% Page Numbers: Yes First Page: 1
Heading:
MODEL 1: preamble.mc September 19, 1979 10:42 AM %
% 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 10:46 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:
Aplusl
cntFcn
aluLTO
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[r01,62526]; 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[skipif, BRGO[tsd] BATS[.+2,#1,#2]];
m[skipUless, BRGO[tsd] DBAT[.+1,#2,#1,#2]];
m[loopChk, BRGO[tsd] DBAT[.+1,#2,#1,#3]];
m[loopUntil, BRGO[tsd] DBAT[.+1,#2,#1,#3]]: * if #1 then goto .+1 else goto #2
m[loopWhile, BRGO[tsd] DBAT[.+2,#1,#3]]: * if #1 then goto #2 else goto .+1
* April 17, 1979 10:47 PM

```
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
rv[chkSimulatingRtn, 0];
rv[FixSimRtn, 0];
rv[chkRunSimRtn, 0];
rnRegion[randomRM];
knowBase[randomRM];
rv[randm0, 134134];  rv[randm1, 054206];
rv[randm2, 036711];  rv[randm3, 103625];
rv[randm4, 117263];  rv[randm5, 154737];
rv[randm6, 041344];  rv[randm7, 006712];
knowBase[defaultRegion];
mp[flags.condition0P, 200];  * bit that indicates conditional simulating
mp[flags.condition0Kp, 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[taskDevice, 12];  mc[taskDeviceC, taskDevice];
mc[taskMask, 377];  set[taskShift, 0];
mc[taskMask, 177400];  set[taskShift, 10];
m[1h, byt0[ and[rshift[1#1, 10], 377] ] byt1[ and[#1, 377]]];
* assemble data for left half of IM
m[1h, byt2[ and[rshift[1#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)]
ilc((hold&taskSim + #1)]
```

* preamble.mc   25-Sep-80 14:22:08
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[(return, 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[returnP, subroutine[]
  ilc[(stkpl-1)]
  subroutine[]
  ilc[(link~(stack&-1))]
  ilc[(return)]
];

m[returnPAndBranch, subroutine[]
  ilc[(link~(stack&-1))]
  ilc[(return, P(O ~ #1))]
];

m[returnPAndBranch, subroutine[]
  ilc[(stkpl-1)]
  ilc[(link~(stack&-1))]
  ilc[(return, P(O ~ #1))]
];

m[getRandom, ilc[(RBASE ~ rbase[randX])]
  ilc[(randX ~ (randX)+1, Bdispatch ~ randX)]
  ilc[(call[random], RBASE ~ rbase[randM0])]}
  ilc[(RBASE ~ rbase[defaultRegion])]]
]; /* Returns random number in T, leaves RBASE=defaultRegion */

knowRbase[defaultRegion];
This file defines ALU operations for the program. Definitions for all possible ALU operations are given here with one line/definition. The lines for the 20 operations enabled should have the "n" replace **d by consecutive values from 0 to 17. 16 should be "NOT A" for the shifter and 17 is normally used as a **s a variable by BitBlit (i.e., not defined here), but can be used otherwise, if desired. Other lines **should be commented out.

Note that the letter "E" added as an argument where noted below converts the operation to emulator-only **. This is intended for locations used as variables by BitBlt, which must restore the smashed ALUFM 1 **ocations to the proper value before exit.

The "A" and "B" operations must both be defined because many macros optionally route using either A or **B for RB, T, MD, and Q (B path is preferred). Since ALUF[0] is the default for microinstructions, it ** should contain a non-arithmetic operation to preserve CARRY and OVERFLOW branch conditions.

Also, when running Midas the "B" operation is stored in ALUFM 0 and the NOT A operation in ALUFM 16, so ** these should be loaded the same way by the microprogram to avoid confusion, unless there is some goo **d reason to do otherwise.

Note that an important choice must be made between the arithmetic and logical versions of ALU-A. The a **rithmetic version allows XORCARRY with ALU-A but smashes OVERFLOW and CARRY branch conditions on ALU- **A.
ABOPB[PD+,0,25]; *ALU-B (use n=0)
ABOPA[PD+,1,0]; *ALU+A
*(arithmetic--so XORCARRY ok. Requires no more
*time than boolean "A" in absence of carryin)
* ABOPA[PD+,n,37]; *ALU-A (logical, so XORCARRY illegal but OVERFLOW
*and CARRY branch conditions not smashed).

**"NOT B" and "NOT A" must be both defined also.
ABOPB[NOT+,15,13]; *NOT B
ABOPA[NOT+,16,1]; *NOT A (use n=16 for shifter)

*Operations of no args (4th arg = letter E makes emulator-only)
ZALUOP[AO,2,31]; *AO [= all zeroes]
ZALUOP[A1,3,7]; *A1 [= all ones]

*Arithmetic operations of two args [do not accept "slow" (external)
*B sources] (5th arg = letter E makes emulator-only)
SALUOP[+,A,14]; *A+B
SALUOP[-,6,222]; *A-B

*Boolean operations of two args (5th arg = letter E makes emulator-only)
**Note how synonyms are defined for # (XOR) and = (XNOR, EQV)
ALUOP[+,AND,6,35]; *A AND B
ALUOP[+,OR,7,27]; *A OR B
ALUOP[#,A,10,23]; XALUOP[+,XOR,.,n]; *A # B (A XOR B)
ALUOP[-,A,11,33]; *A AND NOT B [= A AND (NOT B)]
ALUOP[.,OR,12,17]; *A OR NOT B [= A OR (NOT B)]
* ALUOP[NOT,AND,.,21]; *NOT A AND B [= (NOT A) AND B]
* ALUOP[NOT,OR,.,6]; *NOT A OR B [= (NOT A) OR B]
* ALUOP[NOT,AND,NOT,.,11]; *NOT A AND NOT B [= (NOT A) AND (NOT B)]
* ALUOP[NOT,OR,NOT,.,3]; *NOT A OR NOT B [= (NOT A) OR (NOT B)]

*Operations of one arg (5th arg = letter E makes emulator-only)
* AOP[2,.,6]; *2A
* AOP[2,1,n,206]; *2A+1
AOP[+,1,13,200]; *A+1
AOP[-,1,14,36]; *A-1
% Page Numbers: Yes First Page: 1

Heading:
kernel.mc February 1, 1980 9:20 PM %

TITLE[KERNEL]:
IM[ILC,O];
TOP LEVEL:
* February 1, 1980 9:20 PM
restartDiagnostic:
BEGIN:
  goto[im0];
afterKernel1:
  goto[beginKernel12];
afterKernel2:
  goto[beginKernel13];
afterKernel3:
  goto[beginKernel14];
afterKernel4:
T+RO;
  * RO 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=#(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 525252B 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[dbgTb1s,100];
11:  branch[11], at[dbgTb1s,0];
12:  noop, at[dbgTb1s,1];
     branch[12], at[dbgTb1s,2];
13:  noop, at[dbgTb1s,3];
     noop, at[dbgTb1s,4];
     branch[13], at[dbgTb1s,5];
14:  noop, at[dbgTb1s,6];
     noop, at[dbgTb1s,7];
     noop, at[dbgTb1s,10];
     branch[14], at[dbgTb1s,11];
15:  noop, at[dbgTb1s,12];
     noop, at[dbgTb1s,13];
     noop, at[dbgTb1s,14];
     noop, at[dbgTb1s,15];
     branch[15], at[dbgTb1s,16];
16:  noop, at[dbgTb1s,17];
     noop, at[dbgTb1s,20];
     noop, at[dbgTb1s,21];
     noop, at[dbgTb1s,22];
     noop, at[dbgTb1s,23];
     branch[16], at[dbgTb1s,24];
END;
% Page Numbers: Yes First Page: 1
Heading:
kernell.mc May 8, 1979 11:40 AM %
* INSERT[DIALU.MC];
* TITLE[PROG1];
* INSERT[PREAMBLE.MC];
%
May 8, 1979 11:40 AM
  Add RoodByPass tests at enbd of xorBypass
March 26, 1979 10:58 AM
  Add overflow test
March 10, 1979 6:43 PM
  Add test 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
%
%
TEST DESCRIPTION
(singletest)  Check RM to T, T to RM movement
aluEQO  check the fast branch code
aluLT0  check the fast branch code
rEven  check the fast branch code
rGEO  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;
kernell:
IM0:  T=RMI;  *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=00;
       * check that FF,0 works
IM24:  T=77400C;
IM25:  T=1B15;  * check that 0,FF works
IM26:  T=376C;
* September 15, 1978 10:19 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. A
**ACTUALLY, the initial set of tests will check amux sources too!

T contains the value received.
%
aluEQOFF:
t+B0;
skpUnless[ALU=0], rscr+(A+t);
error; * Thinks bit0 is zero
aluEqOFF:
t+B1;
skpUnless[ALU=0], rscr+(A+t);
error; * Thinks bit1 is zero

aluEqOFFB1:
t+B2;
skpUnless[ALU=0], rscr+(A+t);
error; * Thinks bit2 is zero
aluEqOFFB2:
t+B3;
skpUnless[ALU=0], rscr+(A+t);
error; * Thinks bit3 is zero
aluEqOFFB3:
t+B4;
skpUnless[ALU=0], rscr+(A+t);
error; * Thinks bit4 is zero
aluEqOFFB4:
t+B5;
skpUnless[ALU=0], rscr+(A+t);
error; * Thinks bit5 is zero
aluEqOFFB5:
t+B6;
skpUnless[ALU=0], rscr+(A+t);
error; * Thinks bit6 is zero
aluEqOFFB6:
t+B7;
skpUnless[ALU=0], rscr+(A+t);
error; * Thinks bit7 is zero
error;  * Thinks bit7 is zero
  t=B8;
skpUnless[ALU=0], rscr~(A+t);
aluEqOFFB8: error;  * Thinks bit8 is zero
      skpUnless[ALU=0]; error;  * Thinks bit8 is zero
  t=B9;
skpUnless[ALU=0], rscr~(A+t);
aluEqOFFB9: 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);
aluEqOFFB10: error;  * Thinks bit10 is zero
      skpUnless[ALU=0]; error;  * Thinks bit10 is zero
  t=B11;
skpUnless[ALU=0], rscr~(A+t);
aluEqOFFB11: error;  * Thinks bit11 is zero
      skpUnless[ALU=0]; error;  * Thinks bit11 is zero
  t=B12;
skpUnless[ALU=0], rscr~(A+t);
aluEqOFFB12: error;  * Thinks bit12 is zero
      skpUnless[ALU=0]; error;  * Thinks bit12 is zero
  t=B13;
skpUnless[ALU=0], rscr~(A+t);
aluEqOFFB13: 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);
aluEqOFFB14: 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);
aluEqOFFB15: 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!

%

aluEqORT:
  t=r0;
skipf[alu=0];
error;  * Thinks r0 is zero
rscr~t;
skipf[alu=0], t=r1;
error;
  skpUnless[alu=0];
error; * Thinks r1 is zero
  rscr=t;
skpUnless[alu=0],t=rm1;
  error;

aluEqORTM1:
  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;

aluEqORT01:
  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

% aluLTORT:
  t=rhigh1;
skip[alu<0];
  error; * Thinks rhigh1 >=0
  rsr=t;
  skip[alu<0];
  error; * Thinks T (=RIGH1) >=0
  t=r10;
  skip[alu<0];
aluLTORT10:
  error; * Thinks r10 >= 0
  rsr=t;
  skip[alu<0];
  error; * Thinks T (=r10) >=0
  t=r1;
  skipUnless[alu<0];
aluLTORT1: error; * Thinks r1<0
  rsr=t;
  skipUnless[alu<0];
  error; * Thinks T (=r1) >=0
  t=r01;
  skipUnless[alu<0];
aluLTORT01:
  error; * Thinks r10 >= 0
  rsr=t;
  skipUnless[alu<0];
  error; * Thinks T (=r10) >=0

* TEST FOR RESULT EVEN

rEven:
  skip[r even], t=r0;
  error; * thinks r0 odd
  skipUnless[r even], t=r1;
  error; * Thinks r1 EVEN
  skip[r even], t=rhigh1;
  error; * Thinks rhigh1 ODD
  skipUnless[r even], t=r01;
  error; * Thinks r01 EVEN
  skip[r even], t=r10;
  error; * Thinks r10 ODD

rGEO:
  skip[r >=0], t=r1;
  error; * Thinks r1 <0
  skip[r>=0], t=r01;
  error; * Thinks r01 <0
  skip[r>=0], t=r0;
  error; * Thinks r0 <0
  skipUnless[r>=0], t=rml;
  error; * Thinks RM1>=0
  skipUnless[r>=0], t=rhigh1; * Thinks rhigh1 >=0
error;
* January 9, 1979 10:42 AM

reschedTest
Set and clear resched; see if we can branch on its value.

reschedTest:
call[checkTaskNum], t+t-t;
skpif[ALU=0];
branch[afterResched];

noreschedule[];
skpif[reschedule'];
reschedErr1: * we just cleared resched, yet error; * branch condition thinks it is set.
reschedule[];
skpif[reschedule];
reschedErr2: * we just set resched, yet the error; * branch condition doesn't realize it.
noreschedule[];
afterResched:
September 15, 1978 10:56 AM
TEST XOR USING ALU=0, USE NOOP TO AVOID BYPASS.

Generally, \( T \leftarrow \text{RSCR} \leftarrow \text{someFFconstant}; \)
\( T \leftarrow T\oplus \text{RSCR} \)
IF \( T \) is non zero, there was an error: one bits in \( T \) indicate
the problem.

\[
xor\text{NoBypass}:\]
\[
t\leftarrow (\text{rscr})+B0;\]
\[
\text{noop}; t\leftarrow t\#(\text{rscr});\]
\[
\text{skpif[alu=0]};\]
\[
\text{error}; \quad * (T \leftarrow B0 \text{ xor (RSCR)}) \quad \text{NE} \quad 0\]
\[
t\leftarrow (\text{rscr})+B1;\]
\[
\text{noop}; t\leftarrow t\#(\text{rscr});\]
\[
\text{skpif[alu=0]};\]
\[
\text{error}; \quad * (T \leftarrow B1 \text{ xor (RSCR)}) \quad \text{NE} \quad 0\]

\[
xor\text{NoBypassB2}:\]
\[
t\leftarrow (\text{rscr})+B2;\]
\[
\text{noop}; t\leftarrow t\#(\text{rscr});\]
\[
\text{skpif[alu=0]};\]
\[
\text{error}; \quad * (T \leftarrow B2 \text{ xor (RSCR)}) \quad \text{NE} \quad 0\]
\[
t\leftarrow (\text{rscr})+B3;\]
\[
\text{noop}; t\leftarrow t\#(\text{rscr});\]
\[
\text{skpif[alu=0]};\]
\[
\text{error}; \quad * (T \leftarrow B3 \text{ xor (RSCR)}) \quad \text{NE} \quad 0\]
\[
t\leftarrow (\text{rscr})+B4;\]
\[
\text{noop}; t\leftarrow t\#(\text{rscr});\]
\[
\text{skpif[alu=0]};\]
\[
\text{error}; \quad * (T \leftarrow B4 \text{ xor (RSCR)}) \quad \text{NE} \quad 0\]

\[
xor\text{NoBypassB4}:\]
\[
\text{error}; \quad * (T \leftarrow B4 \text{ xor (RSCR)}) \quad \text{NE} \quad 0\]
\[
t\leftarrow (\text{rscr})+B5;\]
\[
\text{noop}; t\leftarrow t\#(\text{rscr});\]
\[
\text{skpif[alu=0]};\]
\[
\text{error}; \quad * (T \leftarrow B5 \text{ xor (RSCR)}) \quad \text{NE} \quad 0\]

\[
xor\text{NoBypassB6}:\]
\[
t\leftarrow (\text{rscr})+B6;\]
\[
\text{noop}; t\leftarrow t\#(\text{rscr});\]
\[
\text{skpif[alu=0]};\]
\[
\text{error}; \quad * (T \leftarrow B6 \text{ xor (RSCR)}) \quad \text{NE} \quad 0\]
\[
t\leftarrow (\text{rscr})+B7;\]
\[
\text{noop}; t\leftarrow t\#(\text{rscr});\]
\[
\text{skpif[alu=0]};\]
\[
\text{error}; \quad * (T \leftarrow B7 \text{ xor (RSCR)}) \quad \text{NE} \quad 0\]

\[
xor\text{NoBypassB8}:\]
\[
t\leftarrow (\text{rscr})+B8;\]
\[
\text{noop}; t\leftarrow t\#(\text{rscr});\]
\[
\text{skpif[alu=0]};\]
\[
\text{error}; \quad * (T \leftarrow B8 \text{ xor (RSCR)}) \quad \text{NE} \quad 0\]
\[
t\leftarrow (\text{rscr})+B9;\]
\[
\text{noop}; t\leftarrow t\#(\text{rscr});\]
\[
\text{skpif[alu=0]};\]
\[
\text{error}; \quad * (T \leftarrow B9 \text{ xor (RSCR)}) \quad \text{NE} \quad 0\]

\[
xor\text{NoBypassB10}:\]
\[
t\leftarrow (\text{rscr})+B10;\]
\[
\text{noop}; t\leftarrow t\#(\text{rscr});\]
\[
\text{skpif[alu=0]};\]
\[
\text{error}; \quad * (T \leftarrow B8 \text{ xor (RSCR)}) \quad \text{NE} \quad 0\]
\[
t\leftarrow (\text{rscr})+B11;\]
\[
\text{noop}; t\leftarrow t\#(\text{rscr});\]
\[
\text{skpif[alu=0]};\]
error;  *(T - B10 xor (RSCR) ) NE 0

t+=rscr+B12;
noop:t-t#(rscr);
skipf[alu=0];

xorNoBypassB12:
  error;  *(T - B12 xor (RSCR) ) NE 0

t+=rscr+B13;
noop:t-t#(rscr);
skipf[alu=0];
error;  *(T - B13 xor (RSCR) ) NE 0

xorNoBypassB14:
  t+=rscr+B14;
  noop:t-t#(rscr);
skipf[alu=0];
error;  *(T - B14 xor (RSCR) ) NE 0

t+=rscr+B15;
noop:t-t#(rscr);
skipf[alu=0];
error;  *(T - B15 xor (RSCR) ) NE 0
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.

```c
rvrel[rmx10, 10];
bypass:
  RBASE ~ 0s;
  t=rmx0+cml;  /* 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];
bypassErr0:  /* bypass associated w/ rm addr 0 doesn't error; * seem to work*/
  t=rmx1+cml;  /* 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];
bypassErr1:  /* bypass associated w/ rm addr 1 doesn't error; * seem to work*/
  t=rmx2+cml;  /* 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];
bypassErr2:  /* bypass associated w/ rm addr 2 doesn't error; * seem to work*/
  t=rmx4+cml;  /* 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];
bypassErr4:  /* bypass associated w/ rm addr 4 doesn't error; * seem to work*/
  t=rmx10+cml; /* 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];
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+cml;  /* 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+cml;  /* 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+cml;  /* 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*/

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:

xorBypass2:

xorBypass4:

xorBypass6:

xorBypass8:

xorBypass10:
t+t#(rscr);
skipf[ALU=0];
error; *(T+B11 xor (RSCR)) NE 0

t+(rscr)+B12;
t+t#(rscr);
skipf[ALU=0];
xorBypassB12:
error; *(T+B12 xor (RSCR)) NE 0

t+(rscr)+B13;
t+t#(rscr);
skipf[ALU=0];
error; *(T+B13 xor (RSCR)) NE 0

t+(rscr)+B14;
t+t#(rscr);
skipf[ALU=0];
xorBypassB14:
error; *(T+B14 xor (RSCR)) NE 0

t+(rscr)+B15;
t+t#(rscr);
skipf[ALU=0];
error; *(T+B15 xor (RSCR)) NE 0

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

rscr += t-t;
skipUnless[R ODD], rscr;
RoddByPassErr1: * fast branch r odd bypass doesn't work.
error; * rscr has zero in it.
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=6C;
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+(rml);  * 0=1+-1
skpif[alu=0];
error;

AplusBi:
  t=r10;
  t=t+(r01);
  t=t#(rml);  * -1=52525+125252
  skpif[alu=0];
  error;

AplusBj:
  t=high1;
  t=t+(high1);  * 0=100000+100000
  skpif[alu=0];
  error;

AplusBk:
  t=hight1;
  t=t+(rml);
  rscr=77777C;
  t=t#(rscr);  * 77777=100000+177777
  skpif[alu=0];
  error;

AplusBl:
  t=rml;
  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#(rml);
  skpIF[alu=0];
  error;  * T SHOULD HAVE BEEN -1

aMinusBb:
  t=r0;
  t=t-(rml);   * t+0 - (-1)
  t=t#(r1);
  skpIF[alu=0];
  error;  * T SHOULD HAVE BEEN 1

aMinusBc:
  t=r0;
  t=t-(rhg1);   * t+ 0 - (100000)
  t=t#(rhg1);
  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#(ro);
  skpIF[alu=0];
  error;  * T SHOULD HAVE BEEN 0

aMinusBf:
  t=rscr=177C;
  t=t-(rscr);   * t = 177 - 177
  t=t#(ro);
  skpIF[alu=0];
  error;  * T SHOULD HAVE BEEN 0

aMinusBg:
  t=rscr=377C;
  t=t-(rscr);   * t = 377 - 377
  t=t#(ro);
  skpIF[alu=0];
  error;  * T SHOULD HAVE BEEN 0

aMinusBh:
  t=rscr=400C;
  t=t-(rscr);   * t = 400 - 400
  t=t#(ro);
  skpIF[alu=0];
  error;  * T SHOULD HAVE BEEN 0

aMinusBi:
  t=rscr=777C;
  t=t-(rscr);   * t = 777 - 777
  t=t#(ro);
  skpIF[alu=0];
  error;  * T SHOULD HAVE BEEN 0

aMinusBj:
  t=rscr=1777C;
  t=t-(rscr);   * t = 1777 - 1777
  t=t#(ro);
  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=r1;
t=t+(r0);
skpUnless[carry];
error; * r1 + 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=r1;
t=t-(r1);
skpiIf[carry];
error; * -1 -(+1) SHOULD CAUSE CARRY

carryYesb:
t=r1;
t=t+(rhigh1);
skpiIf[carry];
error; * -1 + 100000 SHOULD CAUSE CARRY

carryYesc:
t=r1;
t=t+(r1);
skpiIf[carry];
error; * -1 + 1 SHOULD CAUSE CARRY

carryYesd:
t=rhigh1;
t=t-(r01);
skpiIf[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

carryOpsf:
  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
reschedule;
* March 10, 1979 6:42 PM
* test the branch conditions when reschedule is ON
  t+r0;
  t=t-(r1);  * t+0-1
  skpUnless[carry].t+t+(rm1);  * t-1+-1
  error;  * 0-1 SHOULD NOT CAUSE CARRY

carryOpsRb:
  skp1f[carry]. t=t+(rhigh1);  * t+ -2+100000
  error;  * -1+-1 SHOULD CAUSE CARRY

carryOpsRc:
  skp1f[carry]. t=t+(rhigh1);  * t= 77776 + 100000
  error;  * 77776 + 100000 SHOULD NOT CAUSE CARRY

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

carryOpsRe:
  t+rm1;
  t=t-(r1);  * t+-1-(+1)
  skp1f[carry]. t=t-(r01);  * t+-2-r01
  error;  * -1+-1 SHOULD CAUSE CARRY

carryOpsRf:
  skp1f[carry];
  error;  * 177776 - 52525 SHOULD CAUSE CARRY

t+r0;
  skp1f[ALU=0];
rescheq0br:
  error;
  t=r1;
  skp1f[ALU#0];
reschne0br:
  error;
  skp1f[r even)]; B=r0;
reschevenbr:
  error;
  skp1f[r odd)]; B=r1;
reschoddbr:
  error;
  t=rhigh1;
  skp1f[alu<0];
reschlt0br:
  error;
  t+r0;
  skp1f[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=rml;
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=(rml)+(rml),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=t+AND(rml),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=rml,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=rml,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=(rml)+(rml),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;
Perform an exhaustive test of the overflow condition. Even though we expect the arithmetic result of $R^{M+T}$ 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 |
|          | 01 | 00 | 01 | 00 |

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

```assembly
overflowTest:
    t + t - t;
    t + t + t;
    skpif[overflow']
overflowErr0: * 0+0 should not cause overflow error;
    t=t-t;
    t+t+(x01);
    skpif[overflow']
overflowErr1: * see 0 + 01 entry error;
    t + t - t;
    t + t + (x10);
    skpif[overflow']
overflowErr2: * see 0 + 10 entry error;
    t=t-t;
    t+t+(x11);
    skpif[overflow']
overflowErr3: * see 0 + 11 entry error;
    t=rscr*x01; * keep x01 in rscr for a while
    t=t+(0c);
    skpif[overflow']
overflowErr4: * see 01 + 0 entry error;
    t=rscr;
    t+t+(x01);
    skpif[overflow']
overflowErr5: * see 01 + 01 entry error;
    t=rscr;
    t+t+(x10);
    skpif[overflow']
overflowErr6: * see 01 + 10 entry error;
    t=rscr;
    t+t+(x11);
    skpif[overflow']
overflowErr7: * see 01 + 11 entry error;
    t=rscr*x10; * keep x10 in rscr for a while
    t=t+(0c);
    skpif[overflow']
overflowErr10: * see 10 + 0 entry error;
    t=rscr;
    t+t+(x01);
    skpif[overflow']
overflowErr11: * see 10 + 01 entry
```

error;
t+rscr;
t + t + (x10);
skipif[overflow];
overflerr12:
  error;  * see 10 + 10 entry
t+rscr;
t + t + (x11);
skipif[overflow];
overflerr13:  * see 10 + 11 entry
  error;
    t + rscr *x11;  * keep x11 in rscr for a while
t + t + (0c);
skipif[overflow'];
overflerr14:  * see 11 + 0 entry
  error;
t+rscr;
t + t + (x01);
skipif[overflow'];
overflerr15:  * see 11 + 01 entry
  error;
t+rscr;
t + t + (x10);
skipif[overflow];
overflerr16:
  error;  * see 11 + 10 entry
t+rscr;
t + t + (x11);
skipif[overflow'];
overflerr17:  * see 11 + 11 entry
  error;
goto[afterKernel11];
% Page Numbers: Yes First Page: 1
Heading:
kernel2.mc January 18, 1979 2:07 PM
* INSERT[DIALU.MC]:
* TITLE[KERNEL2]:
* INSERT[PREAMBLE.MC]:
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
rstkTestO write different RM address from one read
%
%
Add tioaTest
%

January 18, 1979 2:07 PM
* 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++; -- 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
NotAtest:
\[ t \leftarrow \text{not}(A \oplus 0); \]
\[ t \leftarrow \text{t}#(\text{rm1}); \]
\[ \text{skpif}[\text{ALU}=0]; \]
\[ \text{error}; \]
\[ * \rightarrow \text{R0} \ # \ \text{RM} \]

NotAb:
\[ t \leftarrow \text{not}(A \oplus 1); \]
\[ t \leftarrow \text{t}#(\text{r0}); \]
\[ \text{skpif}[\text{ALU}=0]; \]
\[ \text{error}; \]
\[ * \rightarrow \text{t} \ # \ \text{r0} \]

NotAc:
\[ t \leftarrow \text{not}(A \oplus \text{rm1}); \]
\[ t \leftarrow \text{t}#(\text{r10}); \]
\[ \text{skpif}[\text{ALU}=0]; \]
\[ \text{error}; \]
\[ * \rightarrow \text{rm1} \ # \ \text{r0} \]

NotAd:
\[ t \leftarrow \text{not}(A \oplus 10); \]
\[ t \leftarrow \text{t}#(\text{r01}); \]
\[ \text{skpif}[\text{ALU}=0]; \]
\[ \text{error}; \]
\[ * \rightarrow \text{r01} \ # \ \text{r01} \]

NotAe:
\[ \text{rscr} \leftarrow 177776; \]
\[ t \leftarrow \text{not}(A \oplus 11); \]
\[ t \leftarrow \text{t}#(\text{rscr}); \]
\[ \text{skpif}[\text{ALU}=0]; \]
\[ \text{error}; \]
\[ * \rightarrow \text{r1} \ # \ 177776 \]

NotBtest:
\[ t \leftarrow \text{not}(B \oplus 0); \]
\[ t \leftarrow \text{t}#(\text{rm1}); \]
\[ \text{skpif}[\text{ALU}=0]; \]
\[ \text{error}; \]
\[ * \rightarrow \text{R0} \ # \ \text{RM} \]

NotBTestb:
\[ t \leftarrow \text{not}(B \oplus 1); \]
\[ t \leftarrow \text{t}#(\text{r0}); \]
\[ \text{skpif}[\text{ALU}=0]; \]
\[ \text{error}; \]
\[ * \rightarrow \text{t} \ # \ \text{r0} \]

NotBTestc:
\[ t \leftarrow \text{not}(B \oplus 11); \]
\[ t \leftarrow \text{t}#(\text{r10}); \]
\[ \text{skpif}[\text{ALU}=0]; \]
\[ \text{error}; \]
\[ * \rightarrow \text{rm1} \ # \ \text{r0} \]

NotBTestd:
\[ t \leftarrow \text{not}(B \oplus 10); \]
\[ t \leftarrow \text{t}#(\text{r01}); \]
\[ \text{skpif}[\text{ALU}=0]; \]
\[ \text{error}; \]
\[ * \rightarrow \text{r01} \ # \ \text{r01} \]

NotBTeste:
\[ \text{rscr} \leftarrow 177776; \]
\[ t \leftarrow \text{not}(B \oplus 1); \]
\[ t \leftarrow \text{t}#(\text{rscr}); \]
\[ \text{skpif}[\text{ALU}=0]; \]
\[ \text{error}; \]
\[ * \rightarrow \text{r1} \ # \ 177776 \]

% Test A AND B
Assume a,b source dont matter. Ie.,
\[ t \leftarrow (b \oplus t) \text{ and } (a \oplus r) = \]
\[ t \leftarrow (a \oplus t) \text{ and } (b \oplus r) \]

AandBtest:
\[ t \leftarrow \text{rm1}; \]
\[ t \leftarrow \text{tAND}(\text{rm1}); \]
t+=#(rm1);
skp if[ALU=0];
error;  * (rm1 AND rm1) #rm1

t+r01;
t+AND(r10);
skp if[ALU=0];
error;  * (r01 AND r10)

t+r0;
t+AND(rm1);
skp if[ALU=0];
error;  * r0 AND rm1

%  
Test A or B.  
Assume same as Aand B test.
%
AorBtest:
t=+rm1;
t=+OR(r0);
t=+=#(rm1);
skp if[ALU=0];
error;  * (rm1 OR r0) # rm1

AorBtestb:
t=r01;
t=+OR(r10);
t=+=#(rm1);
skp if[ALU=0];
error;  * (r01 OR r10) # rm1

AorBtestc:
t=+rm1;
t=+OR(rm1);
t=+=#(rm1);
skp if[ALU=0];
error;  * (rm1 OR rm1) # rm1

AorBtestd:
t=r01;
t=+OR(r01);
t=+=#(r01);
skp if[ALU=0];
error;  * (r01 OR r01) # r01

AorBteste:
t=r10;
t=+OR(r10);
t=+=#(r10);
skp if[ALU=0];
error;  * (r10 OR r10) # r10

AorBtestf:
t=+(r0)OR(r0);
t=+=#(r0);
skp if[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:
  rscre=7777C;  * BEGIN W/ MAX LINK VALUE & COUNT DOWN
linkL:
  link = rscre;
  t = link;
  t = t and (77777C);  * ISOLATE 15 BITS 'CAUSE OF DMUX DATA
  t=t#(rscre);
  skpif[alu=0];
linkErr1:
  error;  * LINK DOESN'T HAVE THE VALUE WE LOADED
  rscre=(rscre)-1;
  dblBranch[linkL,afterLink, alu<0];

afterLink:
* November 3, 1978 6:40 PM

TEST Q: READ AND WRITE

FOR I IN [0..17777B] 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;
  PD = q;
  skpif[ALU=0];
qr0Lerr:
  * r0 lsh 1 should be zero
  error;
  q = r01;
  q lsh 1;
  (q) # (r01);
  skpif[ALU=0];
qr10Lerr:
  * r10 lsh1 should be r01. (zero fill)
  error;
  q = rml;
  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;
  PD = q;
  skpif[ALU=0];
qhigh1Lerr:
  * rhigh1 (100000B) lsh1 w/ zero fill should
  error; * zero
  q = r0;
  q rsh 1;
  PD = q;
  skpif[ALU=0];
qr0Rerr:
  error; * zero rsh1 should be zero
  q = r10;
  q rsh 1;
  (q) # (r01);
  skpif[ALU=0];
qr10Rerr:
  * r10 rsh 1 w/ zero fill
  error; * should be r01
  q = rml;
  q rsh 1;
  t = 77777c;
  (q) # t;
  skpif[ALU=0];
qrm1Rerr:
  * -1 rsh 1 w/ zero fill should be 77777B
error;

go to rhigh1;
q rsh 1;   * q = 100000B rsh 1
t = 40000C;
(a) # t;
skipif[ALU=0];
go to rhigh1Rerr;   * rhigh1 rsh 1 should be 40000B
error;
Test the processor's ability to read and write TIOA. Write TIOAk from both FF constants and from RM.

```plaintext
tioaTest:
    t = 377c;
    cnt = t;
    rscr2 = t - t;

tioa:
    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, tioa, rscr2, (rscr2) + (b7)];  * increment rscr2

* Here are device declarations to keep micro happy. We use them to set Tioa directly from FF.
device[dvc5, b131];  device[dvc6, b141];  device[dvc7, b151];
mc[tioa.0thru4C, b0,b1,b2,b3,b4];
mc[tioa.mask, 177400];

tioa = r0;  * zero all the bits 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];

tioaErr2:  * 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

afterTioa:
```
* 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
stkL:
  STKP=rscr;  * LOAD STKP FROM rscr
  t = (TIOA&STKP);
  t=t AND (rscr2);  * READ AND MASK THE VALUE
  t=t#(rscr);
  skpif[ALU=0];
stkErr:
  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.
**ng. 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 # (rmx4);  * 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;
% 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
% [sibling,
%  rm[#1, add[1shift[#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 = cml;  * 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 = cml;  * 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 + cml;  * 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 + cml;  * 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];

rbaseFF3Err:  * 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 + cml;  * 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];

rbaseFF4Err:  * succeed in writing rm with rbase=0 from error;  * ff field. t = real val, rmx0=expected val.

  rmx10 = q;  * restore old value
  q = rmx10;  * save current value for "source" rm
  t = rmx10 + cml;  * 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];
% 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[1]; (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[1]; (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..] DO RBASE[2] + RBASE[1]; (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-rlm1=r10;
t#(Q);
skpUnless[ALU=0], t+t#(rm1);

rstkTest24:
error;
skipf[ALU=0];
error;
t=rm1=rlh1;
t#(Q);
skipUnless[ALU=0], t+t#(rm1);
error;
skipf[ALU=0];
error;
t=rm1=rscr;
t#(Q);
skipUnless[ALU=0], t+t#(rm1);

rstkTest26:
error;
skipf[ALU=0];
error;
t=rm1=rscr2;
t#(Q);
skipUnless[ALU=0], t+t#(rm1);
error;
skipf[ALU=0];
error;
rm1=Q;

* FOR I IN [0..7] DO RBASE[3] + RBASE[I]; (EXCEPT FOR *RBASE[3])

rstkTest3:
Q-rl01;
t=rl01+rl0;
t#(Q);
skipUnless[ALU=0], t+t#(rl01);
error;
skipf[ALU=0];
error;
t=rl01+rl1;
t#(Q);
skipUnless[ALU=0], t+t#(rl01);

rstkTest32:
error;
skipf[ALU=0];
error;
t=rl01+rm1;
t#(Q);
skipUnless[ALU=0], t+t#(rl01);
error;
skipf[ALU=0];
error;
t=rl01+rl10;
t#(Q);
skipUnless[ALU=0], t+t#(rl01);

rstkTest34:
error;
skipf[ALU=0];
error;
t=rl01=rlh1;
t#(Q);
skipUnless[ALU=0], t+t#(rl01);
error;
skipf[ALU=0];
error;
t=rl01=rscr;
t#(Q);
skipUnless[ALU=0], t+t#(rl01);
rstkTest36:
  error;
skplf[ALU=0];
  error;
  t+r01+rscr2;
  t#(Q);
  skpUnless[ALU=0], t+t#(r01);
  error;
skplf[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;
skplf[ALU=0];
  error;
  t+r10+=r1;
  t#(Q);
  skpUnless[ALU=0], t+t#(r10);

rstkTest42:
  error;
skplf[ALU=0];
  error;
  t+r10+=r01;
  t#(Q);
  skpUnless[ALU=0], t+t#(r10);
  error;
skplf[ALU=0];
  error;
  t+r10+=rm1;
  t#(Q);
  skpUnless[ALU=0], t+t#(r10);

rstkTest44:
  error;
skplf[ALU=0];
  error;
  t+r10+=rhigh1;
  t#(Q);
  skpUnless[ALU=0], t+t#(r10);
  error;
skplf[ALU=0];
  error;
  t+r10+=rscr;
  t#(Q);
  skpUnless[ALU=0], t+t#(r10);

rstkTest46:
  error;
skplf[ALU=0];
  error;
  t+r10+=rscr2;
  t#(Q);
  skpUnless[ALU=0], t+t#(r10);
  error;
skplf[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+rml;
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;

FOR I IN [0 .. 7] DO RBASE[6] + RBASE[I]: (EXCEPT FOR +RBASE[6])
rstkTest58:
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;
skipf[ALU=0];
  error;
  t=rscr+rm1;
  t#(Q);
skipUnless[ALU=0], t+t#(rscr);
  error;
skipf[ALU=0];
  error;

rstkTest66:
  error;
skipf[ALU=0];
  error;
  t=rscr+rscr2;
  t#(Q);
skipUnless[ALU=0], t+t#(rscr);
  error;
skipf[ALU=0];
  error;

  FOR I IN [0..7] DO RBASE[7] + RBASE[I]; (EXCEPT FOR +RBASE[7])

rstkTest7:
  Q=rscr2;
  t=rscr2+r0;
  t#(Q);
skipUnless[ALU=0], t+t#(rscr2);
  error;
skipf[ALU=0];
  error;
  t=rscr2+r1;
  t#(Q);
skipUnless[ALU=0], t+t#(rscr2);
  error;
skipf[ALU=0];
  error;

rstkTest72:
  error;
skipf[ALU=0];
  error;
  t=rscr2+r01;
  t#(Q);
skipUnless[ALU=0], t+t#(rscr2);
  error;
skipf[ALU=0];
  error;
  t=rscr2+r10;
  t#(Q);
skipUnless[ALU=0], t+t#(rscr2);
  error;
skipf[ALU=0];
  error;

rstkTest74:
  error;
skipf[ALU=0];
  error;
  t=rscr2+rhigh1;
  t#(Q);
skipUnless[ALU=0], t+t#(rscr2);
  error;
skipf[ALU=0];
  error;
  t=rscr2+rscr;
  t#(Q);
skipUnless[ALU=0], t+t#(rscr2);
  error;
skipf[ALU=0];
  error;

rstkTest76:
  error;
skipf[ALU=0];
  error;
t=rscr2+rm1;
\texttt{t\#(Q); skpUnless[ALU=0], t\#(rscr2); error; skpIf[ALU=0]; error; rscr2\#Q;}
goto[afterKernel12];
% Page Numbers: Yes First Page: 1
Heading:
kernel3.mc November 13, 1978 10:43 AM
* INSERT[DIALU.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
Rrhs  test RM shiftlmask, rsh r[i]; 0<=i<=15
Trsh  test T shiftlmask, rsh t[i]; 0<=i<=15
TRLyTest  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=rO;
  SHC=rscr;
  t+=SHC;
  t=(rscr)##(t);
  branch[.,+2,ALU=0];
  error;
  rscr=(rscr)+1;
  loopUntil[ALU=0,SHCRLW];

\% 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 (bit 0) is zero and all other bits are one.

\% Rlsh:
  t=r1;
  rscr=t;
  t=B15;
  rscr += lsh[rscr,0];
  t=t##(rscr);
  skp1f[ALU=0];
  error;
  t=r1;
  rscr=t;
  t=B14;
  rscr += lsh[rscr,1];
  t=t##(rscr);
  skp1f[ALU=0];
  error;

\% Rlsh2:
  t=r1;
  rscr=t;
  t=B13;
  rscr += lsh[rscr,2];
  t=t##(rscr);
  skp1f[ALU=0];
  error;
  t=r1;
  rscr=t;
t=B12;
rscr <- lsh[rscr,3];
t=t#(rscr);
skipf[ALU=0];
error;

Rlsh4:
   t=r1;
rscr=t;
t=B11;
rscr = lsh[rscr,4];
t=t#(rscr);
skipf[ALU=0];
error;
   t=r1;
rscr=t;
t=B10;
rscr = lsh[rscr,5];
t=t#(rscr);
skipf[ALU=0];
error;

Rlsh6:
   t=r1;
rscr=t;
t=B9;
rscr = lsh[rscr,6];
t=t#(rscr);
skipf[ALU=0];
error;
   t=r1;
rscr=t;
t=B8;
rscr = lsh[rscr,7];
t=t#(rscr);
skipf[ALU=0];
error;

Rlsh8:
   t=r1;
rscr=t;
t=B7;
rscr = lsh[rscr,10];
t=t#(rscr);
skipf[ALU=0];
error;
   t=r1;
rscr=t;
t=B6;
rscr = lsh[rscr,11];
t=t#(rscr);
skipf[ALU=0];
error;

Rlsh10:
   t=r1;
rscr=t;
t=B5;
rscr = lsh[rscr,12];
t=t#(rscr);
skipf[ALU=0];
error;
   t=r1;
rscr=t;
t=B4;
rscr = lsh[rscr,13];
t=t#(rscr);
skipf[ALU=0];
error;

Rlsh12:
   t=r1;
rscr+t;
t+B3;
rscr = lsh[rscr,14];
t+t#(rscr);
skipf[ALU=0];
error;

t+r1;

rscr+t;
t+B2;
rscr = lsh[rscr,15];
t+t#(rscr);
skipf[ALU=0];
error;

R1sh14:

t+r1;

rscr+t;
t+B1;
rscr = lsh[rscr,16];
t+t#(rscr);
skipf[ALU=0];
error;

t+r1;

rscr+t;
t+RHIGH1;
rscr = lsh[rscr,17];
t+t#(rscr);
skipf[ALU=0];
error;
* October 20, 1978  10:32 AM

Tlsh:

```plaintext
t=rscr=B15;
noop;
t=t#(rscr);
skpif[ALU=0];
error;
t=r1;
rscr=B14;
t=t#(rscr);
skpif[ALU=0];
error;
```

Tlsh2:

```plaintext
t=r1;
rscr=B13;
t=t#(rscr);
skpif[ALU=0];
error;
t=r1;
rscr=B12;
t=t#(rscr);
skpif[ALU=0];
error;
```

Tlsh4:

```plaintext
t=r1;
rscr=B11;
t=t#(rscr);
skpif[ALU=0];
error;
t=r1;
rscr=B10;
t=t#(rscr);
skpif[ALU=0];
error;
```

Tlsh6:

```plaintext
t=r1;
rscr=B9;
t=t#(rscr);
skpif[ALU=0];
error;
t=r1;
rscr=B8;
t=t#(rscr);
skpif[ALU=0];
error;
```

Tlsh8:

```plaintext
t=r1;
rscr=B7;
t=t#(rscr);
skpif[ALU=0];
error;
t=r1;
rscr=B6;
t=t#(rscr);
skpif[ALU=0];
```
error;

T1sh10:
  t=r1;
  rscr=B5;
  t=lsh[t,12];
  t=t!(rscr);
  skp if[ALU=0];
  error;

  t=r1;
  rscr=B4;
  t=lsh[t,13];
  t=t!(rscr);
  skp if[ALU=0];
  error;

T1sh12:
  t=r1;
  rscr=B3;
  t=lsh[t,14];
  t=t!(rscr);
  skp if[ALU=0];
  error;

  t=r1;
  rscr=B2;
  t=lsh[t,15];
  t=t!(rscr);
  skp if[ALU=0];
  error;

T1sh14:
  t=r1;
  rscr=B1;
  t=lsh[t,16];
  t=t!(rscr);
  skp if[ALU=0];
  error;

  t=r1;
  rscr=RHIGH1;
  t=lsh[t,17];
  t=t!(rscr);
  skp if[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;
rsr = rsh[rscr,0];
t=t#(rscr);
skipf[ALU=0];
error;

Rrsh1:
 rscr=Q;
t=B1;
rsr = rsh[rscr,1];
t=t#(rscr);
skipf[ALU=0];
error;

Rrsh2:
 rscr=Q;
t=B2;
rsr = rsh[rscr,2];
t=t#(rscr);
skipf[ALU=0];
error;

Rrsh3:
 rscr=Q;
t=B3;
rsr = rsh[rscr,3];
t=t#(rscr);
skipf[ALU=0];
error;

Rrsh4:
 rscr=Q;
t=B4;
rsr = rsh[rscr,4];
t=t#(rscr);
skipf[ALU=0];
error;

Rrsh5:
 rscr=Q;
t=B5;
rsr = rsh[rscr,5];
t=t#(rscr);
skipf[ALU=0];
error;

Rrsh6:
 rscr=Q;
t=B6;
rsr = rsh[rscr,6];
t=t#(rscr);
skipf[ALU=0];
error;

Rrsh7:
 rscr=Q;
t=B7;
rsr = rsh[rscr,7];
t=t#(rscr);
skipf[ALU=0];
error;

Rrsh8:
 rscr=Q;
t=B8;
rsr = rsh[rscr,8];
t=t#(rscr);
skipf[ALU=0];
error;
rsr=Q;
t+B9;
rscr = rsh[rscr,11];
t=t#(rscr);
skpif[ALU=0];
error;

Rrsh10:
rsr=Q;
t+B10;
rscr = rsh[rscr,12];
t=t#(rscr);
skpif[ALU=0];
error;

Rrsh12:
rsr=Q;
t+B12;
rscr = rsh[rscr,14];
t=t#(rscr);
skpif[ALU=0];
error;

Rrsh14:
rsr=Q;
t+B14;
rscr = rsh[rscr,16];
t=t#(rscr);
skpif[ALU=0];
error;

rsr=Q;
t+B16;
rscr = rsh[rscr,17];
t=t#(rscr);
skpif[ALU=0];
error;
Trsh:
   "Trshift1;" * Temporary EXPEDIENT
   t=rscr=RHIGH1:
   NOOP:
   t=rsh[t.0];
   t=t#(rscr);
   skpif[ALU=0];
   error;

Trshift1:
   t=rhs[t.1];
   rscr=B1;
   t=t#(rscr);
   skpif[ALU=0];
   error;

Trsh2:
   t=rhigh1;
   rscr=B2;
   t=rsh[t.2];
   t=t#(rscr);
   skpif[ALU=0];
   error;

Trsh1:
   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;

Trsh6:
   t=rhigh1;
   rscr=B6;
   t=rsh[t.6];
   t=t#(rscr);
   skpif[ALU=0];
   error;

Trsh8:
   t=rhigh1;
   rscr=B8;
   t=rsh[t.10];
   t=t#(rscr);
   skpif[ALU=0];
   error;

Trsh10:
   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;

trhigh1;
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;

trhig1;
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;

trhig1;
rscr=B15;
t=rsh[t,17];
t=t#(rscr);
skpif[ALU=0];
error;
These tests work by cycling by $0, 1, \ldots, 17B$. The predicted result is kept in RSCR2 and the actual result XOR's with the predicted result is kept in T. Note that each test is done twice: once with one "1" bit and all the rest "0" bits, and once with 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=+0;
scr2=B15;
* RSCR2 ← PREDICTED RESULT
t=lc[y[t.r1,0]];
*t=+(rscr2);
скрф[alu=0];
error;
*t=+rm1;
scr2=NB15;
* RSCR2 ← PREDICTED RESULT
*t=lc[y[t.rm2,0]];
t=+t=(rscr2);
скрф[alu=0];
error;

*t=+0;
scr2=B14;
* RSCR2 ← PREDICTED RESULT
*t=lc[y[t.r1,1]];
t=+t=(rscr2);
скрф[alu=0];
error;
*t=+rm1;
scr2=NB14;
* RSCR2 ← PREDICTED RESULT
*t=lc[y[t.rm2,1]];
t=+t=(rscr2);
скрф[alu=0];
error;

TRlcy2:

```
t=+0;
scr2=B13;
* RSCR2 ← PREDICTED RESULT
*t=lc[y[t.r1,2]];
t=+t=(rscr2);
скрф[alu=0];
error;
*t=+rm1;
scr2=NB13;
* RSCR2 ← PREDICTED RESULT
*t=lc[y[t.rm2,2]];
t=+t=(rscr2);
скрф[alu=0];
error;

```
t=+0;
scr2=B12;
* RSCR2 ← PREDICTED RESULT
*t=lc[y[t.r1,3]];
t=+t=(rscr2);
скрф[alu=0];
error;
*t=+rm1;
scr2=NB12;
* RSCR2 ← PREDICTED RESULT
*t=lc[y[t.rm2,3]];
t=+t=(rscr2);
скрф[alu=0];
error;

TRlcy4:

```
t=+0;
scr2=B11;
* RSCR2 ← PREDICTED RESULT
*t=lc[y[t.r1,4]];
t=+t=(rscr2);
скрф[alu=0];
error;
*t=+rm1;
scr2=NB11;
* RSCR2 ← PREDICTED RESULT
```
t+=lc[y[t, rm2, 4]];
  t+=#(rscr2);
  skpif[alu=0];
  error;

  t+r0;
  rscr2=B10;  \* RSCR2 + PREDICTED RESULT
  t+=lc[y[t, r1, 5]];
  t+=#(rscr2);
  skpif[alu=0];
  error;
  rscr2=NB10;  \* RSCR2 + PREDICTED RESULT
  t+=lc[y[t, rm2, 5]];
  t+=#(rscr2);
  skpif[alu=0];
  error;

  TRlcyl0:
  t=r0;
  rscr2=B9;  \* RSCR2 + PREDICTED RESULT
  t+=lc[y[t, r1, 6]];
  t+=#(rscr2);
  skpif[alu=0];
  error;
  rscr2=NB9;  \* RSCR2 + PREDICTED RESULT
  t+=lc[y[t, rm2, 6]];
  t+=#(rscr2);
  skpif[alu=0];
  error;

  t+r0;
  rscr2=B6;  \* RSCR2 + PREDICTED RESULT
  t+=lc[y[t, r1, 7]];
  t+=#(rscr2);
  skpif[alu=0];
  error;
  rscr2=NB6;  \* RSCR2 + PREDICTED RESULT
  t+=lc[y[t, rm2, 7]];
  t+=#(rscr2);
  skpif[alu=0];
  error;

  TRlcyl8:
  t=r0;
  rscr2=B7;  \* RSCR2 + PREDICTED RESULT
  t+=lc[y[t, r1, 10]];
  t+=#(rscr2);
  skpif[alu=0];
  error;
  rscr2=NB7;  \* RSCR2 + PREDICTED RESULT
  t+=lc[y[t, rm2, 10]];
  t+=#(rscr2);
  skpif[alu=0];
  error;

  t+r0;
  rscr2=B6;  \* RSCR2 + PREDICTED RESULT
  t+=lc[y[t, r1, 11]];
  t+=#(rscr2);
  skpif[alu=0];
  error;
  rscr2=NB6;  \* RSCR2 + PREDICTED RESULT
  t+=lc[y[t, rm2, 11]];
  t+=#(rscr2);
  skpif[alu=0];
  error;

  TRlcyl0:
  t=r0;
  rscr2=B5;  \* RSCR2 + PREDICTED RESULT
  t+=lc[y[t, r1, 12]];
t+=t#(rscr2);
skp1f[alu=0];  
error;
t+=rml;
 rscr2=NB5;  
* RSCR2 + PREDICTED RESULT
 t+=lcy[t, rm2, 12];
t+=t#(rscr2);
skp1f[alu=0];  
error;

 t+=r0;
 rscr2=BA4;  
* RSCR2 + PREDICTED RESULT
 t+=lcy[t, rl, 13];
t+=t#(rscr2);
skp1f[alu=0];  
error;
t+=rml;
 rscr2=NB4;  
* RSCR2 + PREDICTED RESULT
 t+=lcy[t, rm2, 13];
t+=t#(rscr2);
skp1f[alu=0];  
error;

 TRlcyl2:
 t+=r0;
 rscr2=BA3;  
* RSCR2 + PREDICTED RESULT
 t+=lcy[t, rl, 14];
t+=t#(rscr2);
skp1f[alu=0];  
error;
t+=rml;
 rscr2=NB3;  
* RSCR2 + PREDICTED RESULT
 t+=lcy[t, rm2, 14];
t+=t#(rscr2);
skp1f[alu=0];  
error;

 t+=r0;
 rscr2=BA2;  
* RSCR2 + PREDICTED RESULT
 t+=lcy[t, rl, 15];
t+=t#(rscr2);
skp1f[alu=0];  
error;
t+=rml;
 rscr2=NB2;  
* RSCR2 + PREDICTED RESULT
 t+=lcy[t, rm2, 15];
t+=t#(rscr2);
skp1f[alu=0];  
error;

 TRlcyl4:
 t+=r0;
 rscr2=BA1;  
* RSCR2 + PREDICTED RESULT
 t+=lcy[t, rl, 16];
t+=t#(rscr2);
skp1f[alu=0];  
error;
t+=rml;
 rscr2=NB1;  
* RSCR2 + PREDICTED RESULT
 t+=lcy[t, rm2, 16];
t+=t#(rscr2);
skp1f[alu=0];  
error;

 t+=r0;
 rscr2=BA0;  
* RSCR2 + PREDICTED RESULT
 t+=lcy[t, rl, 17];
t+=t#(rscr2);
skp1f[alu=0];  
error;
t+=rml;
 rscr2=NB0;  
* RSCR2 + PREDICTED RESULT
 t+=lcy[t, rm2, 17];
t+=t#(rscr2);
skp1f[alu=0];  
error;
RT1cyTest:

* RSCR2 HOLDS THE PREDICTED RESULT, T HOLDS ACTUAL RESULT

\[
\begin{align*}
t & = r1; \\
\text{t} & = r1; \\
\text{t} & = r1; \\
\text{t} & = r1; \\
\text{t} & = r1; \\
\text{t} & = r1; \\
\text{t} & = r1; \\
\text{t} & = r1;
\end{align*}
\]
skpif[alu=0];
error;
t=r1;
sr2+NB10;
  rscr2 = PREDICTED RESULT
  t=lcy[r1,t,5];
t=t#(rscr2);
skpif[alu=0];
error;
t=r1;
sr2+NB10;
  rscr2 = PREDICTED RESULT
  t=lcy[r1,t,6];
t=t#(rscr2);
skpif[alu=0];
error;
t=r1;
sr2+NB10;
  rscr2 = PREDICTED RESULT
  t=lcy[r1,t,7];
t=t#(rscr2);
skpif[alu=0];
error;

RTlcy8:

  rscr2 = PREDICTED RESULT
  t=r1;
sr2+NB8;
  t=lcy[r0,t,6];
t=t#(rscr2);
skpif[alu=0];
error;
t=r1;
sr2+NB8;
  t=lcy[r0,t,7];
t=t#(rscr2);
skpif[alu=0];
error;

RTlcy8:

  rscr2 = PREDICTED RESULT
  t=r1;
sr2+NB7;
  t=lcy[r0,t,10];
t=t#(rscr2);
skpif[alu=0];
error;
t=r1;
sr2+NB7;
  t=lcy[r0,t,11];
t=t#(rscr2);
skpif[alu=0];
error;

RTlcy10:

  rscr2 = PREDICTED RESULT
  t=r1;
sr2+NB5;
  t=lcy[r0,t,12];
t=t#(rscr2);
skpif[alu=0];
error;
t=r1;
sr2+NB5;
  t=lcy[r0,t,13];
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;

R1cy12:
  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;

R1cy14:
  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;

R1cyDone:
  r01 = NOT(r10);  * REDEFINE r01 !!!!!!!
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);
          skp1f(ALU=0);
rcy16Err1:  * r10 rcy 1 should be r01 error;
          t := r01;
          t := rcy[t, t, 1];  * try it again from t
          t # (r10);
          skp1f(ALU=0);
rcy16Err2:  * r10 rcy 1 should be r01. (done from error;  * t this time)
          t := rcy[r1, r1, 1];
          t # (rhigh1);
          skp1f(ALU=0);
rcy16Err3:  * 1 rcy 1 should be 1000000B error;
          t := r1;
          t := rcy[t, t, 1];
          t # (rhigh1);
          skp1f(ALU=0);
rcy16Err4:  * 1 rcy 1 should be 1000000B. (done from t error;  * time).
          t := rcy[r10, r10, 1];
          t # (r01);
          skp1f(ALU=0);
rcy16Err5:  * r10 rcy 1 should be r01 error;
          t := r10;
          t := rcy[t, t, 1];  * t := r10 rcy 1
          t # (r01);
          skp1f(ALU=0);
rcy16Err6:  * r10 rcy 1 should be r01 error;  * done from t this time.
          t := rcy[r01, r01, 2];
          t # (r01);
          skp1f(ALU=0);
rcy16Err7:  * r01 rcy 2 should be r01 error;
          t := rcy[r01, r01, 3];
          t # (r10);
          skp1f(ALU=0);
rcy16Err8:  * r01 rcy 3 should be r10 error;
          t := rcy[r01, r01, 10];
          t # (r01);
          skp1f(ALU=0);
rcy16Err9:  * r01 rcy 10 should be r01 error;

lcyTest:  lcy[r01, r01, 1];
          t # (r10);
          skp1f(ALU=0);
lcy16Err1:  * r01 lcy 1 should be r10 error;
          t := lcy[rhigh1, rhigh1, 1];
          t # (r1);
          skp1f(ALU=0);
lcy16Err2:  * 1000000B lcyd 1 should be 1 error;
t + lcy[r1, r1, 1];
t#(2c);
skpif[ALU=0];
lcy16Err3: * 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];
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;
cycleTest Test the cycle machinery by generating all possible values for the r, t, and co
**unt 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 = shcVal AND 40B;
    Shc.b = shcVal AND 20B;
    Shc.count = shcVals AND 17B;
    r = getPattern[pats];
    t = IF numberOfZeroBitsGr1[r] THEN -1 ELSE 0;
    result = doShift[];
  end of shcVals loop
ENDDO;
  -- end of pats loop
BEGIN
  tCycle: CARDINAL = 60B; -- 2 highest bits in shcVal are 1
  rCycle: CARDINAL = 0; -- 2 highest bits in shcVal are 0
  trtCycle: CARDINAL = 40B; -- highest bit in shcVal is 1
  rtCycle: CARDINAL = 20B; -- 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;
  savedVa1ue = left AND saveMask;
  right = LeftShift[right, shiftCount];
  savedVa1ue = 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
    IGNORE: TYPE = [0..377B]
    POS: TYPE = [0..17B]
    SIZE: TYPE = [0..17B]
]

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 ~ BITS [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

(AActually 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[RO1]];
r4BitMsk ~ 17C; * RENAME RO1 AS r4BitMsk !!!
RM[lastShC, IP[RSCR]]: * RENAME RSCR AS lastShC

RFWFtest:
Q~RO;
* 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
  skip[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

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

`t=q;
WF=t;
lastShC=SHC;

* CHECK LMASK: COMPUTE pos
rscr2 + o;
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 + o;
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);

WFSHFTC:  * 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 !!!!!!
# 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=7777C;
  t=t#(rsCr);
  skpIf[ALU=0];
  error:  * -1 RSH[1] SHOULD BE 7777B
  t=(PD+r10)rsh 1;
  t=t#(r01);
  skpIf[ALU=0];
  error:  * (ALTERNATING 10) rsh 1 SHOULD BE (ALT. 01)
```

* TEST RESULT + ALU RCV[1] (RESULT[0]+ALU[15])

```
aluRCV:
  t=(PD+rm1)rcy 1;
  t=t#(rm1);
  skpIf[ALU=0];
  error:  * -1 RCV[1] SHOULD BE -1
  t=(PD+r0)rcy 1;
  t=t#(r0);
  skpIf[ALU=0];
  error:  * 0 RCV[1] SHOULD BE 0
  t=(PD+r10)rcy 1;
  t=t#(r01);
  skpIf[ALU=0];
  error:  * (ALTERNATING 10) RCV[1] SHOULD BE (ALT 01)
  t=(PD+r01)rcy 1;
  t=t#(r0);
  skpIf[ALU=0];
  error:  * (ALT 01) RCV[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;
rsocr(r1)=(r1);
t=t#(rsocr);
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;
rsocr=CM2;
t=t#(rsocr);
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;
rsocr=(r1)+(r1);
t=t#(rsocr);
skpif[ALU=0];
error;  * 1 LCY SHOULD BE 2
EXHAUSTIVE TEST OF ALU SHIFT FUNCTIONS

FOR Q IN[0...177777] DO
  rsr2 = Q rsh 1;
  t = predictedRSH[I];
  IF t XOR rsr2 THEN ERROR;

  rsr2 = Q rcy 1;
  t = predictedRCY[I];
  IF t XOR rsr2 THEN ERROR;

  rsr2 = Q Arsh 1;
  t = predictedARSH[I];
  IF t XOR rsr2 THEN ERROR;

  rsr2 = Q lsh 1;
  t = predictedLSH[I];
  IF t XOR rsr2 THEN ERROR;

  rsr2 = Q rsh 1;
  t = predictedRSH[I];
  IF t XOR rsr2 THEN ERROR;

  rsr2 = Q lcy 1;
  t = predictedLCY[I];
  IF t XOR rsr2 THEN ERROR;

ENDLOOP;

% aluSHTEST:  * USE Q AS LOOP VARIABLE
aluSHL:  * TOP OF LOOP

* RSH TEST
  rsr = Q;
  t = (PD<--rsr)rsh 1;
  rsr = rsh[rsr,1];
  t = #rsc1; 
  branch[.+2,ALU=0];

RSHER:  error;  * PREDICTED RESULT DIFFERENT FROM REAL ONE

* RCY TEST
  rsr = Q;  * Q IS LOOP VARIABLE
  t = (PD<--rsr)rcy 1;
  rsr2 = t;  * REAL RESULT IN rsr2
  t = rsh[rsr,1];
  skpif[R EVEN], (PD<--rsr);  * ADD HIGH BIT IF NECESSARY
  t = t+rh1;  * PREDICTED RESULT IN T
  t = #rsc2;
  branch[.+2,ALU=0];

RCYER:  error;  * T = (PREDICTED T) XOR rsr2

* ARSH TEST
  rsr = Q;  * Q IS LOOP VARIABLE
  t = (PD<--rsr)Arsh 1;
  rsr2 = t;  * rsr2 = ACTUAL RESULT
  t = rsh[rsr,1];
  PD = Q;  * ADD SIGN BIT IF REQUIRED
  skpunless[ALU<0];  * ADD SIGN BIT IF REQUIRED
  t = t+rh1;
  t = #rsc2;
  skpif[ALU=0];

ARSHER:  error;  * t = (PREDICTED T) XOR rsr2

* LSH TEST
  rsr = Q;  * Q IS LOOP VARIABLE
  rsr2 = (PD<Q)lsh 1;  * rsr2 = ACTUAL RESULT
  t = lsh[rsr,1];
t=t#(rscr2);
skipif[ALU=0];

LSHER:
  error;  * t= (PREDICTED T) XOR rscr2

* LCY TEST
  t=rscr + 0;  * Q IS LOOP VARIABLE
  rscr2 + (PD-t)cy 1;  * rscr2 = ACTUAL RESULT
  t=lsh[rscr,1];  * t= PREDICTED RESULT
  rscr=(rscr);
  skipunless[ALU<0];
  t=t+(r1);  * t= T + 1 FOR CYCLED BIT 0
  t=t#(rscr2);
  skipif[ALU=0];

LCYER:
  error;  * T + (PREDICTED T) XOR rscr2
  t+(r1)+t(Q);
  dblBranch[+1,aluSHL,ALU=0],Q+t;
goto[afterkernel13];
Fix bug in computation of correct value for stkp after performing stack+1-.

Add bypass checking to stack test.

Add call to checkTaskNum at beginKernel14 to skip the stack tests when we're not executing in ta

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 preceding 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

beginKernel14:
  call[checkTaskNum], t ← r0;
skpif[ALU=0];
  branch[stkXitTopL];  * don't try task 0 tests.
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+[stkp] ~ val: increment stkp by one, then place val into stack[stkp]
- The strategy 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).

FOR top2StkpBits IN [0 .. 3] DO
  FOR index IN [0 .. stkpMax] DO -- check simple loading of stkp and stk
    i ~ index + LeftShift[top2StkpBits, 6];
    stk ~ i;
    IF POINTERS[].stkError THEN ERROR; -- underflow or overflow
    IF POINTERS[].stk # i THEN ERROR; -- stk not the value we loaded
    stk[stkp] ~ 0;
    IF stk[stkp] # 0 THEN ERROR; -- loaded zero, got back something different.
    stk[stkp] ~ -1;
    IF stk[stkp] # -1 THEN ERROR; -- loaded -1, got back something different.
    stk[stkp] ~ Alternating01;
    IF stk[stkp] # Alternating01 THEN ERROR; -- loaded Alternating01, got back something different.
    stk[stkp] ~ Alternating10;
    IF stk[stkp] # Alternating10 THEN ERROR; -- loaded Alternating10, got back something different.

    stk[stkp] ~ i; -- init current stack to stk[i] ~ i
    IF stk[stkp] # i THEN ERROR;
  ENDLOOP:

FOR i IN [0 .. STKPMAX] DO -- test other operations within that stack
  i ~ index + LeftShift[top2StkpBits, 6];
  stk ~ i;
  IF i # STKPMAX THEN -- begin +, -1 operations
    BEGIN -- begin w/ operations that use current stk for loading rm
      stack&+1[stkp] ~ -1;
      IF POINTERS[].stack # i+1 THEN ERROR; -- auto increment of stk failed
      IF stack[stkp] # i+1 THEN ERROR; -- got wrong data after increment
      stack&-1[stkp] ~ i+1;
      IF stack[stkp] # i THEN ERROR; -- rewrite current data, decrement stkp
      stack+[stkp] ~ -1;
      IF stack[stkp] # -1 THEN ERROR; -- auto decrement of stk failed
      IF stack[stkp] # 0 THEN ERROR; -- got wrong data after decrement
      stack[stkp] ~ i+1; -- set stk to check data
      IF stack[stkp] # i+1 THEN ERROR; -- wrong data during auto decrement
      IF i+1 < stkpMax THEN -- check +, -2 operations
        BEGIN -- begin w/ operations that use current stk for loading rm
          stack&+2[stkp] ~ -1;
          IF POINTERS[].stack # i+2 THEN ERROR; -- auto increment of stk failed
          IF stack[stkp] # i+2 THEN ERROR; -- got wrong data after increment
          stack&-2[stkp] ~ i+2;
          IF stack[stkp] # i THEN ERROR; -- rewrite current data, decrement stkp
          stack-1[stkp] ~ i;
          IF stack[stkp] # -1 THEN ERROR; -- auto decrement of stk failed
          IF stack[stkp] # 0 THEN ERROR; -- got wrong data after decrement
          stack[i] ~ i; -- reset data at "current" stk
          -- now test operations that modify stk before loading rm
          stack[stkp] ~ i+1;
          IF stack[stkp] # i THEN ERROR; -- auto increment failed
          stack[stkp] ~ i;
          IF stack[stkp] # 0 THEN ERROR; -- bypass error
          stack[stkp] ~ i+1;
          IF stack[stkp] # 0 THEN ERROR; -- bypass error
          stack[stkp] ~ i;
          -- reset data at "current" stk
          END;
        END;
      END;
    END;
  END;
  IF i+1 < stkpMax THEN -- check +, -2 operations
    BEGIN -- begin w/ operations that use current stk for loading rm
      stack&+2[stkp] ~ -1;
      IF POINTERS[].stack # i+2 THEN ERROR; -- auto increment of stk failed
      IF stack[stkp] # i+2 THEN ERROR; -- got wrong data after increment
      stack&-2[stkp] ~ i+2;
      IF stack[stkp] # i THEN ERROR; -- rewrite current data, decrement stkp
      stack-1[stkp] ~ i;
      IF stack[stkp] # -1 THEN ERROR; -- auto decrement of stk failed
      IF stack[stkp] # 0 THEN ERROR; -- got wrong data
      stack[i] ~ i; -- reset data at current stk
      stk ~ i+2;
      -- set stk to check data
      IF stack[stkp] # i+2 THEN ERROR; -- wrong data during auto decrement
stkp + i;  -- reset stkp

-- now test operations that modify stkp before loading rm
stack+2[stkp] += -1;  -- increment stkp, then load rm
IF POINTERS[].stack # i+2 THEN ERROR;  -- stkp auto increment failed
IF stack[stkp] # -1 THEN ERROR;  -- didn't get data we wrote
stack[stkp] += i+2;  -- fix clobbered location
stack-2[stkp] += -1;  -- decrement stkp, then load rm
IF POINTERS[].stack # i THEN ERROR;  -- stkp auto decrement failed
IF stack[stkp] # -1 THEN ERROR;  -- didn't get data we wrote
stack[stkp] += 1;  -- fix clobbered location
END:

IF i+2 < stkpMAX THEN  -- check +,- 3 operations
BEGIN  -- begin w/ operations that use current stkp for loading rm
stack+3[stkp] += -1;  -- increment stkp, then load rm
IF POINTERS[].stack # i+3 THEN ERROR;  -- stkp auto increment failed
IF stack[stkp] # i+3 THEN ERROR;  -- stkp auto increment failed
stack&-2[stkp] += i+2;  -- rewrite current data, decrement stkp
IF POINTERS[].stack # i THEN ERROR;  -- auto decrement of stkp failed
IF stack[stkp] # -1 THEN ERROR;  -- got wrong data
stack[stkp] += i+3;  -- set stkp to check data
stack[stkp] += 1;  -- reset data at current stkp
END;  -- end of top2StkpBits loop

ENDLOOP;  -- end of top2StkpBits loop
stkTest Test the various stack operations

stkTest:
* initialize the top2bits loop
   call[iTopStkBits];

stkTopL:
* top of "top 2 bits of stk" loop
   call[nextTopStkBits];
   skpif[ALU#0];
   branch[stkXitTopL];
   noop;

stkIL:
   call[nextStkPAddr];
   * initialize stack index [1..maxStkXC]
   call[iStkPAddr];
   * initialize stack index [l..maxStkXC]
   skpif[ALU#0];
   branch[stkXit];
   skpif[ALU=0];

stkIErr0:
* got stack underflow or overflow
   error;
   call[getNextStkAddr]. rscr + t;
   t # (rscr);  * compare real stkP with value we loaded
   skpif[ALU=0];

stkIErr1:
* t = tskP. rscr = value we loaded
   error;

stkIErr2:
* wrote zero, got back something else
   error;  * Q = value from stack
   t = rl;
   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 = rsr;
   skpif[ALU=0];

stkIErr6:
* wrote stkP from rscr. Q = value from stack
   error;  * read it into t. they aren't the same
   branch[stkIL];

stkXit:
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 + 1
   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&+1Err0:  * got stack underflow or overflow
* got stack underflow or overflow
   error;
   call[getRealStkAddr], rscr=t+1;  * compare stackPAddr from Pointers w/ expected valu
   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 valu, rscr = stack's valu from Pointers
   t = rscr;
   * restore t
* 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;  * compare expected stkP (rscr) with
   t # (rscr):  * actual stkP (t)
   skpif[alu=0];
stkM1AddrErr:  * auto decrement failed
   error;
   t = cm1;
   t = t # (Q=stack)
   skpif[alu=0];  * see if original stack&+1 = cm1 worked
stkM1ValErr:  * 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
   stkPErr13:  * got stack underflow or overflow
   error;
rscr + t = t+1;
* check the data modified during "stack-&-1" instruction
stk = t;   
* compare tos with expected valu
stkMValErr: Q = value from stack
* t = bad bits, rscr = expected value
t = rscr + (rscr) + t=1;  
* t, rscr = "i"
stkp = t;   
* stkp is at i+1 now. Fix it.
call[chkStkErr];
skpif[ALU=0];
stkpErr1:  
* 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;   
* reset stack
* stack+1 stack+1 stack+1 stack+1 stack+1 stack+1 stack+1 stack+1 stack+1
rscr = (rscr) + t=1;   
* compute expected stkp value
t = cm1;
stack+1 = t;   
* stkp + i+1, stack[stkp] = -1
call[chkStkErr];
skpif[ALU=0];
stkpErr2:  
* got stack underflow or overflow
error;
call[getRealStkAddr];
(rscr) # t;  
skpif[ALU=0];
stkPAddrErr2:  
* expected Rscr, got stkp 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
stkMValErr3:  
* t = bad bits
t = rscr;  
* restore t
* 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];
stkpErr3:  
* 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];
stkpErr4:  
* got stack underflow or overflow
error;
call[getRealStkAddr];
(rscr) # t;  
* see if real stkp (t) matches expected stkp (rscr)
skpif[ALU=0];
stkMAddrErr2:
error;
t ← cm1;
t ← t # (Q=stack);  * compare tos with -1
skpif[ALU=0];
stkM1ValErr2:  * Q = value from stack
t ← rscr;  * restore t
  error;  * t = bad bits, expected -1
  stkErr1:  * got stack underflow or overflow
  error;  * for placement
  afterStkTest1:

stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
  stkM1ValErr2:
* November 30, 1978 6:07 PM

%remember, don't 'execute if i=1, if i+2=77%
\begin{verbatim}
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];
\end{verbatim}

\begin{verbatim}
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 i, or this is first time
error;  * thru, and the initialization didn't work properly.
* Q=value from stack
\end{verbatim}

* 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&P2Err0;  * got stack underflow or overflow
error;
\begin{verbatim}
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
\end{verbatim}

* 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;
\begin{verbatim}
call[getRealStkAddr], rscr + t+(2c);
t # (rscr);
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
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}

* stack+2 stack+2 stack+2 stack+2 stack+2 stack+2 stack+2 stack+2
stack+2 + (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];
stkpErr28: * 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 + t;  * reset stack which was clobbered by "stack+2~cm1"
   call[chkStkErr];
   skpif[ALU=0];
stkpErr28: * 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];
stkpM2AddrErr2: error;
   t = cm1;
   t = t # (Q~stack);  * compare tos with -1
   skpif[ALU=0];
stkpM2ValErr2: * 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%
  * placement for afterStkTest2 branch
  * isolate the index (exclude top 2 bits)
  * skip this test if it cause overflow
  * the stack entry at location in t. or this is first time

  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];
    stk&+3Err0: * got stack underflow or overflow
    error;
    call[getRealStkAddr], rscr+t+(3c); * compare stackPAddr from Pointers w/ expected val
    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
    stkM3AddrErr: * auto decrement failed
    error;
    t + cm1;
    t + t # (Q=stack);
    stkpErr32: * 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
    rscr + t; * restore t
    stkpErr33: * stack underflow or overflow
    error;
    stkP3AddrErr3: * stack underflow or overflow
    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 t + t; * stack["i+3"] + i+3, stackP + i.
      call[chkStkErr];
      stkM3AddrErr: * compare expected stkP (rscr) with
      skpif[ALU=0]; * actual stkP (t)
      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
      stkpErr32: * got stack underflow or overflow
      error;
      call[getRealStkAddr], rscr-t+(3c); * compare stkPAddr from Pointers w/ expected val
      stkP3AddrErr: * auto increment of StackP failed. rscr = expected value,
      error; * t = value from Pointers
      rscr + t + t+(3C);
      stkP3ValErr2: * stack&+3 seems to have clobbered the
      error; * (i+3)th value. t = bad bits. Q = value from stack
      stkpErr33: * got stack underflow or overflow
      error; * t = bad bits, rscr = expected value
      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
        stkp + t; * stkp is at i+3 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;
    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%

```c
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];

```

```c
 t # (0==stack);  /* see if stack[stkP] = stkP */
skp[if[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. */
```

```c
Q=value from stack
```

* Simulate stack&+4 -- hardware can perform stack&+3 as maximum increment

```c
stack&+3 = cm1;  /* stack[stackP] = -1, then stackP = stackP+4 */
stkP+1;  /* simulate +4 */
call[chkStkErr];
skp[if[ALU=0];
stk&+4Err0:  /* got stack underflow or overflow */
error;
```

```c
call[getRealStkAddr], rscr+t+(4c);  /* compare stackPAddr from Pointers w/ expected val */
t # (rscr);
skp[if[ALU=0];
stkP4AddrErr:  /* auto increment of StackP failed. rscr = expected value, */
error;  /* t = value from Pointers */
t += # (0==stack);
skp[if[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

```c
stack&-4 = t;  /* stack["i+4"] = i+4, stackP = i. */
call[chkStkErr];
skp[if[ALU=0];
stkP4Err2:  /* got stack underflow or overflow */
error;
```

```c
call[getRealStkAddr], rscr+t-(4c);  /* compare expected stkP (rscr) with */
t # (rscr);  /* actual stkP (t) */
stkM4AddrErr:  /* auto decrement failed */
error;
```

```c
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

```c
(t += rscr-(rscr)-(4c);  /* t, rscr + "i" */
```
stkP + t;  * stkP is at i+4c now. Fix it.

call[chkStkErr];
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];
    stkPErr45:  * got stack underflow or overflow
        error;
        call[getRealStkAddr];
            (rscr) # t;
            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
                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];
    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];
        stkPErr47:  * got stack underflow or overflow
            error;
            call[getRealStkAddr];
                (rscr) # t;  * see if real stkP (t) matches expected stkP (rscr)
                stkP4AddrErr2:  * stkP matches expected stkP (rscr)
                    error;
                    t + cm1;
                    t + t # (Q+stack);  * compare tos with -1
                    stkP4ValErr2:  * 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];
        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 indices are 6 bits long.
    * Return (stackPAddr OR stackPtopBits) in T. It's an 8 bit address.
    * ALU0 => valid value.

    klink + link;
    t = stackPAddr + (stackPAddr) + 1;    * increment the index
    t and (77c);    * check for 6 bit overflow
    skip[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;
    t = stackPtopBits + (stackPtopBits)+(100c);
    t - (400c);
    skip[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 = Pointsers[];
    return, rscr2 + (rscr2) AND (pointers.stkErr);
    top level;

stkXitTopL:
TEST CARRY20 FUNCTION

This function causes a 1 to 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);
  skp[f(ALU=0)];
  error; * T NE 17B + 0 + CARRY20

  t=rscr;
  t=t+(r1),CARRY20; * t=17B+1+CARRY20
  rscr2=20C;
  t=t#(rscr2);
  skp[f(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);
  skp[f(ALU=0)];
  error; * T NE 20B=0+0+CARRY20

  t=r0;
  t=t-(rscr2); * t=20B=(0-20B)
  t=t+(r0),CARRY20;
  skp[f(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];
  xorCarrey:
  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!
TEST USAVEDCARRY

This function causes the alu carry bit from the last instruction to be used as the carry in 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),USAVEDCARRY;  * T=O+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),USAVEDCARRY;  * 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),USAVEDCARRY;  * 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),USAVEDCARRY;  * 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),USAVEDCARRY;  * 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:

\[ H[0:15] = H[0:14] + CARRY, ALU[0:14] \]

These tests invoke mulCheck, a subroutine that performs two services:

1) \( T=\{R+\text{SCR},\text{MULTIPLY}\} \)
2) \( \text{SCR}=1 \) IF Q[14] BRANCH IS TAKEN (ZERO OTHERWISE)

ERRORS are numbered 1 thru 3:
- \( \text{mulXerr1} \rightarrow T \) value wrong (H3)
- \( \text{mulXerr2} \rightarrow Q[14] \) branch wrong
- \( \text{mulXerr3} \rightarrow Q \) value wrong

% multiplyTest:

* Q[14]=0, CARRY=0, ALU15=0
  t=Q-r0;  
  rscfr=t;  
  call[mulCheck], * t+rscfr,MULTIPLY==ALU+0, CARRY=0  
  skipf[R EVEN], rscfr2=rscfr2;  

mulAerr1:
  error; * TOOK Q[14]=1 BRANCH  
  t=t;  
  skipf[ALU=0];  

mulAerr2:
  error; * CARRY,(0+0)/2 SHOULD BE ZERO  
  t=Q;  
  skipf[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;  
  rscfr=t;  
  call[mulCheck], t=rml; * t+rscfr,MULTIPLY==ALU=1, CARRY=0  
  skipf[R EVEN], rscfr2=rscfr2;  

mulBerr1:
  error; * TOOK Q[14]=1 BRANCH  
  t=t#(rhigh1); *-1+1 GENERATES CARRY BIT  
  skipf[ALU=0];  

mulBerr2:
  error; * CARRY,(0+0)/2 SHOULD BE 100000  
  t=Q; *-1+1 WOULD LEAVE ALU15=0  
  skipf[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  
  Q=r0;  
  rscfr=t;  
  call[mulCheck], * t+rscfr,MULTIPLY==ALU=1, CARRY=0  
  skipf[R EVEN], rscfr2=rscfr2;  

mulCerr1:
  error; * TOOK Q[14]=1 BRANCH  
  t=t; * 0+1== CARRY<0  
  skipf[ALU=0];  

mulCerr2:
  error; * CARRY,(0+1)/2 SHOULD BE 0  
  t=(rhigh1)#(Q); * 0+1 WOULD LEAVE ALU15=1  
  skipf[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+rsc=rhigh1;
call[mulCheck],t+t+(r1);
  * t+t+rsc,MULTIPLY=>ALU=1, CARRY=1
  skrip[R EVEN], rsc2=rsc2;
mulDerr1:
  error; * TOOK Q[14]=1 BRANCH
  t+t(#(rhigh1));
  * 1000001+100000=> CARRY+1
  skrip[ALU=0];
mulDerr2:
  error; * CARRY,,(1000001+100000)/2 SHOULD BE 100000
  t+(rhigh1)#(Q); * 1000001+100000 WOULD LEAVE ALU15=1
  skrip[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;
  rsc=r;
call[mulCheck], * t+t+rsc,MULTIPLY=>ALU=0, CARRY=0
  skrip[R ODD], rsc2=rsc2;
mulEerr1:
  error; * DIDN'T TAKE Q[14]=1 BRANCH
  t+t;
  skrip[ALU=0];
mulEerr2:
  error; * CARRY,,(0+0)/2 SHOULD BE ZERO
  t+(r1)#(Q);
  skrip[ALU=0];
mulEerr3:
  error; * ALU15,,Q[0:14] SHOULD BE 1
multiplyF:
  * Q[14]=1, CARRY=1, ALU15=0
  t+(r1)+(r1);
  Q=t;
  * Q[14]=1
  rsc=r;
call[mulCheck], * t+t+rsc,MULTIPLY=>ALU=1, CARRY=1
  skrip[R ODD], rsc2=rsc2;
mulFerr1:
  error; * DIDN'T TAKE Q[14]=1 BRANCH
  t+t(#(rhigh1));
  *-1+1 GENERATES CARRY BIT
  skrip[ALU=0];
mulFerr2:
  error; * CARRY,,(0+0)/2 SHOULD BE 100000
  t+(r1)#(Q); *-1+1 WOULD LEAVE ALU15=0
  skrip[ALU=0];
mulFerr3:
  error; * ALU15,,Q[0:14] SHOULD BE 1
multiplyG:
  * Q[14]=1, CARRY=0, ALU15=1
  t+(r1)+(r1);
  Q=t;
  * Q[14]=1
  t=r0;
  rsc=r;
call[mulCheck], * t+t+rsc,MULTIPLY=>ALU=1, CARRY=0
  skrip[R ODD], rsc2=rsc2;
mulGerr1:
  error; * DIDN'T TAKE Q[14]=1 BRANCH
  t+t;
  * 0+1==> CARRY=0
  skrip[ALU=0];
mulGerr2:
  error; * CARRY,,(0+1)/2 SHOULD BE 0
t=(rhigh1)+1;  * 0+1 WOULD LEAVE ALU15=1
  t=t#(Q);
  skpip[ALU=0];
mulGerr3:
  error;  * ALU15,.Q[0:14] SHOULD BE 100001

*  Q[14]=1, CARRY=1, ALU15=1

multiplyH:  *  Q[0]=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
  skpip[R ODD]. rscr2=rscr2;
mulHerr1:
  error;  * DIDN'T TAKE Q[14]=1 BRANCH
  t=t#(rhigh1);  * 1000001+100000=> CARRY=1
  skpip[ALU=0];

mulHerr2:
  error;  * CARRY,.(1000001+100000)/2 SHOULD BE 100000
  t=(rhigh1)+1;  * 1000001+100000 WOULD LEAVE ALU15=1
  t=t#(Q);
  skpip[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
  skpip[R EVEN]. rscr2=rscr2;
mulJerr1:
  error;  * TOOK Q[14]=1 BRANCH
  t=t;
  skpip[ALU=0];
mulJerr2:
  error;  * CARRY,.(0+0)/2 SHOULD BE ZERO
  t=(r01) RSH 1;
  t=t#(Q);
  skpip[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
  skpip[R ODD]. rscr2=rscr2;
mulKerr1:
  error;  * DIDN'T TAKE Q[14]=1 BRANCH
  t=t;
  skpip[ALU=0];
mulKerr2:
  error;  * CARRY,.(0+0)/2 SHOULD BE ZERO
  t=(r10) RSH 1;
  t=t#(Q);
  skpip[ALU=0];
mulKerr3:
  error;  * ALU15,.Q[0:14] SHOULD BE (r10) RSH 1

mulDone: BRANCH[afterMul];
This subroutine performs, 
\( t = t + (r_{scr}) \), MULTIPLY;

It sets \( r_{scr2} = 0 \) if \( Q[14] \) branch DID NOT HAPPEN.
It sets \( r_{scr2} = 1 \) if \( Q[14] \) branch DID HAPPEN.

\( t \) and \( r_{scr2} \) are Clobbered! It assumes \( r_0 = 0, r_1 = 1 \).

```
% SUBROUTINE;
mulCheck:
  \( t = t + (r_{scr}) \), MULTIPLY, GLOBAL, AT[700];
  GOTO[.+1];
  r_{scr2} = r_0, AT[701];
  RETURN;
  r_{scr2} = r_1, AT[703];
  RETURN;
TOP LEVEL;

afterMul:
```
* 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

divTest:
  * Q0=0, CARRY=0
  Q=r0;
t=(r1)+(r1),DIVIDE;  *t=1+1,DIVIDE==) CARRY=0,
t=t+(Q); 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=rhl:
  t=(r1);  * T = 100001
t=(rhl),DIVIDE;  * t=100001+100000,DIVIDE==)ALU=1, CARRY=1
t=t+(Q); 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=rhl:
  t=(r1)+(r1),DIVIDE;  * t=1+1,DIVIDE==)ALU=2, CARRY=0
  t=t+(Q); skpIf[ALU=0]; error:  * 2*(2)..Q[0]=5
  t=(Q);
skpIf[ALU=0]; error:  * Q[1:15],..CARRY SHOULD BE 0

divD:
  * Q0=1, CARRY=1
  Q=rhl:
  t=(r1);  * SET Q[0] TO ONE
  t=t+(rhl),DIVIDE;  * t=100001+100000,DIVIDE==)ALU=1, CARRY=1
  t=t+(Q); skpIf[ALU=0]; error:  * 2*(1)..Q[0]=3
  t=(r1)+Q;
skpIf[ALU=0]; error:  * 2*0..CARRY SHOULD BE 1

divE:
  * Q0=0, CARRY=0
  Q=r0;
t=(rhl)+1;  * T = 100001
  t=t+(rhl),DIVIDE;  * t=100001+100000,DIVIDE==)ALU=1, CARRY=1
  t=t+(Q); 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);
skipif(ALU=0);
error; \* 2*(1+1)=4, Q0=0

\* t=(r1)#(Q);
skipif(ALU=0);
error; \* CARRY' WAS 1, Q SHOULD BE 1

CdivB:

\* Q0=0, CARRY=1
Q=r0;
t=right1; \* T = 100001
\* t=t+(right1), CDIVIDE; \* t=100001+100000, DIVIDE => ALU=1, CARRY=1
\* t=t#(2c);
skipif(ALU=0);
error; \* 2*(1+1)=4, Q0=0

\* t=(Q);
skipif(ALU=0);
error; \* 2*0, CARRY' SHOULD BE 0

CdivC:

\* Q0=1, CARRY=0
Q=right1;
t=(r1)+(r1), CDIVIDE; \* t=1+1, DIVIDE => ALU=2, CARRY=0
\* t=t#(5c);
skipif(ALU=0);
error; \* 2*(1), Q[0]=3

\* t=(r1)#(Q);
skipif(ALU=0);
error; \* Q[1:15], CARRY' SHOULD BE 1

CdivD:

\* Q0=1, CARRY=0
T=Q=right1; \* SET Q[0] TO ONE
T=T+(right1); \* T = 100001
\* T+right1, CDIVIDE; \* T=100001+100000, DIVIDE => ALU=1, CARRY=1
\* T=T#(3c);
skipif(ALU=0);
ERROR; \* 2*(1), Q[0]=3

\* T=(Q);
skipif(ALU=0);
ERROR; \* 2*0, CARRY' SHOULD BE 0

CdivE:

\* Q=R01=Q0=0, CARRY=1
Q=R01;
T=right1+1; \* T = 100001
\* T+right1, CDIVIDE; \* T=100001+100000, DIVIDE => ALU=1, CARRY=1
\* T=T#(2c);
skipif(ALU=0);
ERROR; \* 2*(1), Q[0]=2

\* T=(R01) LSH 1;
\* T=T#(Q);
skipif(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;
rsr2=7C;  * THIS WILL BE A 3 BIT MASK
%
slowBrL:
  * TOP OF LOOP
  t=rml;
  rsrc=t;
  t=q;
  BDISPATCH-t;  * DO DISPATCH W/ BITS IN T (=Q)
  branch[BDTbl];
slowBRnoBr:  * should have branched and didn't
  error;
%
bdConverge:
t=(rsrc2)AND(q);  * MASK DISPATCH VALUE
  t=t#(rsrc);
  branch[.+2,ALU=0];
slowBRbadBr:  * SHOULD HAVE BRANCHED AND DIDN'T
  error;  * DIDN'T GO WHERE WE EXPECTED
  t=(r1)+(q);  * GET NEXT VALUE FOR DISPATCH
  loopChk[slowBrL,CNT#0&-1],q=t;
%
afterBD: goto[afterKerne14];
SET[BDTblLoc,6110];
BDTbl:
goto[bdConverge], rsrc=r0, AT[BDTblLoc];
goto[bdConverge], rsrc=r1, AT[BDTblLoc,1];
goto[bdConverge], rsrc=r2, AT[BDTblLoc,2];
goto[bdConverge], rsrc=r3, AT[BDTblLoc,3];
goto[bdConverge], rsrc=r4, AT[BDTblLoc,4];
goto[bdConverge], rsrc=r5, AT[BDTblLoc,5];
goto[bdConverge], rsrc=r6, AT[BDTblLoc,6];
goto[bdConverge], rsrc=r7, AT[BDTblLoc,7];
goto[bdConverge], rsrc=r8, AT[BDTblLoc,8];
goto[bdConverge], rsrc=r9, AT[BDTblLoc,9];
goto[bdConverge], rsrc=10C, AT[BDTblLoc,10];  * shouldn't be here
  goto[bdConverge], rsrc=11C, AT[BDTblLoc,11];  * shouldn't be here
  goto[bdConverge], rsrc=12C, AT[BDTblLoc,12];  * shouldn't be here
  goto[bdConverge], rsrc=13C, AT[BDTblLoc,13];  * shouldn't be here
  goto[bdConverge], rsrc=14C, AT[BDTblLoc,14];  * shouldn't be here
  goto[bdConverge], rsrc=15C, AT[BDTblLoc,15];  * shouldn't be here
  goto[bdConverge], rsrc=16C, AT[BDTblLoc,16];  * shouldn't be here
  goto[bdConverge], rsrc=17C, AT[BDTblLoc,17];  * shouldn't be here
This file defines ALU operations for the program. Definitions for all possible ALU operations are given here with one line/definition. The lines for the 20 operations enabled should have the "n" replaced by consecutive values from 0 to 17. 16 should be "NOT A" for the shifter and 17 is normally used a variable by BitBlt (i.e., not defined here), but can be used otherwise, if desired. Other lines should be commented out.

Note that the letter "E" added as an argument where noted below converts the operation to emulator-only. This is intended for locations used as variables by BitBlt, which must restore the smashed ALUFM locations to the proper value before exit.

The "A" and "B" operations must both be defined because many macros optionally route using either A or B for RB, T, MD, and Q (B path is preferred). Since ALUF[O] is the default for microinstructions, it should contain a non-arithmetic operation to preserve CARRY and OVERFLOW branch conditions.

Also, when running Midas the "B" operation is stored in ALUFM 0 and the NOT A operation in ALUFM 16, so these should be loaded the same way by the microprogram to avoid confusion, unless there is some good reason to do otherwise.

Note that an important choice must be made between the arithmetic and logical versions of ALU+A. The arithmetic version allows XORCARRY with ALU+A but smashes OVERFLOW and CARRY branch conditions on ALU+A.
ABOPB[PD+,0,25];  *ALU=B (use n=0)
ABOPA[PD+,1,0];  *ALU=A
 *(arithmetic--so XORCARRY ok. Requires no more
   *time than boolean "A" in absence of carryin)
   *
   ABOPA[PD-,n,37];  *ALU=A (logical), so XORCARRY illegal but OVERFLOW
   *and CARRY branch conditions not smashed).

*"NOT B" and "NOT A" must be both defined also.
ABOPB[NOT-,15,13];  *NOT B
ABOPA[NOT-,16,1];  *NOT A (use n=16 for shifter)

*Operations of no args (4th arg = letter E makes emulator-only)
ZALUOP[A0,2,31];  *A0 [= all zeroes]
ZALUOP[A1,3,7];  *A1 [= all ones]

*Arithmetic operations of two args [do not accept "slow" (external)
B sources] (5th arg = letter E makes emulator-only)
SALUOP[+,*,4,14];  *A+B
   *
   *SALUOP[+,+1,n,214];  *A+B+1
SALUOP[-,-5,222];  *A-B
   *
   *SALUOP[-,-1,n,22];  *A-B-1

*Boolean operations of two args (5th arg = letter E makes emulator-only)
**Note how synonyms are defined for # (XOR) and = (XNOR, EQV)
ALUOP[AND,,6,35];  *A AND B
ALUOP[OR,,7,27];  *A OR B
ALUOP[#,10,23];  XALUOP[XOR,n];  *A # B (A XOR B)
   *
   *ALUOP[-,n,35];  XALUOP[XNOR,n];  XALUOP[,EQV,n];
ALUOP[AND NOT,,11,33];  *A AND NOT B [= A AND (NOT B)]
ALUOP[OR NOT,,12,17];  *A OR NOT B [= A OR (NOT B)]
   *
   *ALUOP[NOT,AND,,n,21];  *NOT A AND B [= (NOT A) AND B]
   *
   *ALUOP[NOT,OR,,n,6];  *NOT A OR B [= (NOT A) OR B]
   *
   *ALUOP[NOT,AND NOT,,n,11];  *NOT A AND NOT B [= (NOT A) AND (NOT B)]
   *
   *ALUOP[NOT,OR NOT,,n,3];  *NOT A OR NOT B [= (NOT A) OR (NOT B)]

*Operations of one arg (5th arg = letter E makes emulator-only)
   *
   *AOP[2,,n,6];  *2A
   *
   *AOP[2,+1,n,206];  *2A+1
AOP[+,1,13,200];  *A+1
AOP[,-1,14,36];  *A-1
% Page Numbers: Yes First Page: 1
Heading:
postamble.mc February 1, 1980 6:24 PM %
TITLE[POSTAMBLE];
TOP LEVEL;
% February 1, 1980 6:24 PM
  Fix goto[preBegin], described below, into goto[restartDiagnostic]. Postamble already defines a
**nd 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 statement became part of a
** comment. Unfortunately, automatic line breaks made the statement look as if it were still there rat
**her than making it look like part of the comment line.
September 19, 1979 4:23 PM
  Fix placement errors associated with bumping locations from makeholdvalue and from checksimulat
**ing.
September 19, 1979 3:48 PM
  Bump locations to fit postamble with current os/microD: reallyDone, checkFlags global, make che
**ckFlags 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 t
**he hold simulator may be functioning. Make getIM*, putIM* routines global.
September 16, 1979 1:27 PM
  Bump 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 17, 1979 3:49 PM
  Add breakpoint to "done", and fix, again, a bug associated with task simulation. Set defaultFl
**asP (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 (thi
**s implies the whole value was inverted).
January 26, 1979 10:41 AM
  Change taskCirculate code to accommodate taskSim wakesup for task 10D, 12B
January 18, 1979 5:13 PM
  Modify checkTaskNum to use the RM value, currentTaskNum, and modify taskCircInc to keep the cop
**y 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
done location where diagnostics go when they are finished --gives control to postamble code the incr
**ments iterations, implements hold and task simulation and task circulation.
reallyDone location where postamble inits 2 rm locations then performs "GoTo BEGIN"
restart Reinit diagnostic state, then restart the diagnostic.
incTaskFreq Increment the task frequency counter
incHoldFreq Increment the hold frequency counter
makeHoldValue Construst the "Hold&TaskSim" value from holdFreq and taskFreq, given that each is enab
This code presumes RO=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 increments 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 occurred. 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 (i.e., the time before it gives control to "done") the task simulator is 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 occurrence 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 1M, a routine to return a random number, and routines to initialize a task's PC and to notify it.
**June 17, 1979 4:49 PM**

POSTAMBLE CONSTANTS

* random number generator may have to be moved to "global" call location if extensively used!

```plaintext
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!

```plaintext
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.

```plaintext
flags.taskSim defined in preamble!
flags.holdSim defined in preamble!
flags.simulating defined in preamble!
```

```plaintext
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**
- The task simulator and taskSim[1:3] form a counter that determines the number of cycles before a task **wakeup** occurs.

**Hold simulation**
- 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**
- 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).

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

```plaintext
rmRegion[rmForKernelRtn];
knowRbase[rmForKernelRtn];
```

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

```plaintext
knowRbase[rm2ForKernelRtn];  * defined in preamble because of macros
```

```plaintext
* that reference randV, randX
```

```plaintext
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] & flags.testTasks;  * bookkeeping is done. switch tasks if required
skipp[ALUFO];
branch[preBegin];  * xit if not running other tasks

taskCircInit: * now that bookkeeping is done, switch tasks if required
noop;  * for placement.
call[checkTaskNum];
scr = t;  * scr + nextTask
  t + preBeginLocC;  * link + t + preBeginLoc
subroutine;
taskCirc:
  zeroHold[scr2];  * turn off hold-task sim during ldtpc, wakeup
  link = t;
  t = scr;
  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 = scr = a1;
goto[begin], scr2 = t;

restart:  * restart diagnostics from "initial" state
  randV = t = t;
  restart random number generator
  randX = t = t;
  scr = t = t;  * restart hold/task simulator stuff
  call[putIMRH], t + holdFreqLocC;
  scr = t = t;
  call[putIMRH], t + taskFreqLocC;
  scr = t = t;
  call[putIMRH], t + holdValueLocC;
  scr = 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
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;
  level;

  call[checkFlags], t=flags.taskSim; * see if taskSim enabled
  branch[writeTaskSim, alu=0], t=r0; * use 0 if not enabled
  t=taskFreqLocC; * increment 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;
  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];
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];
   skipIf[alu#0];
   branch[simCtrl0];
   t+holdFreqLocC; * rsr2 += holdFreq
   call[readByte3], t+holdFreqLocC;
   rsr2 += t;

   call[readByte3], t+taskFreqLocC; * t += taskFreq

   t+lsb[t, sim.taskShift]; * position hold and task values
   t+= (t & (sim.taskMask));
   rsr2 += lsb[rsr2, sim.holdShift];
   rsr2 += (rsr2) and (sim.holdMask);
   rsr2 += (rsr2) and (377c);

   rsr2 += (t) + (rsr2); * taskFreq, holdFreq
   rsr += rsr2;

   % 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], rsr += t-t; * write zero into holdValueLoc

   simCtrlRtn:
      goto[topLvlPostRtn];
* September 19, 1979 9:09 PM

**chkSimulating**: subroutine;
  saveReturn[chkSimulatingRtn];
  call[chkSimulating];
  dblbranch[chkSimYes, chkSimNo, alu=0];
chkSimYes: * run the simulator
  t ← (r0)+1;  * rtn w/ alu#0
chkSimNo:
  returnAndBranch[chkSimulatingRtn, t];

* September 19, 1979 9:19 PM

**chkSimulating**: PROCEDURE RETURNS[weAreSimulating: BOOLEAN] =
BEGIN
  weAreSimulating ← FALSE;
  IF flags.Simulating THEN
      weAreSimulating ← TRUE;
    END;
  END;
END;
* 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 flag
**s.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 o
***f 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 = (ZOC); * 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
    rsr = t;
    call[putIMRH], t = nextTaskLocC;
    noop; * for placement

  taskCircRtn:
    goto[topLv1PostRtn];
* 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

incItrsHi16:

  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];
resetHold: subroutine;  * special subroutine called by IM manipulating
* code. This subroutine 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;
oldscr ← t;
t ← rscr2;
oldrscr ← t;
t ← rscr2;
oldrscr2 ← t;  * link, t, rscr, and rscr2 are now saved.

** 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];

** knowRBbase[rmForKernelRtn];  * restore link, t, rscr, rscr2, then return
RBbase ← rbbase[rmForKernelRtn];
t ← oldscr;
rscr ← t;
t ← oldrscr2;
rscr2 ← t;
subroutine;
link ← resetHoldRtn;
return, t ← oldt, RBbase ← rbbase[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[rnForKernelRtn];
  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 instruct
  **ion, 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;
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 phase 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 significant byte in IM
  top level; * read byte 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
  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;
  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 (bl);   * see if the data is inverted. If so, b1 will
    skipf[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 (bl);   * see if the data is inverted. If so, b1 will
    skipf[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 = IMHL

    top level;
    call[resetHold];
    subroutine;
    return;

putIMRH: subroutine, global;   * T = addr, RSCR = value, clobber RSCR2
    rscr2 = link;
    link + t;
    zeroHold[t];   * disable task/hold Sim before touching IM

    top level;
    t = rscr;
    IMRLH = POK + t;
    call[resetHold];
    subroutine;
    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;
    IMLHLH = POK + t;
    call[resetHold];
    subroutine;
    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 (bl);   * see if the data is inverted. If so, b1 will
    skipf[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.
notifyTask: subroutine;
    rscr~ link;
    bigBD+ispacht~; t;
    top level;
    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 ~ rbasa[mainPostRtn];
    link ~ mainPostRtn;
    return, RBASE ~ rbase[defaultRegion];

scopeTrigger: subroutine;
    t ~ a0, global;
    TIOA ~ t, T=0;
    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
  \[\text{random; \text{goto}[\text{random1}]. \text{t} \rightarrow \text{rndm0, RBASE} \rightarrow \text{rbase[\text{randV}], at[\text{randomTloc,0}]; \text{knowRbase[\text{randomRM}];}}\]
  \[\text{goto[\text{random1}]. \text{t} \rightarrow \text{rndm1, RBASE} \rightarrow \text{rbase[\text{randV}], at[\text{randomTloc,1}]; \text{knowRbase[\text{randomRM}];}}\]
  \[\text{goto[\text{random1}]. \text{t} \rightarrow \text{rndm2, RBASE} \rightarrow \text{rbase[\text{randV}], at[\text{randomTloc,2}]; \text{knowRbase[\text{randomRM}];}}\]
  \[\text{goto[\text{random1}]. \text{t} \rightarrow \text{rndm3, RBASE} \rightarrow \text{rbase[\text{randV}], at[\text{randomTloc,3}]; \text{knowRbase[\text{randomRM}];}}\]
  \[\text{goto[\text{random1}]. \text{t} \rightarrow \text{rndm4, RBASE} \rightarrow \text{rbase[\text{randV}], at[\text{randomTloc,4}]; \text{knowRbase[\text{randomRM}];}}\]
  \[\text{goto[\text{random1}]. \text{t} \rightarrow \text{rndm5, RBASE} \rightarrow \text{rbase[\text{randV}], at[\text{randomTloc,5}]; \text{knowRbase[\text{randomRM}];}}\]
  \[\text{goto[\text{random1}]. \text{t} \rightarrow \text{rndm6, RBASE} \rightarrow \text{rbase[\text{randV}], at[\text{randomTloc,6}]; \text{knowRbase[\text{randomRM}];}}\]
  \[\text{goto[\text{random1}]. \text{t} \rightarrow \text{rndm7, RBASE} \rightarrow \text{rbase[\text{randV}], at[\text{randomTloc,7}]; \text{knowRbase[\text{randomRM}];}}\]

\[\text{random1: \text{return, \text{t} \rightarrow \text{randV} \rightarrow (\text{randV})+\text{t}; \text{knowRbase[\text{defaultRegion}];}}\]

\[\text{saveRandState: subroutine; \text{* remember random number seed}}\]
  \[\text{RBASE} \leftarrow \text{rbase[\text{randV}]; \text{oldRandV} \leftarrow \text{randV}; \text{oldRandX} \leftarrow \text{randX}; \text{return, RBASE} \leftarrow \text{rbase[\text{defaultRegion}];}}\]

\[\text{restoreRandState: subroutine; \text{* restore remembered random number seed}}\]
  \[\text{RBASE} \leftarrow \text{rbase[\text{randV}]; \text{randV} \leftarrow \text{oldRandV}; \text{randX} \leftarrow \text{oldRandX}; \text{return, RBASE} \leftarrow \text{rbase[\text{defaultRegion}];}}\]

\[\text{getRandV: subroutine; \text{RBASE} \leftarrow \text{rbase[\text{randV}]; \text{RETURN, \text{t} \rightarrow \text{randV, RBASE} \rightarrow \text{rbase[\text{defaultRegion}];}}\]
* 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];  * xor new bits
    t + t # (rscr2);
    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~aO;  * 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~aO;  * 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 o
**ff the hold simulator.

enableConditionalTask sets flags.conditionOK and flags.conditional, then it calls makeHoldValue to forc
**e 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.condition);  * 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
  **t flags if undefined
  data[(Flags: lh[O] rh[defaultFlagsP]. at[flagsLoc])];  * CONTROL FLAGS
  data[(taskFreq: lh[O] rh[O]. at[taskFreqLoc])];  * task sim value
  data[(holdFreq: lh[O] rh[O]. at[holdFreqLoc])];  * hold sim value
  data[(nextTask: lh[O] rh[O]. at[nextTaskLoc])];  * next task value
  data[(holdValue: lh[O] rh[O]. at[holdValueLoc])];  * current hold value
  data[(iterations: lh[O] rh[O]. at[itrsLoc])];  * iteration count

postDone: noop;