, 1981 11:44 AM This routine was modified to provide for setting the cache to an arbitrary
value.

zeroCache0:
pushReturn[];
top level;
branch[setCache], t_ a0;

setCache0:
stkp+1;
stack_ link;

setCache:
rscr_ t; * value to use is in rscr

call[iCDvaCtrl], t_r0;

zcVal: * FOR va IN [0..3777B] DO
call[nextVa];
skpif[alu#0], t_rscr; * exit if required. setup t=0
branch[zcXit];

branch[zcVal],DBuf _t,store_va; * cache[va] _ value

zcXit:
returnP[];

```

```
* October 14, 1978 3:13 PM
  subroutine;
pcGetHi8:
  RBASE _ rbase[pcHi8];
  return, t _ pcHi8, RBASE _ rbase[defaultRegion];

pcGetCflags:
  RBASE _ rbase[pcFlags];
  return, t _ pcFlags, RBASE _ rbase[defaultRegion];

knowRbase[defaultRegion];
  top level;

memSubrsDDone:                                ;
```


title[memSubrsS];
top level;
subroutine;

Table with 2 columns: Routine and Description. Lists various routines like iSmunchCtrl, iSvaCtrl, etc. with their descriptions.

%
June 17, 1981 9:31 AM
Move sPingPong, dirtyWriteLoop and doScheckOut into memMisc's memDesperateA.mc
June 16, 1981 10:59 AM
Add call to xGetMapICs in iSboard
June 9, 1981 11:12 AM
Init memState from within iSboard.
May 20, 1981 5:18 PM
Fix problems in sPingPong
May 18, 1981 2:12 PM
Change nextSPat to cycle random number seed.
May 14, 1981 4:51 PM
Miscellaneous fixes to make microcode smaller.
July 1, 1979 3:39 PM
Cause doScheckOut to set mcr.noWake before it tries to load Brs.
July 1, 1979 3:04 PM
Fix doScheckOut to vacate the cache where required (code was buggy) & add scope loops.
June 28, 1979 11:20 AM
Move sGetConfig into memSubrsX and rename it.
June 27, 1979 6:59 PM
Force sBrHi and sBrLo to be zero as default in doScheckout, remove mcr.disCF from default mcr value.
June 26, 1979 4:43 PM
Make another change to doScheckOut.
June 25, 1979 10:14 AM
Change doScheckOut as per requests (fixes several bugs, thnx to k pier).
June 7, 1979 2:07 PM
Fix bug in sex of skip instruction in sGetICtype.
May 7, 1979 2:16 PM
Update sGetICtype to Rev C standards for Config.
January 17, 1979 9:12 AM
remove saSetMcr entirely: setting mcr now independent of setting sMCRvictim
January 16, 1979 2:32 PM
change MCRvictim handling: sMCRvictim may be set by any routine and that value becomes the one

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

sUseDefaultMcr uses (ie., really causes it to occur in MCR). However, the test-wide value (ie., the one set by program default or by the operator) is kept in a separate place, and *that* value may be restored by calling **sRestoreMcrVictim**.

January 15, 1979 2:58 PM

reenable **dirtyWriteLoop**, **sPingPong**

January 13, 1979 12:43 PM

procSRN_r0 inside **iSboard**, remove **dirtyWriteLoop**, **sPingPong**: get memA to place

January 12, 1979 3:17 PM

new **sGetConfig** that notices that storage begins w/ Module 1,2, or 3, pattern3 diddles

%

knowRbase[defaultRegion];

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

* April 23, 1978 9:56 PM

```

iSmunchCtrl: subroutine;
* CLOBBER T, RSCR, RSCR2

    saveReturn[Srlink];

* requires rewriting the Map as well AND that
    sVaHiX _ t_ al;
    t _ (r0)-(20c);
    svaX _ t;
    rscr _ (rscr) - (rscr);
    call[setBR], rscr2 _ (rscr2) - (rscr2);
    returnUsing[Srlink];

iSvaCtrl: subroutine;
* CLOBBER T, RSCR, RSCR2

    saveReturn[Srlink];

* requires rewriting the Map as well AND that
    sVaHiX _ t_ al;
    svaX _ t;
    rscr _ (rscr) - (rscr);
    call[setBR], rscr2 _ (rscr2) - (rscr2);
    returnUsing[Srlink];

iSvaDownCtrl: subroutine;
    saveReturn[Srlink];
    RBASE _ rbase[sMaxBrHi];
    t_sMaxBrHi;
    svaX _ t-t;
    sVaHiX _ t, RBASE _ rbase[defaultRegion];
    rscr _ t;
    call[setBR], rscr2 _ t-t;
    returnUsing[Srlink];

iSmunchDownCtrl: subroutine;
    saveReturn[Srlink];
    t _ (r0)-(20c);
    svaX _ t;
    RBASE _ rbase[sMaxBrHi];
    t_sMaxBrHi;
    sVaHiX _ t, RBASE _ rbase[defaultRegion];
    rscr _ t;
    call[setBR], rscr2 _ t-t;
    returnUsing[Srlink];

restoreBrHi: subroutine;
    saveReturn[Srlink];
    RBASE _ rbase[sVaHiX];
    t _ sVaHiX, RBASE _ rbase[defaultRegion];
    rscr _ t;
    call[setBR], rscr2 _ t-t;
    returnUsing[Srlink];

iSpatCtrl: subroutine;
    Spatx _ t_ al;
    return, curSPattern _ t;

isChaosX: subroutine;
    return, sChaosX _ t_ al;

nextChaosX: subroutine;
    RBASE _ rbase[sChaosX];
    t _ sChaosX_(sChaosX) + 1, RBASE _ rbase[defaultRegion];
    return, t - (sChaosEndC);
    * alu=0 IF last pattern

getSsubrScr: subroutine;
    RBASE _ rbase[sSubrScr];
    return, t _ sSubrScr, RBASE _ rbase[defaultRegion];

```

* initialize 24 bit va loop control

* NOTE: this code (down ctrl, too) won't work properly when va[0:1]#0. THAT

* may present other problems
* begin w/ 0-munchSize

* initialize 24 bit va loop control

* NOTE: this code (down ctrl, too) won't work properly when va[0:1]#0. THAT

* may present other problems

* restore current BRHI after its been clobbered

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

```
getSsavedHoldValue: subroutine;  
  RBASE _ rbase[SsavedHoldValue];  
  return, t _ SsavedHoldValue, RBASE _ rbase[defaultRegion];
```

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

* June 16, 1981 10:58 AM

%

Initialize the Storage Boards

Init the storage boards in preparation for storage diagnostics. Turn on or off error correction as required by memState.useTestSyn. Simulate a cache w/ only one column, as required by sMCRvictim (RM location). This subroutine also presets the map, clears the cache flags and determines the current memory configuration.

%

```

iSboard: subroutine;
  saveReturn[Srlink];

  t _ FaultInfo'[];          * remove any waiting wakeups

  call[iMemState];

  call[xGetConfig];
  call[xGetMapICs];
  call[clearCacheFlags];    * make the cache vacant
  call[presetMap];          * initialize the map
  call[clearCacheFlags];    * make the cache vacant

  t _ memState.useTestSyn;
  call[checkMemState];
  skipif[alu=0], t _ t-t;    * don't use error correction
  t _ (200c);                * use error correction
  call[setTestSyn];          * causes cache to be non-vacant
  call[clearCacheFlags];    * make the cache vacant

  rscr _ t-t;                * set our base register to zero
  call[setBR], rscr2 _ t-t;

  call[saNoWakeOn];          * default = set mcr.noWake
  call[sRestoreMcrVictim];  * get MCRvictim (>3 => set MCR w/ 0)
  procSRN _ r0;
  returnUsing[Srlink];

```

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

* January 17, 1979 9:31 AM

```

%
getSmcrVictim      returns t = sMCRvictim
sRestoreMcrVictim resets sMCRvictim from memState, sets MCR
sUseMcrVictim     sets sMCRvictim, memState
sUseDefaultMcr    sets MCR w/ current sMCRvictim
%
getSmcrVictim: subroutine;
    RBASE _ rbase[sMCRvictim];
    return, t _ sMCRvictim, RBASE _ rbase[defaultRegion];
sRestoreMcrVictim: subroutine;
    pushReturn[];
    call[getmemState];
    rscr _ rsh[t, memState.mcrVictimShift];
    t and (memState.usingOneColumn);
    skipif[ALU=0];
    skip, t _ (rscr) and (3c);
    t _ 7c;
    sMCRvictim _ t;
    call[sUseDefaultMcr];
    returnP[];
%

```

sUseMcrVictim

This routine "permanently" sets the mcrVictim used the the diagnostics. This means that every time **isboard** gets called or every time the operator invokes **sRestoreMcrVictim**, **sMCRvictim** will be set to select the indicated column of the cache (or to allow full use of the cache). Any given piece of code may change the value of **sMCRvictim**, an r-register, to temporarily cause the cache to behave as if it were only one column wide. Once that code has done this, that value of **sMCRvictim** will continue to be used by **sUseDefaultMcr** until **sRestoreMcrVictim** gets called.

```

%
sUseMcrVictim: subroutine;
    pushReturnAndT[];
    (stack) and (not[3]C);
    skipif[ALU=0];
    branch[sUseMcrVicOff];

* set sMCRvictim with a "non-null" value. This means we set it to a value IN [0..3]
    t _ (stack) and (3c);
    t _ lsh[t, memState.mcrVictimShift];
    rscr _ t or (memState.usingOneColumn);
sUseMcrVicSet:
    t _ (stack);
    sMCRvictim _ t;
    t _ rscr;
    stack+1 _ t;
    call[getMemState];
    rscr _ (add[memState.usingOneColumn!, memState.mcrVictim!]C);
    rscr _ not(rscr);
    t _ t and (rscr);
    t _ t or (stack&-1);
    call[putMemState];
    pReturnP[];
sUseMcrVicOff:
    rscr _ t-t;
    branch[sUseMcrVicSet];

sUseDefaultMcr: subroutine;
    pushReturn[];
    call[getSmcrVictim];
    t and (not[3]C);
    dblBranch[sUseDefaultVic, sUseDefaultAll, ALU=0];

*BEWARE: uses Smlink; clobbers rscr, rscr2, t
* get MCRvictim (>3 => set MCR w/ 0)
* see if we're using a victim

sUseDefaultAll:
    branch[sUseDefaultSet], t _ t-t;

sUseDefaultVic:
    noop;
    call[makeUseMcrV];

* for placement.
* enter w/ t = column, rtns t = mcr value

sUseDefaultSet:
    call[saOrMcrNoWake];
    call[setMCR];
* OR proper noWake into current mcr val.

```

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

```
returnP[];
```

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

* January 16, 1979 3:08 PM

* enter w/ t = mcr bits

* return w/ t = mcr bits OR (correct value for mcr.noWake -- same as memState.noWake)

```

saOrMcrNoWake: subroutine;
  saveReturnAndT[saOrMcrNoWakeRtn, sSubrScr];
  call[checkMemState], t _ (memState.noWake);
  skipif[alu=0], rscr _ t-t;
  rscr _ mcr.noWake;
  call[getsSubrScr];
  t _ t or (rscr);
  returnUsing[saOrMcrNoWakeRtn];

```

* January 5, 1979 2:34 AM

* RTN ALU#0 IF error checking is turned off

* applies to storage data test

```

sChkNoErrs: subroutine;
  saveReturn[sRlink];
  call[checkMemState], t _ memState.noErrs;
  returnAndBranch[sRlink, t];

```

```

sNoErrsOn: subroutine;
  saveReturn[sRlink];
  call[getMemState];
  t _ t OR (memState.noErrs);
  call[putMemState];
  returnUsing[sRlink];

```

```

sNoErrsOff: subroutine;
  saveReturn[sRlink];
  call[getMemState];
  rscr _ not(memState.noErrs);
  call[putMemState], t _ t and (rscr);
  returnUsing[sRlink];

```


DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

* January 27, 1978 11:30 AM

%

```

nextSva: PROCEDURE RETURNS [va: Low16BitsOfVa, validity: BOOLEAN]
  BEGIN
  RETURN[incSva[inc: 1]];
  END;
nextSvaDown: PROCEDURE RETURNS[va: Low16BitsOfVa, validity: BOOLEAN]=
  BEGIN
  RETURN[ incSva[inc: -1] ];
  END;
nextSmunch: PROCEDURE RETURNS [va: Low16BitsOfVa, validity: BOOLEAN] =
  BEGIN
  RETURN[incSva[inc: 16]];
  END;
incSva: PROCEDURE[increment: CARDINAL] RETURNS[va: Low16BitsOfVa, validity: BOOLEAN] =
  BEGIN
  oldLow _ low;
  low _ low + increment;
  signChange _ (oldLow xor low) AND (b0);
  IF signChange THEN
    IF increment >0 AND (( low BITAND b0) = 0) THEN
      -- overflow if msb is zero
      hiVa _ hiVa + 1;
      IF hiVa = lastVa THEN RETURN[nil, FALSE];
      SetBr[hiVa, 0];
      END;
    IF increment <0 AND ((low BITAND b0) # 0) THEN
      -- underflow if msb is 1
      hiVa _ hiVa -1;
      IF hiVa <0 THEN RETURN[nil, FALSE];
      SetBR[hiVa,0];
      END;
  RETURN[low, TRUE];
  END;

```

%

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

```

* October 16, 1978 10:06 AM
knowRbase[defaultRegion];
nextSva: subroutine; * return t = low 16 bits of next va.
* automatically set BRHI. Clobber T, RSCR, RSCR2
  saveReturn[Srlink];
  t _ 1c;
  call[incSva]; * REMEMBERED the ALU=0 condition in rscr

  returnAndBranch[Srlink, rscr]; * use Srlink for return, use val of rscr for fastbr

nextSmunch: subroutine; * return t = low 16 bits of next va.
* automatically set BRHI. Clobber T, RSCR, RSCR2
  saveReturn[Srlink];
  t _ 20c;
  call[incSva]; * REMEMBERED the ALU=0 condition in rscr

  returnAndBranch[Srlink, rscr]; * use Srlink for return, use val of rscr for fastbr

nextSvaDown: subroutine; * return t = low 16 bits of next va.
* automatically set BRHI. Clobber T, RSCR, RSCR2
  saveReturn[Srlink];
  t _ cml;
  call[incSva]; * REMEMBEREDED the ALU=0 condition in rscr

  returnAndBranch[Srlink, rscr]; * use Srlink for return, use val of rscr for fastbr

incSva: subroutine; * Enter w/ T = increment to sVaX
* return t = low 16 bits of next Sva, alu=0 result in rscr
* is fast branch condition, clobber T, RSCR, RSCR2. call setBR if required.

  saveReturnAndT[incSvaRtn, SsubrScr];

  RBASE _ rbase[sSubrScr];
  t _ sSubrScr;
  SvaX _ (SvaX)+t; * increment SvaX

  t _ (SvaX)-t; * compute signChange
  t _ t # (SvaX);
  t _ t and (B0);
  branch[incNormal,alu=0], sSubrScr _ sSubrScr; * IF signChange THEN ...
  branch[incOverFlChk, alu>=0]; * IF increment >=0 THEN

incUnderFlChk:
  (SvaX) and (b0);
  skipif[alu#0]; * IF SvaX AND b0 =0
  branch[incNormal];

  t _ sVaHiX _ (sVaHiX) -1; * THEN decrement sVaHiX
  dblbranch[incRtnDone, incSetBr, alu<0];

incOverFlChk:
  (SvaX) and (b0);
  skipif[alu=0]; * IF SvaX AND b0 # 0
  branch[incNormal];

  t _ sVaHiX _ (sVaHiX)+1; * THEN increment sVaHiX
  t-(sMaxBrHi);
  dblbranch[incRtnDone, incSetBr, alu=0];

incSetBr:
  RBASE _ rbase[defaultRegion];
  rscr _ t;
  call[setBR], rscr2 _ (rscr2) - (rscr2); * set BRHi
  noop;

incNormal:
  subroutine;
  RBASE _ rbase[incSvaRtn];
  link _ incSvaRtn;
  RBASE _ rbase[svaX];
  t _ svaX, RBASE _ rbase[defaultRegion];
  return, rscr _ r1; * return w/ t=lowva, alu result #0

```

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

incRtnDone:

```
subroutine;
RBASE _ rbase[incSvaRtn];
link _ incSvaRtn;
t _ t-1, RBASE _ rbase[defaultRegion]; * return w/ t undefined,
return, rscr _ t-t; * return w/ t undefined, alu result =0
```

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

* May 18, 1981 2:11 PM

nextSpat: * compute next pattern index; return in t

%

This routine increments SpatX, the pattern control variable that determines which pattern the routine getSPat will generate. For simple patterns like cycled 1 and cycled 0, nextSpat also generates the value for the current pattern. The routine getSPat computes the values for the more complicated patterns.

%

* pattern1 IN [0..pat1EndC) = cycled 1
 * pattern2 IN [Spat1EndC..Spat2EndC) = cycled 0
 * pattern3 IN [Spat2EndC..Spat3EndC) = cycled 32 bit va (only 16 bits returned)
 * pattern4 IN [Spat3EndC..Spat4EndC) = pseudo random numbers

* Return w/ end of patterns as branch condition ("ALU=0" means end of patterns)

RBASE_ rbase[rmForStoreLoops];
 SRlink _ link; * save return link
 top level;
 t_Spatx_(Spatx)+1; * SpatX in T while finding current pattern
 t-(Spat1EndC); * see if IN [0..pat1EndC)
 branch[nxtSPat2, alu>=0],t_t; * goto[nextPat2, ~IN [0..Spat1EndC)]

* IN [0..pat1EndC)
 skipif[alu#0]; * see if just initialized
 skip, curSPattern _ lc; * patX=0 ==> begin with one
 curSPattern_(curSPattern)+(curSPattern);
 branch[nxtSPatXit],t_Spatx;

nxtSPat2:

t-(Spat2EndC);
 branch[nxtSPat3, alu>=0]; * goto[nxtSPat3, ~IN [0..Spat2EndC)

* IN [Spat1EndC..Spat2EndC)
 * pattern2 = left cycled 0
 curSPattern _ lcy[curSPattern, curSPattern, 1];
 t - (Spat1EndC); * If first time thru pattern 2, init
 skipif[ALU#0]; * curSPattern for left cycled zero pattern.
 curSPattern _ not(b15); * curSPattern _ 177776 on first time thru
 branch[nxtSPatXit];

nxtSPat3:

t - (Spat3EndC);
 branch[nxtSPat4, alu>=0]; * goto[nxtSPat4, ~IN[0..Spat3EndC)]

* IN [Spat2EndC..Spat3EndC)
 * pattern3 = cycled 32bit va (use low 16 bits)
 branch[nxtSPatXit]; * pattern3 (cycled va) done by getSPat.

nxtSPat4:

t - (Spat4EndC);
 branch[nxtSPatXit, alu>=0];

* IN [Spat3EndC..Spat4EndC)
 * pattern4 = pseudo random numbers
 noop;
 call[cycleRandV];
 RBASE_ rbase[SRLink]; * pattern4 (random numbers) done by getSPat
 branch[nxtSPatXit];

nxtSPatXit:

link _ SRlink; * restore return link, fix up rbase,
 subroutine;
 RBASE_ rbase[defaultRegion];
 return,t-(SlastPatC); * return w/ proper branch condition

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

* January 12, 1979 6:38 PM

STORAGE Test General Subroutines

getSPat:

* Enter w/ T = VA. rtn w/ next pattern in T

%

Currently there are four sets of patterns:

pattern1	left cycled 1 bit
pattern2	left cycled 0 bit
pattern3	left cycled va
pattern4	random numbers

Pattern1 and **pattern2** return the same value every time getSPat is called, given a constant value for SpatX. Ie., the range for SpatX in pattern 1 is [0..Spat1EndC), and when SpatX is zero, getSPat always returns 1, when SpatX is one, getSPat always returns 2, etc.

Pattern2 is similar to pattern1 except that it is a bitwise complement, ie., a single zero bit with the remaining bits all ones.

Pattern3 returns the current va (treated as a 32 bit number) left cycled. The calling program provides the current va (low 16 bits only) in T. The first time pattern3 is in effect, it returns the current va. The next time **nextSpat** is called, the cycle count for pattern 3 increments and **getSpat** returns the va left cycled 1, etc. When the current va is even, it returns the low 16 bits of svaHiX,,currentSva (appropriately cycled) and when it is odd it returns the low 16 bits of currentSva,,sVaHiX (appropriately cycled).

Pattern4 returns a random number.

%

mc[shc.AisT, b2];

mc[shc.BisT, b3];

set[shc.countShift, 10];

mc[shc.maskShiftCount, b4,b5,b6,b7]

mc[shc.rmt, shc.BisT!];

* deal w/ rm,,t

RBASE_ rbase[rmForStoreLoops];

SRlink _ link;

top level;

(Spatx)-(Spat1EndC);

branch[getSP2,ALU>=0];

* see which pattern we are using

* IN [0..Spat1EndC)

* pattern 1 returns a cycled 1 bit.

branch[getSPXit], t_curSPattern;

* nextSpat did our work

getSP2:

(Spatx)-(Spat2EndC);

branch[getSP3, ALU>=0];

* IN [Spat1EndC..Spat2EndC)

* pattern 1 returns a cycled 0 bit.

branch[getSPXit], t _ curSPattern;

* nextSpat did our work

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

* January 12, 1979 6:48 PM

getSP3:

```
(SpatX) - (Spat3EndC);
branch[getSP4, ALU>=0];
```

* IN [Spat2EndC..Spat3EndC)

%

Pattern3 returns a cycled version of the current va.

If current va is even,

```
return the left cycle of sVaHiX,,currentVa;
```

otherwise,

```
return the left cycle of currentVa,,sVaHiX.
```

Begin by moving the current va into curSPattern, and then construct a shifter control value that will provide the correct cycle value. Always construct a SHC value that cycles rm,,t.

%

```
curSPattern _ t; * ENTERED W/ T = low 16 bits of va
```

```
t _ shc.rmt; * construct SHC constant that will
sSubrScr _ SpatX; * left cycle rm,,t by the correct amount
sSubrScr _ (sSubrScr) - (Spat2EndC); * Spat2EndC is the first valid index for this pattern
sSubrScr _ lsh[sSubrScr, shc.countShift];
sSubrScr _ (sSubrScr) and (shc.maskShiftCount);
t _ t or (sSubrScr); * now we have the correct shc value
SHC _ t;
```

```
t _ curSPattern; * we'll cycle rm,,t
```

```
t _ shiftNoMask, sVaHiX; * = lcy[sVaHiX, t, <current cycle count>]
```

```
branch[getSPxit], curSPattern _ t;
```

```
skpif[R ODD], curSPattern; * decide which half of a cycled va to use
branch[getSP3even];
```

getSP3odd: * lcy[currentVa,,sVaHiX, <current cycle count>]

```
t _ sVaHiX;
```

```
t _ shiftNoMask, curSPattern; * lcy[rm,,t, <current shift>];
```

```
branch[getSPxit], curSPattern _ t;
```

getSP3even: * lcy[sVaHiX,,currentVa, <current cycle count>]

```
t _ curSPattern;
```

```
t _ shiftNoMask, sVaHiX; * t _ lcy[rm,,t, <current shift>]
```

```
branch[getSPxit], curSPattern _ t;
```

getSP4:

```
(SpatX) - (Spat4EndC);
```

```
branch[getSPxit, ALU>=0];
```

* IN [Spat3EndC..Spat4EndC)

* pattern4 returns pseudo ranom numbers

```
noop;
```

```
getRandom[]; * IN [Spat2EndC..Spat3EndC) (random numbers)
```

```
RBASE _ rbase[SpatX];
```

```
branch[getSPxit], curSPattern _ t;
```

getSPxit:

```
link _ SRlink;
```

```
subroutine;
```

```
return, RBASE_ rbase[defaultRegion];
```

getCurSPattern: subroutine; * return curSPattern in T

```
saveReturn[SRlink];
```

```
RBASE _ rbase[curSPattern];
```

```
t _ curSPattern;
```

```
returnUsing[SRlink];
```

DORADO: memSubrsS.mc June 17, 1981 9:31 AM %

* May 19, 1981 11:28 AM

zeroMemory: subroutine; * zero contents of memory. clobber T, RSCR, RSCR2

```

q_ t;
saveReturn[zeroMemoryRtn];
branch[doSetMemory], t_q;

```

setMemory: subroutine;

```

q_ t;
saveReturn[zeroMemoryRtn];

```

doSetMemory:

```

call[iSmunchCtrl];

```

setMemL:

```

call[nextSmunch]; * returns t = next sva
skipif[alu#0];
branch[setMemLxit];

```

```

cnt _ 17s; * inner loop to zero current munch
rscr _ q;

```

setMemIL:

```

store _ t, DBuf _ rscr;
loopuntil[cnt=0&-1, setMemIL], t _ t +1;

```

```

branch[setMemL];

```

setMemLxit:**setMemLRtn:**

```

returnUsing[zeroMemoryRtn];

```

saveMemAddr: subroutine; * save SvaHi, SvaX

```

t _ Sva, RBASE _ rbase[SvaHiX];
sVaXold _ t;
t _ SvaHiX;
SvaHiOld _ t;
return, RBASE _ rbase[defaultRegion];

```

fetchMemAddr: subroutine; * return SvaXold, SvaHiOld in rscr2, rscr

```

saveReturn[Srlink];
RBASE _ rbase[SvaHiOld];
t _ SvaXold;
rscr2 _ t;
t _ SvaHiOld;
rscr _ t;
returnUsing[Srlink];

```

getSvaXold: * return value of SvaXold in T

```

saveReturn[Srlink];
RBASE _ rbase[SvaXold];
t _ SvaXold;
returnUsing[Srlink];

```

title[memSubrsX];

Table Of Contents
by
Order of Occurrence

ROUTINE	DESCRIPTION
iMrowCtrl	Initialize map row loop control
iMcolCtrl	Initialize map column loop control
iMpatCtrl	Initialize map wait loop control
iMwaitCtrl	Initialize map wait loop control
iMapPageCtrl	Initialize map page loop control
iMpageDownCtrl	Initialize map page control for decrementing addresses
nextMcol	Return t=next map column
nextMrow	Return t=next map row
nextMwait	Return t = next map wait value
nextMpage:	Return t = next map page
nextMpageDown	Return t = next map page, decrementing addresses
getMpage	Return t = current map page
nextMpat	Return t = next map pattern, ALU=0 means no more patterns
getMpat:	Return t = current map pattern.
resetMap	Initialize the hardware map
setTestSyn	Set testsyndrome from T
presetMap	Initialize map beginning at page 0
getXmapBase	Return t = xMapBase
testMap	Midas subroutine for testing the map
resetTag	Kick start the tag bits
xVacateCacheRow	Vacate cache row for t = va
readMap	Read map entry
writeMap	Write a map entry
waitForMapT	wait for mapbuf busy to go away. Clobber T
setBrforMap:	Set Br to reference a map entry, given Mrow and Mcol
xSetBRforPage	Set BR to point to map page in T
xWriteMapPage	Write an entry into map, t = virtual page index, rscr = real page
xZeroMap	Zero the entire map
xGetConfig	Get current memory system configuration.
xGetMapICs	Init xPageEndHi, xPageEndLo, 32bit lastpage val
XgetSnModules	Return sNmodules
xGetICtype	Return t = icType = BrHi offset per module
xCountModules	Return t = num storage modules; set sNmodules, too
getMsubrScr	sReturn t = mSubrScr
xGetPipe4:	return pipe4 with all inverted bits hi true
xBoardLoop	Midas subroutine to exercise map
xRWDmux	Write and read a dmux value, rtn t=ALUFM value =PD(branches will work)

May 14, 1981 4:52 PM

Misc changes to make code smaller.

February 10, 1981 11:30 AM

Fix bugs in setBrForMap; fix bugs in nexMpage, nextMpageDown.

February 9, 1981 9:52 AM

Change map Page iterators to handle 64K, 256K ics, add xGetMapICs to read mufflers and findout map IC size. Change getMPat to keep row,,col in separate bytes rather than a field = size of ras addr.

July 1, 1979 2:50 PM

change xVacateCacheRow to use putCFmem, cause WriteMap to keep its data on Bmux one cycle earlier.

June 28, 1979 11:17 AM

Move sGetConfig into this file -- to centralize location of procedures required for memory system initialization. Add table of contents

May 29, 1979 2:39 PM

Set mcr w/ disHold, noWake during xZeroMap.

January 26, 1979 3:17 PM

Add B_FaultInfo' to resetMap to turn off any faults that may have occurred.

January 25, 1979 5:41 PM

Add xGetPipe2, xGetNumFlts.

January 12, 1979 5:05 PM

cause presetMap to add xMapBase to the real page number when writing map (handle situation wherein there's no module 0, etc.)

SUBROUTINE;


```
knowRbase[defaultRegion];

iMrowCtrl:
    return, MrowX _ t_ al;
iMcolCtrl:
    return, Mcolx _ t_ al;
iMpatCtrl:
    t_cml;
    Mpatx _ t;
    t _ curMpatLow _ t-t;
    return, curMPatHi _ t;
iMwaitCtrl:
    t _ (r0)-(MwaitIncrC);
    return, curMWait _ t;
iMapPageCtrl:
    xPageXLo_ t_ al;
    return, xPageXHi_t;
iMpageDownCtrl:
    RBASE_ rbase[xEndPageLo];
    xPageXLo _ xEndPageLo;
    xPageXHi _ xEndPageHi;
    return, RBASE_rbase[rscr];
```

* February 10, 1981 4:24 PM

knowRbase[rmForMapLoops];

```

nextMcol: subroutine;                                * compute next column; return it in T
* Mcol IN [0..McolEndC)

    RBASE_ rbase[Mcolx];                                * compute end of columns as branch condition
    t_Mcolx_(Mcolx)+1, RBASE_ rbase[xChipEndRasCas];
    return, t-(xChipEndRasCas), RBASE_ rbase[defaultRegion];

nextMrow: subroutine;                                * compute next row; return it in T
* MrowX IN [0..MrowEndC). Mrow _ MrowX.
*WRITE DIRECTLY INTO DESTIN REGISTER, Mrow.
* use this hack 'cause we can't switch rbase and then perform the exit test

    RBASE_ rbase[rmForMapLoops];                        * compute end of row as branch condition
    t _ Mrowx_(Mrowx)+1;
    Mrow _ t;                                           * WRITE DIRECTLY INTO Mrow

    t _ Mrowx, RBASE_ rbase[xChipEndRasCas]; * sigh. put loop ctrl where we can access it.
    return, t-(xChipEndRasCas), RBASE_ rbase[defaultRegion];

nextMwait: subroutine;                               * compute next wait return it in T
* curMwait IN [0..MwaitEndC), incremented by MwaitIncrC

    RBASE_ rbase[rmForMapLoops];                        * compute end of columns as branch condition
    t_ curMwait _ (curMwait) + (MwaitIncrC);
    RBASE_ rbase[defaultRegion];
    return,t-(MwaitEndC);

nextMpage: subroutine;                               * compute next map page (increment by 1)
    RBASE _ rbase[xPageXLo];
    t _ xPageXLo _ (xPageXLo) + 1;                       * increment page number
    t-(xEndPageLo), RBASE_ rbase[defaultRegion];
    skipif[ALU=0], FreezeBC;
    return, FreezeBC;                                   * return w/ fast branch condition

    RBASE_ rbase[xPageXHi];
    t_ xPageXHi_ (xPageXHi)+1;                           * must increment hi value.
    t-(xEndPageHi);                                     * check for done, too.
    t_ xPageXLo, FreezeBC;
    skipif[ALU=0], RBASE_ rbase[defaultRegion];
    return, PD_ al;                                     * more to do, alu#0
    return, PD_ a0;                                     * done. alu=0

nextMpageDown: subroutine;                           * rtn w/ ALU=0 => no more pages.
    RBASE _ rbase[xPageXLo];
    t _ xPageXLo _ (xPageXLo) - 1;
    t # (cml);                                           * stop when underflow taskes us to -1??
    skipif[ALU=0];
    return, PD_ al, RBASE _ rbase[defaultRegion];

    knowRbase[xPageXLo];
    xPageXHi_ (xPageXHi)-1, RBASE_ rbase[defaultRegion];
    skipif[alu<0];
    return, PD_ al;
    return, PD_ t-t;                                     * rtn ALU=0 => no more pages to write

getMpage: subroutine;                               * return current map page without incrementing
    RBASE _ rbase[xPageXLo];
    return, t _ xPageXLo, RBASE _ rbase[defaultRegion];

```

```

* February 9, 1981 10:20 AM
nextMpat: subroutine;
* pattern1 IN [0..pat1EndC)
* pattern2 IN [Mpat1EndC..Mpat2EndC)

    pushReturn[];
    RBASE _ rbase[MpatX];
    t_MpatX_(MpatX)+1;
    t-(Mpat1EndC);
    branch[nxtMpat2, alu>=0],t_t;

* IN [0..pat1EndC)
    skipif[alu#0];
    skip, t _ curMPatLow _ 1c;
    t _ curMPatLow _ (curMPatLow) lsh 1;
    MpatLow _ t;
    (MpatX) - (20c);
    skipif[ALU#0], t _ curMpatHi _ (curMpatHi) lsh 1;
    t _ curMpatHi _ 1c;
    MpatHi _ t;
    branch[nxtMpatXit], t _ MpatX;

nxtMpat2:
    t-(Mpat2EndC);
    branch[nxtMpat3, alu>=0];

* IN [Mpat1EndC..Mpat2EndC)
    curMpatHi _ t-t;
    branch[nxtMpatXit], curMPatLow _ t-t;

nxtMpat3:
    branch[nxtMpatXit];

nxtMpatXit:
    link _ stack&-1;
    subroutine;
    RBASE_ rbase[defaultRegion];
    return,t-(MlastPatC);

```

* compute next pattern index; return in t
= cycled 1
= function of current Mrow, Mcol

* compute end of patterns as branch condition

* see if IN [0..pat1EndC)
* goto[nxtPat2, ~IN [0..pat1EndC)]

* see if just initialized
* patX=0 ==> begin with one

* see if we've overflowed into hi bits
* left shift current MpatHi
* we've been left shifting zero. set it to 1

* see if IN[pat1EndC..pat2EndC)
* goto[nxtPat3, ~IN[pat1EndC..pat2EndC)]

* Zero it

* code for next patter goes here

* restore return link, fix up rbase,

* return w/ proper branch condition

* December 13, 1978 1:16 PM

MAP Test General Subroutines

getMPat:

* compute current pattern; return low bits in T

%

Currently there are two sets of patterns. The first set consists of a cycled one. The second pattern produces unique values based on row and column.

pattern1: maintained by the nextPattern code -- it changes when "next pattern" is called. pattern2 changes more frequently (for every column!)

pattern2: OR the quantity (row lshift 3) + column into curPattern.

Eg., if curPattern = 0, row = 22, col = 3 THEN result = 223

%

```

pushReturn[];
RBASE _ rbase[MpatX];
(Mpatx)-(Mpat1EndC);          * see which pattern we are using
branch[getMP2,alu>=0];
t _ curMpatHi;
MpatHi _ t;
t _ curMpatLow;
branch[getMPxit], MpatLow _ t;

```

getMP2:

* return pattern 2 or greater

```

(Mpatx)-(mpat2EndC);
branch[getMP3, alu>=0];
t _ curMpatHi;

```

```

RBASE _ rbase[defaultRegion];          * fetch value of Mrow, Mcol from defaultRegion
MpatHi _ t;
t _ Mrow;
MsubrScr _ t;
t _ Mcol, RBASE _ rbase[rmForMapLoops];

```

```

MsubrScr _ lsh[MsubrScr, 10];          * Mrow, Mcol separate bytes
t_t OR (curMpatLow);                  * curPattern,,column
t_t OR (MsubrScr);                    * curPattern,,row,,column
branch[getMPxit], MpatLow _ t;

```

getMP3:

* add code for pattern 3 here

```

branch[getMPxit];

```

getMPxit:

```

link _ stack&-1;
subroutine;
return, RBASE_ rbase[defaultRegion];

```

getMwait:

* return current value of map Wait in T

```

RBASE _ rbase[rmForMapLoops];
return, t _ curMwait, RBASE _ rbase[defaultRegion];

```

* January 26, 1979 3:18 PM

resetMap:

* when the machine powers up, the
 * map may be in an arbitrary state.
 * kick-start the automata by performing two fetches. Cope w/ "tag" bit by
 * performing a reference that hits and then punch on the map 8 times to wake it up.

```

pushReturn[];
set[xmcrv, OR[mcr.fdmMiss!, mcr.disHold!, mcr.disCF!, mcr.disBR!, mcr.noWake!]];

t_AND[xmcrv, 177400]C;
t _ t OR (AND[xmcrv, 377]C);
call[setMCR];

fetch _ r0;
t _ 40C;
call[longWait];
fetch _ r0;
t _ 40C;
call[longWait];

call[resetTag], t_r0;

cnt _ 10s;

```

* init MCR w/ virtually everything turned off

* perform the two fetches, separated by waits

* reset our tag bit. use va=0

* punch on the map a bunch of times

resetML:

```

noop;
rscr _ t-t;
call[writeMap], rscr2 _ t-t;
loopWhile[cnt#0&-1, resetML];

t _ mcr.disHold;
t _ t OR (mcr.noWake);
call[setMCR];
B_FaultInfo';

returnP[];

```

* reset mcr for further map testing:
 * use disHold, noWake

* January 1, 1979 3:11 PM

setTestSyn:

```

saveReturnAndT[Drlink, rscr];

t _ 62c;
call[longWait];
t _ mcr.noWake;
call[setMCR];
t _ rscr;
rscr _ t-t;
rscr2 _ t-t;
call[setBR];
STORE _ r0, DBuf _ t;
loadTestSyndrome[t];
t _ 62c;
call[longWait];
t_ FaultInfo';
returnUsing[Drlink];

```

* retrieve original value of t

* February 9, 1981 9:50 AM

presetMap:

```

pushReturn[];
call[resetMap];
t _ FaultInfo';
call[clearCacheFlags];
call[setMCR], t_t-t;

```

* initialize map beginning at page 0

* clear any pending wakeups
 * assure beingLoaded not set in cache
 * clear mcr

```

call[iMapPageCtrl];

```

pmL:

```

call[nextMpage];
branch[pmLxit, alu=0];
noop;

```

* get next page number

```

call[xSetBRforPage];
call[getMpage];
call[getXmapBase], rscr2 _ t;

```

* expects t = page number
 * get page number in t

```
rscr2 _ t + (rscr2);          * Construct offset for situation where
rscr _ t-t;                  * there's no module 0. xMapBase is set by
call[writeMap];              * sGetConfig. Write map w/ page number.
branch[pml];
```

pmLxit:

```
returnP[];
```

getXmapBase: subroutine;

```
RBASE _ rbase[xMapBase];
return, t _ xMapBase, RBASE _ rbase[defaultRegion];
```

* February 9, 1981 9:43 AM

```

testMap:
* RSCR = cycles to wait, RSCR2 #0 => call resetMap
  pushReturn[];
  q _ rscr;
  rscr2 _ rscr2;
  skipif[alu=0];
  call[resetMap];
  call[setBRforMap];
  rscr _ MpatHi;
  rscr2 _ MpatLow;
  call[writeMap];

  rscr2 _ q;
  rscr2 _ (rscr2) - (120C);
  skipif[alu>=0];
  branch[testMapChk];

  rscr2 _ (rscr2) + 1;
  branch[., alu#0],rscr2 _ (rscr2) -1;

testMapChk:
  call[readMap];
  t _ MpatLow;
  t _ t # (rscr);
  skipif[alu=0];

xTestMapErr3:
  error;

  t _ ldf[rscr2, 2, pipe4.dirtyShift];
  t#(MpatHi);
  skipif[ALU=0];

xTestMapErr4:
  error;
  * true bits inverted.

  returnP[];

resetTag:
* T = address to reference.

  pushReturn[];
  rscr _ t;
  set[xmcrv, OR[mcr.noRef!, mcr.disHold!, mcr.disCF!, mcr.disBR!, mcr.noWake!]];
  t _ AND[xmcrv,177400]C;
  t_t or (AND[xmcrv, 377]C);
  call[setMCR];

  DBuf _ rscr, STORE _ rscr;
  call[longWait], t _ 10C;
  fetch _ rscr;

  returnP[];

```

* Mcol = column, Mrow = row, Mpat = pattern,
* params and return link are saved. proceed...
* save cycles to wait in Q for a while
* see if we should call resetMap
* reset and write the map
* now wait if required. retrieve wait count
* subtract 80 cycles for cost of readMap
* fast branch might be zero -- fix it
* WAIT LOOP
* Mrow,Mcol = address. Q = time waited
* Mpat = expected, rscr = pipe3
* rt justify the dirty, wprotect bits from
* the copy of Pipe4 that is in rscr2.
* T= wprotect,dirty from pipe4, MpatHi
* is value of those bits as we wrote them.
* rscr2 contains a copy of pipe4, w/ low
* kick start the tag bit for our task.

* July 1, 1979 2:47 PM

xVacateCacheRow: * T = va. Cause all the columns of the cache entry for
va to be vacant by setting the vacant bit. Use useMcrV to write each column in succession by using
CFLAGS_.

```
pushReturnAndT[];  
cnt _ 3s;
```

xVacateCacheRowL:

```
rscr _ cflags.vacant;  
rscr2 _ cnt;  
call[putCFmem], t _ stack;  
loopUntil[CNT=0&-1, xvacateCacheRowL];  
pReturnP[];
```


* May 14, 1981 3:12 PM

%

Read the map for position row, column and return with rscr = pipe3 and rscr2 = pipe4.

To read the map, vacate the cache entries for the cache row that the va represents. There is one bit of overlap, va[15], between the va bits that address the map column, va[9:15] and the bits that address the cache row, va[15:19]. Pass that bit as a parameter to the vacate CachRow procedure. Set the BR appropriately?

READ MAP TAKES ABOUT 80 CYCLES

%

readMap:

```

pushReturn[];
t _ mcr.disHold;
call[setMcr],t_t OR (mcr.noWake);          * no wakeups please

call[setBrforMap];                         * read map for current row, column
RMap _ r0, call[waitForMapT];              * don't try to get data until it's stable.

rscr _ not(Pipe3');
call[xGetPipe4];                            * use subroutine since pipe4' is complicated

rscr2 _ t;
returnP[];

```

writeMap: subroutine;

= MapBuf data

```

pushReturn[];
call[waitForMapT];
rscr _ lsh[rscr, 16];
tioa _ rscr;
t _ t-t;

```

* ENTER w/ rscr = 2 bits for tio, rt justified, rscr2

writeMapX:

```

B_rscr2;
map_t, MapBuf_rscr2;
call[waitForMapT];

returnP[];

```

* kept it on bmux early

waitForMapT: subroutine;

```

noop;
t _ pipe5;
t _ t and (pipe5.MbufBusy);
branch[.-2, alu#0];
return;

```

* wait for mapbuf busy to go away.
* clobber T

* February 10, 1981 5:18 PM

setBrForMap:

* given Mrow and Mcol, construct a va and put it in the current base register. note that Mrow and Mcol contain up to 9 bits of VA apiece. This code is so arbitrary because the choice of which bits in the VA map into the map ras/cas bits is rather complicated. The code reverses that mapping; ie., given ras and cas it generates the va. Naturally as the page size changes the mapping of the bits changes, too. See the diagram "Dorado Addressing" by Clark and McDaniel. Use a color printer.

```
mc[storagePageSize, 400];
mc[pageIs256C, 400];
mc[pageIs1KC, 1000];
mc[pageIs4KC, 4000];
mc[Mrow.b0, b7];
mc[Mrow.b1, b8];
mc[Mrow.2thru8, b7,377 ];
mc[Mcol.b0, b7];
mc[Mcol.b1, b8];
mc[Mcol.b5, b12];
mc[Mcol.b6, b13];
mc[Mcol.b7, b14];
mc[Mcol.b8, b15];
mc[Mcol.78, b14,b15 ];
mc[Mcol.2thru8, 177 ];
mc[Mcol.2thru4, b9,b10,b11 ];
mc[Mcol.2thru6, b9,b10,b11,b12,b13 ];
set[ras.b0justify, 10];
set[ras.b1justify, 7];
```

```
pushReturn[];
t _ storagePageSize;
t - (pageIs256C);
skpif[ALU=0];
branch[setBrForMap2];
```

* with a drawing of the correspondance
* between various Dorado addresses, this
* might be understandable.

* This code works for 256K map chips. Mrow.b0 and Mcol.b0 are always 0 unless we are using 256K map chips. Mrow.b1 and Mcol.b1 are always 0 unless we're using 64Kk map chips. Mrow.b2thru8, Mcol.b2thru8 always contain legitimate map ras/cas values.

```
t_ ldf[Mrow, 1, ras.b0justify];
rscr_ lsh[t, 11];
t_ ldf[Mrow, 1, ras.b1justify];
t_ lsh[t, 7];
rscr_ (rscr) or t;

t_ ldf[Mrow, 6, 1];
rscr_ t or (rscr);

t_ ldf[Mcol, 1, ras.b0justify];
t_ lsh[t, 10];
t_ t or (rscr);
t_ ldf[Mcol, 1, ras.b1justify];
t_ lsh[t, 6];
t_ t or (rscr);

rscr2 _ t-t;
skpif[r even], Mrow;
rscr2 _ 100000C;
t _ (Mcol) and (Mcol.2thru8);
t _ lsh[t, 10];
branch[setBrForMapDoIt], rscr2 _ t or (rscr2);
```

* bit 0 of map row for 256K chips
* position in BrHi.
* bit 1 of map row for 256K and 64K chips
* position it correctly and add to BrHi
* value we are constructing

* all map chips use this range of values

* 256K chips's cas contributes this bit

* bit 1 of map row for 256K and 64K chips
* position it correctly and add to BrHi
* BrHi is done. Whew!

* compute BrLo
* add va[16] if required

* isolate low 6 bits of column
* correspnd to va[17:23];

setBrForMap2: * February 10, 1981 10:20 AM I don't believe this works.

```
t _ storagePageSize;
t - (pageIs1Kc);
skpif[ALU=0];
branch[setBrForMap3];
t _ (Mrow) and (Mrow.b0);
rscr _ lsh[t, 3];
t _ (Mrow) and (Mrow.b1);
rscr _ (rscr) or t;
t _ ldf[Mrow, 6, 1];
rscr _ (rscr) or t;
```

* corresponds to va[4]
* correspondsk to va[8];
* correspond to va[10:15]

```

t _ (Mcol) and (Mcol.b0);
t _ lsh[t, 2]; * corresponds to va[5];
rscr _ (rscr) or t;
t _ (Mcol) and (Mcol.78);
t _ lsh[t, 10]; * correspond to va[6:7]
rscr _ (rscr) or t;
t _ (Mcol) and (Mcol.b1);
t _ t rsh 1;
rscr _ (rscr) or t; * corresponds to va[9]

rscr2 _ t-t;
skpif[r even], Mrow;
rscr2 _ 100000c; * corresponds to va[16]
t _ (Mcol) and (Mcol.2thru6);
t _ lsh[t, 10]; * correspond to va[17:21]
branch[setBrForMapDoIt], rscr2 _ (rscr2) or t;

setBrForMap3:
t _ storagePageSize;
t - (pageIs4Kc);
skpif[ALU=0];
error; * impossible configuration (neither 256 wd nor 1k nor
4k)

t _ ldf[Mrow, 6, 1];
rscr _ t; * correspond to va[10:15]
t _ (Mrow) and (Mrow.b1);
rscr _ (rscr) or t; * corresponds to va[8]
t _ (Mcol) and (Mcol.b8);
t _ lsh[t, 13]; * corresponds to va[4]
rscr _ (rscr) or t;
t _ (Mcol) and (Mcol.b5);
t _ lsh[t, 7]; * corresponds to va[5]
rscr _ (rscr) or t;
t _ (Mcol) and (Mcol.b7);
t _ lsh[t, 10]; * corresponds to va[6]
rscr _ (rscr) or t;
t _ (Mcol) and (Mcol.b6);
t _ lsh[t, 6]; * corresponds to va[7]
rscr _ (rscr) or t;
t _ (Mcol) and (Mcol.b1);
t _ t rsh 1; * corresponds to va[9]
rscr _ (rscr) or t;

t _ (Mcol) and (Mcol.2thru4);
rscr2 _ lsh[t, 10]; * correspond to va[17:19]
skpif[r even], Mrow;
rscr2 _ (rscr2) or (100000c); * corresponds to va[16]
noop;

setBrForMapDoIt:
call[setBR];
returnP[];

```

* December 13, 1978 6:40 PM

xSetBRforPage:

* will reference that virtual page. Assume 256 words/ page. CLOBBER T, rscr, rscr2

pushReturn[];

rscr _ t;

rscr2 _ t;

rscr _ rsh[rscr, 10];

rscr2 _ lsh[rscr2, 10];

call[setBR];

* rscr _ brHI = top 8 bits of page num

* rscr2 _ brLO = (low 8 bits of page num) lsh 8

* rscr = brHI, rscr2 = brLO

returnP[];

* December 13, 1978 2:59 PM

%

xWriteMapPage

Write an entry into the map: Enter with t = virtual page index and with rscr = the real page that corresponds to that virtual page.

Clobber BR!

%

xWriteMapPage: subroutine;

pushReturn[];

rscr2 _ t;

t _ rscr;

MsubrScr _ t;

* move real page into MsubrScr

%

a virtual page index maps into BR values by performing a left shift of 8. The high order 8 bits of the index shift into BrHi and the low order bits of the index shift into the high order 8 bits of BrLO

%

t _ rsh[rscr2, nBitsInPage];

rscr _ t;

rscr2 _ lsh[rscr2, nBitsInPage];

call[setBR];

call[getMsubrScr];

rscr _ t-t;

call[writeMap], rscr2 _ t;

* setup BR so that addr "0" references virtual page

returnP[];

xReadMapPage: subroutine;

pushReturn[];

call[xSetBrForPage];

call[waitForMapT];

RMap _ r0;

call[waitForMapT];

rscr _ not(Pipe3');

call[xGetPipe4];

rscr2 _ t;

t _ rscr;

rscr _ rscr2;

returnP[];

* enter w/ t = page number, exit w/ t =

* real page number, rscr = pipe4 (w/ all hi true bits)

* save pipe 4 till we can get it into rscr

* t _ real page number

* rscr _ pipe4 (w/bits properly inverted)

* May 29, 1979 2:39 PM

%

xZeroMap Write zero into all the entries of the map (ignore wp,
dirty)

%

xZeroMap:

```
pushReturn[];
t _ (OR[mcr.disHold!,mcr.noWake!]C);
B_FaultInfo'[];
call[setMCR];
```

```
call[iMapPageCtrl];
```

xZeroMapL:

```
call[nextMpage];
skipif[ALU#0];
branch[xZeroMapXit];
rscr _ t-t;
call[xWriteMapPage];
branch[xZeroMapL];
```

xZeroMapXit:

```
returnP[];
```

```

* June 28, 1979 1:13 PM
xGetConfig: subroutine;                                * return w/ t = maxBrHi.
%
Set sMaxBrHi, a register that contains the last valid BrHi +1.
Set xMapBase, a register that contains the offset that PresetMap adds to real page numbers when
initializing the map. xMapBase enables the storage diagnostics to run with contiguous storage modules
that need not begin with module 0.
%
    pushReturn[];

    call[xCountModules];
    sNmodules _ t;
    skipif[alu#0];
    error;                                                * no modules!!!
    noop;

    call[XgetSnModules];
    call[xGetICtype];
    error;                                                * initialize sNmodules
    error;                                                * T: 1=>4K, 4=>16K, 16=>64K
%
    Note: the "ic type" is a number that happens to be the correct value for sMaxBrHi, given only
    one module of that type. Ie., If there are 3 modules, sMaxBrHi _ 3 * icType. Remember that sMaxBrHi
    is one greater than the last valid value for BrHi in this configuration.
%
xConfigBrHi:
    call[XgetSnModules], rscr _ t;                        * Remember icTyp in rscr.
    t _ t-1;                                             * begin w/ nModules-1
    cnt _ t;                                             * rscr contains our increment for sMaxBrHi
    t _ t-t;
    loopUntil[cnt=0&-1, .], t _ t + (rscr);             * multiply by adding
    sMaxBrHi _ t;                                       * Set sMaxBrHi.
%
    (set xMapBase) Now we must find the real page number for the first page in storage. Usually the
    map initialization code sets map[i] _ i. However, if we are missing modules this won't work.
    Furthermore, this code only works when the installed modules are contiguous. Ie., M2, M3 is a valid
    configuration while M0, M3 is not valid (where Mi refers to memory module i). The restriction to
    contiguous modules occurs because the presetMap code doesn't recognize when it crosses a "module
    boundary" when it writes the map. At module boundaries it should check to see if the module really
    exists!
%
xConfigMapBase:
    t _ rscr;                                           * remember, rscr contains the ictype
    rscr2 _ t-t;                                        * now right shift ictype,,0 by the number
    t _ rcy[t,rscr2, nBitsInPage];                     * of address bits in a map page. This gives
    rscr _ t;                                           * the "map increment" for missing modules.
    rscr2 _ not(Config');
    t _ (rscr2) and (config.m0);
    skipif[ALU=0];
    branch[xConfigMBxit], t _ r0;                       * Module 0 is in place

    (rscr2) and (config.m1);
    skipif[ALU=0];
    branch[xConfigMBxit], t _ rscr;                     * missing only one module

    (rscr2) and (config.m2);
    skipif[ALU=0];
    branch[xConfigMBxit], t _ (rscr) + (rscr);          * missing module 0, module 1

    (rscr2) and (config.m3);
    skipif[ALU=0], t _ (rscr) + (rscr);                 * t _ 2 * mapIncrement
    branch[xConfigMBxit], t _ t + (rscr);               * t _ t + mapIncrement

    error;                                              * can't find any modules. There's been a serious
error.

xConfigMBxit:
    xMapBase _ t;                                       * Set xMapBase.

    RBASE _ rbase[sMaxBrHi];
    t _ sMaxBrHi, RBASE _ rbase[defaultRegion];
xGetCLXit:
    ReturnP[];

```

* February 9, 1981 10:14 AM

% **xGetMapICs** This routine initializes the RM locations xPageEndHi,, xPageEndLo so that the loop routines will work properly for the current map ic size.

Assume 1510=DMUX address for three consecutive muffler values that define the signals, MapIs16K, MapIs64K, MapIs256K.

```
EXIT w/
    xEndPageHi,,xEndPageLo          IC size
    0,,40000                        16K
    1,,0                             64K
    4,,0                             256K
%
```

set[MapIs16K, 1510]; * assume this address muffler correctly.

xGetMapICs:

```
pushReturn[];
t_AND[MapIs16K, 177400]C;
call[xRWDmux], rscr2_ t_ t or (AND[MapIs16K, 377]C);
branch[xMapIs16K, ALU<0];
noop; * for placement
call[xRWDmux], rscr2_ t_ (rscr2)+1;
branch[xMapIs64K, ALU<0];
noop; * for placement
call[xRWDmux], rscr2_ t_ (rscr2)+1;
skipif[alu<0];
```

xGetMapICErr: * none of the muffler values we
error; * tried was non-zero.

```
* Map is 256K
xEndPageLo_a0;
t_4c;
xEndPageHi_t;
t_1000c;
xChipEndRasCas_ t;
```

xGetMapICRtn:

```
returnP[];
```

xMapIs16K:

```
xEndPageHi_ a0;
t_ 40000c;
xEndPageLo_ t;
t_ 200C;
branch[xGetMapICRtn], xChipEndRasCas_ t;
```

xMapIs64K:

```
t_ xEndPageLo_ a0;
xEndPageHi_ t+1;
t_ 400C;
branch[xGetMapICRtn], xChipEndRasCas_ t;
```

* June 7, 1979 2:07 PM

%

```
XgetSnModules      return sNmodules
xGetICType         return t = icType = BrHi offset per module
xCountModules      return t = num storage modules, set sNmodules, too.
```

%

```
XgetSnModules: subroutine;
  RBASE _ rbase[rmForStoreLoops];
  return, t _ sNmodules, RBASE _ rbase[defaultRegion];
```

%

This routine uses Config to determine the chip size used in the storage boards. The value it returns describes the maximum BrHi +1 for a single module of the storage boards, given the chip size. Ie., If there are 3 modules, 3 * (number returned in T from calling this routine) is the maximum BrHi +1 for the current memory configuration.

%

```
xGetICType: subroutine;                                * RETURN T =(1=>4k, 4=>16K, 16=64K, 64 =>256K)
  pushReturn[];
```

```
  rscr _ not(Config');
```

```
  rscr _ ldf[rscr, config.icTypeSZ, config.icTypePos];
```

```
  PD _ rscr;
```

```
  skipif[ALU#0], (rscr) # (1c);                        * test against 1 incase it's not zero
  branch[xGetIcRtn], t _ 1c;                          * config.icType=0 ==> 4K chips
```

```
  skipif[ALU#0], (rscr) # (2c);                        * test against 2 incase it's not zero
  branch[xGetIcRtn], t _ 4c;                          * config.icType=1 ==> 16K chips
```

```
  skipif[ALU#0], t _ 64c;                              * set to 256K chips here
  branch[xGetIcRtn], t _ 16c;                         * reset to 64K chips if config.icType=2
```

```
xGetIcRtn:
```

```
  returnP[];
```

* January 12, 1979 3:22 PM

```
xCountModules: subroutine;                            * return T=num storage modules. set sNmodules!
  pushReturn[];
```

```
  rscr _ not(Config');
  noop;
```

```
  rscr _ ldf[rscr, config.modSZ, config.modPOS];      * modules rt justified
```

```
  t _ config.modSZC;                                  * set cnt to (num bits in module field) -1
  cnt _ t;
```

```
  t _ t-t;
```

```
xCountML:
```

```
  skipif[r even], rscr;
  t _ t + 1;
  rscr _ (rscr) rsh 1;
  loopuntil[cnt=0&-1, xCountML];
```

```
  sNmodules _ t;
  returnP[];
```

* January 25, 1979 5:42 PM

%

Code for returning values in T.

%

```
getMsubrScr: subroutine;
  RBASE _ rbase[mSubrScr];
  return, t _ MsubrScr, RBASE _ rbase[defaultRegion];
```

```
xGetPipe4: subroutine;
```

```
  t _ not(pipe4');
  t _ t # (pipe4.sexChange0);
  return, t _ t # (pipe4.sexChange1);
```

* return Pipe4 with all inverted bits
* fixed up such that a '1' bit implies
* the "true" condition


```
xGetPipe2: subroutine;  
    return, t _ not(pipe2');  
xGetNumFlts: subroutine;  
    t _ not(pipe2');  
    return, t _ ldf[t, pipe2.nFaultsSize, pipe2.nFaultsShift];
```

* December 18, 1978 10:20 AM

% exerciser for map: to be used when bringing up new boards

%

```
xBoardLoop: top level;
  RBASE _ rbase[defaultRegion];
  call[setMbase], t _ r0;
  rscr _ t-t;
  call[setBR], rscr2 _ t-t;
  t _ mcr.disHold;
  t _ t OR (mcr.noWake);
  call[setMCR];
  t _ 1c;
  stkp _ t;
```

```
xBdL:
  call[xUp];
  branch[.-1];
```

```
xUp: subroutine;
  pushReturn[];
  RBASE _ rbase[Mwait];
  call[longWait], t _ Mwait;
  t _ MwriteVal;
  Map _ Maddr1, MapBuf _ t;
  call[longWait], t _ Mwait;
  t _ not(MwriteVal);
  Map _ Maddr2, MapBuf _ t;
  noop;
```

```
xup2:
  call[longWait], t _ Mwait;
  RMap _ Maddr1;
  call[longWait], t _ Mwait;
  Mread1 _ not(Pipe3');
  noop;
  RMap _ Maddr2;
  call[longWait], t _ Mwait;
  Mread2 _ not(Pipe3');
  RBASE _ rbase[defaultRegion];
  returnP[];
```

* February 6, 1981 2:06 PM

```
xRWDmux: pushReturn[];
  rscr_ 14C;
```

```
xDmuxL:
  T_ T+(MidasStrobe_ T); * Shift address bit from B[4]
  Noop;
  rscr_ (rscr)-1;
  Branch[xDmuxL, ALU#0];
  T_ ALUFMem, rscr, Branch[.+2, R>=0]; * T_ DMuxData
  UseDMD;
  returnPandBranch[t];
```

```
knowRbase[defaultRegion];
  top level;
```

```
mapSubrsDone: noop ;
```