MESA MICROCODE

Version 41

: Mesa.Mu - Instruction fetch and general subroutines
: Last modified by Johnsson - July 20, 1979 8:42 AM
: addresses changed for RAM operation, Johnsson November 5, 1979 4:05 PM

: Get definitions for ALTO and MESA

#AltoConsts23.mu;
#Mesab.mu;

'uCodeVersion' is used by RunMesa to determine what version of the Mesa microcode is in ROM1. This version number should be incremented by 1 for every official release of the microcode. 'uCodeVersion' is mapped by RunMesa to the actual version number (which appears as a comment above). The reason for this mapping is the limited number of constants in the Alto constants ROM, otherwise, we would obviously have assigned 'uCodeVersion' the true microcode version number.

The current table in RunMesa should have the following correspondences:

<table>
<thead>
<tr>
<th>uCodeVersion</th>
<th>Microcode version</th>
<th>Mesa release</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>34</td>
<td>4.1</td>
</tr>
<tr>
<td>1</td>
<td>39</td>
<td>5.0</td>
</tr>
<tr>
<td>2</td>
<td>41</td>
<td>6.0</td>
</tr>
</tbody>
</table>

SuCodeVersion $2;

: Completely rewritten by Roy Levin, Sept-Oct. 1977
: Modified by Johnsson; July 25, 1977 10:20 AM
: First version assembled 5 June 1975.
: Developed from Lampson's MESA.U of 21 March 1975.
GLOBAL CONVENTIONS AND ASSUMPTIONS

1) Stack representation:
   stkp=0 => stack is empty
   stkp=10 => stack is full
   The validity checking that determines if the stack pointer is
   within this range is somewhat perfunctory. The approach taken is
   to include specific checks only where there absence would not lead
   to some catastrophic error. Hence, the stack is not checked for
   underflow, since allowing it to become negative will cause a disaster
   on the next stack dispatch.

2) Notation:
   Instruction labels correspond to opcodes in the obvious way. Suffixes
   of A and B (capitalized) refer to alignment in memory. 'A' is
   intended
to suggest the right-hand byte of a memory word; 'B' is intended to
suggest the left-hand byte. Labels terminating in a lower-case letter
generally name local branch points within a particular group of
opcodes. (Exception: subroutine names.) Labels terminating in 'x'
generally
exist only to satisfy alignment requirements imposed by various
dispatches (most commonly IR- and B/A in instruction fetch).

3) Tasking:
   Every effort has been made to ensure that a 'TASK' appears
approximately every 12 instructions. Occasionally, this has
not been possible, but (it is hoped that) violations occur only
in infrequently executed code segments.

4) New symbols:
   In a few cases, the definitions of the standard Alto package
(ALTOCONSTS23.MU) have not been quite suitable to the needs of this
microcode. Rather than change the standard package, we have defined
new symbols (with names beginning with 'm') that are to be used
instead of their standard counterparts. All such definitions
appear together in Mesab.Mu.

5) Subroutine returns:
   Normally, subroutine returns using IDISP require one to deal with
(the nuisance of) the dispatch caused by loading IR. Happily,
however, no such dispatch occurs for 'msr0' and 'sr1' (the relevant
bits are 0). To cut down on alignment restrictions, some subroutines
assume they are called with only one of two returns and can therefore
ignore the possibility of a pending IR- dispatch. Such subroutines
are clearly noted in the comments.

6) Frame pointer registers (lp and gp):
   These registers normally (i.e. except during Xfer) contain the
addresses of local 2 and global 1, respectively. This optimizes
accesses in such bytecodes as LL3 and SG2, which would otherwise
require another cycle.
There is a fundamental difficulty in the selection of addresses that are known and used outside the Mesa emulator. The problem arises in trying to select a single set of addresses that can be used regardless of the Alto's control memory configuration. In effect, this cannot be done. If an Alto has only a RAM (in addition, of course, to its basic ROM, ROMO), then the problem does not arise. However, suppose the Alto has both a RAM and a second ROM, ROM1. Then, when it is necessary to move from a control memory to one of the other two, the choice is conditioned on (1) the memory from which the transfer is occurring, and (2) bit 1 of the target address. Since we expect that, in most cases, an Alto running Mesa will have the Mesa emulator in ROM1, the externally-known addresses have been chosen to work in that case. They will also work, without alteration, on an Alto that has no ROM1. However, if it is necessary to run Mesa on an Alto with ROM1 and it is desired to use a Mesa emulator residing in the RAM (say, for debugging purposes), then the address values in the RAM version must be altered. This implies changes in both the RAM code itself and the Nova code that invokes the RAM (via the Nova JMPRAM instruction). Details concerning the necessary changes for re-assembly appear with the definitions below.

Note concerning Alto IVs and Alto IIIs with retrofitted 3K control RAMs:

The above comments apply uniformly to these machines if "RAM" is systematically replaced by "RAMI" and "ROM" is systematically replaced by "RAMO".

%I,1777,0,nextBa; forced to location 0 to save a word in JRAM

Emulator Entry Point Definitions

These addresses are known by the Nova code that interfaces to the emulator and by RAM code executing with the Mesa emulator in ROM1. They have been chosen so that both such "users" can use the same value. Precisely, this means that bit 1 (the 400 bit) must be set in the address. In a RAM version of the Mesa emulator intended to execute on an Alto with a second ROM, bit 1 must be zero.

%I,1777,20,Mgo; Normal entry to Mesa Emulator - load state of process specified by AC0.
%I,1777,400,next,nextA; Return to 'next' to continue in current Mesa process after Nova or RAM execution.
$Minterpret $L004400,0,0; Documentation refers to 'next' this way.
%I,1777,776,DStr1,Mstopc; Return addresses for 'Savestate'. By standard convention, 'Mstopc' must be at 777.
Linkage from Mesa emulator to ROMO

The Mesa emulator uses a number of subroutines that reside in ROMO. In posting a return address, the emulator must be aware of the control memory in which it resides, RAM or ROM1. These return addresses must satisfy the following constraint:

- no ROM1 extant or emulator in ROM1 => bit 1 of address must be 1
- ROM1 extant and emulator in RAM => bit 1 of address must be 0

In addition, since these addresses must be passed as data to ROMO, it is desirable that they be available in the Alto's constants ROM. Finally, it is desirable that they be chosen not to mess up too many pre-defs. It should be noted that these issues do not affect the destination location in ROMO, since its address remains fixed (even with respect to bit 1 mapping) whether the Mesa emulator is in RAM or ROM1. [Note pertaining to retrofitted Alttos with 3K RAMs: to avoid confusion, the comments above and below have not been revised to discuss 3K control RAMs, although the values suggested are compatible with such machines.]

MUL/DIV linkage:

An additional constraint peculiar to the MUL/DIV microcode is that the high-order 7 bits of the return address be 1's. Hence, the recommended values are:

- no ROM1 extant or emulator in ROM1 => MUL/DIVretloc = 177576B (OK to be odd)
- ROM1 extant and emulator in RAM => MUL/DIVretloc = 177162B (OK to be odd)

$ROMMUL $L004120,0,0; MUL routine address (120B) in ROMO
$ROMDIV $L004121,0,0; DIV routine address (121B) in ROMO
$MULDIVretloc $177162; (may be even or odd)

The third value in the following pre-def must be: ((MULDIVretloc-2) AND 777B)

%1,1777,160,MULDIVret,MULDIVret1;

BITBLT linkage:

An additional constraint peculiar to the BITBLT microcode is that the high-order 7 bits of the return address be 1's. Hence, the recommended values are:

- no ROM1 extant or emulator in ROM1 => BITBLTret = 177577B
- ROM1 extant and emulator in RAM => BITBLTret = 177175B

$ROMBITBLT $L004124,0,0; BITBLT routine address (124B) in ROMO
$BITBLTret $177175; (may be even or odd)

The third value in the following pre-def must be: (BITBLTret AND 777B)-1

%1,1777,174,BITBLTintr,BITBLTdone;

CYCLE linkage:

A special constraint here is that WFretloc be odd. Recommended values are:

- no ROM1 extant or emulator in ROM1 =>
  - Fieldretloc = 452B, WFretloc = 523B
- ROM1 extant and emulator in RAM =>
  - Fieldretloc = 34104B, WFretloc = 14023B

$RAMCYCX $L004022,0,0; CYCLE routine address (22B) in ROMO
$Fieldretloc $34104; RAMCYCX return to Fieldsub (even or odd)
$WFretloc $14023; RAMCYCX return to WF (must be odd)

The third value in the following pre-def must be: (Fieldretloc AND 1777B)

%1,1777,104,Fieldrc;

The third value in the following pre-def must be: (WFretloc AND 1777B)-1

%1,1777,22,WFnzct,WFret;
Instruction fetch

State at entry:
1) \( i_b \) holds either the next instruction byte to interpret (right-justified) or 0 if a new word must be fetched.
2) control enters at one of the following points:
   a) next: \( i_b \) must be interpreted
   b) nextA: \( i_b \) is assumed to be uninteresting and a new instruction word is to be fetched.
   c) nextXB: a new word is to be fetched, and interpretation is to begin with the odd byte.
   d) nextAdeaf: similar to 'nextA', but does not check for pending interrupts.
   e) nextXBdeaf: similar to 'nextXB', but does not check for pending interrupts.

State at exit:
1) \( i_b \) is in an acceptable state for subsequent entry.
2) \( T \) contains the value 1.
3) A branch (1) is pending if \( i_b = 0 \), meaning the next instruction may return to 'nextA'. (This is subsequently referred to as "ball 1", and code that nullifies its effect is labelled as "dropping ball 1".)
4) If a branch (1) is pending, \( L = 0 \). If no branch is pending, \( L = 1 \).
Address pre-definitions for bytecode dispatch table.

Table must have 2 high-order bits on for BUS branch at 'nextAni'.

Warning! Many address inter-dependencies exist - think (at least) twice before re-ordering. Inserting new opcodes in previously unused slots, however, is safe.

X7,1777,1400,NOOP,ME,MRE,MXW,MXD,NOTIFY,BCAST,REQUEUE; 000-007
X7,1777,1420,LL8,LLD,SL0,SL1,SL2,SL3,SL4,SL5; 010-017
X7,1777,1430,SL6,SL7,SLB,PL0,PL1,PL2,PL3,LG0; 020-027
X7,1777,1440,LF1,LF2,LF3,LF4,LF5,LF6,LF7,LF8; 030-037
X7,1777,1450,LG0,LG1,LG2,LG3,LG4,LG5,LG6,LG7; 040-047
X7,1777,1460,LG8,LG9,LG10,LG11,LG12,LG13,LG14,LG15; 050-057
X7,1777,1470,LLW,LLN,LADDB,GADDB,...; 060-067
X7,1777,1470,LIW,LINB,LADDB,GADDB,...; 070-077
X7,1777,1470,LIW,LINB,LADDB,GADDB,...; 080-087
X7,1777,1470,LIW,LINB,LADDB,GADDB,...; 090-097
X7,1777,1470,LIW,LINB,LADDB,GADDB,...; 100-107
X7,1777,1510,RF,WF,RF,RF,RF,RF,RF,RF; 110-117
X7,1777,1520,RIDE,WLR,RXD,RXD,RXD,RXD,RXD; 120-127
X7,1777,1540,; 140-147
X7,1777,1550,; 150-157
X7,1777,1560,; 160-167
X7,1777,1570,DUP,RF,RF,RF,RF,RF,RF; 170-177
X7,1777,1600,J4,J4,J4,J4,J4,J4,J4,J4; 180-187
X7,1777,1610,J5,J5,J5,J5,J5,J5,J5,J5; 190-197
X7,1777,1620,J6,J6,J6,J6,J6,J6,J6,J6; 200-207
X7,1777,1630,J6,J6,J6,J6,J6,J6,J6,J6; 210-217
X7,1777,1640,J7,J7,J7,J7,J7,J7,J7,J7; 220-227
X7,1777,1650,J8,J8,J8,J8,J8,J8,J8,J8; 230-237
X7,1777,1660,J9,J9,J9,J9,J9,J9,J9,J9; 240-247
X7,1777,1670,J10,J10,J10,J10,J10,J10,J10,J10; 250-257
X7,1777,1680,J11,J11,J11,J11,J11,J11,J11,J11; 260-267
X7,1777,1690,J12,J12,J12,J12,J12,J12,J12,J12; 270-277
X7,1777,1700,EFC0,EFC1,EFC2,EFC3,EFC4,EFC5,EFC6,EFC7; 280-287
X7,1777,1710,EFC8,EFC9,EFC10,EFC11,EFC12,EFC13,EFC14,EFC15; 290-297
X7,1777,1720,EFCB,LF1C,LF2C,LF3C,LF4C,LF5C,LF6C,LF7C; 300-307
X7,1777,1730,LF8C,...; 310-317
X7,1777,1740,LFCB,SFC,RET,LLKB,PORTO,PORTI,KFCB; 320-327
X7,1777,1750,DESC,DESCBS,BLT,,BLT,,ALOC,,FREE; 330-337
X7,1777,1760,IWDC,DWDC,STOP,CATCH,MISC,BITBLT,STARTIO,STARTIO; 340-347
X7,1777,1770,DO,TST,STST,WR,WR,WR,WR,STKUF; 350-357
X7,1777,1780,DO,TST,STST,WR,WR,WR,WR,STKUF; 360-367
X7,1777,1790,DO,TST,STST,WR,WR,WR,WR,STKUF; 370-377
; Main interpreter loop

; Enter here to interpret ib. Control passes here to process odd byte
; of previously fetched word or when preceding opcode "forgot" it should
; go to 'nextA'. A 'TASK' should appear in the instruction preceding
; the one that branched here.

next:     L=0, :nextBa;          (if from JRAM, switch banks)
nextBa:   SINK+ib, BUS;
            ib=L, T=0+1, BUS=0, :NOOP;
            dispatch on ib
            establish exit state

; NOOP - must be opcode 0
; control also comes here from certain jump instructions

!1,1,nextAput;

NOOP:     L=mpc+T, TASK, :nextAput;
; Enter here to fetch new word and interpret even byte. A 'TASK' should
; appear in the instruction preceding the one that branched here.

nextA:  L=MAR+mpc+1, :nextAcom;  ; initiate fetch

; Enter here when fetch address has been computed and left in L. A 'TASK'
; should appear in the instruction that branches here.

nextAput:  temp=L;
            L=MAR+temp, :nextAcom;

; Enter here to do what 'nextA' does but without checking for interrupts

nextAdeaf:  L=MAR+mpc+1;
nextAdeafa:  mpc=L, BUS=0, :nextAcomx;

; Common fetch code for 'nextA' and 'nextAput'

!1,2,nextAi,nextAni;
!1,2,nextAini,nextAii;

nextAcom:  mpc=L;
            SINK=WWW, BUS=0;
nextAcomx:  T=177400, :nextAi;

; No interrupt pending. Dispatch on even byte, store odd byte in ib.

nextAni:  L=MD AND T, BUS, :nextAgo;
nextAgo:  ib=L LCY 8, L+T+0+1, :NOOP;

; Interrupt pending - check if enabled.

nextAi:  L=MD;
           SINK=wdc, BUS=0;
           T=M.T, :nextAini;
nextAini:  SINK=M, L=T, BUS, :nextAgo;

; Interrupt pending and enabled.

!1,2,nextXBini,nextXBii;

nextAii:  L=mpc-l;
           mpc=L, L=0, :nextXBii;

; check wakeup counter
; isolate left byte
; dispatch even byte

; back up mpc for Savpcinframe
: Enter here to fetch word and interpret odd byte only (odd-destination jumps).
: !1.2,nextXBi,nextXBi;
  
  nextXB: L=MAR=mpc+T;
  SINK=NNW, BUS=0, :nextXBdeaf; : check pending interrupts
  
  Enter here (with branch (1) pending) from Xfer to do what 'nextXB' does but without
  checking for interrupts. L has appropriate word PC.
  
  nextXBdeaf: mpc=L, :nextXBi;
  
  No interrupt pending. Store odd byte in ib.
  
  nextXBni:
  L=MD, TASK, :nextXBini;
  
  nextXBini: ib=L LCY 8, :next;
  skip over even byte (TASK
  prevents L=0, :nextBa)
  
  Interrupt pending - check if enabled.
  
  nextXB1:
  SINK=wdc, BUS=0, :nextXBini;
  check wakeup counter
  
  Interrupt pending and enabled.
  
  nextXB1i: ib=L, :Intstop;
  ib = 0 for even, =0 for odd
The two most heavily used subroutines (Popsub and Getalpha) often share common return points. In addition, some of these return points have additional addressing requirements. Accordingly, the following predefinitions have been rather carefully constructed to accommodate all of these requirements.

Any alteration is fraught with peril.

[A historical note: an attempt to merge in the returns from FetchAB as well failed because more than 31D distinct return points were then required. Without adding new constants to the ROM, the extra returns could not be accommodated. However, for Popsub alone, additional returns are possible - see Xpopsub.]

Return Points (sr0-sr17)
17,20,Fieldra,SFCr,pushTB,pushTA,LLBr,LGBr,SLBr,SGBr, LADRBr,GADRBr,RFr,Xret,INCr,RBr,WBr,Xpopret;

Extended Return Points (sr20-sr37)
Note: KFCr and EFCr must be odd!
17,20,XbrkBr,KFCr,LFCr,EFCr,WSDBra,DBLr,LINBr,LDIVf, Dpush,Dpop,RDOr,Splitcomr,RXLPrb,WXLPrb,MISCr,;

Returns for Xpopsub only
17,20,WSTRrB,WSTRrA,JRAMr,WRr,STARTIOr,PORTOr,WDoOr,ALLOCrx, FREErx,NEGr,RFSr,RFSr,WFSr,DESCBcom,RFCr,NILCKr; (more could be appended)

Extended Return Machinery (via Xret)
1,2,XretB,XretA;

Xret: SINK=DISP, BUS, :XretB;
XretB: :XbrkBr;
XretA: SINK=0, BUS=0, :XbrkBr; keep ball 1 in air
Pop subroutine:
Entry conditions:
Normal IR linkage
Exit conditions:
Stack popped into T and L

!1.1.Popsub; shakes B/A dispatch
!7.1.Popsuba; shakes IR= dispatch

Popsub: L=stk-1, BUS, TASK, :Popsuba;
Popsuba: stk=L, :Tpop; old stk>0

Xpop subroutine:
Entry conditions:
L has return number
Exit conditions:
Stack popped into T and L
Invoking instruction should specify 'TASK'

!1.1.Xpopsub; shakes B/A dispatch

Xpopsub: saveret=L;
Tpop: IR=sr17, :Popsub;
Xpopret: SINK=saveret, BUS;
:WSTRrB;

Note: putting Tpop here makes stack underflow logic work if stk=0
; Getalpha subroutine:
; Entry conditions:
; L untouched from instruction fetch
; Exit conditions:
; alpha byte in T
; branch 1 pending if return to 'nextA' desirable
; L=0 if branch 1 pending, L=1 if no branch pending
;-----------------------------------------------------------------
!1.2.Getalpha.GetalphaA:
!7.1.Getalpha;
!7.1.GetalphaAx;

Getalpha: T=ib, IDISP;
Getalpha: ib=L RSH 1, L=0, BUS=O, :Fieldra;
GetalphaA: L=MAR+mpc+1;
GetalphaAx: mpc=L;
T=177400;
L=MD AND T, T=MD;
Getalphab: T=377.T, IDISP;
ib=L LCY 8, L=0+1, :Fieldra;

; FetchAB subroutine:
; Entry conditions: none
; Exit conditions:
; T: (mpc)+1>
; ib: unchanged (caller must ensure return to 'nextA')
;-----------------------------------------------------------------
!1.1.FetchAB;
!7.1.FetchABx;
!7.10.LIWr,JWr,......:

FetchAB: L=MAR+mpc+1, :FetchABx;
FetchABx: mpc=L, IDISP;
T=MD, :LIWr;
Splitalpha subroutine:

Entry conditions:
L: return index
entry at Splitalpha if instruction is A-aligned, entry at SplitalphaB if instruction is B-aligned
entry at Splitcom splits byte in T (used by field instructions)

Exit conditions:
lefthalf: alpha[0-3]
righthalf: alpha[4-7]

1.2. Splitalpha, SplitalphaB:
1.1. Splitx;
1.2. Splitout0.Splitout1;
7,10,RILPr,RIGPr,WILPr,RXLPr,WXLPr,Fieldrb,;
subroutine returns

Splitalpha: saveret=L, L=0+1, :Splitcom;
SplitalphaB: saveret=L, L=0, BUS=0, :Splitcom;

%1:10,RILPr,RIGPr,WILPr,RXLPr,WXLPr,Fieldrb,;

Splitcom: IR=sr33, :Getalpha;
Splitcomr: L=17 AND T, :Splittx;
Splittx: righthalf=L, L=T, TASK;
temp=L;
L=temp, BUS;
temp=L LCY 8, SH<0, :Split0;

Split0: L=T=0, :Splitout0;
Split1: L=T=ONE, :Splitout0;
Split2: L=T=2, :Splitout0;
Split3: L=T=3, :Splitout0;
Split4: L=T=4, :Splitout0;
Split5: L=T=5, :Splitout0;
Split6: L=T=6, :Splitout0;
Split7: L=T=7, :Splitout0;

Splitout1: L=10+T, :Splitout0;

Splittout0: SINK=saveret, BUS, TASK;
lefthalf=L, :RILPr;

lefthalf: alpha[0-3]
; Dispatches

; Pop-into-T (and L) dispatch:
; dispatches on old stkp, so Tpop0 = 1 mod 20B.

Tpop0:     L+T=stk0, IDISP, :Tpopexit;
Tpop1:     L+T=stk1, IDISP, :Tpopexit;
Tpop2:     L+T=stk2, IDISP, :Tpopexit;
Tpop3:     L+T=stk3, IDISP, :Tpopexit;
Tpop4:     L+T=stk4, IDISP, :Tpopexit;
Tpop5:     L+T=stk5, IDISP, :Tpopexit;
Tpop6:     L+T=stk6, IDISP, :Tpopexit;
Tpop7:     L+T=stk7, IDISP, :Tpopexit;

Tpopexit:    :Fieldra;                        to permit TASK in Popsub
**PushMD dispatch:**

- Pushes memory value on stack.
- The invoking instruction must load MAR and may optionally keep ball 1 in the air by having a branch pending. That is, entry at 'pushMD' will cause control to pass to 'next', while entry at 'pushMDA' will cause control to pass to 'nextA'.

```plaintext
!3,4,pushMD,pushMDA,StoreB,StoreA;
17,20,push0,push1,push2,push3,push4,push5,push6,push7,push10,...

pushMD:   L=stk+1, IR=stk+1;  (IR+ causes no branch)  
          stk+L, T=O+1, :pushMDA;

pushMDA:  L=stk+1, IR=stk+1;  (IR+ causes no branch)  
          stk+L, T=O, :pushMDA;

pushMDa:  SINK=DISP, L=T, BUS;  
          L=MD, SH=O, TASK, :push0;
```

---

**PushT dispatch:**

- Pushes T on stack.
- The invoking instruction may optionally keep ball 1 in the air by having a branch pending. That is, entry at 'pushTB' will cause control to pass to 'next', while entry at 'pushTA' will cause control to pass to 'nextA'.

```plaintext
!1,2,pushT1B,pushT1A;  keep ball 1 in air

pushTB:   L=stk+1, BUS, :pushT1B;  
pushTA:   L=stk+1, BUS, :pushT1A;

pushT1B:  stk+L, L=T, TASK, :push0;  
pushT1A:  stk+L, BUS=O, L=T, TASK, :push0;  BUS=O keeps branch pending
```

---

**Push dispatch:**

- Strictly vanilla-flavored.
- May (but need not) have branch (1) pending if return to 'nextA' is desired.
- Invoking instruction should specify TASK.

```plaintext
; Note: the following pre-def occurs here so that dpushof1 can be referenced in push0
!17,20,dpush1,dpush2,dpush3,dpush4,dpush5,dpush6,dpush7,dpushof1,dpushof2,...;

push0:    stk0=L, :next;  honor TASK, stack overflow
push1:    stk1=L, :next;
push2:    stk2=L, :next;
push3:    stk3=L, :next;
push4:    stk4=L, :next;
push5:    stk5=L, :next;
push6:    stk6=L, :next;
push7:    stk7=L, :next;
push10:   :dpushof1;
```
Double-word push dispatch:
- picks up alpha from ib, adds it to T, then pushes <result> and
  <result+1>.
- entry at 'Dpusha' substitutes L for ib.
- entry at 'Dpushc' and 'DpB' is used by RR 6 logic.
- entry at 'dpush' is used by MUL/DIV/LDIV logic.
- returns to 'nextA' <=> ib = 0 or entry at 'Dpush'

!1,2,DpA,DpB;
!1,1,Dpushb;
!5,2,Dpushx,RCLKr;

Dpush: MAR=L+ib+T, :DpB;
Dpusha: SINK=ib, BUS=0;
        MAR=L+M+T, :DpA;
DpA: IR=0, :Dpushb;
DpB: IR=2000, :Dpushb;
Dpushb: temp=L, :Dpushx;
Dpushx: L=MD, TASK, :Dpushc;
Dpushc: taskhole=L;
        T=0+1;
        L=L+taskhole;
        MAR=M+T+1;
        stkp=L;
        SINK=stk, BUS, :dpush;
dpush: T=MD, :dpush;
dpush1: stk0=L, L=T, TASK, mACSOURCE, :push1;
dpush2: stk1=L, L=T, TASK, mACSOURCE, :push2;
dpush3: stk2=L, L=T, TASK, mACSOURCE, :push3;
dpush4: stk3=L, L=T, TASK, mACSOURCE, :push4;
dpush5: stk4=L, L=T, TASK, mACSOURCE, :push5;
dpush6: stk5=L, L=T, TASK, mACSOURCE, :push6;
dpush7: stk6=L, L=T, TASK, mACSOURCE, :push7;
dpushof1: T+sStackOverflow, :KFCr;
dpushof2: T+sStackOverflow, :KFCr;

shakes B/A dispatch from RCLK
shakes IR=2000 dispatch and
provides return to RCLK

L: address of low half
mACSOURCE will produce 0
mACSOURCE will produce 1
temp: address of low half
taskhole: low half bits
fetch high half
stk = stk+2
L: low half bits
dispatch on new stkp
T: high half bits

stack cells are S-registers,
so mACSOURCE does not affect
addressing.
; TOS+T dispatch:
; adds TOS to T, then initiates memory operation on result.
; used as both dispatch table and subroutine - fall-through to 'pushMD'.
; dispatches on old stkp, so MAStkT0 = 1 mod 20B.

!17,20,MAStkT,MAStkT0,MAStkT1,MAStkT2,MAStkT3,MAStkT4,MAStkT5,MAStkT6,MAStkT7,.....:

MAStkT0: MAR=stk0+T, :pushMD;
MAStkT1: MAR=stk1+T, :pushMD;
MAStkT2: MAR=stk2+T, :pushMD;
MAStkT3: MAR=stk3+T, :pushMD;
MAStkT4: MAR=stk4+T, :pushMD;
MAStkT5: MAR=stk5+T, :pushMD;
MAStkT6: MAR=stk6+T, :pushMD;
MAStkT7: MAR=stk7+T, :pushMD;

; Common exit used to reset the stack pointer
; the instruction that branches here should have a 'TASK'
; Setstkp must be odd, StkOflw used by PUSH

!17,11,Setstkp,.....,StkOflw:

Setstkp: stkp=L, :next; branch (1) may be pending
StkOflw: :dpushof1; honor TASK, dpushof1 is odd

; Stack Underflow Handling

StkUf: T=sStackUnderflow, :KFCr; catches dispatch of stkp = -1
Store dispatch:

- pops TOS to MD.
- called from many places.
- dispatches on old stk., so MDpop0 = 1 mod 20B.
- The invoking instruction must load MAR and may optionally keep ball 1
  in the air by having a branch pending. That is, entry at 'StoreB' will
  cause control to pass to 'next', while entry at 'StoreA' will cause
  control to pass to 'nextA'.

!1, 2, StoreBa, StoreAa;
!17, 20, MDpopuf, MDpop0, MDpop1, MDpop2, MDpop3, MDpop4, MDpop5, MDpop6, MDpop7,......:

StoreB: L=stk0-1, BUS;
StoreBa: stkp=L, TASK, :MDpopuf;
StoreA: L=stk0-1, BUS;
StoreAa: stkp=L, BUS=0, TASK, :MDpopuf;
MOpop0: MD=stk0, :next;  keep branch (1) alive
MOpop1: MD=stk1, :next;
MOpop2: MD=stk2, :next;
MOpop3: MD=stk3, :next;
MOpop4: MD=stk4, :next;
MOpop5: MD=stk5, :next;
MOpop6: MD=stk6, :next;
MOpop7: MD=stk7, :next;

Double-word pop dispatch:

- picks up alpha from ib, adds it to T, then pops stack into result and
  result+1
- entry at 'Dpopa' substitutes L for ib.
- returns to 'nextA' <=> ib = 0 or entry at 'Dpop'

!17, 20, dpopuf2, dpopuf1, dpop1, dpop2, dpop3, dpop4, dpop5, dpop6, dpop7,......:

!1, 1, Dpopb; required by placement of
MOpopuf only.

Dpop: L=T+ib+T+1;
MOpopuf: IR=0, :Dpopb;
Dpopa: L=T+M+T+1;
Dpopb: MAR=T, temp=L;
dpopuf2: L=stk0-1, BUS;
stkp=L, TASK, :dpopuf2;
dpopuf1: :StkUf;
dpop1: MD=stk1, :Dpopx;
dpop2: MD=stk2, :Dpopx;
dpop3: MD=stk3, :Dpopx;
dpop4: MD=stk4, :Dpopx;
dpop5: MD=stk5, :Dpopx;
dpop6: MD=stk6, :Dpopx;
dpop7: MD=stk7, :Dpopx;
Dpopx: SINK=DISP, BUS=0;
MASstkT: MAR=temp-1, :StoreB;

Note: MOpopuf is merely a convenient label which leads
to a BUS dispatch on stkp in
the case that stkp is -1. It
is used by the Store dispatch
above.

stack underflow, honor TASK
; Get operation-specific code from other files

#Mesac.mu;
#Mesad.mu;
; MesabROM.Mu - Registers, miscellaneous symbols and constants
; Last modified by Levin - November 5, 1979 3:17 PM

; R memories used by code in ROMO, correct to AltoCode23.Mu

; Nova Emulator Registers (some used by Mesa as well)

$AC3   $R0;
$MASK1 $R0;
$AC2   $R1;
$AC1   $R2;
$YMUL  $R2;
$RETURN $R2;
$ACO   $R3;
$SKEW  $R3;
$NW    $R4;
$SAD   $R5;
$CYRET $R6;
$TEMP  $R6;
$PC    $R6;
$XREG  $R7;
$CYCOUT $R7;
$WIDTH $R7;
$PLICR $R7;
$XH    $R10;
$SDESTY $R10;
$SWORDZ $R10;
$DWAX  $R35;
$STARTBITSM1 $R36;
$MASK  $R36;
$SWA   $R36;
$SDESTX $R36;
$LREG  $R40;
$NLINES $R41;
$RAST1 $R42;
$SRCX  $R43;
$SKMSK $R43;
$SRCY  $R44;
$RAST2 $R44;
$CONST $R45;
$TWICE $R45;
$HCNT  $R46;
$VINC  $R46;
$HINC  $R47;
$NWORDS $R50;
$MASK2 $R51;
; Registers used by standard Nova I/O controllers
;
; All names have been prefixed with 'x' to prevent conflicts when MesabROM is
; used by XMesa clients to assemble MesaXRAM with other microcode.
;
; Model 31 Disk
$xKWDCT $R31;
$xKWDCTW $R31;
$xCKSUMR $R32;
$xCKSUMRW $R32;
$xKNMAR $R33;
$xKNMARW $R33;
$xDCBR $R34;

; Display
$CURX $R20;
$CURDATA $R21;
$XCBR $R22;
$XAECL $R23;
$XLC $R24;
$XMTEMP $R25;
$xHTAB $R26;
$xVPOS $R27;
$xDWA $R30;

; Ethernet
$ECNTR $R12;
$EPNTR $R13;

; Memory Refresh
$xCLOCKTEMP $R11;
$xR37 $R37;

; Audio (obsolete)
$xAudioWdCt $R71;
$xAudioData $R72;
; Registers used by Mesa Emulator

; R registers

$\text{temp} \quad R35; \quad \text{Temporary (smashed by BITBLT)}
$\text{temp2} \quad R36; \quad \text{Temporary (smashed by BITBLT)}
$\text{mpc} \quad R15; \quad \text{R register holds Mesa PC (points at word last read)}
$\text{stkp} \quad R16; \quad \text{stack pointer \[0-10\] 0 empty, 10 full}
$\text{XTSreg} \quad R17; \quad \text{xfer trap state}

; Registers shared by Nova and Mesa emulators

$\text{brkbyte} \quad R0; \quad \text{(AC3) bytecode to execute after a breakpoint}
$\text{mx} \quad R1; \quad \text{Warning! brkbyte must be reset to 0 after ROMO calls!}
$\text{saveret} \quad R2; \quad \text{(AC2) x register for XFER}
$\text{newfield} \quad R3; \quad \text{Warning! smashed by BITBLT and MUL/DIV/LDIV}
$\text{count} \quad R5; \quad \text{scratch R register used for counting}
$\text{taskhole} \quad R7; \quad \text{pigeonhole for saving things across TASKs}
$\text{ib} \quad R10; \quad \text{Instruction byte, 0 if none (0, byte)}
$\text{clockreg} \quad R37; \quad \text{Low-order bits of real-time clock}

; S registers, can't shift into them, BUS not zero while storing.

$\text{my} \quad R51; \quad \text{y register for XFER}
$\text{lp} \quad R52; \quad \text{local pointer}
$\text{gp} \quad R53; \quad \text{global pointer}
$\text{cp} \quad R54; \quad \text{code pointer}
$\text{ATPreg} \quad R55; \quad \text{allocation trap parameter}
$\text{OTPreg} \quad R56; \quad \text{other trap parameter}
$\text{XTPreg} \quad R57; \quad \text{xfer trap parameter}
$\text{wdc} \quad R70; \quad \text{Wakeup disable counter}

; Mesa evaluation stack

$\text{stk0} \quad R60; \quad \text{stack (bottom)}
$\text{stk1} \quad R61; \quad \text{stack}
$\text{stk2} \quad R62; \quad \text{stack}
$\text{stk3} \quad R63; \quad \text{stack}
$\text{stk4} \quad R64; \quad \text{stack}
$\text{stk5} \quad R65; \quad \text{stack}
$\text{stk6} \quad R66; \quad \text{stack}
$\text{stk7} \quad R67; \quad \text{stack (top)}

; Miscellaneous S registers

$\text{mask} \quad R41; \quad \text{used by string instructions, among others}
$\text{unused1} \quad R42; \quad \text{not safe across call to BITBLT}
$\text{unused2} \quad R43; \quad \text{not safe across call to BITBLT}
$\text{alpha} \quad R44; \quad \text{alpha byte (among other things)}
$\text{index} \quad R45; \quad \text{frame size index (among other things)}
$\text{entry} \quad R46; \quad \text{allocation table entry address (among other things)}
$\text{frame} \quad R47; \quad \text{allocated frame pointer (among other things)}
$\text{righthalf} \quad R41; \quad \text{right 4 bits of alpha or beta}
$\text{lefthalf} \quad R45; \quad \text{left 4 bits of alpha or beta}
$\text{unused3} \quad R46; \quad \text{not safe across call to BITBLT}
Mnemonic constants for subroutine return indices used by BUS dispatch.

$ret0 $L0, 12000, 100; zero is always special
$ret1 $1;
$ret2 $2;
$ret3 $3;
$ret4 $4;
$ret5 $5;
$ret6 $6;
$ret7 $7;
$ret10 $10;
$ret11 $11;
$ret12 $12;
$ret13 $13;
$ret14 $14;
$ret15 $15;
$ret16 $16;
$ret17 $17;
$ret20 $20;
$ret21 $21;
$ret22 $22;
$ret23 $23;
$ret24 $24;
$ret25 $25;
$ret26 $26;
$ret27 $27;
$ret30 $30;
$ret31 $31;
$ret37 $37;
; Mesa Trap codes - index into sd vector

$sBRK $L0.12000,100; Breakpoint
$sStackError $2;
$sStackUnderflow $2; (trap handler distinguishes underflow from
$sStackOverflow $2; overflow by stkp value)
$sXferTrap $4;
$sAllocTrap $6;
$sControlFault $7;
$sSwapTrap $10;
$sUnbound $13;
$sBoundsFault $20;
$sPointerFault $21; must equal sBoundsFault+1
$sBoundsFaultm1 $17; must equal sBoundsFault-1

; Low- and high-core address definitions

$HardMRE $20; location which forces MRE to drop to Nova code
$CurrentState $23; location holding address of current state
$NovaDVloc $25; dispatch vector for Nova code
$avm1 $777; base of allocation vector for frames (-1)
$sdoffset $100; offset to base of sd from av
$gftm1 $1377; base of global frame table (-1)
$BankReg $177740; address of emulator's bank register

; Constants in ROM, but with unpleasant names

$12 $12; for function calls
$-12 $177766; for Savestate
$400 $400; for JB

; Frame offsets and other software/microcode agreements

$lpoffset $6; local frame overhead + 2
$nlpoffset $177771; = -(lpoffset + 1)
$nlpoffset1 $177770; = -(lpoffset + 2)
$pcoffset $1; offset from local frame base to saved pc
$npcoffset $6; = -(lpoffset+1+pcoffset) [see Savpcinframe]
$sretlinkoffset $2; offset from local frame base to return link
$nrretlinkoffset $177774; = -(lpoffset-retlinkoffset)
$gpoffset $4; global frame overhead + 1
$ng offset $177773; = -(gpoffset + 1)
$gfloffset $L0.12000,100; offset from global frame base to gfi word (=0)
$ngfloffset $4; = gpoffset-gfloffset [see XferGfz]
$gpcpoffset $1; offset from global frame base to code pointer
$gfcimask $177600; offset from high code pointer to global 1
$gfimask $177600; mask to isolate gfi in global frame word 0
$enmask $37; mask to isolate entry number/4

; Symbols to be used instead of ones in the standard definitions

$smACSOURCE $L024016,000000,00000000; sets only F2. ACSOURCE also sets BS and RSEL
$msr0 $L000000,012000,000100; IDISP => 0, no IR= dispatch, a 'special' zero
$BUSAND-T $L000000,054016,000040; sets ALUF = 15B, doesn't require defined bus
; Linkages between ROM1 and RAM for overflow microcode

; Fixed locations in ROM1

$romnext  $L004400,0,0;  must correspond to next
$romnextA $L004401,0,0;  must correspond to nextA
$romIntstop $L004406,0,0;  must correspond to Intstop
$romUntail $L004407,0,0;  must correspond to Untail
$romMgo   $L004420,0,0;  must correspond to Mgo
$romXfer   $L004431,0,0;  must correspond to Xfer

; Fixed locations in RAM

$ramBLTloop $L004403,0,0;  must correspond to BLTloop
$ramBLTint  $L004405,0,0;  must correspond to BLTint
$ramOverflow $L004410,0,0;  RR, BLTL, WR
                       DADD, DSUB, DCOMP, DUCOMP

;
The following requirements are assumed:
1) J2-J9, JB are usable (in that order) as subroutine returns (by JEQx and JNEx).
2) since J2-J9 and JB are opcode entry points, they must meet requirements set by opcode dispatch.

Jn - jump PC-relative

!1,2,JnA,Jbranchf;

J2: L=ONE, :JnA;
J3: L=2, :JnA;
J4: L=3, :JnA;
J5: L=4, :JnA;
J6: L=5, :JnA;
J7: L=6, :JnA;
J8: L=7, :JnA;
J9: L=10, :JnA;
JnA: L=M-1, :Jbranchf; A-aligned - adjust distance

JB - jump PC-relative by alpha, assuming:
JB is A-aligned
Note: JEQB and JNEB come here with branch (1) pending

!1,1,JBx;
!1,1,Jbranch;

JB: T=ib, :JBx;
JBx: L=400 OR T; IR=M; L=DISP-1, :Jbranch; +DISP will do sign extension
400 above causes branch (1)
L: ib (sign extended) - 1

JW - jump PC-relative by alphabeta, assuming:
if JW is A-aligned, B byte is irrelevant
alpha in B byte, beta in A byte of word after JW

JW: IR=srl, :FetchAB; returns to JW
JWr: L=ALLONES+T, :Jbranch; L: alphabeta-1

Jump destination determination
L has (signed) distance from even byte of word addressed by mpc+1

!1,2,Jforward,Jbackward;
!1,2,Jeven,Jodd;

Jbranch: T=0+1, SHC0; dispatch fwd/bkwd target
Jbranchf: SINK=M, BUSODD, TASK, :Jforward; dispatch even/odd target
Jforward: temp=L RSH 1, :Jeven; stash positive word offset
Jbackward: temp=L MRSH 1, :Jeven; stash negative word offset
Jeven: T=temp+1, :NOOP; fetch and execute even byte
Jodd: T=temp+1, :nextXB; fetch and execute odd byte
JZEQB - if TOS (popped) = 0, jump PC-relative by alpha, assuming:
  stack has precisely one element
  JZEQB is A-aligned (also ensures no pending branch at entry)
\[\begin{align*}
1, 2, Jcz, Jco; \\
JZEQB: & \quad \text{SINK=stk0, BUS=0; test TOS = 0} \\
& \quad \text{L=stkP-1, TASK, :Jcz;}
\end{align*}\]

JZNEB - if TOS (popped) <= 0, jump PC-relative by alpha, assuming:
  stack has precisely one element
  JZNEB is A-aligned (also ensures no pending branch at entry)
\[\begin{align*}
1, 2, JZNEBne, JZNEBeq; \\
JZNEB: & \quad \text{SINK=stk0, BUS=0; test TOS = 0} \\
& \quad \text{L=stkP-1, TASK, :JZNEBne;}
\end{align*}\]
\[\begin{align*}
JZNEBne: & \quad \text{stkP=L, :JB; branch, pick up alpha} \\
JZNEBeq: & \quad \text{stkP=L, :nextA; no branch, alignment => nextA}
\end{align*}\]
; JEQn - if TOS (popped) = TOS (popped), jump PC-relative by n, assuming:
; stack has precisely two elements

!1,2,JEQnB,JEQnA;
!7,1,JEQNEcom;

JEQ2: IR=sr0, L+T, :JEQnA; shake IR+ dispatch
returns to J2

JEQ3: IR=sr1, L+T, :JEQnB;
returns to J3

JEQ4: IR=sr2, L+T, :JEQnB;
returns to J4

JEQ5: IR=sr3, L+T, :JEQnB;
returns to J5

JEQ6: IR=sr4, L+T, :JEQnB;
returns to J6

JEQ7: IR=sr5, L+T, :JEQnB;
returns to J7

JEQ8: IR=sr6, L+T, :JEQnB;
returns to J8

JEQ9: IR=sr7, L+T, :JEQnB;
returns to J9

; JEQB - if TOS (popped) = TOS (popped), jump PC-relative by alpha, assuming:
; stack has precisely two elements
; JEQB is A-aligned (also ensures no pending branch at entry)

JEQB: IR=srlO, :JEQnA; returns to JB

; JEQ common code

!1,2,JEQcom,JNEcom;

JEQnB: temp=L RSH 1, L+T, :JEQNEcom;
temp:0, L:1 (for JEQNEcom)
JEQnA: temp=L, L+T, :JEQNEcom;
temp:1, L:1 (for JEQNEcom)

!1,2,JEQne,JEQeq;

JEQcom: L+stkp-T-1, :JEQne;
L: old stkp - 2

JEQne: SINK=temp, BUS, TASK, :Setstkp;
no jump, reset stkp
JEQeq: stkp=L, IDISP, :JEQNExxx;
jump, set stkp, then dispatch
dispatch EQ/NE
even/odd dispatch

; JEQ/JNE common code

!7,1,JEQNEcom; appears above with JEQn
!1,2,JEQcom,JNEcom; appears above with JEQB

JEQNEcom: T+stk1;
dispatch EQ/NE
test outcome and return
L+stk0-T, SH=0;
T+0+1, SH=0, :JEQcom;

JEQNExxx: SINK=temp, BUS, :J2;
even/odd dispatch
; JNEn - if TOS (popped) = TOS (popped), jump PC-relative by n, assuming:
; stack has precisely two elements
;-----------------------------------------------------------------------
1, 2, JNEnB, JNEnA:

JNE2:  IR=sr0, L=T, :JNEnB; returns to J2
JNE3:  IR=sr1, L=T, :JNEnB; returns to J3
JNE4:  IR=sr2, L=T, :JNEnB; returns to J4
JNE5:  IR=sr3, L=T, :JNEnB; returns to J5
JNE6:  IR=sr4, L=T, :JNEnB; returns to J6
JNE7:  IR=sr5, L=T, :JNEnB; returns to J7
JNE8:  IR=sr6, L=T, :JNEnB; returns to J8
JNE9:  IR=sr7, L=T, :JNEnB; returns to J9

; JNEB - if TOS (popped) = TOS (popped), jump PC-relative by alpha, assuming:
; stack has precisely two elements
; JNEB is A-aligned (also ensures no pending branch at entry)
;-----------------------------------------------------------------------
JNEB:  IR=sr10, :JNEnA; returns to JB

; JNE common code
;-----------------------------------------------------------------------
JNEB:  temp=L RSH 1, L=O, :JEQNEcom; temp:O, L:O
JNEA:  temp=L, L=O, :JEQNEcom; temp:1, L:O
1, 2, JNEeq

JNEcom:  L=stkP-T-1, :JNEeq; L: old stkP - 2
JNEeq:  stkP=L, IDISP, :JEQNEeqx; jump, set stkP, then dispatch
JNEeq:  SINK=temp, BUS, TASK, :SetstkP; no jump, reset stkP
JrB - for r in {L, LE, GE, UL, ULE, UG, UGE}
if TOS (popped) r TOS (popped), jump PC-relative by alpha, assuming:
stack has precisely two elements
JrB is A-aligned (also ensures no pending branch at entry)

The values loaded into IR are not returns but encoded actions:

- Bit 12: 0 => branch if carry zero
- Bit 15: 0 => perform add-complement before testing carry

(These values were chosen because of the masks available for use with +DISP
in the existing constants ROM. Note that IR+ causes no dispatch.)

<table>
<thead>
<tr>
<th>JrB</th>
<th>IR</th>
<th>:Jscale</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>JLB</td>
<td>IR=10</td>
<td>:Jscale</td>
<td>adc, branch if carry one</td>
</tr>
<tr>
<td>JLEB</td>
<td>IR=11</td>
<td>:Jscale</td>
<td>sub, branch if carry one</td>
</tr>
<tr>
<td>JGB</td>
<td>IR=ONE</td>
<td>:Jscale</td>
<td>sub, branch if carry zero</td>
</tr>
<tr>
<td>JGEB</td>
<td>IR=0</td>
<td>:Jscale</td>
<td>adc, branch if carry zero</td>
</tr>
</tbody>
</table>

JULB: IR=10, :Jnoscale; adc, branch if carry one
JULEB: IR=11, :Jnoscale; sub, branch if carry one
JUGB: IR=ONE, :Jnoscale; sub, branch if carry zero
JUGEB: IR=0, :Jnoscale; adc, branch if carry zero

Comparison "subroutine":

!1,2, JADC, JSUB;
!1,2, JCE, JCO; appears above with JZEQB
!1,2, JNOZ, JBO;
!1,2, JBO, JNOBO;

Jscale: T+77777, :Jadjust;
Jnoscale: T=ALLONES, :Jadjust;
Jadjust: L=stk1+T+1;
        temp=L;
        SINK=DISP, BUSODD;
        T=stk0+T+1, :Jadc;
Jadc: L=temp-T-1, :Jcommon;
Jsub: L=temp-T, :Jcommon;
Jcommon: T=ONE;
        L=stk-1, ALUCY;
        SINK=DISP, SINK=1gm10, BUS=0, TASK, :Jcz;
        Jcz: stkp=L, :Jnobs;
        stkp=L, :Jco;
        Jnobs: L=mpc+1, TASK, :nextAput;
        Jnobo: L=mpc+1, TASK, :nextAput;
        Jbo: T+1b, :Jbx;
        Jb: T+1b, :Jbx;

warning: not T=0+1
no jump, alignment=>nextAa
jump
jump
no jump, alignment=>nextAa
JIW: see Principles of Operation for description
assumes:
stack contains precisely two elements
if JIW is A-aligned, B byte is irrelevant
alpha in B byte, beta in A byte of word after JIW

!1.2.JIuge,JIu1;
!1.1.JIWx;

JIW: L=stk0-T-1, TASK, :JIWx;
     stkp=L;
     T=stk0;
     L=MAR+mpc+1;
     mpc+L;
     L=stk1-T-1;
     ALUCY;
     T=MD, :JIuge;
     stkp=stk0-2

JIWx: L=stk0-L;
      load alphabeta
     do unsigned compare

JIuge: L=mpc+1, TASK, :nextAput;
      out of bounds - to 'nextA'
      (removing this TASK saves a
       word, but leaves a run of
       15 instructions)
     fetch <<cp>+alphabeta+X>

JIu1: L=cp+T, TASK;
     taskhole=L;
     T=taskhole;
     MAR=stk0+T;
     NOP;
     L=MD-1, :Jbranch;
     L: offset
; Loads
; Note: These instructions keep track of their parity

; LLn - push (lp+n)
; Note: LL3 must be odd!

; Note: lp is offset by 2, hence the adjustments below

LL0:           MAR=lp-T-1, :pushMD;
LL1:           MAR=lp-1, :pushMD;
LL2:           MAR=lp, :pushMD;
LL3:           MAR=lp+T, :pushMD;
LL4:           MAR=lp+T+1, :pushMD;
LL5:           T+3, SH=0, :LL3;             pick up ball 1
LL6:           T+4, SH=0, :LL3;             pick up ball 1
LL7:           T+5, SH=0, :LL3;             pick up ball 1

; LLB - push (lp+alpha)

LLB:           IR=IR4, :Getalpha;          returns to LLBr
LLBr:          T=nlpoffset+T+1, SH=0, :LL3; undiddle lp, pick up ball 1

; LLDB - push (lp+alpha), push (lp+alpha+1)
; LLDB is A-aligned (also ensures no pending branch at entry)

LLDB:          T=lp, :LDcommon;
LDcommon:      T=nlpoffset+T+1, :Dpush;

; LGn - push <<gp>+n>
; Note: LG2 must be odd!

; Note: gp is offset by 1, hence the adjustments below

LGO: MAR+gp-1, :pushMD;
LGI: MAR+gp, :pushMD;
LG2: MAR+gp+T, :pushMD;
LG3: MAR+gp+T+1, :pushMD;
LG4: T+3, SH=0, :LG2;  pick up ball 1
LG5: T+4, SH=0, :LG2;  pick up ball 1
LG6: T+5, SH=0, :LG2;  pick up ball 1
LG7: T+6, SH=0, :LG2;  pick up ball 1

LGB: IR=sr5, :Getalpha;  returns to LGBr
LGBr: T-ngpoffset+T+1, SH=0, :LG2;  undiddle gp, pick up ball 1

LGDB: T=gp+T+1, :LDcommon;  T: gp-gpoffset+lpoffset
Lin - push n
!1,2,LIOxB,LIOxA; keep ball 1 in air

; Note: all BUS dispatches use old stkp value, not incremented one

LIO: L=stkP+1, BUS, :LIOxB;
L11: L=stkP+1, BUS, :pushT1B;
L12: T=2, :pushTB;
L13: T=3, :pushTB;
L14: T=4, :pushTB;
L15: T=5, :pushTB;
L16: T=6, :pushTB;
LIOxB: stkP=L, L=0, TASK, :push0;
LIOxA: stkP=L, BUS=0, L=0, TASK, :push0; BUS=0 keeps branch pending

LIN1 - push -1

LIN1: T=ALLONES, :pushTB;

LINI - push 100000

LINI: T=100000, :pushTB;

LIB - push alpha

LIB: IR=sr2, :Getalpha; returns to pushTB
Note: pushT1B will handle any pending branch

LINB - push (alpha OR 37780)

LINB: IR=sr26, :Getalpha; returns to LINBr
LINBr: T=177400 OR T, :pushTB;

LIW - push alphabeta, assuming:
; if LIW is A-aligned, B byte is irrelevant
; alpha in B byte, beta in A byte of word after LIW

LIW: IR=msr0, :FetchAB; returns to LIWr
LIWr: L=stkP+1, BUS, :pushT1A; duplicates pushTA, but because of overlapping return points, we can't use it
; Stores

; SLn - <<lp>+n>>TOS (popped)
; Note: SL3 is odd!

; Note: lp is offset by 2, hence the adjustments below

SL0:       MAR=lp-T-1, :StoreB;
SL1:       MAR=lp-1, :StoreB;
SL2:       MAR=lp, :StoreB;
SL3:       MAR=lp+T, :StoreB;
SL4:       MAR=lp+T+1, :StoreB;
SL5:       T=3, SH=0, :SL3;
SL6:       T=4, SH=0, :SL3;
SL7:       T=5, SH=0, :SL3;

; SLB - <<lp>+alpha>>TOS (popped)

SLB:       IR=sr6, :Getalpha;
SLBr:      T=nlpoffset+T+1, SH=0, :SL3;  returns to SLBr
           undiddle lp, pick up ball 1

; SLOB - <<lp>+alpha+1>>TOS (popped), <<lp>+alpha>>TOS (popped), assuming:
; SLOB is A-aligned (also ensures no pending branch at entry)

SLOB:      T=lp, :SDcommon;
SDcommon:  T=nlpoffset+T+1, :Dop;
; SGN - <<gp>+n>+TOS (popped)
; Note: SG2 must be odd!

; Note: gp is offset by 1, hence the adjustments below
SG0:     MAR gp-1, StoreB;
SG1:     MAR gp, StoreB;
SG2:     MAR gp+1, StoreB;
SG3:     MAR gp+2, StoreB;

; SGB - <<gp>+alpha>+TOS (popped)
SGB:     IR sr7, Getalpha;   returns to SGBr
SGBr:    T=ngpoffset+1, SH=0, SG2;
          undiddle gp, pick up ball 1

; SGDB - <<gp>+alpha+1>+TOS (popped), <<gp>+alpha>+TOS (popped), assuming:
; SGDB is A-aligned (also ensures no pending branch at entry)
SGDB:    T=gp+1, SDcommon;
          T: gp-gpoffset+1poffset
; P u t s
;

;---------------------------------------------------------------
; P L n - <<lp>+n>TOS (stack is not popped)
;---------------------------------------------------------------
!1,1,PLcommon;  

; Note: lp is offset by 2, hence the adjustments below

PLO: MAR+lp-T-1, SH=0, :PLcommon;  pick up ball 1
PL1: MAR+lp-1, SH=0, :PLcommon;
PL2: MAR+lp, SH=0, :PLcommon;
PL3: MAR+lp+T, SH=0, :PLcommon;

PLcommon: L=stkp, BUS, :StoreBa;  don't decrement stkp
### Binary Operations

Warning! Before altering this list, be certain you understand the additional addressing requirements imposed on some of these return locations! However, it is safe to add new return points at the end of the list.

37,40,ADDr,SUBr,ANDr,ORr,XORr,MULr,DIVr,LDIVr,SHIFTr,EXCHr,RSTRr,WSTRr,WSBr,WSOr,WSFr,WFr, WSDBrb,WFBr,BNDCKr,

---

### Binary Operations Common Code

#### Entry Conditions:
- Both IR and T hold return number. (More precisely, entry at 'BincomB' requires return number in IR, entry at 'BincomA' requires return number in T.)

#### Exit Conditions:
- Left operand in L (M), right operand in T
- stkp positioned for subsequent push (i.e. points at left operand)
- Dispatch pending (for pushO) on return
- If entry occurred at BincomA, IR has been modified so that mACSOURCE will produce 1

#### Dispatches on stk-1, so Binpop1 = 1 mod 208

17,20,Binpop,Binpop1,Binpop2,Binpop3,Binpop4,Binpop5,Binpop6,Binpop7,

1,2,BincomB,BincomA;
4,1,Bincomx; shake IR+ in BincomA

#### BincomB:
- L+T-stkp-1, :Bincomx;
- value for dispatch into Binpop

#### Bincomx:
- stkp=L, L=T;
- L=M-1, BUS, TASK;
- L: value for push dispatch
- stash briefly

#### BincomA:
- L=2000 OR T;
- make mACSOURCE produce 1

#### Binpop:
- IR=M, :BincomB;
- IR+M, :BincomB;

#### Binpop1:
- T=stk1;
- T=stk1, :Binend;

#### Binpop2:
- L=stk0, :Binend;
- L=stk1, :Binend;

#### Binpop3:
- T=stk2;
- T=stk3;

#### Binpop4:
- L=stk2, :Binend;
- L=stk3, :Binend;

#### Binpop5:
- T=stk4;
- T=stk5;

#### Binpop6:
- L=stk4, :Binend;
- L=stk5, :Binend;

#### Binpop7:
- T=stk6;
- T=stk7;

#### Binend:
- SINK=DISP, BUS;
- SINK=disp, BUS, :ADDr;
- perform return dispatch
- perform push dispatch
; ADD - replace <TOS> with sum of top two stack elements

ADD: IR=T-ret0, :BincomB;
ADDr: L=M+T, mACSOURCE, TASK, :push0; M addressing unaffected

; ADD01 - replace stk0 with <stk0>+<stk1>

!1,1,ADD01x; drop ball 1
ADD01: T=stk1-1, :ADD01x;
ADD01x: T=stk0+T+1, SH=0;
L=stk0-1, :pushTIB; pick up ball 1
no dispatch => to push0

; SUB - replace <TOS> with difference of top two stack elements

SUB: IR=T-ret1, :BincomB;
SUBr: L=M-T, mACSOURCE, TASK, :push0; M addressing unaffected

; AND - replace <TOS> with AND of top two stack elements

AND: IR=T-ret2, :BincomB;
ANDr: L=M AND T, mACSOURCE, TASK, :push0; M addressing unaffected

; OR - replace <TOS> with OR of top two stack elements

OR: IR=T-ret3, :BincomB;
ORr: L=M OR T, mACSOURCE, TASK, :push0; M addressing unaffected

; XOR - replace <TOS> with XOR of top two stack elements

XOR: IR=T-ret4, :BincomB;
XORr: L=M XOR T, mACSOURCE, TASK, :push0; M addressing unaffected
; MUL - replace <TOS> with product of top two stack elements
; high-order bits of product recoverable by PUSH
;----------------------------------------------------------
!1,1,MULDIVcoma;
!1,2,GoROMMUL,GoROMDIV;
!7,2,MULx,DIVx:

MUL: IR=T=ret5, :BincomB;
MULr: AC1=L, L+T, :MULDIVcoma;
MULDIVcoma: AC2=L, L+0, :MULx;
MULx: AC0=L, T+0, :MULDIVcomb;
DIVx: AC0=L, T+0+1, BUS=0, :MULDIVcomb;
MULDIVcomb: L=MULDIVretloc-T-1, SWMODE, :GoROMMUL;
GoROMMUL: PC=L, :ROMMUL;
GoROMDIV: PC=L, :ROMDIV;
MULDIVret: :MULDIVret1;
MULDIVret1: T=AC1;
       L=stkp+1;
       L+T, SINK=M, BUS;
       T=AC0, :dpush;

; DIV - push quotient of top two stack elements (popped)
; remainder recoverable by PUSH
;----------------------------------------------------------
DIV: IR=T=ret6, :BincomB;
DIVr: AC1=L, L+T, BUS=0, :MULDIVcoma;

; LDIV - push quotient of <TOS-1>,<TOS-2>/<TOS> (all popped)
; remainder recoverable by PUSH
;----------------------------------------------------------
LDIV: IR=sr27, :Popsub;
LDIVr: AC2=L;
       IR=T=ret7, :BincomB;
LDIVr: AC1=L, L+T, IR+0, :DIVx;

shakes stack dispatch
also shakes bus dispatch
shakes stack dispatch
also shakes bus dispatch
ac1
L, T<<0,
MULDIVret loc-T-1,
SWMODE,
GoROMMUL;
go to ROM multiply

No divide - someday a trap
perhaps, but garbage now.
Normal return
Note! not a subroutine
call, but a direct
dispatch.

get divisor
stash it
L: low bits, T: high bits
stash low part of dividend
and ensure mACSOURCE of 0.
SHIFT - replace <TOS> with <TOS-1> shifted by <TOS>

<TOS> 0 => left shift, <TOS> < 0 => right shift

shakes stack dispatch

!7,1,SHIFTx;
!1,2,Lshift,Rshift;
!1,2,DoShift,Shiftdone;
!1,2,DoRight,DoLeft;
!1,1,Shiftdonex;

SHIFT:
IR=T=setio, :BincomB;

SHIFTr:
temp=L, L=T, TASK, :SHIFTx;

SHIFTx:
count=L;
L=T=count;
L=-count, SH<0;
IR=sl, :Lshift;

Lshift:
L=37 AND T, TASK, :Shiftcom;

Rshift:
T=37, IR=37;
L=M AND T, TASK, :Shiftcom;

Shiftcom:
count=L, :Shiftloop;

Shiftloop:
L=count-1, BUS=0;
count=L, IDISP, :DoShift;

DoShift:
L=temp, TASK, :DoRight;

DoRight:
temp=L RSH 1, :Shiftloop;

DoLeft:
temp=L LSH 1, :Shiftloop;

Shiftdone:
SINK=temp2, BUS, :Shiftdonex;

Shiftdonex:
L=temp, TASK, :push0;
; Double-Precision Arithmetic

; DSUB - subtract two double-word quantities, assuming:
; stack contains precisely 4 elements

!1,1,DSUBsub:
!3,4,DASTail...,DCOMP;
!1,1,Dsetstkp;

; DADD - add two double-word quantities, assuming:
; stack contains precisely 4 elements

!1,1,DADDx;
!1,2,DADDnocarry,DADDcarry:

DADD:
T:low bits of right operand
L:low half of sum
stash, test carry
T:high bits of right operand

DADDx:
stk2~T;
stk0~L, ALUCY;
stk3~T;

DADDnocarry:
stk1~T;
stk1~L;

DADDcarry:
stk1~T+1;
stk1~L;

; DSUB - subtract two double-word quantities, assuming:
; stack contains precisely 4 elements

DSUB:
IR~msr0, :DSUBsub;

; Double-precision subtract subroutine

; DSUBborrow,DSUBnoborrow:

!1,2,DSUBborrow,DSUBnoborrow:
!7,1,DSUBx:

DSUBsub:
T:low bits of right operand
L:low half of difference
borrow = -carry
T:high bits of right operand

DSUBx:
stk2~T;
stk0~L, ALUCY;
stk3~T;

DSUBborrow:
stk1~T-1, IDISP, :DASTail;

DSUBnoborrow:
stk1~T, IDISP, :DASTail;

; Common exit code

DASTail:
stk1~L, ALUCY, :DASTail;
DASTail:
T:2, :Dsetstkp;
Dsetstkp:
L:stkp-T, TASK, :Setstkp;
DCOMP - compare two long integers, assuming:
stack contains precisely 4 elements
result left on stack is -1, 0, or +1 (single-precision)
(i.e. result = sign(stk1, stk0 DSUB stk3, stk2)

DUCOMP - compare two long cardinals, assuming:
stack contains precisely 4 elements
result left on stack is -1, 0, or +1 (single-precision)
(i.e. result = sign(stk1, stk0 DSUB stk3, stk2)
Range Checking

NILCK - check TOS for NIL (0), trap if so

\[
\begin{align*}
\text{NILCK:} & \quad \text{L=ret17, :Xpopsub;} \quad \text{returns to NILCKr} \\
\text{NILCKr:} & \quad \text{T=ONE, SH=0, :NILCKpush;} \quad \text{test TOS=0} \\
\text{NILCKpush:} & \quad \text{L=stk\text{p}+T, :InRange;} \\
\text{InRange:} & \quad \text{SINK=ib, BUS=0, TASK, :Setstkp;} \quad \text{pick up ball 1} \\
\text{OutOfRange:} & \quad \text{T=BoundsFaultm1+T1, :KFCr;} \quad \text{T:SD index; go trap} \\
\end{align*}
\]

BNDCK - check subrange inclusion

if TOS-1 \text{-IN [0..TOS]} then trap (test is unsigned)

only TOS is popped off

\[
\begin{align*}
\text{BNDCK:} & \quad \text{IR=T=ret22, :BincomB;} \quad \text{returns to BNDCKr} \\
\text{BNDCKr:} & \quad \text{L=M-T, :BNDCKx;} \quad \text{L: value, T: limit} \\
\text{BNDCKx:} & \quad \text{T=0, ALUCY, :NILCKpush;} \\
\end{align*}
\]
; Reads
; Note: RBr must be odd!

; An - TOS<<TOS>+n>

R0: T=0, SH=0, :RBr;
R1: T=ONE, SH=0, :RBr;
R2: T=2, SH=0, :RBr;
R3: T=3, SH=0, :RBr;
R4: T=4, SH=0, :RBr;

; RB - TOS<<TOS>+alpha>, assuming:

!1,2,ReadB,ReadA; keep ball 1 in air
RB: IR=sr15, :Getalpha;
RBr: L=stkp-1, BUS, :ReadB;
ReadB: stkp=L, :MAStkT;
ReadA: stkp=L, BUS=0, :MAStkT;

; RDB - temp<<TOS>+alpha, push <<temp>>, push <<temp>+1>, assuming:
; RDB is A-aligned (also ensures no pending branch at entry)
RDB: IR=sr30, :Popsub;

; RD0 - temp<<TOS>, push <<temp>>, push <<temp>+1>
RD0: IR=sr32, :Popsub;
RD0r: L=0, :Dpusha;
RILP - push <<lp>alpha[0-3]+alpha[4-7]>

RILP:  L=ret0, :Splitalpha;  get two 4-bit values
       T=lp, :RIPcom;  T:address of local 2

RILPr: ...

RIGP - push <<gp>+alpha[0-3]+alpha[4-7]>

RIGP:  L=ret1, :Splitalpha;  get two 4-bit values
       T=gp+1, :RIPcom;  T:address of global 2

RIGPr: ...

set up return to pushMD

T:address of local or global 0

start memory cycle

T:local/global value

start fetch/store

RILO - push <<lp>>

RILO:  MAR=lp-T-1, :RILxB;  fetch local 0
       RILxB:  IR=msr0, L=0, :IPcomx;  to pushMD
       RILxA:  IR=srl1, L=srl1 AND T, :IPcomx;  to pushMDA, L=0(!)

RXLP - TOS<<TOS>+<<lp>+alpha[0-3]+alpha[4-7]>

RXLP:  L=ret3, :Splitalpha;  will return to RXLPra
       RXLPra:  IR=sr34, :Popsub;
       RXLPrb:  L=righthalf+T, TASK;
                  righthalf=L, :RILPr;  fetch TOS
                  L:TOS+alpha[4-7]
                  now act like RILP
; W n - <<TOS> (popped)+n>-><TOS> (popped)

!1,2,WnB,WnA;  
keep ball 1 in air

WO:  T=0, :WnB;
W1:  T=ONE, :WnB;
W2:  T=2, :WnB;
WnB:  IR=sr2, :Wsub;
WnA:  IR=sr3, :Wsub;

; W nB - (popped)

returns to StoreB

returns to StoreA

; W nA - (popped)

; Write subroutine:

!7,1,Wsubx;

; WB - <<TOS> (popped)+alpha>-><TOS-1> (popped)

; WB - act like WB but with stack values reversed, assuming: WSB is A-aligned (also ensures no pending branch at entry)

; WSO - act like WSB but with alpha value of zero

!7,1,WSBr;

alignment requires BincomA

WSB:  IR=T-ret14, :BincomA;
WSBr:  T=M, L=T, :WSBr;
WSBx:  MAR=Lb+T, :WScom;
WScom:  temp=temp+L;
WScomA:  L=stk-1;
       MD=temp;
       MACSOURCE, TASK, :Setstk;

WSO:  IR=T-ret15, :BincomB;
WSOr:  T=M, L=T, :WSOr;
WSOx:  MAR=L, :WScom;

WSOx - act like WSB but with alpha value of zero

shake stack dispatch

shake stack dispatch
; WILP - <<lp>+alpha[0-3]>+alpha[4-7] + <TOS> (popped)

WILP:    L=ret2, :Splitalpha;
WILPr:     IR=sr2;
            T=lp, :IPcom;

; WXLP - <TOS> + <<lp>+alpha[0-3]>+alpha[4-7] + <TOS> -1 (both popped)

WXLP:    L=ret4, :Splitalpha;
WXLPra:   IR=sr35, :Popsub;
            L=righthalf+T, TASK;
            righthalf=L, :WILPr;

; WOB - temp'alpha+<TOS> (popped), pop into <temp>+1 and <temp>, assuming:
; WOB is A-aligned (also ensures no pending branch at entry)

WOB:     IR=sr31, :Popsub;

; WOO - temp'alpha+<TOS> (popped), pop into <temp>+1 and <temp>

WOO:     L=ret6, TASK, :Xpopsub;
            L<-O, :Opopa;

; WSDB - like WDB but with address below data words, assuming:
; WSDB is A-aligned (also ensures no pending branch at entry)

WSDB:     IR=sr24, :Popsub;
            saveret=L;
            IR=T=ret20, :BincomA;
            T=M, L=T, :WSDBx;
            MAR+T=ib+T+1;
            temp=L, L=T;
            temp2=L, TASK;
            MD=saveret;
            MAR=temp2-1, :WScoma;

; WDB - temp'=alpha+<TOS> (popped), pop into <temp>+1 and <temp>, assuming:
; WDB is A-aligned (also ensures no pending branch at entry)
; Unary operations

; INC - TOS + <TOS>+1

INC:              IR=sr14, :Popsub;
INCr:             T=0+T+1, :pushTB;

; NEG - TOS - <TOS>

NEG:              L=ret11, TASK, :xpopsub;
NEGr:             L=O-T, :Untail;

; DBL - TOS = 2*<TOS>

DBL:              IR=sr25, :Popsub;
DBLr:             L=M+T, :Untail;

; Unary operation common code

Untail:           T=M, :pushTB;
; Stack and Miscellaneous Operations

; PUSH - add 1 to stack pointer
; !1,1,PUSHx;

PUSH: L=stk+1, BUS, :PUSHx;    BUS checks for overflow
PUSHx: SINK+ib, BUS=0, TASK, :Setstk;  pick up ball 1

; POP - subtract 1 from stack pointer
; !1,1,POPx;

POP: L=stk-1, SH=0, TASK, :Setstk;  L=0 <=> branch 1 pending
; need not check stk=0

; DUP - temp< TOS> (popped), push <temp>, push <temp>
; !1,1,DUPx;

DUP: IR=sr2, :DUPx;  returns to pushTB
DUPx: L=stk, BUS, TASK, :Popsuba;  don't pop stack

; EXCH - exchange top two stack elements
; !1,1,EXCHx;

EXCH: IR=ret2, :EXCHx;  drop ball 1
EXCHx: L=stk-1;
EXCHr: T=M+1, BUS, TASK, :Bincomd;  dispatch on stk-1
EXCHR: L=stk, L=T, :dpush;  set temp2=stk
        Note: dispatch using temp2

; LADRB - push alpha+lp (undiddled)
; !1,1,LADRBx;

LADRB: IR=sr10, :Getalpha;
LADRBx: T=n1poffset+T+1, :LADRBx;  shake branch from Getalpha
LADRBx: L=lp+T, :Untail;

; GADRB - push alpha+gp (undiddled)
; !1,1,GADRBx;

GADRB: IR=sr11, :Getalpha;
GADRBx: T=ngpoffset+T+1, :GADRBx;  shake branch from Getalpha
GADRBx: L=gp+T, :Untail;  returns to GADRBx
;  String Operations
;  !7.1,STRsub;   shake stack dispatch
;  !1.2,STRsubA,STRsubB;
;  !1.2,RSTRrx,WSTRrx;

STRsub:  L=stkp-1;
        stkp=L;
        L=Lb+T;
        SINK=M, BUSODD, TASK;
        count=L RSH 1, :STRsubA:

STRsubA: L=177400, :STRsubcom;
        STRsub:

STRsubB: L=377, :STRsubcom;
        STRsub:

STRsubcom: T= temp;
        MAR= count+T;
        T=M;
        SINK= DISP, BUSODD;
        mask=L, SH<0, :RSTRrx;

; RSTR - push byte of string using base (<TOS-1>) and index (<TOS>)
;  assumes RSTR is A-aligned (no pending branch at entry)
;  !1.2, RSTRB, RSTRA;

RSTR:    IR=T=ret12, :BincomB;
RSTRr:   temp=L, :STRsub;
RSTRrx:  L=MD AND T, TASK, :RSTRB;

RSTRB:   temp=L, :RSTRcom;
RSTRA:   temp=L LCY 8, :RSTRcom;
RSTRcom: T= temp, :pushTA;

; WSTR - pop <TOS-2> into string byte using base (<TOS-1>) and index (<TOS>)
;  assumes WSTR is A-aligned (no pending branch at entry)
;  !1.2, WSTRB, WSTRA;

WSTR:    IR=T=ret13, :BincomB;
WSTRA:   temp=L, :STRsub;
WSTRAx:  L=MD AND NOT T, :WSTRA;
WSTRB:   temp2=L, L=ret0, TASK, :Xpopsub;
WSTRA:   temp2=L, L=ret0+1, TASK, :Xpopsub;
WSTRAx:  taskhole=L LCY 8;
WSTRB:   T= taskhole, :WSTRA;

WSTRA:   T=mask.T;
L= temp2 OR T;
T= temp;
MAR= count+T;
TASK;
MD=M, :nextA;
Field Instructions

!1.2.RFrr,WFrr;
!7.1.Fieldsub;
!7.1.WFr: (required by WSFr) is implicit in ret17 (!)

 RF - push field specified by beta in word at <TOS> (popped) + alpha
   if RF is A-aligned, B byte is irrelevant
   alpha in B byte, beta in A byte of word after RF

            IR<-sr12, :Popsub;
            L<-ret0, :Fieldsub;
            T<-mask.T, :pushTA;

           alignment requires pushTA

 WF - pop data in <TOS-1> into field specified by beta in word at <TOS> (popped) + alpha
   alpha in B byte, beta in A byte of word after WF

            L: new data, T: address
            (actually, L:ret1)
            IR<-T<-ret17, :BincomB;
            newfield=L, L<-ret0+1, :Fieldsub;
            T<-mask.T, :WFrr;

            set old field bits to zero
            save new field bits
            merge old and new
            stash briefly
            get position, test for zero
            get return address from ROM
            PC=L;
            L<-20-T, SWMODE;
            T=-CYCOUT, :RAMCYCX;
            MF<-frame;
            L<-stkp-1;
            MD=CYCOUT, TASK, :JZNEBeq;

            stash return
            go remaining count to cycle
            pop remaining word
            stash data, go update stkp

 WSF - like WF, but with top two stack elements reversed
   if WSF is A-aligned, B byte is irrelevant
   alpha in B byte, beta in A byte of word after WSF

            L: address, T: new data
            IR<-T<-ret16, :BincomB;
            L=T, T=M, :WFr;
RFS - like RF, but with a word containing alpha and beta on top of stack
if RFS is A-aligned, B byte is irrelevant

RFS:    L-ret12, TASK, :Xpopsub; get alpha and beta
        temp=L; stash for WFSa
RFSra:  L-ret13, TASK, :Xpopsub; T:address
RFSrb:  L-ret0, BUS=0, :Fieldsub; returns quickly to WFSa

WFS - like WF, but with a word containing alpha and beta on top of stack
if WFS is A-aligned, B byte is irrelevant

WFS:    L-ret14, TASK, :Xpopsub; get alpha and beta
        temp=L; stash temporarily
        IR-T-ret21, :BincomB; L:new data, T:address
WFSra:  L-ret0+1, BUS=0, :Fieldsub; returns quickly to WFSa
WFSrb:  frame=L; stash address
        T-177400; to separate alpha and beta
WFSa:   L-temp AND T, T-temp, :Getalphab; L:alpha, T:both
        T:address returns to Fieldra

RFC - like RF, but uses <cp>+<alpha>+<TOS> as address
if RFC is A-aligned, B byte is irrelevant
alpha in B byte, beta in A byte of word after RF

RFC:    L-ret16, TASK, :Xpopsub; get index into code segment
RFCr:   L=cp+T; T=M, :RFr; T:address
Field instructions common code

Entry conditions:
L holds return offset
T holds base address

Exit conditions:
mask: right-justified mask
frame: updated address, including alpha
index: left cycles needed to right-justify field [0-15]
L,T: data word from location <frame> cycled left <index> bits

Fieldsub:
Fieldsuba:
Fieldra:
Fieldrb:
Fieldrc:
Frame Allocation

Alloc subroutine:
  allocates a frame
  Entry conditions:
  frame size index (fsi) in T
  Exit conditions:
  frame pointer in L, T, and frame
  if allocation fails, alternate return address is taken and
  temp2 is shifted left by 1 (for ALLOC)

 ALLOCr,XferGr;
 ALLOCr,XferGrf;
 AllocO,Alloc1,Alloc2,Alloc3;
 !17,1,Alloc:

 AllocSub:

 Alloc:

 AllocO:

 Alloc1:

 Alloc2:

 Allocp:

 Alloc3:

 subroutine returns
 failure returns
 dispatch on pointer flag
 shake IR dispatch
 fetch av entry
 save av entry address
 mask for pointer flags
 start reading pointer
 branch on bits 14:15
 new entry for frame vector
 new value of vector entry
 update frame vector
 establish exit conditions
 update and return
 restore argument, take failure return
 restore parameter
 allocation failed
 indirection: index+index/4
 (treat type 3 as type 2)
Free subroutine:
  frees a frame
  Entry conditions: address of frame is in 'frame'
  Exit conditions: 'frame' left pointing at released frame (for LSTF)

!3,4,RETr,FREEr,LSTFr,;
!17,1,Freex;
FreeSub: MAR=frame-1;
  NOP;
  T=MD;
  L=MAR+avm1+T+1;
  entry=L;
  L=MD;
  MAR=frame;
  temp=L, TASK;
  MD=temp;
  MAR=entry;
  IDISP, TASK;
  MD=frame, :RETr;

FreeSub returns
  shake IR= dispatch
  start read of fsi word
  wait for memory
  T=index
  fetch av entry
  save av entry address
  read current pointer
  write it into current frame
  write!
  entry points at frame
  free
; ALLOC - allocate a frame whose fsi is specified by <TOS> (popped)

; 1.1, Savpcinframe;
; 7.10, XferGT, Xfer, Mstopr, PORTOpc, LSTr, ALLOCfrf, ..
; 1.2, doAllocTrap, XferGfz;

ALLOC:
ALLOCr:
ALLOCfr:

L<-ret7, TASK, :Xpopsub;
temp2<-L LSH 1, IR"ms rO, :AllocSub;
L<-stkp+1, BUS, :pushTIB;

returns to ALLOCr

 Inform software that allocation failed

; doAllocTrap: ATPreg+-L;
T<-sAllocTrap, :Mtrap;

FREE - release the frame whose address is <TOS> (popped)

FREE:
FREEr:
FREEfr:

L<-ret10, TASK, :Xpopsub;
frame=-L, TASK;
IR"sr1, :FreeSub;
next;
; Descriptor Instructions

; DESCB - push <<gp>+gfi offset>+2*alpha+1 (masking gfi word appropriately)
; DESCB is assumed to be A-aligned (no pending branch at entry)

DESCB:
  T=gp;
  T=ngpoffset+T+1, :DESCBcom;

DESCBcom:
  MAR=gfioffset+T;
  T=gfimask;
  T=M+T;
  L=ib+T, T=ib;
  T=M+T+1, :pushTA;

; DESCBS - push <<TOS>+gfi offset>+2*alpha+1 (masking gfi word appropriately)
; DESCBS is assumed to be A-aligned (no pending branch at entry)

DESCBS:
  L=ret15, TASK, :Xpopsub;

returns to DESCBcom
Transfer Operations

Savpcinframe subroutine:
  stashes C-relative (mpc,ib) in current local frame
  undiddles lp into my and lp
  Entry conditions: none
  Exit conditions:
    current frame+1 holds pc relative to code segment base (+ = even, - = odd)
    lp is undiddled
    my has undiddled lp (source link for Xfer)

\[
\begin{align*}
\text{Savpcx:} & \quad L=\text{mpc}-T; \\
\text{SINK} & = \text{ib}, \text{BUS}=0; \\
\text{T} & = \text{M}, \text{:Spcodd}; \\
\text{Spcodd:} & \quad L=0-T, \text{TASK}, \text{:Spcopc}; \\
\text{Spcopc:} & \quad \text{taskhole}+L; \\
\text{L} & = 0; \\
\text{T} & = \text{npcoffset}; \\
\text{MAR} & = \text{lp}-T, \text{T}+\text{lp}; \\
\text{ib} & = L; \\
\text{L} & = \text{npcoffset}+T+1; \\
\text{MD} & = \text{taskhole}; \\
\text{my} & = L, \text{IDISP}, \text{TASK}; \\
\text{lp} & = \text{:XferGT};
\end{align*}
\]

Savpcinframe: \quad T=cp, :Savpcx;

\begin{align*}
\text{Savpcx:} & \quad L=\text{mpc}-T; \\
\text{SINK} & = \text{ib}, \text{BUS}=0; \\
\text{T} & = \text{M}, \text{:Spcodd}; \\
\text{Spcodd:} & \quad L=0-T, \text{TASK}, \text{:Spcopc}; \\
\text{Spcopc:} & \quad \text{taskhole}+L; \\
\text{L} & = 0; \\
\text{T} & = \text{npcoffset}; \\
\text{MAR} & = \text{lp}-T, \text{T}+\text{lp}; \\
\text{ib} & = L; \\
\text{L} & = \text{npcoffset}+T+1; \\
\text{MD} & = \text{taskhole}; \\
\text{my} & = L, \text{IDISP}, \text{TASK}; \\
\text{lp} & = \text{:XferGT};
\end{align*}
Loadgc subroutine:
Load global pointer and code pointer given local pointer or GFT pointer
Entry conditions:
T contains either local frame pointer or GFT pointer
memory fetch of T has been started
pending branch (1) catches zero pointer
Exit conditions:
lp diddled (to framebase+6)
mpc set from second word of entry (PC or EV offset)
first word of code segment set to 1 (used by code swapper)
Assumes only 2 callers

!1.2.XferOr,XferIr;
!1.2.Loadgc.LoadgcTrap;
!1.2.LoadgcOK,LoadgcNull;
!1.2.LoadgcIn,LoadgcSwap;

Loadgc:  L=lpoffset+T;
          lp=L;
          T=M;
          L=M;
          MAR=cpoffset+T;
          mpc=L, L=T;
          L=gpoffset+T, SH=0;
          T=M, BUSODD, :LoadgcOK;

LoadgcOK:  MAR=T, :LoadgcIn;

LoadgcIn:  gp=L, L=T;
          cp=L, IDISP, TASK;
          MD=ONE, :XferOr;

; picked up global frame of zero somewhere, call it unbound
; !1.1,Stashmx;
LoadgcNull:  T=sUnbound, :Stashmx;

; swapped code segment, trap to software
; LoadgcSwap:  T=sSwapTrap, :Stashmx;

; destination link = 0
; LoadgcTrap:  T=sControlFault, :Mtrap;
CheckXferTrap subroutine:
Handles Xfer trapping

Entry conditions:
IR: return number in DISP
T: parameter to be passed to trap routine

Exit conditions:
if trapping enabled, initiates trap and doesn’t return.

CheckXferTrap: L=XTSreg, BUSODD;
SINK=DISP, BUS, :NoXferTrap;
returns from CheckXferTrap

NoXferTrap: XTSreg=L RSH 1, :Xfers;
reset XTSreg[15] to 0 or 1

DoXferTrap: L=DISP, :DoXferTrapx;
tell trap handler which case
DoXferTrapx: XTSreg=L LCY 8, L=T;
XTreg=L;
T=sXferTrap, :Mtrap;
off to trap sequence
Xfer open subroutine:
  decodes general destination link for Xfer
  Entry conditions:
  source link in my
  destination link in mx
  Exit conditions:
  if destination is frame pointer, does complete xfer and exits to Ifetch.
  if destination is procedure descriptor, locates global frame and entry
  number, then exits to 'XferG'.

Xfer0: Xfer1, Xfer2, Xfer3;
  T=mx;
  IR=0, :CheckXferTrap;
  L=3 AND T;
  SINK=M, L=T, BUS;
  SH=0, MAR=T, :Xfer0;

mx[14:15] = 00
  Destination link is frame pointer

Xfer0: IR=msr0, :Loadgc;
  L=T=mpc;

Dead trap condition:
  If 'brkbyte' = 0, we are proceeding from a breakpoint.
  pc points to the BRK instruction:
  even pc => fetch word, stash left byte in ib, and execute brkbyte
  odd pc = clear ib, execute brkbyte

Xfer0B: L=MAR=cp+T; :nextAdeafa;
  L=MAR=cp-T, SH=0, :nextBdeaf;

Not proceeding from a breakpoint - simply pick up next instruction

Xfer1: IB=L, :XbrkB;
  IR=sr20;
  L=MAR=cp+T, :GetalphaAx;

Xfer2: L=cp-T;
  mpc=L, L=0, BUS=0, :XbrkBr;

Xfer3: SINK=brkbyte, BUS=0;
  SH<0, L=0, :Xdodbreak;

Proceeding from a breakpoint - dispatch brkbyte and clear it

XbrkB: IR=sr20;
  L=MAR=cp+T, :GetalphaAx;

XbrkA: L=cp-T;
  mpc=L, L=0, BUS=0, :XbrkBr;

XbrkBr: SINK=brkbyte, BUS, :XbrkBgo;
  clear brkbyte, act like nextA

XbrkBgo: brkbyte=L, RSH 1, T=0+1, :NOOP;
  XbrkAgo: brkbyte=L, T=0+1, BUS=0, :NOOP;
  dispatch brkbyte

set up by Loadstate
  dispatch even/odd pc

offset from cp: - odd. + even

here if BRK at even byte
  set up ib (return to XbrkBr)

here if BRK at odd byte
  ib already zero (to Xbrk ago)

dispatch brkbyte

clear brkbyte, act like nextA
  clear brkbyte, act like next
\[
\text{mx}[14-15] = 01 \\
\text{Destination link is procedure descriptor:}
\]
\[
\text{mx}[0-8]: \text{GFT index (gfi)} \\
\text{mx}[9-13]: \text{EV bias, or entry number (en)}
\]

Xfer1:
\[
\text{temp}=L \text{ RSH 1; } \\
\text{count}=L \text{ MSH 1; } \\
\text{L}=\text{count}, \text{ TASK; } \\
\text{count}=L \text{ LCY 1; } \\
\text{T}=\text{count}; \\
T=1777.T; \\
\text{MAR}=\text{gftm}1+T+1; \\
\text{IR}=sr1, : \text{Loadgc; }
\]

Xfer1r:
\[
\text{L}=\text{temp}, \text{ TASK; } \\
\text{count}=L \text{ RSH 1; } \\
T=\text{count; } \\
T=\text{enmask.T; } \\
\text{L}=\text{mpc}+T+1, \text{ TASK; } \\
\text{count}=L \text{ LSH 1, : XferG; }
\]

Xfer2:
\[
\text{NOP; } \\
T=\text{MD, : Xfers; }
\]

Xfer3:
\[
T=s\text{Unbound, : Stashmx; }
\]
XferG open subroutine:
allocates new frame and patches links
Entry conditions:
'count' holds index into code segment entry vector
assumes lp is undiddled (in case of AllocTrap)
assumes gp (undiddled) and cp set up
Exit conditions:
exits to instruction fetch (or AllocTrap)

---

; Pick up new pc from specified entry in entry vector

XferGT:  T=count;
         IR=ONE, :CheckXferTrap;

XferG:   T=count;
         MAR=cp+T;
         T=cp-1;
         IR=s1;
         L=MD+T;
         T=MD;
         mpc=L;
         T=377.T, :AllocSub;

; Stash source link in new frame, establishing dynamic link

XferGr:  MAR=retlinkoffset+T;
         L=lpoffset+T;
         lp=L;
         MD=my;

; Stash new global pointer in new frame (same for local call)

         MAR=T;
         T=gpoffset;
         L=gp-T, TASK;
         MD=M, :nextAdeaf;

; Frame allocation failed - push destination link, then trap

!l,2,doAllocTrap,XferGfz;

(appears with ALLOC)

XferGfz:  L=mx, BUS=0;
          T=count-1, :doAllocTrap;
          if destination link is zero (i.e. local procedure call), we must first
          fabricate the destination link

         L=T, T=ngfifoffset;
         MAR=gp+T;
         count=L LSH 1;
         L=counter-1;
         T=gfimask;
         T=MD.T;
         L=M+T, :doAllocTrap;
Getlink subroutine:
    fetches control link from either global frame or code segment
    Entry conditions:
       temp: - (index of desired link + 1)
       IR: DISP field zero/non-zero to select return point (2 callers only)
    Exit conditions:
       L,T: desired control link

Return points
shake IR+ in KFCB

Getlink: T=gp;
         MAR+T-ngpoffset+T+1;
         L=temp+T, T=temp;
         taskhole=L;
         L=cp+T;
         SINK=MD, BUSODD, TASK;
         temp2=L, :frametlink;
         MAR=taskhole, :Fetchlink;
         MAR+temp2, :Fetchlink;
         SINK=DISP, BUS=0;
         L=T+MD, :EFCgetr;
       fetch link from frame
       fetch link from code
       dispatch to caller
; EFCn - perform XFER to destination specified by external link n

EFC0: IR=ONE, T=ONE-1, :EFCr; 0th control link
EFC1: IR=T=ONE, :EFCr; 1st control link
EFC2: IR=T=2, :EFCr;
EFC3: IR=T=3, :EFCr;
EFC4: IR=T=4, :EFCr;
EFC5: IR=T=5, :EFCr;
EFC6: IR=T=6, :EFCr;
EFC7: IR=T=7, :EFCr;
EFC8: IR=T=8, :EFCr;
EFC9: IR=T=9, :EFCr;
EFC10: IR=T=10, :EFCr;
EFC11: IR=T=11, :EFCr;
EFC12: IR=T=12, :EFCr;
EFC13: IR=T=13, :EFCr;
EFC14: IR=T=14, :EFCr;
EFC15: IR=T=15, :EFCr;
EFC16: IR=T=16, :EFCr;
EFC17: IR=T=17, :EFCr;

; EFCB - perform XFER to destination specified by external link 'alpha'

EFCB: IR=sr23, :Getalpha; shake B/A dispatch (Getalpha)
EFCr: L=O-T-1, TASK, :EFCdoGetlink; fetch link number
EFCdoGetlink: temp=L, :Getlink; L:=(link number+1)
EFCgetr: IR=sr1, :SFCr; stash index for Getlink
SFC: IR=sr1, :Popsub; for Savpcinframe; no branch
SFCr: mx=L, :Savpcinframe; set dest link, return to Xfer
; KFCB - Xfer using destination <<<SD+alpha>
; 111KFCr: implicit in KFCr's return number (21B)
; 111KFCx:
; 771Fetchlink; appears with Getlink

; KFCB: IR=sr21, :Getalpha;
; KFCr: IR=avm1, T=avm1+T+1, , :KFCx;
; KFCx: MAR=sdoffset+T, :Fetchlink;

shake B/A dispatch (Getalpha)
fetch alpha
DISP must be non zero
Fetchlink shakes IR= dispatch

; BRK - Breakpoint (equivalent to KFC 0)

; Trap sequence:
used to report various faults during Xfer
Entry conditions:
T: index in SD through which to trap
Savepcinframe has already been called
entry at Stashmx puts destination link in OTPreg before trapping

; 111.Stashmx; above with loadgc code

Stashmx: L=mx;
; OTPreg=L, :Mtrap;
can't TASK, T has trap index

Mtrap: T=avm1+T+1;
; MAR=sdoffset+T;
NOP;
fetch dest link for trap

Mtrapa: L=MD, TASK;
; mx=L, :Xfer;
(enter here from PORTO)
; LFCn - call local procedure n (i.e. within same global frame)

\[ LFC1: \quad L+2, :LFCx; \]
\[ LFC2: \quad L+3, :LFCx; \]
\[ LFC3: \quad L+4, :LFCx; \]
\[ LFC4: \quad L+5, :LFCx; \]
\[ LFC5: \quad L+6, :LFCx; \]
\[ LFC6: \quad L+7, :LFCx; \]
\[ LFC7: \quad L+10, :LFCx; \]
\[ LFC8: \quad L+11, :LFCx; \]

LFCx:
\[ \text{count}=L \quad \text{LSH} \quad 1, \quad L=0, \quad \text{IR}=\text{msr}0, \quad :SFCr; \]

; shake B/A dispatch

; LFCB - call local procedure number 'alpha' (i.e. within same global frame)

LFCB:
\[ \text{IR}=\text{sr22}, :\text{Getalpha}; \]

LFCr:
\[ L=0+T+1, :LFCx; \]
; RET - Return from function call.

!1,1.RETx;

RET:
T+=1p, :RETx;
shake B/A branch

RETx:
IR=2, :CheckXferTrap;
local pointer

RETxr:
MAR=nretlinkoffset+T;
get previous local frame
L=nlpoffset+T+1;
frame=L;
L=MD;
stash for 'free'
mx=L, L=O, IR=msr0, TASK;
pick up prev frame pointer
my=L, :FreeSub;
mx points to caller
RETn:
T-=mx, :Xfers;
clear my and go free frame
xfer back to caller

; LINKB - store back link to enclosing context into local O
; LINKB is assumed to be A-aligned (no pending branch at entry)

LINKB:
MAR=lp-T-1;
address of local O
T=ib;
L=mx-T, TASK;
L: mx-alpha
MD=M, :nextA;
local O + mx-alpha

; LLKB - push external link 'alpha'
; LLKB is assumed to be A-aligned (no pending branch at entry)

LLKB:
T=ib;
T:alpha
L=0-T-1, IR=0, :EFCdoGetlink;
L:-(alpha+1), go call Getlink
LLKBr:
pushTA;
alignment requires pushTA
Port Operations

PORTO - PORT Out (XFER thru PORT addressed by TOS)

PORTO:    IR=sr3, :Savpcinframe;  undiddle lp into my
PORTOpc:  L=ret5, TASK, :Xpopsub;  returns to PORTOr
PORTOr:   MAR=T;
         L=T;
         MD=my;
         MAR=M+1;
         my=L, :Mtrapa;
         frame addr to word 0 of PORT
         second word of PORT
         source link to PORT address

PORTI - PORT In (Fix up PORT return, always immediately after PORTO)
; assumes that my and mx remain from previous xfer
!1,1,PORTIx;
!1,2,PORTInz,PORTIz;

PORTI:       MAR=mx, :PORTIx;  first word of PORT
PORTIx:      SINK=my, BUS=0;
            TASK, :PORTInz;
PORTInz:     MD=0;
            MAR=mx+1;
            TASK, :PORTIz;
PORTIz:      MD=my, :next;
            store it as second word
            store my or zero
; State Switching

Savestate subroutine:
; saves state of pre-empted emulation
; Entry conditions:
; L holds address where state is to be saved
; assumes undiddled lp
; Exit conditions:
; lp, stkp, and stack (from base to min[depth+2,8]) saved

; !1,2.DSTr1.Mstopc; actually appears as %1,1777,776,DSTr1,Mstopc; and is located
; in the front of the main file (Mesa.mu).

!17,20,Sav0r,Savir,Sav2r,Sav3r,Sav4r,Sav5r,Sav6r,Sav7r,Sav10r,Sav11r,DSTr,....;
!1,2,Savok,Savmax;

Savestate: temp=L;
Savestatea: T=-12+1;
            L=lp, :Savsuba;
Sav1lr: L=stkpv, :Savsub;
Sav10r: T=stkpv+1;
        L=-7+T;
        L=0+T+1, ALUCY;
        temp2=L, L=0-T, :Savok;
Savmax: T=-7;
        L=stk7, :Savsuba;
Savok: SINK=temp2, BUS;
        count=L, :Sav0r;
Sav7r: L=stk6, :Savsub;
Sav6r: L=stk5, :Savsub;
Sav5r: L=stk4, :Savsub;
Sav4r: L=stk3, :Savsub;
Sav3r: L=stk2, :Savsub;
Sav2r: L=stk1, :Savsub;
Sav1r: L=stk0, :Savsub;
Sav0r: SINK=DISP, BUS;
        T=-12, :DSTr1;

; Remember, T is negative
Savsub: T=count;
Savsuba: temp2=L, L=0+T+1;
          MAR=temp-T;
          count=L, L=0-T;
          SINK=M, BUS, TASK;
          MD=temp2, :Sav0r;

            i.e. T==11
            check if stkp > 5 or negative
            stkp > 5 => save all
            stkp < 6 => save to stkp+2
            return to caller
            (for DST's benefit)

            dispatch on pos. value
Loadstate subroutine:
load state for emulation
Entry conditions:
L points to block from which state is to be loaded
Exit conditions:
stkp, mx, my, and stack (from base to min[stk+2,8]) loaded
(i.e. two words past TOS are saved, if they exist)
Note: if stkp underflows but an interrupt is taken before we detect it, the subsequent Loadstate (invoked by Mgo) will see 377B in the high byte of stkp. Thinking this a breakpoint resumption, we will load the state, then dispatch the 377 (via brkbyte) in Xfer0, causing a branch to StkuF (!) This is not a fool-proof check against a bad stkp value at entry, but it does protect against the most common kinds of stack errors.

Loadstate: temp=L, IR=msr0, NovaIntrOn;
Lsr: T=12, :Ldsb;
Lsr12: my=L, :Ldsb;
Lsr11: mx=L, :Ldsb;
Lsr10: stkp=L;
    T=stkp;
    L=177400 AND T;
    brkbyte=L LCY 8;
    L=T-177400;
    L=7+T;
    L=T, SH<0;
    stkp=L, T=0+T+1, :Lsmax;
Lsmax: T=7, :Ldsb;
Lsr7: stk7=L, :Ldsb;
Lsr6: stk6=L, :Ldsb;
Lsr5: stk5=L, :Ldsb;
Lsr4: stk4=L, :Ldsb;
Lsr3: stk3=L, :Ldsb;
Lsr2: stk2=L, :Ldsb;
Lsr1: stk1=L, :Ldsb;
Lsr0: stk0=L, :Xfer;
Ldsb: T=count;
Ldsb: MAR=temp+T;
    L=ALLONES+T;
    count=L, T=T;
    SINK=M, BUS;
    L+MD, TASK, :Lsr0;
DST - dump state at block starting at \langle LP \rangle + \alpha, reset stack pointer
assumes DST is A-aligned (also ensures no pending branch at entry)

\begin{align*}
DST:\quad & T\leftarrow ib; \\
& T\leftarrow lp+T+1; \\
& L\leftarrow lpoftsetl+T+1, TASK; \\
& temp=L, IR\leftarrow ret0, :Savestatea; \\
DSTr1:\quad & L\leftarrow my, :SaveSuba; \\
DSTr:\quad & temp=L, L\leftarrow 0, TASK, BUS=0, :Setstkp; \\
& zap stkp, return to 'nextA'
\end{align*}

LST - load state from block starting at \langle LP \rangle + \alpha
assumes LST is A-aligned (also ensures no pending branch at entry)

\begin{align*}
LST:\quad & L\leftarrow ib; \\
& temp=L, L\leftarrow 0, TASK; \\
& ib\leftarrow L; \\
& IR\leftarrow sr4, :Savpcinframe; \\
LSTr:\quad & T\leftarrow temp; \\
& L\leftarrow lp+T, TASK, :Loadstate; \\
& get alpha back \\
& \text{lp already undiddled }
\end{align*}

LSTF - load state from block starting at \langle LP \rangle + \alpha, then free frame
assumes LSTF is A-aligned (also ensures no pending branch at entry)

\begin{align*}
LSTF:\quad & T\leftarrow lpoftset; \\
& L\leftarrow lp offset, TASK; \\
& frame=L; \\
& IR\leftarrow sr2, :FreeSub; \\
LSTFr:\quad & T\leftarrow frame; \\
& L\leftarrow ib+T, TASK, :Loadstate; \\
& set up by FreeSub \\
& get state from dead frame
; Emulator Access

; RR - push <emulator register alpha>, where:
;   RR is A-aligned (also ensures no pending branch at entry)
;   alpha: 1 => wdc, 2 => XTSreg, 3 => XTPreg, 4 => ATPreg,
;   5 => OTPreg
; 7.10.RR0.RR1.RR2.RR3.RR4.RR5,;
RR:  SINK'=ib, BUS; dispatch on alpha
RR0:  :RR0;
RR1:  T=wdc, :pushTA;
RR2:  T+XTSreg, :pushTA;
RR3:  T+XTPreg, :pushTA;
RR4:  T+ATPreg, :pushTA;
RR5:  T+OTPreg, :pushTA;

; WR - emulator register alpha + <TOS> (popped), where:
;   WR is A-aligned (also ensures no pending branch at entry)
;   alpha: 1 => wdc, 2 => XTSreg
; 7.10.WR0.WR1.WR2,;
WR:  L=ret3, TASK, :Xpopsub;
WR0:  SINK'=ib, BUS; dispatch on alpha
WR1:  wdc=L, :nextA;
WR2:  XTSreg=L, :nextA;

; JRAM - JMPRAM for Mesa programs (when emulator is in ROM1)
; 7.10.JRAMr,JRAMr;
JRAM:  L=ret2, TASK, :Xpopsub;
JRAMr:  SINK=M, BUS, SWMODE, :next; BUS applied to 'nextBa' (=0)
; Process/Monitor Support

!1,MoveParms1; shake B/A dispatch
!1,MoveParms2; shake B/A dispatch
!1,MoveParms3; shake B/A dispatch
!1,MoveParms4; shake B/A dispatch

; ME, MRE - Monitor Entry and Re-entry
; MXD - Monitor Exit and Depart

!1,FastMREx; drop ball 1
!1,FastEEx; drop ball 1
!7,1,FastEExxx; shake IR=ISME/ISMXD
!1,2,MXDr,MEr; shake IR=ISMRE

%3,17,14,MXDrr,MErr,MRErr;
!1,2,FastEEtrap1,MEXDdone;
!1,2,FastEEtrap2,MREdone;

; The following constants are carefully chosen to agree with the above pre-defs

$ISME $6001;
$ISMRE $66403;
$ISMXD $402;

ME: IR=ISME, :FastEEx;
MXD: IR=ISMXD, :FastEEx;
MRE: MAR=HardMRE, :FastMREx;
FastMREx: IR=ISMRE, :MXDr;
FastEEx: MAR=stk0, IDISP, :FastEExxx;
FastEExxx: MAR=stk0, SH=O, :MXDrr;

; Note: if control goes to FastEEtrap1 or FastEEtrap2, AC1 or AC2 will be smashed,
; but their contents aren't guaranteed anyway.
; Note also that MErr and MXDrr cannot TASK.

MXDr: L=MD, mACSOURCE, :FastEExxx;
MEXDdone: MD=M, L=T, TASK, :Setstkp;

; Note: if control goes to FastEEtrap1 or FastEEtrap2, AC1 or AC2 will be smashed,
; but their contents aren't guaranteed anyway.
; Note also that MErr and MXDrr cannot TASK.

L=T, T=O, :FastEEtrap1;
L=100000, T:O (stkp value)

T=O+1, :FastEEtrap2;
L:0, T:1 (stkp value)

L=0+1, TASK, :FastEEtrap2;
L:1 (stkp value)

MD=M, L=T, TASK, :Setstkp;

queue empty, treat as ME
; MXW - Monitor Exit and Wait

MXW: IR=4, :MoveParms3; 3 parameters

; NOTIFY, BCAST - Awaken process(es) from condition variable

NOTIFY: IR=5, :MoveParms1; 1 parameter
BCAST: IR=6, :MoveParms1; 1 parameter

; REQUEUE - Move process from queue to queue

REQUEUE: IR=7, :MoveParms3; 3 parameter

; Parameter Transfer for Nova code linkages
; Entry Conditions:
; T: 1
; IR: dispatch vector index of Nova code to execute

:MoveParms4: L=stk3, TASK;
; AC3=L;
MoveParms3: L=stk2, TASK;
FastEEtrap2: AC2=L;
MoveParms2: L=stk1, TASK;
FastEEtrap1: AC1=L;
MoveParms1: L=stk0, TASK;
AC0=L;
L=0, TASK;
stkp=L;
T=DISP+1, :STOP;

if you uncomment this, don't forget the pre-def above!
(enter here from MRE)
(enter here from ME/MXD)

indicate stack empty
**Miscellaneous Operations**

- **CATCH** - an emulator no-op of length 2.
  - CATCH is assumed to be A-aligned (no pending branch at entry)

  ```
  CATCH: L=mpc+1, TASK, :nextAput;  
  duplicate of 'nextA'
  ```

- **STOP** - return to Nova at 'NovaDVloc+1'
  - control also comes here from process opcodes with T set appropriately

  ```
  !1,1,GotoNova;  
  shake B/A dispatch
  ```

- **STARTIO** - perform Nova-like I/O function

  ```
  STARTIO: L=ret4, TASK, :Xpopsub;  
  get argument in L
  STARTIOr: SINK=M, STARTF, :next;
  ```

- **MISC** - escape hatch for more than 256 opcodes

  ```
  !5,2,Dpushx,RCLKr;  
  appears with Dpush
  ```

  ```
  MISC: IR=sr36, :Getalpha;  
  get argument in L
  MISCr: L=CLOCKLOC-1, IR=CLOCKLOC, :Dpushb;  
  throws away alpha for now
  (and macsoure of 0)
  Dpushb shakes B/A dispatch
  RCLKr: L=clockreg, :Dpushc;
  don't TASK here!
  ```
; BLT - block transfer
; assumes stack has precisely three elements:
; stk0 - address of first word to read
; stk1 - count of words to move
; stk2 - address of first word to write
; the instruction is interruptible and leaves a state suitable
; for re-execution if an interrupt must be honored.

!1,1,bltx;
!1,2,blinttintd,bllloop;
!1,2,bllntoint,bllint;
!1,2,blltmore,blltodd;
!1,1,bllintx;

blt:
    stk7=l, l=t, task, :blt;
bltx:
    temp=l, :bllloop;

blntloop:
    l=t=stk1-1, bus=0, :bllntoint;
blntoint:
    stk1=l, l=bus and ~t, :blltmore;

blltmore:
    t=temp-1;
    mar=stk0+t;
    stk0=l;
    stk2=stk2;
    stk2=l, l=t;
    sink=nw, bus=0, task;
    md=m, :blltintpend;

blltintpend:
    sink=wdc, bus=0, :bllloop;

; Must take an interrupt if here (via BLT or BITBLT)

blint:
    sink=stk7, bus=0, :blintx;
blintx:
    l=mpc-1, :blt;

bltintpend:
    l=mpc-1, :blt;

bltmore:
    stk7=0 <= branch pending
    stk7=0 <=> branch pending
    stash source offset (+1)

shakes entry B/A branch

update source fetch

start source fetch

source data

start dest. write

update dest. pointer

check pending interrupts

loop or check further

check if interrupts enabled

test even/odd pc

prepare to back up

even - back up pc, clear ib

odd - set ib non-zero

bkd completed

bltodd:
    mp=1, 0, task, :bltodd;
    ib=1, :intstop;

; BLTC - block transfer from code segment
; assumes stack has precisely three elements:
; stk0 - offset from code base of first word to read
; stk1 - count of words to move
; stk2 - address of first word to write
; the instruction is interruptible and leaves a state suitable
; for re-execution if an interrupt must be honored.

!1,1,bltcx;

bltc:
    stk7=l, :bltcx;

bltcx:
    l=cp+1, task, :bltx;
    bltc: t=cp+1, task, :blt;
; BITBLT - do BITBLT using ROM subroutine
; If BITBLT A-aligned, B byte will be ignored

!1,1,BITBLT;
!7,1,DoBITBLT;
!3,4,Mstop..NovalIntrOff,DoBITBLT;

BITBLT:
    stk7=L, :BITBLT;
BITBLT:
    L=stk0, TASK;
    AC2=L;
    L=stk1, TASK;
    AC1=L;
    SINK=wdc, BUS=0;
    IR=sr3, :NovalIntrOff;
DoBITBLT:
    L=BITBLTret, SWMODE, :DoBITBLT;
DoBITBLT:
    PC=L, L=0, :ROMBITBLT;
BITBLTdone:
    IR=sr1, :NovalIntrOn;
BITBLTdone:
    brkbyte=L, BUS=0, TASK, :Setstk;
BITBLTinr:
    L=AC1, TASK;
    stk1=L, :BLTint;

* shake B/A dispatch
* shake IR= dispatch
* includes NovalIntrOff returns
* save even/odd across ROM call
* stash descriptor table
* check if Mesa interrupts off
* if so, shut off Nova's
* get return address
* L=0 for Alto II ROMO "feature"
* ensure Nova interrupts are on
* don't bother to validate stk
* pick up intermediate state
* stash instruction state
Subroutines to Enable/Disable Nova Interrupts

- 13,4,Mstop,NovaIntrOff,DoBITBLT; appears with BITBLT
- 11,2,Lsr,BITBLTdone;
- 17,1,NovaIntrOffx;

NovaIntrOff:  T=100000;
NovaIntrOffx:  L=NWW OR T, TASK, IDISP;
              NWW=L, :Mstop;
NovaIntrOn:   T=100000;
              L=NWW AND NOT T, IDISP;
              NWW=L, L=0, :Lsr;

IWDC - Increment Wakeup Disable Counter (disable interrupts)

- 11,2,IDnz,IDz;

IWDC:  L=wdc+1, TASK, :IDnz;  skip check for interrupts

DWDC - Decrement Wakeup Disable Counter (enable interrupts)

- 11,1,DWDCx;

DWDC:  MAR=WWLOC, :DWDCx;  OR WW into NWW
DWDCx:  T=NWW;
        L=MD OR T, TASK;
        NWW=L;
        SINK=ib, BUS=0;
        L=wdc-1, TASK, :IDnz;

Ensure that one instruction will execute before an interrupt is taken

IDnz:  wdc=L, :next;
IDz:   wdc=L, :next Adeaf;

Entry to Mesa Emulation

ACO holds address of current process state block
Location 'PSBloc' is assumed to hold the same value

Mgo:  L=ACO, :Loadstate;
Nova Interface

$START $L004020,0,0; Academic emulator return address

Transfer to Nova code
Entry conditions:
- L contains Nova PC to use
Exit conditions:
- Control transfers to ROMO at location 'START' to do Nova emulation
- Nova PC points to code to be executed
- Except for parameters expected by the target code, all Nova ACs contain garbage
- Nova interrupts are disabled

GotoNova: PC=L, IR=mso, :NovaIntrOff;
stash Nova PC, return to Mstop

Control comes here when an interrupt must be taken. Control will pass to the Nova emulator with interrupts enabled.

Intstop: L=NovaDVloc, TASK;
PC=L, :Mstop;
resume at Nova loc. 30B

Stash the Mesa pc and dump the current process state, then start fetching Nova instructions.

Mstop: IR=sr2, :Savpcinframe;
save mpc for Nova code
Mstopr: MAR=CurrentState;
get current state address
IR=retl;
will return to 'Mstopc'
L=MD, :Savestate;
dump the state

The following instruction must be at location 'SWRET', by convention.

Mstopc: L=UCodversion, SWMODE;
stash ucode version number
cp=L, :START;
off to the Nova ...
Horizontal bit blit.

Entry point is 5A0B. If this is used with the floating point package, put this file after the floating point entry predfs.

Entry point is 540B. The only way this routine distinguishes HBlt from XHBlt is by inspection of stkp. If its is NOT 1 then XHBlt is used instead of HBlt.

Put this file after the floating point entry vector in file Float.mu.

$ROMMUL $L004120.0.0; MUL routine address (120B) in ROMO
$SHRMCC $S600;
$SHLRMC $600;

3 and 50 should be available
Move seven parameters from memory to S registers
and calculate deltay, deltay, and init point P to point A
x1,17,0, Got0,, Got2,, Got4,, Got6; Return locs for mem reads.

11,2, Punt, Read;
11,2, Step4, Step3;

HBlt:

Got2:

Got0:

Got4:

Got6:

Read:

MAR = stk0 + T;
MAR = stk0 + T;
MAR = stk0 + T;
MAR = stk0 + T;

-- read dbmr and y
-- get ready for (y*dbmr)
-- read flags and dbca
-- read x1 and x2
-- IF (x1 <= x2) goto read
-- return to Got[0-6]
Compute the effective word begin & end address

```plaintext
addr \sim dbca + (y \times dbmr) + x1/16;
endAddr \sim dbca + (y \times dbmr) + x2/16;
```

GetAddr:

```plaintext
L \sim PC, TASK;
HoldPC \sim L;
T \sim ALLONES;
L \sim MRMC XOR T, SWMODE;
PC \sim L, :ROMMUL;
MULret: L \sim HoldPC, TASK; -- restore the return addr
PC \sim L;
T \sim dbca; -- result is in AC1
L \sim AC1 + T;
T \sim M;
```

L \sim x1:
Temp \sim L RSH 1;
L \sim Temp;
Temp \sim L RSH 1;
L \sim Temp;
Temp \sim L RSH 1;
L \sim Temp;
Temp \sim L RSH 1;
L \sim Temp + T;
addr \sim L;

L \sim x2;
Temp \sim L RSH 1;
L \sim Temp;
Temp \sim L RSH 1;
L \sim Temp;
Temp \sim L RSH 1;
L \sim Temp;
Temp \sim L RSH 1;
L \sim Temp + T, TASK:
endAddr \sim L;

T \sim 0 + 1;
L \sim flags AND T;
Mflag \sim L;
T \sim 7;
L \sim flags AND T;
OpCode \sim L RSH 1;

!1,2.UseXHB,Step1;
; -- If not HBlt then alter memory bank
L \sim stkp - 1;
T \sim 3, SH=0, :UseXHB;

UseXHB: L \sim savBnk AND NOT T;
T \sim stk1, T;
MAR \sim BnkAddr;
L \sim M OR T;
MD \sim M, :Step1;

-- mainloop has for steps:
; -- Step1: The left hand side of the blt.
; -- Step2: The right hand side of the blt.
; -- Step3: The middle of the blt.
; -- Step4: Finished.
; -- Narrow blts causes steps 1 and 3 to be performed together.
%17,37,0, LftEnd0, LftEnd1, LftEnd2, LftEnd3, LftEnd4, LftEnd5, LftEnd6, LftEnd7, LftEnd8, LftEnd9, LftEnd10, LftEnd11, LftEnd12, LftEnd13, LftEnd14, LftEnd15;
;
; -- Step 1 - Draw the furthest left hand side of the scanline.
; -- Bits [15-offset1 .. 0] are set to 1's.
Step1:
T \sim 17;
; -- SINK \sim x1 AND 17
L \sim x1 AND T;
SINK \sim M, BUS;
T \sim endAddr, :LftEnd0;
LftEnd0: L \sim ALLONES, :ContLE;
LftEnd1: L \sim 7777, :ContLE;
HBlt.mu 24-Jul-81 19:03:01

LftEnd2: L + 37777, :ContLE;
LftEnd3: L + 17777, :ContLE;
LftEnd4: L + 7777, :ContLE;
LftEnd5: L + 3777, :ContLE;
LftEnd6: L + 1777, :ContLE;
LftEnd7: L + 777, :ContLE;
LftEnd8: L + 377, :ContLE;
LftEnd9: L + 177, :ContLE;
LftEnd10: L + 77, :ContLE;
LftEnd11: L + 37, :ContLE;
LftEnd12: L + 17, :ContLE;
LftEnd13: L + 7, :ContLE;
LftEnd14: L + 3, :ContLE;
LftEnd15: L + 0 + 1, :ContLE;

!1,2, ContSt1, Stplt03;
%3,7,0, mRead, XmRead;
%3,7,0, mWrite, XmWrite;

ContLE: Smask * L;
L * addr - T;
-- T + endAddr from above
TASK, SH=0;
:ContSt1;

ContSt1: SINK * Mflag, BUS;
L * ALLONES, :mRead;
Stplt03: T * 17, :Step3;

!1,2, ContSt2, ContSt3;
%17,37,0, RhtEnd0, RhtEnd1, RhtEnd2, RhtEnd3, RhtEnd4, RhtEnd5,
RhtEnd6, RhtEnd7, RhtEnd8, RhtEnd9, RhtEnd10, RhtEnd11,
RhtEnd12, RhtEnd13, RhtEnd14, RhtEnd15;

/* Scanline. Bits [0 .. offset2] are left alone.*/
Step3:
L * x2 AND T;
-- SINK * x2 AND 17
SINK * M, BUS;
:RhtEnd0;

RhtEnd0: T * 77777, :ContRE;
RhtEnd1: T * 37777, :ContRE;
RhtEnd2: T * 17777, :ContRE;
RhtEnd3: T * 7777, :ContRE;
RhtEnd4: T * 3777, :ContRE;
RhtEnd5: T * 1777, :ContRE;
RhtEnd6: T * 777, :ContRE;
RhtEnd7: T * 377, :ContRE;
RhtEnd8: T * 177, :ContRE;
RhtEnd9: T * 77, :ContRE;
RhtEnd10: T * 37, :ContRE;
RhtEnd11: T * 17, :ContRE;
RhtEnd12: T * 7, :ContRE;
RhtEnd13: T * 3, :ContRE;
RhtEnd14: T * 0 + 1, :ContRE;
RhtEnd15: T * 0, :ContRE;

ContRE: L + Smask AND NOT T, TASK;
Smask * L;

ContSt3: SINK * Mflag, BUS;
L * ALLONES, :mRead;

/ * The 1's in Smask describes which bit locations in the word
* / That will be affected by HBlt.
%7,17,0, op0, op1, op2, op3;
mRead: MAR * addr, :ContMB;
XmRead: XMAR * addr, :ContMB;
ContMB: T * Smask;
SINK * OpCode, BUS;
Smask * L, :op0;

op0: L + MD AND NOT T;
-- dest * gray
T * gray . T;
L + M OR T, :ContBd2;

op1: T * gray . T;
-- dest * dest OR gray
L + MD OR T, :ContBd2;

op2: T * gray . T;
-- dest * dest XOR gray
L + MD XOR T, :ContBd2;
op3: \[ T = \text{gray} \cdot T; \quad \text{-- dest = (NOT gray) AND dest} \]

\[ L = \text{AllOnes} \oplus \text{XOR} \cdot T; \]

\[ T = \text{MD}; \]

\[ L = M \land T, : \text{ContBd2}; \]

ContBd2: \[ \text{SINK} = \text{Mflag}, \text{TASK}, \text{BUS}; \]

\[ \text{Temp} = L, : \text{mWrite}; \]

mWrite: \[ \text{MAR} + \text{addr}, : \text{Step2}; \]

XmWrite: \[ \text{XMAR} + \text{addr}, : \text{Step2}; \]

Step2:

\[ T = \text{endAddr} - 1; \]

\[ L = \text{addr} - T; \]

\[ \text{MD} = \text{Temp}; \]

\[ L = \text{addr} + 1, \text{SH} = 0; \]

\[ \text{addr} = L, : \text{ContSt2}; \quad \text{-- If(addr<endAddr) goto ContSt3} \]

ContSt2:

\[ L = \text{addr} - T - 1; \]

\[ \text{SH} = 0; \]

\[ T = 17, : \text{Step4}; \quad \text{-- If(addr=endAddr) goto Step3} \]

-- Step 4 is where we return to mesa.

Punt: \[ T = 0; \quad \text{-- NOP, premature exit} \]

Step4:

\[ \text{MAR} + \text{BnkAddr}; \]

\[ L = 0; \]

\[ \text{MD} = \text{savBnk}, \text{TASK}; \]

\[ \text{stkp} = L; \]

\[ \text{SWMODE}; \]

: romnextA;
Checksum.mu -- Ram ByteCode to compute PupChecksums

; Last modified HGM June 30, 1980 6:02 PM
; Last modified Johnsson; September 22, 1980 9:12 AM

; Assumes that AltoConsts23.mu and XMesaRAM.mu (which includes Mesab.mu)
; have been included and the following predef has appeared.
; %7, 1777, 1402, PupChecksum;

; PupChecksum:
;    PROCEDURE[initialSum: CARDINAL, address: POINTER, count: CARDINAL]
;     RETURNS[resultSum: CARDINAL]
;    initialSum: must be zero initially (used to restart after interrupts).
;    address: address of block.
;    count: length of block (words) NB: Zero won't work!
;    Returns the ones-complement add-and-cycle checksum over the block.
;    Entry point is Ram address 1402.
;    Timing: 9 cycles/word
;    2484 cycles (= 422 microseconds) per maximum-length Pup

; $MTEMP $R25:
; !l,2,PCMayI,PCNoI;
; !l,2,PCDisI,PCDoI;
; !l,2,PCOkCy,PCZCy;
; !1,2,PCNoCy,PCCy;
; !l,2,PCNoMZ,PCMinZ;
; !l,1,PCDoI1;

PupChecksum:
    L~ stk0;
    temp~ L;
    MAR .. L .. stk1, :PCLp1;

; Top of main loop.
; Each iteration adds the next data word to the partial sum, adds 1 if
; the addition caused a carry, and left-cycles the result one bit.
; Due to ALU function availability, the first addition is actually done
; as (new data word)+(partial sum -1)+1, which causes an erroneous carry
; if (partial sum)=0. Hence we make a special test for this case.

PCLp1: SINK~ NWW, BUS=0;
    Test for interrupts
    stk1~ L, :PCMayI;
        [PCMayI, PCNoI] Update pointer
    PCNoI: T= temp-1, BUS=0, :PCDisI;
        [PCDisI, PCDoI] Get partial sum -1
    PCDisI: L= T- MD+T+1, :PCOKCy;
        [PCOKCy, PCZCy] Add new word +1
    PCOKCy: stk2~ L, :ALUCy;
        Test for carry out, decrement count
    ALUCy: L= stk2-1, PCNoCy;
        [PCNoCy, PCCy] Update count
    PCNoCy: L= T, SH=0, TASK, :PCLast;
        No carry, test count=0
    PCCy: MTEMP= L, L= T+ 0+T+1, SH=0, TASK;
        Do end-around carry, test count=0
    PCLast: temp~ L MLSH 1, :PCLoop;
        [PCLoop, PCDone] Left cycle 1

; Here if partial sum was zero -- suppress test of bogus carry caused by
; MD+(temp-1)+T+1 computation.

PCZCy: L= stk2-1, :PCLp2;

; Here when done
PCDOne: L= temp-1;
        Test for minus zero (ones-complement)
    L= ONE, SH=0;
    Define stack to contain one thing
    PCDoI1: stkp~ L, L= 0, :PCNoMZ;
        [PCNoMZ, PCMinZ]
    PCNoMZ: L= temp, TASK, :PCGoEm;
        Minus zero, change to plus zero
    PCGoEm: stk0~ L, :Emulator;
        Put result on stack

; Here when possible interrupt pending.
; Note that if the interrupt does not take, we read MD one cycle too late.
; This works only on Alto-II.
PCMayI: SINK~ wdc, BUS=0, :PCNoI;
        Let it take only if wdc=0

; Here when interrupt definitely pending.
; Assume that the JRAM was the A-byte, so back up mpc and set ib to zero
; to force the Interpreter to re-fetch the current word and also test again
; for the interrupt we know is pending.
PCDoI: L= mpc-1;
PCDoI1: mpc L, L= 0, TASK;
ib= l;
L= stkp+1, :PCDn1;

Emulator:
SWMODE:
:romnextA:

[PCDOI1] Back up mpc, squash BUS=0
ib= 0
Push Ram address back onto stack

Switch to Rom1
Mesa emulator entry point
Microcode source for Alto running
Mesa6 and using microcode floating point.
adapted from bcpl float microcode (Sproull, Maleson)
Copywrite Xerox Corporation 1980
Last modified by Stewart September 23, 1980 3:24 PM
Last modified by Johnsson September 24, 1980 8:25 AM

Entry points for FP
!17,20,FAdd,FSub,FMul,FDiv,FComp,FFix,FFloat,FFixI,FFixC,FSticky,FRem,FRound,FRoundI,FRoundC,

Registers used internally to float microcode
these should all be available during execution of a mesa byte code

<table>
<thead>
<tr>
<th>R registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>$mSAD $R1; mx</td>
</tr>
<tr>
<td>$N2 $R2; saveret</td>
</tr>
<tr>
<td>$M2 $R3; newfield</td>
</tr>
<tr>
<td>$M1 $R5; count</td>
</tr>
<tr>
<td>$Arg1 $R7; taskhole</td>
</tr>
<tr>
<td>$W1 $R35; temp</td>
</tr>
<tr>
<td>$ArgO $R36; temp2</td>
</tr>
<tr>
<td>$ShiftCount $R36; entry --used only in add/sub</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Mode $R41; mask --used only in add/sub</td>
</tr>
<tr>
<td>$S1 $R43; unused2 --sign</td>
</tr>
<tr>
<td>$Mxreg $R44; alpha</td>
</tr>
<tr>
<td>$Arg2 $R50; unused3</td>
</tr>
<tr>
<td>$E1 $R55; ATPreg --exponent</td>
</tr>
<tr>
<td>$E2 $R57; XTPreg</td>
</tr>
<tr>
<td>$Sticky $R72; Sticky flag</td>
</tr>
</tbody>
</table>

The sticky bit for inexact result is implemented as follows: The sign bit indicates whether trap on inexact result is enabled, the LSB is the actual sticky bit.

Microcode subroutines are defined and called from Mesa programs as shown in the following example:

Misc: L=T, TASK;
  OTPreg=L;
  L=stkp, TASK;
  savestkp=L;
  store stack pointer in savestkp
  T=177757; T=-21B
  L=OTPreg+T+1; L IN [0..11B] if FP instr.
  L=15-T, SHC0;
  T=OTPreg+T+1; L IN [0..11B] if FP instr.

MiscOK1: ArgO=L; MiscOK2;

MiscOK2: SINK=Arg0, BUS, TASK;
  NOP; :FAdd; !17,20,xxx;

MiscSmall: NOP; MiscBig; !1,1, MiscBig;
MiscBig: NOP, TASK, FPTrap;

Microcode subroutines are defined and called from Mesa programs as shown in the following example:

MiscAlpha: CARDINAL = 20B; -- Alpha Byte of instruction --
CalluRoutine: PROCEDURE[x, y: REAL] RETURNS [z: REAL] =
  MACHINE CODE BEG
Mopcodes.zMISC. MiscAlpha;
END;

[]]=CalluRoutine[a, b]; -- the call --

!!1.2.LowNZerol.LowZerol; define before use!
!!1.1.FCRet;

; returns control to emulator in Rom1 (or Ram1 on 3K machines).
; TASK in instruction calling retCom
;
retCom: stkp=L;  
SWMODE; Switch to Rom1  
L+T=0;:romnext; Mesa emulator entry point
;
; pushes Arg0, Arg1 and returns control to emulator in Rom1  
; called with stkp correct  
; Remember, in Mesa, the M.S. word (Arg0) is on top of stack
!
!!17.20.LTpush0.LTpush1.LTpush2.LTpush3.LTpush4.LTpush5.LTpush6.LTpush7.LTpush8,. . . ; FPdpush:  
SINKstkp.BUS;  
T=Arg1.;LTpush0;
  
LTpush0: stk1=L.L+T.TASK;  
stk0=L.;LTpushCom;
LTpush1: stk2=L.L+T.TASK;  
stk1=L.;LTpushCom;
LTpush2: stk3=L.L+T.TASK;  
stk2=L.;LTpushCom;
LTpush3: stk4=L.L+T.TASK;  
stk3=L.;LTpushCom;
LTpush4: stk5=L.L+T.TASK;  
stk4=L.;LTpushCom;
LTpush5: stk6=L.L+T.TASK;  
stk5=L.;LTpushCom;
LTpush6: stk7=L.L+T.TASK;  
stk6=L.;LTpushCom;
LTpush7: NOP.TASK.;RamStkErr;  
LTpush8: NOP.TASK.;RamStkErr;  
LTpushCom: T=2;  
L=stkp+T.TASK.:retCom;

; pushes Arg0 and returns control to emulator in Rom1  
; called with stkp correct
!
!!17.20.LUpush0.LUpush1.LUpush2.LUpush3.LUpush4.LUpush5.LUpush6.LUpush7.LUpush8,. . . ;

; cannot be a TASK in instruction coming here

ShortRet:  
T=stkp+1.BUS;  
L=Arg0.;LUpush0;
  
LUpush0: stk0=L.L+T.TASK.:retCom;
LUpush1: stk1=L.L+T.TASK.:retCom;
LUpush2: stk2=L.L+T.TASK.:retCom;
LUpush3: stk3=L.L+T.TASK.:retCom;
LUpush4: stk4=L.L+T.TASK.:retCom;
LUpush5: stk5=L.L+T.TASK.:retCom;
LUpush6: stk6=L.L+T.TASK.:retCom;
LUpush7: stk7=L.L+T.TASK.:retCom;
LUpush8: NOP.TASK.;RamStkErr;

; Code to trap through SD in case of error  
; Control gets to error handler with  
; whatever was on stack at start of faulted instr.  
; OTPreg is alpha byte of faulted instruction

; The next line must change to track changes in Mesa emulator
The way this works is to branch to OutOfRange in the bounds check code with the SD index - sBoundsCheck in T
OTPreg is already loaded with the instruction

; TASK in instruction calling this one
FPTrap: NOP;
T=100;
L=17 OR T,TASK; T=137 (our SD Index) (minus sBoundsFault)
ramKFC: taskhole=L;
L=savestkp,TASK;
stkp=L;
T=taskhole,SWMODE;
L=0..romOOR; OutOfRange (cause KFC)

; TASK in instruction coming here
RamStkErr: NOP;
T=sBoundsFaultm1+1;
L=2-T,TASK,:ramKFC; KFCB, 2 (minus sBoundsFault)

;---------------------------------------------------------------
; multiply subroutine
;---------------------------------------------------------------
!7,10.MulRet,MulRet1,MulRet2,MulRet3;
!7,1,ramMulA; shake IR= dispatch
!1,2,DOMUL,NOMUL;
!1,2,MPYL,MPYA;
!1,2,NADDIER,ADDIER;
!1,2,NOSPILL,SPILL;
!1,2,NOADDX,ADDX;
!1,2,NOSPILLX,SPILLX;

ramMul: L=Arg2-1, BUS=0;
ramMulA: mSAD=L,L=0,:DOMUL; !1,2,DOMUL,NOMUL;

DOMUL: T=10+1;
Mxreg"l;

MPYL: L=Arg1,BUSODD;
T=Arg0,:NOADDIER;

NOADDIER: Arg1=MRSH 1,L=T,T=0,:NOSPILL; !1,2,NOSPILL,SPILL;

ADDIER: L=Arg1,ALUCY,:NOADDIER;

ADDX: L=Arg1,BUSODD;
T=Arg0,:NOADDX;

NOADDX: Arg1=MRSH 1,L,T,T=0,:NOSPILL;

ADDX: L=Arg1,ALUCY,:NOADDX;

SPILL: T=ONE;

NOSPILL:Arg0=L MRSH 1;

NOMUL: T=Arg0,:NOSPILLX; !1,2,NOSPILLX,SPILLX;

NOSPILLX:Arg0=L MRSH 1;

MPYA: IDISP,TASK;

NO TASK in the instruction getting here

DPop: T=2;

!7,1,DPopA;

DPopA: L=stkp-T,BUS,TASK;

LRpop0: NOP,TASK,:RamStkErr; stkp=0!
LRpop1: NOP, TASK,: RamStkErr; stk=1!
LRpop2: L=stk1;
LRpop3: L=stk2;
LRpop4: L=stk3;
LRpop5: L=stk4;
LRpop6: L=stk5;
LRpop7: L=stk6;
LRpop8: L=stk7;
LRpopCom: SH<0,: DPopRet;

UnPack: load up arguments into registers

Purpose is to unpack the two float numbers on mesa stack and save them in S,E,M,N 1 and 2.
We unpack the b argument first, so Fix can jump into middle and just unpack a.
This code uses a threading idea. Once IR has been set up with sr0 or sr1, the IDISP return can be used with even 1 instruction subroutines.

LoadArgs: SubRet"L; save return address
IR=sr0,: DPop;
DPopRet: Arg0=L,T,IDISP,: UPPos; Arg1=L,T,IDISP,: UPNeg;

UPPos: Arg1=L=0,TASK,: PVbSign; Arg1=L=0-1,TASK,: PVbSign; Store S=1
PVbSign: S2=L,: UPC1; Unpack Common 1

UPC1: T=377,IDISP;
L=Arg1 AND T,: UPR1b; Low 8 bits of mantissa
!1,2, UPR1b,: UPR1;
UPR1b: N2=L LCY 8,: UPC2; Store in left half word

UPC2: L=Arg1 AND NOT T; Middle 8 bits of mantissa
Arg1=L LCY 8, IDISP; Store in right half word
; Now here we are using 377 instead of 177, but it doesn't matter because we will or in a one bit there anyway, later.
L=Arg0 AND T, TASK,: UPR2b; High 7 bits of mantissa
!1,2, UPR2b,: UPR2;

UPR2b: M2=L LCY 8; Store in left half word
T=100000; hidden bit
T=M2 OR T.IDISP,: UPC4; high 7 bits of mantissa

UPC4: L=Arg1 OR T, TASK,: UPR4b; next 8 bits
!1,2, UPR4b,: UPR4;

UPR4b: M2=L,: UPC5; mantissa finished

Here we use 177600 instead of 77600, but the left shift clears it.
The SH=0 test works because the test depends on L from the previous microinstruction plus shifter operation during current microinstruction

UPC5: T=177600; exponent mask
L=Arg0 AND T;
ArgO\rightarrow L, LSH 1, SH=0; \quad \text{exponent left justified now}
L=ArgO, IDISP, :UPBias; \quad !1,2,UPBias, UPNoBias;

UPBias: ArgO\rightarrow L, LCY 8, :UPBias1; \quad \text{exponent right justified now}
; \quad !1,1,UPBias1;

UPBias1: T=377;
L=ArgO\rightarrow T, IDISP; \quad \text{check for exp=377}
T=177, SH=0, :PVR6b; \quad !1,2,PVR6b, PVR6;
PVR6b: L=ArgO\rightarrow T, TASK, :PVBCom; \quad !1,2,PVBCom, PVBNaN;
PVBCom: E=2, L,:PackedVectora;

PVBNaN: NOP,:FFTrap; \quad \text{Instruction after TASK!}

; Test if mantissa was zero, if so then true zero, else denormalized
UPNoBias: T=100000,:PVR5b; \quad !1,2,PVR5b, PVR5;
PVR5b: SINK=N2, BUS=0;
L = M2 XOR T, IDISP, :UPDN; \quad !1,2,UPDN, UPZeroT;

UPDN: NOP, TASK, :PVBNaN; \quad !1,1,PVBNaN;
UPZeroT: NOP, SH=0,:PVR7b; \quad !1,2,PVR7b, PVR7;
PVR7b: M2=L,:PVDNb; \quad !1,2,PVDNb, PVBZero;
PVDN: NOP, TASK,:FFTrap;
PVBZero: L=E2, TASK,:PVBCom; \quad \text{don't diddle sign}
; \quad ----------------------------------------------------------
; \quad \text{now unpack TOS}
; \quad !1,2,LRET, PVBNaN;
!1,2, PVDNb, PVBZero;

; Cannot be TASK in instruction coming here
PackedVectora: IR=sr1,:DPop;
DPopRet1: ArgO\rightarrow L, L+T, IDISP, :UPPos; \quad !1,2,UPPos, UPNeg;

PVSign: S1=L,:UPC1;

UPR1: M1=L, LCY 8,:UPC2; \quad \text{Store in left half word}

UPR2: M1=L, LCY 8; \quad \text{Store in left half word}
T=100000; \quad \text{hidden bit}
T=MI OR T, IDISP, :UPC4; \quad \text{high 7 bits of mantissa}

UPR4: M1=L,:UPC5; \quad \text{mantissa finished}
PVR6: L=ArgO\rightarrow T, TASK,:LRET; \quad !1,2, LRET, PVBNaN;
PVR5: SINK=N1, BUS=0;
L = M1 XOR T, IDISP,:UPDN; \quad !1,2,UPDN, UPZeroT;
PVR7: M1=L,:PVDN; \quad !1,2,PVDN, PVBZero;
PVDNb: NOP, TASK,:FFTrap;
PVBZero: L=E1, TASK,:LRET; \quad \text{true zero}

PVBNaN: NOP,:FFTrap; \quad \text{Instruction after TASK!}

LRET: E1=L;
SINK=SubRet, BUS, TASK; \quad \text{and, the big return}
NOP,:LoadRet; [LoadRet, LoadRet1, LoadRet2, LoadRet3]

; repackage into Arg0, Arg1, push and return
; \quad ----------------------------------------------------------
!1,2, FSTNZero,FSTZero;
!1,2,Round, NoRound;
!1,2, PMRound, MidRound;
!1,1, MRnd1;
!1,2, MidRPlus1, MidRPlus0;
!1,2, RPlus0, RPlus1;
!1,2, FSTNoR2, FSTR2;
!1,2, FSTNoSh, FSTSh;
!1,2, IRRNoTrap, IRRTrap;
!1,1, NoRound1;
!1,2, NoExpUF, ExpUF;
!1,2, NoExpOV, ExpOV;
!1,2, FSTNeg, FSTPos;
RePack: SINK=M1,BUS=0; check for zero result
; do a form of rounding, by checking value in low N1 bits
T=377.,FSTNZero: !1,2,FSTNZero,FSTZero;
FSTZero: L=0.,LowZero1; LowZero1 will set sign
FSTNZero: L=T=N1.T;

; after the subtract in the next instruction, SH=0 if the result is halfway
; between representable numbers. SH=0 if the result is larger in magnitude
; than halfway

L=200-T,SH=0;
T=377.,SH=0,:Round!:1,2,Round,NoRound;
Round: NOP,SH<0:PMRound!:1,2,PMRound,MidRound;
MidRound: T=N1,:MRnd1;!1,1,MRnd1;
; but to be safe !1,1,MRnd1;
MRnd1: L=400 AND T;
NOP,SH=0;
T=377.:MidRPlus1!:1,2,MidRPlus1,MidRPlus0;
MidRPlus0: L=M1 AND T,TASK,:NoRound1;
MidRPlus1: L=N1+T+1,:RoundPlus;
PMRound: T=377.,RPlus0!:1,2,RPlus0,RPlus1;
RPlus0: L=M1 AND T,TASK,:NoRound1;
FSTR: T=400;
L=N1+T;
RoundPlus: M1=L,ALUCY;
L=M1+1,:FSTNoR2;
FSTNoR2: M1=L,ALUCY,TASK,:FSTNoR2;
FSTSh: L=T=M1: low order
M1=L RSH 1;
L=N1,TASK;
N1=L MRSH 1;
T=E1+1,TASK;
E1=;
FSTNoSh: T=0+1; sticky bit for inexact result
L=Sticky OR T;
Sticky=L,SH<0;
T=377.:IRNoTrap!:1,2,IRNoTrap,IRTralp;
IRTrap: NOP,TASK,:FPTrap;
IRNoTrap: L=M1 AND T,TASK,:NoRound1;
NoRound: L=M1 AND T,TASK,:NoRound1!:1,1,NoRound1;
NoRound1: Arg0=L; low 8 bits, r.j. (middle mantissa)
T = 177400;
L=M1 AND T;
M1=L LSH 1: high 7 bits, l.j. with h.b. shifted out
T=N1.T: high 8 bits, l.j.
L=Arg0 OR T,TASK;
Arg1=L LCY 8: ready for FPdpush

T=176;
L=E1+T;
T=E1+T+1,SH<0,TASK;
Arg0=L LCY 8,:NoExpUF!:1,2,NoExpUF,ExpUF;
; At this point, the exponent is in left half of Arg0, but not tested for OV yet
NoExpUF: T=E1;
L=177-T;
L=M1,SH<0; Swab M1, to r.j. mantissa
M1=L LCY 8,:NoExpOV!:1,2,NoExpOV,ExpOV;
ExpUF: NOP,TASK,:FPTrap;
ExpOV: NOP,TASK,:FPTrap;

; If we get here, the exp in l.h. of Arg0 is in range
NoExpOV: T=Arg0; r.h. zero, so don't mask
L=M1 OR T,TASK;
Arg0=L RSH 1,:FSTSh: ready for FPdpush
; at this point, M1,.N1 has everything but sign

; Set sign bit
FSTSh: SINK=S1,BUS=0;
L=T=Arg0,:FSTNeg!:1,2,FSTNeg,FSTPos;
Float: L=100000 OR T;
FSTPos: Arg0 + L::FPpush;

; Float: a long integer is on the stack
---------------------------------------------------------------------

!1,2,FltPos,FltNeg;
!1,2,FltLNZ,FltLZ;
!1,2,FltHNZ,FltHZ;
!1,2,FltCont,FltAllZ;
!1,2,FltMore,FltNorm;

FFloat: IR~sr2,:DPop;  
DPopRet2: M1=L,L=T,:FltPos; !1,2,FltPos,FltNeg;
FltPos: N1=L,L=0,TASK,:FltSign;
FltNeg: L=0-T; negate the double word, store S1=-1  
      N1=L,SH=0;  
      T+M1,:FltLNZ; !1,2,FltLNZ,FltLZ;

FltLNZ: L=0-T-1,:FltStore; complement  
FltLZ: L=0-T,:FltStore; negate if low word 0  
FltStore: M1=L,L=0-1,TASK,:FltSign; set sign=-1  
FltSign: S1=L;

; now, double word LShift until normalized  
L=37,TASK;  
E1=L: 31 decimal if already normalized

; we will always shift at least once, so max exponent will be 30  
SINK=M1,BUS=0,TASK;  
NOP,:FltHNZ; !1,2,FltHNZ,FltHZ;

FltHZ: T=M1,BUS=0;  
      L=17,:FltCont; !1,2,FltCont,FltAllZ;
FltAllZ: L=0,:LowZero1; S1 known to be 0  
       FltCont: E1=L,L=T; 16 shifts like wildfire  
       M1=L,L=0,TASK;  
       N1=L,:FltHNZ;

FltHNZ: L=M1;  
      T=M1,SH<0;  
      M1=LM1SHL1,L=T,:FltMore; !1,2,FltMore,FltNorm;
FltMore: N1=L LSH 1;  
       L=E1-1,TASK;  
       E1=L,:FltHNZ;

; We just shifted out the leading one, so put it back.  
FltNorm: L=M1;  
       T=ONE,TASK;  
       M1=LM1MSH 1,:RePack;

; Remainder: not implemented

FRem: NOP,TASK,:FPTrap;

; Round to Integer
---------------------------------------------------------------------

!1,2,FRIEPlus,FRIENeg;
!1,2,FRIEOK,FRIEOv;
!1,2,FRIStik,FRIStik;
!1,2,FRIShift,FRIDone;
!1,2,FRINE,FRIPlus1A;
!1,2,FRINPlus1,FRIPlus1;
!1,2,FRIPOv,FRIPOv;
!1,2,FixINeg,FixIPos;

FRoundI: L=7,TASK,:SavePVA; (see Fix)  
LoadRet7: L=T=E1+1;  
       L=20-T-1,SH<0; E1 must be positive  
       E1=L,SH<0,:FRIEPlus; !1,2,FRIEPlus,FRIENeg;
FRIEPlus:  L+T=M1,TASK,:FRIEOK;  E1 must be < 16 decimal
FRIEOK:  M1+L RSH 1;
         L+T=N1,TASK,BUSODD;
         M1=L+MRSH1,:FRINStik;  !1,2,FRINStik,FRStik;
FRINStik:  L=E1+1,BUS=0,TASK;
          E1=L,:FRIShift;  !1,2,FRIShift,FRIDone;
FRIShift:  T=O+1;
           L=N1 OR T,TASK;
           N1=L,:FRINStik;

; IF N1=100000B then let Mesa handle it.  IF N1>100000B then add 1 to M1
FRIDone:  T=N1,BUS=0;
         L=100000-T,:FRINE:  !1,2,FRINE,FRIPlus1A;
FRINE:  NOP,SH<0;
         L=M1+1,SH=0,:FRINPlus1;  !1,2,FRINPlus1,FRIPlus1;

; 100000 bit may have been on.  SH=0 will branch if so
FRINPlus1:  SINK=S1,BUS=0,:FRIPNOv;  !1,1,FRIPNOv,FRIPOv;

; complete the +1.  The pending SH=0 branch will not go
FRIPlus1A:  M1+L,SH<0,:FRIPPlus1A;  !1,1,FRIPPlus1A;
FRIPPlus1A:  SINK=S1,BUS=0,:FRIPNOv;  !1,2,FRIPNOv,FRIPOv;
FRIPNOv:  L+T=M1,:FixINeg;  !1,2,FixINeg,FixIPos;

FRINeg:  L=0,TASK,:FCRet;  store 0 and return
          !1,1,FCRet;

; Overflow here is a little funny:
; FixI of 100000B traps, but is really OK, this shouldn't
; happen very often, so the Mesa code can handle it
FRIPOv:  NOP,TASK,:FTrap;  FixExponentOverflow (trap)
FRPOv:  NOP,:MiscBig;

; complete the +1.  The pending SH=0 branch will not go
FRIPlus1:  L+T=M1,TASK,:SavePVA;
LoadRel11:  L=T=E1+1;
          L=177740+T,SH<0;
          L=LastT+1,ALUCY,:FRGem1;  !1,2,FRGem1,FRlsm1;
FRGem1:  L=37-T-1,SH=0,:FRls31;  !1,2,FRls31,FRge31;
          !1,2,FRle29,FR30;
          !1,2,FRNext,FRDone;
          !1,2,FRLoop,FRStik;
          !1,4,FRD300,FRD301,FRD302,FRD303;
          !1,2,FRDNoAdd,FRDAdd;
          !1,2,FRDNoCy,FRDCY;
          !1,2,FRDNOv,FRDOv;
          !1,2,FixShift,FixSgn;  This occurs here first
          !1,1,FixENeg1;  This occurs here first
FRRound:  L=11,TASK,:SavePVA;
LoadRel11:  L=T=E1+1;
          L=177740+T,SH<0;
          L=LastL+1,ALUCY,:FRGem1;  !1,2,FRGem1,FRlsm1;
FRGem1:  L=37-T-1,SH=0,:FRls31;  !1,2,FRls31,FRge31;
          !1,2,FRle29,FR30;
FRle29:  L=S2-1,BUS=0,TASK,:FRLoop1;
FRLoop:  L=S2-1,BUS=0,TASK;
FRLoop1:  S2=L,:FRNext;  !1,2,FRNext,FRDone;
FRNext:  L=T=M1;
          M1=L RSH 1;
          L=N1,TASK,BUSODD;
          M1=L+MRSH1,:FRLoop;  !1,2,FRLoop,FRStik;
FRStik:  T=O+1;
           L=N1 OR T,TASK;
           N1=L,:FRLoop;
FRDone:  T=3;
           L=N1 AND T;
           M2=L;
           L=T=M1;
           M1=L RSH 1;
           L=N1,TASK;
           N1=L+MRSH1,:FRLastSh;
FR30:  L=0,TASK;
    M2=L;
FRLastSh:   L=T+M1;
    M1=L RSH 1;
    L=N1,TASK;
    N1=L MRSH 1;
    SINK=M2,BUS;
    L=N1+1,:FRD300; !3,4,FRD300,FRD301,FRD302,FRD303;
FRD300:  NOP,:FixSgn;
FRD301:  NOP,:FixSgn;
FRD302:  SINK=N1,BUSODD;
    L=N1+1,:FRDNoAdd;    !1,2,FRDNoAdd,FRDAdd;
FRDNoAdd: NOP,:FixSgn;
FRDAdd:   N1=L,ALUCY,:FRD303a;
FRD303a:  L=M1+1,:FRDNoCy;    !1,2,FRDNoCy,FRDCy;
FRDNoCy: NOP,:FixSgn;
FRDCy:   M1=L,SH<0,TASK;
    NOP,:FRDNoV;    !1,2,FRDNoV,FRDOv;
FRDNoV:  NOP,:FixSgn;
FRDOv:   NOP,TASK,:FPTrap;
FRlm1:   L=0,:FixENeg1;    ; !1,1,FixENeg1;
FRge31:  NOP,:MiscBig;    ; !1,1,MiscBig:

;---------------------------------------------------------------
;     Round to Cardinal
;---------------------------------------------------------------
;   !1,2,FRCVNeg,FRCVOK;
;   !1,2,FRCShift,FRCENeg;
;   !1,2,FRECOK,FRECov;
;   !1,2,FRCMore,FRCDone;
;   !1,2,FRCNStik,FRCStik;
;   !1,2,FRC+,FRCPlus1A;
;   !1,2,FRCNPlus1,FRCPlus1;
;   !1,2,FRCPOv,FRCPOv;
FRoundC:  L=10,TASK,:SavePVA;    (see Fix)
LoadRet10: SINK=S1,BUS=0; Value must be positive.
    L=T+E1+1,:FRCVNeg;    !1,2,FRCVNeg,FRCVOK;
FRCVOK:   L=20-T,SH<0; E1 must be positive
    E1+L,SH<0,:FRCShift;    !1,2,FRCEOK,FRCEOv;
    E1 must be < 16 decimal
FRCShift: L=E1-1,:FRCEOK;    !1,2,FRECOK,FRECov;
FRECOK:   E1=L,SH<0;
    L=T+M1,:FRCMore;    !1,2,FRCMore,FRCDone;
FRCMore:  M1=L RSH 1;
    L=N1,TASK,BUSODD;
    N1=L MRSH1,:FRCStik;    !1,2,FRCNStik,FRCStik;
FRCNStik: L=E1-1,:FRECOK;
FRCStik:   T=0+1;
    L=N1 OR T,TASK;
    N1=L,:FRCNStik;
FRCDone:  T=N1,BUS=0;
    L=100000-T,:FRCN;
    !1,2,FRCN,FRCPlus1A;
FRCN:    NOP,SH<0;
    L=M1+1,SH=0,:FRCPNOv;    !1,2,FRCNPlus1,FRCPlus1;
    100000 bit might have been on, SH=0 will branch if so
FRCPlus1: L=M1,:FRCPNOv;    !1,2,FRCN,FRCPlus1;
FRCNPlus1: M1=L,ALUCY,:FRCPlus1A;    !1,1,FRCPlus1A;
FRCPlus1A: L=M1,:FRCPNOv;    !1,2,FRCN,FRCPlus1;
FRCPOv:   Arg0+L,:ShortRet;
FRCNeg:   L=0,TASK,:FCretr;
    store 0 and return ; !1,1,FCRet;
FRCVNeg:  NOP,TASK,:FPTrap;
FRCPOv:   NOP,:MiscBig;    ; !1,1,MiscBig;
Fix

; Fix

!1,2,FixEPlus,FixENeg;
!1,2,FixEOk,FixEOv;
; !1,2,FixShift,FixSgn; This occurs earlier
!1,2,FixNeg,FixPos;
!1,2,FixLNZ,FixLZ;
; !1,1,FixENeg1; This occurs earlier

FFix: L=3,TASK;
SavePVA: SubRet=L,:PackedVectora; middle of unpack routine!
LoadRet3: L=T=E1;
    L=37-T-1,SH<0; E1 must be positive
    E1=0,SH<0,:FixEPlus; !1,2,FixEPlus,FixENeg;

FixEPlus: L=4=M1,:FixEOk; E1 must be < 31 decimal
FixShift: L=4=M1,:FixEOk; !1,2,FixEOk,FixEOv;
FixEOk: M1=L,RSH 1;
    L=N1,TASK;
    N1=L,MRSH 1;
    L=E1-1,BUS=0;
    E1=L,:FixShift; !1,2,FixShift,FixSgn;

FixSgn: SINK=S1,BUS=0;
    L=N1,:FixNeg; !1,2,FixNeg,FixPos;
FixPos: Arg1=L;
    L=M1,TASK,:FixStore;
FixNeg: L=0-T;
    Arg1=L,SH=0; negate the double word
    T=M1,:FixLNZ; !1,2,FixLNZ,FixLZ;

FixLNZ: L=0-T-1,TASK,:FixStore; complement
FixLZ: L=0-T,TASK,:FixStore; negate if low word 0
FixStore: Arg0=L,:FPpush;

FixENeg: L=0; !1,1,FixENeg1;
FixENeg1: Argi=L,TASK,:FixStore; store 0 and return
FixEOv: NOP,TASK,:FPTrap; FixExponentOverflow (trap)

......

FixC Fix to CARDINAL
......

!1,2,FixCVNeg,FixCVok;
!1,2,FixCShift,FixCENeg;
!1,2,FixCEOk,FixCEOv;
!1,2,FixCMore,FixCDone;

FFixC: L=4,TASK,:SavePVA; (see FFix)
LoadRet4: SINK=S1,BUS=0; Value must be positive.
    L=T=E1,:FixCVNeg; !1,2,FixCVNeg,FixCVok;
FixCVOK: L=17-T,SH<0; E1 must be positive
    E1=L,SH<0,:FixCShift; !1,2,FixCShift,FixCENeg;

; E1 must be < 16 decimal
FixCShift: L=E1-1,:FixCEOk; !1,2,FixCEOk,FixCEOv;
FixCEOk: E1=L,SH<0;
    L=M1,TASK,:FixCMore; !1,2,FixCMore,FixCDone;
FixCMore: M1=L,RSH 1,:FixCShift;

; FixCDone called from FixI as well
FixCDone: Arg0=L,:ShortRet;

FixCENeg: L=0,TASK,:FCRet; store 0 and return
    ; !1,1,FCRet;
FixCEOv: NOP,TASK,:FPTrap; FixExponentOverflow (trap)
FixCVNeg: NOP,TASK,:FPTrap; FixCValueNegative (trap)

......

FixI Fix to INTEGER
......

!1,2,FixIEPlus,FixIENeg;
Float.mu

1,2,FixIEOK,FixIEOv;
1,2,FixIShift,FixIDone;
1,2,FixINeg,FixIPos;

FFixi: L=5,TASK,:SavePVA; (see FFix)
LoadRet5: L=T=E1;
   L=7-T-1,SHC0; E1 must be positive
   E1.L,SHC0,:FixIEPlus; !1,2,FixIEPlus,FixIENeg;
FixIEPlus: L=M1,TASK,:FixIEOK; E1 must be < 15 decimal
FixIShift: L=M1,TASK,:FixIEOK; !1,2,FixIEOK,FixIEOv;
FixIEOK: M1=L RSH 1;
   L=E1-1,BUS=0,TASK;
   E1:L,:FixIShift; !1,2,FixIShift,FixIDone;
FixIDone: SINK=S1,BUS=0;
   L=T+M1,:FixINeg;

FixIPos: NOP,TASK,:FixCDone;
FixIENeg: L=0-T,TASK,:FixCDone;

Overflow here is a little funny;
; FixI of 10000000 traps, but is really OK, this shouldn't
; happen very often, so the Mesa code can handle it
FixIEOv: NOP,TASK,:FPTrap; FixIExponentOverflow (trap)

Mul: floating point multiply

MulNZero, MulZero;
MulINZero1, MulZero1;
MulNoCy, FMNoCy, FMNoCy1, FMNoCy2;
MulCy, FMCy, FMCy1, FMCy2;
MulNLL, FMNoLL;
MulNorm, MulINorm;
MulZero2, MulINZero2;

FMul: L=0,TASK,:LoadArgs;
LoadRet: T=E1; add exponents, like in any multiply
   L=E2+T+1,TASK;
   E1:L;
   T=S1; and xor signs
   L=S2 XOR T,TASK;
   S1=L;

; Putting the argument with zeros on the right wins because that loop is
; only four cycles per bit, (see ramMul code)
   L=M1,TASK,BUS=0; first multiply: high*low
   Arg2=L,:MulINZero; !1,2,MulINZero,MulZero;
MulINZero: L=N2;
   Arg1=L,L=0,TASK,:MulZeroa;
MulZero: L=0,:LowZero1; return 0
MulZeroa: ArgO=L;
   IR=srO,:ramMul;
MulRet: L=ArgO,TASK;
   Here we will start using S2, and E2 to hold some temporary stuff
   S2=L; high result
   L=Arg1,TASK;
   E2=L; low result
   L=M2,TASK,BUS=0; second multiply: other high*other low
   Arg2=L,:MulINZero1; !1,2,MulINZero1,MulZero1;
MulINZero1: L=N1; second multiply: other high*other low
   Arg1=L,L=0,TASK,:MulZeroa1;
MulZero1: L=0,:LowZero1;
MulZeroa1: ArgO=L;
   IR=sr1,:ramMul;
MulRet1: T=Arg1;
L=E2+T; add results, set carry if overflow
E2=L,ALUCY;
T=Arg0,:FMMoCy1,!,2,FMMoCy,FMCy;
FMNoCy: L=S2+T,:FMCyS;
FMCy: L=S2+T+1,:FMCyS;
FMCyS: S2=L,ALUCY;
L=0,:MulNoCy1,!:1,2,MulNoCy1,MulCry;
MulCry: L=ONE, TASK;

; Use SubRet to hold carry bit
MulCry:
   SubRet=L;
   L=N1; third multiply: low*low
   Arg1=L,L=0, TASK;
   Arg0=L;
   L=N2,TASK;
   Arg2=L;
   IR=sr2,:ramMul;
MulRet2: T=Arg0; Arg1=0 (low result)
   L=E2+T;
   E2=L,ALUCY;
   L=S2+1,:FMNoCy1; !1,2,FMNoCy1,FMCy1;
FMCy1: L=S2,ALUCY;
FMNoCy1: SubRet+1,:FMNoCy1; !1,2,FMNoCy2,FMCy2;
FMCy2: SubRet+L;
FMNoCy2:
   Arg0=L;

; last multiply: high*high (plus stuff left in Arg0)
   L=M1, TASK;
   Arg1=L;
   L=M2, TASK;
   Arg2=L;
   IR=sr3,:ramMul;
MulRet3: SINK=E2,BUS=0;
   L=T-Arg1,:FMLL; !1,2,FMLL,FMMoLL;
FMLL:
   L=ONE OR T, TASK,:FMNoLL; sticky bit if third word#0
FMNoLL: N1=L;
   T=Arg0;
   L=SubRet+T; add in possible carry
   M1=L,SH<0; now, check normalization
   T=N1,:MulNorm; 7 instructions since last TASK
      : !1,2,MulNorm,MulNoNorm;
   MulNorm:
      M1=L MLSH 1; 8
      L=N1,SH=0;
      N1=L LSH 1,:MulNzer02; 10 !1,2,MulNzer02,MulZero2;
   MulNzer02:
      L=E1-1, Task; decrement exponent to account for shift
   MulDone: E1=L,:RePack;
   MulNoNorm:
      L=E1, TASK,:MulDone;
   MulZero2: L=0,:LowZero1;

; FDiv floating point divide
;---------------------------------------------------------------
   !1,2,DivOK,DivErr;
   !1,2,DivOK1,DivZero;
   !1,2,DAddNoCy,DAddCy;
   !1,2,DSubNoCy,DSubCy;
   !1,2,DivRes0,DivRes1;
   !1,2,DivMore,DivDone;
   !1,2,DivAdd,DivSub;
   !1,2,DivLoop,DivNorm;
   !1,2,DivLast0,DivLast1;
   !1,2,DivStik,DivNoStik;
   !1,2,DivLNOCy,DivLocy;
   !1,2,DivS1,DivLC1;
FDiv: L=ONE, TASK,:LoadArgs;
LoadRet1: T=E2+1;
   T=-7+T+1;
   L=E1-T, TASK;
   E1=L;
   T=S2;
   L=S1 XOR T, TASK;
S1+L;

; pre right-shift
L-T=M2,BUS=0;
M2=L RSH 1,:DivOK; 1,2,DivOK,DivErr;
DivErr: NOP,TASK,:FPtrap;
DivOK: L=N2,TASK;
N2=L MRSH 1;
L=M1,BUS=0;
Arg0=L RSH 1,:DivOK1; 1,2,DivOK1,DivZero;
DivZero: L=0,:LowZero1;
DivOK1: L=N1;
Arg1=L MRSH 1,L=0;
N1=L; will be msb result
L=30+1,TASK; set E2 to NumLoops-1
E2=L;
T=N2,:DivSub;

DivAdd: L=Arg1+T;
Arg1=L ALUCY;
T=M2,:AddNoCy; 1,2,AddNoCy,AddCy;
AddNoCy: L=Arg0+T,:DOpCom;
AddCy: L=Arg0+T+1,:DOpCom;
DivSub: L=Arg1-T;
Arg1=L ALUCY;
T=M2,:SubNoCy; 1,2,SubNoCy,SubCy;
SubNoCy: L=Arg0-T,:DOpCom;
SubCy: L=Arg0-T,:DOpCom;

; If the operation carries, then the next operation
; should be a subtract and the result bit should be a one.
; If the operation does not carry, then the next operation
; should be an add and the result bit should be a zero.
DOpCom: Arg0=L ALUCY;
L=N1,:DivRes0; 1,2,DivRes0,DivRes1;
DivRes1: L=N1+T+1;
N1=L,:ResCom;
DivRes0: N1=L LSH 1,:ResCom;

ResCom: L=M1,TASK;
M1=L MSLH 1;
L=E2-1,TASK,BUS=0;
E2=L,:DivMore; 1,2,DivMore,DivDone;

; Now double the a operand the result sign bit will always be the same
DivMore: L=Arg1;
Arg1=L LSH 1;
L=Arg0,TASK;
Arg0=L MSH 1;

; do double add or subtract according to previous bit of result
SINK=N1,BUSODD;
T=N2,:DivAdd; 1,2,DivAdd,DivSub;

DivDone: L=T=N1;
S2=L,:DivDone2;
DivDone1: L=T=N1; normalize result
DivDone2: N1=L LSH 1;
M1=L MSLH 1;
M2=L MSLH 1,:DivLoop; 1,2,DivLoop,DivNorm;
DivLoop: L=E1-1,TASK;
E1=L,:DivDone1;

; If the last bit of result was a 1 AND Arg0=0 Then EXACT
; If the last bit of result was a 0 AND Arg0=0 Then EXACT
DivNorm: SINK=S2,BUSODD;
T=Arg1,:DivLast0; 1,2,DivLast0,DivLast1;
DivLast1: L=Arg0 OR T;
NOP,SH=0,:DivLC1;
DivLC1: L=N1+1,TASK,:DivStik; 1,2,DivStik,DivNoStik;
DivLastO: L-N2+T;
    NOP,AUCY;
    T-Arg0.:DivLONoCy: !1,2,DivLONoCy.DivLOCy;
DivLONoCy: L=M2+T,SH=0,:DivLCom;
DivLOCy: L=M2+T+1,SH=0,:DivLCom;
DivLCom: L=N1+1,SH=0,:DivS1: !1,2,DivS1.DivLC1;

DivStik: N1-L,:RePack;
DivNoStik: NOP,:RePack;
DivS1: N1-L,TASK,:DivNoStik; ; !1,1,DivNoStik;

; floating point add and subtract
;---------------------------------------------------
!1,2,Sh,NoShz;
!1,2,Sh1,NoSh;
!1,2,E1lsE2,E1grE2;
!1,2,NoFix,Fix;
!1,2,Sh1NSTik,Sh1Stik;
!1,2,More,Shifted;
!1,2,ExpOK,ExpWrite;
!1,2,NoFix1,Fix1;
!1,2,Sh2NSTik,Sh2Stik;
!1,2,More1,Shifted1;

FAdd: L=0,TASK,:StoreMode;
FSub: L=ALLONES,TASK;
StoreMode: Mode=L;
L=2,TASK,:LoadArgs;

;Preshift arguments until they match
LoadRet2: T=M1: mantissa zero check
        L=M2 AND T; one OR the other = 0
        SINK=LastL,BUS=0;

        T=E1.:Sh;   !1,2,Sh,NoShz;
        L=E2-T; if exponents are the same, no shift either
        SINK=LastL,BUS=0;

        ShiftCount=L.:Sh1;   !1,2,Sh1,NoSh;
        Sh1: TASK,SH=0;
        NOP,:E1lsE2;   !1,2,E1lsE2,E1grE2;
        E1lsE2: L=E2,TASK;
        E1=L; we'll shift until exp matches E2
        T=ShiftCount;
        L=37-T;
        TASK,SH<0;  37 is max number of shifts, if SH ge 0 then fix
        NOP,:NoFix;
        !1,2,NoFix,Fix;
        L=T=M1;

More: M1=L RSH 1;
        L=N1,TASK,BUSODD; sticky bits
        N1=L MRSH 1,:Sh1NSTik;   !1,2,Sh1NSTik,Sh1Stik;

        Sh1NSTik: L=ShiftCount-1;
        ShiftCount=L,SH=0;
        L=T=More;   !1,2,More,Shifted;

        Sh1Stik: T=0+1;
        L=N1 OR T,TASK;
        N1=L,:Sh1NSTik;

Fix: L=0; set both words of mantissa1 to 0
        M1=L,L=0+1,TASK;
        N1=L,:EndShift; keep sticky bit set

Shifted: NOP,:EndShift;
NoSh: NOP,:EndShift;
NoShz: SINK=M1,BUS=0; if first arg is zero, then E1=E2
        L=E2,TASK,:ExpOK;   !1,2,ExpOK,ExpWrite;
ExpWrite: E1=L;
ExpOK: NOP,:EndShift;

E1grE2: T=ShiftCount; actually, negative shift count
        L=37+T;
TASK,SH<0;
NOP,:,NoFix1; !1,2,NoFix1,Fix1;
NoFix1: L=T+M2;
More1: M2=L RSH 1;
    L->N2,TASK,BUSODD; sticky denormalize
    N2=MRSH 1,.Sh2NStik; !1,2,Sh2NStik,Sh2Stik;
Sh2NStik: L=ShiftCount+1;
    ShiftCount=L,SH=0;
    L=T-M2,:More1; !1,2,More1,Shifted1;
Sh2Stik: T=O+1;
    L=N2 OR T,TASK;
    N2=L,.Sh2NStik;
Fix1: L=O;
    M2=L,L=O+1,TASK; keep sticky bit set
    N2=L,.EndShift;
Shifted1: NOP,:EndShift;
:end of PRESHIFT
:now: ADD1 is Add(+ +), Add(- -), Sub(+ -), Sub(- +)
:and ADD2 is Add(+ -), Add(- +), Sub(+ +), Sub(- -)
:so: ADD1 if ((S1 XOR S2) XOR MODE) eq 0, and ADD2 otherwise
!1,2,ADD1,ADD2:
!1,2,A1NoCry,A1Cry;
!1,2,A1xNoCry,A1xCry;
!1,2,A1AddS,A1AddS;
!1,2,A2NoCry,A2Cry;
!1,2,A2Sign,A2NoSign;
!1,2,LowNZero,LowZero;
!1,2,HiNZero,HiZero;
!1,2,A2Norm,A2NoNorm;
;!1,2,A2NoCryL,A2CryL;
;!1,2,LowNZero1,LowZero1; defined above to avoid avoid predef error
EndShift: T=S1;
    L=S2 XOR T; 0 if same, -1 if different
    T=Mode;
    L=Lastl XOR T; 0 if ADD1, -1 if ADD2
    TASK,SH<0;
    NOP,:ADD1; !1,2,ADD1,ADD2;
ADD1: T=N1;
    L=N2+T;
    N1=L,ALUCY;
T=M1,.A1NoCry,.A1Cry;
A1Cry: L=M2+T+1,.A1Store;
A1NoCry: L=M2+T;
A1Store: M1=L,ALUCY,TASK;
    NOP,:A1xNoCry,.A1xCry;
A1xCry: T=L+M1; post shift
    M1=L RSH 1;
    L=N1,TASK,BUSODD;
    N1=MRSH 1,.A1AddS; !1,2,A1AddS,A1AddS;
A1AddS: T=O-1;
    L=N1 OR T,TASK;
    N1=L;
A1AddS: T=100000;
    L=M1 OR T,TASK; high order bit should have been shifted in
    M1=L;
    L=E1+1,TASK;
    E1=L,:RePack;
A1xNoCry: NOP,:RePack;
ADD2: T=N2;
    L=N1-T;
    N1=L,ALUCY; low order result
T=M2,:A2NoCry; !1,2,A2NoCry,A2Cry;
A2NoCry: L=M1-T-1,:A2C; no carry, do one's complement subtract
A2Cry: L=M1-T; carry, do two's complement subtract
A2C: M1=L,ALUCY,TASK;
    NOP,:A2Sign; if no carry, sign changed!!!!
    ;!1,2,A2Sign,A2NoSign;
A2Sign: T=N1,BUS=0;  double length negate starts here
   L=O-T,:LowNZero;   !1,2,LowNZero,LowZero;
LowNZero:  N1=L,T=O-1;
   L=M1 XOR T,:A2Cx;  complement
LowZero:   T=M1;
   L=O-T;  negate (note that N1 is already 0, so no need to update it)
A2Cx:    M1=L,T=O-1;
   L=S1 XOR T,TASK; complement sign
  S1=L;
A2NoSign:  L=O,TASK;
   ShiftCount=L;
   L=M1,BUS=0;
   NOP,:HiNZero;  !1,2,HiNZero,HiZero;
HiNZero:   TASK,SH<0;
   NOP,:A2Norm;  !1,2,A2Norm,A2NoNorm;
A2Norm:  L=N1;
   NOP,SH<0;
   N1=L SH I,T=O,:A2NoCryL;  !1,2,A2NoCryL,A2CryL;
A2CryL:  T= ALLONES;
A2NoCryL:  L=M1;
   L=M1 ML SH 1;
   L=ShiftCount+1,TASK;
   ShiftCount=L;
   L=M1,:HiNZero;
A2NoNorm:  T=ShiftCount;
   L=E1-T,TASK,:MulDone;
HiZero:   L=N1,BUS=0;
   M1=L,L=O,:LowNZero1;  !1,2,LowNZero1,LowZero1;
LowNZero1:  N1=L;  zero out low order
   L=O,TASK;
   ShiftCount=L;  16 shifts done like wildfire
   L=M1,:HiNZero;
   !1,2,LNZNeg,LNZPos;
LowZero1:  SINK=S1,BUS=0;
   T=100000,:LNZNeg;
   !1,2,LNZNeg,LNZPos;
LNZNeg:  T=O,:LNZNeg;
   LNZNeg:  Arg1=L,L-T,TASK;
   Arg0=L,:FPdpush;

; Compare
;---------------------------------------------------------------
; !1,2,FCAZ,FCAZ;
; !1,2,FCAZBNZ,FCAZBNZ;
; !1,2,FCSD,FCSS;
; !1,2,FCEDiff,FCESame;
; !1,2,FCSSgn,FCSSgnB;
; !1,1,FCESame;
; !1,2,FCMDiff,FCMSame;
; !1,1,FCMSame;
; !1,2,FCNDiff,FCNSame;
; !1,1,FCRgt; used farther up
; !1,2,FCAZBNZ,FCAZBNZ;
; !1,2,FCA1sB,FCAGrB;
; !1,2,FCA1sB1;
FComp:  L=6,TASK,:LoadArgs;
LoadRet6:  SINK=M1,BUS=0;
   SINK=M2,BUS=0,TASK,:FCANZ;  !1,2,FCANZ,FCAZ;
FANZ:  NOP,:FCANZBNZ;  !1,2,FCANZBNZ,FCAZBNZ;
FANZBNZ:  SINK=S1,BUS=0,TASK,:FCAST;  return according to sign of A
FANZBNZ:  T=S1;  A and B not 0
   L=S2 XOR T;
   NOP,TASK,SH=0;
   NOP,:FCSD;  !1,2,FCSD,FCSS;
FCSD:  SINK=S1,BUS=0,TASK,:FCAST;  return according to sign of A
FCSS:  T=E2;
   L=E1-T;
   L=M1,SH=0;
L=M2-T,SH<0,:FCEDiff; !1,2,FCEDiff,FCESame;

FCEDiff: NOP,:FCSgnA; !1,2,FCSgnA,FCSgnB;
FCSgnA: SINK+S1,BUS=O,TASK,:FCAST;
FCSgnB: SINK+S2,BUS=O,TASK,:FCBST;

; In what follows, we do unsigned compares. ALUCY will branch if
; a > b on execution of a-b
FCESame: T=M1,SH=0,:FCESame1; !1,1,FCESame1;
FCESame1: L=M2-T,ALUCY,:FCMDiff; !1,2,FCMDiff,FCMSame;

FCMDiff: NOP,:FCSgnA; (!1,2,FCSgnA,FCSgnB;)
FCMSame: NOP,SH=0,:FCMSame1; !1,1,FCMSame1;
FCMSame1: L=O,TASK,:FCRet; !1,1,FCRet;

; BUS=0 in instruction calling FCAST, will branch if op. 1 is plus
FCAS: NOP,:FCASB; !1,2,FCASB,FCASB;
FCASB: L=O+1,TASK,:FCRet; !1,1,FCRet;
FCASB: L=O-1,TASK,:FCRet;

; BUS=0 in instruction calling FCAST, will branch if op. 2 is plus
FCBST: NOP,:FCBST; !1,2,FCBST,FCBST;
FCBST: L=O+1,TASK,:FCRet; !1,1,FCRet;
FCBST: L=O-1,TASK,:FCRet;

FCRet: ArgO~L,ShortRet; called from FSticky also

; Sticky (microcode copy of sticky flags)
;-----------------------------------------------
;-----------------------------------------------
!1,2,FSErr,FSOk;

FSticky: L=stk0-1,TASK;
stk0=stk0
T=Sticky;
SINK=stk0,BUS=0;
L=stk0,:FSErr; !1,2,FSErr,FSOk;
FSOk: Sticky~L,L=T,TASK,:FCRet;
FSErr: NOP,TASK,:RamStkErr;
uCodeVersion 'is used by RunMesa to determine what version of the Mesa microcode is in ROM. This version number should be incremented by 1 for every official release of the microcode. 'uCodeVersion' is mapped by RunMesa to the actual version number (which appears as a comment above). The reason for this mapping is the limited number of constants in the Alto constants ROM; otherwise, we would obviously have assigned 'uCodeVersion' the true microcode version number.

The current table in RunMesa should have the following correspondences:

<table>
<thead>
<tr>
<th>uCodeVersion</th>
<th>Microcode version</th>
<th>Mesa release</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>34</td>
<td>4.1</td>
</tr>
<tr>
<td>1</td>
<td>39</td>
<td>5.0</td>
</tr>
<tr>
<td>2</td>
<td>41</td>
<td>6.0</td>
</tr>
</tbody>
</table>

sCodeVersion $2;

;Completely rewritten by Roy Levin, Sept-Oct. 1977
;Modified by Johnsson, July 25, 1977 10:20 AM
;First version assembled 5 June 1975.
;Developed from Lampson's MESA.U of 21 March 1975.

GLOBAL CONVENTIONS AND ASSUMPTIONS

1) Stack representation:
   stkP=O => stack is empty
   stkP=10 => stack is full
   The validity checking that determines if the stack pointer is within this range is somewhat perfunctory. The approach taken is to include specific checks only where there absence would not lead to some catastrophic error. Hence, the stack is not checked for underflow, since allowing it to become negative will cause a disaster on the next stack dispatch.

2) Notation:
   Instruction labels correspond to opcodes in the obvious way. Suffixes of A and B (capitalized) refer to alignment in memory. 'A' is intended to suggest the right-hand byte of a memory word; 'B' is intended to suggest the left-hand byte. Labels terminating in a lower-case letter generally name local branch points within a particular group of opcodes. (Exception: subroutine names.) Labels terminating in 'x' generally exist only to satisfy alignment requirements imposed by various dispatches (most commonly IR~ and BfA in instruction fetch).

3) Tasking:
   Every effort has been made to ensure that a 'TASK' appears approximately every 12 instructions. Occasionally, this has not been possible, but (it is hoped that) violations occur only in infrequently executed code segments.

4) New symbols:
   In a few cases, the definitions of the standard Alto package (AltoConsts23.MU) have not been quite suitable to the needs of this microcode. Rather than change the standard package, we have defined new symbols (with names beginning with 'm') that are to be used instead of their standard counterparts. All such definitions appear together in Mesab.Mu.

5) Subroutine returns:
   Normally, subroutine returns using IDISP require one to deal with (the nuisance of) the dispatch caused by loading IR. Happily, however,
no such dispatch occurs for 'msr0' and 'sr1' (the relevant bits are 0). To cut down on alignment restrictions, some subroutines assume they are called with only one of two returns and can therefore ignore the possibility of a pending IR dispatch. Such subroutines are clearly noted in the comments.

6) Frame pointer registers (lp and gp):
   These registers normally (i.e. except during Xfer) contain the addresses of local 2 and global 1, respectively. This optimizes accesses in such bytecodes as LL3 and SG2, which would otherwise require another cycle.
There is a fundamental difficulty in the selection of addresses that are known and used outside the Mesa emulator. The problem arises in trying to select a single set of addresses that can be used regardless of the Alto's control memory configuration. In effect, this cannot be done. If an Alto has only a RAM (in addition, of course, to its basic ROM, ROMO), then the problem does not arise. However, suppose the Alto has both a RAM and a second ROM, ROM1. Then, when it is necessary to move from a control memory to one of the other two, the choice is conditioned on (1) the memory from which the transfer is occurring, and (2) bit 1 of the target address. Since we expect that, in most cases, an Alto running Mesa will have the Mesa emulator in ROM1, the externally-known addresses have been chosen to work in that case. They will also work, without alteration, on an Alto that has no ROM1. However, if it is necessary to run Mesa on an Alto with ROM1 and it is desired to use a Mesa emulator residing in the RAM (say, for debugging purposes), then the address values in the RAM version must be altered. This implies changes in both the RAM code itself and the Nova code that invokes the RAM (via the Nova JMPRAM instruction). Details concerning the necessary changes for re-assembly appear with the definitions below.

Note concerning Alto IVs and Alto IIs with retrofitted 3K control RAMs:

The above comments apply uniformly to these machines if "RAM" is systematically replaced by "RAMI" and "RaMI" is systematically replaced by "RAMO".

%I,1777,0,nextBa; forced to location 0 to save a word in JRAM

Emulator Entry Point Definitions

These addresses are known by the Nova code that interfaces to the emulator and by RAM code executing with the Mesa emulator in ROM1. They have been chosen so that both such "users" can use the same value. Precisely, this means that bit 1 (the 400 bit) must be set in the address. In a RAM version of the Mesa emulator intended to execute on an Alto with a second ROM, bit 1 must be zero.

%I,1777,420,Mgo; Normal entry to Mesa Emulator - load state of process specified by ACO.

%I,1777,400,next,nextA; Return to 'next' to continue in current Mesa process after Nova or RAM execution.

$Minterpret $L004400,0,0; Documentation refers to 'next' this way.

%I,1777,776,DSTrl,Mstopc; Return addresses for 'Savestate'. By standard convention, 'Mstopc' must be at 777.

Linkage from RAM to ROM1

The following predefs must correspond to the label definitions in MesabROM.mu

%I,1777,406,Intstop; must correspond to romIntstop
%I,1777,407,Untail; must correspond to romUntail
%7,1777,430,XferGT,Xfer,Mstopr,PORTOp,LSTr,ALOCrfr; Xfer must agree with romXfer
Linkage from Mesa emulator to ROMO

The Mesa emulator uses a number of subroutines that reside in ROMO. In posting a return address, the emulator must be aware of the control memory in which it resides, RAM or ROM1. These return addresses must satisfy the following constraint:

- no ROM1 extant or emulator in ROM1 => bit 1 of address must be 1
- ROM1 extant and emulator in RAM => bit 1 of address must be 0

In addition, since these addresses must be passed as data to ROMO, it is desirable that they be available in the Alto's constants ROM. Finally, it is desirable that they be chosen not to mess up too many pre-defs. It should be noted that these issues do not affect the destination location in ROMO, since its address remains fixed (even with respect to bit 1 mapping) whether the Mesa emulator is in RAM or ROM1. [Note pertaining to retrofitted Alto IIs with 3K RAMs: to avoid confusion, the comments above and below have not been revised to discuss 3K control RAMs, although the values suggested are compatible with such machines.]

MUL/DIV linkage:

An additional constraint peculiar to the MUL/DIV microcode is that the high-order bits of the return address be 1's. Hence, the recommended values are:

- no ROM1 extant or emulator in ROM1 => MULDIVretloc = 177576B (OK to be odd)
- ROM1 extant and emulator in RAM => MULDIVretloc = 177162B (OK to be odd)

The third value in the following pre-def must be: ((MULDIVretloc-2) AND 777B)

CYCLE linkage:

A special constraint here is that WFretloc be odd. Recommended values are:

- no ROM1 extant or emulator in ROM1 => Fieldretloc = 452B, WFretloc = 523B
- ROM1 extant and emulator in RAM => Fieldretloc = 34104B, WFretloc = 14023B

The third value in the following pre-def must be: (Fieldretloc AND 1777B)
Instruction fetch

State at entry:
1) \( ib \) holds either the next instruction byte to interpret (right-justified) or 0 if a new word must be fetched.
2) Control enters at one of the following points:
   a) next: \( ib \) must be interpreted
   b) nextA: \( ib \) is assumed to be uninteresting and a new instruction word is to be fetched.
   c) nextXB: a new word is to be fetched, and interpretation is to begin with the odd byte.
   d) nextAdeaf: similar to 'nextA', but does not check for pending interrupts.
   e) nextXBdeaf: similar to 'nextXB', but does not check for pending interrupts.

State at exit:
1) \( ib \) is in an acceptable state for subsequent entry.
2) \( T \) contains the value 1.
3) A branch (1) is pending if \( ib = 0 \), meaning the next instruction may return to 'nextA'. (This is subsequently referred to as "ball 1", and code that nullifies its effect is labelled as "dropping ball 1".)
4) If a branch (1) is pending, \( L = 0 \). If no branch is pending, \( L = 1 \).
; Address pre-definitions for bytecode dispatch table.

; Table must have 2 high-order bits on for BUS branch at 'nextAni'.

; Warning! Many address inter-dependencies exist - think (at least) twice
; before re-ordering. Inserting new opcodes in previously unused slots,
; however, is safe.

; XMESA Note: RBL, WBL, and BLTL exist for XMESA only.

%7.1777.1400, NOOP, ME, MRE, MXW, MXD, NOTIFY, BCAST, REQUEUE;
%7.1777.1410, LL0, LL1, LL2, LL3, LL4, LL5, LL6, LL7;
%7.1777.1420, LLB, LLDB, SL0, SL1, SL2, SL3, SL4, SL5;
%7.1777.1430, SL6, SL7, SLB, PL0, PL1, PL2, PL3, LG0;
%7.1777.1440, LG1, LG2, LG3, LG4, LG5, LG6, LG7, LG8;
%7.1777.1450, LGDB, SG0, SG1, SG2, SG3, SGB, LG0, LI1;
%7.1777.1460, LIZ, LI3, LI4, LI5, LI6, LIN1, LINI, LIGB;
%7.1777.1470, LIW, LIMB, LADR8, GADR8, ....
%7.1777.1500, R0, R1, R2, R3, R4, RB, WO, WI;
%7.1777.1510, W2, WB, RF, WF, RDB, RDD, WDB, WDO;
%7.1777.1520, RSTR, WSTR, RXLP, WXLPI, RIGP, WILP, RILO;
%7.1777.1530, WSO, WSB, WSF, WDB, RFC, RFS, WFS, RBL;
%7.1777.1540, W, WBL, ....
%7.1777.1550, ....;
%7.1777.1560, , SLDB, SGDB, PUSH, POP, EXCH, LINKB;
%7.1777.1570, DUP, NILCK, BNDCK, ....;
%7.1777.1600, J2, J3, J4, J5, J6, J7, J8, J9;
%7.1777.1610, JB, JW, JEQ2, JEQ3, JEQ4, JEQ5, JEQ6, JEQ7;
%7.1777.1620, JEQ8, JEQ9, JEQ, JNE2, JNE3, JNE4, JNE5, JNE6;
%7.1777.1630, JNE7, JNE8, JNE9, JNE, JLEB, JGB, JLB;
%7.1777.1640, JUB, JUGB, JUBL, JUEB, JUEB, JUEB, JUEB, JUEB;
%7.1777.1650, ADD, SUB, MUL, DBL, DIV, LDIV, NEG, INC;
%7.1777.1660, AND, OR, XOR, SHIFT, DADD, DSUB, DCOMP, DCOMP;
%7.1777.1670, ADD01, ....;
%7.1777.1700, EFC0, EFC1, EFC2, EFC3, EFC4, EFC5, EFC6, EFC7;
%7.1777.1710, EFC8, EFC9, EFC10, EFC11, EFC12, EFC13, EFC14, EFC15;
%7.1777.1720, EFCB, LFC1, LFC2, LFC3, LFC4, LFC5, EFC6, LFC7;
%7.1777.1730, LCanghai, LCanghai, LCanghai, LCanghai, LCanghai, LCanghai;
%7.1777.1740, , LFCB, SFC, REI, LLLB, PORTO, PORTI, KFCB;
%7.1777.1750, DESCGB, DESCGB, BLT, BILT, BLTC, , ALLOC, FREE;
%7.1777.1760, IWDC, DBC, DBC, STOP, CATCH, MISC, BITBLT, STARTIO, JRAM;
%7.1777.1770, DST, LST, LSTF, , WR, RR, BRK, STKUF;
%7.1777.1780, -300-307
Main interpreter loop

Enter here to interpret ib. Control passes here to process odd byte of previously fetched word or when preceding opcode "forgot" it should go to 'nextA'. A 'TASK' should appear in the instruction preceding the one that branched here.

next: L=0, :nextBa;
nextBa: SINK+ib, BUS;
ib=L, T=0+1, BUS=0, :NOOP;

(NOOP - must be opcode 0
control also comes here from certain jump instructions)

!1,1,nextAput;

NOOP: L=mpc+T, TASK, :nextAput;
; Enter here to fetch new word and interpret even byte. A 'TASK' should appear in the
; instruction preceding the one that branched here.

nextA:    L=XMAR=mpc+1, :nextAcom;    ; initiate fetch

; Enter here when fetch address has been computed and left in L. A 'TASK' should
; appear in the instruction that branches here.

nextAput:  temp=L;                ; stash to permit TASKing
         L=XMAR-temp, :nextAcom;

; Enter here to do what 'nextA' does but without checking for interrupts

nextAdef:  L=XMAR=mpc+1;
nextAdefa:  mpc=L, BUS=0, :nextAcomx;

; Common fetch code for 'nextA' and 'nextAput'
!1,2,nextA1,nextA1i;
!1,2,nextA1i1,nextA1i1i;

nextAcom:  mpc=L;
         SINK=WWW, BUS=0;    ; updated pc
nextAcomx:  T=177400, :nextA1;    ; check pending interrupts

; No interrupt pending. Dispatch on even byte, store odd byte in ib.

nextA1i:   L=MD AND T, BUS, :nextAgo;
nextAgo:    ib=L LCY 8, L=T=0+1, :NOOP;

; Interrupt pending - check if enabled.

nextA1i:    L=MD;
         SINK=wdc, BUS=0;
         T=M.T, :nextA1i1;
nextA1i1:   SINK=M, L=T, BUS, :nextAgo;

; Interrupt pending and enabled.
!1,2,nextXB1i1i,nextXB1i1i;

nextA1i1i:  L=mpc-1;
         mpc=L, L=0, :nextXB1i1;

. . .

. . .
; Enter here to fetch word and interpret odd byte only (odd-destination jumps).
; !1,2,nextXB1,nextXBni:

nextXB:      L=XMAR=mpc+1;
             SINK=NWW, BUS=0, :nextXBdeaf;      ; check pending interrupts
             
; Enter here (with branch (1) pending) from Xfer to do what 'nextXB' does but without
; checking for interrupts. L has appropriate word PC.

nextXBdeaf:  mpc=L, :nextXB1;

; No interrupt pending. Store odd byte in ib.

nextXBni:    L=MD, TASK, :nextXBini;
nextXBini:   ib=L LCY 8, :next;
             
; Interrupt pending - check if enabled.

nextXB1:     SINK=wdc, BUS=0, :nextXBni;
             
; Interrupt pending and enabled.

nextXBii:    ib=L, :Intstop;
             1b = 0 for even, ~= 0 for odd
Subroutines

The two most heavily used subroutines (Popsub and Getalpha) often share common return points. In addition, some of these return points have additional addressing requirements. Accordingly, the following predefinitions have been rather carefully constructed to accommodate all of these requirements. Any alteration is fraught with peril.

[A historical note: an attempt to merge in the returns from FetchAB as well failed because more than 310 distinct return points were then required. Without adding new constants to the ROM, the extra returns could not be accommodated. However, for Popsub alone, additional returns are possible - see Xpopsub.]

Return Points (sr0-sr17)

!17,20,Fieldra,SFCr,pushTB,pushTA,LLBr,LGBr,SLBr,SGBr, LADBRBr,GADBRBr,RFr,Xret,INCr,RBr,WBr,Xpopret;

Extended Return Points (sr20-sr37)

Note: KFCr and EFCr must be odd!

!17,20,XbrkBr,KFCr,LFCr,EFCr,WSDBra,DBLr,LINBr,LDIVr, Dpush,Dpop,RODr,Spltcomr,RXLPrb,WXLPrb,MISCr,RWBLa;

Returns for Xpopsub only

!17,20,WSTRrB,WSTRrA,JRAMr,WRr,STARTIr,PORTOr,WDOr,ALLOCrx, FREErx,NEGr,RFSra,RFSrb,WFSra,DESCBcom,RFCr,NILCKr;

Extended Return Machinery (via Xret)

!1,2,XretB,XretA:

Xret: SINK=DISP, BUS, :XretB;

XretB: :XbrkBr;

XretA: SINK=0, BUS=0, :XbrkBr; keep ball 1 in air
: Pop subroutine:
  Entry conditions:
  Normal IR linkage
  Exit conditions:
  Stack popped into T and L

!1,1.Popsub;  shakes B/A dispatch
!7,1.Popsuba;  shakes IR- dispatch
!17,20.Tpop,Ttop0,Ttop1,Ttop2,Ttop3,Ttop4,Ttop5,Ttop6,Ttop7.......;

Popsub:     L=stk p-1, BUS, TASK, :Popsuba;
Popsuba:    stk p=L, :Tpop; old stk p > 0

: Xpop subroutine:
  Entry conditions:
  L has return number
  Exit conditions:
  Stack popped into T and L
  Invoking instruction should specify 'TASK'

!1,1.Xpopsub;  shakes B/A dispatch

Xpopsub:    saveret=L;
Tpop:       IR=sr17, :Popsub;

Xpopret:    SINK=saveret. BUS;  :WSTRrB;
Note: putting Tpop here makes
stack underflow logic work if
stk p=0
: Getalpha subroutine:
  Entry conditions:  
  : L untouched from instruction fetch
  : Exit conditions:  
  : alpha byte in T  
  : branch 1 pending if return to 'nextA' desirable  
  : L=0 if branch 1 pending, L=1 if no branch pending

: Getalpha:  
  T=ib, IDISP;
: Getalphax:  
  ib=L RSH 1, L=0, BUS=0, :Fieldra;
: Getalphax:  
  ib=0, set branch 1 pending
: GetalphaA:  
  L=XMAR+mpc+1;
: GetalphaAx:  
  mpc=L;
  T=177400;
  L=MD AND T, T=MD;
: Getalphab:  
  T=377.T, IDISP;
  ib=L LCY 8, L=0+1, :Fieldra;
  T now has alpha
  return: no branch pending

: FetchAB subroutine:
  Entry conditions: none
  Exit conditions:  
  : T: <mpc>+1>  
  : ib: unchanged (caller must ensure return to 'nextA')

: FetchAB:  
  L=XMAR+mpc+1, :FetchABx;
: FetchABx:  
  drops ball 1
: FetchABx:  
  shake IR= dispatch
: FetchABx:  
  mask for new ib
: FetchABx:  
  L: new ib, T: whole word
: FetchABx:  
  T now has alpha
: FetchABx:  
  return: no branch pending

: FetchABx:  
  shake IR= dispatch
: FetchABx:  
  returns points
Splitalpha subroutine:

Entry conditions:
- L: return index
- entry at Splitalpha if instruction is A-aligned, entry at SplitalphaB if instruction is B-aligned
- entry at Splitalphar splits byte in T (used by field instructions)

Exit conditions:
- lefthalf: alpha[0-3]
- righthalf: alpha[4-7]

------------------------------------------------------------------

!1,2,Splitalpha,SplitalphaB; subroutine returns
!1,1,Splitx; drop ball 1
%!60,377,217,Split0,Split1,Split2,Split3,Split4,Split5,Split6,Split7;
!1,2,Splitout0,Splitout1;
!7,10,RILPr,RIGPr,WILPr,RXLPra,WXLPra,Fieldrb,..; subroutine returns

Splitalpha: saveret=L, L=0+1, :Splitcom;
SplitalphaB: saveret=L, L=0, BUS=0, :Splitcom;

Splitalphar: IR=sr33, :Getalpha;
Splitcom: L=17 AND T, :Splitx;
Splitx: righthalf=L, L=T, TASK;
      temp=L;
      L=temp, BUS;
      temp=L LCY 8, SH<O, :Split0;
      L,T:alpha[1-3]

Split0: L=T=0, :Splitout0;
Split1: L=T=ONE, :Splitout0;
Split2: L=T=2, :Splitout0;
Split3: L=T=3, :Splitout0;
Split4: L=T=4, :Splitout0;
Split5: L=T=5, :Splitout0;
Split6: L=T=6, :Splitout0;
Split7: L=T=7, :Splitout0;

Splitout0: SINK=saveret, BUS, TASK;
          lefthalf=L, :RILPr;
          L:alpha[0-3]
          dispatch return
          lefthalf:alpha[0-3]
; Dispatches
;
; Pop-into-T (and L) dispatch:
; dispatches on old stkp, so Tpop0 = 1 mod 20B.
;
Tpop0:     L+T-stk0, IDISP, :Tpopexit;
Tpop1:     L+T-stk1, IDISP, :Tpopexit;
Tpop2:     L+T-stk2, IDISP, :Tpopexit;
Tpop3:     L+T-stk3, IDISP, :Tpopexit;
Tpop4:     L+T-stk4, IDISP, :Tpopexit;
Tpop5:     L+T-stk5, IDISP, :Tpopexit;
Tpop6:     L+T-stk6, IDISP, :Tpopexit;
Tpop7:     L+T-stk7, IDISP, :Tpopexit;

Tpopexit:  :Fieldra;  to permit TASK in Popsub
pushMD dispatch:
  pushes memory value on stack
  The invoking instruction must load MAR and may optionally keep ball 1 in the air by having a branch pending. That is, entry at 'pushMD' will cause control to pass to 'next', while entry at 'pushMDA' will cause control to pass to 'nextA'.

pushMD:  L<-stkP+1, IR<-stkP;
          stkP=L, T=0+1, :pushMDa;
          (IR+- causes no branch)
pushMDA: L<-stkP+1, IR<-stkP;
          stkP=L, T=0, :pushMDa;
          (IR- causes no branch)
pushMDA: SINK=DISP, L=T, BUS;
          L=MD, SH=0, TASK, :push0;
          dispatch on old stkP value

Push-T dispatch:
  pushes T on stack
  The invoking instruction may optionally keep ball 1 in the air by having a branch pending. That is, entry at 'pushTB' will cause control to pass to 'next', while entry at 'pushTA' will cause control to pass to 'nextA'.

pushTB:  L<-stkP+1, BUS, :pushTIB;
pushTA:  L<-stkP+1, BUS, :pushTIA;
pushTIB: stkP=L, L=T, TASK, :push0;
pushTIA: stkP=L, BUS=0, L=T, TASK, :push0;
          BUS=0 keeps branch pending

push dispatch:
  strictly vanilla-flavored
  may (but need not) have branch (1) pending if return to 'nextA' is desired
  invoking instruction should specify TASK

Note: the following pre-def occurs here so that dpushof1 can be referenced in push10

dpush, .dpush1, dpush2, dpush3, dpush4, dpush5, dpush6, dpush7, dpushof1, dpushof2,....;
push0: stk0=L, :next;
push1: stk1=L, :next;
push2: stk2=L, :next;
push3: stk3=L, :next;
push4: stk4=L, :next;
push5: stk5=L, :next;
push6: stk6=L, :next;
push7: stk7=L, :next;
push10: :dpushof1; honor TASK, stack overflow
Double-word push dispatch:
- picks up alpha from ib, adds it to T, then pushes <result> and <result+1>
- entry at 'Dpusha' substitutes L for ib.
- entry at 'Dpushc' is used by RCLK logic.
- entry at 'dpush' is used by MUL/DIV/IDIV logic.
- returns to 'nextA' if ib = 0 or entry at 'Dpush'

11,2,DpA,DpB;
4,1,Dpushx;

Dpush:        MAR=L-ib+T, :DbB;
Dpusha:       SINK-ib, BUS=0;
               MAR=L+M+T, :DpA;
DpA:          IR=0, :Dpushb;
DpB:          IR=2000, :Dpushb;
Dpushb:       temp=L, :Dpushx;
Dpushx:       L=MD, TASK, :Dpushc;
Dpushc:       taskhole=L;
               T=0+1;
               L=stkp+T+1;
               MAR=temp+1;
               stkp=L;
               L=taskhole;
               SINK=stkp, BUS, :dpush;

dpush:        T=MD, :dpush;

dpush1:       stk0=L, L+T, TASK, mACSOURCE, :push1;
dpush2:       stk1=L, L+T, TASK, mACSOURCE, :push2;
dpush3:       stk2=L, L+T, TASK, mACSOURCE, :push3;
dpush4:       stk3=L, L+T, TASK, mACSOURCE, :push4;
dpush5:       stk4=L, L+T, TASK, mACSOURCE, :push5;
dpush6:       stk5=L, L+T, TASK, mACSOURCE, :push6;
dpush7:       stk6=L, L+T, TASK, mACSOURCE, :push7;
dpushof1:     T=sStackOverflow, :KFCr;
dpushof2:     T=sStackOverflow, :KFCr;
TOS+T dispatch:
  adds TOS to T, then initiates memory operation on result.
  used as both dispatch table and subroutine - fall-through to 'pushMD'.
  dispatches on old stkp, so MAsTkT0 = 1 mod 20B.

!17,20,MAStkT,MAStkT0,MAStkT1,MAStkT2,MAStkT3,MAStkT4,MAStkT5,MAStkT6,MAStkT7......:

MAStkT0:  MAR+stk0+T. :pushMD;
MAStkT1:  MAR+stk1+T. :pushMD;
MAStkT2:  MAR+stk2+T. :pushMD;
MAStkT3:  MAR+stk3+T. :pushMD;
MAStkT4:  MAR+stk4+T. :pushMD;
MAStkT5:  MAR+stk5+T. :pushMD;
MAStkT6:  MAR+stk6+T. :pushMD;
MAStkT7:  MAR+stk7+T. :pushMD;

; Common exit used to reset the stack pointer
; the instruction that branches here should have a 'TASK'
; Setstkp must be odd, StkOf1w used by PUSH
!17,11,Setstkp,......,StkOf1w:

Setstkp:  stkp=L. :next;  branch (1) may be pending
StkOf1w:   :dpushof1;  honor TASK, dpushof1 is odd

; Stack Underflow Handling

StkUf:   T=sStackUnderflow, :KFCr;  catches dispatch of stkp = -1
Store dispatch:
: pops TOS to MD.
: called from many places.
: dispatches on old stkp, so MDpop0 = 1 mod 20B.
: The invoking instruction must load MAR and may optionally keep ball 1
: in the air by having a branch pending. That is, entry at 'StoreB' will
: cause control to pass to 'next', while entry at 'StoreA' will cause
: control to pass to 'nextA'.

!1.2.StoreBa, StoreAa;
!17.20.MDpopu, MDpop0, MDpop1, MDpop2, MDpop3, MDpop4, MDpop5, MDpop6, MDpop7,......;

StoreB:
L=stk-1, BUS;

StoreBa:
stk=stk, TASK, :MDpopuf;

StoreAa:
L=stk-1, BUS;

MDpop0:
MD=stk0, next;

MDpop1:
MD=stk1, next;

MDpop2:
MD=stk2, next;

MDpop3:
MD=stk3, next;

MDpop4:
MD=stk4, next;

MDpop5:
MD=stk5, next;

MDpop6:
MD=stk6, next;

MDpop7:
MD=stk7, next;

Double-word pop dispatch:
: picks up alpha from ib, adds it to T, then pops stack into result and
: result+1
: entry at 'Dpopa' substitutes L for ib.
: returns to 'nextA' <= ib = 0 or entry at 'Dpop'

!17.20, dpopuf1, dpop1, dpop2, dpop3, dpop4, dpop5, dpop6, dpop7,......;
!11.1, Dpopb: required by placement of

Dpop:
L=T+2ib+T+1;

MDpopuf:
IR=0, :Dpopb;

Dpopa:
L=T+2M+T+1;

Dpopb:
MAR=T, temp=L;
dpopuf2:
L=stk-1, BUS;

stk=stk, TASK, :dpopuf2;

dpopuf1: :StkUF;
dpop1: MD=stk1, :Dpopx;
dpop2: MD=stk2, :Dpopx;
dpop3: MD=stk3, :Dpopx;
dpop4: MD=stk4, :Dpopx;
dpop5: MD=stk5, :Dpopx;
dpop6: MD=stk6, :Dpopx;
dpop7: MD=stk7, :Dpopx;

Dpopx: SINK=DISP, BUS=0;

MAStkT: MAR=temp-1, :StoreB;

Note: MDpopuf is merely a
convenient label which leads to a BUS dispatch on stkp in
the case that stkp is -1. It is used by the Store dispatch
above.
; Get operation-specific code from other files

#MesacROM.mu;
#MesadROM.mu;
Jumps

The following requirements are assumed:
1) J2-J9, JB are usable (in that order) as subroutine returns (by JEQx and JNEx).
2) since J2-J9 and JB are opcode entry points, they must meet requirements set by opcode dispatch.

Jn - jump PC-relative

!1,2,JnA,Jbranchf;
J2:  L=ONE, :JnA;
J3:  L=2, :JnA;
J4:  L=3, :JnA;
J5:  L=4, :JnA;
J6:  L=5, :JnA;
J7:  L=6, :JnA;
J8:  L=7, :JnA;
J9:  L=10, :JnA;
JnA:  L=M-1, :Jbranchf;  A-aligned - adjust distance

JB - jump PC-relative by alpha, assuming:
  JB is A-aligned
Note: JEQB and JNEB come here with branch (1) pending

!1,1,JBx;  shake JEQB/JNEB branch
!1,1,Jbranch;
JB:  T=ib, :JBx;
JBx:  L=400 OR T;  +DISP will do sign extension
       IR=M;
       L=DISP-1, :Jbranch;  400 above causes branch (1)
                          L: ib (sign extended) - 1

JW - jump PC-relative by alphabeta, assuming:
  if JW is A-aligned, B byte is irrelevant
  alpha in B byte, beta in A byte of word after JW

JW:  IR=sr1, :FetchAB;  returns to JW
JWr:  L=ALLONES+T, :Jbranch;  L: alphabeta-1

Jump destination determination
L has (signed) distance from even byte of word addressed by mpc+1

!1,2,Jforward,Jbackward;
!1,2,Jeven,Jodd;
Jbranch:  T=0+1, SH<0;
Jbranchf:  SINK=M, BUSOODD, TASK, :Jforward;
Jforward:  temp=L RSH 1, :Jeven;
Jbackward:  temp=L MRSH 1, :Jeven;
Jeven:  T=temp+1, :NOOP;
Jodd:  T=temp+1, :nextXB;
JZEQB - if TOS (popped) = 0, jump PC-relative by alpha, assuming:
stack has precisely one element
JZEQB is A-aligned (also ensures no pending branch at entry)

!1,2,Jcz,Jco;

JZEQB:  SINK=stk0, BUS=0;
        L=stkp-1, TASK, :Jcz;

JZNEB - if TOS (popped) ≠ 0, jump PC-relative by alpha, assuming:
stack has precisely one element
JZNEB is A-aligned (also ensures no pending branch at entry)

!1,2,JZNEBne,JZNEBeq;

JZNEB:  SINK=stk0, BUS=0;
        L=stkp-1, TASK, :JZNEBne;

JZNEBne:  stkp=L, :JB;
JZNEBeq:  stkp=L, :nextA;

; JEQn - if TOS (popped) = TOS (popped), jump PC-relative by n, assuming:
; stack has precisely two elements

!1,2,JEQnB,JEQnA; shake IR+ dispatch

JEQ2: IR=SR0, L=T, :JEQnB; returns to J2
JEQ3: IR=SR1, L=T, :JEQnB; returns to J3
JEQ4: IR=SR2, L=T, :JEQnB; returns to J4
JEQ5: IR=SR3, L=T, :JEQnB; returns to J5
JEQ6: IR=SR4, L=T, :JEQnB; returns to J6
JEQ7: IR=SR5, L=T, :JEQnB; returns to J7
JEQ8: IR=SR6, L=T, :JEQnB; returns to J8
JEQ9: IR=SR7, L=T, :JEQnB; returns to J9

; JEQB - if TOS (popped) = TOS (popped), jump PC-relative by alpha, assuming:
; stack has precisely two elements
; JEQB is A-aligned (also ensures no pending branch at entry)

JEQB: IR=SR10, :JEQnA; returns to JB

; JEQ common code

!1,2,JEQcom,JEQNEcom; return points from JEQNEcom

JEQcom: TEMP=L RSH 1, L=T, :JEQcom; temp:0, L:1 (for JEQcom)
JEQNEcom: TEMP=L, L=T, :JEQNEcom; temp:1, L:1 (for JEQNEcom)

!1,2,JEQne,JEQeq;

JEQcom: L=stkP-T-1, :JEQne; L: old stkP - 2
JEQne: SINK=TEMP, BUS, TASK, :SetstkP;
JEQeq: stkP=L, IDISP, :JEQNExxx;

; JEQ/JNE common code

; !7,1,JEQNEcom; appears above with JEQ
; !1,2,JEQcom,JEQNEcom; appears above with JEQB

JEQNEcom: T=stk1;
L=stkO-T, SH=0;
T O=1, SH=0, :JEQcom;
dispatch EQ/NE
JEQNEeq: SINK=TEMP, BUS, :J2;
even/odd dispatch
; JNEn - if TOS (popped) == TOS (popped), jump PC-relative by n, assuming:
; stack has precisely two elements

!1,2,JNeNb,JNeA:
JNE2:      IR=sr0, L=T, :JNeNb;                   returns to J2
JNE3:      IR=sr1, L=T, :JNeNb;                   returns to J3
JNE4:      IR=sr2, L=T, :JNeNb;                   returns to J4
JNE5:      IR=sr3, L=T, :JNeNb;                   returns to J5
JNE6:      IR=sr4, L=T, :JNeNb;                   returns to J6
JNE7:      IR=sr5, L=T, :JNeNb;                   returns to J7
JNE8:      IR=sr6, L=T, :JNeNb;                   returns to J8
JNE9:      IR=sr7, L=T, :JNeNb;                   returns to J9

; JNEB - if TOS (popped) = TOS (popped), jump PC-relative by alpha, assuming:
; stack has precisely two elements
; JNEB is A-aligned (also ensures no pending branch at entry)

JNEB:      IR=sr10, :JNeA;                      returns to JB

; JNE common code

JNeB:      temp=L RSH 1, L=0, :JEqNEcom;         temp:0, L:0
JNeA:      temp=L, L=0, :JEqNEcom;              temp:1, L:0

!1,2,JNeNe,JNeeq:
JNecom:    L=stkp-T-1, :JNeene;                 L: old stkp - 2
JNeene:    stkp=L, IDISP...=JEqNExxx;          jump, set stkp, then dispatch
JNeeq:     SINK=temp, BUS, TASK, :Setstkp;    no jump, reset stkp
JrB - for r in {L,LE,G,GE,UL,ULE,UG,UGE}
if TOS (popped) r TOS (popped), jump PC-relative by alpha, assuming:
stack has precisely two elements
JrB is A-aligned (also ensures no pending branch at entry)

The values loaded into IR are not returns but encoded actions:
Bit 12: 0 => branch if carry zero
1 => branch if carry one (mask value: 10)
Bit 15: 0 => perform add-complement before testing carry
1 => perform subtract before testing carry (mask value: 1)
(These values were chosen because of the masks available for use with +DISP
in the existing constants ROM. Note that IR= causes no dispatch.)

JLB: IR=10, Jscale; adc, branch if carry one
JLEB: IR=11, Jscale; sub, branch if carry one
JGB: IR=ONE, Jscale; sub, branch if carry zero
JGEB: IR=0, Jscale; adc, branch if carry zero
JULB: IR=10, Jnoscale; adc, branch if carry one
JULEB: IR=11, Jnoscale; sub, branch if carry one
JUGB: IR=ONE, Jnoscale; sub, branch if carry zero
JUGEB: IR=0, Jnoscale; adc, branch if carry zero

Comparison "subroutine":
!1,2,Jadc,Jsub;
!1,2,Jcz,Jco; appears above with JZEQB
!1,2,Jnobz,Jbz;
!1,2,Jbo,Jnobo;

Jscale: T=77777, Jadjust;
Jnoscale: T=ALLOanes, Jadjust;
Jadjust: L=stk1+T+1;
temp=L;
SINK=DISP, BUSODD;
T=stkO+T+1, Jadc;
Jadc: L=temp-T-1, Jcommon;
Jsub: L=temp-T, Jcommon;
Jcommon: T=ONE;
L=stkP-T-1, ALUCY;
SINK=DISP, SINK=1gm10, BUS=0, TASK, Jcz;
Jcz: stkP=L, Jnobz;
Jco: stkP=L, Jbo;
Jnobz: L=mpc+1, TASK, JnextAput;
Jbz: T=ib, Jbx;
Jbo: T=ib, Jbx;
Jnobo: L=mpc+1, TASK, JnextAput;

warning: not T=0+1
warning: not T=0+1
warning: not T=0+1
warning: not T=0+1
warning: not T=0+1
warning: not T=0+1
warning: not T=0+1
warning: not T=0+1
JIW - see Principles of Operation for description

assumes:

stack contains precisely two elements

if JIW is A-aligned, B byte is irrelevant

alpha in B byte, beta in A byte of word after JIW

!1,2,JIuge,Jlul;
!1,1,JIWx;

JIW: L=stkP-T-1, TASK, :JIWx;
    stkP=L;
    T=stkO;
    L=XMAR+mpc+1;
    mpc=L;
    L=stk1-T-1;
    ALUCY;
    T=MD, :JIuge;

JIWx: stkP=stkP-2
      load alphabeta
      do unsigned compare

JIuge: L=mpc+1, TASK, :nextAput;
      out of bounds - to 'nextA'
      (removing this TASK saves a word, but leaves a run of
       16 instructions)
      fetch <<cp>alphabeta+X>

Jlu1: L=cp+T, TASK;
      taskhole=L;
      T=taskhole;
      XMAR=stkO+T;
      NOP;
      L=MD-1, :Jbranch;
      L: offset
; Loads
;
; Note: These instructions keep track of their parity

; Lln - push <<lp>+n>
; Note: LL3 must be odd!

; Note: lp is offset by 2, hence the adjustments below

LL0:       MAR lp-T-1, :pushMD;
LL1:       MAR lp-1, :pushMD;
LL2:       MAR lp, :pushMD;
LL3:       MAR lp+T, :pushMD;
LL4:       MAR lp+T+1, :pushMD;
LL5:       T=3, SH=0, :LL3;       pick up ball 1
LL6:       T=4, SH=0, :LL3;       pick up ball 1
LL7:       T=5, SH=0, :LL3;       pick up ball 1

; LLB - push <<lp>+alpha>

LLB:       IR=sr4, :Getalpha;       returns to LLBr
LLBr:      T=lpoffset+T+1, SH=0, :LL3;       undiddle lp, pick up ball 1

; LLDB - push <<lp>+alpha>, push <<lp>+alpha+1>
; LLDB is A-aligned (also ensures no pending branch at entry)

LLDB:      T=lp, :LDcommon;
LDcommon:  T=lpoffset+T+1, :Dpush;
\texttt{LGN - push \langle gp \rangle + n>}

Note: \texttt{LG2} must be odd!

\texttt{Note: gp is offset by 1, hence the adjustments below}

\texttt{LGO: MAR\texttt{\textasciitilde}gp-1, \texttt{:pushMD};}
\texttt{LG1: MAR\texttt{\textasciitilde}gp, \texttt{:pushMD};}
\texttt{LG2: MAR\texttt{\textasciitilde}gp+1, \texttt{:pushMD};}
\texttt{LG3: MAR\texttt{\textasciitilde}gp+T+1, \texttt{:pushMD};}
\texttt{LG4: T=3, SH=0, \texttt{:LG2}; pick up ball 1}
\texttt{LG5: T=4, SH=0, \texttt{:LG2}; pick up ball 1}
\texttt{LG6: T=5, SH=0, \texttt{:LG2}; pick up ball 1}
\texttt{LG7: T=6, SH=0, \texttt{:LG2}; pick up ball 1}

\texttt{LGB - push \langle gp \rangle + \alpha>}

\texttt{LGB: IR\texttt{\textasciitilde}sr5, \texttt{:Getalpha; returns to LGBr}}
\texttt{LGBr: T=ngpoffset+T+1, SH=0, \texttt{:LG2; undiddle gp, pick up ball 1}}

\texttt{LGDB - push \langle gp \rangle + \alpha>, push \langle gp \rangle + \alpha+1>}

\texttt{LGDB is A-aligned (also ensures no pending branch at entry)}

\texttt{LGDB: T=gp+T+1, \texttt{:LDcommon;}}
\texttt{T: gp-gpoffset+lpoffset}
; LIN - push n
------------------------------------------------------------------
1,2,LI0xB,LI0xA; keep ball 1 in air
; Note: all BUS dispatches use old stkp value, not incremented one

LI0: L=stkp+1, BUS, :LI0xB;
LI1: L=stkp+1, BUS, :pushTIB;
LI2: T=2, :pushTB;
LI3: T=3, :pushTB;
LI4: T=4, :pushTB;
LI5: T=5, :pushTB;
LI6: T=6, :pushTB;

LI0xB: stkp=L, L=0, TASK, :push0;
LI0xA: stkp=L, BUS=0, L=0, TASK, :push0; BUS=0 keeps branch pending

; LIN1 - push -1
------------------------------------------------------------------
LIN1: T=ALLONES, :pushTB;

; LIN1 - push 100000
------------------------------------------------------------------
LINI: T=100000, :pushTB;

; LIB - push alpha
------------------------------------------------------------------
LIB: IR=sr2, :Getalpha; returns to pushTB
     Note: pushTIB will handle any pending branch

; LINB - push (alpha OR 377B8)
------------------------------------------------------------------
LINB: IR=sr26, :Getalpha;
LINBr: T=177400 OR T, :pushTB;

; LIW - push alphabeta, assuming:
; if LIW is A-aligned, B byte is irrelevant
; alpha in B byte, beta in A byte of word after LIW
------------------------------------------------------------------
LIW: IR=msr0, :FetchAB;
LIWr: L=stkp+1, BUS, :pushTIA;

returns to LIWr duplicates pushTA, but because of overlapping return points, we can't use it
; Stores

; SLn - <<lp>n>TOS (popped)
; Note: SL3 is odd!

; Note: lp is offset by 2, hence the adjustments below
SL0:    MAR=lp-T-1, :StoreB;
SL1:    MAR=lp-1, :StoreB;
SL2:    MAR=lp, :StoreB;
SL3:    MAR=lp+T, :StoreB;
SL4:    MAR=lp+T+1, :StoreB;
SL5:    T=3, SH=0, :SL3;
SL6:    T=4, SH=0, :SL3;
SL7:    T=5, SH=0, :SL3;

; SLB - <<lp>alpha>TOS (popped)

SLB:    IR=sr6, :Getalpha; returns to SLBr
SLBr:   T=nlpoffset+T+1, SH=0, :SL3; undiddle lp, pick up ball 1

; SLDB - <<lp>alpha+1>TOS (popped), <<lp>alpha>TOS (popped), assuming:
; SLDB is A-aligned (also ensures no pending branch at entry)
SLDB:   T=lp, :SDcommon;
SDcommon: T=nlpoffset+T+1, :Dpop;
; Note: gp is offset by 1, hence the adjustments below

SG0: \[ \text{MAR}\{\text{gp}-1\}, : \text{StoreB}; \]
SG1: \[ \text{MAR}\{\text{gp}\}, : \text{StoreB}; \]
SG2: \[ \text{MAR}\{\text{gp}+1\}, : \text{StoreB}; \]
SG3: \[ \text{MAR}\{\text{gp}+T+1\}, : \text{StoreB}; \]

SGB: \[ \text{IR}=\text{sr7}, \text{:Getalpha}; \]
SGBr: \[ \text{T}=\text{ngpoffset}+T+1, \text{SH}=0, :\text{SG2}; \text{undiddle gp, pick up ball 1} \]

SGDB: \[ \text{T}=\text{gp}+T+1, : \text{SDcommon}; \text{T}: \text{gp-gpoffset}+l\text{poffset} \]
Put s

Put s

PLn - <<lp>+n>+TOS (stack is not popped)

!1,1,PLcommon; drop ball 1

Note: lp is offset by 2, hence the adjustments below

PLO: MAR=lp-T-1, SH=0, :PLcommon; pick up ball 1
PL1: MAR=lp-1, SH=0, :PLcommon;
PL2: MAR=lp, SH=0, :PLcommon;
PL3: MAR=lp+T, SH=0, :PLcommon;

PLcommon: L=stkP, BUS, :Store8a; don't decrement stkP
Warning! Before altering this list, be certain you understand the additional addressing requirements imposed on some of these return locations! However, it is safe to add new return points at the end of the list.

; Binary operations

Both IR and T hold return number. (More precisely, entry at 'BincomB' requires return number in IR, entry at 'BincomA' requires return number in T.)

Exit conditions:
left operand in L (M), right operand in T
stkp positioned for subsequent push (i.e., points at left operand)
dispatch pending (for pushO) on return
if entry occurred at BincomA, IR has been modified so that mACSOURCE will produce 1

; Binary operations common code

Entry conditions:
Both IR and T hold return number.

Exit conditions:
Left operand in L (M), right operand in T
stkp positioned for subsequent push (i.e., points at left operand)
Dispatch pending (for pushO) on return
If entry occurred at BincomA, IR has been modified so that mACSOURCE will produce 1

; dispatches on stkp-1, so Binpop1 = 1 mod 208

; shake IR- in BincomA

BincomB: L=T=stkp-1, :Bincomx;
Bincomx: stkp=L, L=T;
Bincomd: temp2=L, :Binpop;
BincomA: L=2000 OR T;
Binpop: IR=M, :BincomB;
Binpop1: T=stk1;
Binpop2: T=stk2;
Binpop3: T=stk3;
Binpop4: L=stk4;
Binpop5: L=stk5;
Binpop6: L=stk6;
Binpop7: L=stk7;
Binend: SINK=DISP, BUS;

; value for dispatch into Binpop
; L: value for push dispatch
; stash briefly
; make mACSOURCE produce 1
; perform return dispatch
; perform push dispatch
ADD; replace <TOS> with sum of top two stack elements

ADD: IR=T=ret0, :BincomB;
ADDr: L=M+T, mACSOURCE, TASK, :push0; M addressing unaffected

ADD01; replace stk0 with <stk0>+<stk1>

!1,1,ADD01x;
ADD01: T=stk1-1, :ADD01x;
ADD01x: T=stk0+T+1, SH=0;
L=stk0-1, :pushT1B;

SUB; replace <TOS> with difference of top two stack elements

SUB: IR=T=ret1, :BincomB;
SUBr: L=M-T, mACSOURCE, TASK, :push0; M addressing unaffected

AND; replace <TOS> with AND of top two stack elements

AND: IR=T=ret2, :BincomB;
ANDr: L=M AND T, mACSOURCE, TASK, :push0; M addressing unaffected

OR; replace <TOS> with OR of top two stack elements

OR: IR=T=ret3, :BincomB;
ORr: L=M OR T, mACSOURCE, TASK, :push0; M addressing unaffected

XOR; replace <TOS> with XOR of top two stack elements

XOR: IR=T=ret4, :BincomB;
XORr: L=M XOR T, mACSOURCE, TASK, :push0; M addressing unaffected
Mesac-ROM.mu 24-Jul-81 19:03:01

---

; MUL - replace <TOS> with product of top two stack elements
; high-order bits of product recoverable by PUSH

; 7.1. MULDIVcoma:
; 1.2. GoROMMUL, GoROMDIV:
; 7.2. MULx.DIVx:

MUL:      IR=T-ret5, :BincomB;                     shakes stack dispatch
MULr:     AC1=L, L=T, :MULDIVcoma;               stash multiplicand
MULDIVcoma:  AC2=L, L=0, :MULx;                  stash multiplier or divisor
MULx:     AC0=L, T=0, :MULDIVcomb;               AC0=0 keeps ROM happy
DIVx:     AC0=L, T=0+1, BUS=0, :MULDIVcomb;     BUS=0  => GoROMDIV
MULDIVcomb:  L=MULDIVretloc=T-1, SWMODE, :GoROMMUL;  prepare return address
GoROMMUL:  PC=L, :ROMMUL;                      go to ROM multiply
GoROMDIV:  PC=L, :ROMDIV;                      go to ROM divide
MULDIVret:  :MULDIVret1;                      No divide - someday a trap
MULDIVret1:  T=AC1;
             L=stkpp+1;
             L=T, SINK=M, BUS;
             T=AC0, :dpush; Note! not a subroutine
                          call, but a direct
                          dispatch.

; DIV - push quotient of top two stack elements (popped)
; remainder recoverable by PUSH

;------------------------------------------------------------------
;------------------------------------------------------------------

DIV:      IR=T-ret6, :BincomB;                     BUS=0  => DIVx
DIVr:     AC1=L, L=T, BUS=0, :MULDIVcoma;

;------------------------------------------------------------------

; LDIV - push quotient of <TOS-1>,<TOS-2>/<TOS> (all popped)
; remainder recoverable by PUSH

;------------------------------------------------------------------

LDIV:     IR=sr27, :Popsub;  get divisor
LDIVf:    AC2=L;
           IR=T-ret7, :BincomB;  stash it
LDIVr:    AC1=L, L=T, IR=0, :DIVx;  L:low bits, T:high bits
                              stash low part of dividend
                              and ensure mACSOURCE of 0.
SHIFT - replace <TOS> with <TOS-1> shifted by <TOS>
;  <TOS> > 0 = left shift, <TOS> < 0 = right shift

!7,1,SHIFTx;
!1,2,Lshift,Rshift;
!1,2,DoShift,Shiftdone;
!1,2,DoRight,DoLeft;
!1,1,Shiftdonex;

SHIFT:       IR=T=ret10, :BincomB;
 SHIFTx:     temp=L, L=T, TASK, :SHIFTx;
    count=L;
     L=T=count;
      L=0-T, SH<0;
         IR=srl, :Lshift;

Lshift:     L=37 AND T, TASK, :Shiftcom;

Rshift:     T=37, IR=37;
    L=M AND T, TASK, :Shiftcom;

Shiftcom:   count=L, :Shiftloop;

Shiftloop:  L=count-1, BUS=0;
    count=L, IDISP, :DoShift;

DoShift:    L=temp, TASK, :DoRight;

DoRight:    temp=L RSH 1, :Shiftloop;

DoLeft:     temp=L LSH 1, :Shiftloop;

Shiftdone:  SINK=temp2, BUS, :Shiftdonex;

Shiftdonex: L=temp, TASK, :push0;

shakes stack dispatch
L: value, T: count
L: -count, T: count
IR= causes no branch
mask to reasonable size
equivalent to IR=msr0
mask to reasonable size
test for completion
dispatch to push result
Double-Precision Arithmetic

DADD - add two double-word quantities, assuming:
stack contains precisely 4 elements
1. L+4, SWMODE, :DoRamDoubles; shake B/A dispatch
DADD: L+4, SWMODE, :DoRamDoubles; drop ball 1
DoRamDoubles: SINK=M, BUS, TASK, :ramOverflow; go to overflow code in RAM

DSUB - subtract two double-word quantities, assuming:
stack contains precisely 4 elements
DSUB: L+5, SWMODE, :DoRamDoubles; drop ball 1

DCOMP - compare two long integers, assuming:
stack contains precisely 4 elements
result left on stack is -1, 0, or +1 (single-precision)
(i.e. result = sign(stk1, stk0 DSUB stk3, stk2))
DCOMP: L+6, SWMODE, :DoRamDoubles; drop ball 1

DUCOMP - compare two long cardinals, assuming:
stack contains precisely 4 elements
result left on stack is -1, 0, or +1 (single-precision)
(i.e. result = sign(stk1, stk0 DSUB stk3, stk2))
DUCOMP: L+7, SWMODE, :DoRamDoubles; drop ball 1
; Range Checking

; NILCK - check TOS for NIL (0), trap if so

!1,2.InRange,OutOfRange;

NILCK: L=ret17, :Xpopsub;
NILCKr: T=ONE, SH=0, :NILCKpush;
NILCKpush: L=stk+T, :InRange;
InRange: SINK+ib, BUS=0, TASK, :Setstkp;
OutOfRange: T=sBoundsFaultm1+T+1, :KFCr;

; BNDCK - check subrange inclusion
; if TOS-1 -IN [0..TOS) then trap (test is unsigned)
; only TOS is popped off

!7,1,BNDCKx;

BNDCK: IR=T=ret22, :BincomB;
BNDCKr: L=M-T, :BNDCKx;
BNDCKx: T=0, ALUCY, :NILCKpush;
; Reads

; Note: RBr must be odd!

; Rn - TOS<<TOS>>n

R0: T=0, SH=0, :RBr;
R1: T=ONE, SH=0, :RBr;
R2: T=2, SH=0, :RBr;
R3: T=3, SH=0, :RBr;
R4: T=4, SH=0, :RBr;

; RB - TOS<<TOS>>alpha>, assuming:

; !1,2,ReadB,ReadA; keep ball 1 in air

RB: IR=sr15, :Getalpha;
RBr: L=stkp-1, BUS, :ReadB;
ReadB: stkp=L, :MAStkT;
ReadA: stkp=L, BUS=O, :MAStkT;

; RDB - temp<<TOS>>+alpha, push <<temp>>, push <<temp>+1>, assuming:
; RDB is A-aligned (also ensures no pending branch at entry)

RDB: IR=sr30, :Popsub;

; RDO - temp<<TOS>>, push <<temp>>, push <<temp>+1>

RDO: IR=sr32, :Popsub;
RDOr: L=0, :Opusha;
\begin{verbatim}
; RILP  - push <<ip>+alpha[0-3]+alpha[4-7]>
RILP:    L=ret0, :Splitalpha;  get two 4-bit values
RILPr:   T=lp, :RIPcom;       T:address of local 2

; RIGP - push <<gp>+alpha[0-3]+alpha[4-7]>
RIGP:    L=ret1, :Splitalpha;  get two 4-bit values
RIGPr:   T=gp+1, :RIPcom;     T:address of global 2

RIPcom:  IR=msr0, :IPcom;     set up return to pushMD
IPcom:   T=-3+T+1;           T:address of local or global 0
         MAR=lethalf+T;      start memory cycle
         L=righthalf;
IPcomx:  T=MD, IDISP;        T:local/global value
         MAR=M+T, :pushMD;

; RIL0 - push <<<lp>>>>
RIL0:    MAR=lp-T-1, :RILxB;  fetch local 0
RILxB:   IR=msr0, L+0, :IPcomx;
RILxA:   IR=sr1, L=sr1 AND T, :IPcomx;
         to pushMD
         to pushMDA, L+0(!)

; RXLP - TOS<<TOS>+<<ip>+alpha[0-3]+alpha[4-7]>
RXLP:    L=ret3, :Splitalpha;  will return to RXLPra
RXLPra:  IR=sr34, :Popsub;   fetch TOS
RXLPrb:  L=righthalf+T, TASK; L: TOS+alpha[4-7]
         righthalf=L, :RILPr;  now act like RILP
\end{verbatim}
; writes
;-------------
; Wn - <<TOS> (popped)+n><-TOS> (popped)
; 1,2,WnB,WnA;  keep ball 1 in air
WO:  T+0, :WnB;
W1:  T+ONE, :WnB;
W2:  T+2, :WnB;
WnB:  IR=sr2, :Wsub;
WnA:  IR=sr3, :Wsub;

; returns to StoreB
; returns to StoreA

; Write subroutine:
;-------------
; 1,1,Wsubx;
Wsub:  L=stkp-1, BUS, :Wsubx;
Wsubx:  stkp=L, IDISP, :MAStkT;

; WB - <<TOS> (popped)+alpha>-<TOS-1> (popped)
;-------------
WB:  IR=sr16, :Getalpha;
WBr:  :WnB;

; returns to WBr
branch may be pending

; WSB - act like WB but with stack values reversed, assuming:
; WSB is A-aligned (also ensures no pending branch at entry)
;-------------
!7,1,WSBx;
WSB:  IR=T+ret14, :BincomA;
WSBr:  T=M, L=T, :WSBx;
WSBx:  MAR=ib+T, :WScom;
WScom:  temp=L;
WScoma:  L=stkp-1;
        MD=temp;
        MACSOURCE, TASK, :Setstkp;

; WSO - act like WSB but with alpha value of zero
;-------------
!7,1,WSOx;
WSO:  IR=T+ret15, :BincomB;
WSOr:  T=M, L=T, :WSOx;
WSOx:  MAR=T, :WScom;
; WILP - \((l+p)+\alpha[0-3]+\alpha[4-7] - \langle\text{TOS}\rangle\) (popped)

WILP:  
L=ret2; :Splitalpha;

get halves of alpha

WILPr:  
IR=sr2;
T+lp; :IPcom;

IPcom will exit to StoreB
prepare to undiddle

; WXLP - \(\langle\text{TOS}\rangle + (l+p)+\alpha[0-3]+\alpha[4-7] + \langle\text{TOS}-1\rangle\) (both popped)

WXLP:  
L=ret4; :Splitalpha;

get halves of alpha

WXLPra:  
IR=sr35; :Popsub;

fetch TOS

WXLPrb:  
L=righthalf+T, TASK;

L:TOS+alpha[4-7]
now act like WILP

WILPr;  
IR=sr2; :IPcom;

IPcom will exit to StoreB
prepare to undiddle

; WXLPra:  
L=righthalf+T, TASK;

WILPr;

get halves of alpha

; WXLPrb:  
L=righthalf+T, TASK;

L:TOS+alpha[4-7]
now act like WILP

WXLPx:  
IR=sr2; :IPcom;

get halves of alpha

WILPr;

fetch TOS

L:TOS+alpha[4-7]
now act like WILP

; WXLPx:  
IR=sr2; :IPcom;

get halves of alpha

WILPr;

fetch TOS

L:TOS+alpha[4-7]
now act like WILP

; WXLPx:  
IR=sr2; :IPcom;

get halves of alpha

WILPr;

fetch TOS

L:TOS+alpha[4-7]
now act like WILP

; WXLPx:  
IR=sr2; :IPcom;

get halves of alpha

WILPr;

fetch TOS

L:TOS+alpha[4-7]
now act like WILP

; WDB - temp+alpha+(\langle TOS\rangle \) (popped), pop into \langle\text{temp}\rangle+1 and \langle\text{temp}\rangle, assuming:
; WDB is A-aligned (also ensures no pending branch at entry)

WDB:  
IR=sr31; :Popsub;
returns to Dpop

; WDO - temp+(\langle TOS\rangle \) (popped), pop into \langle\text{temp}\rangle+1 and \langle\text{temp}\rangle

WDO:  
L=ret6, TASK, :Xpopsub;
returns to WDOr

WDOr:  
L+0, :Dpopa;

; WSOB - like WDB but with address below data words, assuming:
; WSOB is A-aligned (also ensures no pending branch at entry)

!7,1,WSOBx;

WSOB:  
IR=sr24, :Popsub;
get low data word

WSOBra:  
saveret=L;
stash it briefly

IR=T=ret20, :BincomA;
alignment requires BincomA

WSOBrb:  
T=M, L+T, :WSOBx;
start store of low data word

WSOBx:  
MAR=T+ib+T+1;
temp=L, L+T;
temp2=L, TASK;
temp:high data

MAR=temp2-1, :WScoma;
temp2:updated address

get low data word

stash it briefly

alignment requires BincomA

start store of low data word

start store of high data word
; Long Pointer operations

!1,1,RWBLcom:

; RBL - like RB, but uses a long pointer

RBL:
   L=M AND NOT T, T=M, SH=0, :RWBLcom;
   L: ret0, T: L at entry

; WBL - like WB, but uses a long pointer

WBL:
   L=T, T=M, SH=0, :RWBLcom;
   L: ret1, T: L at entry

; Common long pointer code

!1,2,RWBLcomB,RWBLcomA;
!1,1,RWBLxa;
!1,1,RWBLxb;
!7,1,WBLx;
!3,4,RBLra,WBLra,WBLrc,;
!3,4,RWBLdone,RBLdone, ,WBLdone;

RWBLcom:
   entry=L, L=T, :RWBLcomB;
   stash return, restore L

RWBLcomB:
   IR=sr37, :Getalpha;

RWBLcomA:
   IR=sr37, :GetalphaA;

RWBLra:
   IR=ret23, L=T, :RWBLxa;
   L: alpha byte

RWBLxa:
   alpha=L, :BincomB;
   stash alpha, get long pointer

RWBLrb:
   MAR=BankReg, :RWBLxb;
   fetch bank register

RWBLxb:
   L=T, T=M;
   temp=L;
   L=alpha+T;
   T=MD;
   temp=temp, unsure
   frame=L, L=T;
   taskhole=L, TASK;
   reaccess bank register
   MD=temp;
   WBLx;

WBLx:
   XMAR=frame;
   L=entry+1, BUS;
   start memory access
   entry=L, L=T, :RWBLcomB;
   dispatch RBL/WBL
   (L=T for WBLrc only)

RWBLtail:
   T=MD, :RWBLtail;
   T: data from memory
   returns to RWBLrb

WBLra:
   IR=ret24, :BincomB;
   T: data to write

WBLrc:
   MD=M, :RWBLtail;
   stash data in memory

RWBLLdone:
   MAR=BankReg;
   SINK=entry, BUS;
   dispatch return

RWBLdone:
   MD=taskhole, :RWBLdone;
   restore bank register

RBLdone:
   L=temp2+1, BUS, :pushT1B;
   temp2: original stk=2

WBLdone:
   L=temp2, TASK, :SetstkP;
   temp2: original stk=3
Unary operations

XMESA Note: Untail is wired down by a pre-def in MesaROM.mu

INC - TOS + <TOS>+1

INC: IR=sr14, :Popsub;
INCr: T=O+T+1, :pushTB;

NEG - TOS + -<TOS>

NEG: L=ret11, TASK, :Xpopsub;
NEGr: L=O-T, :Untail;

DBL - TOS + 2*<TOS>

DBL: IR=sr25, :Popsub;
DBLr: L=M+T, :Untail;

Unary operation common code

Untail: T=M, :pushTB;
; Stack and Miscellaneous Operations

; PUSH - add 1 to stack pointer

!1,1,PUSHx;

PUSH: L=stk+1, BUS, :PUSHx;
PUSHx: SINK=ib, BUS=0, TASK, :Setstk;
BUS checks for overflow
pick up ball 1

; POP - subtract 1 from stack pointer

POP: L=stk-1, SH=0, TASK, :Setstk;
L=0 <-> branch 1 pending
need not check stk=0

; DUP - temp<TOS> (popped), push <temp>, push <temp>

!1,1,DUPx;

DUP: IR=sr2, :DUPx;
DUPx: L=stk, BUS, TASK, :Popsuba;
returns to pushTB
don't pop stack

; EXCH - exchange top two stack elements

!1,1,EXCHx;

EXCH: IR=ret1, :EXCHx;
EXCHx: L=stk-1;
L=M+1, BUS, TASK, :Bincomd;
EXCHr: T=M, L=T, :dpush;
drop ball 1
dispatch on stk-1
set temp2=stk
Note: dispatch using temp2

; LADRB - push alpha+lp (undiddled)

!1,1,LADRBx;

LADRB: IR=sr10, :Getalpha;
LADRBx: T=lp+offset+T+1, :LADRBx;
LADRBx: L=lp+T, :Untail;
shake branch from Getalpha
returns to LADRBx

; GADRB - push alpha+gp (undiddled)

!1,1,GADRBx;

GADRB: IR=sr10, :Getalpha;
GADRBx: T=gp+offset+T+1, :GADRBx;
GADRBx: L=gp+T, :Untail;
shake branch from Getalpha
returns to GADRBx
String Operations

!7,1,STRsub; shake stack dispatch
!1,2,STRsubA,STRsubB;
!1,2,RSTRrx,WSTRrx;

STRsub: L+stk-1;
        stk+L;
        L+1b+T;
        SINK=M, BUSODD, TASK;
        count=L RSH 1, :STRsubA;

STRsubA: L+177400, :STRsubcom;
STRsubB: L+377, :STRsubcom;

STRsubcom: T-temp;
           MAR=count+T;
           T+M;
           SINK=DISP, BUSODD;
           mask=L, SH<0, :RSTRrx;

; RSTR - push byte of string using base (<TOS-1>) and index (<TOS>);
; assumes RSTR is A-aligned (no pending branch at entry)

!1,2,RSTRB,RSTRA;

RSTR: IR=T-ret12, :BincomB;
RSTRr: temp=L, :STRsub;
RSTRrx: L=MD AND T, TASK, :RSTRB;

RSTRB: temp=L, :RSTRcom;
RSTRA: temp=L LCY 8, :RSTRcom;

RSTRcom: T-temp, :pushTA;

; WSTR - pop <TOS-2> into string byte using base (<TOS-1>) and index (<TOS>);
; assumes WSTR is A-aligned (no pending branch at entry)

!1,2,WSTRB,WSTRA;

WSTR: IR=T-ret13, :BincomB;
WSTRAr: temp=L, :STRsub;
WSTRAx: L=MD AND NOT T, :WSTRB;

WSTRB: temp2=L, L-ret0, TASK, :xpopsub;
WSTRA: temp2=L, L-ret0+1, TASK, :xpopsub;

WSTRArA: taskhole=L LCY 8;
WSTRA: T-taskhole, :WSTRA;

WSTRB: T=mask.T;
       L-temp2 OR T;
       T-temp;
       MAR=count+T;
       TASK;
       MD=M, :nextA;

; move new data to odd byte

; retrieve string address
; F i e l d I n s t r u c t i o n s

; temp2 is coded as follows:
; 0 - RF, RFS
; 1 - WF, WSF, WFS
; 2 - RFC

%1,2,RFr,WFrr;
%7,1,Fieldsub;
%7,1,WFr: (required by WSF) is implicit in ret17 (!)

; RF - push field specified by beta in word at <TOS> (popped) + alpha
; if RF is A-aligned, B byte is irrelevant
; alpha in B byte, beta in A byte of word after RF

RF: IR=sr12, :Popsub;
RFr: L=ret0, :Fieldsub;
RFrr: T=mask.T, :pushTA;

; WF - pop data in <TOS-1> into field specified by beta in word at <TOS> (popped) + alpha
; if WF is A-aligned, B byte is irrelevant
; alpha in B byte, beta in A byte of word after WF

WF: IR=T+ret17, :BincomB;
WFrr: newfield=L, L=ret0+1, :Fieldsub;

WFrr: T=mask;
L=M AND NOT T;
temp+L;
T=newfield.T;
L=mask OR T, TASK;
CYCOUT=L;
T=index, BUS=0;
L=WFretloc, :WFnzct;

WFnzct: PC=L;
L=20-T, SWMODE;
T=CYCOUT, :RAMCYC;

WFret: MAR=frame;
L=stk-1;
MD=CYCOUT, TASK, :JZNEBeq;

; WSF - like WF, but with top two stack elements reversed
; if WSF is A-aligned, B byte is irrelevant
; alpha in B byte, beta in A byte of word after WSF

WSF: IR=T+ret16, :BincomB;
WSFr: L=T, T=M, :WFr;

returns from Fieldsub
shakes stack dispatch

L:new data, T:address
(actually, L=ret1)

set old field bits to zero
stash result
save new field bits
merge old and new
stash briefly
get position, test for zero
get return address from ROM

stash return
L:remaining count to cycle
go cycle remaining amount
start memory
pop remaining word
stash data, go update stkp

L:address, T:new data
; RFS - like RF, but with a word containing alpha and beta on top of stack
; if RFS is A-aligned, B byte is irrelevant

RFS:                  L-ret12, TASK, :Xpopsub;                  get alpha and beta
                       temp+L;                             stash for WFSa
                       L-ret13, TASK, :Xpopsub;             T:address
RFSra:                L-ret0, BUS=0, :Fieldsub;               returns quickly to WFSa
RFSrb:                L-ret0, BUS=0, :Fieldsub;               

; WFS - like WF, but with a word containing alpha and beta on top of stack
; if WFS is A-aligned, B byte is irrelevant

!1,2, Fieldsuba, WFSa;

WFS:                  L-ret14, TASK, :Xpopsub;                  get alpha and beta
                       temp+L;                             stash temporarily
                       IR=T-ret21, :Bincom6;                L:new data, T:address
WFSra:                newfield=L, L-ret0+1, BUS=0, :Fieldsub;  returns quickly to WFSa
WFSrb:                frame=L;                                  stash address
t                        T=177400;                            to separate alpha and beta
WFSa:                 L-temp AND T, T-temp, :Getalphab;       L:alpha, T:both
                       L-ret14, TASK, :Xpopsub;                  returns to Fieldra

; RFC - like RF, but uses <cp>+<alpha>+<TOS> as address
; if RFC is A-aligned, B byte is irrelevant
; alpha in B byte, beta in A byte of word after RF

RFC:                  L-ret16, TASK, :Xpopsub;                  get index into code segment
                       L=cp+T;                             T:address
                       T=M;                               returns to RFrr
                       L-ret2, :Fieldsub;               


; Field instructions common code
; Entry conditions:
; L holds return offset
; T holds base address
; Exit conditions:
; mask: right-justified mask
; frame: updated address, including alpha
; index: left cycles needed to right-justify field [0-15]
; L,T: data word from location <frame> cycled left <index> bits

; 2,3,1,NotCodeSeg,IsCodeSeg:

Fieldsub:  temp2=L, L+T, IR=msr0, TASK, :Fieldsuba;  stash return
Fieldsuba:  frame=L, :GetAlphaA;  stash base address
Fieldra:  L+d5;  T: beta, ib: alpha
get two halves of beta
Fieldrb:  T=righthalf;
MAR=MASKTAB+T;
T=lefthalf+T+1;
L=17 AND T;
index=L;
L=MD, TASK;
mask=L;
SINK=temp2, BUS;
T=frame, :NotCodeSeg;
L=MAR=ib+T, :StashFieldLoc;
NotCodeSeg:  T=frame, :DoCycle;
IsCodeSeg:  XMAR=ib+T, :DoCycle;
StashFieldLoc:  frame=L, :DoCycle;
DoCycle:  L=Fieldretloc;
PC=L;
T=MD, SWMODE;
L=INDEX, :RAMCYCX;
Fieldrc:  SINK=temp2, BUSODD;
L=T=CYCOUT, :RFrr;  data word into T for cycle
count to cycle, go do it
cycled data word in L and T

 stash posit on
stash mask
stash mask
stash position
mask to 4 bits
start fetch of mask
index for MASKTAB
get base address
add alpha
add alpha
return dispatch
stash updated address for WF
return location from RAMCYCX
get two halves of beta
index for MASKTAB
stash return
stash base address
T: beta, ib: alpha
Frame Allocation

Alloc subroutine:
allocates a frame
Entry conditions:
frame size index (fsi) in T
Exit conditions:
frame pointer in L, T, and frame
if allocation fails, alternate return address is taken and
temp2 is shifted left by 1 (for ALLOC)

!1,2,ALLOCr,XferGr;
!1,2,ALLOCr,XferGrf;
!3,4,Alloc0,Alloc1,Alloc2,Alloc3;
if more than 2 callers, un-comment the following pre-definition:
!17,1,Allocx;

AllocSub:
L+avm1+T+1, TASK, :Allocx;
Allocx:
entry=L;
L+MAR=entry;
T=3;
L+MD AND T, T+MD;
temp=L, L+MAR=T;
SINK=temp, BUS;
frame=L, :Alloc0;

; Bits 14:15 = 00, a frame of the right index is queued for allocation

Alloc0:
L+MD, TASK;
temp=L;
MAR=entry;
L+T=frame, IDISP;
MD=temp, :ALLOCr;

; Bits 14:15 = 01, allocation list empty: restore argument, take failure return

Alloc1:
L+temp2, IDISP, TASK;
temp2=L RSH 1, :ALLOCr;

; Bits 14:15 = 10, a pointer to an alternate list to use

Alloc2:
temp=L RSH 1, :Allocp;
Allocp:
L+temp, TASK;
temp=L RSH 1;
T=temp, :AllocSub;

Alloc3:
temp=L RSH 1, :Allocp;
(treat type 3 as type 2)
Free subroutine:
  frees a frame
Entry conditions: address of frame is in 'frame'
Exit conditions: 'frame' left pointing at released frame (for LSTF)

FreeSub returns
shake IR= dispatch
start read of fsi word
wait for memory
T=index
fetch av entry
save av entry address
read current pointer
write it into current frame
write!
entry points at frame
free
ALLOC - allocate a frame whose fsi is specified by <TOS> (popped)

(Here so ALLOCrf can call it)

The following logically belongs here; however, because the entry point to general Xfer is known to the outside world, the real declaration appears in MesaROM.mu.

!7,10,XferGT,Xfer,Mstopr,PORTOpc,LSTr,ALLOCrfr,:

return points for Savpcinframe

!1,2,doAllocTrap,XferGrf; used by XferGrf

ALLOC:

ALLOCrx:

ALLOCr:

L-ret 7, TASK, :Xpopsub;

temp2=L LSH 1, IR=msrD, :AllocSub;

L-stkp+1, BUS, :pushTIB;

returns to ALLOCrx

L,T: fsi
duplicates pushTIB

Allocation failed - save mpc, undiddle lp, push fsi*4 on stack, then trap

ALLOCrf:

ALLOCrfr:

IR= sr5, :Savpcinframe;

L-temp2, TASK, :doAllocTrap;

failure because lists empty

pick up trap parameter

Inform software that allocation failed

doAllocTrap:

ATPreg=L;

T=AllocTrap, :Mtrap;

store param. to trap proc.

go trap to software

FREE - release the frame whose address is <TOS> (popped)

FREE:

FREErx:

FREEr:

L-ret 10, TASK, :Xpopsub;

frame=L, TASK;

IR= sr1, :FreeSub;

next;

returns to FREErx
; Descriptor Instructions

; DESCB - push <<gp>+gfi offset>+2*alpha+1 (masking gfi word appropriately)
; DESCB is assumed to be A-aligned (no pending branch at entry)

DESCB:

T=gp;
T=ngp+offset+T+1, :DESCBcom;

DESCBcom:

MAR=gfi+offset+T;
T=gfi;mask;
T=M.D.T;
L=ib+T, T=ib;
T=M+T+1, :pushTA;

; DESCBS - push <TOS>+gfi offset>+2*alpha+1 (masking gfi word appropriately)
; DESCBS is assumed to be A-aligned (no pending branch at entry)

DESCBS:

L=ret15, TASK, :Xpopsub;

returns to DESCBcom
Savpcinframe subroutine:

stashes C-relative (mpc, ib) in current local frame
undiddles lp into my and lp
Entry conditions: none
Exit conditions:
current frame+1 holds pc relative to code segment base (+ = even, - = odd)
lp is undiddled
my has undiddled lp (source link for Xfer)

Savpcinframe:
T+cp, :Savpcx;
L=mpc-T;
SINK=ib, BUS=0;
T+M, :Spcood;

Savpcx:
L=0-T, TASK, :Spcood;

Spcood:
L=0-T, TASK, :Spcood;

Spceven:
L=0+T+1, TASK, :Spcoopc;

Spcoopc:
taskhole=L;
L=0;
T=ncpcoffset;
MAR=lp-T, T=lp;
ib=L;
L=nlpcoffset+T+1;
MD=taskhole;
my=L, IDISP, TASK;
lp=L, :XferGT;

required by PORTO returns (appear with ALLOC)
shake IR dispatch pc odd or even

code segment base
L is code-relative pc
check for odd or even pc
pick up pc word addr

pc value to save
(can't merge above - TASK)
offset to pc stash
(MAR=lp-ncpcoffset, T=lp)
clear ib for XferG
L:undiddled lp
stash pc in frame+pcoffset
store undiddled lp
Loadgc subroutine:
load global pointer and code pointer given local pointer or GFT pointer

Entry conditions:
- T contains either local frame pointer or GFT pointer
- Memory fetch of T has been started
- Pending branch (1) catches zero pointer

Exit conditions:
- lp diddled (to framebase+6)
- mpc set from second word of entry (PC or EV offset)
- First word of code segment set to 1 (used by code swapper)

Assumes only 2 callers

Loadgc:  L=1poffset+T;
           lp=L;
           T=MD;
           L=MD;
           MAR=cpoffset+T;
           mpc=L, L=T;
           L=cpoffset+T, SH=0;
           taskhole=L, :LoadgcOK;

LoadgcOK:  L=MD, BUSODD, TASK;
            cp=L, :LoadgcIn;

LoadgcIn:  MAR=BankReg;
            T=taskhole+1;
            L=gp-T-1;
            T=T4, SH=0;
            L=MD AND T, :DoLoad;

DoLoad:  temp2=L, :LoadgcShift;

LoadgcShift:  newfield=L RSH 1, L=0-T, :LoadgcDiv2;

LoadgcDiv2:  L=newfield, SH<0, TASK, :LoadgcShift;

LoadgcDiv4:  L=gppoffset+T;
           gp=L;
           T=177400;
           L=MD AND T, T=MD;
           T=3.T, SH=0;
           MAR=BankReg, :LoadgcNoXM;

LoadgcNoXM:  T=newfield, :LoadgcIsXM;

LoadgcIsXM:  L=temp2 OR T, TASK;
             MD=M;

NoLoad:  XMAR=cp;
        IDISP, TASK;
         MD=ONE, :XferOr;

picked up global frame of zero somewhere, call it unbound

LoadgcNull:  T=sUnbound, :Stashmx;

LoadgcSwap:  T=sSwapTrap, :Stashmx;

LoadgcTrap:  T=sControlFault, :Mtrap;
CheckXferTrap subroutine:
Handles Xfer trapping

Entry conditions:
- IR: return number in DISP
- T: parameter to be passed to trap routine

Exit conditions:
- if trapping enabled, initiates trap and doesn't return.

-----------------------------------------------------------------

returns from CheckXferTrap

CheckXferTrap: L=XTSreg, BUSODD;
SINK=DISP, BUS, :NoXferTrap;

NoXferTrap: XTSreg=L RSH 1, :Xfers;
reset XTSreg[15] to 0 or 1

DoXferTrap: L=DISP, :DoXferTrapx;
tell trap handler which case
L: trap parameter

DoXferTrapx: XTSreg=L LCY 8, L+T;
off to trap sequence
XTPreg=L;
T=sxferTrap, :Mtrap;
Xfer open subroutine:
  decodes general destination link for Xfer
Entry conditions:
  source link in my
  destination link in mx
Exit conditions:
  if destination is frame pointer, does complete xfer and exits to Ifetch.
  if destination is procedure descriptor, locates global frame and entry
number, then exits to 'XferG'.

\[3,4,\text{Xfer0}, \text{Xfer1}, \text{Xfer2}, \text{Xfer3};\]

Xfer:
  \(T=mx;\)
  \(\text{IR}=0, \text{CheckXferTrap};\)
Xfers:
  \(L=3 \text{ AND } T;\)
  \(\text{SINK}=M, L=T, \text{BUS};\)
  \(\text{SH}=0, \text{MAR}=T, \text{Xfer0};\)

\(mx[14\text{ - }15] = 00\)
  Destination link is frame pointer

Xfer0:
  \(\text{IR}=\text{msr0}, \text{XferO};\)
Xfer0r:
  \(L=\text{mpc};\)

If 'brkbyte' \(!= 0\), we are proceeding from a breakpoint.
  pc points to the BRK instruction:
  even pc \(\Rightarrow\) fetch word, stash left byte in ib, and execute brkbyte
  odd pc \(\Rightarrow\) clear ib, execute brkbyte

\[11,2,\text{Xdobreak}, \text{Xnobreak};\]
\[11,2,\text{Xfer0B}, \text{Xfer0A};\]
\[11,2,\text{XbrkB}, \text{XbrkA};\]
\[11,2,\text{XbrkBgo}, \text{XbrkAgo};\]

\(\text{SINK}=\text{brkbyte}, \text{BUS}=0;\)
  set up by LoadgCNull if dest link = 0
\(\text{SH}<0, L=0, \text{Xdobreak};\)
  offset from cp: - odd, + even

Not proceeding from a breakpoint - simply pick up next instruction

Xnobreak:
  \(\text{:Xfer0B};\)
Xfer0B:
  \(L=\text{XMAR}=cp+T, \text{XferOA};\)
Xfer0A:
  \(L=\text{XMAR}=cp-T;\)
  \(\text{mpc}=L, \text{XferOB};\)

Proceeding from a breakpoint - dispatch brkbyte and clear it

Xdobreak:
  \(ib=L, \text{XbrkB};\)
XbrkB:
  \(\text{IR}=\text{sr20};\)
  \(L=\text{XMAR}=cp+T, \text{XbrkA};\)
XbrkA:
  \(L=cp-T;\)
  \(\text{mpc}=L, L=0, \text{BUS}=0, \text{XbrkB};\)
XbrkBgo:
  \(\text{SINK}=\text{brkbyte}, \text{BUS}, \text{XbrkAgo};\)
XbrkAgo:
  \(\text{brkbyte}=L, RSH=1, T=0+1, \text{NOOP};\)
  \(\text{brkbyte}=L, T=0+1, \text{BUS}=0, \text{NOOP};\)
\textbf{Destination link is procedure descriptor:}
\begin{itemize}
\item [$\text{mx[0-8]}$]: GFT index ($gfi$)
\item [$\text{mx[9-13]}$]: EV bias, or entry number ($en$)
\end{itemize}

\textbf{Xfer1:}
\begin{itemize}
\item $\text{temp} = \text{RSH} 1$
\item $\text{count} = \text{MLSH} 1$
\item $\text{L} = \text{count, TASK}$
\item $\text{count} = \text{LCY} 8$
\item $\text{L} = \text{count, TASK}$
\item $\text{L} = \text{L} \text{LSH} 1$
\item $\text{T} = \text{count}$
\item $\text{T} = \text{T} \text{LSH} 1$
\item $\text{MAR} = \text{gfmt} + \text{T} + 1$
\item $\text{IR} = \text{sr1}, : \text{Loadgc}$
\end{itemize}

\textbf{Xfer1r:}
\begin{itemize}
\item $\text{L} = \text{temp, TASK}$
\item $\text{count} = \text{L} \text{LSH} 1$
\item $\text{T} = \text{count}$
\item $\text{T} = \text{enmask} + \text{T}$
\item $\text{L} = \text{mpc} + \text{T} + 1, \text{TASK}$
\item $\text{count} = \text{L} \text{LSH} 1, : \text{XferG}$
\end{itemize}

\textbf{Xfer2:}
\begin{itemize}
\item $\text{NOP}$
\item $\text{T} = \text{MD, : Xfers}$
\end{itemize}

\textbf{Xfer3:}
\begin{itemize}
\item $\text{T} = \text{sunbound, : Stashmx}$
\end{itemize}
XferG open subroutine:
allocates new frame and patches links

Entry conditions:
- 'count' holds index into code segment entry vector
- assumes lp is undiddled (in case of AllocTrap)
- assumes gp (undiddled) and cp set up

Exit conditions:
- exits to instruction fetch (or AllocTrap)

---

Pick up new pc from specified entry in entry vector

```
T<count;
IR<ONE, :CheckXferTrap;
```

```
T<count;
XMAR=cp+T;
T=cp-1;
```

```
IR=sr1;
L=MD+1;
T=MD;
mpc=L;
T=377.T,:AllocSub;
```

---

Stash source link in new frame, establishing dynamic link

```
XferGr: MAR=retlinkoffset+T;
L=1poffset+T;
lp=L;
MD=my;
```

---

Stash new global pointer in new frame (same for local call)

```
MAR=T;
T=gpoffset;
L=gp-T, TASK;
MD=M, :nextAdead;
```

---

Frame allocation failed - push destination link, then trap

```
L=mx, BUS=0;
T<count-1,:doAllocTrap;
```

if destination link is zero (i.e. local procedure call), we must first fabricate the destination link

```
XferGfz: L=T, T-ngfioffset;
MAR=gp-T;
count=L LSH 1;
L=count-1;
T=gfimask;
T=MD.T;
L=M+T,:doAllocTrap;
```

---

Parameter to CheckXferTrap
index into entry vector
fetch of new pc and fsi
point just before bytes
(main loop increments mpc)
note: does not cause branch
relocate pc from cseg base
second word contains fsi
new pc setup, ib already 0
mask for size index

T has new frame base
diddle new lp
install diddled lp
source link to new frame

write gp to word 0 of frame
offset to point at gf base
subtract off offset
global pointer stashed, GO!

(appears with ALLOC)

pick up destination, test = 0
T:2*ep+1

offset from gp to gfi word
start fetch of gfi word
count:4*ep+2
L:4*ep+1
mask to save gfi only
T:gfi
L:gfi+4*ep+1 (descriptor)
Getlink subroutine:
fetches control link from either global frame or code segment

Entry conditions:
- temp: - (index of desired link + 1)
- IR: DISP field zero/non-zero to select return point (2 callers only)

Exit conditions:
- L,T: desired control link

Getlink:
!1.2,EFCgetr,LLKBr;
!1.2,framelink,codelink;
!7.1,Fetchlink:

return points
shake IR~ in KFCB

diddled frame address
fetch word 0 of global frame
L:address of link in frame
stash it
L:address of link in code
test bit 15 of word zero
stash code link address

fgetlink: T=gp;
MAR=T-ngpoffset+T+1;
L=temp+T, T=temp;
taskhole=L;
L=cp+T;
SINK=MD, BUSODD, TASK;
temp2=L, :framelink;

fgetlink: MAR=taskhole, :Fetchlink;
XMAR=temp2, :Fetchlink;

Fetchlink: SINK=DISP, BUS=0;
L=T+MD, :EFCgetr;
dispatch to caller
\textbf{EFCn} - perform XFER to destination specified by external link n
\textbf{EFCr}; implicit in EFCr's return number (23B)

\begin{verbatim}
EFC0:   IR=ONE, T=ONE-1, :EFCr;           0th control link
EFC1:   IR=T=ONE, :EFCr;                   1st control link
EFC2:   IR=T+2, :EFCr;
EFC3:   IR=T+3, :EFCr;
EFC4:   IR=T+4, :EFCr;
EFC5:   IR=T+5, :EFCr;
EFC6:   IR=T+6, :EFCr;
EFC7:   IR=T+7, :EFCr;
EFC8:   IR=T+10, :EFCr;
EFC9:   IR=T+11, :EFCr;
EFC10:  IR=T+12, :EFCr;
EFC11:  IR=T+13, :EFCr;
EFC12:  IR=T+14, :EFCr;
EFC13:  IR=T+15, :EFCr;
EFC14:  IR=T+16, :EFCr;
EFC15:  IR=T+17, :EFCr;
\end{verbatim}

\textbf{EFCB} - perform XFER to destination specified by external link 'alpha'

\begin{verbatim}
!1,1,EFCdoGetlink;
\end{verbatim}

\begin{verbatim}
EFCB:   IR=sr23, :Getalpha;                shake B/A dispatch (Getalpha)
EFCr:   L=O-T-1, TASK, :EFCdoGetlink;      fetch link number
        L:=(link number+1)
EFCdoGetlink: temp=L, :Getlink;           stash index for Getlink
        IR=srl, :SFCr;                        for Savpcinframe; no branch
EFCgetr: IR=srl, :SFCr;
\end{verbatim}

\textbf{SFC} - Stack Function Call (using descriptor on top of stack)

\begin{verbatim}
SFC:   IR=srl, :Popsub;                    get dest link for xfer
       mx=L, :Savpcinframe;                 now assume IR still has srl
SFCr:  IR=srl, :Savpcinframe;              set dest link, return to xfer
\end{verbatim}
KFCB - Xfer using destination <<SD>+alpha>

: !1,1,KFCr; implicit in KFCr's return number (21B)
: !1,1,KFCx; shake B/A dispatch (Getalpha)
: !7,1,Fetchlink; appears with Getlink

KFCB: IR<-sr21, :Getalpha;
KFCr: IR<-avm1, T<-avm1+T+1, :KFCx;
KFCx: MAR<-sloffset+T, :Fetchlink;

!1,1,KFCr; implicit in KFCr's return number (21B)

!7,1,Fetchlink; appears with Getlink

fetch alpha

DISP must be non zero

Fetchlink shakes IR+ dispatch

BRK - Breakpoint (equivalent to KFC 0)

BRK: ib=L, T<-sBRK, :KFCr;

ib = 0 <=> BRK B-aligned

Trap sequence:
used to report various faults during Xfer
Entry conditions:
T: index in SD through which to trap
Savepcinframe has already been called
entry at Stashmx puts destination link in OTPreg before trapping

: !1,1,Stashmx; above with Loadgc code

Stashmx: L<-mx;

OTPreg=L, :Mtrap;

Mtrap: T<-avm1+T+1;
MAR<-sloffset+T;
NOP;

Mtrapa: L=MD, TASK;
mx=L, :Xfer;

can't TASK, T has trap index

fetch dest link for trap

(enter here from PORT0)
; LFCn - call local procedure n (i.e. within same global frame)

; LFC1: L+2, :LFCx;
LFC2: L+3, :LFCx;
LFC3: L+4, :LFCx;
LFC4: L+5, :LFCx;
LFC5: L+6, :LFCx;
LFC6: L+7, :LFCx;
LFC7: L+10, :LFCx;
LFC8: L+11, :LFCx;

; LFCx: count=L LSH 1, L+0, IR=msr0, :SFCr;
; stash index of proc. (*2)
; dest link = 0 for local call
; will return to XferG

; LFCB - call local procedure number 'alpha' (i.e. within same global frame)

LFCB: IR=sr22, :Getalpha;
LFCr: L+0+T+1, :LFCx;
; RET - Return from function call.
; --------------------------------------------------------------
!!1,RETx;

RET: T=lp, :RETx;

RETx: IR=2, :CheckXferTrap;

RETr: MAR=nretlinkoffset+T;
L=nloffset+T+1;
frame=L;
L=MD;
mx=L, L=O, IR=msrO, TASK;
my=L, :FreeSub;

RETr: T=mx, :Xfers;

; LINKB - store back link to enclosing context into local 0
; LINKB is assumed to be A-aligned (no pending branch at entry)
;
LINKB: MAR=lp-T-1;
T=ib;
L=mx-T, TASK;
MD=M, :nextA;

; LLKB - push external link 'alpha'
; LLKB is assumed to be A-aligned (no pending branch at entry)
;
LLKB: T=ib;
L=O-T-1, IR=O, :EfCdoGetlink;
LLKB: :pushTA;

T:alpha
L:-(alpha+1), go call Getlink
alignment requires pushTA
Port Operations

PORTO - PORT Out (XFER thru PORT addressed by TOS)

PORTO: IR=sr3, :Savpcinframe; undiddle lp into my
PORTopc: L=ret5, TASK, :Xpopsub; returns to PORTOr
PORTOr: MAR=T; fetch from TOS
        L=T;
        MD=my;
        MAR=M+1;
        my=L, :Mtrapa;
        frame addr to word 0 of PORT
second word of PORT
source link to PORT address

PORTI - PORT In (Fix up PORT return, always immediately after PORTO)
assumes that my and mx remain from previous xfer

!1,1,PORTIx;
!1,2,PORTinz,PORTIz;

PORTI: MAR=mx, :PORTIx; first word of PORT
PORTIx: SINK=my, BUS=0;
        TASK, :PORTinz;
PORTinz: MD=0;
        MAR=mx+1;
        TASK, :PORTIz;
PORTIz: MD=my, :next;
        store it as second word
        store my or zero
; State Switching

; Savestate subroutine:
; saves state of pre-empted emulation
; Entry conditions:
;  L holds address where state is to be saved
;  assumes undiddled lp
; Exit conditions:
;  lp, stkp, and stack (from base to min[depth+2,8]) saved

; !1,2,DSTr1,Mstopc; actually appears as %1,1777,776,DSTr1,Mstopc; and is located
; in the front of the main file (Mesa.mu).

!17,20,Sav0r,Sav1r,Sav2r,Sav3r,Sav4r,Sav5r,Sav6r,Sav7r,Sav10r,Sav11r,DSTr,.....;
!1,2,Savok,Savmax;

Savestate: temp=L;
Savestatea: T=-12+1;
    L=lp, :Savsuba;
Sav11r: L=stkp, :Savsub;
Sav10r: T=stkp+1;
    L=-7+T;
    L=0+T+1, ALUCY;
    temp2=L, L=0-T, :Savok;
    check if stkp > 6 or negative
    L=stkp+2
    L=-stkp-1
Savmax: T=-7;
    L=stk7, :Savsuba;
    stkp > 5 => save all
    stkp < 6 => save to stkp+2
Savok: SINK=temp2, BUS;
    count=L, :Sav0r;
Sav7r: L=stk6, :Savsub;
Sav6r: L=stk5, :Savsub;
Sav5r: L=stk4, :Savsub;
Sav4r: L=stk3, :Savsub;
Sav3r: L=stk2, :Savsub;
Sav2r: L=stk1, :Savsub;
Sav1r: L=stk0, :Savsub;
Sav0r: SINK=DISP, BUS;
    T=-12, :DSTr1;
    return to caller
    (for DST's benefit)

; Remember, T is negative

Savsub: T=count;
Savsuba: temp2=L, L=0+T+1;
            MAR=temp-T;
            count=L, L=0-T;
            SINK=M, BUS, TASK;
            MD=temp2, :Sav0r;
            dispatch on pos. value
Loadstate subroutine:  
load state for emulation  
Entry conditions:  
L points to block from which state is to be loaded  
Exit conditions:  
stk, mx, my, and stack (from base to min[stk+2,8]) loaded  
(i.e. two words past TOS are saved, if they exist)  
Note: if stk underflows but an interrupt is taken before we detect it, the subsequent Loadstate (invoked by Mgo) will see 377B in the high byte of stk. Thinking this a breakpoint resumption, we will load the state, then dispatch the 377 (via brkbyte) in Xfer0, causing a branch to StkUf (!) This is not a fool-proof check against a bad stk value at entry, but it does protect against the most common kinds of stack errors.

Loadstate:  temp=L, IR=msr0, :NovaIntr0n:  
Lsr:  
T=12, :Ldsuba;  
Lsr12:  my=L, :Ldsub;  
Lsr11:  mx=L, :Ldsub;  
Lsr10:  stk=L;  
T=stk;  
L=177400 AND T;  
brkbyte=L LCY 8;  
L=T+T;  
L=T+T;  
L=stk;  
stk=stk+1, T:stk+1  
Lmax:  
T=7, :Ldsuba;  
Lsr7:  stk7=L, :Ldsub;  
Lsr6:  stk6=L, :Ldsub;  
Lsr5:  stk5=L, :Ldsub;  
Lsr4:  stk4=L, :Ldsub;  
Lsr3:  stk3=L, :Ldsub;  
Lsr2:  stk2=L, :Ldsub;  
Lsr1:  stk1=L, :Ldsub;  
Lsr0:  stk0=L, :Xfer;  
Ldsub:  
T=count;  
L=ALLONES+T;  
count=L, L=T;  
SINK=M, BUS;  
L=MD, TASK, :Lsr0;  
Ldsuba:  
MAR=stk+T;  
L=ALLONES+T;  
count=L, L=T;  
SINK=M, BUS;  
L=MD, TASK, :Lsr0;
; DST - dump state at block starting at <LP>+alpha, reset stack pointer
; assumes DST is A-aligned (also ensures no pending branch at entry)

DST:
  T=ib;
  T=lp+T+1;
  L=lp+offset+T+1, TASK;
  temp=L, IR=retO, :Savestatea;
DSTr1:
  L=my, :Savsuba;
DSTr:
  temp=L, L=0, TASK, BUS=0, :Setstkp;
          L:lp-offset+alpha
          get alpha
          temp=L, L=0, TASK;
          ib=L;
          IR=sr4, :Savpcinframe;
          T...
          temp;
          L=lp+T, TASK, :Loadstate;
          L=lp+offset+alpha
          save my too!
          make Savpcinframe happy
          returns to LStr
          get alpha back
          lp already undiddled

; LST - load state from block starting at <LP>+alpha
; assumes LST is A-aligned (also ensures no pending branch at entry)

LST:
  L=ib;
  temp=L, L=0, TASK;
  ib=L;
  IR=sr4, :Savpcinframe;
  T...
  temp;
  L=lp+T, TASK, :Loadstate;
          L:lp-offset+alpha
          get alpha back
          make Savpcinframe happy
          returns to LSTr
          set up by FreeSub

; LSTF - load state from block starting at <LP>+alpha, then free frame
; assumes LSTF is A-aligned (also ensures no pending branch at entry)

LSTF:
  T=lp+offset;
  L=lp-T, TASK;
  frame=L;
  IR=sr2, :FreeSub;
LSTFr:
  T=frame;
  L=ib+T, TASK, :Loadstate;
          L:lp-offset+alpha
          set up by FreeSub
          get state from dead frame
Emulator Access

RR - push <emulator register alpha>, where:
- RR is A-aligned (also ensures no pending branch at entry)
- alpha: 1 => wdc, 2 => XTSreg, 3 => XTPreg, 4 => ATPreg,
  5 => OTPreg

1.1. DoRamRWB:

RR: L=0, SWMODE, :DoRamRWB;
DoRamRWB: SINK+M, BUS, L=T, :ramOverflow;

RR:

DoRamRWB:

L=0, SWMODE, :DoRamRWB;

SINK+M, BUS, L=T, :ramOverflow;

WR:

L=ret3, TASK, :Xpopsub;

L=ret2, TASK, :Xpopsub;

L=ret2, TASK, :Xpopsub;

JRAM: L=ret2, TASK, :Xpopsub;
JRAMr: SINK+M, BUS, SWMODE, :next;

JRAM:

SINK=M, BUS, SWMODE, :next;

BUS applied to 'nextBa' (=0)
Process/Monitor Support

1.1.MoveParms1; shake B/A dispatch
1.1.MoveParms2; shake B/A dispatch
1.1.MoveParms3; shake B/A dispatch
1.1.MoveParms4; shake B/A dispatch

ME, MRE - Monitor Entry and Re-entry
MXD - Monitor Exit and Depart

1.1.FastMREx; drop ball 1
1.1.FastEEex; drop ball 1
1.1.FastEEexx; shake IR~isME/isMXD
1.2.MXDr, MEr;
1.1.FastEEexxx; shake IR~isMRE

%3, 17, 14, MXDr, MErr, MRErr;
1.2.FastEEtrap1, MEXDdone;
1.2.FastEEtrap2, MREdone;

The following constants are carefully chosen to agree with the above pre-defs

$_{isME} = $6001;
$_{isMRE} = $65403;
$_{isMXD} = $402;
ME: IR~isME, FastEEex;
MXD: IR~isMXD, FastEEex;
MRE: MAR~HardMRE, FastMREx;
FastMREx: IR~isMRE, MXDr;
FastEEex: MAR~stkO, IDISP, FastEEexx;
FastEEexx: T=100000, MXDr;
MXDr: L=MD, mACSOURCE, FastEEexxx;
MER: L=MD-T, mACSOURCE, FastEEexxx;
FastEEexxx: MAR~stkO, SH=O, MXDrr;

Note: if control goes to FastEEtrap1 or FastEEtrap2, AC1 or AC2 will be smashed, but their contents aren't guaranteed anyway.
Note also that MErr and MXDrr cannot TASK.

MXDrr: L=T, T=0, FastEEtrap1;
MERR: T=0+1, FastEEtrap1;
MRErr: L=0+1, TASK, FastEETrap2;
MEXDone: MD=M, L=T, TASK, Setstkp;
MREdone: stkp=L, ME;

Note: if control goes to FastEEtrap1 or FastEEtrap2, AC1 or AC2 will be smashed, but their contents aren't guaranteed anyway.
Note also that MErr and MXDrr cannot TASK.
MXW - Monitor Exit and Wait

MXW: IR=4, MoveParms3; 3 parameters

NOTIFY, BCAST - Awaken process(es) from condition variable

NOTIFY: IR=5, MoveParms1; 1 parameter
BCAST: IR=6, MoveParms1; 1 parameter

REQUEUE - Move process from queue to queue

REQUEUE: IR=7, MoveParms3; 3 parameter

Parameter Transfer for Nova code linkages
Entry Conditions:
T: 1
IR: dispatch vector index of Nova code to execute

MoveParms4: L=stk3, TASK;
AC3=L;
MoveParms3: L=stk2, TASK;
FastEEtrap2: AC2=L;
MoveParms2: L=stk1, TASK;
FastEEtrap1: AC1=L;
MoveParms1: L=stk0, TASK;
AC0=L;

L=0, TASK;
stkp=L;
T=DISP+1, :STOP;

if you uncomment this, don't forget the pre-def above!

(enter here from MRE)
(enter here from ME/MXD)
indicate stack empty
; Miscellaneous Operations

; CATCH - an emulator no-op of length 2.
; CATCH is assumed to be A-aligned (no pending branch at entry)

CATCH:   L=mpc+1, TASK, :nextAput;          duplicate of 'nextA'

; STOP - return to Nova at 'NovaDVloc+1'
; control also comes here from process opcodes with T set appropriately

!1.1.GotoNova;                                  shake B/A dispatch
STOP:    L=NovaDVloc+T, :GotoNova;

; STARTIO - perform Nova-like I/O function

STARTIO:  L=retA, TASK, :Xpopsub;
STARTIOR:  SINK=M, STARTF, :next;

; MISC - escape hatch for more than 256 opcodes

!1.2.RamMisc,RCLK;                             RCLK or something else
!1.1,MISCx;                                    shake B/A branch
MISC:    IR=sr36, :Getalpha;
MISCr:    L=I+T, :MISCx;
MISCx:    L=CLOCKLOC-1, SH=0;
          temp=L, IR=0, :RamMisc;
RCLK:     L=clockreg, :Opushc;
RamMisc:  L=3, SWMODE, :DoRamRWB;              don't TASK here!


; BLT - block transfer
; assumes stack has precisely three elements:
; stk0 - address of first word to read
; stk1 - count of words to move
; stk2 - address of first word to write
; the instruction is interruptible and leaves a state suitable
; for re-execution if an interrupt must be honored.

!1,1,BLTx; shakes entry B/A branch

BLT: stk7=L, SWMODE, :BLTx;
    stk7=0 <> branch pending
BLTx: IR=msr0, :ramBLTloop;
      IR+ is harmless

; BLTL - block transfer (long pointers)
; assumes stack has precisely three elements:
; stk0, stk1 - address of first word to read
; stk2 - count of words to move
; stk3, stk4 - address of first word to write
; the instruction is interruptible and leaves a state suitable
; for re-execution if an interrupt must be honored.

BLTL: stk7=L, L=T, SWMODE, :DoRamRWB;
      stk7=0 <> branch pending, L:1

; BLTC - block transfer from code segment
; assumes stack has precisely three elements:
; stk0 - offset from code base of first word to read
; stk1 - count of words to move
; stk2 - address of first word to write
; the instruction is interruptible and leaves a state suitable
; for re-execution if an interrupt must be honored.

!1,1,BLTCx; shake B/A dispatch

BLTC: stk7=L, SWMODE, :BLTCx;
BLTCx: IR=srl, :ramBLTloop;

; BITBLT - do BITBLT using ROM subroutine
; If BITBLT A-aligned, B byte will be ignored

!1,1,BITBLTx; shake B/A dispatch

BITBLT: stk7=L, :BITBLTx;
BITBLTx: L=10, SWMODE, :DoRamRWB;
      save even/odd across ROM call
; Mesa/Nova Communication
;

; Subroutines to Enable/Disable Nova Interrupts
; currently each subroutine has only one caller
!7,1,NovaIntrOffx;

NovaIntrOff : T=100000;
NovaIntrOffx: L=NWW OR T, TASK, IDISP;
NWW=L, :Mstop;
NovaIntrOn: T=100000;
L=NWW AND NOT T, IDISP;
NWW=L, L=0, :Lsr;

IWDC - Increment Wakeup Disable Counter (disable interrupts)
!1,2,IDnz,IDz;
IWDC: L=wdc+1, TASK, :IDnz;

DWDC - Decrement Wakeup Disable Counter (enable interrupts)
!1,1,DWDCx;
DWDC: MAR=WWLOC, :DWDCx;
DWDCx: T=NWW;
L=MD OR T, TASK;
NWW=L;
SINK=ib, BUS=0;
L=wdc-1, TASK, :IDnz;

; Ensure that one instruction will execute before an interrupt is taken
IDnz: wdc=L, :next;
IDz: wdc=L, :nextAdead;

Entry to Mesa Emulation
; ACO holds address of current process state block
; Location 'PSBloc' is assumed to hold the same value
Mgo: L=ACO, :Loadstate;
Nova Interface

$START $L004020,0,0;   Nova emulator return address

Transfer to Nova code

Entry conditions:
L contains Nova PC to use

Exit conditions:
Control transfers to ROMO at location 'START' to do Nova emulation
Nova PC points to code to be executed
Except for parameters expected by the target code, all Nova ACs contain garbage
Nova interrupts are disabled

GotoNova: PC=L, IR=msr0, :NovaIntrOff;   stash Nova PC, return to Mstop

Control comes here when an interrupt must be taken. Control will pass to the Nova emulator with interrupts enabled.

Intstop: L=NovaDVloc, TASK;   resume at Nova loc. 30B
PC=L, :Mstop;

Stash the Mesa pc and dump the current process state, then start fetching Nova instructions.

Mstop: IR=sr2, :Savpcinfframe;   save mpc for Nova code
Mstopr: MAR=CurrentState;   get current state address
IR=ret1;   will return to 'Mstopc'
L=MD, :Savestate;   dump the state

The following instruction must be at location 'SWRET', by convention.

Mstopc: L=uCodeVersion, SWMODE;   stash ucode version number
cp=L, :START;   off to the Nova ...
Get Alto Definitions (Mesab.mu included internally by XMesaRAM.mu)

#AltoConstats23.mu;

Reserve locations 0-17 of RAM for device tasks (silent boot)

%;17,1777,0,TaskO,Task1,Task2,Task3,Task4,Task5,Task6,Task7,
; Task10,Task11,Task12,Task13,Task14,Task15,Task16,Task17;
; TaskO: TASK, :TaskO;
; Task1: TASK, :Task1;
; Task2: TASK, :Task2;
; Task3: TASK, :Task3;
; Task4: TASK, :Task4;
; Task5: TASK, :Task5;
; Task6: TASK, :Task6;
; Task7: TASK, :Task7;
; Task10: TASK, :Task10;
; Task11: TASK, :Task11;
; Task12: TASK, :Task12;
; Task13: TASK, :Task13;
; Task14: TASK, :Task14;
; Task15: TASK, :Task15;
; Task16: TASK, :Task16;
; Task17: TASK, :Task17;

; Reserve 774-1003 for Ram Utility Area.
; %7, 1777, 774, RU774, RU775, RU776, RU777, RU1000, RU1001, RU1002, RU1003;

; For the moment, just throw these locations away. This is done only
; to squelch the "unused predef" warnings that would otherwise occur.
; If we ever run short of Ram, assign these to real instructions
; somewhere in microcode executed only by the Emulator.
RU774: NOP;
RU775: NOP;
RU776: NOP;
RU777: NOP;
RU1000: NOP;
RU1001: NOP;
RU1002: NOP;
RU1003: NOP;

; Predefs for griffin
; %1,1777,560,HBlt; HBlt is the entry point.
%1,1777,177,MULret;

; Predefs for Pup checksum
; %7, 1777, 1402, PupChecksum;

; Now bring in Mesa overflow microcode
#XMesaRAM.mu;

; MISC - Miscellaneous instructions specified by alpha
; alpha=11 => RCLK has been handled by ROM
; T contains alpha on arrival at MISC in RAM
; Precisely one of the following lines must be commented out.
; MISC: L-0, SWMODE, :Setstkp; dummy MISC implementation
#Float.mu; REAL implementation

; HBlt (Griffin) - this code may be omitted

#HBlt.mu;

; PupChecksum - this code may be omitted

#Checksum.mu;
XMesaRAM.mu - Overflow XMesa microcode from ROM1
version 6, compatible with main microcode >=39
Last modified by Johnsson on April 28, 1980 7:01 PM

Separate assembly requires...

#Mesab.mu;

Entry Point Definitions:
The definitions below must correspond to those in Mesab.

%1,1777,20,GoToROM;
%1,1777,402,BLTintpend,BLTloop;
%3,1777,404,BLTnoint,BLTint,BLTLnoint,BLTLint;
%1,1777,410,Overflow;
%1,1777,411,JramBITBLT;

BITBLT linkage:
An additional constraint peculiar to the BITBLT microcode is that
the high-order 7 bits of the return address be 1's. Hence,
the recommended values are:
no ROM1 extant or emulator in ROM1 => BITBLTret = 177577B
ROM1 extant and emulator in RAM => BITBLTret = 177175B

$ROMBITBLT $L004124,0.0; BITBLT routine address (124B) in ROM0
$BITBLTret $177175; (may be even or odd)
The third value in the following pre-def must be: (BITBLTret AND 777B)-1
%1,1777,174,BITBLTintr,BITBLTdone;

Overflow instruction dispatch
dispatched in ROM1

GoToROM: L=ONE, SWMODE;
gp=L:romMgo;
smash G to disable
optimization on initial entry
; Double Precision Arithmetic

; DADD - add two double-word quantities, assuming:
; stack contains precisely 4 elements
;                   shake B/A dispatch
; 1,1,DADDx;
; 1,2,DADDnocarry,DADDcarry:
DADD: T=<stk2, :DADDx;
DADDx: L=<stk0+T;
       stk0=L, ALUCY;
       T=<stk3, :DADDnocarry;
DADDnocarry: L=<stk1+T, :DASCTail;
DADDcarry: L=<stk1+T+1, :DASCTail;

; DSUB - subtract two double-word quantities, assuming:
; stack contains precisely 4 elements
;                   shake B/A dispatch
IR=msr0, :DSUBsub;

; Double-precision subtract subroutine
; 1,2,DSUBborrow,DSUBnoborrow;
; 7,1,DSUBx:
DSUBsub: T=<stk2, :DSUBx;
DSUBx: L=<stk0+T;
       stk0=L, ALUCY;
       T=<stk3, :DSUBborrow;
DSUBborrow: L=<stk1-T+1, IDISP, :DASCTail;
DSUBnoborrow: L=<stk1-T, IDISP, :DASCTail;

; Common exit code

DASCTail: stk1=L, ALUCY, :DASCTail;
DASCTail: T=2, :Dsetstkp;
Dsetstkp: L=<stkp-T, SWMODE, :Dsetstkp;
Setstkp: stkp=L, :romnext;

'next' has proper SWMODE bit
DCOMP - compare two long integers, assuming:
stack contains precisely 4 elements
result left on stack is -1, 0, or +1 (single-precision)

\[
\begin{align*}
&: 11,1, \text{DCOMPxa}; \\
&: 10,1, \text{DCOMPxb}; \\
&: 11,2, \text{DCOMPnocarry}, \text{DCOMPcarry}; \\
&: 11,2, \text{DCOMPgtr}, \text{DCOMPequal};
\end{align*}
\]

\[
\begin{align*}
&\text{DCOMP:} \quad \text{IR}=T+100000, \quad :\text{DCOMPxa}; \\
&\text{DCOMPxa:} \quad L=\text{stk}1+T, \quad :\text{DCOMPxb}; \\
&\text{DCOMPxb:} \quad \text{stk}1=L; \\
&\quad \text{L}=\text{stk}3+T, \quad \text{TASK}; \\
&\quad \text{stk}3=L, \quad :\text{DSUBsub}; \\
&\text{DCOMPnocarry:} \quad T=\text{stk}0, \quad :\text{DCOMPnocarry}; \\
&\text{DCOMPcarry:} \quad L=0-1, \quad \text{BUS}=0, \quad :\text{DCOMPsetT}; \\
&\quad \text{L}=M \text{ OR } T; \\
&\quad \text{SH}=0; \\
&\text{DCOMPgtr:} \quad T=3, \quad :\text{DCOMPgtr}; \\
&\text{DCOMPequal:} \quad L=0+1, \quad :\text{DCOMPequal}; \\
&\quad \text{stk}0=L, \quad :\text{setstkp}; \\
\end{align*}
\]

DUCOMP - compare two long cardinals, assuming:
stack contains precisely 4 elements
result left on stack is -1, 0, or +1 (single-precision)
(i.e. result = sign(stk1, stk0 DSUB stk3, stk2))

\[
\begin{align*}
&\text{DUCOMP:} \quad \text{IR}=sr3, \quad :\text{DSUBsub}; \\
&\quad \text{returns to DCOMPr}
\end{align*}
\]
Emulator Access

RR - push <emulator register alpha>, where:
- RR is A-aligned (also ensures no pending branch at entry)
  - alpha: 1 => wdc, 2 => XTSreg, 3 => XTPreg, 4 => ATPreg.
  - 5 => OTPreg

17,10,RR0,RR1,RR2,RR3,RR4,RR5,...

RR: SINK=<b>, BUS; dispatch on alpha
RRO: L=0, SWMODE, :RRO; (so SH=0 below will branch)
RR1: L=wdc, SH=0, :romUnta1;
RR2: L=XTSreg, SH=0, :romUnta1;
RR3: L=XTPreg, SH=0, :romUnta1;
RR4: L=ATPreg, SH=0, :romUnta1;
RR5: L=OTPreg, SH=0, :romUnta1;

WR - emulator register alpha = <TOS> (popped), where:
- WR is A-aligned (also ensures no pending branch at entry)
  - alpha: 1 => wdc, 2 => XTSreg

17,10,WRO,WR1,WR2,...

WR: L=ret3, TASK, :Xpopsub; performed in ROM
WRO: SINK=<b>, BUS; dispatch on alpha
WR1: wdc=L, :romnextA;
WR2: XTSreg=L, :romnextA;
; BLT  - block transfer
;   assumes stack has precisely three elements:
;     stk0 - address of first word to read
;     stk1 - count of words to move
;     stk2 - address of first word to write
;   the instruction is interruptible and leaves a state suitable
;   for re-execution if an interrupt must be honored.

!1,2,BLTmore,BLTdone;
!1,2,BLTsource,BLTCsource;
!1,2,BLTeven,BLTodd;
!1,1,BLIntinx;

; Entry sequence in ROM1; actual entry is at BLTloop

;BLT:     stk7=L, SWMODE, :BLTnoint;
;BLTx:    IR=msr0, :ramBLTloop;

BLTloop:  L=T+stk1-1, BUS=0, :BLTnoint;
BLTnoint: stk1=L, L=BUS AND -T, IDISP, :BLTmore;

BLTmore:  T=cp, :BLTsource;

BLTsource: MAR=stk0, :BLTupdate;
BLTCsource: XMAR=stkO+T, :BLTupdate;

BLTupdate: L=stkO+1;
     stkO=L;
     L=stk2+1;
     T=MD;
     MAR=stk2;
     stk2=L, L=T;
     SINK=NWW, BUS=0, TASK;
     MD=M, :BLTintpend;

BLTintpend: SINK=wdc, BUS=0, :BLTloop;

; Must take an interrupt if here (via BLT or BITBLT)

BLTint:   SINK=stk7, BUS=0, :BLTintx;
BLTintx:  L=mpc-1, :BLTeven;
BLTeven:  mpc=L, L=0, :BLToddd;
BLToddd:  ib=L, SWMODE;
:romIntstop;

; BLT completed

BLTdone:  SINK=stk7, BUS=0, SWMODE, :Setstkp;

stk7=0 => return to 'nextA'
;---------------------------------------------------------------------
; BLTL - block transfer (long pointers)
; assumes stack has precisely five words:
; stk0, stk1 - address of first word to read
; stk2 - count of words to move
; stk3, stk4 - address of first word to write
; the instruction is interruptible and leaves a state suitable
; for re-execution if an interrupt must be honored.
; the following are used as temporaries (here and BITBLT):
; stk7 - saved B/A flag from instruction dispatch
; stk6 - saved value of emulator bank register
;

!7,1.BLTLsetBR;
!7,1.BLTLsetBRx;
!7,10.BLTLret0.BLTLret1.BLTLret2.BBret3.BBret4...;
!1,2.BLTLintpend, BLTLloop;
!1,2.BLTLmore.BLTLDone;
!1,2.BLTLnoint.BLTLint;

; Note: ROM1 code does stk7=L

BLTL:       MAR=BankReg;
            T=stk1;
            L=stk1+T;
            temp=L LSH 1, IR=msr0;
            L=MD, TASK;
            stk6=L;
            T=stk4;
            T=3.T, :BLTLsetBR;
            
BLTLloop:   L=T=stk2-1, BUS=0, :BLTLnoint;
BLTLnoint:  stk2=L, :BLTLmore;

BLTLmore:   MAR=stk0;
            L=stk0+1;
            stk0=L;
            L=stk3+1;
            T=MD;
            XMAR=stk3;
            stk3=L, L=T;
            SINK=WWW, BUS=0, TASK;
BLTLret0:   MD=M, :BLTLintpend;

BLTLintpend: SINK=wdc, BUS=0, :BLTLloop;

BLTLint:    IR=sr2, :BLTLsetBR;
BLTLret2:   MD=stk6, :BLTLint;

BLTDone:    IR=sr1, :BLTLsetBR;
BLTLret1:   MD=stk6, L=stk6 AND NOT T, :BLTDone;

BLTLsetBR:  MAR=BankReg, :BLTLsetBRx;
BLTLsetBRx: L=temp OR T, IDISP;
            SINK=0, BUS=0, :BLTLret0;

shakes BUS=0 and IR="
shake IR="
access bank register
high source bits
L: high source *2
temp: high source *4;
L: old bank register
stk6: stashed register
T: high dest bits
(would like to avoid this)
returns to BLTLret0
decrement count, test done
T: -1 the last time
fetch source word
bump source pointer
bump destination pointer
initiate store
L: data
check for possible interrupt
stash data
check if enabled
restore bank before interrupt
BLTint shakes branch
restore bank before exit
BLTDone shakes branch, L=0

(used by BLTLret0 only)
force branch for BLTLret0
others must shake
BITBLT - do BITBLT using ROMO subroutine

If BITBLT A-aligned, B byte will be ignored

temporaries (in addition to BLTL):
stk5 - 0=>short BITBLT, 2=>long BITBLT
temp - holds value from second word of bbtable

from ROM

; 1.1.BITBLTx;  -> shake B/A dispatch
;BITBLT:    stk7=L, :BITBLTx;
;BITBLTx:   L+10, SWMODE, :DoRamRWB;

; 1.2.IntOff, TestLong;  -> save even/odd across ROM call
; 1.2.LongBB, DoBITBLT;
; 2.3.1.BBshortDone, BBlongDone;

JramBITBLT:  L=ib, TASK;
stk7=L;
L=3, TASK;
stkp=L, :BITBLT;
BITBLT:      L=stk0;
AC2=L, L=0, TASK;
stk5=L;
SINK=wdc, BUS=0;
T=100000, :IntOff;
IntOff:      L=WWW OR T;
WWW=L, :TestLong;
TestLong:    MAR=stk0+1;
L=stk1;
AC1=L;
L=MD, BUS=0, TASK;
temp=L, :LongBB;

; 7.1, BBx;  -> shake IR=sr3
LongBB:      MAR=BankReg;
L=2;
stk5=L;
IR=sr3;
L=MD, TASK;
stk6=L, :BLTsetBR;
BBret3:      MD=temp, :DoBITBLT;
DoBITBLT:    L=BITBLTret, SWMODE;
PC=L, L=0, :ROMBITBLT;
BITBLTDone:  T=100000;
L=WWW AND NOT T;
SINK=stk5, BUS;
WWW=L, L=T=0, :BBshortDone;
BBlongDone:  IR=sr4, :BLTsetBR;
BBret4:      MD=stk6, L=1, :BBshortDone;
BBshortDone: brkbyte=L, BUS=0, SWMODE, :Setstkp;

; BITBLTintr:L=AC1;
SINK=stk5, BUS;
stk1=L, :BITBLTint;

shake B/A dispatch
save even/odd across ROM call
also shake SetBR branch
save alignment for interrupts
set stkp for interrupts
check if Mesa interrupts off
if so, shut off Nova's
fetch word for long check
stash intermediate state
BR word
old Bank reg
get return address
L-0 for Alto II ROMO "feature"
don't bother to validate stkp
pick up intermediate state
stash intermediate state