```
%
May 18, 1981  1:29 PM
		Fix getRandom to use a new random number generator
February 5, 1981  1:20 PM
		Adapt to current d1lang.mc
September 19, 1979  10:42 AM
		Try undoing zeroHold fix -- found a different bug that probably accounts for the behavior.
September 18, 1979  5:55 PM
		Try to make zeroHold more reliable: apparently 3 in a row is not enough.
June 17, 1979  5:11 PM
		Change holdValueLoc to accommodate ifu entry points
April 17, 1979  10:45 PM
		Add sim.holdMask, sim.taskMask, sim.holdShift, sim.taskShift.
January 25, 1979  11:05 AM
		Change simTaskLevel to 12B.
January 18, 1979  5:10 PM
		Add currentTaskNum.
%
TITLE[PREAMBLE];
%

		NAMING CONVENTIONS:
			LABELS BEGIN W/ THE OPERATION BEING TESTED:
				Aplus1
				cntFcn
				aluLT0
			lsh => Left shift
			rsh => Right shift
			lcy => Left Cycle
			rcy => Right Cycle

			Register Read/write tests are suffixed with RW:
				cntRW
				shcRW

			LOOP LABELS ARE SUFFIXED AS INNER (IL) AND OUTER(OL) LOOPS (L).
				cntFcnIL		* INNER LOOP
				cntFcnOL		* OUTER LOOP
				cntFcnXITIL			* LABEL FOR EXITING INNER
LOOP
				cntFcnXITOL			* LABEL FOR EXITING
OUTER LOOP
				Aplus1L			* ONLY LOOP
%

RMREGION[DEFAULTREGION];
rv[r0,0];		rv[r1,1];		rv[rm1,177777];
rv[r01,52525];		rv[r10,125252];		rv[rhigh1,100000];
rv[rscr,0];		rv[rscr2,0];		rv[rscr3,0];
rv[rscr4,0];		rv[stackPAddr, 0];	rv[stackPTopBits, 0];
rv[klink, 0];		rv[hack0,0];		rv[hack1, 0];
rv[hack2,0];

rvrel[rmx0, 0];		rvrel[rmx1, 1];		rvrel[rmx2, 2];
rvrel[rmx3, 3];		rvrel[rmx4, 4];		rvrel[rmx5, 5];
rvrel[rmx6, 6];		rvrel[rmx7, 7];		rvrel[rmx10, 10];

* Constants from FF
```

```
nsp[PNB0,100000];
nsp[PNB1,40000];          nsp[PNB2,20000];          nsp[PNB3,10000];
nsp[PNB4,4000];           nsp[PNB5,2000];           nsp[PNB6,1000];
nsp[PNB7,400];            nsp[PNB8,200];            nsp[PNB9,100];
nsp[PNB10,40];            nsp[PNB11,20];            nsp[PNB12,10];
nsp[PNB13,4];             nsp[PNB14,2];             nsp[PNB15,1];

mc[B0,100000];
mc[B1,40000];             mc[B2,20000];             mc[B3,10000];
mc[B4,4000]; mc[B5,2000]; mc[B6,1000];
mc[B7,400];  mc[B8,200];  mc[B9,100];
mc[B10,40];  mc[B11,20];  mc[B12,10];
mc[B13,4];   mc[B14,2];   mc[B15,1];

mc[NB0,PNB0];
mc[NB1,PNB1];             mc[NB2,PNB2];             mc[NB3,PNB3];
mc[NB4,PNB4];             mc[NB5,PNB5];             mc[NB6,PNB6];
mc[NB7,PNB7];             mc[NB8,PNB8];             mc[NB9,PNB9];
mc[NB10,PNB10];           mc[NB11,PNB11];           mc[NB12,PNB12];
mc[NB13,PNB13];           mc[NB14,PNB14];           mc[NB15,PNB15];

mc[CM1,177777];mc[C77400,77400]; mc[C377,377];mc[CM2,-2];
mc[getIMmask, 377];       * isolate IM data after getIm[]!

m[noop, BRANCH[.+1]];
m[skip, BRANCH[.+2]];
m[error, ILC[(BRANCH[ERR])]];
m[skiperr, ILC[(BRANCH[.+2])] ILC[(BRANCH[ERR])]];
m[skpif, BRGO@[TS@] JMP@[.+2,#1,#2] ];
m[skpUnless, BRGO@[TS@] DBL@[.+1,.+2,#1,#2] ];
m[loopChk, BRGO@[TS@] DBL@[#1,.+1,#2,#3] ];
m[loopUntil, BRGO@[TS@] DBL@[.+1,#2,#1,#3]];* if #1 then goto .+1 else goto #2
m[loopWhile, BRGO@[TS@] DBL@[#2,.+1,#1,#3]];* if #1 then goto #2 else goto .+1
```

```
* May 18, 1981  1:36 PM

                rmRegion[rm2ForKernelRtn];
                knowRbase[rm2ForKernelRtn];

                rv[chkSimulatingRtn, 0];
                rv[fixSimRtn, 0];
                rv[chkRunSimRtn, 0];
                rv[currentTaskNum, 0];
                rmRegion[randomRM];
                knowRbase[randomRM];

                rv[rndm0, 134134];          rv[rndm1, 054206];
                rv[rndm2, 036711];          rv[rndm3, 103625];
                rv[rndm4, 117253];          rv[rndm5,154737];
                rv[rndm6, 041344];          rv[rndm7, 006712];
* rm below not used for simple random number generator.
                rv[randV,0];  * current value from random number generator
                rv[randX,0];  * current index into random number jump table
                rv[oldRandV,0];             * saved value
                rv[oldRandX,0];             * saved value

                knowRbase[defaultRegion];

                mp[flags.conditionalP, 200];            * bit that indicates conditional simulating
                mp[flags.conditionOKp, 100];            * bit that indicates conditional simulating is ok

                set[holdValueLoc, 3400];    mc[holdValueLocC, holdValueLoc];
                mc[flags.taskSim, b15];     * NOTE: The "flags" manipulation code
                mc[flags.holdSim, b14];     * works only so long as there are no more
                mc[flags.simulating, flags.taskSim, flags.holdSim];
                set[simTaskLevel, 12]; mc[simTaskLevelC, simTaskLevel];

                mc[sim.holdMask, 377]; set[sim.holdShift, 0];
                mc[sim.taskMask, 177400];  set[sim.taskShift, 10];

m[lh, byt0[ and[rshift[#1,10], 377] ] byt1[ and[#1, 377]]
                ];                  * assemble data for left half of IM
m[rh, byt2[ and[rshift[#1,10], 377] ] byt3[ and[#1, 377]]
                ];                  * assemble data for right half of IM

m[zeroHold, ilc[(#1 _ A0)]
                ilc[(hold&tasksim _ #1)]
                ilc[(hold&tasksim _ #1)]
                ilc[(hold&tasksim _ #1)]
                ];
```

```
* December 11, 1978  3:20 PM
%
                subroutine entry/exit macros
%
m[saveReturn, ilc[(t _ link)]
                top level[]
                ilc[(#1 _ t)]
                ];

m[saveReturnAndT, ilc[(#2 _ t)]
                ilc[(t _ link)]
                top level[]
                ilc[(#1 _ t)]
                ];

m[returnUsing, subroutine[]
                ilc[(RBASE _ rbase[#1])]
                ilc[(link _ #1)]
                ilc[(return, RBASE _ rbase[defaultRegion])]
                ];
m[returnAndBranch, subroutine[]
                ilc[(RBASE_rbase[#1])]
                ilc[(link_#1)]
                ilc[(RBASE_rbase[defaultRegion])]
                ilc[(return, PD_#2)]
                ];
m[pushReturn, subroutine[]
                ilc[(stkp+1)]
                top level[]
                ilc[(stack _ link)]
                ];              * notice that this macro doesn't clobber T !!!
m[pushReturnAndT, subroutine[]
                ilc[(stkp+1)]
                ilc[(stack&+1 _ link)]
                top level[]
                ilc[(stack_t)]
                ];
m[returnP, subroutine[]
                ilc[(link_(stack&-1))]
                ilc[(return)]
                ];
m[pReturnP,
                ilc[(stkp-1)]
                subroutine[]
                ilc[(link_(stack&-1))]
                ilc[(return)]
                ];
m[returnPAndBranch, subroutine[]
                ilc[(link_(stack&-1))]
                ilc[(return, PD_#1)]
                ];
m[pReturnPAndBranch, subroutine[]
                ilc[(stkp-1)]
                ilc[(link_(stack&-1))]
                ilc[(return, PD_#1)]
                ];

*m[getRandom, ilc[(RBASE _ rbase[randX])]
*               ilc[(randX _ (randX)+1, Bdispatch _ randX)]
*               ilc[(call[random], RBASE _ rbase[rndm0])]
*               ilc[(RBASE _ rbase[defaultRegion])]
*               ];              *  Returns random number in T, leaves RBASE= defaultRegion
m[getRandom, ilc[(RBASE_ rbase[rndm0])]
                ilc[(call[random])]
                ilc[(RBASE_ rbase[defaultRegion])]
                ];

knowRbase[defaultRegion];
```

TITLE[POSTAMBLE];
TOP LEVEL;

%

July 11, 1984
Fix assembly problem in notifytask - bigDBispatch was inside a subroutine
May 18, 1981 1:37 PM
Fix random number generator to use a better algorithm. Modify restart code, and the various subroutines associated with random numbers. Add setRandV, cycleRandV.

February 1, 1980 6:24 PM
Fix goto[preBegin], described below, into goto[restartDiagnostic]. Postamble already defines and uses preBegin.

February 1, 1980 11:52 AM
Fix restart to goto[preBegin]. This allows each diagnostic to perform whatever initialization it wants.

September 19, 1979 9:18 PM
Fix another editing bug in chkSimulating, used the wrong bit to check for flags.conditionOK -- just did it wrong.

September 19, 1979 9:08 PM
Fix bug in chkSimulating wherein an edit lost a carriage return and a statment became part of a comment. Unfortunately, automatic line breaks made the statement look as if it were still there rather than making it look like part of the comment line.

September 19, 1979 4:23 PM
Fix placement errors associated with bumming locations from makeholdvalue and from checksimulating.

September 19, 1979 3:48 PM
Bum locations to fit postamble with current os/microD: reallyDone, checkFlags global, make checkFlags callers exploit FF, eliminate noCirculate label, make others shorter..

September 19, 1979 10:41 AM
change callers of getIM*, putIM* to use FF field when calling them.

September 19, 1979 10:18 AM
Create zeroHoldTRscr which loops to zero hold-- called by routines that invoke resetHold when the hold simulator may be functioning. Make getIM*, putIM* routines global.

September 16, 1979 1:27 PM

Bum code to fix storage full problem that occurs because OS 16/6 is bigger than OS 15/5: remove kernel specific patch locations (patch*).

August 1, 1979 3:28 PM
Add scopeTrigger.

June 17, 1979 4:48 PM
Move IM data locations around to accommodate Ifu entry points

April 26, 1979 11:03 AM
Make justReturn global.

April 19, 1979 5:03 PM
Remove calls to incTask/HoldFreq from enable/disableConditionalTask.

April 18, 1979 3:24 PM
Remove DisplayOff from postamble.

April 18, 1979 11:11 AM
Rename chkTaskSim, chkHoldSim, simControl to incTaskFreq, incHoldFreq, makeHoldValue; clean up setHold.

April 17, 1979 10:51 PM
SimControl now masks holdFreq and taskFreq & shifts them w/ constants defined in Postamble.

April 11, 1979 3:49 PM
Add breakpoint to "done", and fix, again, a bug associated with task simulation. Set defaultFlagsP (when postamble defines it) to force taskSim and holdSim.

March 7, 1979 11:42 PM
Set RBASE to defaultRegion upon entry to postamble. thnx to Roger.

February 16, 1979 2:54 PM
Modify routines that read IM to invert the value returned in link if b1 from that value =1 (this implies the whole value was inverted).

January 25, 1979 10:41 AM
Change **taskCirculate** code to accommodate taskSim wakeups for task 10D, 12B
January 18, 1979 5:13 PM
Modify **checkTaskNum** to use the RM value, currentTaskNum, and modify **taskCircInc** to keep the copy in currentTaskNum.

January 15, 1979 1:25 PM
add **justReturn**, a subroutine that just returns
January 9, 1979 12:07 PM
breakpoint on xorTaskSimXit to avoid midas bug
%

%*+++
+++++++++++++++++++++++++

TABLE of CONTENTS, by order of Occurence

ERR	global label where ERROR macro gives control
IMdata	beginning of Postamble’s FLAGS, et c.

%*+++
+++++++++++++++++++++++++
IM[ILC,0];
%*+++
+++++++++++++++++++++++++

This code presumes R0=0 and uses RSCR, RSCR2, T, and Q. It uses a number of other registers in a different RM region.

When Postamble gets control of the processor at "Done", bits in "Flags", a word in IM determine which of Postamble’s functions will occur when it runs. At the least, Postamble inrements at 32 bit number in IM called Iterations. If flags.taskSim is true, the task simulator started. The task simulator awakens after a software controllable number of clocks has occured. The microcode that wakes up must reset the task simulator before it (the microcode) blocks to cause a task wakeup to occur again. The first time a program runs (ie., the time before it gives control to "done") the task simulator and the hold simulator (discussed below) are inactive. Running the task simulator forces task specific hardware functions to effect the state of the machine.

When flags.holdSim is set, Postamble sets the hold simulator to a non-zero value. The 8 bit "hold value" enters a circulating shift register where occurence of a "1" bit at b[0] causes an external hold. This exercises the hold hardware.

The body of postamble contains a number of procedures for user programs, including routines to read and write IM, a routine to return a random number, and routines to initialize a task’s pc and to notify it.

%*+++
+++++++++++++++++++++++++
done:

```
* June 17, 1979  4:49 PM                            POSTAMBLE CONSTANTS

        set[randomTloc, 620];                       * random number generator may have to
* be moved to "global" call location if extensively used!


        set[flagsLoc, 1000];                        mc[flagsLocC, flagsLoc];
        set[taskFreqLoc, 1400];                     mc[taskFreqLocC, taskFreqLoc];
        set[holdFreqLoc, 2000];                     mc[holdFreqLocC, holdFreqLoc];
        set[nextTaskLoc, 2400];                     mc[nextTaskLocC, nextTaskLoc];
        set[itrsLoc, 3000];                         mc[itrsLocC, itrsLoc];
*       holdValueLoc defined in preamble!
        set[preBeginLoc, 4000];                     mc[preBeginLocC, preBeginLoc];
        set[initTloc, 4400];                        mc[initTlocC, initTloc];

        ifdef[simInitLocC,,mc[simInitLocC, initTloc] ];   * define the bmux constant for the
*       address of the task simulator code. If its already been defined, leave it as is.

*       flags.taskSim defined in preamble!
*       flags.holdSim defined in preamble!
*       flags.simulating defined in preamble!
        mc[flags.testTasks, b13];                   * than 8 flags (since READIM rtns a BYTE)
        mc[flags.conditional, flags.conditionalP];  * allow simulating iff flags.simulating
                                                    * AND flags.conditionOK
        mc[flags.conditionOK, flags.conditionOKp];  * enable conditional simulating

%*++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++
```

This portion of the kernel code encapsulates the microdiagnostic with an outer loop. This outer loop has several features that it implements:

task simulation
hold simulation
task switching

Task simulation refers to the taskSim register in the hardware. It is 4 bits wide; taskSim[0] enables the task simulator and taskSim[1:3] form a counter that determines the number of cycles before a task wakeup occurs.

Hold simulation is similar: holdSim is an 8-bit recirculating shift register in which the presence of a 1 in bit 7 causes HOLD two instructions later.

Task switching determines which task will run the microdiagnostics.

These features are controlled by the flags word in IM. If the appropriate bits are set to one, the associated feature will function. The bits are defined above (flags.taskSim, flags.holdSim, flags.testTasks).

```
%*++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++


        rmRegion[rmForKernelRtn];
        knowRbase[rmForKernelRtn];

        rv[setHoldRtn, 0];
        rv[oldt, 0];                                * save t, rscr, rscr2, rtn link for resetHold
        rv[oldrscr, 0];
        rv[oldrscr2, 0];
        rv[resetHoldRtn, 0];
        rv[xorFlagsRtn,0];
        rv[flagSubrsRtn, 0];
        rv[mainPostRtn, 0];
```

```
	knowRbase[rm2ForKernelRtn];				* defined in preamble because of macros
* that reference randV, randX


	knowRbase[defaultRegion];
```

```
* February 1, 1980  6:24 PM                        POSTAMBLE CONTROL CODE

        RBASE _ rbase[defaultRegion], breakpoint;  * set RBASE incase user's is different.
        call[incTaskFreq];
        call[incHoldFreq];
        call[makeHoldValue];
        call[taskCirculate];
        call[incIterations];

        call[checkFlags]t_flags.testTasks;         * bookkeeping is done. switch tasks if required
        skpif[ALU#0];
        branch[preBegin];                          * xit if not running other tasks

taskCircInit:                                      * now that bookkeeping is done, switch tasks if required
        noop;                                      * for placement.
        call[checkTaskNum];
        rscr _ t;                                  * rscr _ nextTask

        t _ preBeginLocC;                          * link _ t _ preBeginLoc
        subroutine;
taskCirc:
        zeroHold[rscr2];                           * turn off hold-task sim during LdTpc_, wakeup
        link _ t;
        t _ rscr;
        top level;
        ldTPC _ t;                                 * tpc[nextTask] _ preBeginLoc
        call[notifyTask];                          * wakeup nextTask: task num in t
set[xtask, 1];
        block;
set[xtask, 0];

preBegin: noop,                                    at[preBeginLoc];
        call[chkRunSimulators];                    * check for simulator conditions and run if required

reallyDone:                                        * LOOP TO BEGIN
        t_RSCR_a1;
        goto[begin], RSCR2_t;

restart:                                           * restart diagnostics from "initial" state
        rndm0 _ t-t;                               * restart random number generator
        randX _ t-t;

        rscr _ t-t;                                * restart hold/task simulator stuff
        call[putIMRH], t _ holdFreqLocC;
        rscr _ t-t;
        call[putIMRH], t_taskFreqLocC;
        rscr _ t-t;
        call[putIMRH], t _ holdValueLocC;

        rscr _ t-t;                                * restart iterations count
        call[putIMRH], t _ itrsLocC;

        branch[restartDiagnostic];                 * special entry point so each diagnostic
* can perform whatever special initialization that it wants to perform
* January 18, 1978  1:51 PM

%*+++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++
        This code sets the taskSim value with the next value if flags.testTasks is true. Otherwise 0 is used.

IF flags.taskSim THEN
          BEGIN                                      -- when hardware counts to 17 it awakens
          taskFreq _ (taskFreq + 1) or 10b;          -- simTask
          IF taskFreq > 15 THEN taskFreq _ 12;       -- always wait min=2 cycles
          END
        ELSE taskFreq _ 0;
IF flags.holdSim THEN
          BEGIN
          holdFreq _ holdFreq+1;
          IF holdFreq >376 THEN holdFreq _ 0;
          END;

        ELSE holdFreq _ 0;

%*+++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++
incTaskFreq: subroutine;
        t _ link;
        mainPostRtn _ t;
        top level;

        call[checkFlags], t_flags.taskSim;           * see if taskSim enabled
        branch[writeTaskSim, alu=0],t_r0;            * use 0 if not enabled
        t_taskFreqLocC;                              * incrment next taskSim
        call[readByte3];
        t_(r1)+(t);
        t-(156C);                                    * Use [1..156). 156 => max wait,
        skpif[alu<0];                                * 1 = > min wait. Beware infinite hold!
        t_r1;                                        * see discussion at simInit, simSet code
        noop;

writeTaskSim:
        rscr _ t;
        call[putIMRH], t_taskFreqLocC;               * update IM location

taskSimRtn:
        goto[topLvlPostRtn];


incHoldFreq: subroutine;                             * see if holdSim enabled
        t _ link;
        mainPostRtn _ t;
        top level;

        call[checkFlags], t_flags.holdSim;
        branch[noHoldSim, alu=0],t_r0;               * use zero if hold not enabled
        t_holdFreqLocC;
        call[readByte3];
        t_t+(r1);
        t-(377c);                                    * IF holdFreq >376
        skpif[alu<0];
        t_r1;                                        * THEN holdFreq _ 1;
        noop;                                        * here for placement
```

```
noHoldSim:
        rscr _ t;                                    * rewrite IM
        call[putIMRH], t _ holdFreqLocC;

holdSimRtn:
        goto[topLvlPostRtn];
```

```
* April 17, 1979  10:51 PM
%*+++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++
```

This code actually controls the task and hold loading. It is responsible for initializing T for the task at simTaskLevel, and it is responsible for initializing HOLD.

The code proceeds by constructing the current value to be loaded into hold and placing it in IM at holdValueLoc. Kernel loads HOLD as its last act before looping to BEGIN.

hold&tasksim_ requires hold value in left byte, task counter value in right byte.

```
%*+++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++
makeHoldValue: subroutine;                  * construct holdValue
    saveReturn[mainPostRtn];

    call[chkSimulating];
    skpif[alu#0];
    branch[simCtrl0];

    t_holdFreqLocC;                         * rscr2 _ holdFreq
    call[readByte3], t_holdFreqLocC;
    rscr2 _ t;

    call[readByte3], t_taskFreqLocC;        * t _ taskFreq

    t_lsh[t, sim.taskShift];                * position hold and task values
    t_t and (sim.taskMask);
    rscr2 _ lsh[rscr2, sim.holdShift];
    rscr2 _ (rscr2) and (sim.holdMask);
    rscr2 _ (rscr2) and (377c);

    rscr2 _ (t) + (rscr2);                  * taskFreq,,holdFreq
    rscr _ rscr2;
%
 now, save combined taskSim, holdSim values in IM. Last thing done before exiting postamble is to set HOLD if simulating.
%
simCtrlWHold:                               * may branch here from simCtrl0
    call[putIMRH], t _ holdValueLocC;       * write holdValue into holdValueLoc
    branch[simCtrlRtn];

simCtrl0:
    branch[simCtrlWHold], rscr _ t-t;       * write zero into holdValueLoc

simCtrlRtn:
    goto[topLvlPostRtn];
```

```
* September 19, 1979  9:09 PM

%*+++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++
            IF chkSimulating[] THEN fixSimulator[];

* cause hold or task simulator to run, if required
%*+++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++


chkRunSimulators: subroutine;
        saveReturn[chkRunSimRtn];
        call[chkSimulating];
        dblBranch[chkRunsimXit, chkRunSimDoIt, alu=0];
chkRunSimDoIt:                                          * run the simulator
        noop;
        call[fixSim];
        noop;

chkRunSimXit:
        returnUsing[chkRunSimRtn];


* September 19, 1979  9:19 PM
%*+++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++
chkSimulating: PROCEDURE RETURNS[weAreSimulating: BOOLEAN] =
            BEGIN
            weAreSimulating _ FALSE;
            IF flags.Simulating THEN
                        IF ~(flags.Conditional) OR (flags.Conditional AND flags.ConditionOK) THEN
                                    weAreSimulating _ TRUE;
            END;

%*+++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++
chkSimulating: subroutine;
        saveReturn[chkSimulatingRtn];
        call[checkFlags], t_flags.simulating;           * check for taskSim OR holdSim
        branch[chkSimNo, alu=0];
        t _ flags.conditional;                          * We're simulating. check
        call[checkFlags], t _ flags.conditional;
        dblbranch[chkSimYes, chkSimCond, alu=0];
chkSimCond:                                             * conditional simulation. check for ok
        t _ flags.conditionOK;
        call[checkFlags];
        skpif[alu=0];
        branch[chkSimYes];                              * conditionOK is set, do it!
        branch[chkSimNo];
chkSimYes:                                              * run the simulator
        t _ (r0)+1;                                     * rtn w/ alu#0
chkSimRtn:
        returnAndBranch[chkSimulatingRtn, t];
chkSimNo:
        branch[chkSimRtn], t _ r0;                      * rtn w/ alu=0
```

```
* January 25, 1979  10:44 AM
%
```

This code controls task circulation for the diagnostics: when flags.testTasks is set, postamble causes successive tasks to execute the diagnostic code when the current task has completed. If flags.taskSim is true the diagnostic is using the taskSimulator to periodically awaken the simulator task; consequently, that task (*simTaskLevel*) must not execute the diagnostic -- otherwise the advantage of the simulator for testing the effects of task switching will be lost.

```
IF ~flags.testTasks THEN RETURN;
temp _ getTaskNum[] + 1;                     -- increment the current number
IF flags.taskSim THEN
    IF temp = simTaskLevel THEN temp _ temp+1;
IF temp > maxTaskLevel THEN temp _ 0;
putTaskNum[temp];                            -- remember it in IM
%
taskCirculate: subroutine;
    saveReturn[mainPostRtn];
    call[checkFlags], t _ flags.testTasks;
    branch[taskCircRtn, ALU=0];              * Don’t bother if not task circulating.
    noop;

    call[checkTaskNum];                      * Increment the current task number.
    t _ t + (r1);                            * Current value came back in t.
    q _ t;                                   * Remember incremented value in Q.

    call[checkFlags], t _ flags.taskSim;
    skpif[ALU#0], t _ q;                     * Now, see if using task simulator.
    branch[taskCircChk];                     * If not task simulating, check for max size.

    t - (simTaskLevelC);                     * Since we’re task simulating, avoid
    skpif[ALU#0];                            * we must avoid simTaskLevel.
    t _ t+1;                                 * Increment over simTask if required.
    noop;

taskCircChk:
    t - (20C);                               * See if tasknum is too big.
    skpif[ALU#0];
    t _ t-t;                                 * We wraparound to zero.

    currentTaskNum _ t;                      * keep it in both RM and IM
    rscr _ t;
    call[putIMRH], t _ nextTaskLocC;
    noop;                                    * for placement

taskCircRtn:
    goto[topLvlPostRtn];
```

```
* January 18, 1978  1:57 PM

incIterations: subroutine;                        * maintain double precision count at incItrsLoc
        t _ link;
        mainPostRtn _ t;
        top level;

        call[getIMRH], t _ itrsLocC;              * increment iteration count at tableloc+1
        rscr _ (t)+1;

        rscr2 _ rscr;                             * copy new itrs
        rscr2 _ (t) #(rscr);                      * see if new b0 # old b0

        rscr2 _ (rscr2) AND (b0);
        skpif[alu#0];
        branch[incItrs2], q _ r0;                 * new b0 = old b0. remember in q and write
        t and (B0);                               * see if b0 went from 0 to 1 or 1 to 0 (carry)
        skpif[alu=0], q_r0;                       * skpif old b0 = 0
        q _ r1;
incItrs2:

        call[putIMRH], t _ itrsLocC;              * T = addr, rscr = value
        rscr2 _ q;
        branch[incItrsRtn,alu=0];                 * goto incItrsRtn if no carry

incItersHi16:
        link _ t;                                 * read hi byte of hi 16 bits
        call[getIMLH];
        rscr _ (t)+1;
        call[putIMLH], t _ itrsLocC;              * T = addr, rscr = value
        noop;                                     * help the instruction placer.

incItrsRtn:
        goto[topLvlPostRtn];
```

```
* March 20, 1978  1:51 PM                          KERNEL - COMMON SUBROUTINES

resetHold: subroutine;                             * special subroutine called by IM manipulating
* code.  This subr saves t, rscr, rscr2 and causes hold to be initialized to the value in
* holdValueLocC. It restores the RM and T values before returning.
        oldT _ t;
        t _ link;
        resetHoldRtn _ t;
        top level;
        t _ rscr;
        oldrscr _ t;
        t _ rscr2;
        oldrscr2 _ t;                              * link, t, rscr, and rscr2 are now saved.

        t _ HoldValueLocC;                         * READ RIGHT HALF, HoldValueLocC
        subroutine;
        link _ t;
        top level;
        readim[3];                                 * read low order byte
        subroutine;
        t _ link;                                  * low byte in t
        t and (b1);                                * see if the data is inverted. If so, b1 will
        skpif[ALU=0];                              * 1, and we must reinvert the data.
        t _ not(t);
        t _ t and (getIMmask);                     * isolate the byte

        rscr _ HoldValueLocC;
        subroutine;
        link _ rscr;
        top level;
        readim[2];                                 * read hi order byte
        subroutine;
        rscr2 _ link;                              * hi byte in rscr2
        (rscr2) and (b1);                          * see if the data is inverted. If so, b1 will
        skpif[ALU=0];                              * 1, and we must reinvert the data.
        rscr2 _ not(rscr2);
        rscr2 _ (rscr2) and (getIMmask);           * isolate the byte

        top level;
        noop;
        rscr2 _ lsh[rscr2, 10];                    * left shift hi byte
        t _ t and (377C);                          * isolate low byte
        t _ t OR (rscr2);                          * add hi byte
        call[setHold];

        knowRbase[rmForKernelRtn];                 * restore link, t, rscr, rscr2, then return
        RBASE _ rbase[rmForKernelRtn];
        t _ oldrscr;
        rscr _ t;
        t _ oldrscr2;
        rscr2 _ t;
        subroutine;
        link _ resetHoldRtn;
        return, t _ oldt, RBASE _ rbase[defaultRegion];
```

```
* June 23, 1978  10:22 AM
setHold: subroutine;                            * ENTER w/ T = HOLD value
* clobber t, rscr, rscr2

    zeroHold[rscr2];                            * kill hold-task sim before polyphas instrs xqt
    rscr2 _ q;                                  * SAVE  Q
    q _ t;                                      * save hold value, then save rtn link
    t _ link;
    setHoldRtn _ t;

    taskingon;
    t _ simInitLocC;                            * defined w/ postamble constants OR in
    * some user specific code (eg., memSubrsA where RM values are defined). This
    * convention allows users to specify their own code to run when the simulator task runs.
    link _ t;                                   * cause task taskSimLevel to put
    top level;
    ldTPC _ simTaskLevelC;                      * proper hold value in T for refresh
    notify[simTaskLevel];                       * after task switch occurs. Remember
                                                * taskSim is a counter. refresh it!
    noop;                                       * wakeup will happen soon
    noop;
    rbase _ RBASE[rmForKernelRtn];
    t_ setHoldRtn, rbase _ RBASE[defaultRegion];
    Q _ rscr2;                                  * restore Q
    subroutine;
    link _ t;
    return;

* This code actually causes T to be set properly and branches to the code that sets HOLD.

            set[xtask, 1];

simInit:
    t _ q,                                      at[initTloc];
simSet:
    hold&tasksim_t;                             * T init'd at simInit
    noop;                                       * this noop doesn't cause hold to count
simBlock:
    branch[simSet], block;                      * count hold, block
%
Note: if t = 14, then hold = 16 when the simulator blocks. The preempted task will execute one instruction, then the
task simulator will waken the simulator task.
%

    set[xtask, 0];

* November 6, 1978  12:07 PM                   MIDAS SUBROUTINE for testing the task simulator
testTaskSim: subroutine;
    rscr _ link;                                * save return in case we later want it
    top level;
    t _ lsh[t, 10];                             * ENTER w/ T = task sim val NOT shifted
    q _ t;                                      * simInit expects q = hold value

    subroutine;
    t _ initTlocC;
    link _ t;
    top level;
    LDTPC _ simTaskLevelC;
    notify[simTaskLevel];
```

```
        noop;
        t _ t - t;                              * t _ 0
        branch[., alu=0], t_t;                  * this shouldn't change
testTaskErr:
        branch[.], breakpoint;

        subroutine;
        link _ rscr;
        return;

fixSim: subroutine;
        t_link;                                 * save return in fixSimRtn
        fixSimRtn _ t;
        top level;

        call[makeHoldValue];                    * compose holdValue and set hardware
        call[getIMRH], t _ holdValueLocC;
        call[setHold];

        returnUsing[fixSimRtn];

zeroHoldTRscr: subroutine;
        t_4c;
        rscr_a0;
zeroHoldTRscrL:
        Hold&TaskSim_rscr;
        t_t-1, Hold&TaskSim_rscr;
        loopUntil[alu<0, zeroHoldTRscrL];
        return;
```

```
* January 18, 1979  5:18 PM                               * READ/WRITE IM
%
```

The subroutines that read and write IM turn OFF hold simulator before touching IM. Before they return to the caller, the invoke "resetHold" to reset the hold register to the contents of "holdValueLoc". By convention, the current value of the two simulator registers is kept in "holdValueLoc" for this express purpose. Zeroing and resetting hold is done because of hardware restrictions: hold and polyphase instructions don't mix.

ReadIM[] instructions are followed by a mask operation with getIMmask because of the interaction between DWATCH (Midas facility) and LINK[0].

```
%

readByte3: subroutine;                                    * CLOBBER T, RSCR!
        zeroHold[rscr];

        rscr _ link;                                      * this routine assumes t points to IM
        link _ t;                                         * it reads the least significan byte in IM

        top level;                                        * read byte 3
        readim[3];
        subroutine;
        t_link;                                           * t = byte3
        t and (b1);                                       * see if the data is inverted. If so, b1 will
        skpif[ALU=0];                                     * 1, and we must reinvert the data.
        t _ not(t);
        t _ t and (getIMmask);                            * isolate the byte

        top level;                                        * reset value of hold and return
        call[resetHold];
        subroutine;
        link _ rscr;
        return;                                           * return w/ byte in t

getIMRH: subroutine, global;                              * CLOBBER T, RSCR, RSCR2!
        zeroHold[rscr];                                   * disable task/hold Sim before touching IM

        rscr _ link;                                      * ENTER w/ T pointing to IM location
        link _ t;

        top level;                                        * read hi byte of right half
        readim[2];
        subroutine;
        rscr2 _ link;                                     * rscr2 = high byte
        (rscr2) and (b1);                                 * see if the data is inverted. If so, b1 will
        skpif[ALU=0];                                     * 1, and we must reinvert the data.
        rscr2 _ not(rscr2);
        rscr2 _ (rscr2) and (getIMmask);                  * isolate the byte

        link _ t;                                         * read low byte of right half
        top level;
        readim[3];
        subroutine;
        t _ link;                                         * t = low byte, rscr2 = hi byte
        t and (b1);                                       * see if the data is inverted. If so, b1 will
        skpif[ALU=0];                                     * 1, and we must reinvert the data.
        t _ not(t);
        t _ t and (getIMmask);                            * isolate the byte

        rscr2 _ lsh[rscr2, 10];
        t _ t + (rscr2);                                  * RETURN w/ T = IMRH
        top level;
        call[resetHold];
        subroutine;
        link _ rscr;
        return;

getIMLH: subroutine, global;                    * CLOBBER T, RSCR, RSCR2!
        zeroHold[rscr];                         * disable task/hold Sim before touching IM

        rscr _ link;                            * ENTER w/ T pointing to IM location
        link _ t;

        top level;                              * read hi byte of left half
        readim[0];
        subroutine;
        rscr2 _ link;                           * rscr2 = hi byte
        (rscr2) and (b1);                       * see if the data is inverted. If so, b1 will
        skpif[ALU=0];                           * 1, and we must reinvert the data.
        rscr2 _ not(rscr2);
        rscr2 _ (rscr2) and (getIMmask);        * isolate the byte

        link _ t;                               * read low byte of left half
        top level;
        readim[1];
        subroutine;                             * CLOBBER T, RSCR, RSCR2!
        t _ link;                               * t = low byte, rscr2 = hi byte
        t and (b1);                             * see if the data is inverted. If so, b1 will
        skpif[ALU=0];                           * 1, and we must reinvert the data.
        t _ not(t);
        t _ t and (getIMmask);                  * isolate the byte

        rscr2 _ lsh[rscr2, 10];
        t _ t + (rscr2);                        * RETURN w/ T = IMLH

        top level;
        call[resetHold];
        subroutine;
        link _ rscr;
        return;

putIMRH: subroutine, global;                    * T = addr, RSCR = value, clobberr RSCR2
        rscr2 _ link;
        link _ t;

        zeroHold[t];                            * disable task/hold Sim before touching IM

        top level;
        t _ rscr;
        IMRHB'POK _ t;

        call[resetHold];
        subroutine;
        link _ rscr2;
        return;
```

```
putIMLH: subroutine, global;                        * T = addr, RSCR = value, Clobber RSCR2
        rscr2 _ rscr;
        rscr _ link;
        link _ t;

        zeroHold[t];                                * disable task/hold Sim before touching IM

        top level;
        t _ rscr2;
        IMLHR0'POK _ t;

        call[resetHold];
        subroutine;
        link _ rscr;
        return;

checkFlags: subroutine, global;                     * CLOBBER T, RSCR, RSCR2
        rscr _ link;                                * this routine assumes t has a bit mask

        zeroHold[rscr2];                            * disable task/hold Sim before touching IM

        rscr2 _ flagsLocC;                          * it reads the flags word in IM
        link _ rscr2;                               * and performs t_tANDflag
        top level;
        readim[3];
        subroutine;
        rscr2 _ link;
        (rscr2) and (b1);                           * see if the data is inverted. If so, b1 will
        skpif[ALU=0];                               * 1, and we must reinvert the data.
        rscr2 _ not(rscr2);
        rscr2 _ (rscr2) and (getIMmask);            * isolate the byte

        top level;
        call[resetHold];
        subroutine;
        link _ rscr;
        return, t _ t AND(rscr2);                   * returnee can do alu=0 fast branch

checkTaskNum: subroutine;                           * enter: T=expected task num,
        rscr_t, RBASE _ rbase[currentTaskNum];      * return: T=current task num, branch condition
        t _ currentTaskNum, RBASE _ rbase[defaultRegion];          * clobber rscr, rscr2

        return, t#(rscr);                           * rtn w/ branch condition, t=current task
* number, rscr = expected task number.
* August 1, 1979  3:30 PM                     other, miscellaneous subroutines
notifyTask: subroutine;
        rscr_ link;
        top level;
        bigBDispatch_t;
        branch[dispatchTbl];
set[nloc, 6600];
dispatchTbl:
        branch[nxit], notify[0],              at[nloc,0];
        branch[nxit], notify[1],              at[nloc,1];
        branch[nxit], notify[2],              at[nloc,2];
        branch[nxit], notify[3],              at[nloc,3];
        branch[nxit], notify[4],              at[nloc,4];
        branch[nxit], notify[5],              at[nloc,5];
        branch[nxit], notify[6],              at[nloc,6];
        branch[nxit], notify[7],              at[nloc,7];
        branch[nxit], notify[10],             at[nloc,10];
        branch[nxit], notify[11],             at[nloc,11];
        branch[nxit], notify[12],             at[nloc,12];
        branch[nxit], notify[13],             at[nloc,13];
        branch[nxit], notify[14],             at[nloc,14];
        branch[nxit], notify[15],             at[nloc,15];
        branch[nxit], notify[16],             at[nloc,16];
        branch[nxit], notify[17],             at[nloc,17];
        branch[.], breakpoint,                at[nloc,20];
        branch[.], breakpoint,                at[nloc,21];

        subroutine;
nxit:
        link _ rscr;
        return;

topLvlPostRtn:
        RBASE _ rbase[mainPostRtn];
        link _ mainPostRtn;
        return, RBASE _ rbase[defaultRegion];

scopeTrigger: subroutine;
        t _ a0, global;
        TIOA _ t, T_a1;
        return, TIOA_t;

justReturn:  * this subroutine ONLY RETURNS.  Calling justReturn forces the instruction
        return, global;                       *  (logically) after the call to occur in the physically
* next location after the call.  This is a way of reserving a noop that can ALWAYS be
* safely patched with a "call".
* April 24, 1978  6:51 PM
        knowRbase[randomRM];
random:
        T_ LSH[rndm0, 11];                                    * T_ 2^9 * R
        T_ T+(rndm0);                                         * (2^9 + 2^0)* R
        T_ LSH[T, 2];                                         * (2^11 + 2^2)* R
        T_ T+(rndm0);                                         * (2^11 + 2^2 + 2^0)* R
        T_ T+(33000C);
        T_ rndm0_ T+(31C), Return;                  * +13849 (= 33031B)

        goto[random1], t _ rndm0, RBASE _ rbase[randV],     at[randomTloc,0];     knowRbase[randomRM];
        goto[random1], t _ rndm1, RBASE _ rbase[randV],     at[randomTloc,1];     knowRbase[randomRM];
        goto[random1], t _ rndm2, RBASE _ rbase[randV],     at[randomTloc,2];     knowRbase[randomRM];
        goto[random1], t _ rndm3, RBASE _ rbase[randV],     at[randomTloc,3];     knowRbase[randomRM];
        goto[random1], t _ rndm4, RBASE _ rbase[randV],     at[randomTloc,4];     knowRbase[randomRM];
        goto[random1], t _ rndm5, RBASE _ rbase[randV],     at[randomTloc,5];     knowRbase[randomRM];
        goto[random1], t _ rndm6, RBASE _ rbase[randV],     at[randomTloc,6];     knowRbase[randomRM];
        goto[random1], t _ rndm7, RBASE _ rbase[randV],     at[randomTloc,7];

random1:
        return, t _ randV _ (randV)+t;
        knowRbase[defaultRegion];

* code below modified to save/restore/use rndm0 rather than randV.
saveRandState: subroutine;                          * remember random number seed
        RBASE _ rbase[randV];
        oldRandV _ rndm0;
        oldRandX _ randX;
        return, RBASE _ rbase[defaultRegion];

restoreRandState: subroutine;                       * restore remembered random number seed
        RBASE _ rbase[randV];
        rndm0 _ oldRandV;
        randX _ oldRandX;
        return, RBASE _ rbase[defaultRegion];

getRandV: subroutine;
        RBASE _ rbase[randV];
        RETURN, t _ rndm0, RBASE _ rbase[defaultRegion];
setRandV: subroutine;
        RETURN, rndm0_ t;
cycleRandV: subroutine;
        RBASE_ rbase[randV];
        rndm0_ (rndm0)+1, RETURN, RBASE_ rbase[defaultRegion];
```

```
* January 20, 1978  3:04 PM                  'FLAGS' manipulating code

xorFlags: subroutine;                        * T = value to XOR into flags
* CLOBBER RSCR, RSCR2, T
        rscr2 _ t;                           * save bits
        t _ link;
        xorFlagsRtn _ t;
        top level;

        t _ flagsLocC;
        call[readByte3];
        t _ t # (rscr2);                     * xor new bits

        rscr _ t;                            * put new value back into IM
        call[putIMRH], t _ flagsLocC;

        returnUsing[xorFlagsRtn];

xorTaskCirc: subroutine;                     * xor the flags.testTasks bit in FLAGS
* CLOBBER RSCR, RSCR2, T
        saveReturn[flagSubrsRtn];
        t _ flags.testTasks;
        call[xorFlags];
        noop;

        returnUsing[flagSubrsRtn];

xorHoldSim: subroutine;                      * xor the flags.holdSim bit in FLAGS
        saveReturn[flagSubrsRtn];
        t _ flags.holdSim;
        call[xorFlags];

        rscr_a0;                             * whether off or on, clear holdFreqLoc
        call[putIMRH], t _ holdFreqLocC;     * holdFreq _ 0

        call[fixSim];
xorHoldSimXit:
        noop, breakpoint;

        returnUsing[flagSubrsRtn];

xorTaskSim: subroutine;                      * xor the flags.taskSim bit in FLAGS
        saveReturn[flagSubrsRtn];
        t _ flags.taskSim;
        call[xorFlags];

        rscr_a0;                             * whether off or on, clear taskFreqLoc
        call[putIMRH], t _ taskFreqLocC;     * taskFreq _ 0

        call[fixSim];                        * fix the holdValueLoc, set hardware

xorTaskSimXit:
        breakpoint, noop;
        returnUsing[flagSubrsRtn];
        top level;

* June 22, 1978  10:15 AM
%
```

This code supports the conditional simulation mechanism. **disableConditionalTask** is a subroutine that requires no parameters. It clears flags.conditionOK and sets flags.conditional. It also turns off the hold simulator.

enableConditionalTask sets flags.conditionOK and flags.conditional, then it calls makeHoldValue to force the hold simulator into working.
%

```
disableConditionalTask: subroutine;
        saveReturn[flagSubrsRtn];
        call[checkFlags], t _ (r0)-1;                * use mask = -1 to force a read of all the bits
        rscr _ not (flags.conditionOK);
        rscr _ t and (rscr);
        rscr _ (rscr) or (flags.conditional);
        rscr _ (rscr) and (377C);                    * isolate lower byte
        call[putIMRH], t _ flagsLocC;

        call[makeHoldValue];                         * compose a new hold value from task and
                                                     * hold simulator sub values
        call[zeroHoldTRscr];                         * stop hold
        call[resetHold];                             * jam the hold register w/ holdValue
        returnUsing[flagSubrsRtn];

enableConditionalTask: subroutine;
        saveReturn[flagSubrsRtn];
        call[checkFlags], t _ (r0)-1;                * use mask = -1 to force a read of all the bits
        noop;                                        * make placement easier
        rscr _ t or (flags.conditionOK);
        rscr _ (rscr) or (flags.conditional);
        noop;                                        * make placement easier
        call[putIMRH], t _ flagsLocC;                * write the new value

        call[makeHoldValue];                         * compose a new hold value from task and
                                                     * hold simulator sub values
        call[zeroHoldTRscr];                         * stop hold
        call[resetHold];                             * jam the hold register w/ holdValue
        returnUsing[flagSubrsRtn];
        top level;


* ERRORs come here!
        branch[err];
SET[ERRLOC,400];
ERR:
        BREAKPOINT,GLOBAL, AT[ERRLOC];
        GOTO[.],BREAKPOINT, AT[ERRLOC,1];
        GOTO[.], AT[ERRLOC,2];

* DATA HELD IN IM

IMdata:
            ifdef[defaultFlagsP,,set[defaultFlagsP,add[flags.taskSim!, flags.holdSim!]]];        * define default flags
if undefined

            data[(Flags: lh[0] rh[defaultFlagsP], at[flagsLoc])];               * CONTROL FLAGS
            data[(taskFreq: lh[0] rh[0], at[taskFreqLoc])];           * task sim value
            data[(holdFreq: lh[0] rh[0], at[holdFreqLoc])];           * hold sim value
            data[(nextTask: lh[0] rh[0], at[nextTaskLoc])];           * next task value
            data[(holdValue: lh[0] rh[0], at[holdValueLoc])];         * current hold value
            data[(iterations: lh[0] rh[0], at[itrsLoc])];    * iteration count

postDone: noop;
```

```
L X Reset
L X Do-it
L X Ld KERNEL                    ; load the microprogram
L B0 Addr RBASE 0
L B1 Addr RBASE 17
L B0 Val 0
L B1 Val 0
L B2 Addr MCR
L B2 Val  1                      ; turn off stack overflow/underflow wakeups from memC
L B4 Addr STACKPTOPBITS
L B5 Addr STACKPADDR
L B19 Addr TOPE
L B18 Addr READY
L B17 Addr PENC
L B16 Addr BNPC
L B15 Addr TNIA
L B14 Addr HACK0
L B10 Addr STK 0
L B11 Addr STK 1
L B12 Addr STK 2
L C0 Addr R0                     ; display common registers
L C1 Addr R1
L C2 Addr RM1
L C3 Addr R01
L C4 Addr R10
L C5 Addr RHIGH1
L C6 Addr RSCR
L C7 Addr RSCR2
L C8 Addr T 20
L C9 Addr CNT
L C10 Addr Q
L C11 Addr FLAGS                 ; control for  hold, task simulator
L C12 Addr ITERATIONS            ; count of iterations
L C13 Addr NEXTTASK
L C14 Addr HOLDVALUE
L A19 Addr TASK 20               ; use task 0 as default
L A19 Val 0
L A7 Addr RBASE 20               ; use rbase 17 as default
L A7 Val 0
L C11 Val 7                      ; TURN ON TASK CIRC, HOLD SIM, TASK SIM
L X DisplayOn                    ; May 18, 1981  11:43 AM
L X TimeOut 100000
L X Go BEGIN
L X Skip 1
L X ShowError Timed out
```

```
TITLE[KERNEL];
IM[ILC,0];
TOP LEVEL;
* October 9, 1986  3:32 PM
restartDiagnostic:
BEGIN:
	goto[im0];
afterKernel1:
	goto[beginKernel2];
afterKernel2:
	goto[beginKernel3];
afterKernel3:
 goto[beginKernel4];
afterKernel4:
goto[beginKernel5];
afterKernel5:


	T_R0;	* R0 SHOULD HAVE ZERO IN IT
	BRANCH[.+2,ALU=0];
	ERROR;

	T_(R1)-1;	* R1 SHOULD HAVE ONE IN IT.
	BRANCH[.+2,ALU=0];
	ERROR;

	T_R1;
	T_T+(RM1);	* RM1 SHOULD HAVE -1 IN IT;
	BRANCH[.+2,ALU=0];
	ERROR;

	T_100000C;
	T_T#(RHIGH1);	* RHIGH1 SHOULD HAVE 100000B
	BRANCH[.+2, ALU=0];
	ERROR;

	T_R10;		* R10 SHOULD HAVE 125252B
	BRANCH[.+2, ALU<0];
	ERROR;

	T_R01;
	DBLBRANCH[.+1, .+2, ALU<0];
	ERROR;		* R01 SHOULD HAVE 52525B IN IT

	T_NOT(R01);
	T_T#(R10);	* R01 SHOULD EQUAL NOT(R10);
	BRANCH[.+2,ALU=0];
	ERROR;		* NOTE THIS IS NOT A COMPLETELY ACCURATE
* TEST FOR CONTENTS OF R10, R01!
	goto[done];
* CODE for midas debugging
	top level;
set[dbgTbls,100];
l1:	branch[l1],	at[dbgTbls,0];
l2:	noop,	at[dbgTbls,1];
	branch[l2],	at[dbgTbls,2];
l3:	noop,	at[dbgTbls,3];
	noop,	at[dbgTbls,4];
	branch[l3],	at[dbgTbls,5];
```

```
l4:  noop,    at[dbgTbls,6];
     noop,    at[dbgTbls,7];
     noop,    at[dbgTbls,10];
     branch[l4],    at[dbgTbls,11];
l5:  noop,    at[dbgTbls,12];
     noop,    at[dbgTbls,13];
     noop,    at[dbgTbls,14];
     noop,    at[dbgTbls,15];
     branch[l5],    at[dbgTbls,16];
l6:  noop,    at[dbgTbls,17];
     noop,    at[dbgTbls,20];
     noop,    at[dbgTbls,21];
     noop,    at[dbgTbls,22];
     noop,    at[dbgTbls,23];
     branch[l6],    at[dbgTbls,24];

END;
```

```
* INSERT[D1ALU.MC];
* TITLE[PROG1];
* INSERT[PREAMBLE.MC];
%
September 22, 1986  5:18 PM
	Removed comments for scope loops that did not exist. Removed several double labels for the same
microinstruction.
September 21, 1981  10:53 AM
	Add comments for various Scope (midas) files.
May 18, 1981  11:08 AM
	Change "bypass" to save and restore values in RBase 0.  Need this because of a change in versions of d1lang.
diagnostic now uses "standard" d1lang.
May 8, 1979  11:40 AM
	Add RoddByPass tests at enbd of xorBypass
March 26, 1979  10:58 AM
	Add overflow test
March 10, 1979  6:43 PM
	Add tst of branch conditions when reschedule is ON.
January 18, 1979  5:23 PM
	Remove checkTaskNum, a temporary kludge that caused reschedTest to fail during task circulation.
January 9, 1979  10:44 AM
	add reschedTest
%

%
			CONTENTS

TEST			DESCRIPTION

(singlestep)		Chec RM to T, T to RM movement
aluEQ0			check the fast branch code
aluLT0			check the fast branch code
rEven			check the fast branch code
rGE0			check the fast branch code
reschedTest		check the reschedule/noreschedule fast branches
xorNoBypass		test XOR alu op
bypass			test bypass decision logic
xorBypass		test XOR alu op, ALLOW BYPASS; R odd bypass test here, too.
(alu ops)		Test various alu operations (A+1, A+B, A-1,A-B)
Carry			Test carry fast branch
(resched+branches)	test effect of resched upon fast branches.
freezeBCtest		Test Freeze BC  function (emulator only)
overflowTest		Test the overflow fast branch function
%
```

```
* September 15, 1978  10:18 AM


%
SINGLE STEP THIS CODE:                     A AND B MULTIPLEXORS
     The point is to determine if it is possible to move data values thru
the alu into different registers.
%


top level;
kernel1:
IM0:        T_RM1;                         *TEST ALL ONES, ALL ZEROS, ALTER. 01, 10
            NOOP;                          * USE NOOP TO AVOID BYPASS LOGIC
IM2:        RSCR_T;
            NOOP;
IM4:        T_R0;                          * TEST 0
            NOOP;
IM6:        RSCR _ T;
            NOOP;

* NOW MOVE IT THRU A MUX

IM14:       T_A_RM1;                       * TEST ALL ONES
            NOOP;
IM16:       RSCR _A_ T;
            NOOP;
IM20:       T_A_R0;                        * TEST ALL ZEROS
            NOOP;
IM22:       RSCR_ A_T;

* CHECK B MUX THRU FF FIELD: SINGLE STEP THIS CODE

IM23:       T_B0;                          * check that FF,0 works
IM24:       T_77400C;
IM25:       T_B15;                         * check that 0,FF works
IM26:       T_376C;
```

```
* September 21, 1981  10:54 AM
%
                              END SINGLE STEPPING !!!


      GIVEN SIMPLE A AND B PATHS, VALIDATE:
      RESULT=0
      RESULT<0
      R>=0
      R EVEN
      CNT=0&+1
%


%     TEST ALU=0  BY CHECKING EVERY BIT IN THE WORD: GET CONSTANTS FROM
FF AND CHECK THEM FOR =0. USE BYPASS LOGIC!!

These tests assume that there is no difference between amux source and bmux source for fast branches. ACTUALLY,
the initial set of tests will check amux
sources too!

      T contains the value received.

%
aluEQ0FF:
      t_B0;
      skpUnless[ALU=0],rscr_(A_t);             * check it thru Amux
      error;                                   * Thinks bit0 is zero
      skpUnless[ALU=0];
      error;                                   * Thinks bit0 is zero

      t_B1;
      skpUnless[ALU=0],rscr_(A_t);             * check it thru Amux
aluEq0FFB1:
      error;                                   * Thinks bit1 is zero
      skpUnless[ALU=0];
      error;                                   * Thinks bit1 is zero

      noop;                                    *here for placement.
      t_B2;
      skpUnless[ALU=0],rscr_(A_t);             * check it thru Amux
aluEq0FFB2:
      error;                                   * Thinks bit2 is zero
      skpUnless[ALU=0];
      error;                                   * Thinks bit2 is zero

      t_B3;
      skpUnless[ALU=0],rscr_(A_t);             * check it thru Amux
      error;                                   * Thinks bit3 is zero
      skpUnless[ALU=0];
      error;                                   * Thinks bit3 is zero

      t_B4;
      skpUnless[ALU=0], rscr_(A_t);            * check it thru Amux
aluEq0FFB4:
      error;                                   * Thinks bit4 is zero
      skpUnless[ALU=0];
      error;                                   * Thinks bit4 is zero
```

```
        t_B5;
        skpUnless[ALU=0], rscr_(A_t);          * check it thru Amux
        error;                                 * Thinks bit5 is zero
        skpUnless[ALU=0];
        error;                                 * Thinks bit5 is zero

        noop;                                  * here for placement.
        t_B6;
        skpUnless[ALU=0], rscr_(A_t);          * check it thru Amux
aluEq0FFB6:
        error;                                 * Thinks bit6 is zero
        skpUnless[ALU=0];
        error;                                 * Thinks bit6 is zero

        t_B7;
        skpUnless[ALU=0], rscr_(A_t);          * check it thru Amux
        error;                                 * Thinks bit7 is zero
        skpUnless[ALU=0];
        error;                                 * Thinks bit7 is zero

        t_B8;
        skpUnless[ALU=0], rscr_(A_t);          * check it thru Amux
aluEq0FFB8:
        error;                                 * Thinks bit8 is zero
        skpUnless[ALU=0];
        error;                                 * Thinks bit8 is zero

        t_B9;
        skpUnless[ALU=0], rscr_(A_t);          * check it thru Amux
        error;                                 * Thinks bit9 is zero
        skpUnless[ALU=0];
        error;                                 * Thinks bit9 is zero

        noop;                                  * here for placement.
        t_B10;
        skpUnless[ALU=0], rscr_(A_t);          * check it thru Amux
aluEq0FFB10:
        error;                                 * Thinks bit10 is zero
        skpUnless[ALU=0];
        error;                                 * Thinks bit10 is zero

        t_B11;
        skpUnless[ALU=0], rscr_(A_t);          * check it thru Amux
        error;                                 * Thinks bit11 is zero
        skpUnless[ALU=0];
        error;                                 * Thinks bit11 is zero

        t_B12;
        skpUnless[ALU=0], rscr_(A_t);          * check it thru Amux
aluEq0FFB12:
        error;                                 * Thinks bit12 is zero
        skpUnless[ALU=0];
        error;                                 * Thinks bit12 is zero

        t_B13;
        skpUnless[ALU=0], rscr_(A_t);          * check it thru Amux
        error;                                 * Thinks bit13 is zero
        skpUnless[ALU=0];
        error;                                  * Thinks bit13 is zero

        noop;                                   * here for placement.
        t_B14;
        skpUnless[ALU=0], rscr_(A_t);           * check it thru Amux
aluEq0FFB14:
        error;                                  * Thinks bit14 is zero
        rscr_(A_t);                             * check it thru Amux
        skpUnless[ALU=0];
        error;                                  * Thinks bit14 is zero

        t_B15;
        skpUnless[ALU=0], rscr_(A_t);           * check it thru Amux
        error;                                  * Thinks bit15 is zero
        skpUnless[ALU=0];
        error;                                  * Thinks bit15 is zero


%
 TEST ALU=0 BY PASSAGE THRU RM AND PASSAGE THRU T

        For all the alu=0 tests, an error implies the wrong branch was taken.
        The known values in RM are used to test the branch

        AVOID BYPASS LOGIC!
%

aluEq0RT:
        t_r0;
        skpif[alu=0];
        error;                                  * Thinks r0 is zero
        rscr_t;
        skpif[alu=0],t_r1;
        error;

        skpUnless[alu=0];
        error;                                  * Thinks r1 is zero
        rscr_t;
        skpUnless[alu=0],t_rm1;
        error;

aluEq0RTM1:
        skpUnless[alu=0];
        error;                                  * Thinks rm1 is zero
        rscr_t;
        skpUnless[alu=0],t_r1;
        error;

        skpUnless[alu=0];
        error;                                  * Thinks r1 is zero
        rscr_t;
        skpUnless[alu=0],t_r01;
        error;

aluEq0RT01:
        skpUnless[alu=0];
        error;                                  * Thinks r01 is zero
        rscr_t;
        skpUnless[alu=0],t_r10;
```

```
error;

skpUnless[alu=0];
error;                                  * Thinks r10 is zero
rscr_t;
skpUnless[alu=0],t_rhigh1;
error;

skpUnless[alu=0];
error;                                  * Thinks rhigh1 is zero
rscr_t;
skpUnless[alu=0];
error;
```

```
%
 TEST RESULT <0

        For all the alu<0 tests, an error implies the wrong branch was taken.
        The known values in RM are used to test the branch

        AVOID BYPASS LOGIC
%
aluLT0RT:
        t_rhigh1;
        skpif[alu<0];
        error;                                  * Thinks rhigh1 >=0
        rscr_t;
        skpif[alu<0];
        error;                                  * Thinks T (=RIGH1) >=0

        t_r10;
        skpif[alu<0];
aluLT0RT10:
        error;                                  * Thinks r10 >= 0
        rscr_t;
        skpif[alu<0];
        error;                                  * Thinks T (=r10) >=0

        t_r1;
        skpUnless[alu<0];
aluLT0RT1:
        error;                                  * Thinks r1<0
        rscr_t;
        skpUnless[alu<0];
        error;                                  * Thinks T (=r1) >=0

        t_r01;
        skpUnless[alu<0];
aluLT0RT01:
        error;                                  * Thinks r10 >= 0
        rscr_t;
        skpUnless[alu<0];
        error;                                  * Thinks T (=r10) >=0



* TEST FOR RESULT EVEN

rEven:
        skpif[r even], t_r0;
        error;                                  * thinks r0 odd

        skpUnless[r even], t_r1;
        error;                                  * Thinks r1 EVEN

        skpif[r even], t_rhigh1;
        error;                                  * Thinks rhigh1 ODD

        skpUnless[r even], t_r01;
        error;                                  * Thinks r01 EVEN

        skpif[r even], t_r10;
```

```
error;                          * Thinks r10 ODD
```

```
rGE0:
        skpif[r >=0],t_r1;
        error;                          * Thinks r1 <0

        skpif[r>=0],t_r01;
        error;                          * Thinks r01 <0

        skpif[r>=0],t_r0;
        error;                          * Thinks r0 <0

        skpUnless[r>=0],t_rm1;
        error;                          * Thinks RM1>=0

        skpUnless[r>=0],t_rhigh1;       * Thinks rhigh1 >=0
        error;
```

```
* April 9, 1982  3:22 PM
%
```

jcnBR

The preceeding tests checked that ff decodes work properly and that the alu bit slice doesn't drop any bits. Now we check that jcn conditional branch opcodes work properly.

This test forces the assembler to use the jcn field to encode the fast branch condition. It does this by forcing the FF field to be used for a constant (B_0C) in the same instruction where the conditional branch occurs.

Since the other tests make sure that the actual branch condition is correctly detected (eg., is ALU=0 or not), this test serves to checkout the jcn decoders and a small amount of other circuitry. Consequently is is not necessary to test for both values of the fast branch: we need only check that when the branch condition is true it gets taken when using jcn encoding.

True condtions for fast branches:
    ALU=0, ALU<0, noCarry, CNT=0, R<0, R odd, noIOattn (noIoattn is tested in resched test)

```
%
jcnBR:
        Pd_ r0;
        skpif[ALU=0], B_0c;                     * use FF field for a constant
jcnBREq0:
        error;                                  * jcn encoded ALU=0 didn't work ok

        Pd_ rm1;
        skpif[ALU<0], B_0c;                     * use FF field for a constant
jcnBRLt0:
        error;                                  * jcn encoded ALU<0 didn't work ok

        t_ (r0)+(r0);
        skpUnless[carry], B_0C;
jcnBRCarry:
        error;                                  * jcn encoded carry didn't work

        cnt_r0;
        branch[.+2,cnt=0&-1], B_0C;
jcnBrCntEq0:
        error;                                  * jcn encoded cnt=0 didn't work

        skpif[r<0], rm1, B_0C;
jcnBrRLt0:
        error;                                  * jcn encoded R<0 didn't work

        skpif[r odd], r1, B_ 0c;
jcnBrRodd:
        error;                                  * jcn encoded R odd didn't work
```

```
* September 21, 1981  11:01 AM
%
        reschedTest
Set and clear resched; see if we can branch on its value.

%
reschedTest:
        call[checkTaskNum], t_t-t;
        skpif[ALU=0];
        branch[reschedXit];

        noreschedule[];
        skpif[reschedule'];
reschedErr1:                            * we just cleared resched, yet
        error;                          * branch condition thinks it is set.
        skpif[reschedule'], B_0C;
reschedErr1a:
        error;                          * jcn encoded br didn't work

        reschedule[];
        skpif[reschedule];
reschedErr2:                            * we just set resched, yet the
        error;                          * branch condition doesn't realize it.
        noreschedule[];
reschedXit:
        noop;
```

```
%
 September 15, 1978  10:56 AM
 TEST XOR USING ALU=0. USE NOOP TO AVOID BYPASS.

        Generally, T _ RSCR_ someFFconstant;
        T _ T#(RSCR)
        IF T is non zero, there was an error: one bits in T indicate
        the problem.

%

xorNoBypass:
        t_(rscr)_B0;
        noop; t_t#(rscr);
        skpif[alu=0];
        error;                                  * (T _ B0 xor (RSCR) ) NE 0

        t_(rscr)_B1;
        noop; t_t#(rscr);
        skpif[alu=0];
        error;                                  * (T _ B1 xor (RSCR) ) NE 0

xorNoBypassB2:
        t_(rscr)_B2;
        noop; t_t#(rscr);
        skpif[alu=0];
        error;                                  * (T _ B2 xor (RSCR) ) NE 0

        t_(rscr)_B3;
        noop; t_t#(rscr);
        skpif[alu=0];
        error;                                  * (T _ B3 xor (RSCR) ) NE 0

        t_(rscr)_B4;
        noop; t_t#(rscr);
        skpif[alu=0];
xorNoBypassB4:
        error;                                  * (T _ B4 xor (RSCR) ) NE 0

        t_(rscr)_B5;
        noop; t_t#(rscr);
        skpif[alu=0];
        error;                                  * (T _ B5 xor (RSCR) ) NE 0

xorNoBypassB6:
        t_(rscr)_B6;
        noop; t_t#(rscr);
        skpif[alu=0];
        error;                                  * (T _ B6 xor (RSCR) ) NE 0

        t_(rscr)_B7;
        noop; t_t#(rscr);
        skpif[alu=0];
        error;                                  * (T _ B7 xor (RSCR) ) NE 0

xorNoBypassB8:
        t_(rscr)_B8;
        noop; t_t#(rscr);
        skpif[alu=0];
        error;                                      * (T _ B8 xor (RSCR) ) NE 0


        t_(rscr)_B9;
        noop; t_t#(rscr);
        skpif[alu=0];
        error;                                      * (T _ B9 xor (RSCR) ) NE 0

**xorNoBypassB10**:
        t_(rscr)_B10;
        noop; t_t#(rscr);
        skpif[alu=0];
        error;                                      * (T _ B8 xor (RSCR) ) NE 0

        t_(rscr)_B11;
        noop; t_t#(rscr);
        skpif[alu=0];
        error;                                      * (T _ B10 xor (RSCR) ) NE 0

        t_(rscr)_B12;
        noop;t_t#(rscr);
        skpif[alu=0];
**xorNoBypassB12**:
        error;                                      * (T _ B12 xor (RSCR) ) NE 0

        t_(rscr)_B13;
        noop;t_t#(rscr);
        skpif[alu=0];
        error;                                      * (T _ B13 xor (RSCR) ) NE 0

**xorNoBypassB14**:
        t_(rscr)_B14;
        noop;t_t#(rscr);
        skpif[alu=0];
        error;                                      * (T _ B14 xor (RSCR) ) NE 0

        t_(rscr)_B15;
        noop;t_t#(rscr);
        skpif[alu=0];
        error;                                      * (T _ B15 xor (RSCR) ) NE 0
```

```
* May 18, 1981  11:12 AM
% bypass
        This code checks the decision portion of the bypass circuitry. There are at least two different issues associated
with bypass: 1) should a bypass be done, and 2) do the bypass data paths work. This test addresses point 1.

%
rvrel[rmx10, 10];
bypass:
        RBASE _ 0s;
        q_ rmx0;
        t_rmx0_cm1;                             * this is the old, stable version
        rmx0_t-t;                               * this is the new version
        t_rmx0;                                 * should use bypassed version of rmx0
        skpif[alu=0], rmx0_q;                   * RESTORE rmx0 here.
bypassErr0:                                     * bypass associated w/ rm addr 0 doesn't
        error;                                  * seem to work

        q_ rmx1;
        t_rmx1_cm1;                             * this is the old, stable version
        rmx1_t-t;                               * this is the new version
        t_rmx1;                                 * should use bypassed version of rmx1
        skpif[alu=0], rmx1_q;                   * RESTORE rmx1 here.
bypassErr1:                                     * bypass associated w/ rm addr 1 doesn't
        error;                                  * seem to work

        q_ rmx2;
        t_rmx2_cm1;                             * this is the old, stable version
        rmx2_t-t;                               * this is the new version
        t_rmx2;                                 * should use bypassed version of rmx2
        skpif[alu=0], rmx2_q;                   * RESTORE rmx2 here.
bypassErr2:                                     * bypass associated w/ rm addr 2 doesn't
        error;                                  * seem to work

        q_ rmx4;
        t_rmx4_cm1;                             * this is the old, stable version
        rmx4_t-t;                               * this is the new version
        t_rmx4;                                 * should use bypassed version of rmx4
        skpif[alu=0], rmx4_q;                   * RESTORE rmx4 here.
bypassErr4:                                     * bypass associated w/ rm addr 4 doesn't
        error;                                  * seem to work

        q_ rmx10;
        t_rmx10_cm1;                            * this is the old, stable version
        rmx10_t-t;                              * this is the new version
        t_rmx10;                                * should use bypassed version of rmx10
        skpif[alu=0], rmx10_q;                  * RESTORE rmx10 here.
bypassErr10:                                    * bypass associated w/ rm addr 10 doesn't
        error;                                  * seem to work
%
        This section of the test works by changing Rbase.
%
        RBASE _ 2s;
        t_rmx0_cm1;                             * this is the old, stable version
        rmx0_t-t;                               * this is the new version
        t_rmx0;                                 * should use bypassed version of rmx0
        skpif[alu=0];
bypassErr20:                                    * bypass associated w/ rm addr 20 doesn't
        error;                                  * seem to work
        RBASE _ 4s;
        t_rmx0_cm1;                          * this is the old, stable version
        rmx0_t-t;                            * this is the new version
        t_rmx0;                              * should use bypassed version of rmx0
        skpif[alu=0];
bypassErr40:                                 * bypass associated w/ rm addr 40 doesn't
        error;                               * seem to work

        RBASE _ 10s;
        t_rmx0_cm1;                          * this is the old, stable version
        rmx0_t-t;                            * this is the new version
        t_rmx0;                              * should use bypassed version of rmx0
        skpif[alu=0];
bypassErr100:                                * bypass associated w/ rm addr 100 doesn't
        error;                               * seem to work
bypassXit:

        RBASE _ rbase[defaultRegion];
```

```
%
 August 30, 1977  6:29 PM
 TEST XOR USING ALU=0.

        Generally, T _ RSCR_ someFFconstant;
        T _ T#(RSCR)
        IF T is non zero, there was an error: one bits in T indicate
        the problem.

%
* TEST XOR USING ALU=0. CHECK BYPASS.

xorBypass:

        t_(rscr)_B0;
        t_t#(rscr);
        skpif[ALU=0];
        error;                                      * (T _ B0 xor (RSCR) ) NE 0

        t_(rscr)_B1;
        t_t#(rscr);
        skpif[ALU=0];
        error;                                      * (T _ B1 xor (RSCR) ) NE 0

        t_(rscr)_B2;
        t_t#(rscr);
        skpif[ALU=0];
xorBypass2:
        error;                                      * (T _ B2 xor (RSCR) ) NE 0

        t_(rscr)_B3;
        t_t#(rscr);
        skpif[ALU=0];
        error;                                      * (T _ B3 xor (RSCR) ) NE 0

        t_(rscr)_B4;
        t_t#(rscr);
        skpif[ALU=0];
xorBypass4:
        error;                                      * (T _ B4 xor (RSCR) ) NE 0

        t_(rscr)_B5;
        t_t#(rscr);
        skpif[ALU=0];
        error;                                      * (T _ B5 xor (RSCR) ) NE 0

        t_(rscr)_B6;
        t_t#(rscr);
        skpif[ALU=0];
xorBypassB6:
        error;                                      * (T _ B6 xor (RSCR) ) NE 0

        t_(rscr)_B7;
        t_t#(rscr);
        skpif[ALU=0];
        error;                                      * (T _ B7 xor (RSCR) ) NE 0

        t_(rscr)_B8;
        t_t#(rscr);
        skpif[ALU=0];
xorBypassB8:
        error;                                  * (T _ B8 xor (RSCR) ) NE 0

        t_(rscr)_B9;
        t_t#(rscr);
        skpif[ALU=0];
        error;                                  * (T _ B9 xor (RSCR) ) NE 0

        t_(rscr)_B10;
        t_t#(rscr);
        skpif[ALU=0];
xorBypassB10:
        error;                                  * (T _ B10 xor (RSCR) ) NE 0

        t_(rscr)_B11;
        t_t#(rscr);
        skpif[ALU=0];
        error;                                  * (T _ B11 xor (RSCR) ) NE 0

        t_(rscr)_B12;
        t_t#(rscr);
        skpif[ALU=0];
xorBypassB12:
        error;                                  * (T _ B12 xor (RSCR) ) NE 0

        t_(rscr)_B13;
        t_t#(rscr);
        skpif[ALU=0];
        error;                                  * (T _ B13 xor (RSCR) ) NE 0

        t_(rscr)_B14;
        t_t#(rscr);
        skpif[ALU=0];
xorBypassB14:
        error;                                  * (T _ B14 xor (RSCR) ) NE 0

        t_(rscr)_B15;
        t_t#(rscr);
        skpif[ALU=0];
        error;                                  * (T _ B15 xor (RSCR) ) NE 0


        rscr _ t-t;
        rscr _ 1c;
        skpif[R ODD], rscr;
RoddByPassErr0:                                 * fast branch r odd bypass doesn't work
        error;                                  * Rscr has 1 in it

        rscr _ t-t;
        skpUnless[R ODD], rscr;
RoddByPassErr1:                                 * fast branch r odd bypass doesn't work.
        error;                                  * rscr has zero in it.
```

```
* August 30, 1977  6:29 PM
%
	TEST ALU ADDITION AND SUBTRACTION:
	A+1		A+B			A-1  A-B

%
Aplus1:
	t_(r0)+1;				* 1=t_r0+1
	t_(r1)#t;
	skpif[alu=0];
	error;

Aplus1b:
	t_(rm1)+1;				* 0=t_rm1+1
	skpif[alu=0];
	error;

Aplus1c:
	rscr_CM2;
	t_(rscr)+1;
	t_t#(rm1);				* (T=-1)=-2+1
	skpif[alu=0];
	error;

Aplus1d:
	rscr_5C;
	t_(rscr)+1;
	rscr_6C;
	t_t#(rscr);				* (T=6)=5+1
	skpif[alu=0];
	error;

AplusB:
	t_r0;
	t_t+(r0);				* 0=0+0
	skpif[alu=0];
	error;

AplusBb:
	t_r1;
	t_t+(r0);
	t_t#(r1);				* 1=1+0
	skpif[alu=0];
	error;

AplusBc:
	t_r0;
	t_t+(r1);
	t_t#(r1);				* 1=0+1
	skpif[alu=0];
	error;

AplusBd:
	t_rm1;
	t_t+(r0);
	t_t#(rm1);				* -1=-1+0
	skpif[alu=0];
	error;
```

```
AplusBe:
        t_rm1;
        t_t+(rm1);
        rscr_177776C;
        t_t#(rscr);                         * -2=-1+-1
        skpif[alu=0];
        error;

AplusBf:
        t_rm1;
        t_t+(r1);                           * 0=-1+1
        skpif[alu=0];
        error;

AplusBg:
        t_r1;
        t_t+(rm1);                          * 0=1+-1
        skpif[alu=0];
        error;

AplusBh:
        t_r01;
        t_t+(r10);
        t_t#(rm1);                          * -1=52525+125252
        skpif[alu=0];
        error;

AplusBi:
        t_r10;
        t_t+(r01);
        t_t#(rm1);                          * -1=125252+52525
        skpif[alu=0];
        error;

AplusBj:
        t_rhigh1;
        t_t+(rhigh1);                       *0=100000+100000
        skpif[alu=0];
        error;

AplusBk:
        t_rhigh1;
        t_t+(rm1);
        rscr_77777C;
        t_t#(rscr);                         * 77777=100000+177777
        skpif[alu=0];
        error;

AplusBl:
        t_rm1;
        t_t+(rhigh1);
        rscr_77777C;
        t_t#(rscr);                         * 77777=177777+100000
        skpif[alu=0];
        error;
```

```
* August 9, 1977  12:30 PM
%
        TEST A-1
%

Aminus1:
        rscr_r0;
Aminus1L:                                     * CHECK A-1 IN A LOOP FOR ALL 16 BIT VALUES.
        t_(rscr)-1;
        t_t+1;
        t_t#(rscr);                           * t_ (rscr-1+1) xor rscr
        skpif[alu=0];
        error;
        rscr_(rscr)+1;                        * rscr IS LOOP CTRL
        dblBranch[.+1,Aminus1L,ALU=0];
```

```
*
%
        TEST A-B
%
aMinusB:
        t_r0;
        t_t-(r1);                               * T _ 0 -1
        t_t#(rm1);
        skpif[alu=0];
        error;                                  * T SHOULD HAVE BEEN -1

aMinusBb:
        t_r0;
        t_t-(rm1);                              * t_0 - (-1)
        t_t#(r1);
        skpif[alu=0];
        error;                                  * T SHOULD HAVE BEEN 1

aMinusBc:
        t_r0;
        t_t-(rhigh1);                           * t_ 0 - (100000)
        t_t#(rhigh1);
        skpif[alu=0];
        error;                                  * T SHOULD HAVE BEEN 100000

aMinusBd:
        t_100C;
        t_t-(r1);                               * t _ 100 -1
        rscr_77C;
        t_t#(rscr);
        skpif[alu=0];
        error;                                  * T SHOULD HAVE BEEN 77

aMinusBe:
        t_rscr_17C;
        t_t-(rscr);                             * t _ 17 - 17
        t_t#(r0);
        skpif[alu=0];
        error;                                  * T SHOULD HAVE BEEN 0

aMinusBf:
        t_rscr_177C;
        t_t-(rscr);                             * t _ 177 - 177
        t_t#(r0);
        skpif[alu=0];
        error;                                  * T SHOULD HAVE BEEN 0

aMinusBg:
        t_rscr_377C;
        t_t-(rscr);                             * t _ 377 - 377
        t_t#(r0);
        skpif[alu=0];
        error;                                  * T SHOULD HAVE BEEN 0

aMinusBh:
        t_rscr_400C;
        t_t-(rscr);                             * t _ 400 - 400
        t_t#(r0);
        skpif[alu=0];
        error;                                      * T SHOULD HAVE BEEN 0

aMinusBi:
        t_rscr_777C;
        t_t-(rscr);                                 * t _ 777 - 777
        t_t#(r0);
        skpif[alu=0];
        error;                                      * T SHOULD HAVE BEEN 0

aMinusBj:
        t_rscr_1777C;
        t_t-(rscr);                                 * t _ 1777 - 1777
        t_t#(r0);
        skpif[alu=0];
        error;                                      * T SHOULD HAVE BEEN 0
```

```
* January 18, 1979  5:24 PM
%
        TEST FAST BRANCH CONDITION: CARRY

        FIRST TEST WHEN WE KNOW THERE IS NO CARRY, THEN TRY TO
        GENERATE A CARRY AND BRANCH ON IT. NOTE: THIS CODE DEPENDS UPON
              THE ALU FUNCTIONS PLUS AND MINUS WORKING.

%
carryNo:
        t_(r0)+(r0);
        skpUnless[carry];
        error;                                      * r0 + r0 SHOULD NOT CAUSE CARRY

carryNob:
        t_rm1;
        t_t+(r0);
        skpUnless[carry];
        error;                                      * rm1 + r0 SHOULD NOT CAUSE CARRY

carryNoc:
        t_r10;
        t_t+(r01);
        skpUnless[carry];
        error;                                      * r10 + r01 SHOULD NOT CAUSE CARRY

carryNod:
        t_77777C;
        t_t+(r0);
        skpUnless[carry];
        error;                                      * 77777C + r0 SHOULD NOT CAUSE CARRY

carryNoe:
        t_r0;
        t_t+(r1);
        skpUnless[carry];
        error;                                      * r0 + r1 SHOULD NOT CAUSE CARRY

carryNof:
        t_r01;
        t_t-(r10);
        skpUnless[carry];
        error;                                      * r01 - r10 SHOULD NOT CAUSE CARRY

carryNog:
        t_r0;
        t_t-(r10);
        skpUnless[carry];
        error;                                      * r0 + r10 SHOULD NOT CAUSE CARRY

* NOW TRY SOMETHINGS THAT SHOULD GENERATE A CARRY

carryYes:
        t_rm1;
        t_t-(r1);
        skpif[carry];
        error;                                      * -1 -(+1) SHOULD CAUSE CARRY

carryYesb:
        t_rm1;
        t_t+(rhigh1);
        skpif[carry];
        error;                                  * -1 + 100000 SHOULD CAUSE CARRY

carryYesc:
        t_rm1;
        t_t+(r1);
        skpif[carry];
        error;                                  * -1 + 1 SHOULD CAUSE CARRY

carryYesd:
        t_rhigh1;
        t_t-(r01);
        skpif[carry];
        error;                                  * 100000 - r01 SHOULD CAUSE CARRY

carryYese:
        t_rhigh1;
        t_t+(rhigh1);
        skpif[carry];
        error;                                  * 100000 + 100000 SHOULD CAUSE CARRY

* NOW COMPLICATE THINGS INTERLEAVING ALU OPS W/ TESTS

carryOps:
        t_r0;
        t_t-(r1);                               * t_0-1
        skpUnless[carry],t_t+(rm1);             * t_-1+-1
        error;                                  * 0-1 SHOULD NOT CAUSE CARRY

carryOpsb:
        skpif[carry], t_t+(rhigh1);             * T _ -2+100000
        error;                                  * -1+-1 SHOULD CAUSE CARRY

carryOpsc:
        skpif[carry],t_t+(rhigh1);              * t_ 77776 + 100000
        error;                                  * -2 + 100000 SHOULD CAUSE CARRY

carryOpsd:
        skpUnless[carry];
        error;                                  * 77776 + 100000 SHOULD NOT CAUSE CARRY

carryOpse:
        t_rm1;
        t_t-(r1);                               * t_ -1-(+1)
        skpif[carry],t_t-(r01);                 * t_ -2 -r01
        error;                                  * -1-1 SHOULD CAUSE CARRY

carryOpsf:
        skpif[carry];
        error;                                  * 177776 - 52525 SHOULD CAUSE CARRY
```

```
* March 10, 1979  6:42 PM
* test the branch conditions when reschedule is ON

        reschedule;
        t_r0;
        t_t-(r1);                               * t_0-1
        skpUnless[carry],t_t+(rm1);             * t_-1+-1
        error;                                  * 0-1 SHOULD NOT CAUSE CARRY
carryOpsRb:
        skpif[carry], t_t+(rhigh1);             * T _ -2+100000
        error;                                  * -1+-1 SHOULD CAUSE CARRY

carryOpsRc:
        skpif[carry],t_t+(rhigh1);              * t_ 77776 + 100000
        error;                                  * -2 + 100000 SHOULD CAUSE CARRY

carryOpsRd:
        skpUnless[carry];
        error;                                  * 77776 + 100000 SHOULD NOT CAUSE CARRY

carryOpsRe:
        t_rm1;
        t_t-(r1);                               * t_ -1-(+1)
        skpif[carry],t_t-(r01);                 * t_ -2 -r01
        error;                                  * -1-1 SHOULD CAUSE CARRY

carryOpsRf:
        skpif[carry];
        error;                                  * 177776 - 52525 SHOULD CAUSE CARRY

        t_r0;
        skpif[ALU=0];
rescheq0br:
        error;
        t_r1;
        skpif[ALU#0];
reschne0br:
        error;
        skpif[r even], B_r0;
reschevenbr:
        error;
        skpif[r odd], B_r1;
reschoddbr:
        error;
        t_rhigh1;
        skpif[alu<0];
reschlt0br:
        error;
        t_r0;
        skpif[alu>=0];
reschge0br:
        error;


        noreschedule;
```

```
* September 15, 1978  11:38 AM
%
        TEST FREEZEBC FUNCTION

Generate the two different branch conditions and freeze them. Force the
carry to be explicitly different, see if the frozen branch is still there.
Unfreeze and make sure the expected results happen.


%
freezeBCtest:
    t_rm1;
    t_t+(r1);        * t_ 0 _ -1+1 (SHOULD CAUSE carry)
    skpif[carry],t_t+(r1),freezeBC;        * FREEZE[carry=1]
    error;

* carry WAS FROZEN. CONTINUE THAT WAY (carry_1, RESULt_0)
freezeBC1a:
    skpif[alu=0],freezeBC;      * ( result was ZERO)
    error;

    t_(rm1)+(rm1),freezeBC;          *  Would normally CAUSE RESULT <0
    skpif[alu>=0],freezeBC;    * ( result was ZERO)
    error;

freezeBC1b:
    t_(r1)+(r1),freezeBC;         * t_ 0+1 (carry Would NORMALLY BE ZERO)
    skpif[carry],freezeBC;
    error;

    t_tAND(rm1),freezeBC;           * t_1 and -1 (TEST IT A FEW MORE TIMES)
    skpif[carry],freezeBC;
    error;     * carry SHOULD HAVE BEEN 1

freezeBC1c:
    t_t+(r0),freezeBC;      * t_1+0
    skpif[carry],freezeBC;
    error;     * carry SHOULD HAVE BEEN 1

* ALLOW A NEW alu RESULT CONDITION, KEEP carry THE SAME(carry=1, RESULT=77777)
freezeBC2a:
    t_rm1,freezeBC;
    t_t+(rhigh1);        * carry_1, RESULt_77777
    t_(r0)+(r0),freezeBC;
    skpif[alu#0],freezeBC;      * result was 77777
    error;

freezeBC2b:
    t_t+(r0),freezeBC;      *  Would normally ZERO carry
    skpif[carry],freezeBC;       * carry SHOULD BE ONE
    error;

freezeBC2c:
    t_rm1,freezeBC;
    t_t+(r0),freezeBC;
    skpif[alu>=0],freezeBC;    * result was 77777
    error;

* FORCE carry_0, RESULt_0
```

```
	t_r0,freezeBC;
	t_t+(r0);	* t_0+0 (SHOULD CAUSE carry_0, RESULt_0)
	t_(rhigh1)+(rhigh1),freezeBC;	* Would NORMALLY CAUSE carry_1

freezeBC3a:
	skpUnless[carry],freezeBC;	* FREEZE IT AT ZERO
	error;	* EXPECTED 0 carry GOT 1 carry

	t_(r1)+(r1),freezeBC;
	skpif[alu=0],freezeBC;	* test it again just to see
	error;

freezeBC3b:
	t_(rm1)+(rm1),freezeBC;
	skpif[alu>=0],freezeBC;	* test it again just to see
	error;

* FORCE carry_0, RESULT _ -1
	t_(rm1)+(rm1);
	t_t+(r1);	* -2+1 ==> carry_0, RESULt_-1

freezeBC4a:
	t_t+(r1),freezeBC;	* -1+1 Would normally CAUSE carry_1
	skpUnless[carry],freezeBC;
	error;

freezeBC4b:
	t_(r1)+(r1),freezeBC;	*Would normally CAUSE alu>=0
	skpUnless[alu>=0],freezeBC;
	error;

freezeBC4c:
	skpUnless[alu=0];
	error;
```

* March 26, 1979 11:04 AM
%*++
++++++++++++++

overflowTest

Perform an exhaustive test of the overflow condition. Even though we expect the arithmetic result of RM+T to be identical to T+RM, we test all possible combinations since the arithmetic gets implemented inside a rather complicated chip.

The tables below show the aluA and aluB inputs, and **the carry out values for b0, b1.** Notice the contents of the table are not the sum of a,b, but the carry out values. The subtraction table shows the original input for B and then its converted value after the number gets converted to a twos complement value (the chip converts it to the twos complement form, then adds).

For Addition

B input=	00	01	10	11
A input				
00	00	00	00	00
01	00	01	00	11
10	00	00	10	10
11	00	11	10	11

Notice these values represent the carry out values for b0,b1 during addition. They presume carry-in to b1 is zero

%*++
++++++++++++++

```
mc[x01, 40000];
mc[x10, 100000];
mc[x11, 140000];
overflowTest:
        t _ t-t;
        t _ t + t;
        skpif[overflow'];
overflErr0:                                         * 0+0 should not cause overflow
        error;
        t_t-t;
        t_t+(x01);
        skpif[overflow'];
overflErr1:                                         * see 0 + 01 entry
        error;
        t _ t-t;
        t _ t + (x10);
        skpif[overflow'];
overflErr2:                                         * see 0 + 10 entry
        error;
        t_t-t;
        t_t+(x11);
        skpif[overflow'];
overflerr3: * see 0 + 11 entry
        error;
        t_rscr_x01;                                 * keep x01 in rscr for a while
        t _ t + (0c);
        skpif[overflow'];
overflErr4:                                         * see 01 + 0 entry
        error;
        t_rscr;
        t _ t + (x01);
        skpif[overflow];                            * FIRST TRY FOR OVERFLOW
overflErr5:                                         * see 01 + 01 entry
        error;
        t_rscr;
        t _ t + (x10);
        skpif[overflow'];
overFLerr6:
        error;                                  * see 01 + 10 entry
        t_rscr;
        t_t+(x11);
        skpif[overflow'];
overflErr7:                                     * see 01 + 11 entry
        error;

        t_rscr_x10;                             * keep x10 in rscr for a while
        t _ t + (0c);
        skpif[overflow'];
overflErr10:                                    * see 10 + 0 entry
        error;
        t_rscr;
        t _ t + (x01);
        skpif[overflow'];
overflErr11:                                    * see 10 + 01 entry
        error;
        t_rscr;
        t _ t + (x10);
        skpif[overflow];
overFLerr12:
        error;                                  * see 10 + 10 entry
        t_rscr;
        t_t+(x11);
        skpif[overflow];
overflErr13:                                    * see 10 + 11 entry
        error;

        t_rscr_x11;                             * keep x11 in rscr for a while
        t _ t + (0c);
        skpif[overflow'];
overflErr14:                                    * see 11 + 0 entry
        error;
        t_rscr;
        t _ t + (x01);
        skpif[overflow'];
overflErr15:                                    * see 11 + 01 entry
        error;
        t_rscr;
        t _ t + (x10);
        skpif[overflow];
overFLerr16:
        error;                                  * see 11 + 10 entry
        t_rscr;
        t_t+(x11);
        skpif[overflow'];
overflErr17:                                    * see 11 + 11 entry
        error;
overFlowTestXit:
goto[afterKernel1];
```

```
* INSERT[D1ALU.MC];
* TITLE[KERNEL2];
* INSERT[PREAMBLE.MC];
* Link test corrected by Frank Vest                    November 1, 1984  10:01 AM
* September 22, 1986  5:42 PM.  Added noops after labels: AfterTioa and
* afterLink to avoid some confusion.
top level;
beginKernel2:
%
                    January 20, 1978  3:13 PM
%

%
TEST                CONTENTS
cntRW               read and write CNT
cntFFrw             read and write CNT, load from FF
cntFcn              test CNT=0&+1 fast branch
NotAtest            test alu op, NOT A
NotBtest            test alu op, NOT B
AandBtest           test alu op, A AND B
AorBtest            test alu op, A OR B
LINKRW              read and write LINK
callTest            global and local subroutine calls
QtestRW             read and write Q, q lsh 1, q rsh 1
tioaTest            load and read tioa from FF and from bmux
STKPtestRW          read and write STKP, perform TIOA&STKP
rstkTest0           write different RM address from one read
%
%
January 18, 1979  2:07 PM
                    Add tioaTest
%
```

```
* October 19, 1978  5:22 PM
%
        TEST ALL THE BITS IN CNT: REMEMBER THAT CNT CAN BE LOADED FROM
             BOTH B AND FF.
%


cntRW:
       t _ cnt _ r0;                                * test loading cnt w/ 0
       rscr _ cnt;
       t _ t # (rscr);                              * t _ bits read from cnt # expected bits
       skpif[ALU=0];
cntErr1:                                            * t = bad bits, rscr = expected
       error;                                       * value of cnt

       t _ cnt _ rm1;                               * test loading cnt w/ -1
       rscr _ cnt;
       t _ t # (rscr);                              * t _ bits read from cnt # expected bits
       skpif[ALU=0];
cntErr2:                                            * t = bad bits, rscr = expected
       error;                                       * value of cnt

       t _ cnt _ r01;                               * test loading cnt w/ alternating 01
       rscr _ cnt;
       t _ t # (rscr);                              * t _ bits read from cnt # expected bits
       skpif[ALU=0];
cntErr3:                                            * t = bad bits, rscr = expected
       error;                                       * value of cnt

       t _ cnt _ r10;                               * test loading cnt w/ alternating 10
       rscr _ cnt;
       t _ t # (rscr);                              * t _ bits read from cnt # expected bits
       skpif[ALU=0];
cntErr4:                                            * t = bad bits, rscr = expected
       error;                                       * value of cnt

* October 19, 1978  8:33 PM
cntFFrw:                                            * TEST FF BITS FOR LOADING cnt
       cnt_1S;
       t_cnt;
       t_t#(r1);                                    * we set it to 1; check the val.
       skpif[ALU=0];
cntFFrw1:                                           * t=bad bits, 1=expected value
       error;

       cnt _ 2s;
       t_cnt;
       t _ t # (2c);
       skpif[ALU=0];
cntFFrw2:                                           * t = bad bits, 2 = expected value
       error;

       cnt _ 4s;
       t _ cnt;
       t _ t # (4c);
       skpif[ALU=0];
cntFFrw3:                                           * t = bad bits, 4 = expected
       error;
```

```
          cnt _ 10s;
          t _ cnt;
          t _ t # (10c);
          skpif[ALU=0];
cntFFrw4:                                    * t = bad bits, 10c = expected
          error;
```

```
* October 30, 1978  1:54 PM
%
           TEST cnt BY LOOPING FOR ALL VALUES OF cnt
          AT POINTS TESTED, cnt AND rscr SHOULD BE EQUAL.

rscr_cnt_-1;                                          -- test cnt for maximum iterations
WHILE cnt NE 0 DO
          cnt_cnt-1;
          IF rscr=0 THEN ERROR;
          rscr_rscr-1;
          ENDLOOP;
IF rscr NE 0 THEN ERROR
%
cntFcn:
          t _ rscr_cm1;                               * t _ rscr _ initial value into cnt
          cnt_t;                                      * cnt _ initial value

cntFcnIL:
          branch[cntFcnXitIL, cnt=0&-1], PD _ rscr;
          skpUnless[ALU=0], PD_rscr;
cntFcnErr1:                                           * value of rscr suggests we
          error;                                      * should have exited
          branch[cntFcnIL], rscr_(rscr)-1;
cntFcnXitIL:
          skpif[ALU=0];
cntFcnErr2:                                           * rscr#0. value of rscr suggests we
          error;                                      * should not have exited.

          cnt _ r0;                                   * test cnt for initial value = zero
          skpif[cnt=0&-1];
cntFcnErr3:                                           * didn't notice first value we loaded
          error;                                      * was zero
```

```
* August 31, 1977  12:48 PM
% Test not A, not B
%

NotAtest:
        t_not(A_r0);
        t_t#(rm1);
        skpif[ALU=0];
        error;                              * ~RO # RM

NotAb:
        t_not(A_rm1);
        t_t#(r0);
        skpif[ALU=0];
        error;                              * ~rm1 # r0

NotAc:
        t_not(A_r01);
        t_t#(r10);
        skpif[ALU=0];
        error;                              * ~r01 # r10

NotAd:
        t_not(A_r10);
        t_t#(r01);
        skpif[ALU=0];
        error;                              * ~r10 # r01

NotAe:
        rscr_177776C;
        t_not(A_r1);
        t_t#(rscr);
        skpif[ALU=0];
        error;                              * ~r1 # 177776

NotBtest:
        t_not(B_r0);
        t_t#(rm1);
        skpif[ALU=0];
        error;                              * ~RO # RM

NotBTestb:
        t_not(B_rm1);
        t_t#(r0);
        skpif[ALU=0];
        error;                              * ~rm1 # r0

NotBTestc:
        t_not(B_r01);
        t_t#(r10);
        skpif[ALU=0];
        error;                              * ~r01 # r10

NotBTestd:
        t_not(B_r10);
        t_t#(r01);
        skpif[ALU=0];
        error;                              * ~r10 # r01
```

```
NotBTeste:
        rscr_177776C;
        t_not(B_r1);
        t_t#(rscr);
        skpif[ALU=0];
        error;                                    * ~r1 # 177776

%
        Test A AND B
        Assume a,b source dont matter. Ie.,
        t_(b_t) and (a_r) =
        t_(a_t) and (b_r)
%
AandBtest:
        t_rm1;
        t_tAND(rm1);
        t_t#(rm1);
        skpif[ALU=0];
        error;                                    * (rm1 AND rm1) #rm1

        t_r01;
        t_tAND(r10);
        skpif[ALU=0];
        error;                                    * (r01 AND r10)

        t_r0;
        t_tAND(rm1);
        skpif[ALU=0];
        error;                                    * r0 AND rm1

%
        Test A orB.
        Assume same as AandB test.
%
AorBtest:
        t_rm1;
        t_tOR(r0);
        t_t#(rm1);
        skpif[ALU=0];
        error;                                    * (rm1 OR r0) # rm1

AorBtestb:
        t_r01;
        t_tOR(r10);
        t_t#(rm1);
        skpif[ALU=0];
        error;                                    * (r01 OR r10) # rm1

AorBtestc:
        t_rm1;
        t_tOR(rm1);
        t_t#(rm1);
        skpif[ALU=0];
        error;                                    * (rm1 OR rm1) # rm1

AorBtestd:
        t_r01;
        t_tOR(r01);
        t_t#(r01);
        skpif[ALU=0];
        error;                                  * (r01 OR r01) # r01

AorBteste:
        t_r10;
        t_tOR(r10);
        t_t#(r10);
        skpif[ALU=0];
        error;                                  * (r10 OR r10) # r10

AorBtestf:
        t_(r0)OR(r0);
        t_t#(r0);
        skpif[ALU=0];
        error;                                  * (r0 OR r0) # r0
```

```
* February 17, 1978  8:51 AM
%
        LINK READ/WRITE TEST + MINOR TEST OF CALL

        FOR I IN[0..7777B] DO
                LINK_I;
                CHECK_LINK;
                CHECK _ BITAND[CHECK,7777B];
                IF CHECK NE LINK THEN ERROR;
                ENDLOOP;
        minor test of LINK, call
%

linkRW:
        rscr_7777C;                             * BEGIN W/ MAX LINK VALUE & COUNT DOWN

linkL:
        link _ rscr;
        t _ link;
        t _ t and (77777C);                     * ISOLATE 15 BITS 'CAUSE OF DMUX DATA
        t_t#(rscr);
        skpif[alu=0];
linkErr1:
        error;                                  * LINK DOESN'T HAVE THE VALUE WE LOADED

        rscr_(rscr)-1;
        dblBranch[afterLink,linkL, alu<0];


afterLink:
        noop;
```

```
* November 3, 1978  6:40 PM
%
        TEST Q: READ AND WRITE

        FOR I IN [0..177777B] DO
                Q_I;
                t_Q XOR I;
                IF T #0 THEN error;
        ENDLOOP;
then test q lsh 1, q rsh 1 w/ selected values
%

QtestRW:
        rscr_r0;

QRWL:
        Q_(rscr);
        t_(A_rscr)#(B_Q);
        skpif[ALU=0];
QrwErr:
        error;
        rscr_(rscr)+1;
        dblBranch[.+1,QRWL,ALU=0];

* now check rsh1, lsh1
        q _ r0;
        q lsh 1;                        * q _ 0 lsh 1
        PD _ q;
        skpif[ALU=0];
qr0Lerr:                                * r0 lsh 1 should be zero
        error;

        q _ r01;
        q lsh 1;                        * q _ r01 lsh 1
        (q) # (r10);
        skpif[alu=0];
qr10Lerr:                               * r10 lsh1 should be r01. (zero fill)
        error;

        q _ rm1;
        q lsh 1;
        t _ cm2;
        (q) # t;
        skpif[ALU=0];
qrm1Lerr:                               * -1 lsh1 w/ zero fill should be -2
        error;

        q _ rhigh1;
        q lsh 1;                        * q _ 100000B lsh 1
        PD _ q;
        skpif[ALU=0];
qrhigh1Lerr:                            * rhigh1 (100000B) lsh1 w/ zero fill should
        error;                          * zero

        q _ r0;
        q rsh 1;                        * zero rsh1 should be zero
        PD _ q;
        skpif[ALU=0];
```

```
qr0Rerr:
        error;                                  * zero rsh1 should be zero

        q _ r10;
        q rsh 1;                                * q _ r10 rsh 1
        (q) # (r01);
        skpif[ALU=0];
qr10Rerr:                                       * r10 rsh 1 w/ zero fill
        error;                                  * should be r01

        q _ rm1;
        q rsh 1;                                * q _ -1 rsh 1;
        t _ 77777c;
        (q) # t;
        skpif[ALU=0];
qrm1Rerr:                                       * -1 rsh 1 w/ zero fill should be 77777B
        error;

        q _ rhigh1;
        q rsh 1;                                * q _ 100000B rsh 1
        t _ 40000C;
        (q) # t;
        skpif[ALU=0];
qrhigh1Rerr:                                    * rhigh1 rsh1 should be 40000B
        error;
```

```
* January 18, 1979  1:29 PM
%
                                        tioaTest
Test the processor's ability to read and write TIOA.  Write TIOAk from both FF constants and from RM.
%
tioaTest:
        t _ 377c;
        cnt _ t;
        rscr2_ t-t;
tioaL:
        tioa _ rscr2;                           * RSCR2 = value we load into Tioa
        call[getTioa];                          * rtn Tioa, still left justified, in t
        rscr _ (rscr2) # t;
        skpif[ALU=0];
tioaErr1:                                       * We wrote tioa w/ contents of rscr2, got
        error;                                  * back the value in t.  Bad bits in rscr.
        loopUntil[cnt=0&-1, tioaL], rscr2 _ (rscr2) + (b7);     * increment rscr2

* Here are device ddeclarations to keep micro happy. We use them to set Tioa directly from FF.
device[dvc5, b13!]; device[dvc6, b14!]; device[dvc7, b15!];
mc[tioa.0thru4C, b0,b1,b2,b3,b4];
mc[tioa.mask, 177400];

        tioa _ r0;                              * zero all the bis of tioa
        tioa[dvc7];                             * should set tioa[5:7] to 1
        call[getTioa];
        rscr _ (t) # (b7);                      * only one bit should be set
        skpif[ALU=0];
tiaErr2:                                        * tioa should be 1, (= 1 lshift 8 = 400)
        error;                                  * t = value of tioa, rscr = bad bits.

        tioa[dvc6];                             * should set tioa[5:7] to 2
        call[getTioa];
        rscr _ t # (b6);                        * tioa should be 2, (= 1 lshift 9 = 1000)
        skpif[ALU=0];
tioaErr3:
        error;                                  * rscr = bad bits, t = tioa left justified

        tioa[dvc5];                             * tioa should be 4 (= 1 lshift 10 = 2000)
        call[getTioa];
        rscr _ t # (b5);
        skpif[ALU=0];
tioaErr4:
        error;                                  * rscr = bad bits, t = tioa left justified

        tioa _ rm1;                             * all ones into tioa
        tioa[dvc7];
        call[getTioa];
        rscr _ tioa.0thru4C;
        rscr _ (rscr) or (b7);                  * only should have set tioa[5:7];
        rscr _ t # (q_rscr);                    * q = expected value
        skpif[ALU=0];
tioaErr5:
        error;                                  * t = tioa, left justified; rscr = bad bits, q = expected value

        tioa[dvc6];
        call[getTioa];                          * set tioa[5:7] to 2
        rscr _ tioa.0thru4C;
        rscr _ (rscr) or (b6);                  * only should have set tioa[5:7];
        rscr _ t # (q_rscr);                    * q = expected value
        skpif[ALU=0];
tioaErr6:                                       * q = expected value
        error;                                  * t = tioa, left justified; rscr = bad bits

        tioa[dvc5];
        call[getTioa];                          * set tioa[5:7] to 4
        rscr _ tioa.0thru4C;
        rscr _ (rscr) or (b5);                  * only should have set tioa[5:7];
        rscr _ t # (q_rscr);                    * q = expected value
        skpif[ALU=0];
tioaErr7:                                       * q = expected value
        error;                                  * t = tioa, left justified; rscr = bad bits

        branch[afterTioa];
getTioa: subroutine;
        t _ TIOA&STKP;
        return, t _ t and (177400C);            * isolate left byte
        top level;

afterTioa:
        noop;
```

```
* October 19, 1978  8:54 PM
%
        TEST STKP: READ AND WRITE

        FOR I IN[0..377B] DO
                STKP_I;
                t_TIOA&STKP[]
                t_t and (stkpMask);
                t_t XOR I;
                IF T # 0 THEN error;
        ENDLOOP;
%

STKPtestRW:
        t_r0;
        rscr_t;                                 * rscr = values loaded into stackp
        rscr2 _ t_377C;                         * MASK TO ISOLATE STACKP
        cnt_t;                                  * mask just happens to be count, too

stkpL:
        STKP_rscr;                              * LOAD STKP FROM rscr
        t _ (TIOA&STKP);
        t_t AND (rscr2);                        * READ AND MASK THE VALUE
        t_t#(rscr);
        skpif[ALU=0];
stkpErr:
        error;                                  * error: DIDN’T READ WHAT WE LOADED
        dblBranch[.+1,stkpL,CNT=0&-1],rscr_(rscr)+1;
```

```
* October 26, 1978  12:03 PM
%
        rstkFF                              Test the FF operation that replaces rstk with a value from the
FF field during rm Writing. Test each bitpath only.
%
rstkFF:
        q _ rmx0;                           * save rmx0
        rmx0 _ t-t;                         * background test rm location w/ zero
        t _ rmx7 _ cm1;                     * KEEP -1 IN RMX7, AND T
        rmx0 _ rmx7;                        * write into RM w/ rstk from FF field
        t # (rmx0);                         * compare target RM w/ expected value
        skpif[alu=0];
rstkFF0Err:                                 * can't write into rstk0 w/ ff
        error;

        rmx0 _ q;                           * restore old value
        q _ rmx1;                           * save rmx1
        rmx1 _ t-t;                         * background test rm location w/ zero
        rmx1 _ rmx7;                        * write into RM w/ rstk from FF field
        t # (rmx1);                         * compare target RM w/ expected value
        skpif[ALU=0];
rstkFF1Err:                                 * can't write into rstk1 w/ ff
        error;

        rmx1 _ q;                           * restore old value
        q _ rmx2;                           * save rmx2
        rmx2 _ t-t;                         * background test rm location w/ zero
        rmx2 _ rmx7;                        * write into RM w/ rstk from FF field
        t # (rmx2);                         * compare target RM w/ expected value
        skpif[ALU=0];
rstkFF2Err:                                 * can't write into rstk2 w/ ff
        error;

        rmx2 _ q;                           * restore old value
        q _ rmx4;                           * save rmx4
        rmx4 _ t-t;                         * background test rm location w/ zero
        rmx4 _ rmx7;                        * write into RM w/ rstk from FF field
        t # (rmx7);                         * compare target RM w/ expected value
        skpif[ALU=0];
rstkFF4Err:                                 * can't write rstk4 w/ ff
        error;

        rmx4 _ q;                           * restore old value
        q _ rmx10;                          * save rmx10
        rmx10 _ t-t;                        * background test rm location w/ zero
        rmx10 _ rmx7;                       * write into RM w/ rstk from FF field
        t # (rmx10);                        * compare target RM w/ expected value
        skpif[ALU=0];
rstkFF10Err:                                * can't write rstk10 w/ ff
        error;

        rmx10 _ q;
```

```
* October 26, 1978  6:14 PM
%
        rbaseFF                                 test the facility that changes the value of rbase when rm storing
occurs.
%
* sibling[FoosBrotherInRegion5, 5, foo]         * declare FoosBrotherInRegion5 as an RM
* location in rmRegion 5 with its rstk value the same as the one for foo. Eg., if foo is
* located at rm addr 17,,12 (rbase = 17, rstk = 12) then FoosBrotherInRegion5 is located
* at rm addr 5,,12
m[sibling,
        rm[#1, add[lshift[#2,4], and[17,ip[#3]]]]
        ];
sibling[rb0rm0, 0, rmx0]; sibling[rb1rm1, 1, rmx1]; sibling[rb2rm2, 2, rmx2];
sibling[rb4rm4, 4, rmx4]; sibling[rb10rm10, 10, rmx10];

rbaseFF:
        rbase _ rbase[defaultRegion];
        q _ rmx0;                               * save current value for "source" rm
        rb0rm0 _ t-t;                           * zero "destination" rm
        rmx0 _ cm1;                             * t _ "source rm" _ -1
        rb0rm0 _ rmx0;                          * "destin" rm (different rbase)_ source rm
        rbase _ 0s;                             * check the result. First fetch the value in
        t _ rmx0, RBASE _ rbase[defaultRegion]; * the destination rm, then compare it to
        t # (rmx0);                             * the source rm. An error means we didn't
        skpif[ALU=0];
rbaseFF0Err:                                    * succeed in writing rm with rbase_0 from
        error;                                  * ff field. t = real val, rmx0=expected val.

        rmx0 _ q;                               * restore old value
        q _ rmx1;                               * save current value for "source" rm
        t _ rmx1 _ cm1;                         * t _ "source rm" _ -1
        rb1rm1 _ t-t;                           * zero "destination" rm
        rb1rm1 _ rmx1;                          * "destin" rm (different rbase)_ source rm
        RBASE _ 1s;                             * check the result. First fetch the value in
        t _ rmx1, RBASE _ rbase[defaultRegion]; * the destination rm, then compare it to
        t # (rmx1);                             * the source rm. An error means we didn't
        skpif[ALU=0];
rbaseFF1Err:                                    * succeed in writing rm with rbase_0 from
        error;                                  * ff field. t = real val, rmx0=expected val.

        rmx1 _ q;                               * restore old value
        q _ rmx2;                               * save current value for "source" rm
        t _ rmx2 _ cm1;                         * t _ "source rm" _ -1
        rb2rm2 _ t-t;                           * zero "destination" rm
        rb2rm2 _ rmx2;                          * "destin" rm (different rbase)_ source rm
        RBASE _ 2s;                             * check the result. First fetch the value in
        t _ rmx2, RBASE _ rbase[defaultRegion]; * the destination rm, then compare it to
        t # (rmx2);                             * the source rm. An error means we didn't
        skpif[ALU=0];
rbaseFF2Err:                                    * succeed in writing rm with rbase_0 from
        error;                                  * ff field. t = real val, rmx0=expected val.

        rmx2 _ q;                               * restore old value
        q _ rmx4;                               * save current value for "source" rm
        t _ rmx4 _ cm1;                         * t _ "source rm" _ -1
        rb4rm4 _ t-t;                           * zero "destination" rm
        rb4rm4 _ rmx4;                          * "destin" rm (different rbase)_ source rm
        RBASE _ 4s;                             * check the result. First fetch the value in
        t _ rmx4, RBASE _ rbase[defaultRegion]; * the destination rm, then compare it to
        t # (rmx4);                             * the source rm. An error means we didn't
        skpif[ALU=0];
rbaseFF4Err:                                    * succeed in writing rm with rbase_0 from
        error;                                  * ff field. t = real val, rmx0=expected val.

        rmx4 _ q;                               * restore old value
        q _ rmx10;                              * save current value for "source" rm
        t _ rmx10 _ cm1;                        * t _ "source rm" _ -1
        rb10rm10 _ t-t;                         * zero "destination" rm
        rb10rm10 _ rmx10;
        RBASE _ 10s;                            * check the result. First fetch the value in
        t _ rmx10,RBASE _ rbase[defaultRegion]; * the destination rm, then compare it to
        t # (rmx10);                            * the source rm. An error means we didn't
        skpif[ALU=0];
rbaseFF10Err:                                   * succeed in writing rm with rbase_0 from
        error;                                  * ff field. t = real val, rmx0=expected val.


%
Test RSTK destination function:

        FOR I IN [0..7] DO
            FOR J IN [0..7] DO
                RBASE[I]_RBASE[J];
                t_RBASE[I];
                IF T#RBASE[J] THEN error;
        ENDLOOP; ENDLOOP;

Of course, this code is "expanded" inline rather than in a loop
%

*           FOR I IN [0..7] DO RBASE[0] _ RBASE[I]; (EXCEPT FOR _RBASE[0])
rstkTest0:
            rscr_3C;
            rscr2_4C;

            Q_r0;
            t_r0_r1;
            t#(Q);
            skpUnless[ALU=0], t_t#(r0);
            error;
            skpif[ALU=0];
            error;

            t_r0_rm1;
            t#(Q);
            skpUnless[ALU=0], t_t#(r0);
rstkTest02:
            error;
            skpif[ALU=0];
            error;

            t_r0_r01;
            t#(Q);
            skpUnless[ALU=0], t_t#(r0);
            error;
            skpif[ALU=0];
            error;
```

```
		t_r0_r10;
		t#(Q);
		skpUnless[ALU=0], t_t#(r0);
rstkTest04:
		error;
		skpif[ALU=0];
		error;

		t_r0_rhigh1;
		t#(Q);
		skpUnless[ALU=0], t_t#(r0);
		error;
		skpif[ALU=0];
		error;

		t_r0_rscr;
		t#(Q);
		skpUnless[ALU=0], t_t#(r0);
rstkTest06:
		error;
		skpif[ALU=0];
		error;

		t_r0_rscr2;
		t#(Q);
		skpUnless[ALU=0], t_t#(r0);
		error;
		skpif[ALU=0];
		error;
		r0_Q;

*		FOR I IN [0..7] DO RBASE[1] _ RBASE[I]; (EXCEPT FOR _RBASE[1])
rstkTest1:
		Q_r1;
		t_r1_r0;
		t#(Q);
		skpUnless[ALU=0], t_t#(r1);
		error;
		skpif[ALU=0];
		error;

		t_r1_rm1;
		t#(Q);
		skpUnless[ALU=0], t_t#(r1);
rstkTest12:
		error;
		skpif[ALU=0];
		error;

		t_r1_r01;
		t#(Q);
		skpUnless[ALU=0], t_t#(r1);
		error;
		skpif[ALU=0];
		error;

		t_r1_r10;
		t#(Q);
		skpUnless[ALU=0], t_t#(r1);
```

```
rstkTest14:
            error;
            skpif[ALU=0];
            error;

            t_r1_rhigh1;
            t#(Q);
            skpUnless[ALU=0], t_t#(r1);
            error;
            skpif[ALU=0];
            error;

            t_r1_rscr;
            t#(Q);
            skpUnless[ALU=0], t_t#(r1);
rstkTest16:
            error;
            skpif[ALU=0];
            error;

            t_r1_rscr2;
            t#(Q);
            skpUnless[ALU=0], t_t#(r1);
            error;
            skpif[ALU=0];
            error;
            r1_Q;

*           FOR I IN [0..7] DO RBASE[2] _ RBASE[I]; (EXCEPT FOR _RBASE[2])
rstkTest2:
            Q_rm1;
            t_rm1_r0;
            t#(Q);
            skpUnless[ALU=0], t_t#(rm1);
            error;
            skpif[ALU=0];
            error;

            t_rm1_r1;
            t#(Q);
            skpUnless[ALU=0], t_t#(rm1);
rstkTest22:
            error;
            skpif[ALU=0];
            error;

            t_rm1_r01;
            t#(Q);
            skpUnless[ALU=0], t_t#(rm1);
            error;
            skpif[ALU=0];
            error;

            t_rm1_r10;
            t#(Q);
            skpUnless[ALU=0], t_t#(rm1);
rstkTest24:
            error;
            skpif[ALU=0];
```

```
                error;

                t_rm1_rhigh1;
                t#(Q);
                skpUnless[ALU=0], t_t#(rm1);
                error;
                skpif[ALU=0];
                error;

                t_rm1_rscr;
                t#(Q);
                skpUnless[ALU=0], t_t#(rm1);
rstkTest26:
                error;
                skpif[ALU=0];
                error;

                t_rm1_rscr2;
                t#(Q);
                skpUnless[ALU=0], t_t#(rm1);
                error;
                skpif[ALU=0];
                error;
                rm1_Q;

*               FOR I IN [0..7] DO RBASE[3] _ RBASE[I]; (EXCEPT FOR _RBASE[3])
rstkTest3:
                Q_r01;
                t_r01_r0;
                t#(Q);
                skpUnless[ALU=0], t_t#(r01);
                error;
                skpif[ALU=0];
                error;

                t_r01_r1;
                t#(Q);
                skpUnless[ALU=0], t_t#(r01);
rstkTest32:
                error;
                skpif[ALU=0];
                error;

                t_r01_rm1;
                t#(Q);
                skpUnless[ALU=0], t_t#(r01);
                error;
                skpif[ALU=0];
                error;

                t_r01_r10;
                t#(Q);
                skpUnless[ALU=0], t_t#(r01);
rstkTest34:
                error;
                skpif[ALU=0];
                error;

                t_r01_rhigh1;
```

```
		t#(Q);
		skpUnless[ALU=0], t_t#(r01);
		error;
		skpif[ALU=0];
		error;

		t_r01_rscr;
		t#(Q);
		skpUnless[ALU=0], t_t#(r01);
rstkTest36:
		error;
		skpif[ALU=0];
		error;

		t_r01_rscr2;
		t#(Q);
		skpUnless[ALU=0], t_t#(r01);
		error;
		skpif[ALU=0];
		error;
		r01_Q;

*		FOR I IN [0..7] DO RBASE[4] _ RBASE[I]; (EXCEPT FOR _RBASE[4])
rstkTest4:
		Q_r10;
		t_r10_r0;
		t#(Q);
		skpUnless[ALU=0], t_t#(r10);
		error;
		skpif[ALU=0];
		error;

		t_r10_r1;
		t#(Q);
		skpUnless[ALU=0], t_t#(r10);
rstkTest42:
		error;
		skpif[ALU=0];
		error;

		t_r10_r01;
		t#(Q);
		skpUnless[ALU=0], t_t#(r10);
		error;
		skpif[ALU=0];
		error;

		t_r10_rm1;
		t#(Q);
		skpUnless[ALU=0], t_t#(r10);
rstkTest44:
		error;
		skpif[ALU=0];
		error;

		t_r10_rhigh1;
		t#(Q);
		skpUnless[ALU=0], t_t#(r10);
		error;
```

```
                skpif[ALU=0];
                error;

                t_r10_rscr;
                t#(Q);
                skpUnless[ALU=0], t_t#(r10);
rstkTest46:
                error;
                skpif[ALU=0];
                error;

                t_r10_rscr2;
                t#(Q);
                skpUnless[ALU=0], t_t#(r10);
                error;
                skpif[ALU=0];
                error;
                r10_Q;

*               FOR I IN [0..7] DO RBASE[5] _ RBASE[I]; (EXCEPT FOR _RBASE[5])
rstkTest5:
                Q_rhigh1;
                t_rhigh1_r0;
                t#(Q);
                skpUnless[ALU=0], t_t#(rhigh1);
                error;
                skpif[ALU=0];
                error;

                t_rhigh1_r1;
                t#(Q);
                skpUnless[ALU=0], t_t#(rhigh1);
rstkTest52:
                error;
                skpif[ALU=0];
                error;

                t_rhigh1_r01;
                t#(Q);
                skpUnless[ALU=0], t_t#(rhigh1);
                error;
                skpif[ALU=0];
                error;

                t_rhigh1_r10;
                t#(Q);
                skpUnless[ALU=0], t_t#(rhigh1);
rstkTest54:
                error;
                skpif[ALU=0];
                error;

                t_rhigh1_rm1;
                t#(Q);
                skpUnless[ALU=0], t_t#(rhigh1);
                error;
                skpif[ALU=0];
                error;
                t_rhigh1_rscr;
                t#(Q);
                skpUnless[ALU=0], t_t#(rhigh1);
rstkTest56:
                error;
                skpif[ALU=0];
                error;

                t_rhigh1_rscr2;
                t#(Q);
                skpUnless[ALU=0], t_t#(rhigh1);
                error;
                skpif[ALU=0];
                error;
                rhigh1_Q;

*               FOR I IN [0..7] DO RBASE[6] _ RBASE[I]; (EXCEPT FOR _RBASE[6])
rstkTest6:
                Q_rscr;
                t_rscr_r0;
                t#(Q);
                skpUnless[ALU=0], t_t#(rscr);
                error;
                skpif[ALU=0];
                error;

                t_rscr_r1;
                t#(Q);
                skpUnless[ALU=0], t_t#(rscr);
rstkTest62:
                error;
                skpif[ALU=0];
                error;

                t_rscr_r01;
                t#(Q);
                skpUnless[ALU=0], t_t#(rscr);
                error;
                skpif[ALU=0];
                error;

                t_rscr_r10;
                t#(Q);
                skpUnless[ALU=0], t_t#(rscr);
rstkTest64:
                error;
                skpif[ALU=0];
                error;

                t_rscr_rhigh1;
                t#(Q);
                skpUnless[ALU=0], t_t#(rscr);
                error;
                skpif[ALU=0];
                error;

                t_rscr_rm1;
                t#(Q);
                skpUnless[ALU=0], t_t#(rscr);
```

```
rstkTest66:
		error;
		skpif[ALU=0];
		error;

		t_rscr_rscr2;
		t#(Q);
		skpUnless[ALU=0], t_t#(rscr);
		error;
		skpif[ALU=0];
		error;
		rscr_Q;

*		FOR I IN [0..7] DO RBASE[7] _ RBASE[I]; (EXCEPT FOR _RBASE[7])
rstkTest7:
		Q_rscr2;
		t_rscr2_r0;
		t#(Q);
		skpUnless[ALU=0], t_t#(rscr2);
		error;
		skpif[ALU=0];
		error;

		t_rscr2_r1;
		t#(Q);
		skpUnless[ALU=0], t_t#(rscr2);
rstkTest72:
		error;
		skpif[ALU=0];
		error;

		t_rscr2_r01;
		t#(Q);
		skpUnless[ALU=0], t_t#(rscr2);
		error;
		skpif[ALU=0];
		error;

		t_rscr2_r10;
		t#(Q);
		skpUnless[ALU=0], t_t#(rscr2);
rstkTest74:
		error;
		skpif[ALU=0];
		error;

		t_rscr2_rhigh1;
		t#(Q);
		skpUnless[ALU=0], t_t#(rscr2);
		error;
		skpif[ALU=0];
		error;

		t_rscr2_rscr;
		t#(Q);
		skpUnless[ALU=0], t_t#(rscr2);
rstkTest76:
		error;
		skpif[ALU=0];
		error;

		t_rscr2_rm1;
		t#(Q);
		skpUnless[ALU=0], t_t#(rscr2);
		error;
		skpif[ALU=0];
		error;
		rscr2_Q;

goto[afterKernel2];
```

```
* INSERT[D1ALU.MC];
* TITLE[KERNEL3];
top level;
beginKernel3: noop;

%
TEST                CONTENTS
SHCtestRW           Read and write SHC
Rlsh                test RM shiftlmask, lsh r[i]; 0<=i<=15
Tlsh                test T shiftlmask, lsh t[i]; 0<=i<=15
Rrsh                test RM shiftlmask, rsh r[i]; 0<=i<=15
Trsh                test T shiftlmask, rsh t[i]; 0<=i<=15
TRlcyTest           test T R lcy[i]; 0<=i<=15. (cycle 0,,1 and 177777,,177776)
RTlcyTest           test R T lcy[i]; 0<=i<=15. (cycle 0,,1 and 177777,,177776)
rcy16, lcy16        test 16 bit cycles with selected bit values
cycleTest           Test 32 bit cycles by generating possible r,t, count values
RFWFtest            test RF_ and WF_
aluRSH              test alu right shift (ie., H3 _ ALU rightshift 1)
aluRCY              test alu right cycle (ie., H3 _ ALU rightCycle 1)
aluARSH             test alu arithmetic right shift (ie., H3 _ ALU rightshift 1, sign preserved)
aluLSH              test alu left shift
aluLCY              test alu left cycle
aluSHTEST           exhaustive test of alu shifts
%
```

```
* August 9, 1977  12:33 PM
%
	TEST SHC: READ AND WRITE

	FOR I IN[0..177777B] DO
		SHC_I;
		T_SHC XOR I;
		IF T#0 THEN ERROR;
	ENDLOOP;

	Note: ShC is a 16 bit register AND the upper three bits [0..2] are not
	used by the shifter!
%
SHCtestRW:
	rscr_r0;
SHCRWL:
	SHC_rscr;
	t _ SHC;
	t_(rscr)#(t);
	branch[.+2,ALU=0];
	error;
	rscr_(rscr)+1;
	loopUntil[ALU=0,SHCRWL];

% TEST THE SHIFTER
	MAKE SURE THAT ALL SHIFTS WORK PROPER AMOUNT
	MAKE SURE ALL MASKS WORK
	MAKE SURE SHIFTS AND MASKS WORK TOGETHER
	These tests work by left (or right) shifting bit15 (bit 0) 0 thru
	15 times. rscr or rscr2 holds the expected value. The result is XOR'd
	with the expected value and those bits are placed in T. If t #0 there
	has been an error.

	test order:
	R shift left
	T shift left
	R shift right
	T shift right

	R T cycle left
	T R cycle left
	R T cycle right
	T R cycle right
	Note: The cycle tests are duplicated with the bits inverted (eg.,
	bit15 {bit0} is zero and all other bits are one.
%

Rlsh:
	t_r1;
	rscr_t;
	t_B15;
	rscr _ lsh[rscr,0];
	t_t#(rscr);
	skpif[ALU=0];
	error;

	t_r1;
	rscr_t;
	t_B14;
```

```
        rscr _ lsh[rscr,1];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Rlsh2:
        t_r1;
        rscr_t;
        t_B13;
        rscr _ lsh[rscr,2];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_r1;
        rscr_t;
        t_B12;
        rscr _ lsh[rscr,3];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Rlsh4:
        t_r1;
        rscr_t;
        t_B11;
        rscr _ lsh[rscr,4];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_r1;
        rscr_t;
        t_B10;
        rscr _ lsh[rscr,5];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Rlsh6:
        t_r1;
        rscr_t;
        t_B9;
        rscr _ lsh[rscr,6];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_r1;
        rscr_t;
        t_B8;
        rscr _ lsh[rscr,7];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Rlsh8:
        t_r1;
        rscr_t;
```

```
	t_B7;
	rscr _ lsh[rscr,10];
	t_t#(rscr);
	skpif[ALU=0];
	error;

	t_r1;
	rscr_t;
	t_B6;
	rscr _ lsh[rscr,11];
	t_t#(rscr);
	skpif[ALU=0];
	error;

Rlsh10:
	t_r1;
	rscr_t;
	t_B5;
	rscr _ lsh[rscr,12];
	t_t#(rscr);
	skpif[ALU=0];
	error;

	t_r1;
	rscr_t;
	t_B4;
	rscr _ lsh[rscr,13];
	t_t#(rscr);
	skpif[ALU=0];
	error;

Rlsh12:
	t_r1;
	rscr_t;
	t_B3;
	rscr _ lsh[rscr,14];
	t_t#(rscr);
	skpif[ALU=0];
	error;

	t_r1;
	rscr_t;
	t_B2;
	rscr _ lsh[rscr,15];
	t_t#(rscr);
	skpif[ALU=0];
	error;

Rlsh14:
	t_r1;
	rscr_t;
	t_B1;
	rscr _ lsh[rscr,16];
	t_t#(rscr);
	skpif[ALU=0];
	error;

	t_r1;
	rscr_t;
```

```
t_RHIGH1;
rscr _ lsh[rscr,17];
t_t#(rscr);
skpif[ALU=0];
error;
```

```
* October 20, 1978  10:32 AM
Tlsh:

        t_rscr_B15;
        noop;
        t_lsh[t,0];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_r1;
        rscr_B14;
        t_lsh[t,1];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Tlsh2:
        t_r1;
        rscr_B13;
        t_lsh[t,2];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_r1;
        rscr_B12;
        t_lsh[t,3];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Tlsh4:
        t_r1;
        rscr_B11;
        t_lsh[t,4];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_r1;
        rscr_B10;
        t_lsh[t,5];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Tlsh6:
        t_r1;
        rscr_B9;
        t_lsh[t,6];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_r1;
        rscr_B8;
        t_lsh[t,7];
        t_t#(rscr);
```

```
	skpif[ALU=0];
	error;

Tlsh8:
	t_r1;
	rscr_B7;
	t_lsh[t,10];
	t_t#(rscr);
	skpif[ALU=0];
	error;

	t_r1;
	rscr_B6;
	t_lsh[t,11];
	t_t#(rscr);
	skpif[ALU=0];
	error;

Tlsh10:
	t_r1;
	rscr_B5;
	t_lsh[t,12];
	t_t#(rscr);
	skpif[ALU=0];
	error;

	t_r1;
	rscr_B4;
	t_lsh[t,13];
	t_t#(rscr);
	skpif[ALU=0];
	error;

Tlsh12:
	t_r1;
	rscr_B3;
	t_lsh[t,14];
	t_t#(rscr);
	skpif[ALU=0];
	error;

	t_r1;
	rscr_B2;
	t_lsh[t,15];
	t_t#(rscr);
	skpif[ALU=0];
	error;

Tlsh14:
	t_r1;
	rscr_B1;
	t_lsh[t,16];
	t_t#(rscr);
	skpif[ALU=0];
	error;

	t_r1;
	rscr_RHIGH1;
	t_lsh[t,17];
```

```
t_t#(rscr);
skpif[ALU=0];
error;
```

```
* October 20, 1978  10:12 AM
%
        KEEP 100000 IN Q FOR THESE TESTS !!!
%
**Rrsh**:
        Q_RHIGH1;
        GOTO[Rrsh1];                                    * Temporary EXPEDIENT
        rscr_Q;
        t_RHIGH1;
        rscr _ rsh[rscr,0];
        t_t#(rscr);
        skpif[ALU=0];
        error;

**Rrsh1**:
        rscr_Q;
        t_B1;
        rscr _ rsh[rscr,1];
        t_t#(rscr);
        skpif[ALU=0];
        error;

**Rrsh2**:
        rscr_Q;
        t_B2;
        rscr _ rsh[rscr,2];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        rscr_Q;
        t_B3;
        rscr _ rsh[rscr,3];
        t_t#(rscr);
        skpif[ALU=0];
        error;

**Rrsh4**:
        rscr_Q;
        t_B4;
        rscr _ rsh[rscr,4];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        rscr_Q;
        t_B5;
        rscr _ rsh[rscr,5];
        t_t#(rscr);
        skpif[ALU=0];
        error;


**Rrsh6**:
        rscr_Q;
        t_B6;
        rscr _ rsh[rscr,6];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        rscr_Q;
        t_B7;
        rscr _ rsh[rscr,7];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Rrsh8:
        rscr_Q;
        t_B8;
        rscr _ rsh[rscr,10];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        rscr_Q;
        t_B9;
        rscr _ rsh[rscr,11];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Rrsh10:
        rscr_Q;
        t_B10;
        rscr _ rsh[rscr,12];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        rscr_Q;
        t_B11;
        rscr _ rsh[rscr,13];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Rrsh12:
        rscr_Q;
        t_B12;
        rscr _ rsh[rscr,14];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        rscr_Q;
        t_B13;
        rscr _ rsh[rscr,15];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Rrsh14:
        rscr_Q;
        t_B14;
        rscr _ rsh[rscr,16];
        t_t#(rscr);
```

```
skpif[ALU=0];
error;

rscr_Q;
t_B15;
rscr _ rsh[rscr,17];
t_t#(rscr);
skpif[ALU=0];
error;
```

```
* October 20, 1978  10:13 AM
Trsh:

        GOTO[Trshift1];                              * Temporary EXPEDIENT
        t_rscr_RHIGH1;
        NOOP;
        T_rsh[t,0];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Trshift1:
        t_rhigh1;
        rscr_B1;
        t_rsh[t,1];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Trsh2:
        t_rhigh1;
        rscr_B2;
        t_rsh[t,2];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_rhigh1;
        rscr_B3;
        t_rsh[t,3];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Trsh4:
        t_rhigh1;
        rscr_B4;
        t_rsh[t,4];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_rhigh1;
        rscr_B5;
        t_rsh[t,5];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Trsh6:
        t_rhigh1;
        rscr_B6;
        t_rsh[t,6];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_rhigh1;
        rscr_B7;
        t_rsh[t,7];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Trsh8:
        t_rhigh1;
        rscr_B8;
        t_rsh[t,10];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_rhigh1;
        rscr_B9;
        t_rsh[t,11];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Trsh10:
        t_rhigh1;
        rscr_B10;
        t_rsh[t,12];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_rhigh1;
        rscr_B11;
        t_rsh[t,13];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Trsh12:
        t_rhigh1;
        rscr_B12;
        t_rsh[t,14];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_rhigh1;
        rscr_B13;
        t_rsh[t,15];
        t_t#(rscr);
        skpif[ALU=0];
        error;

Trsh14:
        t_rhigh1;
        rscr_B14;
        t_rsh[t,16];
        t_t#(rscr);
        skpif[ALU=0];
        error;

        t_rhigh1;
```

```
rscr_B15;
t_rsh[t,17];
t_t#(rscr);
skpif[ALU=0];
error;
```

```
* October 20, 1978  10:14 AM
%
        These tests work by cycling by 0, 1, ...17B. The predicted result
        is kept in RSCR2 and the actual result XOR's w/ predicted result is
        kept in T. Note that each test is done twice: once w/ one "1" bit and all
        the rest "0" bits, and once w/ one "0" bit and all the rest "1" bits.

        FOR THESE TESTS WE WILL REDEFINE R01 TO BE RM2 (-2)!
%
TRlcyTest:
        RM[rm2,IP[R01]];
        rm2 _ CM2;

        t_r0;
        rscr2_B15;                              * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,0];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB15;                             * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,0];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r0;
        rscr2_B14;                              * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,1];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB14;                             * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,1];
        t_t#(rscr2);
        skpif[alu=0];
        error;

TRlcy2:
        t_r0;
        rscr2_B13;                              * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,2];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB13;                             * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,2];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r0;
        rscr2_B12;                              * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,3];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB12;                        * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,3];
        t_t#(rscr2);
        skpif[alu=0];
        error;

TRlcy4:
        t_r0;
        rscr2_B11;                         * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,4];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB11;                        * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,4];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r0;
        rscr2_B10;                         * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,5];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB10;                        * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,5];
        t_t#(rscr2);
        skpif[alu=0];
        error;

TRlcy6:
        t_r0;
        rscr2_B9;                          * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,6];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB9;                         * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,6];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r0;
        rscr2_B8;                          * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,7];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB8;                         * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,7];
        t_t#(rscr2);
        skpif[alu=0];
        error;

**TRlcy8**:
        t_r0;
        rscr2_B7;                         * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,10];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB7;                        * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,10];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r0;
        rscr2_B6;                         * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,11];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB6;                        * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,11];
        t_t#(rscr2);
        skpif[alu=0];
        error;

**TRlcy10**:
        t_r0;
        rscr2_B5;                         * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,12];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB5;                        * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,12];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r0;
        rscr2_B4;                         * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,13];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB4;                        * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,13];
        t_t#(rscr2);
        skpif[alu=0];
        error;

**TRlcy12**:
        t_r0;
        rscr2_B3;                         * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,14];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB3;                          * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,14];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r0;
        rscr2_B2;                           * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,15];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB2;                          * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,15];
        t_t#(rscr2);
        skpif[alu=0];
        error;

TRlcy14:
        t_r0;
        rscr2_B1;                           * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,16];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB1;                          * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,16];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r0;
        rscr2_B0;                           * RSCR2 _ PREDICTED RESULT
        t_lcy[t,r1,17];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm1;
        rscr2_NB0;                          * RSCR2 _ PREDICTED RESULT
        t_lcy[t,rm2,17];
        t_t#(rscr2);
        skpif[alu=0];
        error;
```

```
*
*	October 20, 1978  10:06 AM
RTlcyTest:					*RSCR2 HOLDS THE PREDICTED RESULT, T HOLDS
ACTUAL RESULT

	t_r1;					* rscr2, T _ [0,1] LCY[0]
	rscr2_B15;				* rscr2 _ PREDICTED RESULT
	t_lcy[r0,t,0];
	t_t#(rscr2);
	skpif[alu=0];
	error;
	t_rm2;
	rscr2_NB15;				* rscr2 _ PREDICTED RESULT
	t_lcy[rm1,t,0];
	t_t#(rscr2);
	skpif[alu=0];
	error;

	t_r1;
	rscr2_B14;				* rscr2 _ PREDICTED RESULT
	t_lcy[r0,t,1];
	t_t#(rscr2);
	skpif[alu=0];
	error;
	t_rm2;
	rscr2_NB14;				* rscr2 _ PREDICTED RESULT
	t_lcy[rm1,t,1];
	t_t#(rscr2);
	skpif[alu=0];
	error;

RTlcy2:
	t_r1;
	rscr2_B13;				* rscr2 _ PREDICTED RESULT
	t_lcy[r0,t,2];
	t_t#(rscr2);
	skpif[alu=0];
	error;
	t_rm2;
	rscr2_NB13;				* rscr2 _ PREDICTED RESULT
	t_lcy[rm1,t,2];
	t_t#(rscr2);
	skpif[alu=0];
	error;

	t_r1;
	rscr2_B12;				* rscr2 _ PREDICTED RESULT
	t_lcy[r0,t,3];
	t_t#(rscr2);
	skpif[alu=0];
	error;
	t_rm2;
	rscr2_NB12;				* rscr2 _ PREDICTED RESULT
	t_lcy[rm1,t,3];
	t_t#(rscr2);
	skpif[alu=0];
	error;

RTlcy4:
        t_r1;
        rscr2_B11;                          * rscr2 _ PREDICTED RESULT
        t_lcy[r0,t,4];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm2;
        rscr2_NB11;                         * rscr2 _ PREDICTED RESULT
        t_lcy[rm1,t,4];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r1;
        rscr2_B10;                          * rscr2 _ PREDICTED RESULT
        t_lcy[r0,t,5];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm2;
        rscr2_NB10;                         * rscr2 _ PREDICTED RESULT
        t_lcy[rm1,t,5];
        t_t#(rscr2);
        skpif[alu=0];
        error;

RTlcy6:
        t_r1;
        rscr2_B9;                           * rscr2 _ PREDICTED RESULT
        t_lcy[r0,t,6];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm2;
        rscr2_NB9;                          * rscr2 _ PREDICTED RESULT
        t_lcy[rm1,t,6];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r1;
        rscr2_B8;                           * rscr2 _ PREDICTED RESULT
        t_lcy[r0,t,7];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm2;
        rscr2_NB8;                          * rscr2 _ PREDICTED RESULT
        t_lcy[rm1,t,7];
        t_t#(rscr2);
        skpif[alu=0];
        error;

RTlcy8:
        t_r1;
        rscr2_B7;                           * rscr2 _ PREDICTED RESULT
        t_lcy[r0,t,10];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm2;
        rscr2_NB7;                  * rscr2 _ PREDICTED RESULT
        t_lcy[rm1,t,10];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r1;
        rscr2_B6;                   * rscr2 _ PREDICTED RESULT
        t_lcy[r0,t,11];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm2;
        rscr2_NB6;                  * rscr2 _ PREDICTED RESULT
        t_lcy[rm1,t,11];
        t_t#(rscr2);
        skpif[alu=0];
        error;

RTlcy10:
        t_r1;
        rscr2_B5;                   * rscr2 _ PREDICTED RESULT
        t_lcy[r0,t,12];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm2;
        rscr2_NB5;                  * rscr2 _ PREDICTED RESULT
        t_lcy[rm1,t,12];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r1;
        rscr2_B4;                   * rscr2 _ PREDICTED RESULT
        t_lcy[r0,t,13];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm2;
        rscr2_NB4;                  * rscr2 _ PREDICTED RESULT
        t_lcy[rm1,t,13];
        t_t#(rscr2);
        skpif[alu=0];
        error;

RTlcy12:
        t_r1;
        rscr2_B3;                   * rscr2 _ PREDICTED RESULT
        t_lcy[r0,t,14];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm2;
        RSCR2_NB3;                  * RSCR2 _ PREDICTED RESULT
        t_lcy[rm1,t,14];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r1;
        RSCR2_B2;                                  * RSCR2 _ PREDICTED RESULT
        t_lcy[r0,t,15];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm2;
        RSCR2_NB2;                                 * RSCR2 _ PREDICTED RESULT
        t_lcy[rm1,t,15];
        t_t#(rscr2);
        skpif[alu=0];
        error;

RTlcy14:
        t_r1;
        RSCR2_B1;                                  * RSCR2 _ PREDICTED RESULT
        t_lcy[r0,t,16];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm2;
        RSCR2_NB1;                                 * RSCR2 _ PREDICTED RESULT
        t_lcy[rm1,t,16];
        t_t#(rscr2);
        skpif[alu=0];
        error;

        t_r1;
        RSCR2_B0;                                  * RSCR2 _ PREDICTED RESULT
        t_lcy[r0,t,17];
        t_t#(rscr2);
        skpif[alu=0];
        error;
        t_rm2;
        RSCR2_NB0;                                 * RSCR2 _ PREDICTED RESULT
        t_lcy[rm1,t,17];
        t_t#(rscr2);
        skpif[alu=0];
        error;

RTlcyDone:
        r01 _ NOT(r10);                            * REDEFINE r01 !!!!!!!
```

```
* November 3, 1978  6:43 PM
%
        rcy16, lcy16                        Test the 16 bit cycles with selected
bit values. This is not an exhaustive test.
%
rcyTest:                                    * test 16 bit right cycle
        t _ rcy[r01, r01, 1];
        t # (r10);
        skpif[ALU=0];
rcy16Err1:                                  * r10 rcy 1 should be r01
        error;

        t _ r01;
        t _ rcy[t, t, 1];                   * try it again from t
        t # (r10);
        skpif[ALU=0];
rcy16Err2:                                  * r10 rcy 1 should be r01. (done from
        error;                              * t this time)

        t _ rcy[r1, r1, 1];
        t # (rhigh1);
        skpif[ALU=0];
rcy16Err3:                                  * 1 rcy 1 should be 100000B
        error;

        t _ r1;
        t _ rcy[t, t, 1];
        t # (rhigh1);
        skpif[ALU=0];
rcy16Err4:                                  * 1 rcy 1 should be 100000B. (done from t
        error;                              * time).

        t _ rcy[r10, r10, 1];
        t # (r01);
        skpif[ALU=0];
rcy16Err5:                                  * r10 rcy 1 should be r01
        error;

        t _ r10;
        t _ rcy[t, t, 1];                   * t _ r10 rcy 1
        t # (r01);
        skpif[ALU=0];
rcy16Err6:                                  * r10 rcy 1 should be r01
        error;                              * done from t this time.

        t _ rcy[r01, r01, 2];
        t # (r01);
        skpif[ALU=0];
rcy16Err7:                                  * r01 rcy 2 should be r01
        error;

        t _ rcy[r01, r01, 3];
        t # (r10);
        skpif[ALU=0];
rcy16Err8:                                  * r01 rcy 3 should be r10
        error;

        t _ rcy[r01, r01, 10];
        t # (r01);
        skpif[ALU=0];
rcy16Err9:                                  * r01 rcy 10 should be r01
        error;

lcyTest:
        t _ lcy[r01, r01, 1];
        t # (r10);
        skpif[ALU=0];
lcy16Err1:                                  * r01 lcy 1 should be r10
        error;

        t _ lcy[rhigh1, rhigh1, 1];
        t # (r1);
        skpif[ALU=0];
lcy16Err2:                                  * 100000B lcyd 1 should be 1
        error;

        t _ lcy[r1, r1, 1];
        t#(2c);
        skpif[ALU=0];
lcy16Err3:                                  * 1 lcy 1 should be 2
        error;

        t _ lcy[r10, r10, 1];               * t _ r10 lcy 1
        t # (r01);
        skpif[ALU=0];
lcy16Err4:                                  * r10 lcy 1 should be r01
        error;

        t _ lcy[r10, r10, 2];
        t # (r10);
        skpif[ALU=0];
lcy16Err5:                                  * r10 lcy 2 should be r10
        error;

        t  _ lcy[r10, r10, 3];
        t # (r01);
        skpif[ALU=0];
lcy16Err6:                                  * r10 lcy 3 should be r01
        error;

        t _ lcy[r10, r10, 4];
        t # (r10);
        skpif[ALU=0];
lcy16Err7:                                  * r10 lcy 4 should be r10
        error;

        t _ lcy[r10, r10, 10];
        t # (r10);
        skpif[ALU=0];
lcy16Err8:                                  * r10 lcy 10 should be r10
        error;
```

* November 13, 1978 10:40 AM
%

cycleTest Test the cycle machinery by generating all possible values for the r, t, and count fields in Shc (6 bits). The 16-bit data patterns must test all possible starting positions (ie., bit 0, bit 1, ...). Furthermore, set the "other word" to all 1s when single one-bits are being tested, and set it to all zeros when single zero bits are being tested. Since we test cycling, we don’t set any of the mask fields in ShC.

```
SHC: TYPE = MACHINE DEPENDENT RECORD[
        ShifterIgnores: IN[0..3],               -- bits 0, 1
        a: IN [0..1],                           -- bit 2, shA select. 1 ==> "select T"
        b: IN [0..1],                           -- bit 3, shB select. 1 ==> "select T"
        Count: IN [0..17B],                     -- bits 4:7, shift count
        RMask: IN [0..17B],                     -- bits 8:11
        LMask: IN [0..17B],                     -- bits 12:15
        ];
shcVals: IN [0..77B];                           -- iterate thru all possible counts, sha, shb

FOR pats IN NPats DO
        FOR shcVal In SHCVals DO
                Shc.a _ shcVals AND 40B;
                Shc.b _ shcVals AND 20B;
                Shc.count _ shcVals AND 17B;
                r _ getPattern[pats];
                t _ IF numberOfZeroBitsGr1[r] THEN -1 ELSE 0;
                result _ doShift[];
                expected _ simulateCycle[t,r,shcVal];
                IF result # expected THEN SIGNAL BadShift[result, expected, shcVals];
                ENDLOOP;                        -- end of shcVals loop
        ENDLOOP;                                -- end of pats loop
simulateCycle: PROCEDURE[t, r: WORD, shcVal: SHCVals] RETURNS [expected: WORD] =
        BEGIN
        tCycle: CARDINAL = 60B;                 -- 2 highest bits in shcVal are 1
        rCycle: CARDINAL = 0;                   -- 2 hightest bits in shcVal are 0
        trCycle: CARDINAL = 40B;                -- highest bit in shcVal is 1
        rtCycle: CARDINAL = 20BB;               -- 2nd highest bit in ShcVal is 1
        shAB _ shcVal AND 60B;
        SELECT shAB INTO
                tCycle=> BEGIN
                        left _right_t; END,
                rCycle=> BEGIN
                        left _ right _ r; END,
                trCycle=> BEGIN
                        left _ t, right _ r; END,
                rtCycle=> BEGIN
                        left _ r;
                        right _ t; END,
                END;
        shiftCount _ shcVal AND 17B;
        saveMask _ SELECT shiftCount INTO
                1=>100000;
                2=>140000B;
                3=>160000B;
                4=>170000B;
                5=>174000B;
                6=>176000B;
                7=>177000B;
                8=>177400B;
                9=>177600B;
                10=>177700B;
                11=>177740B;
                12=>177760B;
                13=>177770B;
                14=>177774B;
                15=>177776B;
                0=>177777B,
                END;
        savedValue _ left AND saveMask;
        right _ LeftShift[right, shiftCount];
        savedValue _ RightShift[right, 16-shiftCount];
        result _ savedValue OR left;
        END;
numberOfZeroBitsGr1: PROCEDURE[ x: WORD] RETURNS[result: BOOLEAN] =
        BEGIN
        count _ 0;
        FOR i IN [0..15] DO
                IF (x AND 1) =0 THEN count _ count + 1;
                x _ RightShift[x,1];
                ENDLOOP;
        result _ IF count >1 THEN TRUE ELSE FALSE;
        END;

%
```

```
%
November 17, 1978  2:39 PM

TEST RF AND WF

ShC: TYPE = MACHINE DEPENDENT RECORD [
        IGNORE: TYPE = [0..7B]
        SHIFTCOUNT: TYPE = [0..37B]
        RMASK: TYPE = [0..17B]
        LMASK: TYPE = [0..17B]
        ]

MesaDescriptor: TYPE = MACHINE DEPENDENT RECORD[ -- this is the value stored w/ rf_, wf_
        IGNORE: TYPE = [0..377B] -- IGNORE FIRST BYTE
        POS: TYPE = [0..17B]           -- RIGHT SHIFT OF POS WILL RIGHT JUSTIFY THE FIELD
        SIZE: TYPE = [0..17B]          -- LENGTH OF FIELD IN BITS
        ]

THIS TEST PROCEEDS BY WRITING ShC W/ ALL POSSIBLE RF AND WF VALUES. THEN
ShC IS READ AND CHECKED TO MAKE SURE THAT IT WAS LOADED PROPERLY.

        FOR I IN [0..377B] DO
               RF_I;
               RSCR_SHC;
               SIZE _ I AND 17B;
               POS _ BITSHIFT[I,-4] AND 17B;
               IF RSCR.LMASK # (16-SIZE-1) THEN ERROR;               -- BAD LMASK
               IF RSCR.RMASK # 0 THEN ERROR;              -- BAD RMASK
               IF RSCR.SHIFTCOUNT # (16+pos+size+1) THEN ERROR;                -- BAD SHIFT COUNT
 (Actually this computation isn't quite right.
* let count = 16+pos+size+1. realCount _ (count and 17b).
* IF (realCount and 17b) #0 then realCount _ realCount OR 20B. This funny computation
* accommodates hardware limitations associated w/ carry across boards.

               -- now test wf
               WF_I;
               RSCR_SHC;
               IF RSCR.RMASK # (16-POS-SIZE-1) THEN ERROR;          -- BAD RMASK
               IF RSCR.LMASK NE POS THEN ERROR;         -- BAD LMASK
               IF RSCR.SHIFTCOUNT # (16-pos-size-1) THEN ERROR;                -- BAD SHIFT COUNT

        ENDLOOP;
%

RM[r4BitMsk,IP[R01]];
r4BitMsk _ 17C;                               * RENAME R01 AS r4BitMsk !!!
RM[lastShC, IP[RSCR]];                        * RENAME RSCR AS lastShC

RFWFtest:
        Q_R0;                                 * Q WILL HOLD THE INDEX VARIABLE
        t_377C;
        CNT_t;                                * LOOP LIMIT

RFTESTL:
        t_Q;
        RF_t;
        lastShC_SHC;

* CHECK LMASK
```

```
        T_ (r4BitMsk)AND (Q);               * COMPUTE LMASK (= 16-SIZE-1) FROM INDEX VAR
        rscr2_t;                            * rscr2 _ size
        t_17C;                              * 16-1
        rscr2 _ t - (rscr2);                * rscr2 _ expected Lmask = 16-size-1
        t _ (lastShC) and (17c);            * t _ Lmask from ShC
        t # (rscr2);
        branch[.+2, ALU=0];
RFLMASK:                                    * t = Lmask from ShC, rscr2 = expected Lmask
        error;                              * LMASK FIELD WRONG IN ShC

* CHECK RMASK
        t_(lastShC) and (360c);             * t = isolated Rmask field of ShC
        skpif[ALU=0];
RFRMASK:
        error;                              * RMASK FIELD NOT 0

* CHECK SHIFT COUNT = 16+pos+size+1 (Actually this computation isn't quite right.
* let count = 16+pos+size+1. realCount _ (count and 17b).
* IF (realCount and 17b) #0 then realCount _ realCount OR 20B. This funny computation
* accommodates hardware limitations associated w/ carry across boards.
        rscr2_ (Q);
        t_r4BitMsk;
        rscr2 _ rsh[rscr2,4];
        rscr2 _ t AND (rscr2);              * rscr2 = POS
        t _ 21c;                            * 16 + 1
        t _ t + (rscr2);                    * 16 + 1 + pos
        rscr2 _ q;
        rscr2 _ (rscr2) and (17c);          * isolate size
        rscr2 _ t + (rscr2);                * rscr2 _ 16 + 1 + pos + size
        rscr2 _ (rscr2) and (17c);          * isolate to 17 bits
        skpif[alu=0];                       * see if bit 0 of count is one
        rscr2 _ (rscr2) or (20c);           * set bit0 of count if count[1:4]#0
        rscr2 _ (rscr2) and (17c);          * isolate result to 5 bits

        t_ rsh[LastShC, 10];
RFSHIFTC:
        t#(rscr2);                          * t=value from LastShC, rscr2 = computed value
        skpif[ALU=0];
        error;                              * BAD SHIFT COUNT
```

```
* June 28, 1978  5:06 PM
* NOW TEST WF

WFTEST:
        t_Q;
        WF_t;
        lastShC_SHC;

* CHECK LMASK: COMPUTE pos
        rscr2 _ q;
        noop;
        t _ rsh[rscr2,4];
        rscr2 _ t and (r4BitMsk);               * isolate pos bits in rscr2

        t_lastShC;
        t _ t AND (r4BitMsk);                   * T_ LMASK
WFLMASK:
        t#(rscr2);                              * t=LastShC's Lmask, rscr2 = computed value
        branch[.+2, ALU=0];
        error;                                  * SHC'S LMASK # pos

* CHECK THAT RMASK = 16 - pos - size -1
        rscr2 _ q;
        t _ (r4BitMsk) and (q);                 * isolate size in t
        rscr2 _ rsh[rscr2,4];                   * rscr2 _ pos
        rscr2 _ (rscr2) + t;                    * rscr2 _ pos + size
        t _ 17c;                                * t _ 16 -1
        rscr2 _ t - (rscr2);                    * rscr2 _ 16 - pos - size - 1
        rscr2 _ (rscr2) and (17c);              * isolate to 17 bits
        t _ rsh[lastShC,4];                     * t = ShC's shift count
        t _ t and (r4BitMsk);

WFRMASK:
        t#(rscr2);                              * t = ShC's shift count
        skpif[ALU=0];                           * rscr2 = 16-pos-size-1
        error;                                  * RMASK NE (16-POS-SIZE-1)

* CHECK SHIFT COUNT=16-pos-size-1

        t _ rsh[lastShC,10];                    * put ShC's shift count into t
        t _ t and (37c);
        t#(rscr2);                              * t = ShC's shift count
        skpif[ALU=0];                           * rscr2 = 16-pos-size-1, as computed above
WFSHIFTC:                                       * for the Rmask check
        error;                                  * SHC'S SHIFTCOUNT # POS


        rscr2_(R1) + (Q);
        loopUntil[CNT=0&-1,RFTESTL],Q_(rscr2);
RFXITL:

        R01 _ NOT(R10);                         * RESET R01 !!!!!!
```

```
* October 20, 1978  10:23 AM
%
	TEST ALU SHIFT OPERATIONS

%
* TEST RESULT _ ALU RSH 1 (RESULT[0] _ 0)
aluRSH:
	r01 _ not(r10);			* RESET r01 !!!! INCASE WE SKIPPED RFXITL
	t_(PD_(r1))rsh 1;
	t_t#(r0);
	skpif[ALU=0];
	error;				* 1 RSH[1] SHOULD BE 0

	t_(PD_(rm1))rsh 1;
	rscr_77777C;
	t_t#(rscr);
	skpif[ALU=0];
	error;				* -1 RSH[1] SHOULD BE 77777B

	t_(PD_r10)rsh 1;
	t_t#(r01);
	skpif[ALU=0];
	error;				* (ALTERNATING 10) rsh 1 SHOULD BE (ALT. 01)

* TEST RESULT _ ALU RCY[1] (RESULT[0]_ALU[15])
aluRCY:
	t_(PD_rm1)rcy 1;
	t_t#(rm1);
	skpif[ALU=0];
	error;				* -1 RCY[1] SHOULD BE -1

	t_(PD_r0)rcy 1;
	t_t#(r0);
	skpif[ALU=0];
	error;				* 0 RCY[1] SHOULD BE 0

	t_(PD_r10)rcy 1;
	t_t#(r01);
	skpif[ALU=0];
	error;				* (ALTERNATING 10) RCY[1] SHOULD BE (ALT 01)

	t_(PD_r01)rcy 1;
	t_t#(r10);
	skpif[ALU=0];
	error;				* (ALT 01) RCY[1] SHOULD BE (ALT 10)

* REST RESULT _ ALU Arsh 1 (RESULT[0] _ ALU[0]) (SIGN PRESERVING)
aluARSH:
	t_(PD_rm1)Arsh 1;
	t_t#(rm1);
	skpif[ALU=0];
	error;				* -1 ARSH SHOULD BE -1

	t_(PD_r0)Arsh 1;
	t_t#(r0);
	skpif[ALU=0];
	error;				* 0 ARSH SHOULD BE 0

	t_(PD_rhigh1)Arsh 1;
	rscr_140000C;
	t_t#(rscr);
	skpif[ALU=0];
	error;						* 100000 ARSH SHOULD BE 140000

	t_rhigh1;
	rscr_t+(r01);
	t_(PD_r10)Arsh 1;
	t_t#(rscr);
	skpif[ALU=0];
	error;						* (ALT. 10) ARSH SHOULD BE(ALT. 01+100000)

* TEST RESULT _ ALU lsh 1
aluLSH:
	t_(PD_rhigh1)lsh 1;
	t_t#(r0);
	skpif[ALU=0];
	error;						* 100000 LSH SHOULD BE 0

aluLSHb:
	t_(PD_r1)lsh 1;
	rscr_(r1)+(r1);
	t_t#(rscr);
	skpif[ALU=0];
	error;						* 1 LSH SHOULD BE 2

aluLSHc:
	t_(PD_r01)lsh 1;
	t_t#(r10);
	skpif[ALU=0];
	error;						* (ALT. 01) LSH SHOULD BE (ALT. 10)

aluLSHd:
	t_(PD_rm1)lsh 1;
	rscr_CM2;
	t_t#(rscr);
	skpif[ALU=0];
	error;						* -1 LSH SHOULD BE -2

* TEST RESULT _ ALU LCY1
aluLCY:
	t_(PD_rm1)lcy 1;
	t_t#(rm1);
	skpif[ALU=0];
	error;						* -1 LCY SHOULD BE -1

aluLCYb:
	t_(PD_r10)lcy 1;
	t_t#(r01);
	skpif[ALU=0];
	error;						* (ALT. 10) LCY SHOULD BE (ALT. 01)

aluLCYc:
	t_(PD_r01)lcy 1;
	t_t#(r10);
	skpif[ALU=0];
	error;						* (ALT. 01) LCY SHOULD BE (ALT. 10)

aluLCYd:
	t_(PD_r0)lcy 1;
	t_t#(r0);
	skpif[ALU=0];
	error;					* 0 LCY SHOULD BE 0

aluLCYe:
	t_(PD_r1)lcy 1;
	rscr_(r1)+(r1);
	t_t#(rscr);
	skpif[ALU=0];
	error;					* 1 LCY SHOULD BE 2
```

```
* October 20, 1978  10:24 AM
%
        EXHAUSTIVE TEST OF ALU SHIFT FUNCTIONS

FOR Q IN[0..177777B] DO
        rscr2_Q rsh 1;
        t_predictedRSH[I];
        IF t_(T XOR rscr2) THEN ERROR;

        rscr2_Q )rcy 1;
        t_predictedRCY[I];
        IF t_(T XOR rscr2) THEN ERROR;

        rscr2_Q Arsh 1;
        t_predictedARSH[I];
        IF t_(T XOR rscr2) THEN ERROR;

        rscr2_Q lsh 1;
        t_predictedLSH[I];
        IF t_(T XOR rscr2) THEN ERROR;

        rscr2_Q rsh 1;
        t_predictedRSH[I];
        IF t_(T XOR rscr2) THEN ERROR;

        rscr2_Q lcy 1;
        t_predictedLCY[I];
        IF t_(T XOR rscr2) THEN ERROR;

        ENDLOOP;
%
aluSHTEST:
        Q_r0;                                   * USE Q AS LOOP VARIABLE

aluSHL:                                         * TOP OF LOOP

* RSH TEST
        rscr_Q;
        t_(PD_rscr)rsh 1;
        rscr_rsh[rscr,1];
        t_t#(rscr);
        branch[.+2,ALU=0];
RSHER:
        error;                                  * PREDICTED RESULT DIFFERENT FROM REAL ONE

* RCY TEST
        rscr_Q;                                 * Q IS LOOP VARIABLE
        t_(PD_rscr)rcy 1;
        rscr2_t;                                * REAL RESULT IN rscr2
        t_rsh[rscr,1];
        skpif[R EVEN], (PD_rscr);               * ADD HIGH BIT IF NECESSARY
        t_t+(rhigh1);                           * PREDICTED RESULT IN T
        t_t#(rscr2);
        branch[.+2,ALU=0];
RCYER:
        error;                                  * T _ (PREDICTED T) XOR rscr2

* ARSH TEST
        rscr_Q;                                 * Q IS LOOP VARIABLE
```

```
	t_(PD_rscr)Arsh 1;
	rscr2 _ T;				* rscr2 = ACTUAL RESULT
	t_rsh[rscr,1];
	PD_Q;					* ADD SIGN BIT IF REQUIRED
	skpUnless[ALU<0];			* ADD SIGN BIT IF REQUIRED
	t_t+(rhigh1);

	t_t#(rscr2);
	skpif[ALU=0];
ARSHER:
	error;					* t_(PREDICTED T) XOR rscr2

* LSH TEST
	rscr_Q;					* Q IS LOOP VARIABLE
	rscr2_(PD_Q)lsh 1;			* rscr2 = ACTUAL RESULT
	t_lsh[rscr,1];
	t_t#(rscr2);
	skpif[ALU=0];
LSHER:
	error;					* t_ (PREDICTED T) XOR rscr2

* LCY TEST
	t_rscr _ Q;				* Q IS LOOP VARIABLE
	rscr2 _ (PD_t)lcy 1;			* rscr2 = ACTUAL RESULT
	t_lsh[rscr,1];				* t_ PREDICTED RESULT
	rscr_(rscr);
	skpUnless[ALU<0];
	t_t+(r1);				* t_ T+ 1 FOR CYCLED BIT 0
	t_t#(rscr2);
	skpif[ALU=0];
LCYER:
	error;					* T _ (PREDICTED T) XOR rscr2

	t_(r1)+(Q);
	dblBranch[.+1,aluSHL,ALU=0],Q_t;
goto[afterkernel3];
```

```
%
September 22, 1986  6:04 PM
	Fix bug in label for stk&+4Err0 to be stk&plus4Err0
July 14, 1979  4:53 PM
	Fix bug in computation of correct value for stkp after performing stack+1_.
May 8, 1979  11:53 AM
	Add bypass checking to stack test.
January 25, 1979  1:12 PM
	Add call to checkTaskNum at beginKernel4 to skip the stack tests when we're not executing in task 0.
%
top level;

%
	CONTENTS

TEST	DESCRIPTION

stkTest			test all stack operations
carry20Test		tests CARRY20 function
xorCarryTest		test XORCARRY function (CIN to bit0, provided by ALUFM)
useSavedCarry		test function (use aluCarry from preeceding instr as CIN
multiplyTest		test multiply step fcn (affects Q, result. its a slowbranch, too)
divide			test divide step fcn (affects Q, result)
cdivide			test divide step fcn (affects Q, result)
slowBR			tests 8-way slow dispatch
%
beginKernel4:
	call[checkTaskNum], t _ r0;
	skpif[ALU=0];
	branch[stkXitTopL];				* don't try task 0 tests.
	Hold&TaskSim _ R0;				*Turn off hold and task simulators to avoid errors in task 12.
```

```
* May 8, 1979  11:53 AM
%
	TEST STKP PUSH AND POP OPERATIONS

-- I AND STKP SHOULD BE INCREMENTING TOGETHER.
-- notation: stack&+1[stkp] _ val : place val into stack[stkp], then increment stkp by 1
-- stack+1[stkp] _ val : increment stkp by one, then place val into stack[stkp]
-- The stragegy for this test is to perform all the various stack manipulations (+1, +2, +3,
-- -1, -2, -3, -4, &+1, &+2, &+3, &-1, &-2, &-3, &-4) for every value of stkp that won't
-- cause a hardware error (underflow).  The test knows what to expect in RM by setting
-- each rm location to its address (stack[i] _ i).

%
```

```
* July 14, 1979  4:53 PM
%
	stkTest				Test the various stack operations
%
mc[stkPMaxXC, 77];
mc[pointers.stkOvf, b8];
mc[pointers.stkUnd, b9];
mc[pointers.stkErr,  b8, b9];
stkTest:				* initialize the top2bits loop
	call[iTopStkBits];
stkTopL:				* top of "top 2 bits of stkp" loop
	call[nextTopStkBits];
	skpif[ALU#0];
	branch[stkXitTopL];
	noop;

* This code writes the current stack with the address (stack[stkP] _ stkP).
* It also checks that stkp_, _stack work properly.
	call[iStkPAddr];		* initialize stack index [1..maxStkXC]
stkIL:
	call[nextStkPAddr];		* top of stk init loop. here we check stkp_ and _stack.
	skpif[ALU#0];
	branch[stkiXit];
	stkp _ t;			* load stkp
	call[chkStkErr];
	skpif[ALU=0];
stkiErr0:				* got stack underflow or overflow
	error;

	call[getRealStkAddr], rscr _ t;
	t # (rscr);			* compare real stkp with value we loaded
	skpif[ALU=0];
stkiErr1:				* t = tskp, rscr = value we loaded
	error;

* This is a limited test of the bits in the stack memory: write zero, -1, alternating 10, 01
	t _ stack _ t-t;
	t _ t #(Q_stack) ;
	skpif[ALU=0];
stkiErr2:				* wrote zero, got back something else
	error;				* Q = value from stack

	t _ rm1;
	stack _ t;
	t _ t #(Q_stack) ;
	skpif[ALU=0];
stkiErr3:				* wrote -1 got back something else.
	error;				* t = bad bits Q = value from stack

	t _ r01;
	stack _ t;
	t _ t #(Q_stack) ;
	skpif[ALU=0];
stkiErr4:				* wrote r01 got back something else.
	error;				* t = bad bits. Q = value from stack

	t _ r10;
	stack _ t;
	t _ t #(Q_stack) ;
	skpif[ALU=0];
stkiErr5:				* wrote r10 got back something else.
	error;				* t = bad bits. Q = value from stack

	t _ rscr;			* t _ current index
	stack_t;			* stack[i] _i, then check it
	t # (Q_stack);
	skpif[ALU=0];
stkiErr6:				* wrote stkp from rscr. Q = value from stack
	error;				* read it into t. they aren't the same
	branch[stkiL];
stkiXit:
	noop;
```

```
* July 14, 1979  4:53 PM
* We have successfully written the stack using non incrementing and non decrementing
* operations. Now we test stack&+1_, stack&-1_, stack+1_, stack-1_

    call[iStkPAddr];                    * init the main loop for the main test
stkTestL:                               * top of main loop
    call[nextStkPAddr];                 * get next stack index or exit loop
    skpif[alu#0];
    branch[stkTestXitL];
    stkp _ t;                           * stackP _ i

    call[chkStkErr];
    skpif[ALU=0];
stkpErr10:                              * got stack underflow or overflow
    error;

    rscr _ t and (77c);                 * isolate the index (exclude top 2 bits)
    (rscr)-(stkPMaxXC);                 * skip this test if it would cause overflow
    branch[afterStkTest1, ALU=0];

    t # (Q_stack);                      * see if stack[stkp] = stkp
    skpif[ALU=0];                       * if not, an earlier execution of this loop clobbered
stkpErr1:                               * the stack entry at location in t, or this is first time
    error;                              * thru, and the initialization didn't work properly.
*   Q=value from stack

* stack&+1 stack&+1 stack&+1 stack&+1 stack&+1 stack&+1 stack&+1 stack&+1

    stack&+1 _ cm1;                     * stack[stackP] _ -1, then stackP _ stackP+1
    call[chkStkErr];
    skpif[ALU=0];
stk&Plus1Err0:                          * got stack underflow or overflow
    error;

    call[getRealStkAddr], rscr_t+1;     * compare stackPAddr from Pointers w/ expected val
    t #(rscr);
    skpif[ALU=0];
stkP1AddrErr:                           * auto increment of StackP failed. rscr = expected value,
    error;                              * t = value from Pointers

    t _ t # (Q_stack);
    skpif[ALU=0];
stkP1ValErr:                            * value at stackp is bad. Q = value from stack
    error;                              * t = expected val, rscr = stack's val from Pointers
    t _ rscr;                           * restore t

* stack&-1 stack&-1 stack&-1 stack&-1 stack&-1 stack&-1 stack&-1 stack&-1 stack&-1

    stack&-1 _ t;                       * stack["i+1"] _ i+1, stackp _ i.

    call[chkStkErr];
    skpif[ALU=0];
stkpErr12:                              * got stack underflow or overflow
    error;

    call[getRealStkAddr], rscr _ t-1;
    t # (rscr);                         * compare expected stkP (rscr) with
    skpif[ALU=0];                       * actual stkp (t)
stkM1AddrErr:                           * auto decrement failed
```

```
        error;

        t _ cm1;
        t _ t # (Q_stack);
        skpif[ALU=0];                   * see if original stack&+1 _ cm1 worked
stkP1ValErr2:                           * stack&+1 seems to have clobbered the
        error;                          * (i+1)th value. t = bad bits. Q = value from stack

        t _ rscr;                       * restore t
        (stack)_ t;                     * reset stk[stkp] to contain stkp

        call[chkStkErr];
        skpif[ALU=0];
stkpErr13:                              * got stack underflow or overflow
        error;

        rscr _ t _ t+1;
        stkp _ t;                       * check the data modified during "stack&-1" instruction
        t _ t # (Q_stack);              * compare tos with expected valu
        skpif[ALU=0];
stkM1ValErr:                            *. Q = value from stack
        error;                          * t = bad bits, rscr = expected value

        t _ rscr _ (rscr)-1;            * t, rscr _ "i"
        stkp _ t;                       * stkp is at i+1 now. Fix it.

        call[chkStkErr];
        skpif[ALU=0];
stkpErr14:                              * got stack underflow or overflow
        error;

        Q _ stack;                      * save stack value
        stack _ t-t;
        PD_(stack);
        skpif[ALU=0];
stkByPassErr0:                          * didn't notice that we just zeroed the stack
        error;
        t _ cm1;
        stack _ cm1;
        PD _ (stack) # t;
        skpif[ALU=0];
stkByPassErr1:                          * didn't notice that we just put all ones
        error;                          * in the stack.

        stack _ Q;                      * restore stack

* stack+1 stack+1 stack+1 stack+1 stack+1 stack+1 stack+1 stack+1 stack+1 stack+1

        rscr _ (rscr)+1;                * compute expected stkp value
        t _ cm1;
        stack+1 _ t;                    * stkp _ i+1, stack[stkp] _ -1

        call[chkStkErr];
        skpif[ALU=0];
stkpErr15:                              * got stack underflow or overflow
        error;

        call[getRealStkAddr];
        (rscr) # t;
```

```
	skpif[ALU=0];
stkP1AddrErr2:				* expected Rscr, got stackp in t, they’re different
	error;

	t _ cm1;
	t _ t # (Q_stack);			* check that we loaded -1 into incremented stack location
	skpif[ALU=0];				* Q = value from stack
stkP1ValErr3:				* t = bad bits
	error;
	t _ rscr;				* restore t
```

*** stack-1 stack-1 stack-1 stack-1 stack-1 stack-1 stack-1 stack-1 stack-1 stack-1**

```
	stack _ t;				* reset stack which was clobbered by "stack+1_cm1"

	call[chkStkErr];
	skpif[ALU=0];
stkpErr16:				* got stack underflow or overflow
	error;

	rscr _ t-1;				* compute expected value of rscr
	t _ cm1;
	stack-1 _ t;				* (stack-1) _ "-1"

	call[chkStkErr];
	skpif[ALU=0];
stkpErr17:				* got stack underflow or overflow
	error;

	call[getRealStkAddr];
	(rscr) # t;				* see if real stkp (t) matches expected stkp (rscr)
	skpif[ALU=0];
stkM1AddrErr2:
	error;

	t _ cm1;
	t _ t # (Q_stack);			* compare tos with -1
	skpif[ALU=0];
stkM1ValErr2:				* Q = value from stack
	error;				* t = bad bits, expected -1

	t _ rscr;				* restore t
	(stack) _ t;				* restore addr as value in stack: stack[stkp]_stkp

	call[chkStkErr];
	skpif[ALU=0];
stkpErr18:				* got stack underflow or overflow
	error;
	noop;				* for placement

afterStkTest1:
```

```
* November 30, 1978  6:07 PM
%remember, don't execute if i=1, if i+2=77%
	call[getStkPAddr];
	rscr _ t and (77c);			* isolate the index (exclude top 2 bits)
	rscr _ (rscr)+1;
	(rscr)-(stkPMaxXC);			* skip this test if it would cause overflow
	branch[afterStkTest2, ALU>=0];

	t # (Q_stack);				* see if stack[stkp] = stkp
	skpif[ALU=0];				* if not, an earlier execution of this loop clobbered
stkpErr21:					* the stack entry at location in t, or this is first time
	error;					* thru, and the initialization didn't work properly.
*	Q=value from stack

* stack&+2 stack&+2 stack&+2 stack&+2 stack&+2 stack&+2 stack&+2 stack&+2

	stack&+2 _ cm1;				* stack[stackP] _ -1, then stackP _ stackP+2
	call[chkStkErr];
	skpif[ALU=0];
stk&Plus2Err0:					* got stack underflow or overflow
	error;

	call[getRealStkAddr], rscr_t+(2c);	* compare stackPAddr from Pointers w/ expected val
	t #(rscr);
	skpif[ALU=0];
stkP2AddrErr:					* auto increment of StackP failed. rscr = expected value,
	error;					* t = value from Pointers

	t _ t # (Q_stack);
	skpif[ALU=0];
stkP2ValErr:					* value at stackp is bad. Q = value from stack
	error;					* t = expected val, rscr = stack's val from Pointers
	t _ rscr;				* restore t

* stack&-2 stack&-2 stack&-2 stack&-2 stack&-2 stack&-2 stack&-2 stack&-2 stack&-2

	stack&-2 _ t;				* stack["i+2"] _ i+2, stackp _ i.

	call[chkStkErr];
	skpif[ALU=0];
stkpErr22:					* got stack underflow or overflow
	error;

	call[getRealStkAddr], rscr _ t-(2c);
	t # (rscr);				* compare expected stkP (rscr) with
	skpif[ALU=0];				* actual stkp (t)
stkM2AddrErr:					* auto decrement failed
	error;

	t _ cm1;
	t _ t # (Q_stack);
	skpif[ALU=0];				* see if original stack&+2 _ cm1 worked
stkP2ValErr2:					* stack&+2 seems to have clobbered the
	error;					* (i+2)th value. t = bad bits. Q = value from stack

	t _ rscr;				* restore t
	(stack)_ t;				* reset stk[stkp] to contain stkp

	call[chkStkErr];
	skpif[ALU=0];
stkpErr23:				* got stack underflow or overflow
	error;

	rscr _ t _ t+(2C);
	stkp _ t;			* check the data modified during "stack&-2" instruction
	t _ t # (Q_stack);		* compare tos with expected valu
	skpif[ALU=0];
stkM2ValErr:				*. Q = value from stack
	error;				* t = bad bits, rscr = expected value

* stack+2 stack+2 stack+2 stack+2 stack+2 stack+2 stack+2 stack+2 stack+2 stack+2

	t _ rscr _ (rscr)-(2c);		* t, rscr _ "i"
	stkp _ t;			* stkp is at i+2c now. Fix it.

	call[chkStkErr];
	skpif[ALU=0];
stkpErr24:				* got stack underflow or overflow
	error;

	rscr _ t+(2c);			* compute expected stkp value
	t _ cm1;
	stack+2 _ t;			* stkp _ i+2, stack[stkp] _ -1

	call[chkStkErr];
	skpif[ALU=0];
stkpErr25:				* got stack underflow or overflow
	error;

	call[getRealStkAddr];
	(rscr) # t;
	skpif[ALU=0];
stkP2AddrErr2:				* expected stackP t, got stackp in Rscr, they're different
	error;

	t _ cm1;
	t _ t # (Q_stack);		* check that we loaded -1 into incremented stack location
	skpif[ALU=0];			* Q = value from stack
stkP2ValErr3:				* t = bad bits
	error;
	t _ rscr;			* restore t

* stack-2 stack-2 stack-2 stack-2 stack-2 stack-2 stack-2 stack-2 stack-2 stack-2

	stack _ t;			* reset stack which was clobbered by "stack+2_cm1"

	call[chkStkErr];
	skpif[ALU=0];
stkpErr26:				* got stack underflow or overflow
	error;

	rscr _ t-(2c);			* compute expected value of rscr
	t _ cm1;
	stack-2 _ t;			* (stack-2) _ "-1"

	call[chkStkErr];
	skpif[ALU=0];
stkpErr27:				* got stack underflow or overflow
```

```
	error;

	call[getRealStkAddr];
	(rscr) # t;				* see if real stkp (t) matches expected stkp (rscr)
	skpif[ALU=0];
stkM2AddrErr2:
	error;

	t _ cm1;
	t _ t # (Q_stack);			* compare tos with -1
	skpif[ALU=0];
stkM2ValErr2:				* Q = value from stack
	error;					* t = bad bits, expected -1

	t _ rscr;				* restore t
	(stack) _ t;				* restore addr as value in stack: stack[stkp]_stkp

	call[chkStkErr];
	skpif[ALU=0];
stkpErr228:				* got stack underflow or overflow
	error;
	noop;					* for placement

afterStkTest2:
```

```
* December 1, 1978  3:19 PM
%remember, don't execute if i=1, if i+3>=77%
	noop;				* placement for afterStkTest2 branch
	call[getStkPAddr];
	rscr _ t and (77c);		* isolate the index (exclude top 2 bits)
	rscr _ (rscr)+(2c);
	(rscr)-(stkPMaxXC);		* skip this test if it would cause overflow
	branch[afterStkTest3, ALU>=0];

	t # (Q_stack);			* see if stack[stkp] = stkp
	skpif[ALU=0];			* if not, an earlier execution of this loop clobbered
stkpErr31:				* the stack entry at location in t, or this is first time
	error;				* thru, and the initialization didn't work properly.
*	Q=value from stack

* stack&+3 stack&+3 stack&+3 stack&+3 stack&+3 stack&+3 stack&+3 stack&+3

	stack&+3 _ cm1;			* stack[stackP] _ -1, then stackP _ stackP+3
	call[chkStkErr];
	skpif[ALU=0];
stk&Plus3Err0:				* got stack underflow or overflow
	error;

	call[getRealStkAddr], rscr_t+(3c);	* compare stackPAddr from Pointers w/ expected val
	t #(rscr);
	skpif[ALU=0];
stkP3AddrErr:				* auto increment of StackP failed. rscr = expected value,
	error;				* t = value from Pointers

	t _ t # (Q_stack);
	skpif[ALU=0];
stkP3ValErr:				* value at stackp is bad. Q = value from stack
	error;				* t = expected val, rscr = stack's val from Pointers
	t _ rscr;			* restore t

* stack&-3 stack&-3 stack&-3 stack&-3 stack&-3 stack&-3 stack&-3 stack&-3 stack&-3

	stack&-3 _ t;			* stack["i+3"] _ i+3, stackp _ i.

	call[chkStkErr];
	skpif[ALU=0];
stkpErr32:				* got stack underflow or overflow
	error;

	call[getRealStkAddr], rscr _ t-(3c);
	t # (rscr);			* compare expected stkP (rscr) with
	skpif[ALU=0];			* actual stkp (t)
stkM3AddrErr:				* auto decrement failed
	error;

	t _ cm1;
	t _ t # (Q_stack);
	skpif[ALU=0];			* see if original stack&+3 _ cm1 worked
stkP3ValErr2:				* stack&+3 seems to have clobbered the
	error;				* (i+3)th value. t = bad bits. Q = value from stack

	t _ rscr;			* restore t
	(stack)_ t;			* reset stk[stkp] to contain stkp
```

```
        call[chkStkErr];
        skpif[ALU=0];
stkpErr33:                          * got stack underflow or overflow
        error;

        rscr _ t _ t+(3C);
        stkp _ t;                   * check the data modified during "stack&-3" instruction
        t _ t # (Q_stack);          * compare tos with expected valu
        skpif[ALU=0];
stkM3ValErr:                        *. Q = value from stack
        error;                      * t = bad bits, rscr = expected value

* stack+3 stack+3 stack+3 stack+3 stack+3 stack+3 stack+3 stack+3 stack+3 stack+3

        t _ rscr _ (rscr)-(3c);     * t, rscr _ "i"
        stkp _ t;                   * stkp is at i+3c now. Fix it.

        call[chkStkErr];
        skpif[ALU=0];
stkpErr34:                          * got stack underflow or overflow
        error;

        rscr _ t+(3c);              * compute expected stkp value
        t _ cm1;
        stack+3 _ t;                * stkp _ i+3, stack[stkp] _ -1

        call[chkStkErr];
        skpif[ALU=0];
stkpErr35:                          * got stack underflow or overflow
        error;

        call[getRealStkAddr];
        (rscr) # t;
        skpif[ALU=0];
stkP3AddrErr2:                      * expected stackP t, got stackp in Rscr, they're different
        error;

        t _ cm1;
        t _ t # (Q_stack);          * check that we loaded -1 into incremented stack location
        skpif[ALU=0];               * Q = value from stack
stkP3ValErr3:                       * t = bad bits
        error;
        t _ rscr;                   * restore t

* stack-3 stack-3 stack-3 stack-3 stack-3 stack-3 stack-3 stack-3 stack-3 stack-3

        stack _ t;                  * reset stack which was clobbered by "stack+3_cm1"

        call[chkStkErr];
        skpif[ALU=0];
stkpErr36:                          * got stack underflow or overflow
        error;

        rscr _ t-(3c);              * compute expected value of rscr
        t _ cm1;
        stack-3 _ t;                * (stack-3) _ "-1"

        call[chkStkErr];
        skpif[ALU=0];
```

```
stkpErr37:                              * got stack underflow or overflow
        error;

        call[getRealStkAddr];
        (rscr) # t;                     * see if real stkp (t) matches expected stkp (rscr)
        skpif[ALU=0];
stkM3AddrErr2:
        error;

        t _ cm1;
        t _ t # (Q_stack);              * compare tos with -1
        skpif[ALU=0];
stkM3ValErr2:                           * Q = value from stack
        error;                          * t = bad bits, expected -1

        t _ rscr;                       * restore t
        (stack) _ t;                    * restore addr as value in stack: stack[stkp]_stkp

        call[chkStkErr];
        skpif[ALU=0];
stkpErr38:                              * got stack underflow or overflow
        error;
        noop;                           * for placement

afterStkTest3:
```

```
* December 1, 1978  4:57 PM
%remember, don't execute if i=1, if i+4>=77%
    noop;                               * placement for the afterStkTest3 check
    call[getStkPAddr];
    rscr _ t and (77c);                 * isolate the index (exclude top 2 bits)
    rscr _ (rscr)+(3c);
    (rscr)-(stkPMaxXC);                 * skip this test if it would cause overflow
    branch[afterStkTest4, ALU>=0];

    t # (Q_stack);                      * see if stack[stkp] = stkp
    skpif[ALU=0];                       * if not, an earlier execution of this loop clobbered
stkpErr41:                              * the stack entry at location in t, or this is first time
    error;                              * thru, and the initialization didn't work properly.
*   Q=value from stack

* Simulate stack&+4 -- hardware can perform stack&+3 as maximum increment

    stack&+3 _ cm1;                     * stack[stackP] _ -1, then stackP _ stackP+4
    stkp+1;                             * simulate +4
    call[chkStkErr];
    skpif[ALU=0];
stk&Plus4Err0:                          * got stack underflow or overflow
    error;

    call[getRealStkAddr], rscr_t+(4c);  * compare stackPAddr from Pointers w/ expected val
    t #(rscr);
    skpif[ALU=0];
stkP4AddrErr:                           * auto increment of StackP failed. rscr = expected value,
    error;                              * t = value from Pointers

    t _ t # (Q_stack);
    skpif[ALU=0];
stkP4ValErr:                            * value at stackp is bad. Q = value from stack
    error;                              * t = expected val, rscr = stack's val from Pointers
    t _ rscr;                           * restore t

* stack&-4 stack&-4 stack&-4 stack&-4 stack&-4 stack&-4 stack&-4 stack&-4 stack&-4

    stack&-4 _ t;                       * stack["i+4"] _ i+4, stackp _ i.

    call[chkStkErr];
    skpif[ALU=0];
stkpErr42:                              * got stack underflow or overflow
    error;

    call[getRealStkAddr], rscr _ t-(4c);
    t # (rscr);                         * compare expected stkP (rscr) with
    skpif[ALU=0];                       * actual stkp (t)
stkM4AddrErr:                           * auto decrement failed
    error;

    t _ cm1;
    t _ t # (Q_stack);
    skpif[ALU=0];                       * see if original stack&+4 _ cm1 worked
stkP4ValErr2:                           * stack&+4 seems to have clobbered the
    error;                              * (i+4)th value. t = bad bits. Q = value from stack

    t _ rscr;                           * restore t
    (stack)_ t;                         * reset stk[stkp] to contain stkp
```

```
        call[chkStkErr];
        skpif[ALU=0];
stkpErr43:                              * got stack underflow or overflow
        error;

        rscr _ t _ t+(4C);
        stkp _ t;                       * check the data modified during "stack&-4" instruction
        t _ t # (Q_stack);              * compare tos with expected valu
        skpif[ALU=0];
stkM4ValErr:                            *. Q = value from stack
        error;                          * t = bad bits, rscr = expected value

* stack+4 stack+4 stack+4 stack+4 stack+4 stack+4 stack+4 stack+4 stack+4 stack+4

        t _ rscr _ (rscr)-(4c);         * t, rscr _ "i"
        stkp _ t;                       * stkp is at i+4c now. Fix it.

        call[chkStkErr];
        skpif[ALU=0];
stkpErr44:                              * got stack underflow or overflow
        error;

        rscr _ t+(4c);                  * compute expected stkp value
        t _ cm1;
        stkp+1;                         * simulate stack+4
        stack+3 _ t;                    * stkp _ i+4, stack[stkp] _ -1

        call[chkStkErr];
        skpif[ALU=0];
stkpErr45:                              * got stack underflow or overflow
        error;

        call[getRealStkAddr];
        (rscr) # t;
        skpif[ALU=0];
stkP4AddrErr2:                          * expected stackP t, got stackp in Rscr, they're different
        error;

        t _ cm1;
        t _ t # (Q_stack);              * check that we loaded -1 into incremented stack location
        skpif[ALU=0];                   * Q = value from stack
stkP4ValErr3:                           * t = bad bits
        error;
        t _ rscr;                       * restore t

* stack-4 stack-4 stack-4 stack-4 stack-4 stack-4 stack-4 stack-4 stack-4 stack-4

        stack _ t;                      * reset stack which was clobbered by "stack+4_cm1"

        call[chkStkErr];
        skpif[ALU=0];
stkpErr46:                              * got stack underflow or overflow
        error;

        rscr _ t-(4c);                  * compute expected value of rscr
        t _ cm1;
        stack-4 _ t;                    * (stack-4) _ "-1"
        call[chkStkErr];
        skpif[ALU=0];
stkpErr47:                                  * got stack underflow or overflow
        error;

        call[getRealStkAddr];
        (rscr) # t;                         * see if real stkp (t) matches expected stkp (rscr)
        skpif[ALU=0];
stkM4AddrErr2:
        error;

        t _ cm1;
        t _ t # (Q_stack);                  * compare tos with -1
        skpif[ALU=0];
stkM4ValErr2:                               * Q = value from stack
        error;                              * t = bad bits, expected -1

        t _ rscr;                           * restore t
        (stack) _ t;                        * restore addr as value in stack: stack[stkp]_stkp

        call[chkStkErr];
        skpif[ALU=0];
stkpErr48:                                  * got stack underflow or overflow
        error;
        noop;                               * for placement

afterStkTest4:
        branch[stkTestL];
stkTestXitL:
        branch[stkTopL];
```

```
* November 27, 1978  10:31 AM.

iStkPAddr: subroutine;
      return, stackPAddr _ t-t;                * first valid index is one.

nextStkPAddr: subroutine;                      * stack indeces are 6 bits long.
*     Return (stackPaddr OR stackPtopBits) in T. It's an 8 bit address.
*     ALU#0 =>valid value.

      klink _ link;
      t _ stackPAddr _ (stackPAddr) + 1;       * increment the index
      t and (77c);                             * check for 6 bit overflow
      skpif[ALU=0], t _ t + (stackPTopBits);      * OR the top two bits into returned value
      skip, rscr _ 1c;                         * indicate valid value
      rscr _ t-t;                              * indicate invalid value
      returnAndBranch[klink, rscr];

getStkPAddr: subroutine;
      t _ stackPAddr;
      return, t _ t + (stackPTopBits);

iTopStkBits: subroutine;
      t _ (r0) - (100c);
      return, stackPTopBits _ t;               * first valid index is zero.
nextTopStkBits: subroutine;
      klink _ link;
      top level;
      t _ stackPTopBits _ (stackPTopBits)+(100c);
      t - (400c);
      skpif[ALU#0], rscr _ 1c;
      rscr _ t-t;
      returnAndBranch[klink, rscr];

getRealStkAddr: subroutine;
      t _ TIOA&Stkp;
      return, t _ t and (377c);

chkStkErr: subroutine;                         * rtn w/ ALU#0 ==> stk (underflow or overflow).
* Clobber rscr2 _ Pointers[]
      rscr2 _ Pointers[];
      return, rscr2 _ (rscr2) AND (pointers.stkErr);
      top level;
stkXitTopL:
```

```
* September 3, 1977  2:25 PM
%
        TEST CARRY20 FUNCTION

        This function causes a 1 go be or'd into the carry out bit that is used
as input to bit 11 in the alu. Given that there was not already a carry, this
function has the effect of adding 20B to the value in the alu.

%
carry20Test:
        t_rscr_17C;
        t_t+(r0),CARRY20;
        rscr2 _ 37C;
        t_t#(rscr2);
        skpif[ALU=0];
        error;                              * T NE 17B + 0 + CARRY20

        t_rscr;
        t_t+(r1),CARRY20;                   * t_17B+1+CARRY20
        rscr2_20C;
        t_t#(rscr2);
        skpif[ALU=0];
        error;                              * T NE 17B+1=20(=17B+1+CARRY20)

        t_r0;
        t_t+(r0),CARRY20;                   * t_0+0+CARRY20
        rscr2_20C;
        t_t#(rscr2);
        skpif[ALU=0];
        error;                              * T NE 20B=0+0+CARRY20

        t_r0;
        t_t-(rscr2);                        * t_-20B=(0-20B)
        t_t+(r0),CARRY20;
        skpif[ALU=0];
        error;
```

```
* September 11, 1977  1:57 PM
%
        TEST XORCARRY FUNCTION

        XORCARRY causes the carry in bit for bit 15 of the alu to be xor'd.
Normally this bit is 0. When the bit is one, alu arithmetic functions will
see a carry into bit 0. For A-B the ALUFM is programmed to provide a one
and XORCARRY will leave a result one less than expected.
%
xorCarryTest:
        t_(r0)+(r0),XORCARRY;
        t_t#(r1);
        skpif[ALU=0];
        error;                                  * 1 = 0+0+XORCARRY

        t_r1;
        t_t+(r0),XORCARRY;
        t_t#(2C);
        skpif[ALU=0];
xorCarryb:
        error;                                  * 2= 0+1+XORCARRY

        t_r1;
        t_t+(r1),XORCARRY;
        t_t#(3C);
        skpif[ALU=0];
xorCarryc:
        error;                                  * 3= 1+1+XORCARRY

        t_RM1;
        t_t+(r0),XORCARRY;
        skpif[ALU=0];
xorCarryd:
        error;                                  * 0= -1+XORCARRY

        t_(r0)AND(r0),XORCARRY;
        skpif[ALU=0];
xorCarrye:
        error;                                  * CIN SHOULD BE IGNORED ON LOGICAL OPS!

        t_(r1)AND(r1),XORCARRY;
        t_t#(r1);
        skpif[ALU=0];
xorCarryf:
        error;                                  * SHOULD BE 1. CIN IGNORED ON LOGICAL OPS

        t_(RM1)OR(RM1),XORCARRY;
        t_t#(RM1);
        skpif[ALU=0];
xorCarryg:
        error;                                  * SHOULD BE -1. CIN IGNORED ON LOGICAL OPS

        t_(r1)-(r1),XORCARRY;
        t_t#(RM1);
        skpif[ALU=0];
xorCarryh:
        error;                                  * BWL SEZ THIS SHOULD BE -1. IE.,
* R1-R1 causes 1+1777777, but xorcarry causes the op to become,
* 1+177776 because "A-B" uses carryin = 1, and xorcarry causes it to be
* zero!
* September 12, 1977  9:52 AM
%
	TEST USESAVEDCARRY

This function causes the alu carry bit from the last instruction to be used as
the carry in bit to bit 15 during the current instruction. This bit is usually
provided by the alufm and is usually zero (its the bit complemented by the
xorcarry function).
%
%                                                   commented out
savedCarry:
	T_(RHIGH1)+(RHIGH1);                   * T_0, CARRY_1
	T_T+(R0),USESAVEDCARRY;                * T_0+0+LAST CARRY
	T_T#(R1);
	SKPIF[ALU=0];
	ERROR;                                 * EXPECTED 1, USED LASTCARRY=1

	T_(RM1)+(RM1);                         * T_-2, CARRY _1
	T_T+(R1),USESAVEDCARRY;                * T_-2+1+LAST CARRY
	SKPIF[ALU=0];
savedCarryB:
	ERROR;                                 * EXPECTED 0, USED LASTCARRY=1

	T_(R0)+(R0);                           * T_0, CARRY_0
	T_(R1)+(R1),USESAVEDCARRY;             * T_1+1+LAST CARRY
	T_T#(2C);
	SKPIF[ALU=0];
savedCarryC:
	ERROR;                                 * EXPECTED 2, USED LASTCARRY=0

	T_(R0)+(R0);                           * T_0, CARRY_0
	T_(RM1)+(RM1),USESAVEDCARRY;           * T_(-1)+(-1)+LAST CARRY
	T_T#(177776C);
	SKPIF[ALU=0];
savedCarryD:
	ERROR;                                 * EXPECTED -2, USED LASTCARRY=0

	T_(RM1)+(RM1);                         * T_-2, CARRY_1
	T_(R1)+(R1),USESAVEDCARRY;             * T_1+1+LAST CARRY
	T_T#(3C);
	SKPIF[ALU=0];
savedCarryE:
	ERROR;                                 * EXPECTED 3, USED LASTCARRY=1
                                               commented out%
```

```
* September 14, 1977  12:03 PM
%
        TEST MULTIPLY STEP.

MULTIPLY works as follows:
        H3[0:15] _ CARRY,,ALU[0:14]  ==> CARRY,,ALU/2
        Q[0:15]  _ ALU[15],,Q[0:14]  ==> ALU[15],,Q/2
        Q[14] OR'D INTO TNIA[10] AS SLOW BRANCH! ==> ADDR OR'D 2 IF Q[14]=1

These tests invoke mulCheck, a subroutine that performs two services:
        1)  T_T+RSCR,MULTIPLY
        2)  RSCR2_1 IF Q[14] BRANCH IS TAKEN (ZERO OTHERWISE)

        ERRORS are numbered 1 thru 3:
                mulXerr1 ==> T value wrong (H3)
                mulXerr2 ==> Q[14] branch wrong
                mulXerr3 ==> Q value wrong
%
multiplyTest:
*               Q[14]=0, CARRY=0, ALU15=0
        t_Q_r0;
        rscr_t;
        call[mulCheck];   * t_t+rscr,MULTIPLY==>ALU_0, CARRY_0
        skpif[R EVEN],rscr2_rscr2;
mulAerr1:
        error;      * TOOK Q[14]=1 BRANCH

        t_t;
        skpif[ALU=0];
mulAerr2:
        error;      * CARRY,,(0+0)/2 SHOULD BE ZERO

        t_Q;
        skpif[ALU=0];
mulAerr3:
        error;      * ALU15,,Q[0:14] SHOULD BE ZERO

*               Q[14]=0, CARRY=1, ALU15=0
multiplyB:      * Q=0; t_-1+1,MULTIPLY
        Q_r0;
        rscr_r1;
        call[mulCheck],t_rm1;           * t_t+rscr,MULTIPLY==>ALU_0, CARRY_1
        skpif[R EVEN], rscr2_rscr2;
mulBerr1:
        error;      * TOOK Q[14]=1 BRANCH

        t_t#(rhigh1);       *-1+1 GENERATES CARRY BIT
        skpif[ALU=0];
mulBerr2:
        error;      * CARRY,,(0+0)/2 SHOULD BE 100000

        t_Q;        * -1+1 WOULD LEAVE ALU15=0
        skpif[ALU=0];
mulBerr3:
        error;      * ALU15,,Q[0:14] SHOULD BE ZERO

*               Q[14]=0, CARRY=0, ALU15=1
multiplyC:      * Q=0; t_0+1,MULTIPLY
        t_Q_r0;
        rscr_r1;
        call[mulCheck];    * t_t+rscr,MULTIPLY==>ALU_1, CARRY_0
        skpif[R EVEN], rscr2_rscr2;
mulCerr1:
        error;     * TOOK Q[14]=1 BRANCH

        t_t;       * 0+1==> CARRY_0
        skpif[ALU=0];
mulCerr2:
        error;     * CARRY,,(0+1)/2 SHOULD BE 0

        t_(rhigh1)#(Q);                * 0+1 WOULD LEAVE ALU15=1
        skpif[ALU=0];
mulCerr3:
        error;     * ALU15,,Q[0:14] SHOULD BE 100000

*                  Q[14]=0, CARRY=1, ALU15=1
multiplyD:         * Q=0; t_100001+100000,MULTIPLY
        Q_r0;
        t_rscr_rhigh1;
        call[mulCheck],t_t+(r1);       * t_t+rscr,MULTIPLY==>ALU_1, CARRY_1
        skpif[R EVEN], rscr2_rscr2;
mulDerr1:
        error;     * TOOK Q[14]=1 BRANCH

        t_t#(rhigh1);      * 1000001+100000==> CARRY_1
        skpif[ALU=0];
mulDerr2:
        error;     * CARRY,,(1000001+100000)/2 SHOULD BE 100000

        t_(rhigh1)#(Q);                * 1000001+100000 WOULD LEAVE ALU15=1
        skpif[ALU=0];
mulDerr3:
        error;     * ALU15,,Q[0:14] SHOULD BE 100000

multiplyE:
*                                          Q[14]=1, CARRY=0, ALU15=0
        t_(r1)+(r1);
        Q_t;                               * Q[14]_1
        t_r0;
        rscr_t;
        call[mulCheck];                    * t_t+rscr,MULTIPLY==>ALU_0, CARRY_0
        skpif[R ODD],rscr2_rscr2;
mulEerr1:
        error;                             * DIDN'T TAKE Q[14]=1 BRANCH

        t_t;
        skpif[ALU=0];
mulEerr2:
        error;                             * CARRY,,(0+0)/2 SHOULD BE ZERO

        t_(r1)#(Q);
        skpif[ALU=0];
mulEerr3:
        error;                             * ALU15,,Q[0:14] SHOULD BE 1

*                                          Q[14]=1, CARRY=1, ALU15=0
multiplyF:                                 * Q=1; t_-1+1,MULTIPLY
        t_(r1)+(r1);
        Q_t;                                    * Q[14]_1
        rscr_r1;
        call[mulCheck],t_rm1;                   * t_t+rscr,MULTIPLY==>ALU_0, CARRY_1
        skpif[R ODD], rscr2_rscr2;
mulFerr1:
        error;                                  * DIDN'T TAKE Q[14]=1 BRANCH

        t_t#(rhigh1);                           *-1+1 GENERATES CARRY BIT
        skpif[ALU=0];
mulFerr2:
        error;                                  * CARRY,,(0+0)/2 SHOULD BE 100000

        t_(r1)#(Q);                             * -1+1 WOULD LEAVE ALU15=0
        skpif[ALU=0];
mulFerr3:
        error;                                  * ALU15,,Q[0:14] SHOULD BE 1

*                                               Q[14]=1, CARRY=0, ALU15=1
multiplyG:                                      * Q=1; t_0+1,MULTIPLY
        t_(r1)+(r1);
        Q_t;                                    * Q[14]_1
        t_r0;
        rscr_r1;
        call[mulCheck];                         * t_t+rscr,MULTIPLY==>ALU_1, CARRY_0
        skpif[R ODD], rscr2_rscr2;
mulGerr1:
        error;                                  * DIDN'T TAKE Q[14]=1 BRANCH

        t_t;                                    * 0+1==> CARRY_0
        skpif[ALU=0];
mulGerr2:
        error;                                  * CARRY,,(0+1)/2 SHOULD BE 0

        t_(rhigh1)+1;                           * 0+1 WOULD LEAVE ALU15=1
        t_t#(Q);
        skpif[ALU=0];
mulGerr3:
        error;                                  * ALU15,,Q[0:14] SHOULD BE 100001

*                                               Q[14]=1, CARRY=1, ALU15=1
multiplyH:                                      * Q=2; t_100001+100000,MULTIPLY
        t_(r1)+(r1);
        Q_t;                                    * Q[14]_1
        t_rscr_rhigh1;
        call[mulCheck],t_t+(r1);                * t_t+rscr,MULTIPLY==>ALU_1, CARRY_1
        skpif[R ODD], rscr2_rscr2;
mulHerr1:
        error;                                  * DIDN'T TAKE Q[14]=1 BRANCH

        t_t#(rhigh1);                           * 1000001+100000==> CARRY_1
        skpif[ALU=0];
mulHerr2:
        error;                                  * CARRY,,(1000001+100000)/2 SHOULD BE 100000

        t_(rhigh1)+1;                           * 1000001+100000 WOULD LEAVE ALU15=1
        t_t#(Q);
        skpif[ALU=0];
mulHerr3:
        error;                                  * ALU15,,Q[0:14] SHOULD BE 100001
```

```
multiplyJ:
*                                       Q_r01=>Q[14]=0; CARRY=0,ALU15=0
        t_r01;
        Q_t;                            * Q[14]_0
        t_r0;
        rscr_t;
        call[mulCheck];                 * t_t+rscr,MULTIPLY==>ALU_0, CARRY_0
        skpif[R EVEN],rscr2_rscr2;
mulJerr1:
        error;                          * TOOK Q[14]=1 BRANCH

        t_t;
        skpif[ALU=0];
mulJerr2:
        error;                          * CARRY,,(0+0)/2 SHOULD BE ZERO

        t_(r01) RSH 1;
        t_t#(Q);
        skpif[ALU=0];
mulJerr3:
        error;                          * ALU15,,Q[0:14] SHOULD BE (r01) RSH 1

multiplyK:
*                                       Q_r10=>Q[14]=1; CARRY=0,ALU15=0
        t_r10;
        Q_t;                            * Q[14]_1
        t_r0;
        rscr_t;
        call[mulCheck];                 * t_t+rscr,MULTIPLY==>ALU_0, CARRY_0
        skpif[R ODD],rscr2_rscr2;
mulKerr1:
        error;                          * DIDN'T TAKE Q[14]=1 BRANCH

        t_t;
        skpif[ALU=0];
mulKerr2:
        error;                          * CARRY,,(0+0)/2 SHOULD BE ZERO

        t_(r10) RSH 1;
        t_t#(Q);
        skpif[ALU=0];
mulKerr3:
        error;                          * ALU15,,Q[0:14] SHOULD BE (r10) RSH 1


mulDone: BRANCH[afterMul];
```

```
* September 11, 1977  2:46 PM
%
        MULCHECK

This subroutine performs,
        t_t+(rscr),MULTIPLY;

It sets rscr2=0 IF Q[14] branch DID NOT HAPPEN.
It sets rscr2=1 IF Q[14] branch DID HAPPEN

T and rscr2 ARE CLOBBERED! IT ASSUMES r0=0, r1=1.
%
SUBROUTINE;
mulCheck:
        t_t+(rscr),MULTIPLY, GLOBAL, AT[700];
        GOTO[.+1];
        rscr2_r0,AT[701];
        RETURN;
        rscr2_r1,AT[703];
        RETURN;
TOP LEVEL;

afterMul:
        noop;
```

```
* September 14, 1977  12:03 PM
%
        DIVIDE TEST

        H3 _ ALU[1:15],,Q[0]  ==> H3_ 2*ALU,,Q[0]
        Q  _ Q[1:15], CARRY   ==> Q _ 2*Q,,CARRY
%
divideTest:
*                                       Q0=0, CARRY=0
        Q_r0;
        t_(r1)+(r1),DIVIDE;             *t_1+1,DIVIDE==> CARRY_0,
        t_t#(4C);
        skpif[ALU=0];
        error;                          * 2*(1+1)=4, Q0=0

        t_Q;
        skpif[ALU=0];
        error;                          * CARRY WAS ZERO, Q SHOULD BE ZERO

divB:
*                                       Q0=0, CARRY=1
        Q_r0;
        t_rhigh1;
        t_t+(r1);                       * T = 100001
        t_t+(rhigh1),DIVIDE;            * t_100001+100000,DIVIDE==>ALU=1, CARRY=1
        t_t#(2C);
        skpif[ALU=0];
        error;                          * 2*(1+1)=4, Q0=0

        t_(r1)#(Q);
        skpif[ALU=0];
        error;                          * 2*0,,CARRY SHOULD BE 1


divC:
*                                       Q0=1, CARRY=0
        Q_rhigh1;
        t_(r1)+(r1),DIVIDE;             * t_1+1,DIVIDE==>ALU=2, CARRY=0
        t_t#(5C);
        skpif[ALU=0];
        error;                          * 2*(2),,Q[0]=5

        t_(Q);
        skpif[ALU=0];
        error;                          * Q[1:15],,CARRY SHOULD BE ZERO

divD:
*                                       Q0=1, CARRY=1
        t_Q_rhigh1;                     * SET Q[0] TO ONE
        t_t+(r1);                       * T = 100001
        t_t+(rhigh1),DIVIDE;            * t_100001+100000,DIVIDE==>ALU=1, CARRY=1
        t_t#(3C);
        skpif[ALU=0];
        error;                          * 2*(1),,Q[0]=3

        t_(r1)#(Q);
        skpif[ALU=0];
        error;                          * 2*0,,CARRY SHOULD BE 1
```

```
divE:
*                                   Q_r01=>Q0=0, CARRY=1
          Q_r01;
          t_(rhigh1)+1;             * T = 100001
          t_t+(rhigh1),DIVIDE;      * t_100001+100000,DIVIDE==>ALU=1, CARRY=1
          t_t#(2C);
          skpif[ALU=0];
          error;                    * 2*(1),,Q[0]=2

          t_(r01) LSH 1;
          t_t+(r1);                 * ADD ONE FOR CARRY
          t_t#(Q);
          skpif[ALU=0];
          error;                    * 2*r01,,CARRY SHOULD BE ((r01)LSH 1)+1
```

```
* May 1, 1909  1:09 PM
%
        CDIVIDE TEST

        H3 _ ALU[1:15],,Q[0]  ==> H3_ 2*ALU,,Q[0]
        Q  _ Q[1:15], CARRY   ==> Q _ 2*Q,,CARRY' (COMPLEMENTED CARRY)
%
CdivideTest:
*                                       Q0=0, CARRY=0
        Q_r0;
        t_(r1)+(r1),CDIVIDE;            *t_1+1,DIVIDE==> CARRY_0
        t_t#(4C);
        skpif[ALU=0];
        error;                          * 2*(1+1)=4, Q0=0

        t_(r1)#(Q);
        skpif[ALU=0];
        error;                          * CARRY' WAS 1, Q SHOULD BE 1

CdivB:
*                                       Q0=0, CARRY=1
        Q_r0;
        t_rhigh1;
        t_t+(r1);                       * T = 100001
        t_t+(rhigh1),CDIVIDE;           * t_100001+100000,DIVIDE==>ALU=1, CARRY=1
        t_t#(2C);
        skpif[ALU=0];
        error;                          * 2*(1+1)=4, Q0=0

        t_(Q);
        skpif[ALU=0];
        error;                          * 2*0,,CARRY' SHOULD BE 0


CdivC:
*                                       Q0=1, CARRY=0
        Q_rhigh1;
        t_(r1)+(r1),CDIVIDE;            * t_1+1,DIVIDE==>ALU=2, CARRY=0
        t_t#(5C);
        skpif[ALU=0];
        error;                          * 2*(1),,Q[0]=3

        t_(r1)#(Q);
        skpif[ALU=0];
        error;                          * Q[1:15],,CARRY' SHOULD BE 1

CdivD:
*                                       Q0=1, CARRY=0
        T_Q_rhigh1;                     * SET Q[0] TO ONE
        T_T+(R1);                       * T = 100001
        T_T+(rhigh1),CDIVIDE;           * T_100001+100000,DIVIDE==>ALU=1, CARRY=1
        T_T#(3C);
        skpif[ALU=0];
        ERROR;                          * 2*(1),,Q[0]=3

        T_(Q);
        skpif[ALU=0];
        ERROR;                          * 2*0,,CARRY' SHOULD BE 0
```

```
CdivE:
*                                                   Q_R01=>Q0=0, CARRY=1
        Q_R01;
        T_(rhigh1)+1;                               * T = 100001
        T_T+(rhigh1),CDIVIDE;                       * T_100001+100000,CDIVIDE==>ALU=1, CARRY=1
        T_T#(2C);
        skpif[ALU=0];
        ERROR;                                      * 2*(1),,Q[0]=2

        T_(R01) LSH 1;
        T_T#(Q);
        skpif[ALU=0];
        ERROR;                                      * 2*R01,,CARRY' SHOULD BE (R01)LSH 1
```

```
* September 9, 1977  5:09 PM
%
        TEST 8 WAY SLOW B DISPATCH

        Go thru the loop 16 times to make sure that only the low 3 bits are
or'd into next pc. keep counter in Q, loop control in CNT.
%
slowBr:
        cnt_17s;
        t_q_r0;
        rscr2_7C;                               * THIS WILL BE A 3 BIT MASK

slowBrL:                                        * TOP OF LOOP
        t_rm1;
        rscr_t;
        t_q;
        BDISPATCH_t;                            * DO DISPATCH W/ BITS IN T (=Q)
        branch[BDTbl];
slowBRnoBr:                                     * should have branched and didn't
        error;

bdConverge:
        t_(rscr2)AND(q);                        * MASK DISPATCH VALUE
        t_t#(rscr);
        branch[.+2,ALU=0];
slowBRbadBr:
        error;                                  * DIDN'T GO WHERE WE EXPECTED

        t_(r1)+(q);                             * GET NEXT VALUE FOR DISPATCH
        loopChk[slowBrL,CNT#0&-1],q_t;

afterBD:
                goto[afterKernel4];

SET[BDTblLoc,5110];
BDTbl:
                goto[bdConverge], rscr_r0, AT[BDTblLoc];
                goto[bdConverge], rscr_r1, AT[BDTblLoc,1];
                goto[bdConverge], rscr_2C, AT[BDTblLoc,2];
                goto[bdConverge], rscr_3C, AT[BDTblLoc,3];
                goto[bdConverge], rscr_4C, AT[BDTblLoc,4];
                goto[bdConverge], rscr_5C, AT[BDTblLoc,5];
                goto[bdConverge], rscr_6C, AT[BDTblLoc,6];
                goto[bdConverge], rscr_7C, AT[BDTblLoc,7];
                goto[bdConverge], rscr_10C, AT[BDTblLoc,10];        * shouldn't be here
                goto[bdConverge], rscr_11C, AT[BDTblLoc,11];        * shouldn't be here
                goto[bdConverge], rscr_12C, AT[BDTblLoc,12];        * shouldn't be here
                goto[bdConverge], rscr_13C, AT[BDTblLoc,13];        * shouldn't be here
                goto[bdConverge], rscr_14C, AT[BDTblLoc,14];        * shouldn't be here
                goto[bdConverge], rscr_15C, AT[BDTblLoc,15];        * shouldn't be here
                goto[bdConverge], rscr_16C, AT[BDTblLoc,16];        * shouldn't be here
                goto[bdConverge], rscr_17C, AT[BDTblLoc,17];        * shouldn't be here
```

```
%

                set[TaskCodeloc, 7000];         mc[TaskCodelocC, TaskCodeloc];
                set[TaskErrorloc, 6400];        mc[TaskErrorlocC, TaskErrorloc];
                set[Task17loc, 7400];           mc[Task17locC, Task17loc];
                set[Tasktestloc, 6000];         mc[TasktestlocC, Tasktestloc];
                set[JunkTaskloc, 5400];         mc[JunkTasklocC, JunkTaskloc];
                Toplevel;

beginKernel5:
                call[checkTaskNum], t _ R0;
                skpif[ALU=0];
                branch[AfterKernel5];           *Don't try if not in Task 0
                TaskingOff;
                T _ R0;
                hold&tasksim_t;                 *This will turn off the Task Simulator (Task12).

                Cnt _ RM1;
Kernel5Delay:
                Skpif[cnt=0&-1];
                Goto[Kernel5Delay];             *Try to give Task 12 some time to stop.
                set[xtask,0];
InitTPC:                                        *Inititialize all task tpc's to TaskError code
                Rscr2 _ 1c;                     *Set Rscr2 to 1 for the first task number
                Q _ (RM1);                      *Set Q = 177777, the successful task running will set it back to 0

InitTPCL:
                cnt _ 16S;
                rscr _ 1C;

InitTPCL1:
                TaskingOff;
                T _ TaskErrorlocC;              *Init TPC,s 1 - 17 to TaskError location.
                Link _ T;
                ldTPC _ (Rscr);
                Rscr _ (Rscr)+1;
                Skpif[cnt=0&-1];
                Branch[InitTPCL1];
```

%
TaskSwitch will Notify each task from 1 to 17.
The check for Q=0 would indicate that Q was zeroed by the TaskCode, indicating that the switch happened. I reinitialize all of the TPC's after each test.
All of the unused TPC's are set to TaskError this is a place we should never goto.
We should go to TaskError only when something went wrong like Notifying task 7 and we actually went to task 5.
%

```
TaskSwitch:
        T _ TaskCodelocC;               *TaskSwitch code is run in Task 0.
        Link _ T;
        ldTPC _ (Rscr2);                *Task number is in Rscr2
        Noop;
        Noop;
        TaskingOn;
        T _ (Rscr2);
        Call[NotifyTask];
        noop;
        noop;                           *We should notify Task in Rscr2 then block back to task 0 "Q"
                                        *should be 0 if the correct task number that was notified.

        T _ Q;                          *This is the first instruction to run after TaskCode blocks
        Skpif[Alu=0];
TaskSwitchErr:
        Error;
        Rscr2 _ (Rscr2)+1;              *Increment task number
        Q _ (RM1);                      *Update Q to the current task number
        (Rscr2) # (20C);                *Stop testing when we try to test task 20
        Skpif[Alu=0];
        Branch[InitTPCL];               *Reinitialize all task tpc's for next test
        noop;
        Branch[TaskPreempt];

        set[xtask,17];
TaskCode:
        T _ 0C, at[TaskCodeloc];        *Put a breakpoint here to ovserve each task being Notified
        hold&tasksim_t;                 *This will turn off TestTW just in case it is a problem.
        Q _ T;                          *Put a 0 into Q to prove we went to Taskcode.
        Branch[TaskCode], Block;

TaskError:
        Noop, at[TaskErrorLoc];         *We should never get to here, check Rscr2 for the task
        Error;                          *number we were setting. Check TPC 0 for the last
                                        *microinstruction the EMU ran.
```

```
%
TaskPreempt will Notify each task. The TaskTest code will then notify task 17 there by setting each task's Ready
flip/flop using the preempting logic on the ContA board. The Task17 Code will Block back to the TaskTest code
where Q will be set to 0.
%
TaskPreempt:
            Rscr2 _ 1c;                    *Set Rscr2 to 1 for the first task number to test.
            Q _ (RM1);                     *Set Q to 177777
InitTPCL2:
            Cnt _ 16S;
            Rscr _ 1C;
InitTPCL21:
            TaskingOff;
            T _ TaskErrorlocC;             *Init all task tpc's to TaskError code
            Link _ T;
            ldTPC _ (Rscr);
            Rscr _ (Rscr)+1;
            Skpif[cnt=0&-1];
            Branch[InitTPCL21];
Preempt:
            T _ TaskTestlocC;              *Preempt code is run in Task 0.
            Link _ T;
            ldTPC _ (Rscr2);               *Task number is in Rscr2
            TaskingOn;
            T _ (Rscr2);                   *NotifyTask in a subroutine in Postamble.mc T=Task number.
            Call[NotifyTask];              *This will cause the TaskTest code to be run.

            T _ Q;                         *We will return here from TaskTest code blocking.
            Skpif[Alu=0];                  *Q is set to 0 in the Task Test code.
PreemptErr:                                *The task we were testing is in Rscr2. If Q=0 Testing is ok.
            Error;                         *If Q=17 the task under test did not set it's Ready FF when it
            Rscr2 _ (Rscr2)+1;             *was preempted in the TaskTest code.
            Q _ (RM1);
            (Rscr2) # (17C);               *Stop testing when we get to task 17.
            Skpif[Alu=0];
            Branch[InitTPCL2];             *Reinitialize all task tpc's to TaskError location for the next test.
            Goto[Junktasktest];
TaskTest:
            TaskingOff, at[TaskTestloc];
            T _ Task17locC;
            Link _ T;
            T _ (17s);
            ldTPC _ (T);                   *Task number is 17. Task17 code will be run.
            TaskingOn;                     *Notify task number 17. This will preempt this task and set
            Call[NotifyTask];              *It's Ready FF to indicate it was preempted.

            T _ Q;                         *Task 17 should block back to here with 17 in the Q reg.
            (T) # (17C);
            Skpif[Alu=0];
TaskTestErr:
            Error;                         *If Q is not = 17 then task 17 code didn't run.
            T _ 0C;
            Q _ T;
            Branch[TaskTest], Block;
            set[xtask,17];
Task17:
            T _ 17C, at[Task17loc];        *Set Q to a 17 to prove we went to task 17.
            Q _ T;                         *Put a breakpoint here to observe each ready FF being set
            Branch[Task17], Block;         *as a result of the code at TaskTest  preempting its own task.
```

```
%
This is a test of the 16us. Pendulum signal coming from the Baseboard to the Ifu board on pin number <176>. The
logic is on page 15 of the IFU board.    The Pendulum signal is a 16 us square wave.
The JunkTw is tied to Task 2 on the Conta Board and can be scoped at pin number <13> on the IFU board. The signal
at pin <13> should be about 32 us.     This test is ignored if the IFU board is not installed in the computer.
%
JunkTaskTest:
            TaskingOff;
            IFUMRH_(R01);                 *Check to see if IFU board installed.
            Rscr_IFUMRH';
            T _ (R10);                    *IFUMRH comes in inverted.
            T # (Rscr);
            Skpif[Alu=0];
            Goto[Kernel5Exit];            *No IFU board installed goto Kernel5Exit.
            T _ JunkTasklocC;             *Set up Tpc 2 for 5400.
            Link _ T;                     *5400 = JunkTask.
            T _ (2s);
            ldTPC _ (T);                  *Junk Task is task number is 2.
            Q_(R0);
            Cnt _ RM1;
            AckJunkTW _ (RM1);
            TaskingOn;
JunkTaskSpin:                             *See if we can wake up the Junk Task for 1 count.
            T_ (Q);
            T _ T # (R0);
            Skpif[Alu=0];
            Goto[JunkTaskStart];          *This code will also syschronize the Counter with the microcode
            Skpif[cnt=0&-1];
            Goto[JunkTaskSpin];
PendulumDead:                             *Check pin #<176> on the Baseboard
            Error;
JunkTaskStart:
            Cnt _ RM1;
JunkTaskL:
            Skpif[cnt=0&-1];
            Goto[JunkTaskL];              *Patch JunktaskL+2 to be LongGo[JunktaskL] to  scope board.
            TaskingOff;
            T _ (0c);
            AckJunkTW _ (T);              *Just to be sure we turn off JunkTW.
            T _ Q;                        *T and Q will have the count the JunkTask wakeups.
            Rscr _ 200c;                  *Normal count in Q is 404 with clocks set to 31.7ns.
            Rscr2 _ T - (Rscr);
            SkpUnless[Alu<0];
JunkTaskErrSlow:
            Error;                        *An error here says the counter is too slow!
            Rscr _ 1000c;
            Rscr2 _ T - (Rscr);
            Skpif[Alu<0];
JunkTaskErrFast:
            Error;                        *An error here says the counter is too fast!
            Goto[TestTW];
JunkTask:
            Noop, at[JunkTaskloc];
            T _ (1c);
            AckJunkTW _ (T);
            T _ Q;                        *Set Q to +1 to count the times we ran the JunkTask code.
            T _ (T)+1;
            Q _ T;
            Branch[JunkTask], Block;
```

```
%
This is a test of the TestTW circuit on page 26 of the ProcH board.
The Counter is loaded with a FF=154 (Hold&TaskSim_B) function.
If this test fails the F-16 at location A03 or A04 is usually at fault.
%

TestTW:
            TaskingOff;
            T _ TaskCodelocC;                *Set up Tpc 12 for 7000.
            Link _ T;                        *7000 = TaskCode.
            T _ (12s);
            ldTPC _ (T);                     *TestTW is task number is 12.
            TaskingOn;
            Q _ (RM1);                       *Set Q = 177777, the successful task running will set it back to 0
            T _ (400C);
            hold&tasksim_t;                  *This will set the A04 chip to the smallest count.
            Cnt _ (T);
TestTWL:
            Skpif[cnt=0&-1];
            Goto[TestTWL];                   *Wait for a while to let task 12 wake-up and zero the Q register.

            T _ Q;
            Skpif[Alu=0];                    *Q is set to 0 in the Task Test code.
TestTWError:
            Error;                           *The TestTW  logic on Page 26 of the ProcH board is not working.

Kernel5Exit:
            Goto[AfterKernel5];
```