

```

TITLE[IfuComplex];
IM[ILC,0];
TOP LEVEL;
* November 6, 1978 3:39 PM
%
ifu.cm          assemble and place entire ifu diagnostic
ifuSaveDefc.cm assemble ifuSaveDefcs, ifuDefcs, ifuPostamble
ifuOnly.cm     assemble ifu.mc
iful.cm        assemble iful.mc
ifuPlace.cm    place ifu diagnostic

ifuSaveDefcs.cm assemble IfuSaveDefcs (dlLang, kernelAlu, preamble)
ifuDefcs.cm    assemble IfuDefcs
ifuPostamble.cm assemble Ifu's version of Postamble
ifuToLogin.cm dump all ifu sources onto [ivy]<disk login name>dorado>IfuSources.dm,
                dump ifu.midas, ifu.mb onto [ivy]<disk login name>dorado>ifu.dm
ifuToIvy.cm    dumps sources and .mb, as above except it goes to official dorado directories

ifuFiles.cm    list of all the source files associated with ifu
%
%*****
June 18, 1981 9:49 AM
    Change this file to IfuComplex
May 20, 1981 3:25 PM
    Accommodate split of ifu3.mc into ifu3a, ifu3b, ifu3c
February 1, 1980 6:26 PM
    Change preBegin into restartDiagnostic -- postamble had already defined that label.
February 1, 1980 11:53 AM
    Add preBegin code to zero memoryOK.
October 14, 1979 3:33 PM
    Remove beginIfu4 -- now a separate diagnostic.
September 27, 1979 10:00 AM
    Add call to beginIfu4.
September 19, 1979 11:16 PM
    Add beginIfuTests label.
September 17, 1979 5:16 PM
    call beginIful,3 rather than goto them.
%*****

restartDiagnostic:
    memoryOK_a0;
BEGIN:
    t _ (r0)+1;    * init the stkp to one
    stkp _ t;
    RBASE _ rbase[defaultRegion];
    MEMBX_0s;
    rscr _ A0;
    call[setBR], rscr2 _ A0;
    call[resetIfu];    * remove this 3 line hack asap

beginIfuTests:
    call[beginIfu3a];
    call[beginIfu3b];
    call[beginIfu3c];
    r1 _ (r0)+1;
    goto[done];

* CODE for midas debugging
    top level;

    top level;
11:    branch[l1];
12:    call[justReturn];
        branch[l2];
13:    call[justReturn];
        call[justReturn];
        branch[l3];
14:    call[justReturn];
        call[justReturn];

```



```

TITLE[IfuSimple];
IM[ILC,0];
TOP LEVEL;
* November 6, 1978 3:39 PM
%
ifu.cm assemble and place entire ifu diagnostic
ifuSave.cm assemble ifuSaveDefs, ifuDefs, ifuPostamble
ifuOnly.cm assemble ifu.mc
iful.cm assemble iful.mc
ifuPlace.cm place ifu diagnostic

ifuSaveDefs.cm assemble IfuSaveDefs (dlLang, kernelAlu, preamble)
ifuDefs.cm assemble IfuDefs
ifuPostamble.cm assemble Ifu's version of Postamble
ifuToLogin.cm dump all ifu sources onto [ivy]<disk login name>dorado>IfuSources.dm,
dump ifu.midas, ifu.mb onto [ivy]<disk login name>dorado>ifu.dm
ifuToIvy.cm dumps sources and .mb, as above except it goes to official dorado directories

ifuFiles.cm list of all the source files associated with ifu
%
%*****
June 18, 1981 9:48 AM
Change this file to ifuSimple
May 20, 1981 3:25 PM
Accommodate split of ifu3.mc into ifu3a, ifu3b, ifu3c
February 1, 1980 6:26 PM
Change preBegin into restartDiagnostic -- postamble had already defined that label.
February 1, 1980 11:53 AM
Add preBegin code to zero memoryOK.
October 14, 1979 3:33 PM
Remove beginIfu4 -- now a separate diagnostic.
September 27, 1979 10:00 AM
Add call to beginIfu4.
September 19, 1979 11:16 PM
Add beginIfuTests label.
September 17, 1979 5:16 PM
call beginIful,3 rather than goto them.
%*****

restartDiagnostic:
memoryOK_a0;
BEGIN:
t _ (r0)+1; * init the stkp to one
stkp _ t;
RBASE _ rbase[defaultRegion];
MEMBX_0s;
rscr _ A0;
call[setBR], rscr2 _ A0;
call[resetIfu]; * remove this 3 line hack asap

beginIfuTests:
call[beginIful];
r1 _ (r0)+1;
goto[done];

* CODE for midas debugging
top level;

top level;
11: branch[11];
12: call[justReturn];
branch[12];
13: call[justReturn];
call[justReturn];
branch[13];
14: call[justReturn];
call[justReturn];
call[justReturn];
branch[14];

```

```
15: call[justReturn];
    call[justReturn];
    call[justReturn];
    call[justReturn];
    branch[15];

16: call[justReturn];
    call[justReturn];
    call[justReturn];
    call[justReturn];
    call[justReturn];
    branch[16];

17: call[justReturn];
    call[justReturn];
    call[justReturn];
    call[justReturn];
    call[justReturn];
    call[justReturn];
    branch[17];
```

```
END;
```

```

TITLE[Iful];
%*+++++
                CONTENTS

TEST                DESCRIPTION

beginIful:        Control subroutine that calls each test in this file
i.XXXXXX          IfuReset, PcFG_, getPcX, ifuJump, resched, various scope loops, etc.
ifuMemRW:        read and write ifu memory
iMemRandAddr:    Ifu memory r/w test that uses random addr and random data.
iMemAddr:        Test addressing logic for ifu memory.
ifuMScopeLoop:   infinite loop that reads location 0 then location 1 of IfuM
iSingleStepTest: single step, under Dorado processor control, ifu
itTestCase1:     First test that uses the ifuTest register
%*+++++

%
June 18, 1981 9:26 AM
    Move common code from this file into IfuSubrs.mc
May 18, 1981 2:25 PM
    Fix nextPat to invoke cycleRandV to get better performance from the random number generator.
February 1, 1980 7:58 PM
    Fix stack handling bug in return code of addressing test. Fix old bug in fastStore code that was
    fetching rather than storing.
February 1, 1980 7:48 PM
    Add code to mask out unwanted bits from getIfuLH in addressing test.
February 1, 1980 6:37 PM
    Add iMemAddr, an addressing logic test.
February 1, 1980 5:06 PM
    Improve the set of patterns generated by pat16.
September 19, 1979 1:13 PM
    Reset expectedDispatch<0 after invoking iSingleStepTest.
September 19, 1979 12:55 PM
    Move beginIful to the beginning of the file, add more and better comments, and set up
    iSingleStepTest to loop 400 times.
September 17, 1979 5:29 PM
    Add iSingleStepTest.
July 3, 1979 1:53 AM
    Increment return link in checkException by 1 -- because afterdispatch will also decrement it & we
    need compatability. This increment occurs after the potential error check where the link has been
    decremented to make error interpretation easier.
July 3, 1979 1:17 AM
    Decrement return link in checkException by 1 -- so that it points to where we came from rather
    than where to return to, which is irrelevant..
July 3, 1979 12:06 AM
    Add code to enable dynamically selected exception conditions.
June 29, 1979 11:46 AM
    Modify afterDispatch to save POINTERS into (stack+1), but to return control with stkp = entering
    value. Do this since afterDispatch resets RBASE to defaultRegion, and some tests want to check the
    value that RBASE got set to when the IfuJump occured.
June 20, 1979 9:14 AM
    Add call to resetIfu in iMemRandAddr and cause beginIful to invoke iMemRandAddr.
June 20, 1979 8:50 AM
    Add ifuLHmaskC masking to data in iMemRandAddr
June 20, 1979 7:35 AM
    Add random address generation test (iMemRandAddr)
June 19, 1979 10:34 AM
    Fix register clobbering bug in new version of ifuMemRW.
June 18, 1979 8:56 AM
    Fix bug in 3rd pattern of nextPat16.
June 17, 1979 4:04 PM
    Add entry points for all the exception conditions
June 15, 1979 3:24 PM
    Enable random patterns in iMemRW
June 5, 1979 12:43 PM
    Remove i.testCase2L since test2 has been removed from default diagnostic.
June 5, 1979 11:31 AM
    Add disableConditionTask
May 30, 1979 10:08 AM
    Add random numbers to pat16 code; diddle iMemRW to reset the Ifu only once.
May 9, 1979 9:05 AM
    Fix subroutine mode error in ifuMScopeLoop. Move contents section to top of file & add more

```

comments.

May 4, 1979 10:57 AM

Add scope loop for checking Ram speed.

May 3, 1979 8:44 AM

Add calls to setIUsingInstrSet before invoking initIfuM1Thru(3,16)

April 26, 1979 9:53 AM

Add pat8 subroutines.

April 25, 1979 12:13 PM

Add fastExit, fastFetch, fastStore code.

April 24, 1979 5:27 PM

Remove references to dispatchOffset, simplify afterDispatch.

April 22, 1979 5:10 PM

Move iAddFGParity thru itStep into ifuStepSubrs.mc

February 7, 1979 10:35 AM

Add breakpoints to rampe, kfault, resched entries; explicitly place all the entry points for the instructions.

February 6, 1979 2:16 PM

Add requisite noop after PcF_.

February 5, 1979 12:54 PM

Add ifAd20 -- ifu pause/halt code

February 5, 1979 9:59 AM

Change opifad13,14,16 to do ifuJumps directly.

January 18, 1979 6:44 PM

Fix misc. bugs in pattern16, muffler bit reading code

January 17, 1979 3:52 PM

More Rev-Bb mods: define exception code for instrSet 1 (locations 200,204,214,234,274). Define itMosFhSh.

January 15, 1979 4:52 PM

Rev-Bb mods: instrSet bits, complemented, are or'd into Exception addresses. InstrSet bits 2,3 pick bytes right to left. Cycled 0 pattern for Ifu memory test.

%

* February 1, 1980 6:37 PM

beginIful:

```
pushReturn[];  
call[disableConditionalTask];  
call[ifuMemRW];  
call[iMemRandAddr];  
call[iMemAddr];  
call[iSingleStepTest];  
expectedDispatch_al;  
returnP[];
```

```

* November 30, 1978 9:07 AM
%*****
SINGLE STEP and INITIAL DEBUGGING CODE:
%*****

top level;
i.getPcX: branch[.], t _ not(PcX');

i.ifuJump: ifuJump[0]; * try ifu next macro...

i.ifuJumpF:
    ifuJump[0, r odd], r0; * conditional branch condition is false
    noop;

i.ifuJumpT:
    t _ (r0)+1;
    ifuJump[0, ALU=0]; * conditional branch condition is true
    noop;

i.mos: branch[.], mos _ t; * set instructyion set w/ t

i.nextData: branch[.], t _ ID; * get next data from ifu

i.noResched: branch[.], noReschedule; * NO resched

i.reset: branch[.], IfuReset; * Ifu Reset

i.resched: branch[.], reschedule; * resched

i.setPcFG: branch[.], PcF_t; * SET T

i.test: branch[.], ifuTest _ t; * load TEST register

i.tick: branch[.], ifutick; * Tick

i.WriteLHL: * Write LEFT HALF IFU ram SET. rscr3 = addr, rscr4 = value
    rscr _ rscr4; * value to write kept in rscr4
    call[putIfuLH], t _ rscr3; * address to write kept in rscr3
    branch[i.WriteLHL]; * do it again for scope loop
i.WriteRHL: * Write RIGHT HALF IFU ram. SET rscr3 = addr, rscr4 = value
    rscr _ rscr4; * value to write kept in rscr4
    call[putIfuRH], t _ rscr3; * address to write kept in rscr3
    branch[i.WriteRHL]; * do it again for scope loop

i.ReadLHL: * Read LEFT HALF IFU ram. SET rscr3 = addr
    call[getIfuLH], t _ rscr3; * keep addr to read in rscr3
    branch[i.ReadLHL]; * do it again for scope loop
i.ReadRHL: * Read RIGHT HALF IFU ram. SET rscr3 = addr
    call[getIfuRH], t _ rscr3; * keep addr to read in rscr3
    branch[i.ReadRHL]; * do it again for scope loop

i.testCase1L:
    call[setIUsingInstrSet], t _ 1C;
    call[initIfuM1Thru3];
    call[itTestCase1];
    branch[.-1];
% * commented out since we're removed test2 from ifu default diagnostic
i.testCase2L:
    call[setIUsingInstrSet], t _ 1C;
    call[initIfuM0Thru16];
    call[itTestCase2];
    branch[.-1];

%

i.WR1: *Write, then Read an ifu memory location
    rscr _ 1c; * use FF constants so it's easy to patch.
    t _ 2c;
    call[putIfuRH];
    t_1c;
    call[getIfuRH];
    branch[i.WR1];

```



```
i.RW1:                                *Read, then Write an ifu memory location
    t_1c;                               * use FF constants so it's easy to patch.
    call[getIfuRH];
    rscr _ 1c;
    t _ 2c;
    call[putIfuRH];
    branch[i.RW1];
```

* May 4, 1979 10:59 AM

```

iMemRW                                Read and write the ifu RAM
FOR pat IN NPats DO
    saveRandState[];                    -- remember random number state
    FOR iAddrX IN [0..1777B] DO
        ifuLoMem[addr] _ getPattern[pat, addr];
        ifuHiMem[addr] _ getPattern[pat, addr];
    ENDFOR;

    restoreRandState[];                 -- restore random number state
    FOR iAddrX IN [0..1777B] DO
        IF ifuLoMem[addr] # getPattern[pat,addr] THEN ERROR;
        IF ifuHiMem[addr] # getPattern[pat,addr] THEN ERROR;
    ENDFOR;                               -- end of address loop;

ENDFOR;                                   -- end of pattern loop

```

Notice that this code pushes the stack and keeps the current address there.

```

ifuMemRW:                             * read and write the Ifu memory
    pushReturn[];
    IfuReset[];                         * make sure its stopped
    call[iPat16], stkp+1;                 * init 16 bit patterns. keep addr on stack

iMemPatL:
    call[nextPat16];                     * select next 16 bit pattern or exit
    branch[afterIMemPatL,ALU=0];
    noop;                                 * for placement

    call[saveRandState];
    call[iIAddr];                         * init ifu addresses

iMemAddrWL:
    call[nextIAddr];                     * select next ifu address or exit
    branch[afterIAddrWL,ALU=0], stack _ t; * save current addr on stack!
    noop;

* Write Left Half of Ifu memory
    call[getPat16], t _ stack;            * rtn t = pattern
    rscr _ t;                             * pass pattern in rscr
    call[putIfuLH], t _ stack;           * write ifu left half

* Write Right Half of Ifu memory
    call[getPat16], t _ stack;            * rtn t = pattern
    rscr _ t;                             * pass pattern in rscr
    call[putIfuRH], t _ stack;           * write Ifu right half
    branch[iMemAddrWL];

```

```

* June 19, 1979 10:34 AM

* Now read and check
afterIAddrWL:
    call[restoreRandState];
    call[iIAddr];
iMemAddrRL:
    call[nextIAddr];
    branch[afterIAddrRL, ALU=0], stack _ t;      * save current addr on stack!
    noop;

* Check Left Half
    call[getIfuLH];                               * read back left half
    rscr _ t;                                     * remember LH in rscr.
    call[getPat16], t _ stack;                   * rtns t = pattern
    rscr _ (rscr) and (ifuLHmaskC);              * remove ID count, InstrSet
    t _ t and (ifuLHmaskC);                      * ignore corresponding bits in pattern
    (rscr) # t;
    skipif[ALU=0];
iMemLHErr1:
    error;
    * iAddrx= address that failed
    * left half data not what we wrote.
    * t = expected value; rscr = value from ifu

* Check Right Half
    call[getIfuRH], t _ stack;                   * read back right half
    rscr _ t;                                     * remember RH in rscr.
    call[getPat16], t _ stack;                   * rtns t = pattern

    (rscr) # t;
    skipif[ALU=0];
    * iAddrx= address that failed
iMemRHErr1:
    error;
    * right half data not what we wrote
    * t = expected value, rscr = ifu value

    branch[iMemAddrRL];                          * branch for next address
afterIAddrRL:
    branch[iMemPatL];
afterIMemPatL:
    pReturnP[];
    * extra pop since we've kep iAddr there.

```

* June 20, 1979 9:13 AM

iMemRandAdrrs

This test generates a series of pairs of random addresses, writes into those pairs of locations and then checks the data to see that it is correct.

FOR itrs IN [0..10000] DO

```

    addr2 _ addr1 _ getRandom[] and (1777B);
    UNTIL addr2 # addr1 DO addr2 _ getRandom[] and (1777B);
    lh1 _ getRandom[] and lhMask; rh1 _ getRandom[];
    lh2 _ getRandom[] and lhMask; rh2 _ getRandom[];
    ifuLhMem[addr1] _ lh1; ifuRhMem[addr1] _ rh1;
    ifuLhMem[addr2] _ lh2; ifuRhMem[addr2] _ rh2;
    IF (ifuLhMem[addr1] and ifuLHmask) # lh1 THEN ERROR;
    IF ifuRhMem[addr1] # rh1 THEN ERROR;
    IF (ifuLhMem[addr2] and ifuLHmask) # lh2 THEN ERROR;
    IF ifuRhMem[addr2] # rh2 THEN ERROR;
    ENDL00P;

```

iMemRandAdrrs: pushReturn[];

call[resetIfu];

t _ 10000C;

cnt _ t;

* loop 10000B times

iMemRandAdrrsL:

noop;

* for placement

call[get1Rand];

iAddr1 _ t _ t and (1777C);

iAddr2 _ t;

iMemRandAdrrsL2:

* get 2 unique, random addresses

call[get1Rand];

t _ t and (1777C);

* isolate the bits for ifu ram addresses

t # (iAddr1);

branch[iMemRandAdrrsL2, ALU=0];

* still need different, random address

noop;

* for placement

call[get1Rand], iAddr2 _ t;

* get random values

call[get1Rand], value1left _ t and (IfuLHmaskC);

call[get1Rand], value1right _ t;

call[get1Rand], value2left _ t and (IfuLHmaskC);

* now write random data into our two randomly chosen locations

value2right _ t;

* write into iAddr1

rscr _ value1left;

call[putIfuLh], t _ iAddr1;

rscr _ value1right;

call[putIfuRh], t _ iAddr1;

rscr _ value2left;

* write into iAddr2

call[putIfuLh], t _ iAddr2;

rscr _ value2right;

call[putIfuRh], t _ iAddr2;

* now check that the data is correct.

call[getIfuLh], t _ iAddr1;

t _ t and (IfuLHmaskC);

* remove the extra, non Ifu Ram bits

t # (value1left);

skpip[ALU=0];

iMemRandAdrrsErr1lh:

error;

* value at iAddr1 has left half value in T

* and it doesn't match value1left.

call[getIfuRh], t _ iAddr1;

t # (value1right);

skpip[ALU=0];

iMemRandAdrrsErr1rh:

error;

* value at iAddr1 has right half value in T

* and it doesn't match value1right

call[getIfuLh], t _ iAddr2;

t _ t and (IfuLHmaskC);

* remove the extra, non Ifu Ram bits

```
t # (value2left);
skipif[ALU=0];
iMemRandAddrErr2lh:
error;
                                * value at iAddr2 has left half value in T
                                * and it doesn't match value2left

call[getIfuRh], t _ iAddr2;
t # (value2right);
skipif[ALU=0];
iMemRandAddrErr2rh:
error;
                                * value at iAddr2 has right half value in T
                                * and it doesn't match value2right

loopUntil[cnt=0&-1, iMemRandAddrL];

returnP[];
```

* February 1, 1980 6:39 PM

%*+++++

iMemAdrs

This test checks the memory addressing logic. It assumes that the data bits work. Failures in the data bits may produce a spurious complaint from this test. The algorithm is identical to the one employed in all the microcoded addressing tests for the memory system.

Storage Addressing Test

Background memory w/ 0 then write -1 into ascending addresses. Before the write, check that the word that is about to be written is still zero. Suppose it is non zero. Then there was an addressing error. Remember the address of the non zero word, background memory with zeros again and proceed writing -1s. This time, check the previously clobbered address each time before writing the -1. This approach will catch the reference that clobbers the known location. The same algorithm can be applied for descending addresses, mutatis mutandi, to finish a complete check of the addressing logic.

iAddrCheck: PROCEDURE=

BEGIN

zeroMem: PROCEDURE

BEGIN

FOR i IN IfuMemLimits DO IfuMem[i]_0; ENDLOOP;

END;

findErrUp: PROCEDURE [clobbered: VA] =

BEGIN

zeroMem[];

FOR i IN VA DO

IF IfuMem[clobbered]#0 THEN SIGNAL ErrUp[i-1, clobbered];

IfuMem[i] _ -1;

ENDLOOP;

SIGNAL IntermittentErrUp[clobbered];

END;

findErrDown: PROCEDURE [clobbered: VA] =

BEGIN

zeroMem[];

FOR i DECREASING IN VA DO

IF IfuMem[clobbered] #0 THEN SIGNAL ErrDown[i+1, clobbered];

IfuMem[i] _ -1;

ENDLOOP;

SIGNAL IntermittentErrDown[clobbered];

END;

-- this is the program

zeroMem[];

FOR i IN VA DO

IF IfuMem[i] # 0 THEN FindErrUp[i];

IfuMem[i] _ -1;

ENDLOOP;

zeroMem[];

FOR i DECREASING IN VA DO

IF IfuMem[i] # 0 THEN FindErrDown[i];

IfuMem[i] _ -1;

ENDLOOP;

END;

%*+++++

iMemAdrs:

pushReturn[];

call[zeroIfuMemory];

call[iIAddr],stkp+1;

iAddrUpL:

call[nextIAddr];

branch[iAddrUpLxit, alu=0], stack _ t;

noop;

call[getIfuLH],t_stack; * check the left half

PD_t and (IfuLHmaskC);

skpif[ALU=0];

branch[iAddrUpFindErr];

rscr_al;

call[putIfuLH],t_stack; * write all ones into left half

call[getIfuRH],t_stack; * check the right half

PD_t;

```
skpif[ALU=0];
branch[iAddrUpFindErr];
rscr_al;
call[putIfuRH],t_stack;    * write all ones into right half

branch[iAddrUpL];

iAddrUpLxit:
call[zeroIfuMemory];
call[iIAddrDownCtrl];

iAddrDownL:
call[nextIAddrDown];
branch[iAddrTestDone, alu=0], stack _ t;
noop;

call[getIfuLH],t_stack;    * check the left half
PD_t and (IfuLHmaskC);
skpif[ALU=0];
branch[iAddrDownFindErr];
rscr_al;
call[putIfuLH],t_stack;    * write all ones into left half

call[getIfuRH],t_stack;    * check the right half
PD_t;
skpif[ALU=0];
branch[iAddrDownFindErr];
rscr_al;
call[putIfuRH],t_stack;    * write all ones into right half

branch[iAddrDownL];
```

* January 11, 1979 9:14 AM

iAddrUpFindErr

This test tries to isolate the reference that clobbered the location (denote it the *target location*) that was detected by the **iAddrUpL** loop. The test proceeds as before except that as it ascends memory it continually checks to see if the *target location* has been clobbered yet.

In general, stack[stkp] contains the "current" address, and stack[stkp-1] contains the clobbered address detected in iAddrUpL.

iAddrUpFindErr:

```
t_stack&+1;          * we'll save clobbered (target location)
stack_t;            * location in stack!
```

```
call[zeroIfuMemory];
stkp-1;
call[getIfuLH],t_stack&+1; * reexamine target location
PD_t and (IfuLHmaskC);
skpif[ALU=0],stkp-1;
```

iAddrUpErr1a:

```
error;              * zeroIfuMemory didn't work. stkp has
                    * address in question.
```

```
call[getIfuRH], t_stack&+1;
PD_t;
skpif[alu=0],stkp-1;
```

iAddrUpErr1b:

```
error;              * zeroIfuMemory didn't work. stkp has
                    * address in question.
```

```
call[iIAddr],stkp+1;
```

iAddrUpFindL:

```
call[nextIAddr];
branch[iAddrUpNoFind, alu=0], stack&-1 _ t;
noop;
```

```
call[getIfuLH],t_stack&+1; * check the left half of target addr
PD_t and (IfuLHmaskC);
skpif[ALU=0];
```

iAddrUpErr2a:

```
error;              * the last store, (value on the stack)-1,
                    * clobbered location at (stkp-1).
```

```
rscr_al;
call[putIfuLH],t_stack; * write all ones into left half
```

```
stkp-1;            * must check right half of targert addr
call[getIfuRH],t_stack&+1; * check the right half
PD_t;
```

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

iAddrUpErr2b:

```
error;              * the last store, (value on the stack)-1,
                    * clobbered location at (stkp-1).
```

```
rscr_al;
call[putIfuRH],t_stack; * write all ones into right half
```

```
branch[iAddrUpFindL];
```

iAddrUpNoFind:

```
error;              * intermittent error;
```


* February 1, 1980 7:57 PM

iAddrDownFindErr

This test tries to isolate the reference that clobbered the location (denote it the *target location*) that was detected by the **iAddrDownL** loop. The test proceeds as before except that as it descends memory it continually checks to see if the *target location* has been clobbered yet. Use **Flush_** to force the target munched out of the cache.

In general, `stack[stkp]` contains the "current" address, and `stack[stkp-1]` contains the clobbered address detected in **iAddrUpL**.

iAddrDownFindErr:

```
t_stack&+1;          * we'll save clobbered (target location)
stack_t;            * location in stack!
call[zeroIfuMemory];
stkp-1;
call[getIfuLH],t_stack&+1; * reexamine target location
PD_t and (IfuLHmaskC);
skpif[ALU=0],stkp-1;
```

iAddrDownErr1a:

```
error;              * zeroIfuMemory didn't work. stkp has
                    * address in question.
```

```
call[getIfuRH], t_stack&+1;
PD_t;
skpif[alu=0],stkp-1;
```

iAddrDownErr1b:

```
error;              * zeroIfuMemory didn't work. stkp has
                    * address in question.
```

```
call[iIAddrDownCtrl],stkp+1;
```

iAddrDownFindL:

```
call[nextIAddrDown];
branch[iAddrDownNoFind, alu=0], stack _ t;
noop;
```

```
call[getIfuLH],t_stack&+1; * check the left half of target addr
PD_t and (IfuLHmaskC);
skpif[ALU=0];
```

iAddrDownErr2a:

```
error;              * the last store, (value on the stack)-1,
                    * clobbered location at (stkp-1).
```

```
rscr_al;
call[putIfuLH],t_stack; * write all ones into left half
```

```
stkp-1;            * must check right half of target addr
call[getIfuRH],t_stack&+1; * check the right half
PD_t;
```

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

iAddrDownErr2b:

```
error;              * the last store, (value on the stack)-1,
                    * clobbered location at (stkp-1).
```

```
rscr_al;
call[putIfuRH],t_stack; * write all ones into right half
```

```
branch[iAddrDownFindL];
```

iAddrDownNoFind:

```
error;              * intermittent error;
```

iAddrTestDone:

```
pReturnP[];
```

* February 1, 1980 7:25 PM

zeroIfuMemory:

```
pushReturn[];
call[iIAddr],stkp+1;          * we'll keep the addr on stack
```

ziMemL:

```
call[nextIAddr];
branch[ziMemXit, ALU=0], stack_t;  * exit if no more addresses to zero
rscr_a0;
call[putIfuLH], t_stack;          * zero the left half
rscr_a0;
call[putIfuRH], t_stack;          * zero the right half
branch[ziMemL];
```

ziMemXit:

```
pReturnP[];
returnP[];
```

* May 4, 1979 11:06 AM
top level;

IfuMScopeLoop:

t _ 1c;
stkp _ t;
call[resetIfu];

rscr _ 0c;
t _ 0c;
call[putIfuLH];
rscr _ 0c;
t _ 0c;
call[putIfuRH];

* write zeros in location 0 left half

* write zeros in location 0 right half

rscr _ cml;
t _ 1c;
call[putIfuLH];
rscr _ cml;
t _ 1c;
call[putIfuRH];

* write all ones in location 1 left half

* write all ones in location 1 right half

ifuMScopeL:

t _ 0c;
call[getIfuLH];
t _ 0c;
call[getIfuRH];
t _ 1c;
call[getIfuLH];
t _ 1c;
call[getIfuRH];
branch[ifuMScopeL];

* read location zero

* read location one

* September 17, 1979 6:06 PM

iSingleStepTest

This test uses the Ifu's TEST register to single step the ifu through its paces. The program invokes subroutines that actually clock the Ifu and selected FFs to occur. The "microcode" that we are singlestepping would look like,

```

PcF_2;          * try a 2-byte instruction
IfuJump[0];
t_ID;          * somehow, we get back here after IfuJump
t # 10c;       * we expect 10 for alpha
skpif[ALU=0];

IDerr:
    error;
t_not(pCX');
t # (2c);
skpif[ALU=0];  * we set PcF to 2

PcXerr1:
    error;
t_ID;
t # (2c);      * we should get IL after alpha
skpif[ALU=0];

IDerr2:
    error;

PcF_3c;
IfuJump[0];   * somehow, we get back here after IfuJump
t_ID;
t # 20c;
skpif[ALU=0]; * expect 20=alpha, 100=beta

IDerr3:
    error;
t_not(pCX');
t # (3c);     * we set PcF to 3
skpif[ALU=0];
PcXerr2:
    error;

t_ID;
t # (100c);   * expect 100=beta
skpif[ALU=0];

IdErr4:
    error;
t_ID;
t # (3c);     * should get IL after beta
skpif[ALU=0];

IDerr5:
    error;

```

note that the microcode above is a GROSS OVERSIMPLIFICATION of what is happening. It provides "orientation" only!

The way this test works is to initialize the Ifu ram and then to loop 400B times calling the subroutine that actually tests, in single-step mode, the Ifu.

```

iSingleStepTest:          * single step, under Dorado processor control, ifu
    pushReturn[];
    call[setIusingInstrSet], t_1c;
    call[initIfuM1Thru3];
    t_400c, stkp+1;
    stack_t;

iSingleStepTestL:
    noop;
    call[itTestCase1];    * work happens here
    stack_(stack)-1;
    loopUntil[ALU<0, iSingleStepTestL];
    pReturnP[];

```

* November 24, 1978 3:07 PM

```
%*****
run the ifu thru one test case
%*****
```

```
itTestCase1: subroutine;
  pushReturn[];
```

```
itTestCase10:
  call[itResetSH];          * RESET
  t _ 1c;                  * use instruction set 1
  call[itMosFhSh];
```

```
%
```

OPCODE 1 in this sequence comes from location 2 in the Ifu Ram. It is a two byter. We must step the "2" into J. The ifudata is 10 for this sequence. We write PcF with 2.

```
%
```

```
  call[itNoopFhSh];
  call[itNewPcFhSh], t _ 2c; * NewPc_
  call[itNoopFhSh], t _ 2c; * keep new pc value on Bmux during tick
  call[itNoopFH];          * here for placement
  noop;
```

```
itTestCase11:
```

```
  call[itMemAckSHFH];      * MemAck Sh Fh
  call[itMakeFDSH];       * MakeF_D SH
  call[itNoopFH];
  call[itSetFGSH], t _ 2c; * value for J
  call[itSetFGFH], t _ 10c; * value for H
```

```
itTestCase12:
```

```
  call[itNoopSH];
  call[itNoopFH];
  expectedDispatch _ opAt15; * we expect ifujump to goto 10
  call[itIfuJumpSH];       * IfuJump[0]
  call[itNoopFH];
  call[itNextDataSH];     * _NextData
  t#(10C);
  skipif[ALU=0];
```

```
itCaseErr10:
```

```
  Breakpoint;
  noop;                   * here for placement
  call[itNoopFH];
  call[itGetPcXSH];       * _PcX
  t # (2c);
  skipif[ALU=0];
```

```
itCaseErr12:
```

```
  Breakpoint;            * pcx wasn't correct

  t_1c;                  * patch this instruction if you want
  Q_t;                   * to loop more than once
```

```
itCaseL1:
```

```
  noop;                  * Get Instr Length from _NextData
  call[itNoopFH];
  call[itNextDataSH];   * _NextData
  t#(2c);               * should get instruction length
  skipif[ALU=0];
```

```
itCaseErr13:
```

```
  Breakpoint;
  noop;
  t_(Q)-1;
  loopUntil[ALU=0, itCaseL1],Q_t; * check out instruction length a few times
```

```

* January 17, 1979 4:12 PM
%*****
OPCODE2 in this sequence comes from location 3 in the Ifu Ram. It is a three byter. We must step the
"3" into J. Likewise we step 20 and 100 into H. Allow the normal sequencing of the pipe to increment
PcFG, etc.
%*****
    call[itNoopFH];
    call[itMemAckShFh]; * MemAck Sh Fh
    call[itMakeFDackSh]; * MakeF_D, memAck Sh
    call[itMemAckFH]; * MemAck, Fh
    call[itSetFGFDSh], t _ 3c; * address word 3 in ifu ram (data for j)
    call[itSetFGFh], t _ 20c; * data for h
itTestCase21:
    call[itNoopSH];
    call[itSetFGFH], t _ 100c; *Third byte
    noop; * for placement

    expectedDispatch _ opAt15; * we expect ifujump to goto 10
    call[itIfuJumpSH]; * IfuJump[0]
    call[itNoopFH];

    call[itNextDataSH]; * _NextData
    t#(20C);
    skipif[ALU=0];
itCaseErr20:
    Breakpoint;
    noop; * here for placement

    call[itNoopFH];
    call[itGetPcXSH]; * _PcX for 3 byte opcode
    t # (4c);
    skipif[ALU=0];
itCaseErr22:
    Breakpoint; * pcx wasn't correct

    call[itNoopFH];
    call[itNextDataSH]; * _NextData for 3rd byte
    t # (100c);
    skipif[ALU=0];
itCaseErr23:
    Breakpoint;

    t_1c; * patch this instruction if you want
    Q_t; * to loop more than once
itCaseL2:
    noop; * top of _NextData loop that gets "IL"
    call[itNoopFH]; * when it does next datas
    call[itNextDataSH];
    t#(3c); * should get instruction length
    skipif[ALU=0];
itCaseErr24:
    Breakpoint;
    noop; * for placement
    t_(Q)-1;
    loopUntil[ALU=0, itCaseL2],Q_t; * check out instruction length a few times

itTestCase1Xit:
    returnP[];

```

```
* INSERT[D1ALU.MC];
* TITLE[Ifu2];
* INSERT[PREAMBLE.MC];
%
May 28, 1981 2:56 PM
    Change Bmux to BmuxRM to keep midas happy.
April 22, 1979 5:24 PM
    Move initIfuM0thru16 to ifuRamSubrs
April 19, 1979 10:28 AM
    Call resetIfu before using endIfuWd macro.
January 17, 1979 4:50 PM
    Use instruction set 1, init IfuM field by field.
```

%

%

CONTENTS

TEST DESCRIPTION

```
itTestCase2          Long sequence of opcodes
```

%

* November 30, 1978 9:27 AM

%

itTestCase2:

This is a long sequence of opcodes that we step thru the ifu by using the test register.

ExpectedDispatch remains valid across ifuJumps. ExpectedDispatch<0 ==> don't check where the dispatch came from.

%

itTestCase2: subroutine;

```

    saveReturn[iLink0];
    t _ 1c;
    call[itMosFhSh];                * use instruction set 1
itCase2Clk0:
    t _ (r0)-1;
    clockCount _ t;
    iTick[test.fh];                * 0
    iReset[test.sh];              * 1
    iTick[test.fh];                * 2
    iTick[test.sh];               * 3
    iNewPc[test.fh];              * 4;
    iNewPc[test.sh];              * 5
    BmuxRM _ 40c;
    iTick[test.fh];                * 6
    iTick[test.ShAck];            * 7
    noop;
itCase2Clk10:
    iTick[test.FhAck];            * 10
    iTick[test.ShAckFd];          * 11
    iTick[test.FhAck];            * 12
    iTick[test.ShAckFd, 1];       * 13
    iTick[test.FhAck, 1];         * 14
    iTick[test.ShFd];             * 15
    iTick[test.fh, 2];            * 16
    iTick[test.sh];               * 17
    noop;
itCase2Clk20:
    iTick[test.Fh];                * 20
    expectedDispatch _ opAt15;
    iJump[test.Sh];                * 21
    iTick[test.fh, 21];           * 22
    iJumpData[test.ShAck, 2];     * 23
    iTick[test.FhAck, 22];        * 24
    iJumpData[test.ShAckFd, 2];   * 25
    iTick[test.FhAck, 2];         * 26
    iJumpData[test.ShAckFd, 3];   * 27
itCase2Clk30:
    noop;
    iTick[test.FhAck, 23];        * 30
    iJumpData[test.ShFd];         * 31
    iTick[test.Fh, 24];           * 32
    iJump[test.Sh, 3];            * 33
    iTick[test.Fh, 25];           * 34
    Idata[test.ShAck];            * 35
    iTick[test.FhAck, 26];        * 36
    iJumpData[test.ShAckFd, 25];  * 37
    noop;
itCase2Clk40:
    iTick[test.FhAck, 25];        * 40
    iData[test.ShAckFd, 1];       * 41
    iTick[test.FhAck, 2];         * 42
    iJumpData[test.ShFd];         * 43
    iTick[test.Fh, 27];           * 44
    iJump[test.Sh, 2];            * 45
    iTick[test.Fh, 30];           * 46
    iJumpData[test.ShAck, 3];     * 47
    noop;
itCase2Clk50:
    iTick[test.FhAck, 30];        * 50

```



```

iJumpData[test.ShAckFd, 30];          * 51
iTick[test.FhAck, 30];                * 52
iJumpData[test.ShAckFd, 31];          * 53
iTick[test.FhAck, 31];                * 54
expectedDispatch _ cml;               * turn off dispatch address checking
iJump[test.ShFd];                     * 55
iTick[test.Fh, 32];                  * 56
iJump[test.Sh, 3];                    * 57
noop;

itCase2Clk60:
iTick[test.Fh, 33];                  * 60
iData[test.ShAck];                   * 61
iTick[test.FhAck, 34];                * 62
iJumpData[test.ShFd, 1];              * 63
iTick[test.Fh, 34];                  * 64
iData[test.ShAck, 3];                 * 65
iTick[test.FhAck, 35];                * 66
iJumpData[test.ShAckFd];              * 67
noop;

itCase2Clk70:
iTick[test.FhAck];                   * 70
iJumpData[test.ShFd];                 * 71
iTick[test.Fh, 36];                  * 72
iJump[test.Sh, 2];                   * 73
iTick[test.Fh, 37];                  * 74
iData[test.ShAck, 1];                 * 75
iTick[test.FhAck, 37];                * 76
iJumpData[test.ShAckFd, 37];          * 77
noop;

itCase2Clk100:
iTick[test.FhAck, 37];                * 100
iJumpData[test.ShAckFd, 1];           * 101
iTick[test.FhAck, 3];                 * 102
iJumpData[test.ShFd];                 * 103
iTick[test.Fh, 40];                   * 104
iJump[test.Sh];                       * 105
iTick[test.Fh, 41];                   * 106
iJumpData[test.ShAck, 2];             * 107
noop;

itCase2Clk110:
iTick[test.FhAck, 42];                * 110
iData[test.ShAckFd, 2];               * 111
iTick[test.FhAck, 2];                 * 112
iJumpData[test.ShAckFd, 1];           * 113
iTick[test.FhAck, 10];                * 114
iJumpData[test.ShFd];                 * 115
iTick[test.Fh, 3];                    * 116
iJump[test.Sh, 13];                   * 117
noop;

itCase2Clk120:
iTick[test.Fh, 13];                   * 120
iJumpData[test.ShAck, 7];             * 121
iTick[test.FhAck, 7];                 * 122
iJumpData[test.ShAckFd, 7];           * 123
iTick[test.FhAck, 7];                 * 124
iJump[test.ShAckFd, 7];               * 125
iTick[test.FhAck, 377];                * 126
iJump[test.ShFd, 1];                  * 127
noop;

itCase2Clk130:
iTick[test.Fh, 1];                    * 130
iJump[test.ShAck];                    * 131
iTick[test.FhAck];                     * 132
iJumpData[test.ShAckFd];               * 133
iTick[test.FhAck];                     * 134
iJump[test.ShAckFd, 13];               * 135
iTick[test.FhAck, 3];                  * 136
iJump[test.ShAckFd, 7];                * 137
noop;

itCase2Clk140:
iTick[test.FhAck, 7];                 * 140
iJump[test.ShAckFd, 1];                * 141

```

```

iTick[test.FhAck, 1];          * 142
iJumpData[test.ShAckFd, 3];   * 143
iTick[test.FhAck, 3];        * 144
iJump[test.ShAckFd, 1];      * 145
iTick[test.FhAck, 7];        * 146
iJump[test.ShFd];            * 147
noop;

itCase2Clk150:
iTick[test.Fh, 3];           * 150
iJump[test.Sh];              * 151
iTick[test.Fh];              * 152
iJumpData[test.ShAck, 10];   * 153
iTick[test.FhAck, 10];       * 154
iJumpData[test.ShAckFd, 10]; * 155
iTick[test.FhAck, 10];       * 156
iJump[test.ShAckFd, 10];     * 157
noop;

itCase2Clk160:
iTick[test.FhAck, 3];        * 160
iJump[test.ShFd];            * 161
iTick[test.Fh];              * 162
iJump[test.ShAck];           * 163
iTick[test.FhAck];           * 164
iJumpData[test.ShAckFd];     * 165
iTick[test.FhAck];           * 166
iJump[test.ShAckFd, 1];      * 167
noop;

itCase2Clk170:
iTick[test.FhAck];           * 170
iJump[test.ShAckFd, 12];     * 171
iTick[test.FhAck];           * 172
iJump[test.ShFd];            * 173
iTick[test.Fh];              * 174
iJumpData[test.ShAck];       * 175
iTick[test.FhAck, 12];       * 176
iJumpData[test.ShAckFd, 12]; * 177
noop;

itCase2Clk200:
iTick[test.FhAck, 12];       * 200
iJump[test.ShAckFd, 11];     * 201

iTick[test.FhAck, 12];       * 202
iJump[test.ShFd];            * 203
iTick[test.Fh];              * 204
iJump[test.ShAck];           * 205
iTick[test.FhAck, 43];       * 206
iJumpData[test.ShAckFd, 43]; * 207
noop;

itCase2Clk210:
iTick[test.FhAck, 43];       * 210
iJump[test.ShAckFd, 12];     * 211
iTick[test.FhAck, 11];       * 212
iJump[test.ShAckFd, 12];     * 213
iTick[test.FhAck, 12];       * 214
iJump[test.ShAckFd, 43];     * 215
iTick[test.FhAck, 43];       * 216
iJumpData[test.ShAckFd, 10]; * 217
noop;

itCase2Clk220:
iTick[test.FhAck, 10];       * 220
iJump[test.ShAckFd, 14];     * 221
iTick[test.FhAck, 12];       * 222
iJump[test.ShAckFd, 3];      * 223
iTick[test.FhAck, 3];        * 224
iJump[test.ShAckFd, 44];     * 225
iTick[test.FhAck, 44];       * 226
iJumpData[test.ShAckFd, 5];  * 227
noop;

itCase2Clk230:
iTick[test.FhAck, 5];        * 230
iJump[test.ShAckFd, 12];     * 231
iTick[test.FhAck, 14];       * 232

```

```

iJump[test.ShFd]; * 233
iTick[test.Fh]; * 234
iJump[test.ShAck]; * 235
iTick[test.FhAck, 44]; * 236
iJumpData[test.ShAckFd, 44]; * 237
noop;
itCase2Clk240:
iTick[test.FhAck, 44]; * 240
iJump[test.ShAckFd, 3]; * 241
iTick[test.FhAck, 43]; * 242
iJump[test.ShFd]; * 243
iTick[test.Fh, 44]; * 244
iJump[test.Sh, 10]; * 245
iTick[test.Fh, 5]; * 246
iData[test.ShAck, 1]; * 247
noop;
itCase2Clk250:
iTick[test.FhAck, 1]; * 250
iJumpData[test.ShAckFd, 5]; * 251
iTick[test.FhAck, 5]; * 252
iJumpData[test.ShAckFd, 10]; * 253
iTick[test.FhAck, 10]; * 254
iJump[test.ShAckFd, 2]; * 255
iTick[test.FhAck, 45]; * 256
iJump[test.ShFd, 7]; * 257
noop;
itCase2Clk260:
iTick[test.Fh, 373]; * 260
iJump[test.ShAck]; * 261
iTick[test.FhAck]; * 262
iJumpData[test.ShAckFd]; * 263
iTick[test.FhAck]; * 264
iJumpData[test.ShAckFd]; * 265
iTick[test.FhAck]; * 266
iJump[test.ShAckFd, 1]; * 267
noop;
itCase2Clk270:
iTick[test.FhAck, 5]; * 270
iJump[test.ShAckFd, 10]; * 271
iTick[test.FhAck, 12]; * 272
iJump[test.ShFd, 2]; * 273
iTick[test.Fh, 2]; * 274
iJumpData[test.ShAck, 7]; * 275
iTick[test.FhAck, 7]; * 276
iJumpData[test.ShAckFd, 7]; * 277
noop;
itCase2Clk300:
iTick[test.FhAck, 7]; *300
iJump[test.ShAckFd, 4]; *301
iTick[test.FhAck, 4]; *302
iJump[test.ShFd]; *303
iTick[test.Fh, 5]; *304
iJump[test.Sh]; *305
iTick[test.Fh, 46]; *306
iJumpData[test.ShAck, 5]; *307
noop;
itCase2Clk310:
iTick[test.FhAck, 47]; *310
iJumpData[test.ShAckFd, 5]; *311

iTick[test.FhAck,5]; *312
iData[test.ShAckFd,6]; *313
iTick[test.FhAck,50]; *314
iJumpData[test.ShFd]; *315
iTick[test.Fh,51]; *316
iData[test.Sh,6]; *317
noop;
itCase2Clk320:
iTick[test.Fh,6]; *320
iJumpData[test.Sh,6]; *321
iTick[test.Fh,52]; *322
iData[test.ShAck]; *323

```

```

    iTick[test.FhAck];          *324
    iData[test.ShFd];          *325
    iTick[test.Fh,53];         *326
    iJumpData[test.ShAck,4];   *327
    noop;
itCase2Clk330:
    iTick[test.FhAck,5];       *330
    iData[test.ShAckFd];       *331
    iTick[test.FhAck];         *332
    iData[test.ShFd];          *333
    iTick[test.Fh,54];         *334
    iJumpData[test.Sh,5];      *335
    iTick[test.Fh,55];         *336
    iJumpData[test.ShAck,6];   *337
    noop;
itCase2Clk340:
    iTick[test.FhAck,55];      *340
    iData[test.ShAckFd,55];    *341
    iTick[test.FhAck,55];      *342
    iJumpData[test.ShFd,56];   *343
    iTick[test.Fh,56];         *344
    iData[test.ShAck];         *345
    iTick[test.FhAck,57];      *346
    iJumpData[test.ShFd,6];    *347
    noop;
itCase2Clk350:
    iTick[test.Fh,60];         *350
    iData[test.ShAck];         *351
    iTick[test.FhAck];         *352
    iData[test.ShFd];          *353
    iTick[test.Fh,61];         *354
    iJumpData[test.Sh,4];      *355
    iTick[test.Fh,6];          *356
    iData[test.ShAck];         *357
    noop;
itCase2Clk360:
    iTick[test.FhAck];         *360
    iData[test.ShFd];          *361
    iTick[test.Fh,62];         *362
    iJumpData[test.ShAck];     *363
    iTick[test.FhAck,63];      *364
    iJumpData[test.ShFd,5];    *365
    iTick[test.Fh,63];         *366
    iData[test.ShAck,64];      *367
    noop;
itCase2Clk370:
    iTick[test.FhAck,64];      *370
    iData[test.ShFd,4];        *371
    iTick[test.Fh,64];         *372
    iJumpData[test.ShAck,4];   *373
    iTick[test.FhAck,6];       *374
    iData[test.ShAckFd];       *375
    iTick[test.FhAck];         *376
    iJumpData[test.ShFd];     *377
    noop;
itCase2Clk400:
    iTick[test.Fh,65];         *400
    iJumpData[test.Sh];        *401
    iTick[test.Fh,66];         *402
    iJumpData[test.ShAck,5];   *403
    iTick[test.FhAck,67];      *404
    iData[test.ShAckFd,5];     *405
    iTick[test.FhAck,5];       *406
    iData[test.ShFd,4];        *407
    noop;
itCase2Clk410:
    iTick[test.Fh,15];         *410
    iJumpData[test.ShAck];     *411
    iTick[test.FhAck,1];       *412
    iData[test.ShFd];          *413
    iTick[test.Fh];            *414
    iJumpData[test.Sh];        *415

```

```
    iTick[test.Fh,4];          *416
    iJumpData[test.ShAck];    *417
    noop;
itCase2Clk420:
    iTick[test.FhAck,1];      *420
    iTick[test.ShFd];         *421
    iTick[test.Fh];           *422
    iTick[test.Sh];           *423
    iTick[test.Fh];           *424
    iTick[test.Sh];           *425
    iTick[test.Fh];           *426
    iJump[test.Sh];           *427
    noop;
itCase2Clk430:
    iTick[test.Fh,4];          *430
    iJumpData[test.ShAck];    *431
    returnUsing[iLink0];
```

%*****

Table of Contents

Organized by Occurrence of subroutine in this Listing

Subroutine	Function

beginIfu3a:	"main control", it calls the test subroutines
ifuPcAlphaPipe:	Check the data bits of the Pc pipe and the Alpha pipe
ifuXqtTest:	execution test for Ifu

%	
May 20, 1981 3:10 PM	Construct this file out of ifu3.mc
May 19, 1981 4:50 PM	Begin adding iMiscEffects test (for RestoreStkP, IDFetch_, Fetch_ID, etc.)
February 1, 1980 8:05 PM	Fix miscellaneous comments, fix performance bug in iFastTest.
September 19, 1979 1:14 PM	Set expectedDispatch<0 in beginIfu3 -- for robustness.
September 18, 1979 11:07 AM	Cause tests that enableConditionalHold to disableConditionalTask as well -- seem to have mysterious bug that is associated with tasking. Fix some comments in IfuXqtTest.
September 17, 1979 6:32 PM	Fix beginIfu3 so that it is a subroutine that calls the test implemented here.
August 1, 1979 6:16 PM	Fix bug in ifuBrkInsTest -- failing to load BrkIns from left half of Bmux.
August 1, 1979 3:24 PM	Invert order of BrkIns_, Pcf_.
August 1, 1979 11:11 AM	Rearrange parts of IfuBrkInsTest for easier scope looping.
August 1, 1979 10:45 AM	Add noop to ifuBrkInsTest to accommodate placement problems.
August 1, 1979 10:36 AM	Fix omitted initialization of ifu memory in ifuBrkInsTest, missing skip instr, set instrset in ifu.
August 1, 1979 9:28 AM	Add the ifuBrkInsTest, add misc. comments.
July 3, 1979 2:27 AM	fix stack underflow error in resched test.
July 3, 1979 2:21 AM	Fix bug in iReschedTest -- clobbered the value that we load PcF with.
July 3, 1979 2:08 AM	Add noops for placement purposes. -- apparently setIUsingInstrSet causes some problems.
July 3, 1979 1:55 AM	Fix resched test, further, to handle the fact that reschedules don't occur after one IfuJump but after several.
July 3, 1979 12:37 AM	Cause iRamPEtest to enable the ramPE exception condition.
June 29, 1979 5:02 PM	Fix ifuChaos' ID checking to preserve ID in a register rather than xoring it with the expected value.
June 29, 1979 11:54 AM	Cause ifuChaos to reset memBase and memBX inside the ifuChaosL; check ;Rbase immediately after the IfuJump (in ifuChaos), since afterDispatch leaves POINTERS in a very fragile place (stk+1).
June 29, 1979 10:53 AM	Add missing "coreturn" at sicR2 (inside setIfuChaosRet).
June 24, 1979 6:40 PM	Change ifuChaos to use getIfuMBase, getIfuRBase for checking.
June 19, 1979 9:53 AM	Fix stack bug in iRamtest.
June 18, 1979 9:24 AM	Fix functional bugs in iRamPEtest (check for resched loc, reset rbase, membase, call setUsingInsrSet.
June 18, 1979 9:01 AM	Fix stack bugs in iRamPEtest.
June 17, 1979 4:30 PM	Add iRamPEtest.
June 6, 1979 10:30 AM	Chaos placement errors.
June 5, 1979 11:40 AM	Add calls to enableConditionalTask
June 1, 1979 6:16 PM	

Fix ifuChaos errors.
May 30, 1979 10:16 AM
Extend ifuPcAdderTest to cover regular opcodes and two byte jumps.
May 13, 1979 5:12 PM
Add ifuChaos.
May 3, 1979 3:18 PM
Move stack manipulation & vm fix-up code to different part of loop in ifuPcAlphaPipeTest.
May 3, 1979 2:56 PM
Add call to resetIfu after initIfuM1Thru3 in ifuPcAlphaPipe test.
May 1, 1979 5:08 PM
Add sundry comments.
April 27, 1979 5:19 PM
Add call to resetIfu from within ifuPcAlphaPipe test.
April 26, 1979 5:10 PM
Remove initIfuCache thru ifuTestLoop--moved into ifuTestSubrs.mc.
April 26, 1979 10:18 AM
Add code to use iUsingInstrSet during getCDbyte, putCDbyte; modify the code that puts opcodes in memory to place them in virtual order, Jtest.
April 25, 1979 12:52 PM
Make iFastTest accessible to normal testing. (no longer infinite loop).
April 24, 1979 3:49 PM
Add Pc adder test.
April 24, 1979 9:19 AM
Add ifuTestLoop.
April 24, 1979 8:46 AM
Add PcPipe test, and checkPcX routine & concomitant support code; add top level calls on various tests so that control no longer "falls through".
April 22, 1979 5:25 PM
Move ifuBackground thru ifuCountOnes into ifuRamSubrs.
April 20, 1979 6:09 PM
Fix missing "composeIfuWd[]" in getIfuHalt.
April 20, 1979 3:59 PM
Fix register clobbering bug in putIfuWd (another, different one).
April 19, 1979 4:59 PM
Fix stack manipulation bug in ifuBackGround.
April 19, 1979 11:38 AM
Add comments to ifuOpcode Test, diddle various things.
April 11, 1979 9:53 AM
Fix register clobbering bug in putIfuWd.
April 10, 1979 2:37 PM
Rearrange code to fit into functional categories -- prelude for division into different files.
April 4, 1979 11:28 PM
Add test controls to iterate thru the various tests in memory.
April 4, 1979 6:05 PM
Set memBX to zero, disallow wakeups on single memory errors, cause FastTest to invoke resetIfu subroutine.
March 1, 1979 7:17 PM
Add reschedule test.
March 1, 1979 6:26 PM
Change appendPause to appendHalts (3 words, 6 bytes), replace all pause opcodes with halts.
February 17, 1979 6:12 PM
Add noops for placement.
February 7, 1979 9:03 AM
Flush the first munch of ifu program from cache.
February 6, 1979 3:00 PM
Modify initialization to set All the words in Ifu memory to valid parity (pointing to an instruction that causes the processor to get an error).
February 6, 1979 2:15 PM
Fix bug in construction of program at 400 in memory; add noop after pcF_ before next ifuJump.
February 5, 1979 10:36 AM
Add fast opcodes, program to memory, etc.
January 29, 1979 10:52 AM
Add code to iMem that inits Ifu's code base.
January 26, 1979 4:18 PM
Change iMem to background memory so that the first memory referenes of the ifu code won't cause task 17 wakeups if there's garbage in storage. Remove mesa opcode stuff described below.
January 22, 1979 6:04 PM
Fix bugs in memory checking code...add code that loads ifu memory w/ mesa opcode values. This is strictly a hack to find out how to hand load ifumemory while the micro/microd/midas interface doesn't work.
January 22, 1979 11:02 AM
Fix mos_ bug, change default pcF, background storage with identity and cause the processor to make

memory references, and check for correctness.
January 22, 1979 9:44 AM
Add noops to remove very long ring: placement problem.
January 22, 1979 9:18 AM
Add incClock calls into ifuJump loop; fix problems w/ program at wd 100; add more comments.
January 20, 1979 8:28 AM
Fix putIfuWdPe0 bug, add more "programs" to cache.
January 17, 1979 2:59 PM
Execute from instruction set 3 (because exception addresses get instrSet' or'd into them).
January 10, 1979 4:50 PM
more opcodes into ifuMemory, more programs into cache, stack+1_ID
January 9, 1979 4:51 PM
icStartIfu loop to reset ifu, pcF before beginning icTopL loop
January 2, 1979 11:12 AM
longWait in initIfuCache to avoid memory system misses
%

* May 20, 1981 3:19 PM

beginIfu3a:

pushReturn[];
expectedDispatch_al;

call[ifuPcAlphaPipe]; * check the data bits of the pc pipe
call[ifuXqtTest]; * try various programs

endIfu3a:

returnP[];

* June 5, 1979 11:33 AM

%*+++++

ifuPcAlphaPipe

This test checks the data bits of the Pc pipe, and the data bits of the alpha/H pipe. Proceed by generating different patterns to set into PcFG. After waiting enough time for the Ifu Pipe to fill, perform an Ifu jump and then look at PcX. It should have the same value we originally set into PcFG. Use a two byte opcode and cycle through all possible bit patterns in a byte for the second (alpha) byte.

Don't forget to initialize the byte in memory pointed to by the pattern (and to undo setting that byte afterwards). We'll use instruction set 1, and make use of opcode = 1. This opcode gets used by the "test register" mode diagnostics that appear in Ifu1.mc.

```
call[initIfuCache];    --init the memory system for use by this test
call[initIfuMem1Thru3];
FOR pat IN PatX DO
  oldByte _ getByte[pat];    -- remember the original value where we put opcode
  oldByte2 _ getByte[pat+1];  -- original value where we put operand (alpha)

  putByte[pat,1];    -- write opcode = 1 at byte location = pat
  FOR bytePat IN [0..400B) DO
    putByte[pat+1, bytePat];
    PcFG _ pat;
    cnt _ 20;
    returnLink _ @ L1;

  Jump:
    IFUJUMP[0];

  L1:
    IF cameFromLocation # ExpectedLocation THEN
      IF cameFromLocation = notReadyLocation
        THEN GOTO Jump
      ELSE SIGNAL IfuPcAlphaJumpErr[];
    putByte[pat, oldByte];    -- restore memory to original value
    IF PcX # pat THEN SIGNAL PcPipeErr[];
    IF IfuData # bytePat THEN SIGNAL AlphaPipeErr[];
  ENDLLOOP;
  putByte[pat, oddByte];    -- restore the opcode byte
  putByte[pat+1, oddByte2]; -- restore the operand byte (alpha)
ENDLLOOP;
```

%*+++++

* June 5, 1979 11:33 AM

ifuPcAlphaPipe:

```
pushReturn[];
call[initIfuCache];    * initialize the memory system
call[resetIfu];
call[setIUsingInstrSet], t _ 1C;
noop;    * for placement
call[resetIfu];
call[initIfuM1Thru3];    * init instrset1, IFUM 1-3
call[resetIfu];    * clear breakpending (from writing IfuM)
call[enableConditionalTask];
call[iPat16];
```

ifuPcPipeL:

```
call[nextPat16];
skpif[ALU#0];
branch[ifuAfterPcPipe];
noop;    * for placement.
```

```
call[getCDbyte];    * enter w/ t = byte addr, rtns t =
stack+1_t;    * contents of that byte location.
```

```
call[getPat16];
stack+1_t;    * remember current pattern in stack
call[putCDbyte], rscr_2C;    * use opcode = "2"
```

```
call[iPat8];
```

ifuPcAlphaL:

```
call[nextPat8];    * see if there are more byte patterns
    * for the current PcF value
```

```

    skipif[ALU#0], rscr _ t;          * save byte pattern in rscr
    branch[afterIfuPcAlphaL];
    noop;

    call[putCDbyte], t _ (stack)+1;  * set "alpha" for this opcode. t = addr, rscr = byte
value.                               value.

    PcF _ stack;                     * now test the Pc pipe
    t _ 20C;
    cnt_t;
    call[ifuPcPipeSetJRet];          * try no more than 20B times
                                        * set up return link so we can get
ifuPcAlphaJump:
    IfuJump[0];                      * from our ifu jump.

ifuPcPipeCont:
    (rscr) # (opAt15);               * afterDispatch leaves rscr=location
    skipif[ALU#0];                   * where IfuJump took us.
    branch[ifuPcAlphaCheck];
    loopUntil[CNT=0&-1, ifuPcAlphaJump];

ifuPcAlphaJumpErr:
    error;                            * after 20B times we should succeed in
                                        % executing the opcode (the afterDispatch code takes
around 8 cycles). Since that hasn't happened, something is sadly amiss. RSCR=address where Ifu
dispatched the processor. Dispatches to notReady suggest the IFU can't talk to the memory system;
other exception conditions speak for themselves.
%

ifuPcAlphaCheck:
    t _ not(PcX');
    t # (stack);                      * see if PcX matches our pattern
    skipif[ALU=0];

ifuPcPipeErr:
    error;                            * PcX should be (stack) and is (t).
                                        * data path error or control error.

    call[getPat8];
    rscr _ ID;
    (rscr) # t;
    skipif[ALU=0];

ifuAlphaPipeErr:
    error;                            * rscr = ifuData, t = expected data

    t _ t-t;
    rscr _ (ID)+t;                    * force ID arithmetic path
    t _ (2c);
    t # (rscr);
    skipif[ALU=0];                    * should get instruction length (2)
ifuAlphaPipeErr2:
    error;                            * rscr = IfuData, t = bad bits. Expected 2

    branch[ifuPcAlphaL];

afterIfuPcAlphaL:

    stkp-1;                           * position stack to old byte
    t_stack&+1;                       * put the old value back into memory
    rscr _ t;
    call[putCDbyte], t_stack;
    branch[ifuPcPipeL], stkp-2;      * don't forget pattern & old byte are on stack.

%*****
                                ifuPcPipeSetJRet
This subroutine sets KLINK to point to an instruction that will GOTO ifuPcPipeCont. This is how we
give control back to the PcData routine after an IfuJump.
%*****
ifuPcPipeSetJRet:
    pushReturn[];
    call[ifuPcPipeSetJRet2];
    t _ link;
    klink _ t;
    returnP[];
ifuPcPipeSetJRet2:
    coreturn;
    branch[ifuPcPipeCont];

```

```
ifuAfterPcPipe:  
  top level;  
  call[disableConditionalTask];  
  returnP[];
```

* January 22, 1979 6:21 PM

%

ifuXqtTest Test the ifu using the memory system cache.

The IfuOpCode Test sequences through a group of small 'programs' written into memory. The outer loop, whose top is at **icNextTest** controls which program the test executes. The inner loop whose top is at **icNextTestIteration**, controls how many ifuJumps the test executes before starting the next test.

The inner loop has two phases. The first performs a simple IfuJump and then pushes ID onto the stack (the operator must look at the stack contents to decide if they (the contents) are correct. The second phase exercises the memory system as well as the ifu. It performs a fetch at the same time it executes an IfuJump. If the data returned from that fetch is not the same as the data returned from a subsequent fetch from the same address, an error has occurred. Then the test increments a roving pointer that is forced to stay in the interval [100000B..177777B]. The memory initialization code has written that section of memory with its address (ie., location 135555 contains 135555, etc). The test fetches the contents of memory pointed at by the roving pointer (and complains if the contents is not equal to the address), and stores the address in that location (ie., rewrites the data).

```

initIfuCache[];
resetIfu[];
rovingPtr _ 100000B
FOR test in TestX DO
  PcF_getCurrentProgramLocation[test];
  FOR ifuJumpX IN IterationCount DO
    ReturnLink _ @L1;          -- return to L1 after ifuJump
    IfuJump[0];

  L1:
    pushIDontoStack[];        -- eyeball it to see if it is ok
    popIDfromStack[];
    IF PcX < getBeginLoc[test] or PcX > getEndLoc[test] THEN ERROR;
    IF (ifuJumpX _ IfuJumpX+1) ~IN IterationCount THEN GOTO EXIT;

    ReturnLink _ @L2;        -- return to L2 after ifuJump
    IfuJump[0], tempAddr _ FETCH_ID;

  L2:
    tempV _ MD;
    pushIDontoStack[];        -- eyeball it to see if it is ok
    popIDfromStack[];
    IF PcX < getBeginLoc[test] or PcX > getEndLoc[test] THEN ERROR;
    FETCH_tempAddr;
    IF MD # tempV THEN ERROR;
    rovingPtr _ rovingPtr+17B; -- cross munch boundaries & everything
    IF rovingPtr>0 THEN rovingPtr _ 100000B; -- only look at top half of memory
    FETCH_rovingPtr;
    IF (MD#rovingPtr) THEN ERROR;
    STORE_rovingPtr, DBuf_rovingPtr;
    ENDL00P;

  EXIT:

```

%

* September 18, 1979 11:09 AM

```

ifuXqtTest:
  pushReturn[];
  call[initIfuCache];          * init the environment:
*   the memory system, the ifu memory, the cache
  call[setIUsingInstrSet], t _ 3C;

  noop;                        * space for breakpoint so we can easily patch code.
  call[enableConditionalTask];

icStartIfu:                * reset the ifu, set pcF
  call[resetIfu];
  t _ (r0)-1;
  clockCount _ t;

icSetInstrSet:
  t _ 101400C;                * instruction set 3
  MOS _ t;                    * patch to B_t???
  noop;
  call[initItest];

icNextTest:
  call[nextITest];
  skipif[ALU#0];
  branch[icTestDone];
  noop;
  PcF _ t;                    * start IFU at current location.

  call[initITestCount];

icNextTestIteration:
  call[nextITestCount];
  skipif[ALU#0];
  branch[afterTestCount];

icTopL:                    * top of ifuJump loop
  noop;                        * for placement
  call[incClock];
  call[icGetResumeLoc];
  t _ link;

  klink _ t;                  * afterdispatch uses klink as a return link
  ifuJump[0];                * control will eventually reach afterDispatch

icResume:
  noop;
  stack+1 _ ID;               * place for Severo to patch
  stack+1 _ ID;               * in the worst case (len=3, packed alpha,
  stack+1 _ ID;               * n#17) we must perform 5 _IDs to suck
  stack+1 _ ID;               * the pipe dry and get one IL.
  stack+1 _ ID;

icDownStack:
  noop;                        * can check the stack here
  stkp-4;
  stkp-1;

  call[checkPcX];             * see if PcX is incorrect (approximate
  skipif[ALU=0];             * check, only)
icPcXErr1:                 * PcX is not within the range of the
  error;                      % program we are currently executing. Ie., we're
executing an opcode from a memory location NOT in the program we should be executing.
T = current PcX, RSCR = program begin location, RSCR2 = program end location.
%

  call[nextITestCount];      * see if we're done executing this test
  skipif[ALU#0];
  branch[afterTestCount];

icSkipMemOps:
  noop;                        * noop to patch if required
  call[icGetResume2];
  t _ link;

```

```

klink _ t;

ifuJump[0], stack+1 _ (FETCH _ ID);
icAfterJump2:
  noop;
  stack+1 _ MD;
  stack+1 _ ID;
  stack+1 _ ID;
  stack+1 _ ID;
  stack+1 _ ID;
  stack+1 _ ID;
  stkp-4;
  stkp-1;

  call[checkPcX];
  skipif[ALU=0];
icPcXErr2:
  error;
an opcode from a memory location NOT in the program we should be executing.
T = current PcX, RSCR = program begin location, RSCR2 = program end location.
%

  t _ stack&-1;
  fetch _ stack;
  rscr _ MD;
  (rscr) # t;
  skipif[ALU=0];
icMdErr:
  error;

  stkp-1;

  rscr3 _ (rscr3) + (17C);
  skipif[ALU<0];
  rscr3 _ 100000c;

  fetch _ rscr3;
  t _ md;
  t # (rscr3);
  skipif[ALU=0];
icMd2Err:
  error;
  store _ rscr3, Dbuf _ rscr3;

  branch[icNextTestIteration];

afterTestCount:
  branch[icNextTest];
icTestDone:
  call[disableConditionalTask];
  returnP[];

icGetResumeLoc: subroutine;
  coreturn;
icResumeLoc:
  branch[icResume];
icGetResume2:
  coreturn;
icResume2:
  branch[icAfterjump2];

%*****
                                checkPcX
This subroutine returns ALU=0 IF PcX is within the range of the "current" program in memroy.
Otherwise it returns ALU#0. In either case, it returns with T=PcX, rscr = program begin location and
rscr2 = program end location.
%*****

checkPcX:
  pushReturn[];
  call[getTestBounds];
  t_not(PcX');
  * returns w/ rscr = begin location, RSCR2=
  * end location.

```

```
t-(rscr);
skpif[alu>=0];
branch[checkPcXFail];
(rscr2)-(t);
skpif[alu>=0];
branch[checkPcXFail];
returnPAndBranch[0C];
* no error
checkPcXFail:
returnPAndBranch[1C];
```


%*+++++*

Table of Contents

Organized by Occurrence of subroutine in this Listing

Subroutine	Function
+++++*	
beginIfu3b:	"main control", it calls the test subroutines
ifuPcAdderTest:	Force worst case carry propagation thru Ifu's adder
iMiscEffects:	Check miscellaneous side-effects, etc of IfuJump
iFastTest:	Fast execution test (ifuJumps happen immediately)
%*+++++*	
%	
June 18, 1981 10:13 AM	
Fix bug in iMisc8-14...using wrong linkage for ifujumps.	
June 17, 1981 3:43 PM	
Add test for Amux, Bmux checkout for IfuData	
May 27, 1981 11:42 AM	
Add Fetch_ID test to iMiscEffects.	
May 20, 1981 3:21 PM	
Construct this file out of ifu3.mc	
May 19, 1981 4:50 PM	
Begin adding iMiscEffects test (for RestoreStkP, IDFetch_, Fetch_ID, etc.)	
February 1, 1980 8:05 PM	
Fix miscellaneous comments, fix performance bug in iFastTest.	
September 19, 1979 1:14 PM	
Set expectedDispatch<0 in beginIfu3 -- for robustness.	
September 18, 1979 11:07 AM	
Cause tests that enableConditionalHold to disableConditionalTask as well -- seem to have mysterious bug that is associated with tasking. Fix some comments in IfuXqtTest.	
...	
July 3, 1979 2:08 AM	
Add noops for placement purposes. -- apparently setIUsingInstrSet causes some problems.	
...	
June 5, 1979 11:40 AM	
Add calls to enableConditionalTask	
...	
May 30, 1979 10:16 AM	
Extend ifuPcAdderTest to cover regular opcodes and two byte jumps.	
...	
April 26, 1979 5:10 PM	
Remove initIfuCache thru ifuTestLoop--moved into ifuTestSubrs.mc.	
...	
Make iFastTest accessible to normal testing. (no longer infinite loop).	
April 24, 1979 3:49 PM	
Add Pc adder test.	
%	

* May 20, 1981 3:21 PM

beginIfu3b:

pushReturn[];
expectedDispatch_al;

call[ifuPcAdderTest]; * find out if jumps involving many carry propogates work
call[iMiscEffects];
call[iFastTest];

endIfu3b:

returnP[];

* June 5, 1979 11:35 AM

IfuPcAdderTest

See if the logic that performs jumps works properly when worst case carry conditions obtain. The idea is to have a loop that bounces back and forth between location 0 and location 17777B. There are three test cases: use one byte jumps, use two byte jumps and use a jump and a one byte normal opcode (the jump gets the Pc to the 17777X range and the normal opcode gets the Pc to 0).

ifuPcAdderTest:

```
pushReturn[];
call[initIfuCache];
call[setIUsingInstrSet], t _ 3c;
noop; * for placement.
call[enableConditionalTask];
```

```
FETCH _ r0; * save contents of word 0, word 7777 in
stack+1_MD; * the stack.
t _ 7777C;
FETCH_t;
stack+1 _ MD;
```

ifuPcAdder2: % Test 2 involves two one byte jumps, one at location 0 and the other at location "-1":

Pc (byte address)	Contents of byte
0	one byte jump .-1
177777	one byte jump .+1

```
%
t _ (r0);
rscr _ jumpM1; * one byte (jump .-1) in byte 0,
call[putCDbyte]; * one byte (jump .+1) in byte 177777B
t _ 177777C; * remember, it is instruction set 3
rscr _ jump1;
call[putCDbyte];
t _ 2c; * begin with test 2.
stack+1 _ t; * there are 3 tests we'll perform. keep test indicator
in top of stack
```

ifuPcAdderReset: * Top of the test loop.

```
call[resetIfu];
PcF_r0;
call[ifuSetPcCont];
```

```
call[initITestCount];
```

ifuPcAdderL: * within each test, after an IfuJump, * control comes here

```
call[nextITestCount];
skpif[ALU#0];
branch[afterIfuPcAdder];
```

```
ifuJump[0];
```

ifuPcAdderCont:
branch[ifuPcAdderL];

ifuSetPcCont: * set KLINK so that this test can regain

```
pushReturn[];
call[ispc]; * control from afterDispatch.
t_link;
klink_t;
returnP[];
```

ispc:
coreturn;
branch[ifuPcAdderCont];

* February 1, 1980 8:04 PM

```

    top level;
afterIfuPcAdder:
    PD _ stack _ (stack)-1;          * see if we're done w/ tests;
    branch[ifuPcAdderXit, ALU<0], PD _ stack;
    branch[IfuPcAddr0, ALU#0];      * currently on test 1
ifuPcAdder1:                       % Test 1 involves a 2 byte jump at location zero to
location "-3" where there are 3 one byte, normal opcodes:

    Pc (byte address)                Contents of byte
+++++
    0                                2 byte jump opcode, sign extended
    1                                -3
    177775                            one byte, regular opcode
    177776                            one byte, regular opcode
    177777                            one byte, regular opcode
+++++
%
    t _ cm1;
    call[putCDbyte], rscr _ opNoop;
    t _ cm2;
    call[putCDbyte], rscr _ opNoop;
    t _ 177775C;                      * -3
    call[putCDbyte], rscr _ opNoop;
    t_A0;
    call[putCDbyte], rscr _ jumpML2;
    t _ 1c;
    call[putCDbyte], rscr _ 177775c;  * 2 byte jump to .-3 at location 0
    branch[ifuPcAdderReset];
ifuPcAddr0:                          % test 0 involves two 2 byte jumps between location 0
and location -2:

    Pc (byte address)                Contents of byte
+++++
    0                                2 byte jump, sign extended
    1                                "-2"
    177776                            2 byte jump
    177777                            "2"
+++++
%
    t _ A0;
    call[putCDbyte], rscr _ JumpML2;
    t _ 1c;
    call[putCDbyte], rscr _ cm2;
    t _ cm2;
    call[putCDbyte], rscr _ jumpL2;
    t _ cm1;
    call[putCDbyte], rscr _ 2c;
    branch[ifuPcAdderReset];

ifuPcAdderXit: * done with all our tests
    t _ 77777C, stack&-1; * fix stack which has current test number on it, and then restore clobbered
memory locations.
    STORE _ t, DBuf _ stack&-1;
    t_r0;
    STORE_t, DBuf _ stack&-1;
    call[disableConditionalTask];
    returnP[];

```

* May 27, 1981 11:41 AM

MiscEffects

This test checks that miscellaneous effects associated with an IFUjump work properly:

IDFetch_ must work.

iMiscEffects:

pushReturn[];

%

Test RestoreStkP.

The basic idea is to set StkP to one value, do an IFU jump, set StkP do a new value then to do RestoreStkP. Then StkP should be equal to the original value. We check for original value = 1 and original value=376. There is an implementation problem with this test: the afterDispatch code insists upon writing into the Stack at StkP+1! thus we preserve the value at Stack[2] which is the stack location that will be clobbered when afterdispatch gets control (and StkP=1). We don't care about the other location because it is so "high" in the stack that we never use it.

rscr3_ current stkp,
rscr4_ value of stack[2]

%

call[initIfuCache];
call[setIUsingInstrSet], t_3c;
t_ (r0)+1;
call[setPcF200], rscr4_ t+1;
rscr3_ Tioa&StkP;
call[setStkP], t_ rscr4;
branch[getiMisc1Cont], rscr4_Stack&-1;

* for placement.
* keep "2" in rscr for a moment
* stkp=1 when IfuJump happens
* rscr4_ Stack[2]

iMiscJ1:

IfuJump[];

IfuJump

* get here from getiMisc1Cont.
* processor should save StkP as it executes this

iMisc1Cont:

call[setStkP], t_ a1;
call[iMiscGetStkP], RestoreStkP;
t # (1c);
skpif[alu=0];

* get here from AfterDispatch.
* now set StkP to a different value
* processor restores old StkP
* original value was 1

iMiscErr1:

error;

* RestoreStkP did not restore us
* to zero, the value before ifujump

* Now try the same thing, but use different values for StkP.

call[setStkP], t_ 376C;
call[setPcF200];
branch[getiMisc2Cont];

iMiscJ2:

IfuJump[];

iMisc2Cont:

call[setStkP], t_a0;
call[iMiscGetStkP], RestoreStkP;
t#(376C);
skpif[alu=0];

* RestoreStkP did not restore us to
* 376B, the value before IFU jump.

iMiscErr2:

error;

* Restore the original value of Stack[2], StkP, then return.

call[setStkP], t_ 2c;
t_rscr4;
Stack_ t;
StkP_ rscr3;

* May 27, 1981 3:15 PM

%

Test Fetch_ID. We assume Store_Id will work if Fetch works. The idea is to execute an ifujump for a 3 byte instruction w/ alpha=377C and beta=0.

First we set BrLo=377 THEN perform Fetch_ID. The pipeVa ought to be 377+377. We check for correct ID, correct Va, correct Md.

Set BrLo to zero and perform Fetch_ID. Check PipeVa, Id, Md.

%

```

t_1000C;                * use byte location 1000
call[putCByteAddr];
call[putNextByte], t_ opNoopL3; * Use 3 byte instruction w/
call[putNextByte], t_ 377C;    * ENCODED CONSTANT=3 (throw it away)
call[putNextByte], t_ a0;      * alpha=377B, beta=0
call[setPcF1000];

t_377C;
BrLo_t;
call[longWait],t_10C;
branch[getiMisc3Cont];
iMiscJ3:
IfuJump;
iMisc3Cont:
t_Id;                    * throw away encoded constant.
t_ (Fetch_Id);
rscr_a0;
BrLo_rscr;
t#(377C);                * first id should be 377
skpif[ALU=0];
iMiscErr3a:
error;                   * got incorrect Id (t)

rscr_VaLo;
t_ t+t;
t#(rscr);
skpif[ALU=0];
iMiscErr3b:
error;                   * va is wrong in pipe. should be 377+377

Fetch_t, rscr_Md;
(Md)#(rscr);
skpif[ALU=0];
iMiscErr3c:
error;                   * data from memory is wrong

* Try it again, only use ID=beta=0
t_ (Fetch_Id);
Pd_t;                    * SECOND id should be 0
skpif[ALU=0];
iMiscErr4a:
error;                   * got incorrect Id (t)

rscr_VaLo;
skpif[ALU=0];
iMiscErr4b:
error;                   * va is wrong in pipe. should be 0

Fetch_t, rscr_Md;
(Md)#(rscr);
skpif[ALU=0];
iMiscErr4c:
error;                   * data from memory is wrong

```

* May 28, 1981 9:04 AM

%

Test Ifetch_ code.

Ifetch_ works by replacing the low 8 bits of the current base register with ID. Consequently the address computed by the memory system is the value BR[0..23],Id+Mar. Ie., the concatenation of the high bits of the current base register with Id plus the value on Mar.

Set the low 8 bits of BR to zero and non-zero values to see if Ifetch replaces those bits with Id.

%

```

        call[setPcF1000];                *This test uses BR=0
        branch[getiMisc5Cont];
iMiscJ5:
        IfuJump;
iMisc5Cont:
        t_a0, RisID;                    * throw away encoded constant
        IFetch_t;                       * Ifetch does not advance IFUdata
        t_Id;                            * thus, this ID is same as one for IFetch
        t#(377C);                       * first id should be 377
        skpif[ALU=0];
iMiscErr5a:
        error;                          * got incorrect Id (t)

        rscr_ VaLo;
        (rscr)#t;                       * t=id
        skpIF[alu=0];
iMiscErr5b:
        error;                          * Low bits of va should be 377=ID

        Fetch_t, rscr_ Md;
        (Md)#(rscr);
        skpif[ALU=0];
iMiscErr5c:
        error;

* test IFetch with brlo=377, ID=0 (means va should be zero)

        t_377c;
        BrLo_t, t_a0;
        IFetch_t;                       * IFetch does not advance IfuData
        t_Id;                            * thus, this ID is same as one for IFetch
        rscr_a0;
        BrLo_rscr, Pd_t;
        skpif[ALU=0];                   * ID should be zero!
iMiscErr6a:
        error;                          * got incorrect Id (t)

        rscr_ VaLo;
        (rscr)#t;                       * t=id
        skpIF[alu=0];
iMiscErr6b:
        error;                          * Low bits of va should be 0=ID

        Fetch_t, rscr_ Md;
        (Md)#(rscr);
        skpif[ALU=0];
iMiscErr6c:
        error;

```

* June 18, 1981 10:12 AM

% This test checks that RM & T, Amux and Bmux
The idea is to test the following sets of actions:

t_ A_ ID, B_ opposite bits from ID
RM_ A_ ID, B_ opposite bits from ID
T_ B_ ID, A_ opposite bits from ID
RM_ B_ ID, A_ opposite bits from ID

%

call[setPcf1000];
call[getiMisc7Cont];

iMiscJ7:

IfuJump;

iMisc7Cont:

rscr_ al;

Pd_ Id;

Pd_ Id;

T_ A_ Id, B_ rscr;

skpif[ALU=0];

* throw away encoded constant

* throw away alpha byte

* ID is all zeros

iMisc7Err1:

error;

* should have id=t=0

call[setPcf1000];
call[getiMisc8Cont];

iMiscJ8:

IfuJump;

iMisc8Cont:

t_ al;

Pd_ Id;

Pd_ Id;

rscr_ A_ Id, B_ rscr;

skpif[ALU=0];

* throw away encoded constant

* throw away alpha byte

* ID is all zeros

iMisc8Err1:

error;

* should have id=t=0

call[setPcf1000];
call[getiMisc9Cont];

iMiscJ9:

IfuJump;

iMisc9Cont:

rscr_ al;

Pd_ Id;

Pd_ Id;

T_ T, TisID, A_ rscr;

skpif[ALU=0];

* throw away encoded constant

* throw away alpha byte

* ID is all zeros

iMisc9Err1:

error;

* should have id=t=0

call[setPcf1000];
call[getiMisc10Cont];

iMiscJ10:

IfuJump;

iMisc10Cont:

t_ al;

Pd_ Id;

Pd_ Id;

rscr_ rscr, RisID, A_ rscr;

skpif[ALU=0];

* throw away encoded constant

* throw away alpha byte

* ID is all zeros

iMisc10Err1:

error;

* should have id=t=0

t_1000c;
call[putCDByteAddr];
call[putNextByte], t_ opSign3;
call[putNextByte], t_ 377c;

* use byte location 1000

* Use 3 byte instruction w/ sign bit, no encoded

* constant. alpha byte=377=>ID=-1=177777C

call[setPcf1000];
call[getiMisc11Cont];

iMiscJ11:

IfuJump;

iMisc11Cont:

rscr_ a0;


```
T_ A_ Id, B_ rscr;          * ID is all ones
t # (177777C);
skpif[ALU=0];
iMisc11Err1:              * should have id=t=all ones
error;

call[setPcf1000];
call[getiMisc12Cont];
iMiscJ12:
IfuJump;
iMisc12Cont:
t_ a0;
rscr_ A_ Id, B_ rscr;      * ID is all ones
(rscr) # (177777C);
skpif[ALU=0];
iMisc12Err1:              * should have id=t=all ones
error;

call[setPcf1000];
call[getiMisc13Cont];
iMiscJ13:
IfuJump;
iMisc13Cont:
rscr_ a0;
T_ T, TisId, A_ rscr;     * ID is all ones
t # (177777C);
skpif[ALU=0];
iMisc13Err1:              * should have id=t=0
error;

call[setPcf1000];
call[getiMisc14Cont];
iMiscJ14:
IfuJump;
iMisc14Cont:
t_ a0;
rscr_ rscr, RisID, A_ rscr; * ID is all ones
(rscr) # (177777C);
skpif[ALU=0];
iMisc14Err1:              * should have id=t=all ones
error;

returnP[];
```

* May 28, 1981 1:11 PM
* miscellaneous subroutines for iMisc tests.

```
setPcF200: subroutine;
    t_200C;
    PcF_t, RETURN;
setPcF1000: subroutine;
    t_1000c;
    PcF_t, RETURN;

setStkP: subroutine;
    StkP_t, RETURN;

    top level;
getiMisc1Cont:
    call[iMisc1Setup];
    klink_ link, branch[iMiscJ1];
iMisc1Setup: subroutine;
    coreturn;
    branch[iMisc1Cont];

    top level;
getiMisc2Cont:
    call[iMisc2Setup];
    klink_ link, branch[iMiscJ2];
iMisc2Setup: subroutine;
    coreturn;
    branch[iMisc2Cont];

    top level;
getiMisc3Cont:
    call[iMisc3Setup];
    klink_ link, branch[iMiscJ3];
iMisc3Setup: subroutine;
    coreturn;
    branch[iMisc3Cont];

    top level;
getiMisc5Cont:
    call[iMisc5Setup];
    klink_ link, branch[iMiscJ5];
iMisc5Setup: subroutine;
    coreturn;
    branch[iMisc5Cont];

    top level;
getiMisc7Cont:
    call[iMisc7Setup];
    klink_ link, branch[iMiscJ7];
iMisc7Setup: subroutine;
    coreturn;
    branch[iMisc7Cont];

    top level;
getiMisc8Cont:
    call[iMisc8Setup];
    klink_ link, branch[iMiscJ8];
iMisc8Setup: subroutine;
    coreturn;
    branch[iMisc8Cont];

    top level;
getiMisc9Cont:
    call[iMisc9Setup];
    klink_ link, branch[iMiscJ9];
iMisc9Setup: subroutine;
    coreturn;
    branch[iMisc9Cont];

    top level;
getiMisc10Cont:
    call[iMisc10Setup];
```

```
    klink_ link, branch[iMiscJ10];
iMisc10Setup: subroutine;
    coreturn;
    branch[iMisc10Cont];

    top level;
getiMisc11Cont:
    call[iMisc11Setup];
    klink_ link, branch[iMiscJ11];
iMisc11Setup: subroutine;
    coreturn;
    branch[iMisc11Cont];

    top level;
getiMisc12Cont:
    call[iMisc12Setup];
    klink_ link, branch[iMiscJ12];
iMisc12Setup: subroutine;
    coreturn;
    branch[iMisc12Cont];

    top level;
getiMisc13Cont:
    call[iMisc13Setup];
    klink_ link, branch[iMiscJ13];
iMisc13Setup: subroutine;
    coreturn;
    branch[iMisc13Cont];

    top level;
getiMisc14Cont:
    call[iMisc14Setup];
    klink_ link, branch[iMiscJ14];
iMisc14Setup: subroutine;
    coreturn;
    branch[iMisc14Cont];

iMiscGetStkP: subroutine;
    t_ Tioa&StkP;
    RETURN, t_ t and (377C);
```

* February 1, 1980 8:04 PM

Fast Test

This is a test that executes most ifuJumps very quickly. It "runs" the program written into word location 200 in the memory. The program it runs consists of a sequence of six one byte opcodes followed by "failing" conditional jump followed by a loop to the top of the program.

ifFastTest:

```

pushReturn[];
call[initIfuCache];
call[setIUsingInstrSet], t _ 3c;
noop;          * for placement.
call[enableConditionalTask];
ifuTest _ r0;
call[resetIfu];
t _ (r0)-1;
clockCount _ t;      * for future use.

t _ 101400C;          * use instruction set 3

```

iftSetInstrSet:

```

MOS _ t;

call[iftGetContinueLoc];
t _ link;
klink _ t;

```

iftFlush:

```

t _ 200c;
rscr _ (Flush_t) + t;      * flush our munch, compute byte addr.
t _ 100c;
call[longWait];          * wait arbitrary time for flush to finish.
t _ lastTestCountC;
cnt _ t;      * control how many times we'll loop.
stkp+1;      * increment stack so the opcodes won't

```

iftSetPcF: * clobber our return link.

```

PcF _ rscr;
noop;      * wait for it to take effect.

ifuJump[0];
error;

```

```

subroutine;;

```

iftGetContinueLoc:

```

coreturn;
t _ cnt;
PD_t;
skpif[ALU=0];      * quit if cnt=0
ifuJump[0];

```

afterIft:

```

top level;
call[disableConditionalTask];
pReturnP[];

```

%*****

Table of Contents

Organized by Occurrence of subroutine in this Listing

Subroutine	Function
beginIfu3c:	"main control", it calls the test subroutines
iReschedTest:	test Ifu reschedule for deferred, non-deferred operation
iRamPEtest:	test ramPe exception conditions
ifuChaos:	Test randomly constructed opCodes.
ifuBrkInsTest	Test execution of opcodes sourced from BrkIns

%*****

%

June 21, 1981 1:21 PM
 Fix ifuChaos to use POINTERS value saved by afterDispatch when checking Membase values

June 20, 1981 2:51 PM
 Have to set Jump bit in ifum for RamPe stuff, too.

June 20, 1981 11:06 AM
 Finally found an IFU hardware design bug that causes RamPE exception to be forgotten if ReschedPending is true and the opcode after the one w/ the bad parity has good parity. Fix in microcode is to keep the pause bit logically true. That prevents J from loading valid data after causing a ram pe.

June 18, 1981 1:53 PM
 Add new section to iRamPEtest that actually executes ram pe jumps.

June 18, 1981 10:24 AM
 Add a call to longWait in iRamPEtest to avoid notReady dispatch

May 20, 1981 3:04 PM
 Construct this file out of ifu3.mc

May 19, 1981 4:50 PM
 Begin adding iMiscEffects test (for RestoreStkP, IDFetch_, Fetch_ID, etc.)

...

Fix bug in ifuBrkInsTest -- failing to load BrkIns from left half of Bmux.

August 1, 1979 3:24 PM
 Invert order of BrkIns_, Pcf_.

August 1, 1979 11:11 AM
 Rearrange parts of IfuBrkInsTest for easier scope looping.

August 1, 1979 10:45 AM
 Add noop to ifuBrkInsTest to accommodate placement problems.

August 1, 1979 10:36 AM
 Fix omitted initialization of ifu memory in ifuBrkInsTest, missing skip instr, set instrset in ifu.

August 1, 1979 9:28 AM
 Add the ifuBrkInsTest, add misc. comments.

July 3, 1979 2:27 AM
 fix stack underflow error in resched test.

July 3, 1979 2:21 AM
 Fix bug in iReschedTest -- clobbered the value that we load PcF with.

July 3, 1979 2:08 AM
 Add noops for placement purposes. -- apparently setIUsingInstrSet causes some problems.

July 3, 1979 1:55 AM
 Fix resched test, further, to handle the fact that reschedules don't occur after one IfuJump but after several.

July 3, 1979 12:37 AM
 Cause iRamPEtest to enable the ramPE exception condition.

June 29, 1979 5:02 PM
 Fix ifuChaos' ID checking to preserve ID in a register rather than xoring it with the expected value.

June 29, 1979 11:54 AM
 Cause ifuChaos to reset memBase and memBX inside the ifuChaosL; check ;Rbase immediately after the IfuJump (in ifuChaos), since afterDispatch leaves POINTERS in a very fragile place (stk+1).

June 29, 1979 10:53 AM
 Add missing "coreturn" at sicR2 (inside setIfuChaosRet).

June 24, 1979 6:40 PM
 Change ifuChaos to use getIfuMBase, getIfuRBase for checking.

June 19, 1979 9:53 AM
 Fix stack bug in iRamtest.

June 18, 1979 9:24 AM
 Fix functional bugs in iRamPEtest (check for resched loc, reset rbase, membase, call setUsingInsrSet.

June 18, 1979 9:01 AM

Fix stack bugs in iRamPEtest.
June 17, 1979 4:30 PM
Add iRamPEtest.
June 6, 1979 10:30 AM
Chaos placement errors.
June 5, 1979 11:40 AM
Add calls to enableConditionalTask
June 1, 1979 6:16 PM
Fix ifuChaos errors.

...

May 13, 1979 5:12 PM
Add ifuChause.

...

March 1, 1979 7:17 PM
Add reschedule test.

...

%

*

beginIfu3c:

pushReturn[];

call[iReschedTest];

call[iRamPEtest];

call[ifuChaos];

call[ifuBrkInsTest];

endIfu3c:

returnP[];

* September 18, 1979 11:11 AM

%

RESCHED Test

This test checks that the ifu delays a normal resched by one successful ifuJump and that the reschedNow FF causes an immediate resched.

resetIfu[];

SetPcf[];

Set resched.

DO

Perform an Ifu Jump

If dispatchLocation=resched OR dispatchLocation=Unknown THEN ERROR

If dispatchLocation=notReady THEN LOOP;

EXIT;

ENDLOOP;

Perform an IfuJump

IF dispatchLocation#resched THEN ERROR

Set reschedNow

Perform an IfuJump

If dispatchLocation#resched THEN ERROR;

NoResched[];

IfuReset[];

%

iReschedTest: subroutine;

pushReturn[];

call[initIfuCache];

call[enableConditionalTask];

call[resetIfu];

call[setIUsingInstrSet], t _ 3c;

noop;

* for placement.

t _ exceptions.Resched3;

exceptionsMask _ t;

* this test accepts reschedule exceptions

t _ 2c;

iTestX _ t;

* So that checkPcX will work on test 2

t _ Test2MemByteLocC;

PcF _ t;

call[getIReschedCont1];

t _ link;

reschedule[];

klink _ t;

* vanilla program (test 2)

iReschedL:

IfuJump[0];

iReschedCont1:

(rscr) # (is3reschedLocC);

skpif[ALU#0];

* AfterDispatch leaves rscr= dispatch loc

iReschedErr1a:

error;

* resched dispatch occurred immediately

* after first ifuJump

(rscr) # (opAt15);

skpif[ALU#0];

branch[iResched2];

(rscr) # (is3notReadyLocC);

skpif[ALU#0];

branch[iReschedL];

* got one successful ifuJump. try again.

* not ready only reasonable thing left.

* try again.

iReschedErr1b:

error;

* unknown dispatch. wasn't (resched,

* notReady, expected opcode).

iResched2:

call[getIReschedCont2];

```

t _ link;
klink _ t;
cnt _ 2s;
iReschedL2:
  IfuJump[0];
iReschedCont2:
  t _ rscr;
  call[checkPcX], stack+1 _ t;
  branch[iReschedBad, ALU#0], t _ stack&-1;

  rscr _ t;
  (rscr) # (is3notReadyLocC);
  skipif[ALU#0];
  branch[iReschedL2];

  (rscr) # (is3reschedLocC);
  skipif[ALU#0];
  branch[iResched3Init];

  loopUntil[CNT=0&-1, iReschedL2];
iReschedErr2a:
  error;

iReschedBad:
  error;
T = return link for dispatch

* Normal, delayed resched worked.
* should get resched after small number
* of "normal" ifu dispatches.
* save rscr on stack, then
* see if PcX indicates we're in correct
* "program."
* Either we've come from not ready,
* a "normal" opcode, or a reschedule
* not ready yet. try again.
* perhaps we've had a reschedule
* finally got the reschedule we wanted.
* Small number of "normal" dispatches ok.
* Eventually should get resched dispatch.
* Only got normal opcode dispatches.
* According to PcX, the last IfuJump
* executed an opcode NOT a part of the test program.

```


* March 2, 1979 9:05 AM

* Test reschedNow

iResched3Init:

noReschedule[];
call[resetIfu];

* begin by clearing out the old one

ff256[103];
t _ 100c;
PcF _ t;
call[getIReschedCont3];
t _ link;
klink _ t;

* do reschedNow, setPcf and then IfuJump.

IResched3:

IfuJump[0];

iReschedCont3:

(rscr) # (is3reschedLocC);
skpif[ALU#0];
branch[iReschedTestDone];
(rscr) # (is3NotReadyLocC);
skpif[ALU#0];
branch[iResched3];

* try as long as its not ready

* after reschedNow we should get ONLY

* resched dispatch, not ready or

iReschedErr3a:

error;
* unexpected machine errors.

iReschedTestDone:

call[disableConditionalTask];
noReschedule[];
call[resetIfu];
returnP[];

* March 2, 1979 5:20 PM

getIReschedCont1:

coreturn;
branch[iReschedCont1];

getIReschedCont2:

coreturn;
branch[iReschedCont2];

getIReschedCont3:

coreturn;
branch[iReschedCont3];

* June 20, 1981 2:50 PM

iRamPeTest

Test the ramPE exception logic.

The first test generates real ram PEs and checks that the IFU actually dispatches the processor to the IFU ramPE location. It does this by setting the non-pe bits in the IFUM to all 0s or all 1s. Then it selects one of the three PE bits to have the WRONG value. When the processor executes an IFUJump, the IFU should dispatch the microcode to the appropriate ram pe location. USE reschedNow to guarantee that we don't go off to some arbitrary location if the ram pe logic fails. Ie., We set reschedNow and when the IFUJump occurs, the processor will end up at the resched location UNLESS there is a ram PE.

The second test employs reschedNow to force the IFU to dispatch the processor to a known location, regardless of the IM target address in the IfuM. This way many different bit patterns in IfuM can be tested, without having to dedicate code in IM for ifu entrypoints for all possible IFAD (IM target address, as stored in the Ram) values.

```

iRamPEtest: PROCEDURE =
  BEGIN
    FOR pat IN PatX DO
      lh _ getPat[patX];
      rh _ getPat[patX] AND (not ifu.PEmask);
      call[setRPEJret];
      FOR iAddr IN iAddrX DO
        putIfuWd[iAddr, lh, rh]; --- write patterh w/ good parity
        reschedNow[]; -- force immediate reschedule
        putCDbyte[0, iAddr]; -- write opcode into memory
        setInstrSet[iAddr];
        PcFG _ 0;
        IfuJump[];
      RPEcont:
        IF wentTo # Resched THEN ERROR;
        ENDLOOP; -- try next address in IfuM w/ the current pattern
      ENDLOOP; -- try a new pattern
    END;
  setRPEJret: PROCEDURE =
    BEGIN
      t _ @RPEcont0;
      klink_t;
      RETURN;
    RPEcont0:
      goto[RPEcont];
    END;

```

```

iRamPEtest: pushReturn[];
  call[initIfuCache];
  call[disableConditionalTask];
  set[wantExceptionsRPE, OR[exceptions.RamPe3!, exceptions.RamPe2!, exceptions.RamPe1!,
  exceptions.RamPe0!]];
  t _ (and[wantExceptionsRPE,177400]C);
  t _ t or (and[wantExceptionsRPE, 377]C);
  exceptionsMask _ t;
  call[resetIfu];

  call[iIAddr], stkp+1;          * we'll keep iAddr on stack;

```

%

This test checks the PE bits in IFUM. When ReschedPending is true the IFU hardware has a bug that allows it to forget ram parity errors if the opcode in the pipe following the one with the parity error has good parity. Consequently we force the Pause and Jump bits to be logically true (0). This way we guarantee that J will not fill up and that guarantees that the IFU will not forget the ram parity error. (When ReschedPending is true InstrAddrLd is true. This causes the IFU to reevaluate which exception dispatch it should generate. The ram parity error gets loaded into M during one cycle. Then, when J fills up with an instruction with valid parity, SawRamParityErr goes away and the exception dispatch changes from RamPE to Reschedule! Ugh. That was a hard two days.

The practical consequence of this is that when we background the IFUM w/ all ones, iDoRamPe will zero the Pause bit so we can use our reschedule trick. That means we must set PE2 appropriately for all ones except for the pause bit; ie., we must set pe2 to 1 unless we want to check that bit itself. The reschedule trick guarantees that we don't start executing some random IM location if the Ram PE logic

fails -- we go to the reschedule location instead.

```

%
iRamPeL:
    call[nextIaddr];
    branch[iRamPeLxit, ALU=0], stack_t;

    rscr_ 4c;
    rscr2_ a0;
iRamPeL0:
    call[iDoRamPE], t_ stack;

    rscr_ 5c;
    rscr2_ al;
    call[iDoRamPE], t_ stack;

    rscr_ 2c;
    rscr2_ a0;
iRamPeL1:
    call[iDoRamPE], t_ stack;

    rscr_ 3c;
    rscr2_ al;
    call[iDoRamPE], t_ stack;

    rscr_ 1c;
    rscr2_ a0;
iRamPeL2:
    call[iDoRamPE], t_ stack;

    rscr_ 1c;
    rscr2_ al;
    call[iDoRamPE], t_ stack;

    branch[iRamPeL];
iRamPeLxit:
    stkp-1;

set[wantExceptionsResched, OR[exceptions.Resched3!, exceptions.Resched2!, exceptions.Resched1!,
exceptions.Resched0!]];
    t _ (and[wantExceptionsResched,177400]C);
    t _ t or (and[wantExceptionsResched, 377]C);
    exceptionsMask _ t;
    call[resetIfu];

    call[iPat16];
iRamPePatL:
    call[nextPat16];
    skipif[ALU#0];
    branch[iRamPeXit];

    noop;
    call[getPat16], stkp+1;
    call[getPat16], stack&+1 _ t;
    stack&+1_ t and (NOT[ifu.peMask!]C);
    call[setIRamPeJret];

    call[iIAddr];
iRamPeAddrL:
    call[nextIaddr];
    branch[iRamPeAddrXit, ALU=0];
    stack_t;
    call[resetIfu];
    t _ (stack) or (mos.ifuCmmd);
    mos _ t;

*Set reschedNow and write current pattern into new IfuM location

    ff256[103], stkp-2;
    t _ stack&+1;
    rscr _ t;
    t _ stack&+1;

```

* ramPE is 3 bit field. test PE0 bit
* IFUM has all zeros
* testing PE0 bit

* ramPE is 3 bit field. test PE0 bit
* IFUM has all ones

* ramPE is 3 bit field. test PE1 bit
* IFUM has all zeros
* testing PE1 bit

* ramPE is 3 bit field. test PE1 bit
* IFUM has all ones

* ramPE is 3 bit field. test PE2 bit
* IFUM has all zeros
* testing PE2 bit

* ramPE is 3 bit field. test PE2 bit
* IFUM has all ones

* decrement stack where we kept iAddr

* for placement

* push stack again, keep iAddr at top

* TOS = iaddr, TOS-1 = rh, TOS-2 = LH

* set instruction set from current address

* terrible hack to get reschedNow

```

    rscr2 _ t;
mc[ifu.notTypeMask, not[or[b10!,b11!]]];
    rscr2_ (rscr2)AND (ifu.notTypeMask);
    call[putIfuWd], t _ stack;
* keep Pause and Jump logically true.
* rscr = lh, rscr2 = rh, t = address

*call setIUsingInstrSet so that putting bytes in memory works properly, then load "byte 0" with
current opcode (ifum address).

    t _ ldf[stack, 2, 10];
    call[setIUsingInstrSet];
    t _ stack;
    call[putCDbyte], rscr_ A0;
* isolate "instruction set bits
* make memByte[0] point to current ifu
* address
iRamPEsetPcF:
    PcF _ r0;
    call[longWait], t_30c;
    IfuJump[0];
* now begin ifu at location 0
* a few noops for pipe to fill
iRamPeCont:
    memBase _ 0s;
    RBASE _ rbase[defaultRegion];
    t _ (rscr) and (dispatchLocBaseMask);
    t # (reschedBaseLocC);
    skipif[ALU=0];
iRamPEDispatchErr:
    error;
* we should go to resched only. Any other
* dispatch is an error. We went to "rscr".

    branch[iRamPeAddrL];
* move stack down (below address), try next address

iRamPeAddrXit:
    branch[iRamPePatL], stkp-3;
* try next pattern
* pop stack since pattern loop pushes it.

iRamPeXit:
    t _ a0;
    exceptionsMask _ t;
    call[enableConditionalTask];
    returnP[];

setIRamPeJret:
    pushReturn[];
    call[sirpe2];
    t _ link;
    klink _ t;
    returnP[];

sirpe2:
    coreturn;
    branch[iRamPeCont];

```

* June 20, 1981 2:36 PM

% iDoRamPE

This subroutine executes an IfuJump that SHOULD get a ramPE.

Enter: rscr=bits for ifuRam PE field in IfuMemory

rscr2= background value for IFUM word

t= IFUM address we will test

Always write IFUM so that the pause bit is logically true (0)

%

iDoRamPE: subroutine;

pushReturn[];

stack+1_ t;

* SAVE: stack+1=ifuM address

rscr3_ rscr;

* rscr3= bits for IFUM PE field

rscr4_ rscr2;

* rscr4= background value for IFUM word

call[resetIfu];

t _ (stack) or (mos.ifuCmmd);

* set instruction set from current address

mos _ t;

*Set reschedNow and write current pattern into new IfuM location

ff256[103];

* terrible hack to get reschedNow

rscr_rscr4;

call[putIfuLH], t_stack;

t_ DpF[rscr3, 3, ifu.pe2Shift];

* position bits for pe

rscr4_ (rscr4) AND (NOT[ifu.peMask!]);

rscr_ (rscr4) OR (t);

rscr_ (rscr) and (ifu.notTypeMask);

* mask-out to make pause, jump true

call[putIfuRH], t_stack;

*call setIUsingInstrSet so that putting bytes in memory works properly, then load "byte 0" with current opcode (ifum address).

t _ ldf[stack, 2, 10];

* isolate "instruction set bits

call[setIUsingInstrSet];

t _ stack;

* make memByte[0] point to current ifu

rscr_ t;

* address

call[putCDbyte], t_ a0;

PcF _ r0;

* now begin ifu at location 0

call[longWait], t_30c;

* a few noops for pipe to fill

branch[iSetDoRamPEcont];

iDoRamPEJ:

IfuJump[0];

iDoRamPeCont:

memBase _ 0s;

RBASE _ rbase[defaultRegion];

t _ (rscr) and (dispatchLocBaseMask);

t # (ramPeBaseLocC);

skpif[ALU=0];

iDoRamPEDispatchErr:

error;

* we should go to ramPe only. Any other

* dispatch is an error. We went to "rscr".

* "stack" contains IFUM address that failed to generate ram pe.

* rscr3 contains background value for IFUM word

* rscr4 contains bits for ifuRam PE field in IfuMemory. Expect that these bits are set

* such that only one of the three bytes in the memory should generate a parity error.

* Eg., if rscr4=40000C then one of the bits associated w/ ifu.pe0 is bad

* IF rscr4= 3 one of the bits w/ ifu.pe0 is bad (the background value of IFUM is adjusted

* so that all non-PE bits in IFUM have same value and the test changes the PE bits.

pReturnP[];

iSetDoRamPEcont: top level;

call[iDoRamPeSetup];

klink_link, branch[iDoRamPEJ];

iDoRamPeSetup: subroutine;

coreturn;

branch[iDoRamPECont];

* May 13, 1979 4:11 PM

%*+++++

ifuChaos

Test generates and executes randomly constructed opcodes from randomly chosen memory locations. The test uses tow phases. The first phase involves writing the randomly generated opcode into IFUM and writing the value of Alpha and Beta into storage. The second phase of the test involves executing and checkning the ocode.

The first phase uses random numbers to choose these characteristics of the opcode:

Location of the opcode in storage

Location of the instruction in IFUM

The following characteristics of the instruction:

NX or none

Packed alpha or not

Sign extension or not

Instruction length

Instruction type

IM destination address

RBase and MemBase

The test always computes proper parity.

The test always uses the same IM target address

The first phases leaves a record of how it constructed the opcode for use by the second phase.

The second (checking) phase assures validity of the following:

IM target address (as well as it can)

PcX

_ID sequence

RBase

Mbase

%*+++++

```

* February 1, 1980 8:04 PM
%*****
ifuChaos: PROCEDURE =
    BEGIN
        -- Main Loop
        FOR testX IN ChaosTestRange DO -- execute a bunch of tests
            opCodeRecord _ makeIfuOpcodeRecord[];
            ifuChaosPhase1[opCodeRecord]; -- initialize everything for an opcode
            setIfuPc[opCodeRecord.ifuPc];
            ifuChaosPhase2[opCodeRecord]; -- execute and test the opcode
        ENDLOOP;
    END;
ifuChaosPhase1: PROCEDURE RETURNS [rec: ifuOpcodeRecord] =
    BEGIN
        -- Initialization Code

        putCDbyte[rec.ifuPc, rec.ifuOpcode]; -- write the opcode, alpha and beta into
        putCDbyte[rec.ifuPc+1, rec.ifuAlpha]; -- storage.
        putCDbyte[rec.ifuPc+2, rec.ifuBeta];

        resetIfu[];
        putIfuWd[rec.ifuOpcode, rec.instrHi, rec.instrLow]; -- write opcode into IfuMem
        resetIfu[];

    END;
ifuChaosPhase2: PROCEDURE [rec: ifuOpcodeRecord] =
    BEGIN
        -- Checking Code
        setIfuReturnLink[@phase2Return];
        count _ 25;

        phase2Loop:
            IfuJump[];
        phase2Return:
            IF (count = 0) THEN SIGNAL TooManyNotReadies[];
            IF (cameFrom # opAt15) THEN BEGIN
                IF (cameFrom # notReady) THEN
                    SIGNAL BadDispatchLoc[cameFrom];
                END;
                ELSE BEGIN
                    count _ count-1;
                    goto[phase2Loop];
                END;
            END;

        -- We get here when the test believes the opcode had dispatched correctly.
        memBaseAndRbase _ POINTERS[]; -- machine instruction rtns memBase,,Rbase
        expectMemBaseAndRbase _ getExpectedMemAndRbase[];
        IF getPcX[] # (rec.ifuPc) THEN SIGNAL badPcX[];
        IF memBaseAndRbase # expectMemBaseAndRbase THEN
            SIGNAL badPointers[memBaseAndRbase, expectMemBaseAndRbase];
        FOR i _ 1, i+1 UNTIL i=6 DO
            expectVal _ getExpectedID[rec];
            IF ifuDataCount # (expectedIfuDataCount[]) THEN SIGNAL BadIfuDataCount;
            IF expectVal # (gotValue _ getID[]) THEN SIGNAL BadID[i, expectVal, gotVal];
        ENDLOOP;
    END;
%*****

```

* June 21, 1981 1:20 PM

```

ifuChaos: subroutine;
  pushReturn[];
  call[initIfuChaosX];

ifuChaosL:
  call[nextIfuChaosX];
  skipif[ALU#0];
  branch[afterIfuChaos];
  noop;
  memBase _ 0s;
  memBX _ 0s;
  call[makeIfuOpcodeRecord];
  call[ifuChaosPhase1];

ifuChaosSetPc:
  call[enableConditionalTask];
  call[getIfuPc];
  PcF _ t;
  t _ 25C;
  cnt _ t;
  error
  call[setIfuChaosRet];

ifuChaosPhase2Cont:
  IfuJump[];

ifuChaosPhase2Ret:
  (rscr) # (opAt15);
  skipif[ALU#0];
  branch[ifuChaosPhase2Chk];
  noop;
  call[getIfuNotReadyLoc];
  t # (rscr);
  skipif[ALU=0];

ifuChaosDispatchErr:
  error;

  loopUntil[CNT=0&-1, ifuChaosPhase2Cont];

ifuChaosPhase2Chk:
  stkp+1;
  call[getIfuRbase], rscr _ stack;
  rscr _ (rscr) and (17C);
  t # (rscr);
  skipif[ALU=0];

ifuChaosRBaseErr:
  error;

  call[getIfuPc];
  rscr _ not(PcX');
  (rscr) # t;
  skipif[ALU=0];

ifuChaosPcXErr:
  error;

* check MemBase
  call[getIfuMbase];
  rscr _ stack&-1;
  rscr _ ldf[rscr, 5, 10];
  t # (rscr);
  skipif[ALU=0];

ifuChaosMBaseErr:
  error;

  RBASE _ rbase[defaultRegion];
  call[setMBase], t _ r0;

  cnt _ 6s;

```

* for placement

* undo the effects of any previous
* ifuJump.

* initialize the Ifu memory, storage.

* this controls number of "not readies" before an

* set KLINK

* for placement

* went to unexpected dispatch location.

* get the value of POINTERS that
* afterDispatch has saved.
* isolate rbase from POINTERS
* NOTE: stack= value of POINTERS saved by
* afterdispatch. Don't forget to decrement stkp!
* t = expected rbase, rscr = real rbase

* rscr = PcX, t = expected value for PcX

* scarf POINTERS and decrement StkP. See above.
* isolate membase from POINTERS

* t = expected rbase, rscr = real rbase

* Heretofor we've avoided depending upon
* these values.


```
ifuChaosIDL:
  noop;
  call[getExpectedID];
  rscr _ ID;
  PD _ (rscr) # t;
  skipif[ALU=0];
ifuChaosIDErr:
  error;
  loopUntil[CNT=0&-1, ifuChaosIDL];
  branch[ifuChaosL];

afterIfuChaos:
  call[disableConditionalTask];
  returnP[];

setIfuChaosRet:
  pushReturn[];
  call[sicR2];
  t _ link;
  klink _ t;
  returnP[];
sicR2:
  coreturn;
  branch[ifuChaosPhase2Ret];

* t = expected ID, rscr = actual ID
* (cnt-5) = number of IDs performed
* ifuChaosOpcode = the opcode
* ifuChaosPc = memory location
```

* September 18, 1979 11:13 AM

ifuBrkInsTest

Test the BrkIns register as it would be used for implementing breakpoints:

```
Load BrkIns
Set PcFG
Execute an IfuJump
```

If BrkIns is working properly, we'll execute the opcode loaded in BrkIns, otherwise we'll execute the opcode pointed to by PcFG.

ifuBrkInsTest:

```
pushReturn[];
call[initIfuCache];
call[enableConditionalTask];
noop; * for placement.
call[resetIfu];
call[setIUsingInstrSet], t _ 3c;
call[ifuSetInstrSet], t _ 3c;
call[putCByteAddr], t _ A0; * we'll set PcFG to zero
noop; * for placement.
call[appendHalts];
call[appendHalts]; * lots of halt opcodes which we should never execute!
call[setIfuBrkInsRet1];
```

ifuBrkInsScopeL:

```
ifuReset;
t _ 40c;
cnt _ t;
loopUntil[CNT=0&-1, .];
ifuReset;
call[justReturn]; * for patching into call[scopeTrigger],
t _ lshift[opNoop!, 10]C; * it's needed. Remember, BrkIns loads from
taskingOff; * left half of Bmux
PcF _ r0;
BrkIns _ t;
taskingOn;
cnt _ 17s;
```

ifuBrkInsL1:

```
IfuJump[0];
```

ifuBrkInsCont1:

```
(rscr) # (is3NotReadyLocC);
branch[ifuBrkInsChk, ALU#0];
noop; * for placement
loopUntil[CNT=0&-1, ifuBrkInsL1]; * got not ready. try again
```

ifuBrkInsNotReadyErr1:

```
error; * always went to not ready
* this should not have happened
```

ifuBrkInsChk:

```
(rscr) # (opAt15); * see if we went where we expected
skpif[ALU=0];
```

ifuBrkInsJmpErr1:

```
error; * expected to execute opNoop which would
* have taken us to opAt15
```

* now try it again, except flush the munch from the cache

```
Flush _ 0s;
call[justReturn]; * for patching into call[scopeTrigger],
t _ lshift[opNoop!, 10]C; * it's needed. Remember, BrkIns loads from
taskingOff; * left half of Bmux
PcF _ r0;
BrkIns _ t;
taskingOn;
t _ 100c;
cnt _ t;
call[setIfuBrkInsRet2];
loopUntil[CNT=0&-1, .]; * now wait right here for a long while
IfuJump[0];
```

ifuBrkInsCont2:

```
(rscr) # (opAt15); * should have gone directly to opAt15
skpif[ALU=0];
```

ifuBrkInstJumpErr2:

```
error; * went to "rscr" rather than opAt15
```

```
    call[disableConditionalTask];
    returnP[];

setIfuBrkInsRet1:
    pushReturn[];
    call[setIfuBrkIns1];
    t _ link;
    klink _ t;
    returnP[];
setIfuBrkIns1:
    coreturn;
    branch[ifuBrkInsCont1];

setIfuBrkInsRet2:
    pushReturn[];
    call[setIfuBrkIns2];
    t _ link;
    klink _ t;
    returnP[];
setIfuBrkIns2:
    coreturn;
    branch[ifuBrkInsCont2];

top level;
```

%*+++++

Table of Contents

Organized by Occurence of subroutine in this Listing

Subroutine	Function
+++++	
ifuChaosPhase1:	First half of current chaos test
makeIfuOpcodeRecord:	Creates the data for the current opcode
fixIfuInstr:	Modify ifuInstrRH so that it is legal.
makeExpectedID:	Returns currently expected value of IfuData
getIfuIDsequence:	Returns the encoding for expected _ID sequence
initIfuChaosX:	init ifu chaos index variable
nextIfuChaosX:	return next ifu chaos index variable (ALU=0 => done)
getIfuOpcode:	return current value of ifuOpcode
getIfuAlpha:	return current value of ifuAlpha
getIfuBeta:	return current value of ifuBeta
getIfuInstrLH:	return current value of ifuInstrLH
getIfuInstrRH:	return current value of ifuInstrRH
getIfuInstrIL:	return value of current ifuInstr's IL
getIfuRBase	return value of current ifuInstr's rbase
getIfuMbase	return value of current ifuInstr's membase
setIfuILfield:	modify IL field of an ifu instr in rscr; new value in t
setIfuTypefield:	modify type field of an ifu instr in rscr; new val in t
+++++	

%*+++++

%

September 17, 1979 11:20 AM

Fix bug in SetIfuTypeField -- apparently a lost fix, I remember doing this once before.

July 2, 1979 5:44 PM

Fix seqTable so that 1 byte opcodes w/ packed alpha And encoded constants return IL instead of NX, IL.

July 1, 1979 1:24 PM

Fix more bugs in getExpectedID -- beta is never sign extended.

July 1, 1979 12:55 PM

Fix some bugs in getExpectedID -- called getIfuInstrLH instead of getIfuInstrRH when trying to access sign extension bit.

June 30, 1979 6:30 PM

Fix several more bugs in makeIfuSequence.

June 30, 1979 5:39 PM

Change some labels in makeIfuSequence that still use "getIfu...", fix 2 bugs in computation of the index --using wrong register and failure to set a register. getExpectedID uses wrong mask to isolate an octal digit, sign extension logic is incorrect on IDisNX, IDisAlpha, IDisBeta.

June 29, 1979 5:19 PM

Change makeIfuOpcodeRecord's call to getIfuSequence (no longer exists) into call on makeIfuSequence.

June 29, 1979 5:04 PM

Rename getIfuSequence to makeIfuSequence (since it always reconstructs the ifuSequence value rather than returning the current one), and fix a bug in computing whether or not 10B should get or'd into the table index -- skip/sex bug. Change getExpectedID to access ifuSequence directly rather than calling getIfuSequence.

June 29, 1979 4:18 PM

Add ;s to end of data statements that define the seqTable -- this clobbered the first instruction of getExpectedId.

June 29, 1979 3:31 PM

Change incorrect call to ifuSetInstrSet in ifuChaosPhase1 to setIUsingInstrSet. Fix small bugs in getIfuRbase, getIfuMbase.

June 29, 1979 11:56 AM

ifuChaosPhase1 called getIfuPc where it should have called getIfuOpcode.

June 29, 1979 10:59 AM

Fix bug in makeIfuOpcodeRecord -- neglected to call getIfuSequence and set ifusequence.

June 29, 1979 10:03 AM

Fix bugs associated with fixIfuInstr. In particular, micro does not complain about the construct, "t _ not[ifu.ilMask];" It generates an instruction of "PD_3c" which is completely wrong. The proper syntax (I wrote it wrong) is "t _ not[ifu.ilMask!];C".

June 24, 1979 5:54 PM

Add get(IfuRBase, Mbase, MBx).

June 24, 1979 4:28 PM

Remove all uses of dpf that deposit into rm or t!

June 19, 1979 10:02 AM

Fix fixIfuInstr and makeIfuOpcodeRecord--used hi true ifadm in mior and used wrong register in

fii.

June 17, 1979 6:02 PM

Change location of sequence table to accommodate changes in postamble that accommodate ifu
entrypoint locations.

Fix placement problems in fixInstr, getIfuSequence, readSeqTable

June 5, 1979 10:23 AM

Remove placement errors, add new procedures.

June 1, 1979 6:14 PM

Remove makeifuopcoderecord from ifuchaosphasel so that an infinite loop can be constructed when a
particular opcode goes astray.

May 22, 1979 11:55 AM

Add ifuChaosPhasel, various other subroutines, makeIfuOpcodeRecord's microcode.

May 14, 1979 9:15 PM

Add makeIfuOpcodeRecord, getIfuIDsequence

May 2, 1979 10:53 AM

Construct this file and begin to write Mesa version of getExectedID.

%

* June 29, 1979 3:32 PM

ifuChaosPhase1

This subroutine uses the data record created by makeIfuOpcodeRecord to initialize all the raw data associated with the current test. It uses that data to write into IFUM, and storage.

The "opcode record", which is really a dedicated RM region, contains the following information:

ifuPc	Byte address in storage for the test opcode
ifuAlpha	First data byte (may not be used by instruction)
ifuBeta	Second data byte (may not be used by instruction)
ifuOpcode	The value of the opcode (ie., address, including instruction set bits, in the Ifu memory).
	<other values of no concern here>

ifuChaosPhase1:

pushReturn[];

Write opcode, alpha, beta into storage

```

call[getIfuOpcode];          * write opcode; begin by setting current
t _ ldf[t, 2, 10];          * instruction set indicator
call[setIUsingInstrSet];
call[getIfuOpcode];
call[getIfuPc], rscr _ t;
call[putCDbyte], stack+1 _ t; * push addr of opcode onto stack

call[getIfuAlpha];          * write Alpha
rscr _ t;
call[putCDbyte], t _ (stack)+1;

call[getIfuBeta];          * write Beta
rscr _ t;
call[putCDbyte], t _ (stack&-1) + (2c);
    
```

Write instruction into IfuMem

```

call[resetIfu];
call[getIfuInstrLH];
call[getIfuInstrRH], rscr _ t;
call[getIfuOpcode], rscr2 _ t;
call[putIfuWd];
call[resetIfu];
returnP[];
    
```

* June 29, 1979 5:20 PM

makeIfuOpcodeRecord

Create all the data associated with the new opcode. This includes constructing the high and low words for the Ifu Memory, choosing the opcode value and instruction set, choosing alpha and beta, and assuring that the instruction is legal (no 3 byte jumps, no instructions of type (jump and pause).

```
makeIfuOpcodeRecord: PROCEDURE RETURNS [rec: IfuOpcodeRecord] =
  BEGIN OPEN rec;
  ifuPc _ getRandom[];
  ifuAlpha _ getRandom[] and 377B;
  ifuBeta _ getRandom[] and 377B;
  ifuOpcode _ getRandom[] and 1777B;
  ifuInstrHi _ getRandom[] and ignoreIfAdMmask;
  ifuInstrLH _ getRandom[] and (ignoreParity);
  [ifuInstrHi, ifuInstrLH] _ fixIfuInstr[ifuInstrHi, ifuInstrLH];
  ifusequence _ getIfusequence[ifuInstrHi, ifuInstrLH];
  END;
```

makeIfuOpcodeRecord: subroutine;

pushReturn[];

```
getRandom[];          * 16-bit byte pc
ifuPc _ t;
```

```
getRandom[];          * Alpha and Beta
RBase _ rbase[ifuBeta];
ifuBeta _ ldf[t, 10, 0]; * extract right byte
ifuAlpha _ ldf[t, 10, 10]; * extract left byte
RBASE _ rbase[defaultRegion];
```

```
getRandom[];          * opcode, including instruction set bits
t _ t and (1777C);
ifuOpcode _ t;
```

The instruction, as it appears in the ifu memory, consists of two parts, the left and right halves. The left half contains the packed alpha bit (selected randomly) and the IM target address *that is always chosen to be the IM address, opAt15*. The right half contains the parity bits which this code masks out. Correct parity gets constructed when the words are written into the Ifu memory. The subroutine, fixIfuInstr, modifies il and it (length and type) to guarantee that the instruction is legal.

```
getRandom[];          * The actual instruction in Ifu Memory
rscr _ lshift[1, ifu.paShift]C; * isolate packed alpha
t _ t and (rscr);
```

```
rscr _ not (15c);      * default IfadM = opAt15
rscr _ (rscr) and (ifu.adMask); * Remember that Ifad bits are low true.
t _ t or (rscr);
ifuInstrLH _ t;
```

```
getRandom[];
t _ t and (not[ifu.peMask!]C); * don't set the parity bits in ifuRH
ifuInstrRH _ t;
call[fixIfuInstr];
```

```
call[makeIfuSequence];
ifuSequence _ t;
```

returnP[];

* June 29, 1979 10:08 AM

fixIfuInstr

This subroutine guarantees that the ifuOpcodeRecord is legal. A randomly chosen instruction may contain zero length instructions, 3 byte jumps, or instructions that are both of type pause and jump. These are illegal and must not occur.

```
fixIfuInstr: PROCEDURE[hi: IfuHiPart, low:ifuLowPart] RETURNS [newHi: IfuHiPart, newLow: ifuLowPart] =
  BEGIN
    newHi _ hi;
    newLow _ low;
    IF low.il = 0 THEN newLow.il _ 3;    -- zero length instructions are illegal
    IF (newLow.il = 3) AND (newLow.type = jump) THEN newLow.il _ 2;
    IF (newLow.type=jumpAndPause) THEN newLow.type _ jump;
  END;
```

fixIfuInstr:

```
  pushReturn[];
  call[getIfuInstrRH];
  call[getIfuIL], rscr _ t;          * save ifuInstrRH in rscr

  t _ t # (0c);                     * see if IL = 0
  skipif[ALU=0];
  branch[fixInstr2];               *If so, reset it to 3
  call[setIfuILfield], t _ 3c;     * use rscr = instr, t = new il
  IfuInstrRH _ t;
```

fixInstr2:

```
  call[getIfuIL];
  PD _ t # (3C);
  skipif[ALU=0];                   * (IL bits low true) IF IL=3 AND
  branch[fixInstr3];               * (IL is not 3)

  t _ ldf[rscr, ifu.jumpSize, ifu.jumpShift];
  PD _ t;                           * jump is low true
  branch[fixInstr3, ALU#0], t _ 2c; * ...IF JUMP THEN IL _ 2
  call[setIfuILfield];             * use rscr = instr, t = new il
  IfuInstrRH _ t;
```

fixInstr3:

```
  t _ ldf[rscr, ifu.typeSize, ifu.typeShift];
  PD _ t;                           * IF type = (JUMP AND PAUSE)
  branch[fixInstr4, ALU=0], t _ rscr;
```

fixInstrCont:

```
  IfuInstrRH _ t;
  returnP[];
  top level;
```

fixInstr4:

```
  call[setIfuTypefield], t _ (r0)+1;
  branch[fixInstrCont];
```


* July 2, 1979 5:44 PM

%*****

makeIfuSequence

Return a sequence of values (each value is an octal digit -- 3 bits) that describes the sequence of data that the Ifu delivers in response to "_ID". The rightmost octal digit encodes the type of the current IfuData. As the test performs _IDs, the routine that returns the expected IfuData and right shifts the sequence to keep in synch with the Ifu. Zero encodes "IL", so an unlimited number of _IDs can be properly simulated.

The table, **sequenceTable**, which is addressed by information about the current opcode, contains a sequence for all the relevant different possible sequences that aren't JUMP instructions. Remember that all jump instructions return ID = IL. The indices for the table consist of the concatenation of the following bits: <nx=17><pAlpha><ILO><IL1>. Thus there 20B entries that range [0..17B]. The bit nx=17 means that there's no encoded constant, pAlpha means packed alpha is true, and ILO, IL1 are the instruction length bits.

Here is a table that interprets the octal digits in a sequence:

- 0 => IL
- 1 => nx
- 2 => alpha
- 3 => alpha (packed alpha -- left side)
- 4 => alpha2 (packed alpha -- right side)
- 5 => beta
- 6,7 => ERROR

makeIfuSequence: PROCEDURE[hi:hiInstrPart, low: LowInstrPart] RETURNS[seq: ifuSequence] =

```

BEGIN
sequenceTable = ARRAY[0..17B] OF CARDINAL _
[
-- x refers to index (octal) in the table. The four bits that
-- address the entry are shown before the sequence. Illegal instructions
-- imply zero length instructions.
77777B, -- x=0,<0,0,0,0> illegal
01B, -- x=1,<0,0,0,ill> IL, nx
021B, -- x=2,<0,0,il0,0> IL, alpha, nx
0521B, -- x=3,<0,0,il0,ill> IL, beta, alpha, nx
77777B, -- x=4,<0,pA,0,0> illegal
00B, -- x=5,<0,pA,0,ill> IL USED to be 01, IL nx
0431B, -- x=6,<0,pA,il0,0> IL, pAlpha2, pAlpha1, nx
05431B, -- x=7,<0,pA,il0,ill> IL, beta, pAlpha2, pAlpha1, nx
77777B, -- x=10,<nx17,0,0,0> illegal
00B, -- x=11,<nx17,0,0,ill> IL
02B, -- x=12,<nx17,0,il0,0> IL, alpha
052B, -- x=13,<nx17,0,il0,ill> IL, beta, alpha
77777B, -- x=14,<nx17,pA,0,0> illegal
00B, --x=15,<nx17,pA,0,ill> IL
043B, -- x=16,<nx17,pA,il0,0> IL, alpha2, alphas
0543B, -- x=17,<nx17,pA,il0,ill> IL, beta, alpha2, alphas
];
IF low.type=jump THEN RETURN[seq_0];
IF low.il=0 THEN SIGNAL ZeroLengthInstr[hi, low];
x _ IF hi.nx=17=17 THEN 0 ELSE 10B; -- "nx17"
x _ x or (IF low.pAlpha THEN 4 ELSE 0); -- "pAlpha"
x _ x or (low.il AND 3); -- "ILOIL1"
seq _ sequenceTable[x];
IF seq = 77777 THEN SIGNAL illegalInstruction[hi, low];
RETURN;

```

%*****

makeIfuSequence:

```

pushReturn[];
call[getIfuInstrRH];
rscr _ t; * keep IfuInstrRH in T.
rscr2 _ ldf[t, ifu.jumpSize, ifu.jumpShift];
PD _ rscr2;
skpif[ALU#0];
branch[makeIfuSeqRet], t _ t-t; * jump instrs' seq = 0
noop; * for placement.
call[getIfuIL];
PD _ t;
skpif[alu#0];

```

```

makeIfuSeqErr0:                                * should never see zero length instruction
  error;

  t _ (rscr) and (ifu.nMask);                      * now construct in rscr2 the index into
  (t) # (17c);                                     * the table.
  skipif[ALU=0], rscr2 _ 10c;                       * assume nx = 17
  rscr2 _ a0;                                       * no, nx#17, there's an encoded constant

  call[getIfuInstrLH];                              * check for packed alpha
  t _ ldf[t, ifu.paSize, ifu.paShift];
  PD _ t;
  skipif[ALU=0];
  rscr2 _ (rscr2) or (4c);                          * packed alpha contributes 4B to index

  call[getIfuIL];                                   * IL contributes low 2 bits of index.
  t _ rscr2 _ (rscr2) or t;
  call[readseqTable];
makeIfuSeqRet:
  returnP[];

```

```

%*****
                                     readSeqTable
Actually reads the sequence table given t = index in table.
%*****

```

```

set[seqTableLoc, 5000];
mc[seqTableLocC, seqTableLoc];

```

readSeqTable:

```

  pushReturnAndT[];
  rscr _ seqTableLocC;
  t _ (stack) and (17c);
  t # (stack);
  skipif[ALU=0], t _ t rsh 1;
readSeqTableErr:                                * called with parameter > 7B
  error;
  t _ (rscr) + t;                                   * this is the word address
  branch[readSeqTableRH, r odd], stack;            * for placement
  noop;
  call[getImLh];
  PD _ t # (7c);
readSeqTableChk:
  skipif[ALU#0];
readSeqTableErr2:
  error;                                           * got '7' which is illegal

  pReturnP[];

```

readSeqTableRH:

```

  top level;
  call[getImRh];
  branch[readSeqTableChk], PD _ t # (7c);

```

```

%*****
                                     seqTable

```

The entries in this table must be read left to right in increasing IM address. Ie., seqTable[1] is 01 and seqTable[2] is 21.

```

%*****

```

data[(seqTable:

```

  lh[77777]          rh[01],          at[seqTableLoc, 0]]];
  data[(lh[21]       rh[521],         at[seqTableLoc, 1]]];
  data[(lh[77777]   rh[00],          at[seqTableLoc, 2]]];
  data[(lh[431]     rh[5431],         at[seqTableLoc, 3]]];
  data[(lh[77777]   rh[0],           at[seqTableLoc, 4]]];
  data[(lh[2]       rh[52],          at[seqTableLoc, 5]]];
  data[(lh[77777]   rh[0],           at[seqTableLoc, 6]]];
  data[(lh[43]     rh[543],          at[seqTableLoc, 7]]];

```

* July 1, 1979 1:26 PM

getExpectedID

Return the currently expected value for "ID". We use the value indicated by the right most octal digit in ifuSequence. A side-effect of calling this routine is that the ifuSequence value gets right shifted.

getExpectedID: PROCEDURE[rec: IfuOpcodeRecord] RETURNS[expect: CARDINAL] =

```

BEGIN
  type _ rec.ifuSequence AND 3b;
  rec.ifuSequence _ (rec.ifuSequence rshift 3);
  expect _ SELECT type FROM
    0 => rec.ifuInstrLH.il;
    1 => IF (rec.ifuInstrLH.sign # 0) AND ((rec.ifuInstrLH.nx and 10) # 0)
        THEN 17776B or rec.ifuInstrLH.nx
        ELSE rec.IfuInstrLow and 17B;
    2 => IF (rec.ifuInstrLH.sign # 0) AND ( (rec.ifuAlpha and 200B) # 0)
        THEN 177400B or rec.ifuAlpha
        ELSE rec.ifuAlpha;
    3 => ifuAlpha AND 17B;
    4 =>(ifuAlpha rshift 4) AND 17B;
    5 => rec.ifuBeta;
  default => SIGNAL illegalIDsequence;
  END;
END;
```

getExpectedID:

```

  pushReturn[];
  RBASE _ rbase[ifuSequence];
  t _ ifuSequence, RBASE _ rbase[defaultRegion];
  rscr _ t and (7c); * isolate current sequence digit, and
  t _ rsh[t, 3]; * replace current sequence with right
  call[putIfuSequence]; * shifted version
```

```

  bdispatch _ rscr; * go compute correct ID
  branch[expectedIDis];
  set[IDisTbl, 4200];
```

expectedIDis:

```

  branch[IDisIL], at[IDisTbl, 0];
  branch[IDisnx], at[IDisTbl, 1];
  branch[IDisAlpha], at[IDisTbl, 2];
  branch[IDisAlpha1], at[IDisTbl, 3];
  branch[IDisAlpha2], at[IDisTbl, 4];
  branch[IDisBeta], at[IDisTbl, 5];
  branch[IDisBad], at[IDisTbl, 6];
  branch[IDisBad], at[IDisTbl, 7];
```

IDisIL:

```

  call[getIfuIL];
  branch[getExpectedIDret];
```

IDisNX:

```

  call[getIfuInstrRH];
  branch[getExpectedIDret], t _ t and (ifu.nMask); * t = sign extended nx
```

IDisAlpha:

```

  call[getIfuInstrRH]; * isolate signX into rscr
  rscr _ ldf[t, 1, ifu.signShift];

  call[getIfuAlpha]; * get the value of alpha
  PD _ rscr;
  skipif[ALU#0];
  branch[getExpectedIDret]; * signX is not true, so NO sign extention

  t and (200C); * see if hi order bit of alpha is set.
  skipif[ALU=0];
  t _ t or (177400C); * We must extend the sign
  branch[getExpectedIDret];
```

IDisAlpha1:

```

  call[getIfuAlpha];
  branch[getExpectedIDret], t _ ldf[t, 4, 4];
```

IDisAlpha2:

```
call[getIfuAlpha];  
branch[getExpectedIDret], t _ ldf[t, 4, 0];
```

IDisBeta:

```
call[getIfuBeta];  
branch[getExpectedIDret];
```

IDisBad:

```
error;* rscr = sequence digit, ifuOpcodeRecord contains current info.
```

getExpectedIDret:

```
returnP[];
```

* June 29, 1979 3:33 PM

```

%*+++++
                                     initIfuChaosX
                                     nextIfuChaos

initIfuChaosX      initialize ifu chaos loop control
nextIfuChaosX      increment and return next ifuchaos loop value (ALU=0 => no more)
getIFUxxx          Where xxx are the pieces of the ifuOpcodeRecord
setIfuILfield     rscr = ifuRH, t = hi true, new value for il. Return t = new ifuRH
setIfuTypeField   rscr = ifuRH, t = hi true, new value fo type, Return t = new IfuRH.
%*+++++
initIfuChaosX: subroutine;
    t _ cml;
    return, ifuChaosX _ t;

nextIfuChaosX: subroutine;
    RBASE _ rbase[ifuChaosX];
    t _ ifuChaosX _ (ifuChaosX)+1, RBASE _ rbase[defaultRegion];
    t - (ifuChaosEndC);
    RETURN, t-(ifuChaosEndC);

getIfuOpcode: subroutine;
    RBASE _ rbase[ifuOpcode];
    return, t _ ifuOpcode, RBASE _ rbase[defaultRegion];

getIfuPc: subroutine;
    RBASE _ rbase[ifuPc];
    return, t _ ifuPc, RBASE _ rbase[defaultRegion];

getIfuAlpha: subroutine;
    RBASE _ rbase[ifuAlpha];
    return, t _ ifuAlpha, RBASE _ rbase[defaultRegion];

getIfuBeta: subroutine;
    RBASE _ rbase[ifuBeta];
    return, t _ ifuBeta, RBASE _ rbase[defaultRegion];

getIfuInstrLH: subroutine;
    RBASE _ rbase[ifuInstrLH];
    return, t _ ifuInstrLH, RBASE _ rbase[defaultRegion];

getIfuInstrRH: subroutine;
    RBASE _ rbase[ifuInstrRH];
    return, t _ ifuInstrRH, RBASE _ rbase[defaultRegion];

getIfuIL: subroutine;
    RBASE _ rbase[ifuInstrRH];
    t _ ldf[ifuInstrRH, ifu.ilSize, ifu.ilShift];
    t _ not(t), RBASE _ rbase[defaultRegion]; * return true sex
    return, t _ t and (3c);

getIfuRBase:
    pushReturn[];
    call[getIfuInstrRH];
    rscr2 _ ldf[t, 1, ifu.rbSelShift];
    PD _ rscr2;
    skipif[ALU=0], t _ (r0)+1; * (low true) 0 ==> rbase should be 1
    t _ a0; * (low true) 1 ==> rbase should be 0
    returnP[];

getIfuMBase:
    pushReturn[];
    call[getIfuInstrRH];
    rscr _ ldf[t, 2, ifu.memBLow2Shift];
    t _ ldf[t, 1, ifu.memBK34Shift];
    PD _ t;
    skipif[ALU=0], t _ rscr;
    t _ t or (34C); * add 2 bits
    returnP[];

putIfuSequence:
    return, IfuSequence _ t;

```

setIfuILfield:

```

pushReturnAndT[];                % enter w/ t = new IL bits, rt. justified,
and rscr = current value of ifuRH. Return T = new value for ifuRH. Original value in T is
treated as if it were high true! We invert the bits before modifying the value found in rscr. Thus,
this routine stores low true values into IL. Return with rscr unmodified.
%
```

```

stack _ not(stack);
stack _ (stack) and (ifu.ilMask);    * force into low true value

t _ not[ifu.ilMask!]C;                *remove current IL bits
t _ lcy[t,t, ifu.ILshift];
rscr _ t and (rscr);

stack _ lsh[stack, ifu.ilShift];     * position new il bits
t _ t or (stack&-1);                 * add new bits & decrement stack
returnP[];
```

setIfuTypefield:

```

pushReturnAndT[];                % enter w/ t = new type bits, rt. justified,
and rscr = current value of ifuRH. Return T = new value for ifuRH. Original value in T is treated as
if it were high true! We invert the bits before modifying the value found in rscr. Thus, this
routine stores low true values into type. Return with rscr unmodified.
%
```

```

stack _ not(stack);
stack _ (stack) and (ifu.typeMask);  * force into low true value.

t _ not[ifu.typeMask!]C;            *remove current type bits
t _ lcy[t,t, ifu.typeShift];
rscr _ t and (rscr);

t _ lsh[stack&-1, ifu.typeShift];   * position new type bits
t _ t or (rscr);                     * add new bits & decrement stack
returnP[];
```

```
TITLE[ifuDefs];
%*****
June 17, 1981 3:30 PM
    Add opSign3 for miscEffects test.
May 28, 1981 2:54 PM
    Change Bmux to BmuxRM to eliminate midas unhappiness.
February 1, 1980 5:48 PM
    Add patterns 5 & 6 for the pattern generator constants.
October 10, 1979 3:32 PM
    Add lastEventB
October 9, 1979 10:42 AM
    Add lastEventA
October 8, 1979 7:14 PM
    Add ctrFlag.
September 27, 1979 5:37 PM
    Add ctrLo, ctrHi, beginHi, beginLo
July 2, 1979 11:53 PM
    define exceptions.*, RM locations for exception (ifu exception) handling.
June 24, 1979 5:41 PM
    Add definition for ifu.typeMask.
June 20, 1979 8:14 AM
    Add new RM definitions (in defaultRbase) for ifu random address/data test
June 19, 1979 8:00 AM
    Add new 16 bit pattern constant.
June 18, 1979 9:35 AM
    Add constants to define"base" dispatch locations.
June 17, 1979 4:24 PM
    Move old "opIfad10" to "opIfad22" because of FGParity3 conflict
June 15, 1979 4:21 PM
    Add mos.ifuCmmd
June 7, 1979 5:34 PM
    Set defaultFlagsP to flags.conditionalP.
June 5, 1979 12:41 PM
    Update ifLength macro to use ILlength, ILMask, ILsize symbols.
June 5, 1979 11:29 AM
    Add rbsel and membk34 definitions for ifuchaos.
June 1, 1979 6:32 PM
    More IfuChaos glitches.
May 30, 1979 5:48 PM
    Glitches to accommodate ifuChaos.
May 30, 1979 11:00 AM
    Add MemoryOK for initMem, add new pattern limit for pat16.
May 23, 1979 10:42 AM
    Add rm definitions for ifu chaos.
May 1, 1979 6:58 PM
    Fix ifu.instrSet* for difference between 'geting' and 'seting'.
May 1, 1979 4:58 PM
    Add ifu.InstrSetSize, ifu.InstrSetShift, ifu.InstrSetMask.
April 26, 1979 8:54 AM
    Add iUsingInstrSet, pat8X.
April 25, 1979 12:53 PM
    Add fastexit, fastfetch, faststore definitions.
April 24, 1979 3:47 PM
    Remove dispatchOffset=r01, change iAddrX to r01.
April 23, 1979 9:02 AM
    Move opcode definitions (byte values) from ifu3 to here.
April 19, 1979 11:01 AM
    Increment lastTestCountC, create composeIfuWd macro.
April 5, 1979 12:05 AM
    Add rm & constants to control test execution.
March 2, 1979 9:12 AM
    Add constants for exception conditions.
February 15, 1979 9:00 AM
    Change ifum definition macros to save IM space.
January 26, 1979 5:26 PM
    Make ifum macros call subroutines rather than do all the work in-line.
January 22, 1979 8:27 PM
    Construct IfAdHi, ifAdLow to cope with ifadm values that aren't legal ff constants.
January 22, 1979 9:33 AM
    Make b0 the default trigger value
January 17, 1979 2:50 PM
    pat16End2C, instruction set parameter for endIfuWd
```

%*****

* October 10, 1979 3:34 PM

```
%*+++++
                                IFU RegisterDeclarations
                                Ifu Loop control constants
%*+++++
```

```
RMRegion[ifuSubrsRM];
rv[iLink0,0];
rv[iLink1, 0];
rv[iSubrScr, 0];
rv[triggerCount, 0];
rv[triggerValue, b15];
rv[triggerZero, 0];
rv[clockCount, 0];
rv[currentCDbyte,0];
rv[iTestX, 0];
rv[iTestCountX, 0];
rv[iCurrentTestLoc, 0];
rv[iUsingInstrSet, 0];
rv[pat8X, 0];
rv[ifuChaosX, 0];
rv[memoryOK, 0];
```

```
* current test index
* current iteration count for current test
```

```
RMRegion[ifuChaosRM];
rv[ifuPc, 0];
rv[ifuAlpha, 0];
rv[ifuBeta, 0];
rv[ifuOpcode, 0];
rv[ifuInstrLH, 0];
rv[ifuInstrRH, 0];
rv[ifuSequence, 0];
```

* The exception (Kfault, FGPe, Resched, RamPE) handling code uses these locations.

```
rv[exceptionsMask, 0];
rv[exceptionBit, 0];
rv[exceptionPointers, 0];
```

* Counter values

```
rv[ctrBeginLo, 0];
rv[ctrBeginHi, 0];
rv[ctrElapsedHi, 0];
rv[ctrElapsedLo, 0];
rv[lastEventA, 0];
rv[lastEventB, 0];
```

```
knowRbase[defaultRegion];
```

```
rm[expectedDispatch, IP[r10]];
```

```
rm[pat16, IP[rhigh1]];
rm[pat16X, IP[rm1]];
rm[IAddrX, IP[r01]];
rm[value1right, IP[rscr3]];
rm[value2left, IP[rscr4]];
rm[value2right, IP[stackPAddr]];
rm[bmuxRM, IP[rhigh1]];
rm[FGbits, IP[rm1]];
rm[iAddr1, IP[rhigh1]];
rm[iAddr2, IP[rm1]];
rm[value1left, IP[r01]];
rm[ctrLo, IP[rscr3]];
rm[ctrHi, IP[rscr4]];
rm[ctrFlag, IP[stackPAddr]];
```

```
br[br0,0];
```

```
mc[IfuEndAddrC, 2000];
mc[pat16End1C, 20];
mc[pat16End2C, 40];
mc[pat16End3C, 60];
mc[pat16End4C, 200];
mc[pat16End5C, 201];
mc[pat16End6C, 202];
mc[pat16EndAllC, 202];
mc[lastTestC, 4];
mc[lastTestCountC, 1000];
mc[ifuChaosEndC, 10000];
```

```
* last ifu address + 1
* IN [0..20)
* IN [0..40)
* IN [0..60)
* IN [0..100)
* IN [0..101)
* IN [0..102)
* IN [0..102)
* IN [0..4)
* IN [0..1000)
* IN [0..10000)
```

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

```
* conditional task and hold sim -- used by postamble.
```

```
mc[test0MemByteLocC, 0];
mc[test1MemByteLocC, 60];
mc[test2MemByteLocC, 120];
mc[test3MemByteLocC, 200];
```

```
mc[ifuLHmaskC, 3777]; * IFU mem left half on low 11 bits of Bmux
```

```
* Bit definitions for each exception: The exception handling code assumes "not ready" is always ok.
```

```
mc[exceptions.kfault3, b0]; mc[exceptions.FGPe3, b1];
mc[exceptions.resched3, b2]; mc[exceptions.RamPe3, b3];

mc[exceptions.kfault2, b4]; mc[exceptions.FGPe2, b5];
mc[exceptions.resched2, b6]; mc[exceptions.RamPe2, b7];

mc[exceptions.kfault1, b8]; mc[exceptions.FGPe1, b9];
mc[exceptions.resched1, b10]; mc[exceptions.RamPe1, b11];

mc[exceptions.kfault0, b12]; mc[exceptions.FGPe0, b13];
mc[exceptions.resched0, b14]; mc[exceptions.RamPe0, b15];
```

```

* June 17, 1979 4:25 PM
%*+++++
IFU TEST REGISTER Definitions
%*+++++
mc[test.parity, b8];
mc[test.fault, b9];
mc[test.memAck, b10];
mc[test.makeFD, b11];
mc[test.myfh, b12];
mc[test.mysh, b13];
mc[test.en, b14];
mc[test.myfh', 0];          * zero turns on fh, 1 bit turns it off
mc[test.mysh', 0];          * zero turns on sh, 1 bit turns it off
mc[test.FH, OR[test.en!, test.myfh'!, test.mysh!] ];          * enable, fh' (disable sh)
mc[test.SH, OR[test.en!, test.mysh'!, test.myfh!] ];          * enable, sh' (disable fh)
mc[test.FHSH, OR[test.en!, test.mysh'!, test.myfh'!] ];          * enable, fh', sh'

mc[test.FhAck, OR[test.Fh!, test.memAck!] ];          * enable, fh, memAck
mc[test.FhAckFd, OR[test.FhAck!, test.MakeFd!] ];          * enable, fh, memAck, MakeFd
mc[test.FhFd, OR[test.Fh!, test.MakeFd!] ];          * enable, fh, MakeFd

mc[test.ShAck, OR[test.Sh!, test.memAck!] ];          * enable, sh, memAck
mc[test.ShAckFd, OR[test.ShAck!, test.MakeFD!] ];          * enable, sh, memAck, MakeFd
mc[test.ShFd, OR[test.Sh!, test.MakeFd!] ];          * enable, sh, MakeFd

```

* June 18, 1979 9:40 AM

IFU Opcode and Entry Point constants

The **opAtxx** constants are the absolute IM address for the opcode whose IFADM field in the IFU ram has the value xx. Eg., if an opcode has an Ifadm value = 14, then opAt14 is the absolute location of that opcode's entry point in IM.

* These are entry point declarations

```
mc[opAt1, 177777]; * IFADM=1
mc[opAt13, 54]; * IFADM=13
mc[opAt14, 60]; * IFADM=14
mc[opAt15, 64]; * IFADM = 15
mc[opAt17, 74]; * IFADM = 17
mc[opAt63, 314]; * IFADM = 63
mc[opAt70, 340]; * IFADM = 70
mc[opAt22, 110]; * IFADM = 22
```

* entry point values for instruction set 3

```
mc[is3KfaultLocC, 0];
mc[is3FGParityLocC, 4];
mc[is3reschedLocC, 14];
mc[is3notReadyLocC, 34];
mc[is3ramPELocC, 74];
```

* Base entry point values (ie., the "instruction set bits" have been removed)

```
mc[KfaultBaseLocC, 0];
mc[FGParityBaseLocC, 4];
mc[ReschedBaseLocC, 14];
mc[notReadBaseLocC, 34];
mc[ramPeBaseLocC, 74];
mc[dispatchLocBaseMask, 77]; * base loc = (dispatchAddr) and (dispatchLocBaseMask)
```

* These are opcode declarations

```
mc[opPause, 0];
mc[jump0, 21]; * jump .
mc[jumpM1, 22]; * jump .-1
mc[jump1, 23]; * jump .+1
mc[opNoop, 24]; * no jump
mc[opNoopL2, 25]; * no jump length 2
mc[opNoopL3, 26]; * no jump, length 3
mc[jumpL2, 27]; * jump .+1, length 2
mc[jumpML2, 30]; * jump .-1, length 2
mc[fast1, 31]; * one byte, fast
mc[fastJfail, 32]; * 2 byte failing jump (resets pcF)
mc[fastJ, 33]; * succeed jump (1 or 2 bytes)
mc[fastMJ, 34]; * succeed jump to .-
mc[fastHalt, 35]; * pause instr that invokes "error"
mc[fastExit, 36]; * branch to afterDispatch IF CNT=0&-1
mc[fastFetch, 37]; * force stack <0, FETCH _ STACK
mc[fastStore, 40]; * check that MD=stack, mem[stack]_stack, stack_stack+1
mc[opSign3, 41]; * 3 byte, sign, no encoded constant
```

* June 24, 1979 5:41 PM

IFU field declarations

Defines fields mostly for IfuMemory.

mc[mos.ifuCmmd, b0]; * the bit that causes MOS_ to address Ifu rather than junk IO.

set[ifu.InstrSetSize, 2];
 set[ifu.SetInstrSetShift, 10];
 set[ifu.GetInstrSetShift, 13];
 mc[ifu.InstrSetMask, 3];

set[ifu.paShift, 12];
 set[ifu.paSize, 1];

set[ifu.AdShift, 0];
 set[ifu.AdHi2Shift, 10];
 set[ifu.AdLow8Shift, 0];
 mc[ifu.AdMask, 1777];

set[ifu.signShift, 17];
 set[ifu.signSize, 1];

set[ifu.pe0Shift, 16];
 set[ifu.pe1Shift, 15];
 set[ifu.pe2Shift, 14];
 mc[ifu.peMask, b1,b2,b3];

set[ifu.ILShift, 12];
 set[ifu.ILSize, 2];
 mc[ifu.ILMask, 3];

set[ifu.rbSelSize, 1];
 set[ifu.rbSelShift, 11];
 set[ifu.memBK34Size, 1];
 set[ifu.memBK34Shift, 10];

set[ifu.memBShift, 6];
 set[ifu.memBLow2Shift, 6];
 set[ifu.memBHi2Shift, 10];
 mc[ifu.memBMask, 17];
 mc[ifu.rbSelMask, b12];

set[ifu.typeShift, 4];
 set[ifu.typeSize, 2];
 mc[ifu.typeMask, 3];

set[ifu.pauseShift, 5];
 set[ifu.pauseSize, 1];
 mc[ifu.pauseMask, 1];

set[ifu.jumpShift, 4];
 mc[ifu.jumpMask, 1];
 set[ifu.jumpSize, 1];
 set[ifu.nShift, 0];

mc[ifu.nMask, 17];
 set[ifu.nSize, 4];

* June 17, 1979 4:25 PM

IFU Memory Construction Macros

These macros aid in the construction of Ifu memory values. Ie., they appear in the microcode where the Ifu memory is being initialized with opcodes. They work by constructing assembly-time values for each field in the Ifu memory. Then they construct a constant (done in **composeIfuWd**) that gets loaded into RM at run time. The macro **endIfuWd** understands the calling conventions for the subroutine **putIfuWd**.

m[newIfuWd, set[myPa, 0]
 set[myAd, 0]set[mySign, 0]set[myLength, 0]set[myMemB, 0]set[myPause, 0]set[myJump, 0]set[myN, 0]];
 m[ifPa, set[myPa, lshift[#1, ifu.paShift]]];

```
m[ifAd, set[myad, and[not[#1],ifu.adMask!]]];
m[ifAdHi, set[myad, or[myad, and[and[177400,not[#1]],ifu.adMask!]]]];
m[ifAdLow, set[myad, or[myad, and[and[377,not[#1]],ifu.adMask!]]]];
m[ifSign, set[mysign, lshift[#1,ifu.signShift]];
m[ifLength, set[mylength, lshift[and[not[#1],ifu.ILMask!],ifu.ILShift]];
m[ifMemB, set[mymemb, lshift[and[xor[ifu.rbSelMask!, #1],ifu.memBMask!],ifu.memBShift]];
m[ifPause, set[myPause, lshift[and[not[#1],ifu.pauseMask!],ifu.pauseShift]];
m[ifJump, set[myJump, lshift[and[not[#1],ifu.jumpMask!],ifu.jumpShift]];
m[ifN, set[myN, and[#1, ifu.nMask!]];
m[composeIfuWd,
  ilc[(rscr _ and[or[mypa,myad],177400]C)]
  ilc[(rscr _ (rscr) or(and[or[mypa,myad],377]C))]
  ilc[(rscr2 _ and[177400, or[mysign,mylength,mymemb,mypause,myjump,myn]]C)]
  ilc[(rscr2 _ (rscr2) or (and[377, or[mysign,mylength,mymemb,mypause,myjump,myn]]C))]
]
m[endIfuWd,
  composeIfuWd[]
  ilc[(t_and[#2,377]c)]
  ilc[(t_tor (and[lshift[#1,10],1400]C))]
  ilc[(call[putIfuWd])]
];
```

* January 2, 1979 1:46 PM

* These macros support the use of the test register

```

m[iTick,
    ilc[(FGbits _ (#1))]
    ife[#0,2,(ilc[(FGBits_(FGbits)OR(lshift[#2, 10]C))],)
    ilc[(call[itTick])]
    ];

m[iNewPc,
    ilc[(FGbits _ (#1))]
    ife[#0,2, ilc[(FGBits _ (FGbits) OR (lshift[#2, 10]C) )],]
    ilc[(call[itNewPc])]
    ];

m[iReset,
    ilc[(FGbits _ (#1))]
    ife[#0,2, ilc[(FGBits _ (FGbits) OR (lshift[#2, 10]C) )],]
    ilc[(call[itReset])]
    ];

m[iJump,
    ilc[(FGbits _ (#1))]
    ife[#0,2, ilc[(FGBits _ (FGbits) OR (lshift[#2, 10]C) )],]
    ilc[(call[itJump])]
    ];

m[iJumpData,
    ilc[(FGbits _ (#1))]
    ife[#0,2, ilc[(FGBits _ (FGbits) OR (lshift[#2, 10]C) )],]
    ilc[(call[itJumpData])]
    ];

m[iData,
    ilc[(FGbits _ (#1))]
    ife[#0,2, ilc[(FGBits _ (FGbits) OR (lshift[#2, 10]C) )],]
    ilc[(call[itData])]
    ];

* Initialize Ifu memory: initIfuM[location, leftHalf, rightHalf]

m[initIfuM,
    ilc[(t _ #1c)]
    ilc[(rscr_and[#2,177400]c)]
    ilc[(rscr_(rscr)OR(and[#2,377]c))]
    ilc[(call[putIfuLH])]
    ilc[(t _ #1c)]
    ilc[(rscr_and[#3,177400]c)]
    ilc[(rscr_(rscr)OR(and[#3,377]c))]
    ilc[(call[putIfuRH])]
    ];

```

```
* INSERT[D1ALU.MC];
* TITLE[Ifu];
* INSERT[PREAMBLE.MC];
```

```
%*****
CONTENTS
```

```
Routine DESCRIPTION
```

```
putIfuLH IfuMLH[t] _ rscr
putIfuRH IfuMRH[t] _ rscr
getIfuLH t IfuMLH[t]
getIfuRH t _ IfuMRH[t]
initIfuM1Thru3 Initialize Ifum locs 1-3 in current instruction set.
initIfuM1Thru16 Initialize IfuM locs 1-16B in current instruction set
ifuBackground Write halt opcodes into Ifu memory
getIfuHalt Returns t, rscr = ifuHalt opcode (for IfuMemory)
initIfuMemory Writes selected opcodes into Ifu memory
putIfuWd Writes (rscr,,rscr2) into IfuMemory at T
ifuAddParity Set parity bits for IfumLH, IfumRH (rscr, rscr2)
ifuCountOnes Count the number of one bits in t
```

```
%*****
```

```
%*****
```

```
June 17, 1981 3:34 PM
Add opSign3 instruction.
February 1, 1980 8:06 PM
Fix miscellaneous comments in fast opcodes init.
October 11, 1979 5:08 PM
Set Ifum[opPause] for inst ruction set 3 -- needed by event counters code.
June 15, 1979 4:35 PM
Change putIfuLH, putIfuRH, getIfuLH, getIfuRH to use mos.ifuCmmd
June 5, 1979 1:04 PM
Change symols ifu.Length* to ifu.IL*.
June 5, 1979 11:41 AM
Add calls to disableConditionalTask before writing Ifumemory, initializing regular memory system.
May 4, 1979 10:12 AM
Add calls to resetIfu after writing IfuM.
May 3, 1979 11:35 AM
Add missing composeIfuWd[], which should have been part of earlier fix to invoke putIfuWd
directly.
May 3, 1979 9:59 AM
Remove extraneous endIfuWd[] from initIfum0Thru16.
May 3, 1979 9:07 AM
Make initIfuM1Thru* use iUsingInstrSet and call putIfuWd directly.
April 25, 1979 1:00 PM
Add fastExit, fastFetch, fastStore opcodes.
April 23, 1979 9:33 AM
Change InitIfum1thru3 to use the standard macros.
April 22, 1979 5:11 PM
Create this file from code in ifu1-ifu3.
%*****
```


* January 15, 1979 4:54 PM

Ifu Memory subroutines

This code doesn't reset the Ifu. Caller Beware!

Enter with T = 10 bit IFU address. If writing ifum, rscr = 16 bit value
Exit W/ T = 16 bit value if reading IFUM.

Clobbers T

```

putIfuLH: subroutine;
    t _ t OR (mos.ifuCmmd);
    MOS _ t;
    t _ lsh[t, 10];
    brkins _ t;
    noop;
    IFUMLH _ rscr;
    return, B _ rscr;
    * t = addr, rscr = value
    * ifu data, not junk io
    * this sets the instruction set from B[6:7]
    * left justify the byte
    * set brkins
    * wait for the address to filter to ram
    * must keep data constant on B mux
    * for two cycles.

putIfuRH: subroutine;
    t _ t OR (mos.ifuCmmd);
    MOS _ t;
    t _ lsh[t, 10];
    brkins _ t;
    noop;
    IFUMRH _ rscr;
    return, B _ rscr;
    * t = addr, rscr = value
    * ifu data, not junk io
    * this sets the instruction set from B[6:7]
    * left justify the byte
    * set brkins
    * wait for the address to filter to ram
    * must keep data constant on B mux
    * for two cycles.

getIfuLH: subroutine;
    t _ t OR (mos.ifuCmmd);
    MOS _ t;
    t _ lsh[t, 10];
    brkins _ t;
    noop;
    return, t _ not(IFUMLH');
    * t = addr, returns lh value
    * ifu data, not junk io
    * this sets the instruction set from B[6:7]
    * left justify the byte
    * set brkins
    * wait for the address to filter to ram

getIfuRH: subroutine;
    t _ t OR (mos.ifuCmmd);
    MOS _ t;
    t _ lsh[t, 10];
    brkins _ t;
    noop;
    return, t _ not(IFUMRH');
    * t = addr, returns rh value
    * ifu data, not junk io
    * this sets the instruction set from B[6:7]
    * left justify the byte
    * set brkins
    * wait for the address to filter to ram

```

* June 5, 1979 11:42 AM

```

initIfuM1Thru3: subroutine;
  pushReturn[];

  call[disableConditionalTask];
  call[resetIfu];
  call[getIUsingInstrSet];
  t _ lsh[t, 10];          * position instr set value for writing ifuM
  stack+1 _ t;

  newIfuWd[];
  ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[0]; ifPause[0]; ifJump[0]; ifN[17];
  composeIfuWd[];
  t _ 1c;
  call[putIfuWd], t _ t or (stack);

  newIfuWd[];
  ifPa[0]; ifAd[15]; ifSign[0]; ifLength[2]; ifMemB[0]; ifPause[0]; ifJump[0]; ifN[17];
  composeIfuWd[];
  t _ 2c;
  call[putIfuWd], t _ t or (stack);

  newIfuWd[];
  ifPa[0]; ifAd[15]; ifSign[0]; ifLength[3]; ifMemB[0]; ifPause[0]; ifJump[0]; ifN[17];
  composeIfuWd[];
  t _ 3c;
  call[putIfuWd], t _ t or (stack);

%
  t_(1400C);
  rscr_t OR (375C);
  call[putIfuLH], t _ (r0)+1;          * putIfuLH doesn't clobber rscr!
  t _ (2c);                          * ifuLH[1] _ 1775B
  call[putIfuLH];                    * ifuLH[2] _ 1775B
  t _ (3c);                          * ifuLH[3] _ 1775B;
  call[putIfuLH];

  rscr3 _ (277C);
  t _ (rscr3) OR (75400c);
  rscr _ t;
  call[putIfuRH], t _ (r0)+1;          * ifuRH[1] _ 75677B
  t _ (rscr3) OR (73400c);
  rscr _ t;
  t _ (2c);
  call[putIfuRH];                    * ifuRH[2] _ 37677B
  t _ (rscr3) OR (61400C);
  rscr _ t;
  t _ (3c);
  call[putIfuRH];                    * ifuRH[3] _ 61677B

%
  call[resetIfu];
  writing IfuM.
  pReturnP[];
  * clear breakPending, which gets set as a result of

```

```

* May 4, 1979 10:14 AM
%*****
      Initialize IFU memory for test case 2
%*****
initIfuM0Thru16: subroutine;
      pushReturn[];
      call[disableConditionalTask];
      call[resetIfu];
      call[getIUsingInstrSet];
      t _ lsh[t, 10];          * position instr set value for writing ifum
      stack+1 _ t;

* opcode at iUsingInstrSet,,0: length=1,pause
      newIfuWd[];
      ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[3]; ifPause[1]; ifJump[0]; ifN[17];
      composeIfuWd[];
      call[putIfuWd], t _ stack;

* opcode at iUsingInstrSet,,1: length=1
      newIfuWd[];
      ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[3]; ifPause[0]; ifJump[0]; ifN[17];
      composeIfuWd[];
      t _ 1C;
      call[putIfuWd], t _ t or (stack);

* opcode at iUsingInstrSet,,2: length=2
      newIfuWd[];
      ifPa[0]; ifAd[15]; ifSign[0]; ifLength[2]; ifMemB[3]; ifPause[0]; ifJump[0]; ifN[17];
      composeIfuWd[];
      t _ 2C;
      call[putIfuWd], t _ t or (stack);

* opcode at iUsingInstrSet,,3: length=3
      newIfuWd[];
      ifPa[0]; ifAd[15]; ifSign[0]; ifLength[3]; ifMemB[3]; ifPause[0]; ifJump[0]; ifN[17];
      composeIfuWd[];
      t _ 3C;
      call[putIfuWd], t _ t or (stack);
      noop;          * for placement.

* opcode at iUsingInstrSet,,4: length=1, n=11
      newIfuWd[];
      ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[3]; ifPause[0]; ifJump[0]; ifN[11];
      composeIfuWd[];
      t _ 4C;
      call[putIfuWd], t _ t or (stack);

* opcode at iUsingInstrSet,,5: length=2 ,n=12
      newIfuWd[];
      ifPa[0]; ifAd[15]; ifSign[0]; ifLength[2]; ifMemB[3]; ifPause[0]; ifJump[0]; ifN[12];
      composeIfuWd[];
      t _ 5C;
      call[putIfuWd], t _ t or (stack);

* opcode at iUsingInstrSet,,6: length=3, n=13
      newIfuWd[];
      ifPa[0]; ifAd[15]; ifSign[0]; ifLength[3]; ifMemB[3]; ifPause[0]; ifJump[0]; ifN[13];
      composeIfuWd[];
      t _ 6C;
      call[putIfuWd], t _ t or (stack);

* opcode at iUsingInstrSet,,7: length=2, Sign, jump
      newIfuWd[];
      ifPa[0]; ifAd[15]; ifSign[1]; ifLength[2]; ifMemB[3]; ifPause[0]; ifJump[1]; ifN[17];
      composeIfuWd[];
      t _ 7C;
      call[putIfuWd], t _ t or (stack);
      noop;          * for placement.

* opcode at iUsingInstrSet,,10: length=2, jump

```

```

newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[0]; ifLength[2]; ifMemB[3]; ifPause[0]; ifJump[1]; ifN[17];
composeIfuWd[];
t _ 10C;
call[putIfuWd], t _ t or (stack);

* opcode at iUsingInstrSet,,11: length=1, jump
newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[3]; ifPause[0]; ifJump[1]; ifN[17];
composeIfuWd[];
t _ 11C;
call[putIfuWd], t _ t or (stack);

* opcode at iUsingInstrSet,,12: length=1, jump, n=2
newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[3]; ifPause[0]; ifJump[1]; ifN[2];
composeIfuWd[];
t _ 12C;
call[putIfuWd], t _ t or (stack);

* opcode at iUsingInstrSet,,13: length=1, jump, n=2
newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[3]; ifPause[0]; ifJump[1]; ifN[3];
composeIfuWd[];
t _ 13C;
call[putIfuWd], t _ t or (stack);
noop;                                     * for placement.

* opcode at iUsingInstrSet,,14: length=1, sign, jump, n=17
newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[1]; ifLength[1]; ifMemB[3]; ifPause[0]; ifJump[1]; ifN[17];
composeIfuWd[];
t _ 14C;
call[putIfuWd], t _ t or (stack);

* opcode at iUsingInstrSet,,15: length=1
newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[3]; ifPause[0]; ifJump[0]; ifN[17];
composeIfuWd[];
t _ 15C;
call[putIfuWd], t _ t or (stack);

* opcode at iUsingInstrSet,,16: length=1
newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[3]; ifPause[0]; ifJump[0]; ifN[17];
composeIfuWd[];
t _ 16C;
call[putIfuWd], t _ t or (stack);
call[resetIfu];                           * clear breakPending, which gets set as a result of
writing IfuM.
pReturnP[];

```

* June 5, 1979 11:43 AM

ifuBackground

This subroutine causes all the locations in the Ifu memory to contain parity-valid instructions that will cause the processor to get an error (invoke the ERROR code) if any of them gets executed. This detects the case where the ifu starts executing an uninitialized opcode.

ifuBackground:

pushReturn[];

call[disableConditionalTask];

call[resetIfu];

t _ 1777C;

cnt _ t;

ifuBackgroundL:

noop;

call[getIfuHalt];

* Returns rscr2 = ifumLH, rscr = ifumRH.

t_cnt;

call[putIfuWd], stack+1_t;

* t = address to write. Save cnt since

cnt _ stack&-1;

* putIfuWd will clobber it.

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

call[resetIfu];

* clear breakPending, which gets set as a result of

writing IfuM.

returnP[];

* April 19, 1979 11:10 AM

getIfuHalt

This subroutine returns rscr2 = ifumLH, rscr = ifumRH for an opcode that causes the processor to start executing code that invokes an error. Parity has not been computed.

getIfuHalt:

pushReturn[];

newIfuWd[];

ifPa[0]; ifAd[22]; ifSign[0]; ifLength[1]; ifMemB[0]; ifPause[1]; ifJump[0]; ifN[17];

composeIfuWd[];

t _ rscr2;

returnP[];

* June 17, 1981 3:34 PM

initIfuMemory

Initialize the Ifu memory to contain opcodes for the various tests used in the ifu diagnostic. The majority body of code in this subroutine consists of macros that construct integer values corresponding to the proper bits for the Ifu memory. The endIfuWd macro invokes the subroutine putIfuWd to write those values into the Ifu.

initIfuMemory: subroutine;

pushReturn[];
call[disableConditionalTask];
call[resetIfu];

* **pause** (length 1)

iicPause:

newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[0]; ifPause[1]; ifJump[0]; ifN[17];
endIfuWd[3, opPause!];

* **jump .** (length 0)

iicJump0:

newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[0]; ifPause[0]; ifJump[1]; ifN[0];
endIfuWd[3, jump0!];

Ifu memory

* write the constructed value into location jump0 of

* **jump.-1** (length=1)

iicJumpm1:

newIfuWd[];
ifPa[1]; ifAd[15]; ifSign[1]; ifLength[1]; ifMemB[0]; ifPause[0]; ifJump[1]; ifN[17];
endIfuWd[3, jumpM1!];

* **jump.+1** (length=1)

iicJp1:

newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[0]; ifPause[0]; ifJump[1]; ifN[1];
endIfuWd[3, jump1!];

* **opNoop** (length=1)

iicOpL1:

newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[0]; ifLength[1]; ifMemB[0]; ifPause[0]; ifJump[0]; ifN[4];
endIfuWd[3, opNoop!];

* **opNoopL2** (length=2)

iicopL2:

newIfuWd[];
ifPa[1]; ifAd[15]; ifSign[0]; ifLength[2]; ifMemB[0]; ifPause[0]; ifJump[0]; ifN[15];
endIfuWd[3, opNoopL2!];
noop; * for placement

* **opNoopL3** (length=3)

iicopL3:

newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[0]; ifLength[3]; ifMemB[0]; ifPause[0]; ifJump[0]; ifN[6];
endIfuWd[3, opNoopL3!];

* **jumpL2** (length=2)

iicJpL2:

newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[0]; ifLength[2]; ifMemB[0]; ifPause[0]; ifJump[1]; ifN[1];
endIfuWd[3, jumpL2!];

* **jumpML2** (length=2)

iicJmL2:

newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[1]; ifLength[2]; ifMemB[0]; ifPause[0]; ifJump[1]; ifN[17];
endIfuWd[3, jumpML2!];

* **opSign3** (length=3)

iicOpSign3:

```
newIfuWd[];
ifPa[0]; ifAd[15]; ifSign[1]; ifLength[3]; ifMemB[0]; ifPause[0]; ifJump[0]; ifN[17];
endIfuWd[3, opSign3!];
noop;                                * for placement
```

* February 1, 1980 8:07 PM

Fast OpCodes

The microcode that handles these opcodes executes very quickly; ie., it circumvents the arduous, main loop for the general tests by performing an IfuJump[0] "in line" in the microcode. This implies that tests constructed with these opcodes will never return unless the microcode explicitly notices the test should terminate.

* **fast1** (length=1)

iicFast1:

```
newIfuWd[];
ifPa[0]; ifAd[14]; ifSign[0]; ifLength[1]; ifMemB[0]; ifPause[0]; ifJump[0]; ifN[1];
endIfuWd[3, fast1!];
```

* **fastJfail** (length=2)

iicFastJfail:

```
newIfuWd[];
ifPa[0]; ifAd[13]; ifSign[0]; ifLength[2]; ifMemB[0]; ifPause[0]; ifJump[1]; ifN[17];
endIfuWd[3, fastJfail!];
noop;                                * for placement
```

* **fastJ** (length=2)

iicFastJ:

```
newIfuWd[];
ifPa[0]; ifAd[16]; ifSign[0]; ifLength[2]; ifMemB[0]; ifPause[0]; ifJump[1]; ifN[17];
endIfuWd[3, fastJ!];
```

* **fastMJ** (length=2)

iicFastMJ:

```
newIfuWd[];
ifPa[0]; ifAd[16]; ifSign[1]; ifLength[2]; ifMemB[0]; ifPause[0]; ifJump[1]; ifN[17];
endIfuWd[3, fastMJ!];
noop;                                * for placement
```

* **fastHalt** (length=2)

iicfastHalt:

```
newIfuWd[];
ifPa[0]; ifAd[22]; ifSign[0]; ifLength[1]; ifMemB[0]; ifPause[1]; ifJump[0]; ifN[17];
endIfuWd[3, fastHalt!];
noop;                                * for placement
```

* **fastExit** (length=1)

iicfastExit:

```
newIfuWd[];
ifPa[0]; ifAd[10]; ifSign[0]; ifLength[1]; ifMemB[0]; ifPause[0]; ifJump[0]; ifN[17];
endIfuWd[3, fastExit!];
noop;                                * for placement
```

* **fastFetch** (length=1)

iicfastFetch:

```
newIfuWd[];
ifPa[0]; ifAd[11]; ifSign[0]; ifLength[1]; ifMemB[0]; ifPause[0]; ifJump[0]; ifN[17];
endIfuWd[3, fastFetch!];
noop;                                * for placement
```

* **fastStore** (length=1)

iicfastStore:

```
newIfuWd[];
ifPa[0]; ifAd[12]; ifSign[0]; ifLength[1]; ifMemB[0]; ifPause[0]; ifJump[0]; ifN[17];
endIfuWd[3, fastStore!];
noop;                                * for placement
```

call[resetIfu];

writing IfuM.

returnP[];

* clear breakPending, which gets set as a result of

* April 20, 1979 4:05 PM

putIfuWd

Write both left and right half of IFUM.

Enter with RSCR=ifuLH, RSCR2=ifuRH, T= ifu Address. Caller is responsible for resetting the Ifu before calling this routine! Call **ifuAddParity** to correctly set the parity bits for this word of IFUM. Use **putIfuLh** and **putIfuRh** to write into IFUM.

putIfuWd: subroutine;

pushReturnAndT[];

call[ifuAddParity];

* compute correct parity bits.

* now write rscr into IFUMLH and rscr2 into IFUMRH. Address is in Stack

putIfuWdFinishUp:

call[putIfuLh], t _ stack;

* t _ ifu address, rscr = value for LH

rscr _ rscr2;

call[putIfuRH], t _ stack;

* t = addr, rscr = value for RH

pReturnP[];

* February 6, 1979 2:54 PM

%

ifuAddParity

This routine expects rscr = ifuLH, rscr2 = ifuRH. It sets the parity bits in rscr2 as appropriate. It returns w/ rscr, rscr2 the same, EXCEPT for the correct parity bits set in rscr2.

%

ifuAddParity:

pushReturn[];

* compute pe0: n[0:3], memb low 2(memB[0:1]), instrAddr[0:1]

IfuAddPe0:

t_ldf[rscr2, ifu.nSize, ifu.nShift]; * N first
call[ifuCountOnes];
q_t;

t_ldf[rscr2, 2, ifu.memBlow2Shift]; * now memB[0:1]
call[ifuCountOnes];
t_t+(q);
q_t;

t_ldf[rscr, 2, ifu.AdHi2Shift]; * now InstrAddr[0:1];
call[ifuCountOnes];
t_t+(q);

t and (1c); * assume pe0 = 0. we're computing even parity
skpif[ALU=0], t _ t-t;
t _ (1c);
t_lsh[t, ifu.pe0Shift];
rscr2 _ (rscr2) or t; * done w/ PE0

* compute pe1: instrAddr[2:9];

IfuAddPe1:

t_ldf[rscr, 10, ifu.AdLow8Shift];
call[ifuCountOnes];

t and (1c); * assume pe1 = 0. We're computing even parity
skpif[ALU=0], t_t-t;
t _ 1c;
t_lsh[t, ifu.pe1Shift];
rscr2 _ (rscr2) or t;

* compute pe2: signN, packedAlpha, type[0:1], memBHi2, length[0:1]

IfuAddPe2:

t_ldf[rscr, ifu.paSize, ifu.paShift]; * begin w/ packed alpha
q_t;

t_ldf[rscr2, ifu.signSize, ifu.signShift]; * now ifuSign
call[ifuCountOnes];
t_t + (q);
q _ t;

t_ldf[rscr2, ifu.typeSize, ifu.typeShift]; * now ifuSize
call[ifuCountOnes];
t_t + (q);
q _ t;

t_ldf[rscr2, 2, ifu.memBhi2Shift]; * now ifuBrHi2 (RBaseSelK, memBK)
call[ifuCountOnes];
t _ t + (q);
q _ t;

t_ldf[rscr2, ifu.ILSize, ifu.ILShift]; * now length of opcode
call[ifuCountOnes];
t_t+(q);

t and (1c); * assume even parity
skpif[ALU=0], t _ t-t;
t _ 1c;
t_lsh[t, ifu.pe2Shift];
rscr2 _ (rscr2) or t; * done W/ PE2
returnP[];

* January 7, 1979 6:33 PM

ifuCountOnes: subroutine;

* count the number of one bits in t.

* CLOBBER t, cnt. Return value in T. Push entering value of t onto stack.

pushReturn[];

cnt_17s;

stack+1_t;

t _ t-t;

icoL:

skpif[R EVEN], stack;

t _ t+1;

* increment count of one bits when lsb of stack is one

loopUntil[CNT=0&-1, icoL], stack _ (stack) rsh 1;

pReturnP[];

```
TITLE[ifuStepSubrs];
%
```

CONTENTS

SUBROUTINE

FUNCTION

```
iAddFGParity      add parity bit for presentation to FG bus.
itGetPcXSH       step SH for _PcX'
itIfuJumpSH      step SH for IfuJump[0]
itMakeFDackSH    step SH for memAck & MakeF_D
itMosFhSh        step FH SH for mos_
itMakeFDSH       step SH forMakeF_D
itMemAckFh       step FH for memAck
itMemAckShFh     step SH FH for memAck
itMemAckSH       step SH for memAck
itNewPcFh        step FH for PcF_
itNewPcFhSh     step FhSh for PcF_
itNewPcSH        step SH for PcF_
itNextDataSH    step SH for _ID
itNoopFH         step FH for tick w/ nothing else
itNoopFhSh      step FhSh for tick
itNoopSH         step SH for tick
itResetFH        step FH for reset
itResetSH        step SH for reset
itSetFGFDSH     Step SH for FG_, makeF_D
itSetFGFH        step FH for FG_
itSetFGSH        step SH for FG_
itStep           IfuTest_T, tick
```

%

%

```
June 18, 1981 10:00 AM
```

```
Move incClock to IfuSubrs.
```

```
May 28, 1981 3:01 PM
```

```
Change refs from Bmux to BmuxRM to keep midas happy.
```

```
June 7, 1979 3:06 PM
```

```
Fix typo in itMosFhSh
```

```
June 7, 1979 11:15 AM
```

```
Create setCtemp, make iAddFGParity global and add a temporary kludge to iAddFGParity so that the parity bit is always zero --since there's no longer a hardware path from Test into the FGparity computation.
```

```
June 6, 1979 6:43 PM
```

```
Add missing ; at end if itMemAckSH, before itNewPcFH.
```

```
June 6, 1979 6:04 PM
```

```
Add comments, look for bug in itNewPcFhSh
```

```
April 22, 1979 5:05 PM
```

```
Construct this file from Ifu1, Ifu2.
```

%

* June 7, 1979 3:07 PM

%

test register manipulations

This code performs various operations on the test register, ticks the ifu, etc.

%

m[setCtemp, mc[cTemp, OR[#1!, #2!, #3!, #4!, #5!, #6!]]; * multiple use of this macro will cause harmless, but annoying "errors" during assembly.

```

top level;
call[itResetFH];          * this silly sequence of calls forces the
call[itStep];            * subroutines below to occur on call locs
noop;                    * we never drop thru to next instruction

```

iAddFGParity: subroutine; * **add FG Parity**
* CLOBBERS rscr, rscr2
* OR the proper parity bit into T given that the right byte of T contains the value for FG.
* test.parity causes the 9 bits to have odd parity

```

rscr2 _ rsh[t,10], global; * right justify t
rscr _ test.parity;
cnt _ 7s;                  * IN [0..7]

```

iAddFGPL:
skpif[r even], rscr2 _ (PD_rscr2) rsh 1; * skpif current rscr2 even, then rshift 1
rscr _ (rscr) # (test.Parity);
noop;
loopUntil[CNT=0&-1, iAddFGPL];
* t _ t OR (rscr); * turn this off until the problem w/ FGparity gets
fixed -- then remove all calls to iAddFGParity!
return;

itGetPcXSH: subroutine; * **_PcX**
pushReturn[];
call[iAddFGParity], t _ test.Sh;
IFUTest _ t;
ifuTick;
t _ not(PcX');
noop;
returnP[];

itIfuJumpSH: subroutine; * **IfuJump[0]**
pushReturn[];
call[itiSetIFUJret];
call[iAddFGParity], t _ test.Sh;
IFUTest _ t;
ifuTick;
ifuJump[0];

itIfuJErr:
error;

itIfuJRet:
returnP[];

itiSetIFUJret: subroutine; * initialize KLINK so that afterDispatch will
* return control to itIfuJRet.

```

pushReturn[];
call[itiJret2];
t _ link;
klink _ t;
returnP[];

```

itiJret2: subroutine;
coreturn;
branch[itIfuJRet];

itMakeFDackSH: subroutine; * **MakeF_D, MemAck, SH**
pushReturn[];
setCtemp[test.Sh, test.memAck, test.makeFD];
call[iAddFGParity], t _ cTemp;

```

IFUTest _ t;
ifuTick;
noop;
noop;
returnP[];

itMosFhSh:                                * set the current instruction set
  pushReturnAndT[];
  call[iAddFGParity], t _ test.Fh;
  IFUTest _ t;
  ifuTick;
  B_(stack);
  B_(stack);

  call[iAddFGParity], t _ test.Sh;
  IFUTest _ t;
  ifuTick;
  B_(stack);
  B_(stack);
  pReturnP[];

itMakeFDSH: subroutine;                    * MakeF_D, SH
  pushReturn[];
  setCtemp[test.Sh, test.makeFD];
  call[iAddFGParity], t _ cTemp;
  IFUTest _ t;
  ifuTick;
  noop;
  noop;
  returnP[];

itMemAckFH: subroutine;                    * MemAck, FH
  pushReturn[];
  setCtemp[test.Fh, test.memAck];
  call[iAddFGParity], t _ cTemp;
  IFUTest _ t;
  ifuTick;
  noop;
  noop;
  returnP[];

itMemAckShFh: subroutine;                  * MemAck, FH, SH
  pushReturn[];
  call[itMemAckSH];
  call[itMemAckFH];
  returnP[];

itMemAckSH: subroutine;                    * MemAck, SH
  pushReturn[];
  setCtemp[test.Sh, test.memAck];
  call[iAddFGParity], t _ cTemp;
  IFUTest _ t;
  ifuTick;
  noop;
  noop;
  returnP[];

itNewPcFH: subroutine;                      * PcF_, FH
* enter w/ t = value for new pc
  pushReturnAndT[];
  call[iAddFGParity], t _ test.Fh;
  IFUTest _ t;
  ifuTick;
  PcF _ stack;
  PcF _ stack;
  pReturnP[];
* keep new pc value in rscr
* iAddFGParity returned FG value in T

itNewPcFhSh: subroutine;                    * PcF_, FH, SH
* enter w/ t = new pcF value
  pushReturnAndT[];
  call[itNewPcFH], t_stack;
  call[itNewPcSH], t_stack;
  pReturnP[];

itNewPcSH: subroutine;                      * PcF_, SH

```

```

* enter w/ t = value for new pc
  pushReturnAndT[];
  call[iAddFGParity], t _ test.Sh;
  IFUTest _ t;
  ifuTick;
  PcF _ stack;
  PcF _ stack;
  pReturnP[];

itNextDataSH: subroutine;
  pushReturn[];
  call[iAddFGParity], t _ test.Sh;
  IFUTest _ t;
  ifuTick;
  t _ ID;
  noop;
  returnP[];

itNoopFH: subroutine;
  pushReturnAndT[];
  call[iAddFGParity], t _ test.Fh;
  IFUTest _ t;
  ifuTick;
  B_stack;
  B_stack;
  pReturnP[];

itNoopFhSh: subroutine;
  pushReturnAndT[];
  call[itNoopFH], t_stack;
  call[itNoopSH], t_stack;
  pReturnP[];

itNoopSH: subroutine;
  pushReturnAndT[];
  call[iAddFGParity], t _ test.Sh;
  IFUTest _ t;
  ifuTick;
  B_stack;
  B_stack;
  pReturnP[];

itResetFH: subroutine;
  pushReturn[];
  call[iAddFGParity], t _ test.Fh;
  IFUTest _ t;
  ifuTick;
  ifuReset;
  ifuReset;
  returnP[];

itResetSH: subroutine;
  pushReturn[];
  call[iAddFGParity], t _ test.Sh;
  IFUTest _ t;
  ifuTick;
  ifuReset;
  ifuReset;
  returnP[];

itSetFGFDSH: subroutine;
  pushReturnAndT[];
  rscr _ lsh[t, 10];
setCtemp[test.Sh, test.makeFD];
  call[iAddFGParity], t _ (rscr) OR (cTemp);
  IFUTest _ t;
  ifuTick;
  noop;
  noop;
  pReturnP[];

itSetFGFH: subroutine;
  pushReturnAndT[];
  rscr _ lsh[t, 10];

```

* keep new pc value in rscr

* iAddFGParity returned FG value in T

* _NexData, SH

* tick, FH, BmuxRM _ stack (originally from t)

* tick, FH, SH, BmuxRM _ t

* tick, SH, Bmux _ t

* IfuReset, FH

* IfuReset, SH

* FG[0:7]_t, MakeF_D, SH

* left justify the byte in rscr

* FG[0:7]_t, FH

* left justify the byte in rscr

```
call[iAddFGParity], t _ (rscr) or (test.Fh);
IFUTest _ t;
ifuTick;
noop;
noop;
pReturnP[];
itSetFGSH: subroutine;                                * FG[0:7]_t, SH
pushReturnAndT[];
rscr _ lsh[t, 10];                                     * left justify the byte in rscr
call[iAddFGParity], t _ (rscr) or (test.Sh);
IFUTest _ t;
ifuTick;
noop;
noop;
pReturnP[];

itStep: subroutine;                                    * TEST _ t
* enter w/ T= value for Test register
pushReturnAndT[];
call[iAddFGParity];
IFUTest _ t;
ifuTick;
noop;
noop;
pReturnP[];
```


* June 18, 1981 10:00 AM

%

Simulation subroutines, originally from ifu2.mc

%

```

itTick: subroutine;
  saveReturn[klink];
  call[incClock];
  call[iAddFGParity], t _ FGbits;
  ifuTest _ t;
  ifutick;
  b _ BmuxRM;
  B _ BmuxRM;
  returnUsing[kLink];

itData: subroutine;
  saveReturn[klink];
  call[incClock];
  call[iAddFGParity], t _ FGbits;
  t and (test.myFh);
  skipif[ALU#0];
  error;
  ifuTest _ t;
  ifutick;
  b _ BmuxRM, t _ ID;
  noop;
  returnUsing[kLink];
  * itData, itJumpData, and itJump should
  * never be called when test.Fh
  * we shouldn't be called during test.Fh
  * test.myFh (a "one bit") disables Fh

itJumpData: subroutine;
  saveReturn[kLink];
  call[incClock];
  call[iAddFGParity], t _ FGbits;
  t and (test.myFh);
  skipif[ALU#0];
  error;
  ifuTest _ t;
  ifutick;
  IfuJump[0], B _ BmuxRM, T _ ID;
  noop;
  * we shouldn't be called during test.Fh
  * test.myFh (a "one bit") disables Fh

iAnyIJumpRet:
  returnUsing[kLink];

itJump: subroutine;
  saveReturn[kLink];
  call[incClock];
  call[iAddFGParity], t _ FGbits;
  t and (test.myFh);
  skipif[ALU#0];
  error;
  ifuTest _ t;
  ifutick;
  IfuJump[0], B _ BmuxRM;
  noop;
  returnUsing[kLink];
  * we shouldn't be called during test.Fh
  * test.myFh (a "one bit") disables Fh

itNewPc: subroutine;
  saveReturn[kLink];
  call[incClock];
  call[iAddFGParity], t _ FGbits;
  ifuTest _ t;
  ifutick;
  PcF _ BmuxRM;
  PcF _ BmuxRM;
  returnUsing[kLink];

itReset: subroutine;
  saveReturn[kLink];
  call[incClock];
  call[iAddFGParity], t _ FGbits;

```

```
ifuTest _ t;  
ifutick;  
IfuReset, B _ BmuxRM;  
IfuReset, B _ BmuxRM;  
returnUsing[kLink];
```

```

TITLE[IfuSubrs];
%*+++++
                CONTENTS

TEST                DESCRIPTION

get1Rand:        Get one random number
iGetISubrScr:   Return current value of iSubrScr
incClock         Increment a counter (clockCount) and pulse TIOA for scoping if count has right
value
iPat16:         initialize pat16X
nextPat16:     return t = new pattern, ALU=0 if no more patterns
getPat16:      return current pattern
iPat8:         Initialize pat8X
nextPat8:     Return new 8 bit pattern
getPat8:      Return current 8 bit pattern
iIAddr:       Initialize iAddrX
nextIAddr:    Return next IfuM address
getIAddr:     Return current ifuM address
KfaultX        Memory fault ifu entry point for instruction set X (X IN [0..3])
FGparityX     FGparity ifu entry point for instruction set X (X IN [0..3])
ReschedX      Resched ifu entry point for instruction set X (X IN [0..3])
NotReadyX    Not ready ifu entry point for instruction set X (X IN [0..3])
RamPEX        Ram PE ifu entry point for instruction set X (X IN [0..3])
op0:          Ifu opcode destination
op377:       Ifu opcode destination
opIfAd10:    Ifu opcode destination w/ ifuAd = 10 (address bits in IfuM). fast-Exit.
opIfAd11:    Ifu opcode destination w/ IfuAd = 11. fast-Fetch
opIfAd12:    Ifu opcode destination w/ IfuAd = 12. fast-Store
opIfAd13:    Ifu opcode destination w/ IfuAd = 13. jump-Fail
opIfAd14:    Ifu opcode destination w/ IfuAd = 14. unconditional-Jump
opIfAd15:    Ifu opcode destination w/ IfuAd = 15. most opcodes dispatch to here
opIfAd16:    ifu opcode destination w/ IfuAd = 16. jump-Succeed
opIfAd20Err: ifu opcode destination w/ IfuAd = 20. pause-Halt (should never get here)
afterDispatch: returns control after ifuJump via return link kept in KLINK
readMufBit:  Reads muffler bit, enter w/ t = address
readMuffler: Returns a muffler byte; enter w/ t = muffler address.
%*+++++

%
June 18, 1981 9:59 AM
    Move incClock from ifuStepSubrs to here.
June 18, 1981 9:23 AM
    Construct this file out of ifu1.mc
May 18, 1981 2:25 PM
    Fix nextPat to invoke cycleRandV to get better performance from the random number generator.
February 1, 1980 7:58 PM
    Fix stack handling bug in return code of addressing test. Fix old bug in fastStore code that was
fetching rather than storing.
February 1, 1980 7:48 PM
    Add code to mask out unwanted bits from getIfuLH in addressing test.
February 1, 1980 6:37 PM
    Add iMemAddrs, an addressing logic test.
February 1, 1980 5:06 PM
    Improve the set of patterns generated by pat16.
September 19, 1979 1:13 PM
    Reset expectedDispatch<0 after invoking iSingleStepTest.
September 19, 1979 12:55 PM
    Move beginIfuL to the beginning of the file, add more and better comments, and set up
iSingleStepTest to loop 400 times.
September 17, 1979 5:29 PM
    Add iSingleStepTest.
July 3, 1979 1:53 AM
    Increment return link in checkException by 1 -- because afterdispatch will also decrement it & we
need compatability. This increment occurs after the potential error check where the link has been
decremented to make error interpretation easier.
July 3, 1979 1:17 AM
    Decrement return link in checkException by 1 -- so that it points to where we came from rather
than where to return to, which is irrelevant..
July 3, 1979 12:06 AM
    Add code to enable dynamically selected exception conditions.
June 29, 1979 11:46 AM

```

Modify afterDispatch to save POINTERS into (stack+1), but to return control with stkp = entering value. Do this since afterDispatch resets RBASE to defaultRegion, and some tests want to check the value that RBASE got set to when the IfuJump occurred.

June 20, 1979 9:14 AM

Add call to resetIfu in iMemRandAdrrs and cause beginIful to invoke iMemRandAdrrs.

June 20, 1979 8:50 AM

Add ifuLHmaskC masking to data in iMemRandAdrrs

June 20, 1979 7:35 AM

Add random address generation test (iMemRandAdrrs)

June 19, 1979 10:34 AM

Fix register clobbering bug in new version of ifuMemRW.

June 18, 1979 8:56 AM

Fix bug in 3rd pattern of nextPat16.

June 17, 1979 4:04 PM

Add entry points for all the exception conditions

June 15, 1979 3:24 PM

Enable random patterns in iMemRW

June 5, 1979 12:43 PM

Remove i.testCase2L since test2 has been removed from default diagnostic.

June 5, 1979 11:31 AM

Add disableConditionTask

May 30, 1979 10:08 AM

Add random numbers to pat16 code; diddle iMemRW to reset the Ifu only once.

May 9, 1979 9:05 AM

Fix subroutine mode error in ifuMScopeLoop. Move contents section to top of file & add more comments.

May 4, 1979 10:57 AM

Add scope loop for checking Ram speed.

May 3, 1979 8:44 AM

Add calls to setIUsingInstrSet before invoking initIfuM1Thru(3,16)

April 26, 1979 9:53 AM

Add pat8 subroutines.

April 25, 1979 12:13 PM

Add fastExit, fastFetch, fastStore code.

April 24, 1979 5:27 PM

Remove references to dispatchOffset, simplify afterDispatch.

April 22, 1979 5:10 PM

Move iAddFGParity thru itStep into ifuStepSubrs.mc

February 7, 1979 10:35 AM

Add breakpoints to rampe, kfault, resched entries; explicitly place all the entry points for the instructions.

February 6, 1979 2:16 PM

Add requisite noop after PcF_.

February 5, 1979 12:54 PM

Add ifAd20 -- ifu pause/halt code

February 5, 1979 9:59 AM

Change opifad13,14,16 to do ifuJumps directly.

January 18, 1979 6:44 PM

Fix misc. bugs in pattern16, muffler bit reading code

January 17, 1979 3:52 PM

More Rev-Bb mods: define exception code for instrSet 1 (locations 200,204,214,234,274). Define itMosFhSh.

January 15, 1979 4:52 PM

Rev-Bb mods: instrSet bits, complemented, are or'd into Exception addresses. InstrSet bits 2,3 pick bytes right to left. Cycled 0 pattern for Ifu memory test.

%

* June 18, 1981 9:59 AM

```
get1Rand: pushReturn[];  
  getRandom[];  
  returnP[];
```

```
iGetISubrScr: subroutine;  
  RBASE _ rbase[iSubrScr];  
  return, t _ iSubrScr, RBASE _ rbase[defaultRegion]; * move elsewhere
```

```
incClock: subroutine;  
  RBASE _ rbase[clockCount];  
  t _ clockCount _ (clockCount)+1;  
  t - (triggerCount);  
  skipif[ALU#0], TIOA _ triggerZero;  
  TIOA _ triggerValue;  
  return, RBASE _ rbase[defaultRegion];
```

* May 18, 1981 2:25 PM

iPat16
nextPat16
getPat16

16 bit pattern subroutines

iPat16 initialize pat16X
nextPat16 return t = new pattern, ALU=0 if no more patterns
getPat16 return current pattern
Eventually these will move to postamble.

```

iPat16: subroutine; * initialize 16 bit pattern
    t _ cml;
    return, pat16X _ t;

nextPat16: subroutine; * rtns next 16 bit pattern
    saveReturn[Klink];
    top level;
    RBASE _ rbase[pat16X];
    t _ pat16X _ (pat16X) + 1;
    t - (pat16End1C); * see if done w/ current pattern
    branch[after1Pat16, ALU>=0], PD _ pat16X; * if not done, see if current pattern is zero

* IN [0..pat16End1C)
    skipif[ALU#0]; * init the pattern if this is first time thru
    skip, t _ pat16 _ 1c; * pattern init; set to one
    t _ pat16 _ (pat16) lsh 1; * left shift a one bit
    branch[xitNextPat16ok];

after1Pat16:
    t - (pat16End2C); * see if done w/ pattern 2
    branch[after2Pat16, ALU>=0];

* IN [pat16End2C..pat16End2C)
    t # (pat16End1C); * Pattern 2 = cycled zero bit
    skipif[ALU#0]; * see if first time in Pattern2
    pat16 _ 77777C; * set it to only one zero bit for first time
    pat16 _ t _ lcy[pat16, pat16, 1]; * left cycle it one
    branch[xitNextPat16ok];

after2Pat16: * see if pattern 3
    t - (pat16End3C);
    skipif[ALU<=0];
    branch[after3Pat16]; * see if done w/ pattern 4

* IN [pat16End2C..pat16End3C): Pattern 3
    branch[xitNextPat16ok]; * real work is done by getPat16

after3Pat16:
    t - (pat16End4C);
    skipif[ALU<=0];
    branch[after4Pat16], rscr _ t-t; * all done

* IN [pat16End3C..pat16End4C): Pattern 4
    call[cycleRandV];
    branch[xitNextPat16ok]; * real work is done by getPat16

after4Pat16:
    t - (pat16End5C);
    skipif[ALU<=0];
    branch[after5Pat16]; * all done

* IN [pat16End4C..pat16End5C): Pattern 5
    branch[xitNextPat16ok]; * real work is done by getPat16

after5Pat16:
    t - (pat16End6C);
    skipif[ALU<=0];
    branch[xitNextPat16], rscr _ t-t; * all done

* IN [pat16End5C..pat16End6C): Pattern 6
    branch[xitNextPat16ok]; * real work is done by getPat16

```

```
xitNextPat16ok:
    rscr _ (pat16X)+1;          * put non-zero value in rscr
xitNextPat16:
    returnAndBranch[klink, rscr]; * t = current pattern; ALU#0 means value ok
```

* February 1, 1980 5:39 PM

getPat16: subroutine;

* Enter w/ T = ifu addr. rtn w/ next pattern in T

%

Currently there are four sets of patterns:

pattern1	left cycled 1 bit
pattern2	left cycled 0 bit
pattern3	left cycled index
pattern4	random numbers
pattern5	words of all zeros alternated w/ all ones
pattern6	words of all ones alternated w/ all zeros

Pattern1 and **pattern2** return the same value every time getPat16 is called, given a constant value for pat16X. Ie., the range for pat16X in pattern 1 is [0..pat16End1C), and when pat16X is zero, getPat16 always returns 1, when SpatX is one, getPat16 always returns 2, etc.

Pattern2 is similar to pattern1 except that it is a bitwise complement, ie., a single zero bit with the remaining bits all ones.

Pattern3 returns the current index left cycled. The calling program provides the index (low 16 bits only) in T. The first time pattern3 is in effect, it returns the current index. The next time getPat16 is called, the cycle count for pattern 3 increments and getPat16 returns the index left cycled 1, etc.

Pattern4 returns a random number.

Pattern5 returns a word of all zeros followed by a word of all ones

Pattern6 returns a word of all ones followed by a word of all zeros

%

mc[shc.AisT, b2];

mc[shc.BisT, b3];

set[shc.countShift, 10];

mc[shc.maskShiftCount, b4,b5,b6,b7]

mc[shc.rmt, shc.BisT!];

* deal w/ rm,,t

mc[shc.xrm, 0];

RBASE _ rbase[klink];

* save return and preserve t.

klink _ link;

top level;

RBASE _ rbase[pat16X];

* see which pattern we are using

(pat16X)-(pat16End1C);

branch[get16P2,ALU>=0];

* IN [0..pat16End1C)

pattern 1

* pattern 1 returns a cycled 1 bit.

branch[get16PXit], t_pat16;

* nextPat16 did our work

get16P2:

* return pattern 2 or greater

(pat16X)-(pat16End2C);

branch[get16P3, ALU>=0];

* IN [pat16End1C..pat16End2C)

pattern 2

* pattern 1 returns a cycled 0 bit.

branch[get16PXit], t _ pat16;

* nextPat16 does our work

* February 1, 1980 5:35 PM

get16P3:

```
(pat16X) - (pat16End3C);
branch[get16P4, ALU>=0];
```

* IN [pat16End2C..pat16End3C) *pattern 3*
%

Pattern3 returns a cycled version of the current index.

Begin by moving the current index into curSPattern, and then construct a shifter control value that will provide the correct cycle value. Always construct a SHC value that cycles rm,,t.
%

```
pat16 _ t; * ENTERED W/ T = low 16 bits of index

t _ pat16X; * construct SHC constant. begin w/
t _ lsh[t, shc.countShift]; * positioning the shift count.
t _ t and (shc.maskShiftCount);

t _ t or (shc.rmm); * add control bits
SHC _ t;
t _ pat16 _ shiftNoMask;
branch[get16PXit];
```

get16P4:

```
(pat16X) - (pat16End4C);
branch[get16P5, ALU>=0];
```

* IN [pat16End3C..pat16End4C) *pattern 4*

* pattern4 returns pseudo random numbers
noop;
getRandom[];
RBASE _ rbase[pat16X];
branch[get16PXit], pat16 _ t;

* IN [Spat2EndC..Spat3EndC) (random numbers)

get16P5:

```
(pat16X) - (pat16End5C);
branch[get16P6, ALU>=0];
```

* IN [pat16End4C..pat16End5C) *pattern 5*

* pattern5 returns all zeros alternating w/ all ones
t and (1c);
skpif[alu=0],t_a0;
t_al;
branch[get16PXit], pat16 _ t;

get16P6:

```
(pat16X) - (pat16End6C);
branch[get16PXit, ALU>=0];
```

* IN [pat16End5C..pat16End6C) *pattern 6*

* pattern6 returns all ones alternating w/ all zeros
t and (1c);
skpif[alu#0],t_a0;
t_al;
branch[get16PXit], pat16 _ t;

get16PXit:

```
link _ Klink;
subroutine;
return, RBASE_ rbase[defaultRegion];
```

getCur16pattern: subroutine;

```
saveReturn[Klink];
RBASE _ rbase[pat16];
t _ pat16;
returnUsing[Klink];
```

* return curSPattern in T

* April 26, 1979 9:56 AM

%*****

iPat8
nextPat8
getPat8

This code implements an 8 bit pattern generator. It is similar to the other pattern code except that there is only one pattern and it is a counter that ranges in [0..400).

%*****

iPat8: subroutine;

t _ cml;
return, pat8X _ t;

nextPat8: subroutine;

RBASE _ rbase[pat8X];
t _ pat8X _ (pat8X)+1, RBASE _ rbase[defaultRegion];
return, PD _ t-(400C); * ALU=0 when pattern is done

getPat8: subroutine;

RBASE _ rbase[pat8X];
return, t _ pat8X, RBASE _ rbase[defaultRegion];

* February 1, 1980 7:34 PM

iAddr
nextIAddr
getIAddr

ifu address loop control

```

iAddr: subroutine;                                * initialize Ifu addresses
    t _ cml;
    return, IAddrX _ t;

nextIAddr: subroutine;                            * rtns t = next ifu address (ALU#0 =>valid)
    RBASE _ rbase[IAddrX];
    t _ IAddrX _ (IAddrX)+1, RBASE _ rbase[defaultRegion];
    return, t - (IfuEndAddrC);

getIAddr: subroutine;                             * rtns t = current ifu address (ALU#0 =>valid)
    RBASE _ rbase[IAddrX];
    return, t _ IAddrX, RBASE _ rbase[defaultRegion];

iAddrDownCtrl: subroutine;
    t_IfuEndAddrC;
    return, IAddrX_t;
nextIAddrDown: subroutine;
    RBASE_rbase[IAddrX];
    t_IAddrX_ (IAddrX)-1, RBASE_ rbase[defaultRegion];
    skipif[ALU<0];
    return, PD_ 1c;
    return, PD_ 0c;                                * no more addr

```

* July 3, 1979 12:57 AM

IFU: EXCEPTION CONDITION ENTRY POINTS.

top level;

Instruction Set 3: Exception Condition Entry Points

Exception conditions occur in order of highest priority.

Kfault3: * instruction set 3: Memory fault

```
call[checkException], t _ exceptions.kfault3, at[0];
call[checkException], t _ exceptions.kfault3, at[1];
call[checkException], t _ exceptions.kfault3, at[2];
branch[checkException], t _ exceptions.kfault3, at[3];
```

FGParity3: * instruction set 3: Parity Error in FG data from memD

```
call[checkException], t _ exceptions.FGPe3, at[4];
call[checkException], t _ exceptions.FGPe3, at[5];
call[checkException], t _ exceptions.FGPe3, at[6];
branch[checkException], t _ exceptions.FGPe3, at[7];
```

Resched3: * instruction set 3: Want reschedule

```
call[checkException], t _ exceptions.resched3, at[14];
call[checkException], t _ exceptions.resched3, at[15];
call[checkException], t _ exceptions.resched3, at[16];
branch[checkException], t _ exceptions.resched3, at[17];
```

NotReady3: * instruction set 3: Ifu not yet ready to respond (pipe empty)

```
call[afterDispatch], at[34];
call[afterDispatch], at[35];
call[afterDispatch], at[36];
branch[afterDispatch], at[37];
```

RamPE3: * instruction set 3: Parity Error in Ifu Ram

```
call[checkException], t _ exceptions.RamPe3,, at[74];
call[checkException], t _ exceptions.RamPe3,, at[75];
call[checkException], t _ exceptions.RamPe3,, at[76];
branch[checkException], t _ exceptions.RamPe3, at[77];
```

Instruction Set 2: Exception Condition Entry Points

Exception conditions occur in order of highest priority.

Kfault2: * instruction set 2: Memory Fault

```
call[checkException], t _ exceptions.kfault2, at[100];
call[checkException], t _ exceptions.kfault2, at[101];
call[checkException], t _ exceptions.kfault2, at[102];
branch[checkException], t _ exceptions.kfault2, at[103];
```

FGParity2: * instruction set 2: Parity Error in FG data from memD

```
call[checkException], t _ exceptions.FGPe2, at[104];
call[checkException], t _ exceptions.FGPe2, at[105];
call[checkException], t _ exceptions.FGPe2, at[106];
branch[checkException], t _ exceptions.FGPe2, at[107];
```

Resched2: * instruction set 2: Want reschedule

```
call[checkException], t _ exceptions.resched2, at[114];
call[checkException], t _ exceptions.resched2, at[115];
call[checkException], t _ exceptions.resched2, at[116];
branch[checkException], t _ exceptions.resched2, at[117];
```

NotReady2: * instruction set 2: Ifu not yet ready to respond (pipe empty)

```
call[afterDispatch], at[134];
call[afterDispatch], at[135];
call[afterDispatch], at[136];
branch[afterDispatch], at[137];
```

RamPE2: * instruction set 2: Parity Error in Ifu Ram

```
call[checkException], t _ exceptions.RamPe2, at[174];
call[checkException], t _ exceptions.RamPe2, at[175];
call[checkException], t _ exceptions.RamPe2, at[176];
```

```
branch[checkException], t _ exceptions.RamPe2, at[177];
```

```
*****
```

Instruction Set 1: Exception Condition Entry Points

Exception conditions occur in order of highest priority.

```
*****
```

```
Kfault1: * instruction set 1: Memory Fault
call[checkException], t _ exceptions.kfault1, at[200];
call[checkException], t _ exceptions.kfault1, at[201];
call[checkException], t _ exceptions.kfault1, at[202];
branch[checkException], t _ exceptions.kfault1, at[203];

FGParity1: * instruction set 1: Parity Error in FG data from memD
call[checkException], t _ exceptions.FGPel, at[204];
call[checkException], t _ exceptions.FGPel, at[205];
call[checkException], t _ exceptions.FGPel, at[206];
branch[checkException], t _ exceptions.FGPel, at[207];

Resched1: * instruction set 1: Want reschedule
call[checkException], t _ exceptions.resched1, at[214];
call[checkException], t _ exceptions.resched1, at[215];
call[checkException], t _ exceptions.resched1, at[216];
branch[checkException], t _ exceptions.resched1, at[217];

NotReady1: * instruction set 1: Ifu not yet ready to respond (pipe empty)
call[afterdispatch], at[234];
call[afterDispatch], at[235];
call[afterDispatch], at[236];
branch[afterDispatch], at[237];

RamPE1: * instruction set 1: Parity Error in Ifu Ram
call[checkException], t _ exceptions.RamPel, at[274];
call[checkException], t _ exceptions.RamPel, at[275];
call[checkException], t _ exceptions.RamPel, at[276];
branch[checkException], t _ exceptions.RamPel, at[277];
```

```
*****
```

Instruction Set 0: Exception Condition Entry Points

Exception conditions occur in order of highest priority.

```
*****
```

```
Kfault0: * instruction set 0: Memory Fault
call[checkException], t _ exceptions.kfault0, at[300];
call[checkException], t _ exceptions.kfault0, at[301];
call[checkException], t _ exceptions.kfault0, at[302];
branch[checkException], t _ exceptions.kfault0, at[303];

FGParity0: * instruction set 0: Parity Error in FG data from memD
call[checkException], t _ exceptions.FGPe0, at[304];
call[checkException], t _ exceptions.FGPe0, at[305];
call[checkException], t _ exceptions.FGPe0, at[306];
branch[checkException], t _ exceptions.FGPe0, at[307];

Resched0: * instruction set 0: Want reschedule
call[checkException], t _ exceptions.resched0, at[314];
call[checkException], t _ exceptions.resched0, at[315];
call[checkException], t _ exceptions.resched0, at[316];
branch[checkException], t _ exceptions.resched0, at[317];

NotReady0: * instruction set 0: Ifu not yet ready to respond (pipe empty)
call[afterdispatch], at[334];
call[afterDispatch], at[335];
call[afterDispatch], at[336];
branch[afterDispatch], at[337];

RamPE0: * instruction set 0: Parity Error in Ifu Ram
call[checkException], t _ exceptions.RamPe0, at[374];
call[checkException], t _ exceptions.RamPe0, at[375];
call[checkException], t _ exceptions.RamPe0, at[376];
branch[checkException], t _ exceptions.RamPe0, at[377];
```

* February 1, 1980 8:00 PM

IFU opcodes

This is the microcode that actually executes after an ifuJump.

In general, Klink, an RM location, contains the return link for the current opcode. The "fast" opcodes ignore klink (except for the fastExit opcode that checks CNT=0).

Manually place these opcodes because the diagnostic must know, at run time, where they are located.

about these locations & the names of the labels:

Notice that many labels look like opIfAdXX where XX is a two or three digit number. That number is exactly the number that gets loaded into the Ifu ram to address the location for the label. For example, you see immediately below that **opIfAd10** is placed at location 40. The bits loaded into the Ifu Ram to address opIfAd10 are 10B. If you look at the IfuAddr bits on the Ifu drawings (IFU InstrAddr Bit Slice, p. 7 last time I saw it), you will find the value indicated by the label. Eg. IfuAddr=10 if the Ifu ram has selected the location opIfAd10. The processor provides the remaining two bits of control address in the IfuJump instruction.

opIfAd10: * fast-Exit code: branch to afterDispatch IF CNT=0&-1

```
dblbranch[fastExit0, fastExitCont, CNT=0&-1], at[40];
dblbranch[fastExit0, fastExitCont, CNT=0&-1], at[41];
dblbranch[fastExit0, fastExitCont, CNT=0&-1], at[42];
dblbranch[fastExit0, fastExitCont, CNT=0&-1], at[43];
```

fastExitCont: ifuJump[0];

fastExit0:

```
branch[afterDispatch];
```

opIfAd11: * fast-Fetch code: check stack (must be < 0) and fetch that location.

```
branch[fastFetch0], PD_stack, at[44];
branch[fastFetch0], PD_stack, at[45];
branch[fastFetch0], PD_stack, at[46];
branch[fastFetch0], PD_stack, at[47];
```

fastFetch0:

```
skpif[ALU<0];
stack _ 100000C;
FETCH _ stack, IfuJump[0];
```

opIfAd12: * fast-Store code: store MD into location pointed to by stack, increment stack.

```
branch[fastStore0], (MD) # (stack), at[50];
branch[fastStore0], (MD) # (stack), at[51];
branch[fastStore0], (MD) # (stack), at[52];
branch[fastStore0], (MD) # (stack), at[53];
```

fastStore0:

```
skpif[ALU=0], t _ MD;
```

fastStoreError:

```
error;
stack _ (Store_stack)+1, DBuf_t, IfuJump[0];
```

opIfAd13: * jump-FAIL dispatches here

```
branch[opIf13Jump], t _ (ID) - (PcX'), at[54];
branch[opIf13Jump], t _ (ID) - (PcX'), at[55];
branch[opIf13Jump], t _ (ID) - (PcX'), at[56];
t _ (ID) - (PcX'), at[57];
```

opIf13Jump:

```
t _ t-1;
PcF _ t;
noop; * wait for PcF_ to take effect before the next ifuJump.
IFUJump[0];
```

opIfAd14: * one byte jumps here

```
IFUJump[], stack_ID, at[60];
IFUJump[], stack_ID, at[61];
IFUJump[], stack_ID, at[62];
IFUJump[], stack_ID, at[63];
```

opIfAd15: * very common dispatch location for opcodes

```
call[afterDispatch], at[64];
```

```
call[afterDispatch], at[65];  
call[afterDispatch], at[66];  
branch[afterDispatch], at[67];
```

```
opIfad16: * successful jumps here  
IFUJump[], at[70];  
IFUJump[], at[71];  
IFUJump[], at[72];  
IFUJump[], at[73];
```

```
opIfad22Err: * pause-halt here  
branch[err], at[110];  
branch[err], at[111];  
branch[err], at[112];  
branch[err], at[113];
```

* June 29, 1979 11:45 AM

afterDispatch: subroutine; * come here after ifuNextMacro.

%*****

SOME OTHER routine should set "expectedDispatch" to the absolute value of the location where we expected to dispatch.

SET stack+1 to current value of POINTERS, then reset RBASE to defaultRegion. Return with stkp at its original value (ie., users must increment stkp to see POINTERS).

%*****

stkp+1, global;
stack&-1_POINTERS;

RBASE _ rbase[defaultRegion]; * reset Rbase since Ifu causes it to change
rscr _ link; * compute where we came from
top level;

afterDispatch2: * enter here from checkException
t _ expectedDispatch; * was that where we wanted to go?
skpif[ALU>=0], rscr _ (rscr)-1; * link = where we came from +1
branch[afterDispRet]; * don't check return if expected val<0
t#(rscr);
skpif[ALU=0], t _ rscr2, RBASE _ rbase[klink]; * restore T

afterDispErr:
breakpoint; * expectedDispatch # rscr

afterDispRet:
subroutine;
link_klink;
return, RBASE _ rbase[defaultRegion];

* July 3, 1979 1:53 AM

%*****

The various exception conditions (except for notReady) call this subroutine. If the exception that was called has been enabled, control proceeds through afterdispatch. Otherwise we stop at a breakpoint.

%*****

KnowRbase[defaultRegion]; * depend upon exceptionBit NOT in defaultRegion

checkException: subroutine;
exceptionBit _ t, global;
stkp+1;
stack&-1_POINTERS; * where afterDispatch saves POINTERS
RBASE _ rbase[defaultRegion];
rscr _ link;
rscr _ (rscr) -1;
top level;
RBASE _ rbase[exceptionsMask];
t _ exceptionsMask;
rscr2 _ t;
t _ exceptionBit, RBASE _ rbase[defaultRegion];
t and (rscr2); * see if current exception condition
skpif[ALU#0]; * has been enabled

IfuExceptionErr: * rscr = return link to offending exception condition
error; * presumably klink = return link to source of ifu

jump.

branch[afterDispatch2], rscr _ (rscr)+1; * reincrement exception since afterdispatch will
decrement it.


```

* January 18, 1979 6:47 PM
%*+++++
muffler reading routines
%*+++++
readMufBit: subroutine; * read a muffler bit. Enter w/ t = address
* CLOBBERS T, rscr, rscr2, cnt, stkp
  pushReturn[];
  subroutine; * don't clobber link w/ transfer of ctrl
  cnt _ 10s;

readMufBitsL:
  MidasStrobe _ t;
  noop;
  noop;
  loopUntil[cnt=0&-1, ReadMufBitsL], t _ lsh[t, 1];

  t _ link;
  returnP[];

readMuffler: subroutine; * read a muffler byte and return its value
* in T. Enter with T = muffler address. Return 8 successive bits from that address.
* CLOBBERS T, rscr, rscr2, cnt, stkp

  pushReturnAndT[]; * tos = t
  rscr2 _ t-t; * keep assembled bits in rscr2
  rscr _ t-t; * rscr = loop count
readMufflerL:
  rscr2 _ lsh[rscr2, 1];
  call[readMufBit], t _ stack; * get current bit address
  rscr2 _ (rscr2) or t; * add the current bit

  t _ (stack)+1; * compute address of next bit
  stack _ t;

  (rscr)-(10c); * have we accumulated a byte yet?
  loopUntil[ALU=0, readMufflerL], rscr _ (rscr)+1;

  t _ rscr2;
  pReturnP[]; * tos-1 = return link

```

%*+++++

Table of Contents
Organized by Occurrence of subroutine in this Listing

Subroutine	Function
+++++	
initIfuCache	Init memory, IfuMemory and write opcodes into memory
iMem	Initialize memory system for Ifu
ifuOpcodesToMem	Construct various "programs" in "main" memory
appendHalts	Write three halt opcodes into memory using address in CDbyteAddr.
getCDbyteAddr	Return t = "current cache byte address"
putCDbyteAddr	Set "current cache byte address" with t
putNextByte	Write byte in T into cache at current addr, then increment the addr
putCDbyte	Write byte in rscr into memory byte addr in t
getCDbyte	Returns the byte pointed to by T
getIUsingInstrSet	Returns current value of iUsingInstrSet
setIUsingInstrSet	Sets value of iUsingInstrSet
fixByteAddrForInstrSet	Modify a byte address to accommodate the current instr. set
ResetIfu	Reset ifu, zero IfuTest, clear reschedPending
ifuGetInstrSet	Return the current value of ifu instruction set
ifuSetInstrSet	Set the current value of the ifu instruction set.
initItest	Initialize the test loop control
initTestCount	Initialize the test opcode count
nextITest	Return ALU=0 ==> no more tests, t = memory byte addr of next test
getTestBounds	HIDEOUS IMPLEMENTATION DEPENDENT routine that returns the beginning
and end byte address of the "current" IFU test (test indicated by iTestX).	
getIfuNotReadyLoc	Returns t=not ready location for current instr set.
nextITestCount	Return ALU=0 ==> no more counts, t = current count
ifuTestLoop	infinite loop that invokes a particular (passed in T) ifu executin
test	

%*+++++

%

September 17, 1979 6:37 PM
 Remove reference to afterIfu3.mc
 August 1, 1979 10:38 AM
 Cause ifuSetInstrSet to set iUsingInstrSet.
 July 3, 1979 12:02 AM
 Add code to handle various exception conditions: Cause initIfuCache to clear exceptionsMask.
 June 29, 1979 3:35 PM
 Add slight performance enhancement to memory initialization loop in iMem, fix bug in
 FixByteAddrForInstrSet that prevented it from noticing that instruction set 2 accesses the bytes right
 to left.
 June 28, 1979 11:45 AM
 Call xGetConfig in iMem.
 June 6, 1979 10:32 AM
 Fix placement problems in getIfuNotReadyLoc.
 June 5, 1979 11:46 AM
 Add call to disableConditionalTask in iMem.
 June 4, 1979 10:06 AM
 Add getIfuNotReadyLoc.
 May 30, 1979 10:44 AM
 Add boiler plate to iMem to withstand effects of power-up sequence on Dorado.
 May 4, 1979 10:09 AM
 Add_ID instructions to ifuTestLoop -- to suck data out of pipe in case of 3 byte instrs.
 May 2, 1979 10:33 AM
 Add calls to ifuGetInstrSet, ifuSetInstrSet to force microD to place them properly (they are Midas
 subroutines, only, for the moment).
 May 1, 1979 5:01 PM
 Add getIfuInstrSet, setIfuInstrSet.
 April 27, 1979 4:54 PM
 Fix bug in fixByteAddrForInstrSet.
 April 26, 1979 5:04 PM
 Create this file from ifu3.mc

%

* July 3, 1979 12:03 AM

%

initIfuCache

This code has four main functions:

It initializes the Dorado storage system

It backgrounds all IfuM with valid parity opcodes to a location that invokes error.

It writes opcodes into IfuM[21:30]

It writes a succession of byte codes into the cache

%

initIfuCache: subroutine;

pushReturn[];

t _ a0;

exceptionsMask _ t;

call[iMem], t _ t-t;

call[ifuBackground];

call[initIfuMemory];

call[ifuOpcodesToMem];

returnP[];

* disallow all exceptions but "notReady"

* Use col 0 of cache exclusively.

* write halts into all locations of the Ifu

* write our opcodes into special locations

* write various programs into memory

* June 28, 1979 11:46 AM

```

iMem: subroutine;
pushreturnAndT[];
RBASE _ rbase[MemoryOK];
t _ MemoryOK, RBASE _ rbase[defaultRegion];
branch[iMemXit, ALU#0];
* Background storage so that mem[va] = va for first 65 K memory.

call[disableConditionalTask];
call[xGetConfig];
t _ FaultInfo[];
call[clearCacheFlags];
call[presetMap];
t _ 200c;
call[setTestSyn];
call[clearCacheFlags];
t _ 37C;
call[setMbase];
rscr _ t-t;
call[setBR], rscr2 _ t-t;

MEMBX_0s;
call[setMbase], t _ t-t;
rscr _ t-t;
call[setBR], rscr2 _ t-t;

(stack) - (4c);
skpif[ALU<0];
branch[iMemSetMcr], t _ r0;
t _ lsh[stack, 13];
t _ t or (b2);

iMemSetMcr:
t _ t or (b14);
stack+1 _ t;
call[setMCR], t _ t or (b15);
t _ rscr _ A0;

iMemWriteL:
PreFetch _ rscr;
cnt _ 17s;
loopUntil[CNT=0&-1, .], t _ (STORE_t)+1, DBuf_t; * write current munch
loopUntil[ALU=0, iMemWriteL], rscr _ t + (40C); * increment prefetch pointer

t _ rscr _ 100000C;

iMemWriteL2:
Prefetch _ rscr;
cnt _ 17s;
loopUntil[CNT=0&-1, .], t _ (STORE_t)+1, DBuf_t;
loopUntil[ALU=0, iMemWriteL2], rscr _ t + (40c);
t _ 1c;
MemoryOK _ t;

B _ FaultInfo[];
call[setMCR], t _ stack&-1;

iMemXit:
pReturnP[];

```

* Enter w/ t IN[0..3] ==> use one column
* of cache (t specifies the column).

* only do this once
* Background storage so that mem[va] = va for first 65 K memory.

* set up to handle current configuration.
* clear any waiting faults
* nothing in the cache
* start up the map
* use the error corrector
* nothing in the map (setting Tsyn changed cache)
* init ifu's membase

* Use zero in Ifu's base register

* We'll use membase zero, and zero it, too.

* see if we're supposed to use only one col
* of the cahce

* sigh. we can't fit memdefs, so
* use numbers. should be mcr.useMcrV

* should be mcr.noSEwake;
* remember it for a while
* should be mcr.noWake

* tight loop that writes all of storage
* Prefetch is uninteresting first time thru.

* do it again, just to make sure we flush
* the cache. Otherwise, there may still
* be storage with bad parity.

* remember that we've done all this;

* set MCR for default value.

* January 22, 1979 8:31 PM

%

ifuOpcodesToMem -- write opcodes into cache: **FOR INSTRUCTION SETS 2, 3**
 This initialization works ONLY for instruction sets two and three -- those insrtuction sets take data from the right half FIRST, then the left half of the word. PutNextByte ALWAYS works left to right.

```
word 0      byte 0      (test0MemByteLocC) begins one byte jump .
word 30     byte 60     (test1MemByteLocC) begins two byte jump .
word 50     byte 120    (test2MemByteLocC) begins (two byte) jump .+2, jump .-2
word 100    byte 200    (test3MemByteLocC) a small program whose opcodes cross word boundaries.
word 200    byte 400    (-----) fastTest code (not appropriate to "default" tests.
```

%

ifuOpcodesToMem: subroutine;

```
    pushReturn[];
    call[setIUsingInstrSet], t _ 3c;
%*****
Background all storagte with "identity".
%*****
    t _ cml;
    cnt _ t;
    t _ t-t;
    loopUntil[cnt=0&-1, .], t _ (store_t)+1, DBuf _ t;
%*****
```

Write one-byte "jump ." into **LOCATION 0**, followed by "halts".

```
Logical:
    (jump. fastHalt) (fastHalt fastHalt) (fastHalt fastHalt) (fastHalt <undefinedByte>)
Physical:
    (fastHalt jump.) (fastHalt fastHalt) (fastHalt fastHalt) (fastHalt <undefinedByte>)
%*****
ic3:      * write jump0 into the first 40B bytes in the cache
* At word 0, byte 0
```

```
    call[putCDbyteAddr], t_test0MemByteLocC; * begin writing at location zero

    call[putNextByte], t _ jump0;
    call[putNextByte], t _ fastHalt;
    call[appendHalts]; * try to keep ifu from deep-ending
```

%*****

Write two-byte "jump ." into word **LOCATION 30**, followed by "halts".

```
Logical:
    (jump+ 0) (fastHalt fastHalt) (fastHalt fastHalt) (fastHalt fastHalt)
Physical:
    (0 jump+) (fastHalt fastHalt) (fastHalt fastHalt) (fastHalt fastHalt)
%*****
```

ic4: * place a two byte jump . into cache.
 * at word location 30

```
    t_test1MemByteLocC;
    call[putCDbyteAddr]; * initialize the byte address

    call[putNextByte], t _ jumpL2; * put the offset (=0)
    call[putNextByte], t _ r0; * put the opcode (=2byte jump)
    call[appendHalts]; * try to keep ifu from deep-ending
```

%*****

At word **LOCATION 50**, byte **LOCATION 100** (test2MemByteLocC):

```
Logical:
    (jumpL2 2) (jumpML2- -2)(fastHalt fastHalt) (fastHalt fastHalt) (fastHalt fastHalt)
Physical:
    (2 jumpL2) (-2 jumpML2)(fastHalt fastHalt) (fastHalt fastHalt) (fastHalt fastHalt)
%*****
```

ic5: * place an infinite loop into the cache: jump .+2, jump .-2, where opcode len=2.
 t _ test2MemByteLocC;
 call[putCDbyteAddr];

```

call[putNextByte], t _ jumpL2;
call[putNextByte], t _ 2c;
call[putNextByte], t _ jumpML2;
call[putNextByte], t _ cm2; * now jump .-2

call[appendHalts]; * try to keep ifu from deep-ending

```

iic6:

At word **LOCATION 100**, byte location 200 (test3MemByteLocC):

Logical:

```
(op1 op2)(A op3)(A B)(jumpML2 6)(fastHalt fastHalt)(fastHalt fastHalt)(fastHalt fastHalt)
```

Physical (extracting right to left from memory):

```
(op2 op1)(op3 A1=2)(B2=32 A2=31)(-6 jumpML2)(fastHalt fastHalt)(fastHalt fastHalt)(fastHalt
fastHalt)
```

```

t _ test3MemByteLocC;
call[putCDByteAddr];
noop; * for placement
call[putNextByte], t _ opNoop ; * cross word boundary on 2 byte instr.
call[putNextByte], t _ opNoopL2;
call[putNextByte], t _ 2c;
call[putNextByte], t _ opNoopL3; * cross word boundary on 3 byte instr.
call[putNextByte], t _ 31c;
call[putNextByte], t _ 32c;
call[putNextByte], t _ JumpML2;
call[putNextByte], t _ (add[not[6],1]C); * offset for jumpML2 is -6
call[appendHalts]; * try to keep ifu from deep-ending

```

At word **LOCATION 200**, byte location 400:

Logical:

```
(fastExit fastFetch) (fastStore fast1) (fastFetch fastStore) (fastJfail toHalt) (fastJ top3)
```

(fastHalt fastJ)(-11d fastHalt) <six fastHalts>

Physical (extracting right to left from memory):

```
(fastFetch fastExit) (fast1 fastStore) (fastStore fastFetch) (toHalt fastJfail) (top3 fastJ)
(fastMJ fastHalt)(fastHalt -11d) <six fastHalts>
```

iic7:

```

t _ (lshift[200,1]C); * begin at word 200
call[putCDByteAddr];
noop; * for placement
call[putNextByte], t _ fastExit;
call[putNextByte], t _ fastFetch;
call[putNextByte], t _ fastStore;
call[putNextByte], t _ fast1;
call[putNextByte], t _ fastFetch;
call[putNextByte], t _ fastStore;
call[putNextByte], t _ fastJfail;
call[putNextByte], t _ 10c;
call[putNextByte], t _ fastJ;
call[putNextByte], t _ 3c;
call[putNextByte], t _ fastHalt;
call[putNextByte], t _ fastMJ;
noop; * for placement.
call[putNextByte], t _ (add[not[13],1]C);
call[putNextByte], t _ fastHalt; * -11d = -13B = not(13B)+1
call[putNextByte], t _ fastHalt; * try to keep ifu from deep-ending
call[appendHalts];
call[appendHalts];
t _ 100c;
call[longWait]; * assure that the ifu's mem refs won't miss

```

returnP[];

appendHalts: subroutine;

```

pushReturn[];
call[putNextByte], t _ fastHalt;
call[putNextByte], t _ fastHalt;

```

```
call[putNextByte], t _ fastHalt;  
returnP[];
```

* April 26, 1979 9:00 AM

%

These subroutines support writing opcodes into the cache. Note that several of them use and increment the "current" byte address in the cache:

getCDbyteAddr returns t = "current" cache byte address
setCDbyteAddr t = new value for "current" cache byte address

putNextByte t = byte. place t into "current" cache byte location, increment current addr
putCDbyte t = byte address, rscr = value. Put rscr into indicated byte address
getCDbyte t = byte address, Returns T = byte at that byte location
setIUsingInstrSet t = instr set number. set iUsingInstrSet
getIUsingInstrSet Returns t = current instruction set number
fixByteAddrForInstrSet t = byte address, returns t = byte address modified to reference a byte for the current instruction set. Don't clobber rscr!

%

getCDbyteAddr: subroutine;

```
RBASE _ rbase[currentCDbyte];
t _ currentCDbyte, RBASE_rbase[defaultRegion];
return;
```

putCDbyteAddr: subroutine;

```
RBASE _ rbase[currentCDbyte];
currentCDbyte _ t, RBASE_rbase[defaultRegion];
return;
```

putNextByte: subroutine;

```
noop, global; * enter w/ t = byte to put in "current"
pushReturnAndT[]; * location in cache. Increment the
t _ stack; * current address after writing the byte

rscr _ t; * save the byte to write
call[getCDbyteAddr]; * write it
call[putCDbyte];

call[getCDbyteAddr]; * increment the current address
call[putCDbyteAddr], t _ t+1;
pReturnP[];
```

putCDbyte: subroutine;

```
pushReturn[]; * t = byte address, rscr = byte value
call[fixByteAddrForInstrSet]; * use stack, clobber rscr, t.
stack+1 _ t; * modify byte address if required
t _ stack; * save it on the stack
t and (1c); * see if odd byte
branch[putCDbyteOdd, ALU#0];
```

* **put it into left side**

```
t _ t rsh 1; * fetch the current 16bit value
FETCH _ t;
t _ MD;

rscr _ lsh[rscr, 10]; * position the byte to the left side
t _ t and (377C); * keep the current right side of memory
rscr _ t or (rscr); * Rscr = the new 16 bit value.

t _ (stack) rsh 1; * retrieve memory byte pointer
STORE _ t, DBuf _ rscr;
branch[putCDxit];
```

putCDbyteOdd:

```
t _ t rsh 1; * odd byte address ==> right half of word
FETCH _ t; * fetch the current 16bit value
t _ MD;

rscr _ (rscr) and (377c); * isolate the bits on the right side
t _ t and (177400C); * keep the current left side
rscr _ (rscr) or t; * Rscr = the new 16 bit value.
```

```
t _ (stack) rsh 1; * retrieve memory byte pointer
STORE _ t, DBuf _ rscr;
```

putCDxit:

```
pReturnP[];
```



```

getCDbyte:
    pushReturn[];
    call[fixByteAddrForInstrSet];
    rscr _ t;
    branch[getCDodd, r odd], rscr _ (rscr) rsh 1;
    FETCH _ rscr;
    t _ MD;
    t _ rsh[t, 10];
getCDbyteRtn:
    returnP[];

getCDodd:
    FETCH _ t;
    t _ MD;
    branch[getCDbyteRtn], t _ t and (377C);

* June 29, 1979 3:35 PM
%*****
When we use instruction set 2 or 3, the ifu fetches bytes in the inverse order. Ie., Loc 0 implies
the right byte of word 0 and loc 1 implies the left byte of word 0. This terrible hack accommodates
the Alto's Mesa implementation which addresses bytes in this way.
For the duration, all instruction sets invert the bytes.
%*****
setIUsingInstrSet: subroutine;
    return, iUsingInstrSet _ t;
getIUsingInstrSet:
    RBASE _ rbase[iUsingInstrSet];
    return, t _ iUsingInstrSet, RBASE _ rbase[defaultRegion];
fixByteAddrForInstrSet:
    pushReturnAndT[];

    call[getIUsingInstrSet];
    t-(2c);
    skipif[ALU>=0];
    branch[fbaifiRtn], t _ (stack&-1);

    branch[fbaifiOdd, R ODD], t _ (stack&-1);
    branch[fbaifiRtn], t _ t + 1;
fbaifiOdd:
    t _ t and (177776C);
fbaifiRtn:
    returnP[];
    
```

* enter w/ t = byte address
 * modify byte address if required

* invert the order of the bytes if we
 * are using instr set 3 or 4.
 * rtn w/ t = original input, adjust stack
 * put byte addr back into t, adjust stack
 * it is an even byte address
 * address is odd, IN instructionSet2 or 3.
 * remove low order bit

* August 1, 1979 10:40 AM

%

ResetIfu

Reset the Ifu by performing two IfuReset functions separated by a longWait of 100B cycles.

%

resetIfu: subroutine;

```
pushReturn[];
t _ t-t;
ifuTest_t;
IfuReset[];
noReschedule[];
call[longWait], t _ 100c;
ifuReset[];
returnP[];
```

* May 2, 1979 10:32 AM

top level;

call[ifuGetInstrSet];

call[ifuSetInstrSet];

noop;

* These two subroutine calls force microD

* to place the subroutine entry pts properly.

ifuGetInstrSet: subroutine;

t_ not(IFUMLH');

return, t _ ldf[t, ifu.InstrSetSize, ifu.GetInstrSetShift];

ifuSetInstrSet: subroutine;

t _ t and (ifu.InstrSetMask);

iUsingInstrSet _ t;

t _ lsh[t, ifu.SetInstrSetShift];

t _ t or (100000C);

return, MOS _ t;

* April 24, 1979 8:12 AM

test iteration control:
initItest
initTestCount
nextTest
getTestBounds
nextTestCount

These subroutines cause the main ifu diagnostic to iterate a fixed number of times for each test, then to change to the next test.

initItest: subroutine;

t_cml;
 return, iTestX_t;

initITestCount:

t _ cml;
 return, iTestCountX_t;

nextITest: subroutine;

pushReturn[];
 RBASE _ rbase[iTestX];
 t _ iTestX _ (iTestX)+1, RBASE _ rbase[defaultRegion];
 t # (lastTestC);
 branch[nextITestXDone, ALU=0];
 Bdispatch _ t;
 branch[testMemTable];

* RTN: ALU#0 ==> more tests to do;
 * T = memory byte location of current test.

set[testMemTbl, 1140];

testMemTable:

branch[nextITest2], t _ test0MemByteLocC, at[testMemTbl,0];
 branch[nextITest2], t _ test1MemByteLocC, at[testMemTbl,1];
 branch[nextITest2], t _ test2MemByteLocC, at[testMemTbl,2];
 branch[nextITest2], t _ test3MemByteLocC, at[testMemTbl,3];

nextITest2:

iCurrentTestLoc _ t;
 rscr _ lc;

nextITestXit:

returnPandBranch[rscr];

nextITestXDone:

branch[nextITestXit], rscr _ t-t;

getTestBounds

HIDEOUS DEPENDENCY:

THIS CODE PRESUMES TO KNOW THE LENGTH OF THE PROGRAMS WRITTEN IN MEMORY
 RETURNS RSCR = beginning of the current test program in memory (byte location)
 RSCR2 = end of current test program in memory (byte location)

getTestBounds:

pushReturn[];
 RBASE _ rbase[iTestX];
 t_iTestX, RBASE _ rbase[defaultRegion];
 BDispatch_t;
 branch[testMemTable2];

set[testMemTbl2, 1150];

testMemTable2:

branch[gtbl0], rscr _ test0MemByteLocC, at[testMemTbl2,0];
 branch[gtbl1], rscr _ test1MemByteLocC, at[testMemTbl2,1];
 branch[gtbl2], rscr _ test2MemByteLocC, at[testMemTbl2,2];
 branch[gtbl3], rscr _ test3MemByteLocC, at[testMemTbl2,3];

* This code sets RSCR2 = the upper bound

gtbl0: branch[gtblRtn], rscr2 _ rscr;
gtbl1: branch[gtblRtn], rscr2 _ rscr;
gtbl2: rscr2 _ rscr;
 branch[gtblRtn], rscr2 _ (rscr2) + (2c);
gtbl3: rscr2 _ rscr;
 rscr2 _ (rscr2) + (6c);

gtblRtn:

returnP[];

nextTestCount

This subroutine determines if the current test has executed enough times to go on to the next test.
 ALU#0 means there are more ifuJumps to come.

```

nextITestCount: subroutine;
    pushReturn[];
    RBASE _ rbase[iTestCountX];
    t _ iCurrentTestLoc;
    rscr2 _ t;
    t _ iTestCountX _ (iTestCountX)+1, RBASE _ rbase[defaultRegion];
    t # (lastTestCountC);
    skipif[ALU#0], rscr _ 1c;
    rscr _ t-t;
    t _ rscr2;
    returnPandBranch[rscr];
    
```


* May 4, 1979 10:07 AM

ifuTestLoop

This code implements an infinite loop associated with one of the various ifu execution tests. If all goes well, control will pass from the location of the IfuJump to after Dispatch to the location that performs the ifuJump (again). In the event a random transfer of control occurs, the place in IM where control passes must be patched to the location "ifuTestL" which will reset PCF. Note that we suck 4 IDs from Ifu -- to free pipe in case of 3 byte instructions w/ encoded constant, packed alpha, etc.

ifuTestLoop:

```

top level;
t _ t-1; * enter w/ t = index of ifu test
iTestX _ t;
call[initIfuCache];
call[setIUsingInstrSet], t _ 3c;
call[nextItest];
skpif[ALU#0];

```

ifuTestLErr: * entered with a bad value in T

```

error;
stack+1 _ t; * remember the byte location in the stack.

```

ifuTestL:

```

call[resetIfu];
PcF _ stack;
call[setIfuTestLRet];

```

ifuTestLRet:

```

A_ID;
A_ID;
A_ID;
IfuJump[0], A_ID;

```

setIfuTestLRet:

```

pushReturn[];
call[sitlr2];
t _ link;
klink _ t;
returnP[];

```

sitlr2:

```

coreturn;
branch[ifuTestLRet];

```