```
;-------------------------------------------------------------;
;              M E S A   M I C R O C O D E                    ;
;                      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:
;  uCodeVersion   Microcode version   Mesa release
;       0              34              4.1
;       1              39              5.0
;       2              41              6.0

$uCodeVersion   $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
;       sktp=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.
```

```
;-------------------------------------------------------------
; Location-specific Definitions
;-------------------------------------------------------------


; 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, ROM0), 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 "RAM1" and "ROM1" is systematically
; replaced by "RAM0".

%1,1777,0,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.
;-------------------------------------------------------------

%1,1777,20,Mgo;                         Normal entry to Mesa Emulator - load state
;                                               of process specified by AC0.

%1,1777,400,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.

%1,1777,776,DSTr1,Mstopc;                       Return addresses for 'Savestate'.  By
;                                               standard convention, 'Mstopc' must be at 777.
```

```
;-------------------------------------------------------------
; Linkage from Mesa emulator to ROM0
;       The Mesa emulator uses a number of subroutines that reside in ROM0.
;       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 ROM0,
;       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 ROM0, 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 7 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)
;-------------------------------------------------------------
$ROMMUL         $L004120,0,0;                   MUL routine address (120B) in ROM0
$ROMDIV         $L004121,0,0;                   DIV routine address (121B) in ROM0

$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;               return addresses from MUL/DIV in ROM0

;-------------------------------------------------------------
; 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;              return addresses from BITBLT in ROM0

;-------------------------------------------------------------
; 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 ROM0

$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;                            return address from RAMCYCX to Fieldsub

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

%1,1777,22,WFnzct,WFret;                        return address from RAMCYCX to WF
```

```
;-------------------------------------------------------------
; I n s t r u c t i o n  f e t c h
;
; 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.

%7,1777,1400,NOOP,ME,MRE,MXW,MXD,NOTIFY,BCAST,REQUEUE;          000-007
%7,1777,1410,LL0,LL1,LL2,LL3,LL4,LL5,LL6,LL7;                  010-017
%7,1777,1420,LLB,LLDB,SL0,SL1,SL2,SL3,SL4,SL5;                 020-027
%7,1777,1430,SL6,SL7,SLB,PL0,PL1,PL2,PL3,LG0;                  030-037
%7,1777,1440,LG1,LG2,LG3,LG4,LG5,LG6,LG7,LGB;                  040-047
%7,1777,1450,LGDB,SG0,SG1,SG2,SG3,SGB,LI0,LI1;                 050-057
%7,1777,1460,LI2,LI3,LI4,LI5,LI6,LIN1,LINI,LIB;                060-067
%7,1777,1470,LIW,LINB,LADRB,GADRB,,,,;                         070-077
%7,1777,1500,R0,R1,R2,R3,R4,RB,W0,W1;                          100-107
%7,1777,1510,W2,WB,RF,WF,RDB,RD0,WDB,WD0;                      110-117
%7,1777,1520,RSTR,WSTR,RXLP,WXLP,RILP,RIGP,WILP,RIL0;          120-127
%7,1777,1530,WS0,WSB,WSF,WSDB,RFC,RFS,WFS,;                    130-137
%7,1777,1540,,,,,,,,;                                          140-147
%7,1777,1550,,,,,,,,;                                          150-157
%7,1777,1560,,,SLDB,SGDB,PUSH,POP,EXCH,LINKB;                  160-167
%7,1777,1570,DUP,NILCK,,BNDCK,,,,;                             170-177
%7,1777,1600,J2,J3,J4,J5,J6,J7,J8,J9;                          200-207
%7,1777,1610,JB,JW,JEQ2,JEQ3,JEQ4,JEQ5,JEQ6,JEQ7;              210-217
%7,1777,1620,JEQ8,JEQ9,JEQB,JNE2,JNE3,JNE4,JNE5,JNE6;          220-227
%7,1777,1630,JNE7,JNE8,JNE9,JNEB,JLB,JGEB,JGB,JLEB;            230-237
%7,1777,1640,JULB,JUGEB,JUGB,JULEB,JZEQB,JZNEB,,JIW;           240-247
%7,1777,1650,ADD,SUB,MUL,DBL,DIV,LDIV,NEG,INC;                 250-257
%7,1777,1660,AND,OR,XOR,SHIFT,DADD,DSUB,DCOMP,DUCOMP;          260-267
%7,1777,1670,ADD01,,,,,,,;                                     270-277
%7,1777,1700,EFC0,EFC1,EFC2,EFC3,EFC4,EFC5,EFC6,EFC7;          300-307
%7,1777,1710,EFC8,EFC9,EFC10,EFC11,EFC12,EFC13,EFC14,EFC15;    310-317
%7,1777,1720,EFCB,LFC1,LFC2,LFC3,LFC4,LFC5,LFC6,LFC7;          320-327
%7,1777,1730,LFC8,,,,,,,;                                      330-337
%7,1777,1740,,LFCB,SFC,RET,LLKB,PORTO,PORTI,KFCB;              340-347
%7,1777,1750,DESCB,DESCBS,BLT,,BLTC,,ALLOC,FREE;               350-357
%7,1777,1760,IWDC,DWDC,STOP,CATCH,MISC,BITBLT,STARTIO,JRAM;    360-367
%7,1777,1770,DST,LST,LSTF,,WR,RR,BRK,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;                                   dispatch on ib
                ib←L, T←0+1, BUS=0, :NOOP;                      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;                                          stash to permit TASKing
                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;                                           updated pc
                SINK←NWW, BUS=0;                                 check pending interrupts
nextAcomx:      T←177400, :nextAi;

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

nextAni:        L←MD AND T, BUS, :nextAgo;                       L←"B"↑8, dispatch on "A"
nextAgo:        ib←L LCY 8, L←T←0+1, :NOOP;                      establish exit state

;
;  Interrupt pending - check if enabled.
;

nextAi:         L←MD;
                SINK←wdc, BUS=0;                                 check wakeup counter
                T←M.T, :nextAini;                                isolate left byte
nextAini:       SINK←M, L←T, BUS, :nextAgo;                      dispatch even byte

;
;  Interrupt pending and enabled.
;
!1,2,nextXBini,nextXBii;

nextAii:        L←mpc-1;                                         back up mpc for Savpcinframe
                mpc←L, L←0, :nextXBii;
```

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

nextXB:         L←MAR←mpc+T;
                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, :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.
;

nextXBi:        SINK←wdc, BUS=0, :nextXBni;                     check wakeup counter

;
;  Interrupt pending and enabled.
;

nextXBii:       ib←L, :Intstop;                                 ib = 0 for even, ~= 0 for odd
```

```
;-------------------------------------------------------------
; S u b r o u t i n e s
;-------------------------------------------------------------

;
;  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,RD0r,Splitcomr,RXLPrb,WXLPrb,MISCr,;

; Returns for Xpopsub only

!17,20,WSTRrB,WSTRrA,JRAMr,WRr,STARTIOr,PORT0r,WD0r,ALLOCrx,
       FREErx,NEGr,RFSra,RFSrb,WFSra,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
!17,20,Tpop,Tpop0,Tpop1,Tpop2,Tpop3,Tpop4,Tpop5,Tpop6,Tpop7,,,,,,,;

Popsub:         L←stkp-1, BUS, TASK, :Popsuba;
Popsuba:        stkp←L, :Tpop;                                  old stkp > 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;                               returns to Xpopret
;                                                               Note:  putting Tpop here makes
;                                                               stack underflow logic work if
;                                                               stkp=0

Xpopret:        SINK←saveret, BUS;
                :WSTRrB;
```

```
;-----------------------------------------------------------------
; 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,Getalphax;                                          shake IR← dispatch
!7,1,GetalphaAx;                                         shake IR← dispatch

Getalpha:       T←ib, IDISP;
Getalphax:      ib←L RSH 1, L←0, BUS=0, :Fieldra;        ib←0, set branch 1 pending

GetalphaA:      L←MAR←mpc+1;                             initiate fetch
GetalphaAx:     mpc←L;
                T←177400;                                mask for new ib
                L←MD AND T, T←MD;                        L: new ib, T: whole word
Getalphab:      T←377.T, IDISP;                          T now has alpha
                ib←L LCY 8, L←0+1, :Fieldra;             return: no branch pending


;-----------------------------------------------------------------
; FetchAB subroutine:
;       Entry conditions:  none
;       Exit conditions:
;         T: <<mpc>+1>
;         ib: unchanged (caller must ensure return to 'nextA')
;-----------------------------------------------------------------
!1,1,FetchAB;                                            drops ball 1
!7,1,FetchABx;                                           shakes IR← dispatch
!7,10,LIWr,JWr,,,,,,;                                    return points

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 Splitcomr 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;                                                    drop ball 1
%160,377,217,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;                    L←1 for Getalpha
SplitalphaB:    saveret←L, L←0, BUS=0, :Splitcom;               (keep ball 1 in air)

Splitcom:       IR←sr33, :Getalpha;                             T:alpha[0-7]
Splitcomr:      L←17 AND T, :Splitx;                            L:alpha[4-7]
Splitx:         righthalf←L, L←T, TASK;                         L:alpha, righthalf:alpha[4-7]
                temp←L;                                         temp:alpha
                L←temp, BUS;                                    dispatch on alpha[1-3]
                temp←L LCY 8, SH<0, :Split0;                    dispatch on alpha[0]

Split0:         L←T←0, :Splitout0;                              L,T:alpha[1-3]
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;                             L:alpha[0-3]

Splitout0:      SINK←saveret, BUS, TASK;                        dispatch return
                lefthalf←L, :RILPr;                             lefthalf:alpha[0-3]
```

```
;-------------------------------------------------------------------
; D i s p a t c h e s
;-------------------------------------------------------------------


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

pushMD:         L←stkp+1, IR←stkp;                              (IR← causes no branch)
                stkp←L, T←0+1, :pushMDa;

pushMDA:        L←stkp+1, IR←stkp;                              (IR← causes no branch)
                stkp←L, T←0, :pushMDa;

pushMDa:        SINK←DISP, L←T, BUS;                            dispatch on old stkp value
                L←MD, SH=0, TASK, :push0;

;-------------------------------------------------------------
; 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'.
;-------------------------------------------------------------
!1,2,pushT1B,pushT1A;                                           keep ball 1 in air

pushTB:         L←stkp+1, BUS, :pushT1B;
pushTA:         L←stkp+1, BUS, :pushT1A;

pushT1B:        stkp←L, L←T, TASK, :push0;
pushT1A:        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

!17,20,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' 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;                                                    shakes B/A dispatch from RCLK
!5,2,Dpushx,RCLKr;                                              shakes IR←2000 dispatch and
;                                                               provides return to RCLK

Dpush:          MAR←L←ib+T, :DpB;                               L: address of low half

Dpusha:         SINK←ib, BUS=0;
                MAR←L←M+T, :DpA;

DpA:            IR←0, :Dpushb;                                  mACSOURCE will produce 0
DpB:            IR←2000, :Dpushb;                               mACSOURCE will produce 1

Dpushb:         temp←L, :Dpushx;                                temp: address of low half
Dpushx:         L←MD, TASK, :Dpushc;
Dpushc:         taskhole←L;                                     taskhole: low half bits
                T←0+1;
                L←stkp+T+1;
                MAR←temp+1;                                     fetch high half
                stkp←L;                                         stkp ← stkp+2
                L←taskhole;                                     L: low half bits
                SINK←stkp, BUS, :dpush;                         dispatch on new stkp
dpush:          T←MD, :dpush;                                   T: high half bits

dpush1:         stk0←L, L←T, TASK, mACSOURCE, :push1;           stack cells are S-registers,
dpush2:         stk1←L, L←T, TASK, mACSOURCE, :push2;           so mACSOURCE does not affect
dpush3:         stk2←L, L←T, TASK, mACSOURCE, :push3;           addressing.
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, 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 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,MDpopuf,MDpop0,MDpop1,MDpop2,MDpop3,MDpop4,MDpop5,MDpop6,MDpop7,,,,,,,;

StoreB:         L←stkp-1, BUS;
StoreBa:        stkp←L, TASK, :MDpopuf;
StoreA:         L←stkp-1, BUS;
StoreAa:        stkp←L, BUS=0, TASK, :MDpopuf;                  keep branch (1) alive

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,dpopuf2,dpopuf1,dpop1,dpop2,dpop3,dpop4,dpop5,dpop6,dpop7,,,,,,,;
!1,1,Dpopb;                                                     required by placement of
;                                                               MDpopuf only.

Dpop:           L←T←ib+T+1;
MDpopuf:        IR←0, :Dpopb;                                   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.
Dpopa:          L←T←M+T+1;
                IR←ib, :Dpopb;
Dpopb:          MAR←T, temp←L;
dpopuf2:        L←stkp-1, BUS;
                stkp←L, TASK, :dpopuf2;

dpopuf1:        :StkUf;                                         stack underflow, honor TASK
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;
```

```
;-----------------------------------------------------------------
; 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 ROM0, correct to AltoCode23.Mu
;------------------------------------------------------------------

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

$AC3            $R0;
$MASK1          $R0;
$AC2            $R1;
$AC1            $R2;
$YMUL           $R2;
$RETN           $R2;
$AC0            $R3;
$SKEW           $R3;
$NWW            $R4;
$SAD            $R5;
$CYRET          $R5;
$TEMP           $R5;
$PC             $R6;
$XREG           $R7;
$CYCOUT         $R7;
$WIDTH          $R7;
$PLIER          $R7;
$XH             $R10;
$DESTY          $R10;
$WORD2          $R10;
$DWAX           $R35;
$STARTBITSM1    $R35;
$MASK           $R36;
$SWA            $R36;
$DESTX          $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;
$xCBA           $R22;
$xAECL          $R23;
$xSLC           $R24;
$xMTEMP         $R25;
$xHTAB          $R26;
$xYPOS          $R27;
$xDWA           $R30;

; Ethernet

$xECNTR         $R12;
$xEPNTR         $R13;

; Memory Refresh

$xCLOCKTEMP     $R11;
$xR37           $R37;

; Audio (obsolete)

$xAudioWdCt     $R71;
$xAudioData     $R72;
```

```
;-----------------------------------------------------------------
; Registers used by Mesa Emulator
;-----------------------------------------------------------------

; R registers

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

; Registers shared by Nova and Mesa emulators
;       Nova ACs are set explicitly by Mesa process opcodes and for ROM0 calls
;       Other R-registers smashed by BITBLT and other ROM0 subroutines

$brkbyte        $R0;                    (AC3) bytecode to execute after a breakpoint
;                                       Warning! brkbyte must be reset to 0 after ROM calls!
;                                       (see BITBLT)
$mx             $R1;                    (AC2) x register for XFER
;                                       Warning! smashed by BITBLT and MUL/DIV/LDIV
$saveret        $R2;                    (AC1) R-temporary for return indices and values
$newfield       $R3;                    (AC0) new field bits for WF and friends
;                                       Warning! must be R-register; assumed safe across CYCLE

$count          $R5;                    scratch R register used for counting
$taskhole       $R7;                    pigeonhole for saving things across TASKs
;                                       Warning! smashed by all ROM calls!
$ib             $R10;                   instruction byte, 0 if none (0,,byte)
;                                       Warning! smashed by BITBLT
$clockreg       $R37;                   low-order bits of real-time clock

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

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

; Mesa evaluation stack

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

; Miscellaneous S registers

$mask           $R41;                   used by string instructions, among others
$unused1        $R42;                   not safe across call to BITBLT
$unused2        $R43;                   not safe across call to BITBLT
$alpha          $R44;                   alpha byte (among other things)
$index          $R45;                   frame size index (among other things)
$entry          $R46;                   allocation table entry address (among other things)
$frame          $R47;                   allocated frame pointer (among other things)
$righthalf      $R41;                   right 4 bits of alpha or beta
$lefthalf       $R45;                   left 4 bits of alpha or beta
$unused3        $R50;                   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      $5;                             = -(lpoffset+1+pcoffset) [see Savpcinframe]
$retlinkoffset  $2;                             offset from local frame base to return link
$nretlinkoffset $177774;                        = -(lpoffset-retlinkoffset)

$gpoffset       $4;                             global frame overhead + 1
$ngpoffset      $177773;                        = -(gpoffset + 1)
$gfioffset      $L0,12000,100;                  offset from global frame base to gfi word (=0)
$ngfioffset     $4;                             = gpoffset-gfioffset [see XferGfz]
$cpoffset       $1;                             offset from global frame base to code pointer
$gpcpoffset     $2;                             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
;-----------------------------------------------------------------

$mACSOURCE      $L024016,000000,000000;         sets only F2.  ACSOURCE also sets BS and RSEL
$msr0           $L000000,012000,000100;         IDISP => 0, no IR← dispatch, a 'special' zero
$BUSAND~T       $L000000,054015,000040;         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
```

```
;---------------------------------------------------------------
; Mesac.Mu - Jumps, Load/Store, Read/Write, Binary/Unary/Stack Operators
; Last modified by Johnsson - July 20, 1979  8:59 AM
;---------------------------------------------------------------


;---------------------------------------------------------------
; J u m p s
;---------------------------------------------------------------

; 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;                                                   must be odd (shakes IR← below)

JB:             T←ib, :JBx;
JBx:            L←400 OR T;                                     ←DISP will do sign extension
                IR←M;                                           400 above causes branch (1)
                L←DISP-1, :Jbranch;                             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 JWr
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;                                    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)
;-----------------------------------------------------------------
!1,2,Jcz,Jco;

JZEQB:          SINK←stk0, BUS=0;                               test TOS = 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;                               test TOS = 0
                L←stkp-1, TASK, :JZNEBne;

JZNEBne:        stkp←L, :JB;                                    branch, pick up alpha
JZNEBeq:        stkp←L, :nextA;                                 no branch, alignment => nextA
```

```
;---------------------------------------------------------------
; JEQn - if TOS (popped) = TOS (popped), jump PC-relative by n, assuming:
;       stack has precisely two elements
;---------------------------------------------------------------
!1,2,JEQnB,JEQnA;
!7,1,JEQNEcom;                                          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,JNEcom;                                     return points from JEQNEcom

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


;
;       JEQ/JNE common code
;
; !7,1,JEQNEcom;        appears above with JEQn
; !1,2,JEQcom,JNEcom;   appears above with JEQB

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

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
;---------------------------------------------------------------


JNEnB:          temp←L RSH 1, L←0, :JEQNEcom;           temp:0, L:0
JNEnA:          temp←L, L←0, :JEQNEcom;                 temp:1, L:0

!1,2,JNEne,JNEeq;

JNEcom:         L←stkp-T-1, :JNEne;                     L: old stkp - 2

JNEne:          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←ALLONES, :Jadjust;

Jadjust:        L←stk1+T+1;                                     L:stk1 + (0 or 100000)
                temp←L;
                SINK←DISP, BUSODD;                              dispatch ADC/SUB
                T←stk0+T+1, :Jadc;

Jadc:           L←temp-T-1, :Jcommon;                           perform add complement
Jsub:           L←temp-T, :Jcommon;                             perform subtract

Jcommon:        T←ONE;                                          warning: not T←0+1
                L←stkp-T-1, ALUCY;                              test ADC/SUB outcome
                SINK←DISP, SINK←lgm10, BUS=0, TASK, :Jcz;       dispatch on encoded bit 12

Jcz:            stkp←L, :Jnobz;                                 carry is zero (stkp←stkp-2)
Jco:            stkp←L, :Jbo;                                   carry is one (stkp←stkp-2)

Jnobz:          L←mpc+1, TASK, :nextAput;                       no jump, alignment=>nextAa
Jbz:            T←ib, :JBx;                                     jump
Jbo:            T←ib, :JBx;                                     jump
Jnobo:          L←mpc+1, TASK, :nextAput;                       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,JIul;
!1,1,JIWx;

JIW:            L←stkp-T-1, TASK, :JIWx;                    stkp←stkp-2
JIWx:           stkp←L;
                T←stk0;
                L←MAR←mpc+1;                                load alphabeta
                mpc←L;
                L←stk1-T-1;                                 do unsigned compare
                ALUCY;
                T←MD, :JIuge;

JIuge:          L←mpc+1, TASK, :nextAput;                   out of bounds - to 'nextA'
JIul:           L←cp+T, TASK;                               (removing this TASK saves a
                taskhole←L;                                 word, but leaves a run of
                T←taskhole;                                 15 instructions)
                MAR←stk0+T;                                 fetch <<cp>+alphabeta+X>
                NOP;
                L←MD-1, :Jbranch;                           L: offset
```

```
;---------------------------------------------------------------
; L o a d s
;---------------------------------------------------------------

; 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←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

LG0:            MAR←gp-1, :pushMD;
LG1:            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 - push <<gp>+alpha>
;---------------------------------------------------------------

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


;---------------------------------------------------------------
; LGDB - push <<gp>+alpha>, push <<gp>+alpha+1>
;       LGDB is A-aligned (also ensures no pending branch at entry)
;---------------------------------------------------------------

LGDB:           T←gp+T+1, :LDcommon;                            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, :pushT1B;
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;


;---------------------------------------------------------------
; 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 377B8)
;---------------------------------------------------------------

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
```

```
;---------------------------------------------------------------
; S t o r e s
;---------------------------------------------------------------


;---------------------------------------------------------------
; 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;
```

```
;-----------------------------------------------------------------
; 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+T, :StoreB;
SG3:            MAR←gp+T+1, :StoreB;


;-----------------------------------------------------------------
; SGB - <<gp>+alpha>←TOS (popped)
;-----------------------------------------------------------------

SGB:            IR←sr7, :Getalpha;                              returns to SGBr
SGBr:           T←ngpoffset+T+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+T+1, :SDcommon;                            T: gp-gpoffset+lpoffset
```

```
;-----------------------------------------------------------------
; P u t 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

PL0:            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
```

```
;---------------------------------------------------------------
; B i n a r y   o p e r a t i o n s
;---------------------------------------------------------------

; 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,WFSrb,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 push0) 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 20B

!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
Bincomd:        temp2←L, :Binpop;                               stash briefly

BincomA:        L←2000 OR T;                                    make mACSOURCE produce 1
Binpop:         IR←M, :BincomB;

Binpop1:        T←stk1;
                L←stk0, :Binend;
Binpop2:        T←stk2;
                L←stk1, :Binend;
Binpop3:        T←stk3;
                L←stk2, :Binend;
Binpop4:        T←stk4;
                L←stk3, :Binend;
Binpop5:        T←stk5;
                L←stk4, :Binend;
Binpop6:        T←stk6;
                L←stk5, :Binend;
Binpop7:        T←stk7;
                L←stk6, :Binend;

Binend:         SINK←DISP, BUS;                                 perform return dispatch
                SINK←temp2, BUS, :ADDr;                         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;                               pick up ball 1
                L←stkp-1, :pushT1B;                             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
;---------------------------------------------------------------
!7,1,MULDIVcoma;                                        shakes stack dispatch
!1,2,GoROMMUL,GoROMDIV;
!7,2,MULx,DIVx;                                         also shakes bus dispatch

MUL:            IR←T←ret5, :BincomB;
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
;                                                       perhaps, but garbage now.
MULDIVret1:     T←AC1;                                  Normal return
                L←stkp+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;
DIVr:           AC1←L, L←T, BUS=0, :MULDIVcoma;         BUS=0 => DIVx


;---------------------------------------------------------------
; LDIV - push quotient of <TOS-1>,,<TOS-2>/<TOS> (all popped)
;   remainder recoverable by PUSH
;---------------------------------------------------------------

LDIV:           IR←sr27, :Popsub;                       get divisor
LDIVf:          AC2←L;                                  stash it
                IR←T←ret7, :BincomB;                    L:low bits, T:high bits
LDIVr:          AC1←L, L←T, IR←0, :DIVx;                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;                                                    shakes stack dispatch
!1,2,Lshift,Rshift;
!1,2,DoShift,Shiftdone;
!1,2,DoRight,DoLeft;
!1,1,Shiftdonex;

SHIFT:          IR←T←ret10, :BincomB;
SHIFTr:         temp←L, L←T, TASK, :SHIFTx;                     L: value, T: count
SHIFTx:         count←L;
                L←T←count;
                L←0-T, SH<0;                                    L: -count, T: count
                IR←sr1, :Lshift;                                IR← causes no branch

Lshift:         L←37 AND T, TASK, :Shiftcom;                    mask to reasonable size

Rshift:         T←37, IR←37;                                    equivalent to IR←msr0
                L←M AND T, TASK, :Shiftcom;                     mask to reasonable size

Shiftcom:       count←L, :Shiftloop;

Shiftloop:      L←count-1, BUS=0;                               test for completion
                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;                   dispatch to push result
Shiftdonex:     L←temp, TASK, :push0;
```

```
;---------------------------------------------------------------
; D o u b l e - P r e c i s i o n  A r i t h m e t i c
;---------------------------------------------------------------

!1,1,DSUBsub;                                                   shake B/A dispatch
!3,4,DAStail,,,DCOMPr;                                          returns from DSUBsub
!1,1,Dsetstkp;                                                  shake ALUCY dispatch

;---------------------------------------------------------------
; DADD - add two double-word quantities, assuming:
;       stack contains precisely 4 elements
;---------------------------------------------------------------
!1,1,DADDx;                                                     shake B/A dispatch
!1,2,DADDnocarry,DADDcarry;

DADD:           T←stk2, :DADDx;                                 T:low bits of right operand
DADDx:          L←stk0+T;                                       L:low half of sum
                stk0←L, ALUCY;                                  stash, test carry
                T←stk3, :DADDnocarry;                           T:high bits of right operand

DADDnocarry:    L←stk1+T, :DASCtail;                            L:high half of sum
DADDcarry:      L←stk1+T+1, :DASCtail;                          L:high half of sum


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

DSUB:           IR←msr0, :DSUBsub;


;---------------------------------------------------------------
; Double-precision subtract subroutine
;---------------------------------------------------------------
!1,2,DSUBborrow,DSUBnoborrow;
!7,1,DSUBx;                                                     shake IR← dispatch

DSUBsub:        T←stk2, :DSUBx;                                 T:low bits of right operand
DSUBx:          L←stk0-T;                                       L:low half of difference
                stk0←L, ALUCY;                                  borrow = ~carry
                T←stk3, :DSUBborrow;                            T:high bits of right operand

DSUBborrow:     L←stk1-T-1, IDISP, :DASCtail;                   L:high half of difference
DSUBnoborrow:   L←stk1-T, IDISP, :DASCtail;                     L:high half of difference


;---------------------------------------------------------------
; Common exit code
;---------------------------------------------------------------

DASCtail:       stk1←L, ALUCY, :DAStail;                        carry used by double compares
DAStail:        T←2, :Dsetstkp;                                 adjust stack pointer

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) )
;---------------------------------------------------------------
!1,1,DCOMPxa;                                                   shake B/A dispatch
!10,1,DCOMPxb;                                                  shake IR← dispatch
!1,2,DCOMPnocarry,DCOMPcarry;
!1,2,DCOMPgtr,DCOMPequal;

DCOMP:          IR←T←100000, :DCOMPxa;                          IR←msr0, must shake dispatch
DCOMPxa:        L←stk1+T, :DCOMPxb;                             scale left operand
DCOMPxb:        stk1←L;
                L←stk3+T, TASK;                                 scale right operand
                stk3←L, :DSUBsub;                               do DSUB, return to DCOMPr

DCOMPr:         T←stk0, :DCOMPnocarry;                          L: stk1, ALUCY pending

DCOMPnocarry:   L←0-1, BUS=0, :DCOMPsetT;                       left opnd < right opnd
DCOMPcarry:     L←M OR T;                                       L: stk0 OR stk1
                SH=0;
DCOMPsetT:      T←3, :DCOMPgtr;                                 T: amount to adjust stack

DCOMPgtr:       L←0+1, :DCOMPequal;                             left opnd > right opnd
DCOMPequal:     stk0←L, :Dsetstkp;                              stash result


;---------------------------------------------------------------
; 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:         IR←sr3, :DSUBsub;                               returns to DCOMPr
```

```
;-------------------------------------------------------------
; R a n g e   C h e c k i n g
;-------------------------------------------------------------


;-------------------------------------------------------------
; NILCK - check TOS for NIL (0), trap if so
;-------------------------------------------------------------
!1,2,InRange,OutOfRange;

NILCK:          L←ret17, :Xpopsub;                          returns to NILCKr
NILCKr:         T←ONE, SH=0, :NILCKpush;                    test TOS=0

NILCKpush:      L←stkp+T, :InRange;

InRange:        SINK←ib, BUS=0, TASK, :Setstkp;             pick up ball 1
OutOfRange:     T←sBoundsFaultm1+T+1, :KFCr;                T:SD index; go trap


;-------------------------------------------------------------
; BNDCK - check subrange inclusion
;       if TOS-1 ~IN [0..TOS) then trap (test is unsigned)
;       only TOS is popped off
;-------------------------------------------------------------
!7,1,BNDCKx;                                                shake push dispatch

BNDCK:          IR←T←ret22, :BincomB;                       returns to BNDCKr
BNDCKr:         L←M-T, :BNDCKx;                             L: value, T: limit
BNDCKx:         T←0, ALUCY, :NILCKpush;
```

```
;---------------------------------------------------------------
; R e a d s
;---------------------------------------------------------------

; 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;                     returns to RBr
RBr:            L←stkp-1, BUS, :ReadB;

ReadB:          stkp←L, :MAStkT;                        to pushMD
ReadA:          stkp←L, BUS=0, :MAStkT;                 to pushMDA


;---------------------------------------------------------------
; 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;                       returns to Dpush


;---------------------------------------------------------------
; RD0 - temp←<TOS>, push <<temp>>, push <<temp>+1>
;---------------------------------------------------------------

RD0:            IR←sr32, :Popsub;                       returns to RD0r
RD0r:           L←0, :Dpusha;
```

```
;---------------------------------------------------------------
; RILP - push <<<lp>+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]>
;---------------------------------------------------------------
!3,1,IPcom;                                                     shake IR← at WILPr

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←lefthalf+T;                                 start memory cycle
                L←righthalf;
IPcomx:         T←MD, IDISP;                                    T:local/global value
                MAR←M+T, :pushMD;                               start fetch/store


;---------------------------------------------------------------
; RIL0 - push <<<lp>>>
;---------------------------------------------------------------
!1,2,RILxB,RILxA;

RIL0:           MAR←lp-T-1, :RILxB;                             fetch local 0

RILxB:          IR←msr0, L←0, :IPcomx;                          to pushMD
RILxA:          IR←sr1, L←sr1 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;                               fetch TOS
RXLPrb:         L←righthalf+T, TASK;                            L:TOS+alpha[4-7]
                righthalf←L, :RILPr;                            now act like RILP
```

```
;---------------------------------------------------------------
; W r i t e s
;---------------------------------------------------------------


;---------------------------------------------------------------
; Wn - <<TOS> (popped)+n>←<TOS> (popped)
;---------------------------------------------------------------
!1,2,WnB,WnA;                                                   keep ball 1 in air

W0:             T←0, :WnB;
W1:             T←ONE, :WnB;
W2:             T←2, :WnB;

WnB:            IR←sr2, :Wsub;                                  returns to StoreB

WnA:            IR←sr3, :Wsub;                                  returns to StoreA

;---------------------------------------------------------------
; Write subroutine:
;---------------------------------------------------------------
!7,1,Wsubx;                                                     shake IR← dispatch

Wsub:           L←stkp-1, BUS, :Wsubx;
Wsubx:          stkp←L, IDISP, :MAStkT;


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

WB:             IR←sr16, :Getalpha;                             returns to WBr
WBr:            :WnB;                                           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;                                                      shake stack dispatch

WSB:            IR←T←ret14, :BincomA;                           alignment requires BincomA
WSBr:           T←M, L←T, :WSBx;

WSBx:           MAR←ib+T, :WScom;
WScom:          temp←L;
WScoma:         L←stkp-1;
                MD←temp;
                mACSOURCE, TASK, :Setstkp;


;---------------------------------------------------------------
; WS0 - act like WSB but with alpha value of zero
;---------------------------------------------------------------
!7,1,WS0x;                                                      shake stack dispatch

WS0:            IR←T←ret15, :BincomB;
WS0r:           T←M, L←T, :WS0x;

WS0x:           MAR←T, :WScom;
```

```
;---------------------------------------------------------------
; WILP - <<lp>+alpha[0-3]>+alpha[4-7] ← <TOS> (popped)
;---------------------------------------------------------------

WILP:           L←ret2, :Splitalpha;                        get halves of alpha
WILPr:          IR←sr2;                                     IPcom will exit to StoreB
                T←lp, :IPcom;                               prepare to undiddle


;---------------------------------------------------------------
; WXLP - <TOS>+<<lp>+alpha[0-3]>+alpha[4-7] ← <TOS-1> (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]
                righthalf←L, :WILPr;                        now act like WILP


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

WDB:            IR←sr31, :Popsub;                           returns to Dpop


;---------------------------------------------------------------
; WD0 - temp←<TOS> (popped), pop into <temp>+1 and <temp>
;---------------------------------------------------------------

WD0:            L←ret6, TASK, :Xpopsub;                     returns to WD0r
WD0r:           L←0, :Dpopa;


;---------------------------------------------------------------
; WSDB - like WDB but with address below data words, assuming:
;       WSDB is A-aligned (also ensures no pending branch at entry)
;---------------------------------------------------------------
!7,1,WSDBx;

WSDB:           IR←sr24, :Popsub;                           get low data word
WSDBra:         saveret←L;                                  stash it briefly
                IR←T←ret20, :BincomA;                       alignment requires BincomA
WSDBrb:         T←M, L←T, :WSDBx;                           L:high data, T:address
WSDBx:          MAR←T←ib+T+1;                               start store of low data word
                temp←L, L←T;                                temp:high data
                temp2←L, TASK;                              temp2:updated address
                MD←saveret;                                 stash low data word
                MAR←temp2-1, :WScoma;                       start store of high data word
```

```
;-----------------------------------------------------------------
; U n a r y  o p e r a t i o n s
;-----------------------------------------------------------------


;-----------------------------------------------------------------
; INC - TOS ← <TOS>+1
;-----------------------------------------------------------------

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


;-----------------------------------------------------------------
; NEG - TOS ← -<TOS>
;-----------------------------------------------------------------

NEG:            L←ret11, TASK, :Xpopsub;
NEGr:           L←0-T, :Untail;


;-----------------------------------------------------------------
; DBL - TOS ← 2*<TOS>
;-----------------------------------------------------------------

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


;-----------------------------------------------------------------
; Unary operation common code
;-----------------------------------------------------------------

Untail:         T←M, :pushTB;
```

```
;---------------------------------------------------------------
; S t a c k  a n d  M i s c e l l a n e o u s  O p e r a t i o n s
;---------------------------------------------------------------


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

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


;---------------------------------------------------------------
; POP - subtract 1 from stack pointer
;---------------------------------------------------------------

POP:            L←stkp-1, SH=0, TASK, :Setstkp;                 L=0 <=> branch 1 pending
;                                                               need not check stkp=0


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

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


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

EXCH:           IR←ret11, :EXCHx;
EXCHx:          L←stkp-1;                                       dispatch on stkp-1
                L←M+1, BUS, TASK, :Bincomd;                     set temp2←stkp
EXCHr:          T←M, L←T, :dpush;                               Note: dispatch using temp2


;---------------------------------------------------------------
; LADRB - push alpha+lp (undiddled)
;---------------------------------------------------------------
!1,1,LADRBx;                                                    shake branch from Getalpha

LADRB:          IR←sr10, :Getalpha;                             returns to LADRBr
LADRBr:         T←nlpoffset+T+1, :LADRBx;
LADRBx:         L←lp+T, :Untail;


;---------------------------------------------------------------
; GADRB - push alpha+gp (undiddled)
;---------------------------------------------------------------
!1,1,GADRBx;                                                    shake branch from Getalpha

GADRB:          IR←sr11, :Getalpha;                             returns to GADRBr
GADRBr:         T←ngpoffset+T+1, :GADRBx;
GADRBx:         L←gp+T, :Untail;
```

```
;---------------------------------------------------------------
; S t r i n g   O p e r a t i o n s
;---------------------------------------------------------------
!7,1,STRsub;                                                    shake stack dispatch
!1,2,STRsubA,STRsubB;
!1,2,RSTRrx,WSTRrx;

STRsub:         L←stkp-1;                                       update stack pointer
                stkp←L;
                L←ib+T;                                         compute index and offset
                SINK←M, BUSODD, TASK;
                count←L RSH 1, :STRsubA;

STRsubA:        L←177400, :STRsubcom;                           left byte
STRsubB:        L←377, :STRsubcom;                              right byte

STRsubcom:      T←temp;                                         get string address
                MAR←count+T;                                    start fetch of word
                T←M;                                            move mask to more useful place
                SINK←DISP, BUSODD;                              dispatch to caller
                mask←L, SH<0, :RSTRrx;                          dispatch B/A, mask for WSTR


;---------------------------------------------------------------
; 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;                                stash string base address
RSTRrx:         L←MD AND T, TASK, :RSTRB;                       isolate good bits

RSTRB:          temp←L, :RSTRcom;
RSTRA:          temp←L LCY 8, :RSTRcom;                         right-justify byte

RSTRcom:        T←temp, :pushTA;                                go push result byte


;---------------------------------------------------------------
; 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;
WSTRr:          temp←L, :STRsub;                                stash string base
WSTRrx:         L←MD AND NOT T, :WSTRB;                         isolate good bits

WSTRB:          temp2←L, L←ret0, TASK, :Xpopsub;                stash them, return to WSTRrB
WSTRA:          temp2←L, L←ret0+1, TASK, :Xpopsub;              stash them, return to WSTRrA

WSTRrA:         taskhole←L LCY 8;                               move new data to odd byte
                T←taskhole, :WSTRrB;

WSTRrB:         T←mask.T;
                L←temp2 OR T;
                T←temp;                                         retrieve string address
                MAR←count+T;
                TASK;
                MD←M, :nextA;
```

```
;---------------------------------------------------------------
; F i e l d   I n s t r u c t i o n s
;---------------------------------------------------------------

!1,2,RFrr,WFrr;                                          returns from Fieldsub
!7,1,Fieldsub;                                           shakes stack dispatch
; !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
;---------------------------------------------------------------

RF:             IR←sr12, :Popsub;
RFr:            L←ret0, :Fieldsub;
RFrr:           T←mask.T,  :pushTA;                      alignment requires 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
;---------------------------------------------------------------
; !1,2,WFnzct,WFret; - see location-specific definitions

WF:             IR←T←ret17, :BincomB;                    L:new data, T:address
WFr:            newfield←L, L←ret0+1, :Fieldsub;         (actually, L←ret1)

WFrr:           T←mask;
                L←M AND NOT T;                           set old field bits to zero
                temp←L;                                  stash result
                T←newfield.T;                            save new field bits
                L←temp OR T, TASK;                       merge old and new
                CYCOUT←L;                                stash briefly
                T←index, BUS=0;                          get position, test for zero
                L←WFretloc, :WFnzct;                     get return address from ROM

WFnzct:         PC←L;                                    stash return
                L←20-T, SWMODE;                          L:remaining count to cycle
                T←CYCOUT, :RAMCYCX;                      go cycle remaining amount
WFret:          MAR←frame;                               start memory
                L←stkp-1;                                pop remaining word
                MD←CYCOUT, TASK, :JZNEBeq;               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
;---------------------------------------------------------------

WSF:            IR←T←ret16, :BincomB;                    L:address, T:new data
WSFr:           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
RFSra:          temp←L;                                         stash for WFSa
                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
;-------------------------------------------------------------
!1,2,Fieldsuba,WFSa;

WFS:            L←ret14, TASK, :Xpopsub;                        get alpha and beta
WFSra:          temp←L;                                         stash temporarily
                IR←T←ret21, :BincomB;                           L:new data, T:address
WFSrb:          newfield←L, L←ret0+1, BUS=0, :Fieldsub;         returns quickly to WFSa
WFSa:           frame←L;                                        stash address
                T←177400;                                       to separate alpha and beta
                L←temp AND T, T←temp, :Getalphab;               L:alpha, T:both
;                                                               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:       temp2←L, L←T, IR←msr0, TASK, :Fieldsuba;        stash return
Fieldsuba:      frame←L, :GetalphaA;                            stash base address
;                                                               T: beta, ib: alpha
Fieldra:        L←ret5;
                saveret←L, :Splitcomr;                          get two halves of beta
Fieldrb:        T←righthalf;                                    index for MASKTAB
                MAR←MASKTAB+T;                                  start fetch of mask
                T←lefthalf+T+1;                                 L:left-cycle count
                L←17 AND T;                                     mask to 4 bits
                index←L;                                        stash position
                L←MD, TASK;                                     L:mask for caller's use
                mask←L;                                         stash mask
                T←frame;                                        get base address
                L←MAR←ib+T;                                     add alpha
                frame←L;                                        stash updated address for WF
                L←Fieldretloc;                                  return location from RAMCYCX
                PC←L;
                T←MD, SWMODE;                                   data word into T for cycle
                L←index, :RAMCYCX;                              count to cycle, go do it
Fieldrc:        SINK←temp2, BUS;                                return dispatch
                L←T←CYCOUT, :RFrr;                              cycled data word in L and T
```

```
;---------------------------------------------------------------
; Mesad.Mu - Xfer, State switching, process support, Nova interface
; Last modified by Levin - January 4, 1979  2:18 PM
;---------------------------------------------------------------


;---------------------------------------------------------------
; F r a m e   A l l o c a t i o n
;---------------------------------------------------------------


;---------------------------------------------------------------
; 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;                                     subroutine returns
!1,2,ALLOCrf,XferGrf;                                   failure returns
!3,4,Alloc0,Alloc1,Alloc2,Alloc3;                       dispatch on pointer flag
;       if more than 2 callers, un-comment the following pre-definition:
; !17,1,Allocx;                                         shake IR← dispatch

AllocSub:       L←avm1+T+1, TASK, :Allocx;              fetch av entry

Allocx:         entry←L;                                save av entry address
                L←MAR←entry;
                T←3;                                    mask for pointer flags
                L←MD AND T, T←MD;                       (L←MD AND 3, T←MD)
                temp←L, L←MAR←T;                        start reading pointer
                SINK←temp, BUS;                         branch on bits 14:15
                frame←L, :Alloc0;

;
; Bits 14:15 = 00, a frame of the right index is queued for allocation
;

Alloc0:         L←MD, TASK;                             new entry for frame vector
                temp←L;                                 new value of vector entry
                MAR←entry;                              update frame vector
                L←T←frame, IDISP;                       establish exit conditions
                MD←temp, :ALLOCr;                       update and return

;
; Bits 14:15 = 01, allocation list empty:  restore argument, take failure return
;

Alloc1:         L←temp2, IDISP, TASK;                   restore parameter
                temp2←L LSH 1, :ALLOCrf;                allocation failed

;
; Bits 14:15 = 10, a pointer to an alternate list to use
;

Alloc2:         temp←L RSH 1, :Allocp;                  indirection: index←index/4

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)
;---------------------------------------------------------------

!3,4,RETr,FREEr,LSTFr,;                                 FreeSub returns
!17,1,Freex;                                            shake IR← dispatch

FreeSub:        MAR←frame-1;                            start read of fsi word
Freex:          NOP;                                    wait for memory
                T←MD;                                   T←index
                L←MAR←avm1+T+1;                         fetch av entry
                entry←L;                                save av entry address
                L←MD;                                   read current pointer
                MAR←frame;                              write it into current frame
                temp←L, TASK;
                MD←temp;                                write!
                MAR←entry;                              entry points at frame
                IDISP, TASK;
                MD←frame, :RETr;                        free
```

```
;-----------------------------------------------------------------
; ALLOC - allocate a frame whose fsi is specified by <TOS> (popped)
;-----------------------------------------------------------------
!1,1,Savpcinframe;                                          (here so ALLOCrf can call it)
!7,10,XferGT,Xfer,Mstopr,PORTOpc,LSTr,ALLOCrfr,,;           return points for Savpcinframe
!1,2,doAllocTrap,XferGfz;                                   used by XferGrf

ALLOC:          L←ret7, TASK, :Xpopsub;                     returns to ALLOCrx
ALLOCrx:        temp2←L LSH 1, IR←msr0, :AllocSub;          L,T: fsi
ALLOCr:         L←stkp+1, BUS, :pushT1B;                    duplicates pushTB

;
;  Allocation failed - save mpc, undiddle lp, push fsi*4 on stack, then trap
;

ALLOCrf:        IR←sr5, :Savpcinframe;                      failure because lists empty
ALLOCrfr:       L←temp2, TASK, :doAllocTrap;                pick up trap parameter

;
;  Inform software that allocation failed
;

doAllocTrap:    ATPreg←L;                                   store param. to trap proc.
                T←sAllocTrap, :Mtrap;                       go trap to software


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

FREE:           L←ret10, TASK, :Xpopsub;                    returns to FREErx
FREErx:         frame←L, TASK;
                IR←sr1, :FreeSub;
FREEr:          :next;
```

```
;-------------------------------------------------------------
; D e s c r i p t o r  I n s t r u c t i o n s
;-------------------------------------------------------------


;-------------------------------------------------------------
; 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;                     T:address of frame

DESCBcom:       MAR←gfioffset+T;                                start fetch of gfi word
                T←gfimask;                                      mask to isolate gfi bits
                T←MD.T;                                         T:gfi
                L←ib+T, T←ib;                                   L:gfi+alpha, T:alpha
                T←M+T+1, :pushTA;                               pushTA because A-aligned


;-------------------------------------------------------------
; 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
```

```
;---------------------------------------------------------------
; T r a n s f e r   O p e r a t i o n s
;---------------------------------------------------------------


;---------------------------------------------------------------
; 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)
;---------------------------------------------------------------
; !1,1,Savpcinframe;                                            required by PORTO
; !7,10,XferGT,Xfer,Mstopr,PORTOpc,LSTr,ALLOCrfr,,;             returns (appear with ALLOC)
!7,1,Savpcx;                                                    shake IR← dispatch
!1,2,Spcodd,Spceven;                                            pc odd or even

Savpcinframe:   T←cp, :Savpcx;                                  code segment base
Savpcx:         L←mpc-T;                                        L is code-relative pc
                SINK←ib, BUS=0;                                 check for odd or even pc
                T←M, :Spcodd;                                   pick up pc word addr

Spcodd:         L←0-T, TASK, :Spcopc;                           - pc => odd, this word
Spceven:        L←0+T+1, TASK, :Spcopc;                         + pc => even, next word

Spcopc:         taskhole←L;                                     pc value to save
                L←0;                                            (can't merge above - TASK)
                T←npcoffset;                                    offset to pc stash
                MAR←lp-T, T←lp;                                 (MAR←lp-npcoffset, T←lp)
                ib←L;                                           clear ib for XferG
                L←nlpoffset+T+1;                                L:undiddled lp
                MD←taskhole;                                    stash pc in frame+pcoffset
                my←L, IDISP, TASK;                              store undiddled lp
                lp←L, :XferGT;
```

```
;---------------------------------------------------------------
; 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,Xfer0r,Xfer1r;                                             return points
!1,2,Loadgc,LoadgcTrap;
!1,2,LoadgcOK,LoadgcNull;                                       good global frame or null
!1,2,LoadgcIn,LoadgcSwap;                                       in-core or swapped out

Loadgc:         L←lpoffset+T;                                   diddle (presumed) lp
                lp←L;                                           (only correct if frame ptr)
                T←MD;                                           global frame address
                L←MD;                                           2nd word (PC or EV offset)
                MAR←cpoffset+T;                                 read code pointer
                mpc←L, L←T;                                     copy g to L for null test
                L←gpoffset+T, SH=0;                             diddle gp, test for null
                T←MD, BUSODD, :LoadgcOK;                        check for swapped out

LoadgcOK:       MAR←T, :LoadgcIn;                               write into code segment

LoadgcIn:       gp←L, L←T;                                      set global frame pointer
                cp←L, IDISP, TASK;                              set code pointer
                MD←ONE, :Xfer0r;

;
;       picked up global frame of zero somewhere, call it unbound
;
!1,1,Stashmx;
LoadgcNull:     T←sUnbound, :Stashmx;                           BUSODD may be pending

;
;       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.
;-----------------------------------------------------------------
!3,4,Xfers,XferG,RETxr,;                                    returns from CheckXferTrap
!1,2,NoXferTrap,DoXferTrap;
!3,1,DoXferTrapx;

CheckXferTrap:  L←XTSreg, BUSODD;                           XTSreg[15]=1 => trap
                SINK←DISP, BUS, :NoXferTrap;                dispatch (possible) return

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;                        L:trap parameter
                XTPreg←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'.
;---------------------------------------------------------------

!3,4,Xfer0,Xfer1,Xfer2,Xfer3;                                   destination link type

Xfer:           T←mx;                                           mx[14:15] is dest link type
                IR←0, :CheckXferTrap;
Xfers:          L←3 AND T;                                      extract type bits
                SINK←M, L←T, BUS;                               L:dest link, branch on type
                SH=0, MAR←T, :Xfer0;                            check for link = 0. Memory
;                                                               data is used only if link
;                                                               is frame pointer or indirect

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

Xfer0:          IR←msr0, :Loadgc;                               to LoadgcNull if dest link = 0
Xfer0r:         L←T←mpc;                                        offset from cp: - odd, + even

;
; 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
;
!1,2,Xdobreak,Xnobreak;
!1,2,Xfer0B,Xfer0A;
!1,2,XbrkB,XbrkA;
!1,2,XbrkBgo,XbrkAgo;

                SINK←brkbyte, BUS=0;                            set up by Loadstate
                SH<0, L←0, :Xdobreak;                           dispatch even/odd pc

;
; Not proceeding from a breakpoint - simply pick up next instruction
;

Xnobreak:       :Xfer0B;

Xfer0B:         L←MAR←cp+T, :nextAdeafa;                        fetch word, pc even
Xfer0A:         L←MAR←cp-T, SH=0, :nextXBdeaf;                  fetch word, pc odd (L=0)

;
; Proceeding from a breakpoint - dispatch brkbyte and clear it
;

Xdobreak:       ib←L, :XbrkB;                                   clear ib for XbrkA

XbrkB:          IR←sr20;                                        here if BRK at even byte
                L←MAR←cp+T, :GetalphaAx;                        set up ib (return to XbrkBr)

XbrkA:          L←cp-T;                                         here if BRK at odd byte
                mpc←L, L←0, BUS=0, :XbrkBr;                     ib already zero (to XbrkAgo)

XbrkBr:         SINK←brkbyte, BUS, :XbrkBgo;                    dispatch brkbyte

XbrkBgo:        brkbyte←L RSH 1, T←0+1, :NOOP;                  clear brkbyte, act like nextA
XbrkAgo:        brkbyte←L, T←0+1, BUS=0, :NOOP;                 clear brkbyte, act like next
```

```
;-------------------------------------------------------------
; mx[14-15] = 01
;       Destination link is procedure descriptor:
;         mx[0-8]: GFT index (gfi)
;         mx[9-13]: EV bias, or entry number (en)
;-------------------------------------------------------------

Xfer1:          temp←L RSH 1;                                   temp:ep*2+garbage
                count←L MLSH 1;                                 since L=T, count←L LCY 1;
                L←count, TASK;                                  gfi now in 0-7 and 15
                count←L LCY 8;                                  count:gfi w/high bits garbage
                L←count, TASK;
                count←L LSH 1;                                  count:gfi*2 w/high garbage
                T←count;
                T←1777.T;                                       T:gfi*2
                MAR←gftm1+T+1;                                  fetch GFT[T]
                IR←sr1, :Loadgc;                                pick up two word entry into
;                                                               gp and mpc
Xfer1r:         L←temp, TASK;                                   L:en*2+high bits of garbage
                count←L RSH 1;                                  count:en+high garbage
                T←count;
                T←enmask.T;                                     T:en
                L←mpc+T+1, TASK;                                (mpc has EV base in code seg)
                count←L LSH 1, :XferG;                          count:ep*2


;-------------------------------------------------------------
; mx[14-15] = 10
;       Destination link is indirect:
;         mx[0-15]: address of location holding destination link
;-------------------------------------------------------------

Xfer2:          NOP;                                            wait for memory
                T←MD, :Xfers;

;-------------------------------------------------------------
; mx[14-15] = 11
;       Destination link is unbound:
;         mx[0-15]: passed to trap handler
;-------------------------------------------------------------

Xfer3:          T←sUnbound, :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;                                        parameter to CheckXferTrap
                IR←ONE, :CheckXferTrap;
XferG:          T←count;                                        index into entry vector
                MAR←cp+T;                                       fetch of new pc and fsi
                T←cp-1;                                         point just before bytes
;                                                               (main loop increments mpc)
                IR←sr1;                                         note: does not cause branch
                L←MD+T;                                         relocate pc from cseg base
                T←MD;                                           second word contains fsi
                mpc←L;                                          new pc setup, ib already 0
                T←377.T, :AllocSub;                             mask for size index
;
; Stash source link in new frame, establishing dynamic link
;
XferGr:         MAR←retlinkoffset+T;                            T has new frame base
                L←lpoffset+T;                                   diddle new lp
                lp←L;                                           install diddled lp
                MD←my;                                          source link to new frame

;
; Stash new global pointer in new frame (same for local call)
;
                MAR←T;                                          write gp to word 0 of frame
                T←gpoffset;                                     offset to point at gf base
                L←gp-T, TASK;                                   subtract off offset
                MD←M, :nextAdeaf;                               global pointer stashed, GO!

;
;  Frame allocation failed - push destination link, then trap
;
; !1,2,doAllocTrap,XferGfz;                                     (appears with ALLOC)

XferGrf:        L←mx, BUS=0;                                    pick up destination, test = 0
                T←count-1, :doAllocTrap;                        T:2*ep+1

;       if destination link is zero (i.e. local procedure call), we must first
;       fabricate the destination link

XferGfz:        L←T, T←ngfioffset;                              offset from gp to gfi word
                MAR←gp-T;                                       start fetch of gfi word
                count←L LSH 1;                                  count:4*ep+2
                L←count-1;                                      L:4*ep+1
                T←gfimask;                                      mask to save gfi only
                T←MD.T;                                         T:gfi
                L←M+T, :doAllocTrap;                            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
;-----------------------------------------------------------------
!1,2,EFCgetr,LLKBr;                                             return points
!1,2,framelink,codelink;
!7,1,Fetchlink;                                                 shake IR← in KFCB

Getlink:        T←gp;                                           diddled frame address
                MAR←T←ngpoffset+T+1;                            fetch word 0 of global frame
                L←temp+T, T←temp;                               L:address of link in frame
                taskhole←L;                                     stash it
                L←cp+T;                                         L:address of link in code
                SINK←MD, BUSODD, TASK;                          test bit 15 of word zero
                temp2←L, :framelink;                            stash code link address

framelink:      MAR←taskhole, :Fetchlink;                       fetch link from frame
codelink:       MAR←temp2, :Fetchlink;                          fetch link from code

Fetchlink:      SINK←DISP, BUS=0;                               dispatch to caller
                L←T←MD, :EFCgetr;
```

```
;---------------------------------------------------------------
; EFCn - perform XFER to destination specified by external link n
;---------------------------------------------------------------
; !1,1,EFCr; implicit in EFCr's return number (23B)

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;


;---------------------------------------------------------------
; EFCB - perform XFER to destination specified by external link 'alpha'
;---------------------------------------------------------------
!1,1,EFCdoGetlink;                                          shake B/A dispatch (Getalpha)

EFCB:           IR←sr23, :Getalpha;                         fetch link number
EFCr:           L←0-T-1, TASK, :EFCdoGetlink;               L:-(link number+1)

EFCdoGetlink:   temp←L, :Getlink;                           stash index for Getlink
EFCgetr:        IR←sr1, :SFCr;                              for Savpcinframe; no branch


;---------------------------------------------------------------
; SFC - Stack Function Call (using descriptor on top of stack)
;---------------------------------------------------------------

SFC:            IR←sr1, :Popsub;                            get dest link for xfer
;                                                           now assume IR still has sr1
SFCr:           mx←L, :Savpcinframe;                        set dest link, return to Xfer
```

```
;---------------------------------------------------------------
; 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;                     fetch alpha
KFCr:           IR←avm1, T←avm1+T+1, :KFCx;             DISP must be non zero
KFCx:           MAR←sdoffset+T, :Fetchlink;             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;                                   can't TASK, T has trap index
                OTPreg←L, :Mtrap;

Mtrap:          T←avm1+T+1;
                MAR←sdoffset+T;                         fetch dest link for trap
                NOP;
Mtrapa:         L←MD, TASK;                             (enter here from PORTO)
                mx←L, :Xfer;
```

```
;-----------------------------------------------------------------
; LFCn - call local procedure n (i.e. within same global frame)
;-----------------------------------------------------------------
!1,1,LFCx;                                                      shake B/A dispatch

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.
;---------------------------------------------------------------
!1,1,RETx;                                                      shake B/A branch

RET:            T←lp, :RETx;                                    local pointer

RETx:           IR←2, :CheckXferTrap;
RETxr:          MAR←nretlinkoffset+T;                           get previous local frame
                L←nlpoffset+T+1;
                frame←L;                                        stash for 'Free'
                L←MD;                                           pick up prev frame pointer
                mx←L, L←0, IR←msr0, TASK;                       mx points to caller
                my←L, :FreeSub;                                 clear my and go free frame
RETr:           T←mx, :Xfers;                                   xfer back to caller


;---------------------------------------------------------------
; 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;                                     address of local 0
                T←ib;
                L←mx-T, TASK;                                   L: mx-alpha
                MD←M, :nextA;                                   local 0 ← 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
```

```
;---------------------------------------------------------------
; P o r t   O p e r a t i o n s
;---------------------------------------------------------------


;---------------------------------------------------------------
; 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;                                          frame addr to word 0 of PORT
                MAR←M+1;                                        second word of PORT
                my←L, :Mtrapa;                                  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;                                       store it as second word
                TASK, :PORTIz;
PORTIz:         MD←my, :next;                                   store my or zero
```

```
;---------------------------------------------------------------
; S t a t e   S w i t c h i n g
;---------------------------------------------------------------


;---------------------------------------------------------------
; 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;                                        i.e. T←-11
                L←lp, :Savsuba;
Sav11r:         L←stkp, :Savsub;
Sav10r:         T←stkp+1;
                L←-7+T;                                         check if stkp > 5 or negative
                L←0+T+1, ALUCY;                                 L:stkp+2
                temp2←L, L←0-T, :Savok;                         L:-stkp-1

Savmax:         T←-7;                                           stkp > 5  => save all
                L←stk7, :Savsuba;

Savok:          SINK←temp2, BUS;                                stkp < 6 => save to stkp+2
                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;                                 return to caller
                T←-12, :DSTr1;                                  (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;                                 dispatch on pos. value
                SINK←M, BUS, TASK;
                MD←temp2, :Sav0r;
```

```
;---------------------------------------------------------------
; 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[stkp+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.
;---------------------------------------------------------------
!17,20,Lsr0,Lsr1,Lsr2,Lsr3,Lsr4,Lsr5,Lsr6,Lsr7,Lsr10,Lsr11,Lsr12,,,,,;
!1,2,Lsmax,Ldsuba;
!1,2,Lsr,BITBLTdoner;

Loadstate:      temp←L, IR←msr0, :NovaIntrOn;                   stash pointer
Lsr:            T←12, :Ldsuba;
Lsr12:          my←L, :Ldsub;
Lsr11:          mx←L, :Ldsub;
Lsr10:          stkp←L;
                T←stkp;                                         check for BRK resumption
                L←177400 AND T;                                 (i.e. bytecode in stkp)
                brkbyte←L LCY 8;                                stash for Xfer
                L←T←17.T;                                       mask to 4 bits
                L←-7+T;                                         check stkp > 6
                L←T, SH<0;
                stkp←L, T←0+T+1, :Lsmax;                        T:stkp+1
Lsmax:          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;
Ldsuba:         MAR←temp+T;
                L←ALLONES+T;                                    decr count for next time
                count←L, L←T;                                   use old value for dispatch
                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;                                           get alpha
                T←lp+T+1;
                L←nlpoffset1+T+1, TASK;                         L:lp-lpoffset+alpha
                temp←L, IR←ret0, :Savestatea;
DSTr1:          L←my, :Savsuba;                                 save my too!
DSTr:           temp←L, L←0, TASK, BUS=0, :Setstkp;             zap stkp, return to 'nextA'


;-----------------------------------------------------------------
; 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;                                           make Savpcinframe happy
                IR←sr4, :Savpcinframe;                          returns to LSTr
LSTr:           T←temp;                                         get alpha back
                L←lp+T, TASK, :Loadstate;                       lp already undiddled


;-----------------------------------------------------------------
; 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←lpoffset;
                L←lp-T, TASK;                                   compute frame base
                frame←L;
                IR←sr2, :FreeSub;

LSTFr:          T←frame;                                        set up by FreeSub
                L←ib+T, TASK, :Loadstate;                       get state from dead frame
```

```
;------------------------------------------------------------
; E m u l a t o r   A c c e s s
;------------------------------------------------------------


;------------------------------------------------------------
; 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;
WRr:            SINK←ib, BUS;                                   dispatch on alpha
WR0:            TASK, :WR0;

WR1:            wdc←L, :nextA;
WR2:            XTSreg←L, :nextA;


;------------------------------------------------------------
; JRAM - JMPRAM for Mesa programs (when emulator is in ROM1)
;------------------------------------------------------------

JRAM:           L←ret2, TASK, :Xpopsub;
JRAMr:          SINK←M, BUS, SWMODE, :next;                     BUS applied to 'nextBa' (=0)
```

```
;---------------------------------------------------------------
; P r o c e s s / M o n i t o r  S u p p o r t
;---------------------------------------------------------------
!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,FastMRExx;                                         drop ball 1
!1,1,FastEExx;                                          drop ball 1
!7,1,FastEExx;                                          shake IR←isME/isMXD
!1,2,MXDr,MEr;
!7,1,FastEExxx;                                         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;                                  IDISP:1, DISP:1, mACSOURCE:1
$isMRE          $65403;                                 IDISP:13, DISP:3, mACSOURCE:16
$isMXD          $402;                                   IDISP:0, DISP:2, mACSOURCE:0

ME:             IR←isME, :FastEEx;                      indicate ME instruction
MXD:            IR←isMXD, :FastEEx;                     indicate MXD instruction

MRE:            MAR←HardMRE, :FastMREx;                 <HardMRE> ~= 0 => do Nova code
FastMREx:       IR←isMRE, :MXDr;                        indicate MRE instruction

FastEEx:        MAR←stk0, IDISP, :FastEExx;             fetch monitor lock
FastEExx:       T←100000, :MXDr;                        value of unlocked monitor lock

MXDr:           L←MD, mACSOURCE, :FastEExxx;            L:0 if locked (or queue empty)
MEr:            L←MD-T, mACSOURCE, :FastEExxx;          L:0 if unlocked

FastEExxx:      MAR←stk0, SH=0, :MXDrr;                 start store, test lock state

; 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;                 L:100000, T:0 (stkp value)
MErr:           T←0+1, :FastEEtrap1;                    L:0, T:1 (stkp value)
MRErr:          L←0+1, TASK, :FastEEtrap2;              L:1 (stkp value)

MEXDdone:       MD←M, L←T, TASK, :Setstkp;

MREdone:        stkp←L, :ME;                            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;                                   if you uncomment this, don't
;               AC3←L;                                          forget the pre-def above!
MoveParms3:     L←stk2, TASK;
FastEEtrap2:    AC2←L;                                          (enter here from MRE)
MoveParms2:     L←stk1, TASK;
FastEEtrap1:    AC1←L;                                          (enter here from ME/MXD)
MoveParms1:     L←stk0, TASK;
                AC0←L;

                L←0, TASK;                                      indicate stack empty
                stkp←L;
                T←DISP+1, :STOP;
```

```
;-------------------------------------------------------------
; M i s c e l l a n e o u s  O p e r a t i o n s
;-------------------------------------------------------------


;-------------------------------------------------------------
; 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←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
;                                                               throws away alpha for now
MISCr:          L←CLOCKLOC-1, IR←CLOCKLOC, :Dpushb;             IR← causes branch 1!
;                                                               (and mACSOURCE 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;                                                  shakes entry B/A branch
!1,2,BLTintpend,BLTloop;
!1,2,BLTnoint,BLTint;
!1,2,BLTmore,BLTdone;
!1,2,BLTeven,BLTodd;
!1,1,BLTintx;                                               shake branch from BLTloop

BLT:            stk7←L, L←T, TASK, :BLTx;                   stk7=0 <=> branch pending
BLTx:           temp←L, :BLTloop;                           stash source offset (+1)

BLTloop:        L←T←stk1-1, BUS=0, :BLTnoint;
BLTnoint:       stk1←L, L←BUS AND ~T, :BLTmore;             L←0 on last iteration (value
;                                                           on bus is irrelevant, since T
;                                                           will be -1).

BLTmore:        T←temp-1;                                   T:source offset (0 or cp)
                MAR←stk0+T;                                 start source fetch
                L←stk0+1;
                stk0←L;                                     update source pointer
                L←stk2+1;
                T←MD;                                       source data
                MAR←stk2;                                   start dest. write
                stk2←L, L←T;                                update dest. pointer
                SINK←NWW, BUS=0, TASK;                      check pending interrupts
                MD←M, :BLTintpend;                          loop or check further

BLTintpend:     SINK←wdc, BUS=0, :BLTloop;                  check if interrupts enabled

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

BLTint:         SINK←stk7, BUS=0, :BLTintx;                 test even/odd pc
BLTintx:        L←mpc-1, :BLTeven;                          prepare to back up

BLTeven:        mpc←L, L←0, TASK, :BLTodd;                  even - back up pc, clear ib
BLTodd:         ib←L, :Intstop;                             odd - set ib non-zero

;       BLT completed

BLTdone:        SINK←stk7, BUS=0, TASK, :Setstkp;           stk7=0 => return to 'nextA'


;---------------------------------------------------------------
; 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, :BLTCx;                             if BLT were odd, we could use:
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,BITBLTx;                                                   shake B/A dispatch
!7,1,DoBITBLTx;                                                 shake IR← dispatch
!3,4,Mstop,,NovaIntrOff,DoBITBLT;                               includes NovaIntrOff returns

BITBLT:         stk7←L, :BITBLTx;                               save even/odd across ROM call
BITBLTx:        L←stk0, TASK;
                AC2←L;                                          stash descriptor table
                L←stk1, TASK;
                AC1←L;
                SINK←wdc, BUS=0;                                check if Mesa interrupts off
                IR←sr3, :NovaIntrOff;                           if so, shut off Nova's
DoBITBLT:       L←BITBLTret, SWMODE, :DoBITBLTx;                get return address
DoBITBLTx:      PC←L, L←0, :ROMBITBLT;                          L←0 for Alto II ROM0 "feature"

BITBLTdone:     IR←sr1, :NovaIntrOn;                            ensure Nova interrupts are on
BITBLTdoner:    brkbyte←L, BUS=0, TASK, :Setstkp;               don't bother to validate stkp

BITBLTintr:     L←AC1, TASK;                                    pick up intermediate state
                stk1←L, :BLTint;                                stash instruction state
```

```
;-------------------------------------------------------------
; M e s a / N o v a   C o m m u n i c a t i o n
;-------------------------------------------------------------


;-------------------------------------------------------------
; Subroutines to Enable/Disable Nova Interrupts
;-------------------------------------------------------------
; !3,4,Mstop,,NovaIntrOff,DoBITBLT;                          appears with BITBLT
; !1,2,Lsr,BITBLTdoner;                                      appears with LoadState
!7,1,NovaIntrOffx;                                           shake IR← dispatch

NovaIntrOff:    T←100000;                                    disable bit
NovaIntrOffx:   L←NWW OR T, TASK, IDISP;                     turn it on, dispatch return
                NWW←L, :Mstop;

NovaIntrOn:     T←100000;                                    disable bit
                L←NWW AND NOT T, IDISP;                      turn it off, dispatch return
                NWW←L, L←0, :Lsr;


;-------------------------------------------------------------
; IWDC - Increment Wakeup Disable Counter (disable interrupts)
;-------------------------------------------------------------
!1,2,IDnz,IDz;

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


;-------------------------------------------------------------
; DWDC - Decrement Wakeup Disable Counter (enable interrupts)
;-------------------------------------------------------------
!1,1,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, :nextAdeaf;


;-------------------------------------------------------------
; Entry to Mesa Emulation
;       AC0 holds address of current process state block
;       Location 'PSBloc' is assumed to hold the same value
;-------------------------------------------------------------

Mgo:            L←AC0, :Loadstate;
```

```
;-------------------------------------------------------------
;  N o v a   I n t e r f a c e
;-------------------------------------------------------------
$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 ROM0 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, :Savpcinframe;                          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 ...
```

```
; Mu Float.mu
; Copywrite Xerox Corporation 1980
; Horizontal bit blit.
; Updated by Bob Lyon on November 2, 1979  10:48 AM
; Updated by LStewart on May 14, 1980  7:33 PM
; 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.
; If this is used with the floating point package,
; put this file after the floating point entry predefs.
; That is, put "#HBlt.mu;" after the ram entry vector in file
; Float.mu.
;
$ROMMUL $L004120,0,0;   MUL routine address (120B) in ROM0
; shiftable register declarations
$MRMC   $600;
; shiftable register declarations
$Temp   $R1;
$OpCode $R2;
; non-shiftable register declarations
$flags  $R35;
$dbca   $R36;
$gray   $R55;
$Smask  $R57;
$x1     $R56;
$x2     $R41;
$Mflag  $R45;
$HoldPC $R44;
$addr   $R47;
$endAddr $R51;
$AllOnes $R46;
$savBnk $R43;
$BnkAddr $R42;
; 3 and 50 should be available
; Move seven parameters from memory to S registers
; and claculate deltax, deltay, and init point P to point A
; %1,1777,550,HBlt;             HBlt is the entry point.
%7,17,0, Got0,, Got2,, Got4,, Got6;  Return locs for mem reads.
!1,2, Punt, Read;
!1,2, Step4, Step3;
HBlt:
        T ← ALLONES;
        MAR ← L ← 37 XOR T;
        BnkAddr ← L, L ← T;
        AllOnes ← L;
        L ← MD, TASK;
        savBnk ← L;

        T ← 2, :Read;           -- read dbmr and y
Got2:   T ← MD;
        AC2 ← L, L ← T;
        AC1 ← L, L ← T ← 0;     -- get ready for (y*dbmr)
;
        AC0 ← L, :Read;         -- read flags and dbca
Got0:   T ← MD;
        flags ← L, L ← T, TASK;
        dbca ← L;
;
        T ← 4, :Read;           -- read x1 and x2
Got4:   T ← MD;
        x1 ← L, L ← T;
        x2 ← L;
        L ← x1 - T - 1;
        T ← 6, SH<0;
;
        :Punt;                  -- IF (x1 <= x2) goto read
Got6:   gray ← L, :GetAddr;
;
; -- Read memory subroutine
Read:
        MAR ← stk0 + T;
        L ← T;
        SINK ← M, BUS;
        L ← MD, :Got0;          -- return to Got[0-6]
;
```

```
;
; Compute the effective word begin & end address
; addr ← dbca + (y*dbmr) + x1/16;
; endAddr ← dbca + (y*dbmr) + x2/16;
; %1,1777,177,MULret;
GetAddr:
        L ← PC, TASK;
        HoldPC ← L;
        T ← ALLONES;
        L ← MRMC XOR T, SWMODE;
        PC ← L, :ROMMUL;
MULret: L ← HoldPC, TASK;               -- restore the return addr
        PC ← L;
        T ← dbca;       -- result is in AC1
        L ← AC1 + T;
        T ← M;
;
        L ← x1;
        Temp ← L RSH 1;
        L ← Temp;
        Temp ← L RSH 1;
        L ← Temp;
        Temp ← L RSH 1;
        L ← Temp;
        Temp ← L RSH 1;
        L ← Temp + T;
        addr ← L;
;
        L ← x2;
        Temp ← L RSH 1;
        L ← Temp;
        Temp ← L RSH 1;
        L ← Temp;
        Temp ← L RSH 1;
        L ← Temp;
        Temp ← L RSH 1;
        L ← Temp + T, TASK;
        endAddr ← L;
;
        T ← 0 + 1;
        L ← flags AND T;
        Mflag ← L;
        T ← 7;
        L ← flags AND T;
        OpCode ← L RSH 1;

!1,2,UseXHB,Step1;
; -- If not HBlt then alter memory bank
        L ← stkp - 1;
        T ← 3, SH=0, :UseXHB;
;
UseXHB: L ← savBnk AND NOT T;
        T ← stk1 . T;
        MAR ← BnkAddr;
        L ← M OR T;
        MD ← 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 ← 17;                 -- SINK ← x1 AND 17
        L ← x1 AND T;
        SINK ← M, BUS;
        T ← endAddr, :LftEnd0;
LftEnd0:        L ← ALLONES, :ContLE;
LftEnd1:        L ← 77777,      :ContLE;
```

```
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, Stp1to3;
%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;
Stp1to3: 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;
; -- Step 3 - Draw the furthest right hand side word of the
; -- 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 ← gray . T;           -- dest ← (NOT gray) AND dest
        L ← AllOnes XOR T;
        T ← MD;
        L ← M AND T, :ContBd2;
;
ContBd2: SINK ← Mflag, TASK, BUS;
        Temp ← L, :mWrite;
mWrite: MAR  ← addr, :Step2;
XmWrite: XMAR ← addr, :Step2;
;
Step2:
        T ← endAddr-1;
        L ← addr - T;
        MD ← Temp;
        L ← addr + 1, SH<0;
        addr ← L, :ContSt2;     -- IF(addr<endAddr) goto ContSt3
ContSt2: L ← addr - T - 1;
        SH=0;
        T ← 17, :Step4;         -- IF(addr=endAddr) goto Step3
;
; -- Step 4 is where we return to mesa.
Punt:   T ← 0;                  -- NOP, premature exit
Step4:
        MAR ← BnkAddr;
        L ← 0;
        MD ← savBnk, TASK;
        stkp ← L;
        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;

!1,2,PCMayI,PCNoI;
!1,2,PCDisI,PCDoI;
!1,2,PCOkCy,PCZCy;
!1,2,PCNoCy,PCCy;
!1,2,PCLoop,PCDone;
!1,2,PCNoMZ,PCMinZ;
!1,1,PCDoI1;

PupChecksum:
        L← stk0;                        Must keep partial sum in R reg (not S)
        temp← L;
        MAR← L← stk1, :PCLp1;           Start fetch of first word

; 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.
PCLoop: MAR← L← stk1+1;                 Start fetch of next word
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: L← stk2-1, ALUCY;               Test for carry out, decrement count
PCLp2:  stk2← L, :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
PCDn1:  stkp← L, L← 0, :PCNoMZ;         [PCNoMZ, PCMinZ]
PCNoMZ: L← temp, TASK, :PCGoEm;
PCMinZ: TASK;                           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] Back up mpc, squash BUS=0
PCDoI1: mpc← L, L← 0, TASK;
        ib← L;                          ib← 0
        L← stkp+1, :PCDn1;              Push Ram address back onto stack

Emulator:
        SWMODE;                         Switch to Rom1
        :romnextA;                              Mesa emulator entry point
```

```
;-------------------------------------------------------------
; Float.mu -- 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
;  bytecode

; R registers
$mSAD    $R1;    mx
$N2      $R2;    saveret
$M2      $R3;    newfield
$M1      $R5;    count

$Arg1    $R7;    taskhole

$N1      $R35;   temp

$Arg0    $R36;   temp2
$ShiftCount      $R36;   entry --used only in add/sub

; S registers
$Mode    $R41;   mask --used only in add/sub
$S1      $R43;   unused2  --sign
$Mxreg   $R44;   alpha
$Arg2    $R50;   unused3
$E1      $R55;   ATPreg   --exponent
$S2      $R73;   was OTPreg
$E2      $R57;   XTPreg

$Sticky $R72;   Sticky flag
;  The sticky bit for inexact result is implemented as follows:  The sign
;  bit indicates whether trap on inexact result is enabled, the LSB is the
;  actual sticky bit.
;$OTPreg         $R56;   instruction (saveinst)
$savestkp        $R42;   spot to remember stkp (unused1)
$SubRet $R74;   subroutine return address

$LastL   $R40;   M register

!1,2,MiscOK1,MiscSmall;
!1,2,MiscOK2,MiscBig;
$177757 $177757;         -21B

; Hopefully, Alpha is IN [20B..31B]  (the range for FP)
; There are other FP instructions, but they all trap
MISC:   L←T,TASK;
        OTPreg←L;        store T (Alpha) in saveinst (OTPreg)
        L←stkp,TASK;
        savestkp←L;      store stack pointer in savestkp
        T←177757;        T←-21B
        L←T←OTPreg+T+1; L IN [0..11B] if FP instr.
        L←15-T,SH<0;
        L←T,SH<0,TASK,:MiscOK1; !1,2,MiscOK1,MiscSmall;
MiscOK1:         Arg0←L,:MiscOK2;         !1,2,MiscOK2,MiscBig;
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 BEGIN
```

```
;   Mopcodes.zMISC, MiscAlpha;
;   END;

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


!1,2,LowNZero1,LowZero1;        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:        L←Arg0;
        SINK←stkp,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:        stk6←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
```

```
$romOOR$L004405,0,0;    secret definition

; 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 SDIndex) (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,NOADDIER,ADDIER;
!1,2,NOSPILL,SPILL;
!1,2,NOADDX,ADDX;
!1,2,NOSPILLX,SPILLX;

ramMul: L←Arg2-1, BUS=0;        !7,1,ramMulA;
ramMulA:        mSAD←L,L←0,:DOMUL;      !1,2,DOMUL,NOMUL;
DOMUL:  TASK,L←-10+1;
        Mxreg←L;
MPYL:   L←Arg1,BUSODD;
        T←Arg0,:NOADDIER;       !1,2,NOADDIER,ADDIER;
NOADDIER:
        Arg1←L MRSH 1,L←T,T←0,:NOSPILL; !1,2,NOSPILL,SPILL;
ADDIER: L←T←mSAD+INCT;
        L←Arg1,ALUCY,:NOADDIER;
SPILL:  T←ONE;
NOSPILL:        Arg0←L MRSH 1;
        L←Arg1,BUSODD;
        T←Arg0,:NOADDX; !1,2,NOADDX,ADDX;
NOADDX: Arg1←L MRSH 1,L←T,T←0,:NOSPILLX;        !1,2,NOSPILLX,SPILLX;
ADDX:   L←T←mSAD+INCT;
        L←Arg1,ALUCY,:NOADDX;
SPILLX: T←ONE;
NOSPILLX:       Arg0←L MRSH 1;
        L←Mxreg+1,BUS=0,TASK;
        Mxreg←L,:MPYL;  !1,2,MPYL,MPYA;
NOMUL:  T←Arg0;
        Arg0←L,L←T,IDISP,TASK;
        Arg1←L,:MulRet;
MPYA:   IDISP,TASK;
        NOP,:MulRet;

;---------------------------------------------------------------
;DPopSub:  Pop TOS and NOS into L,T
;---------------------------------------------------------------

!7,10,DPopRet,DPopRet1,DPopRet2;
!7,1,DPopA;             shake IR← dispatch
!17,20,LRpop0,LRpop1,LRpop2,LRpop3,LRpop4,LRpop5,LRpop6,LRpop7,LRpop8,,,,,,,;

; NO TASK in the instruction getting here
DPop:   T←2;    !7,1,DPopA;
DPopA:  L←stkp-T,BUS,TASK;
        stkp←L,:LRpop0;

LRpop0: NOP,TASK,:RamStkErr;    stkp=0!
```

```
LRpop1: NOP,TASK,:RamStkErr;    stkp=1!
LRpop2: L←stk1;
        T←stk0,IDISP,:LRpopCom;
LRpop3: L←stk2;
        T←stk1,IDISP,:LRpopCom;
LRpop4: L←stk3;
        T←stk2,IDISP,:LRpopCom;
LRpop5: L←stk4;
        T←stk3,IDISP,:LRpopCom;
LRpop6: L←stk5;
        T←stk4,IDISP,:LRpopCom;
LRpop7: L←stk6;
        T←stk5,IDISP,:LRpopCom;
LRpop8: L←stk7;
        T←stk6,IDISP,:LRpopCom;
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.

!17,20,LoadRet,LoadRet1,LoadRet2,LoadRet3,LoadRet4,LoadRet5,LoadRet6,LoadRet7,LoadRet10,LoadRet11;
!1,2,UPPos,UPNeg;
!1,2,PVbSign,PVSign;
!1,2,UPR1b,UPR1;
!1,2,UPR2b,UPR2;
!1,2,UPR4b,UPR4;
!1,2,UPBias,UPNoBias;
!1,1,UPBias1;
!1,2,PVR6b,PVR6;
!1,2,PVbCom,PVbNaN;
!1,2,PVR5b,PVR5;
!1,2,UPDN,UPZeroT;
!1,2,PVR7b,PVR7;
!1,2,PVbDNb,PVbZero;

; TASK in the instruction getting here
LoadArgs:       SubRet←L;       save return address
        IR←sr0,:DPop;
DPopRet:        Arg0←L,L←T,IDISP,:UPPos;        !1,2,UPPos,UPNeg;

UPPos:  Arg1←L,L←0,TASK,:PVbSign;       !1,2,PVbSign,PVSign;
UPNeg:  Arg1←L,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;
```

```
        Arg0←L LSH 1,SH=0;      exponent left justified now
        L←Arg0,IDISP,:UPBias;   !1,2,UPBias,UPNoBias;
UPBias: Arg0←L LCY 8,:UPBias1;  exponent right justified now
;               !1,1,UPBias1;
UPBias1:        T←377;
        L←Arg0-T,IDISP; check for exp=377
        T←177,SH=0,:PVR6b;      !1,2,PVR6b,PVR6;
PVR6b:  L←Arg0-T,TASK,:PVbCom;  !1,2,PVbCom,PVbNaN;
PVbCom: E2←L,:PackedVectora;

PVbNaN: NOP,:FPTrap;    Instruction after TASK!

; Test if mantissa was zero, if so then true zero, else denormalized
UPNoBias:       T←100000,:PVR5b;        !1,2,PVR5b,PVR5;

PVR5b:  SINK←N2,BUS=0;
        L ← M2 XOR T,IDISP,:UPDN;       !1,2,UPDN,UPZeroT;

UPDN:   NOP,TASK,:PVbNaN;       ; !1,1,PVbNaN;
UPZeroT:        NOP, SH=0,:PVR7b;       !1,2,PVR7b,PVR7;

PVR7b:  M2←L,:PVbDNb;   !1,2,PVbDNb,PVbZero;
PVbDNb: NOP,TASK,:FPTrap;
PVbZero:        L←E2,TASK,:PVbCom;      don't diddle sign
;---------------------------------------------------------------
; now unpack TOS
;---------------------------------------------------------------
!1,2,LRET,PVNaN;
!1,2,PVDNb,PVZero;

; Cannot be TASK in instruction coming here
PackedVectora: IR←sr1,:DPop;
DPopRet1:       Arg0←L,L←T,IDISP,:UPPos;        ; !1,2,UPPos,UPNeg;

PVSign: S1←L,:UPC1;

UPR1:   N1←L LCY 8,:UPC2;       Store in left half word

UPR2:   M1←L LCY 8;     Store in left half word

        T←100000;       hidden bit
        T←M1 OR T,IDISP,:UPC4;  high 7 bits of mantissa

UPR4:   M1←L,:UPC5;     mantissa finished

PVR6:   L←Arg0-T,TASK,:LRET;    !1,2,LRET,PVNaN;

PVR5:   SINK←N1,BUS=0;
        L ← M1 XOR T,IDISP,:UPDN;       ; !1,2,UPDN,UPZeroT;
PVR7:   M1←L,:PVDNb;    !1,2,PVDNb,PVZero;
PVDNb:  NOP,TASK,:FPTrap;       Denormalized number
PVZero: L←E1,TASK,:LRET;        true zero

PVNaN:  NOP,:FPTrap;    Instruction after TASK!

LRET:   E1←L;
        SINK←SubRet,BUS,TASK;   ;and, the big return
        NOP,:LoadRet;   [LoadRet,LoadRet1,LoadRet2,LoadRet3]

;---------------------------------------------------------------
;repack into Arg0,,Arg1, push and return
;---------------------------------------------------------------
!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,IRNoTrap,IRTrap;
!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;    NO pending branch because SH=0!
;               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;
RPlus1: L←N1+T+1,:RoundPlus;
RPlus0: L←M1 AND T,TASK,:NoRound1;

FSTR:   T←400;
        L←N1+T;
RoundPlus:      N1←L,ALUCY;
        L←M1+1,:FSTNoR2;        !1,2,FSTNoR2,FSTR2;
FSTR2:  M1←L,ALUCY,TASK,:FSTNoR2;
FSTNoR2:        NOP,:FSTNoSh;   !1,2,FSTNoSh,FSTSh;
FSTSh:  L←T←M1; low order
        M1←L RSH 1;
        L←N1,TASK;
        N1←L MRSH 1;
        L←E1+1,TASK;
        E1←L;
FSTNoSh:        T←0+1;  sticky bit for inexact result
        L←Sticky OR T;
        Sticky←L,SH<0;
        T←377,:IRNoTrap;        !1,2,IRNoTrap,IRTrap;
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;
        L←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 M!, to r.j. mantissa
        M1←L LCY 8,:NoExpOV;    !1,2,NoExpOV,ExpOV;

ExpUF:  NOP,TASK,:FPTrap;
ExpOV:  NOP,TASK,:FPTrap;

; If we get herem, 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,:FSTSgn;   ready for FPdpush
; at this point, M1,,N1 has everything but sign

; Set sign bit

FSTSgn: SINK←S1,BUS=0;
        L←T←Arg0,:FSTNeg;       !1,2,FSTNeg,FSTPos;
```

```
FSTNeg: L←100000 OR T;
FSTPos: Arg0 ← L,:FPdpush;

;---------------------------------------------------------------
;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←N1,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←N1,SH<0;
        M1←L MLSH 1,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←L MRSH 1,:RePack;

;---------------------------------------------------------------
;  Remainder:  not implemented
;---------------------------------------------------------------

FRem:   NOP,TASK,:FPTrap;

;---------------------------------------------------------------
;  Round to Integer
;---------------------------------------------------------------
!1,2,FRIEPlus,FRIENeg;
!1,2,FRIEOK,FRIEOv;
!1,2,FRINStik,FRIStik;
!1,2,FRIShift,FRIDone;
!1,2,FRINE,FRIPlus1A;
!1,2,FRINPlus1,FRIPlus1;
!1,2,FRIPNOv,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 < 15 decimal
FRIShift:       L←T←M1,TASK,:FRIEOK;    !1,2,FRIEOK,FRIEOv;
FRIEOK: M1←L RSH 1;
        L←N1,TASK,BUSODD;
        N1←L MRSH1,:FRINStik;   !1,2,FRINStik,FRIStik;
FRINStik:       L←E1-1,BUS=0,TASK;
        E1←L,:FRIShift; !1,2,FRIShift,FRIDone;

FRIStik:        T←0+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
FRIPlus1:       M1←L,SH<0,:FRIPlus1A;   ; !1,1,FRIPlus1A;
FRIPlus1A:      SINK←S1,BUS=0,:FRIPNOv; ; !1,2,FRIPNOv,FRIPOv;
FRIPNOv:        L←T←M1,:FixINeg;        !1,2,FixINeg,FixIPos;

FRIENeg:        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
FRIEOv: NOP,TASK,:FPTrap;       FixIExponentOverflow (trap)
FRIPOv: NOP,:MiscBig;   ; !1,1,MiscBig;

;-------------------------------------------------------------
;  Round to Long Integer
;-------------------------------------------------------------
!1,2,FRgem1,FRlsm1;
!1,2,FRls31,FRge31;
!1,2,FRle29,FR30;
!1,2,FRNext,FRDone;
!1,2,FRLoop,FRStik;
!3,4,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

FRound: L←11,TASK,:SavePVA;
LoadRet11:      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;
FRls31: S2←L,:FRle29;   !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;
        N1←L MRSH 1,:FRLoop;    !1,2,FRLoop,FRStik;

FRStik: T←0+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 MRSH 1,: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;

FRD303: N1←L,ALUCY;
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;

FRlsm1: 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,FRCEOK,FRCEOv;
!1,2,FRCMore,FRCDone;
!1,2,FRCNStik,FRCStik;
!1,2,FRCNE,FRCPlus1A;
!1,2,FRCNPlus1,FRCPlus1;
!1,2,FRCPNOv,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,FRCShift,FRCENeg;

;               E1 must be < 15 decimal
FRCShift:       L←E1-1,:FRCEOK; !1,2,FRCEOK,FRCEOv;
FRCEOK: E1←L,SH<0;
        L←T←M1,:FRCMore;        !1,2,FRCMore,FRCDone;
FRCMore:        M1←L RSH 1;
        L←N1,TASK,BUSODD;
        N1←L MRSH1,:FRCNStik;   !1,2,FRCNStik,FRCStik;
FRCNStik:       L←E1-1,:FRCEOK;

FRCStik:        T←0+1;
        L←N1 OR T,TASK;
        N1←L,:FRCNStik;

FRCDone:        T←N1,BUS=0;
        L←100000-T,:FRCNE;      !1,2,FRCNE,FRCPlus1A;
FRCNE:  NOP,SH<0;
        L←M1+1,SH=0,:FRCNPlus1; !1,2,FRCNPlus1,FRCPlus1;

; 100000 bit might have been on, SH=0 will branch if so
FRCNPlus1:      L←M1,:FRCPNOv;  !1,2,FRCPNOv,FRCPOv;
; the ALUCY branch will branch on overflow
FRCPlus1:       M1←L,ALUCY,:FRCPlus1A;  !1,1,FRCPlus1A;
FRCPlus1A:      L←M1,:FRCPNOv;  ; !1,2,FRCPNOv,FRCPOv;
FRCPNOv:        Arg0←L,:ShortRet;

FRCENeg:        L←0,TASK,:FCRet;        store 0 and return
                ; !1,1,FCRet;

FRCVNeg:        NOP,TASK,:FPTrap;
FRCPOv: NOP,:MiscBig;   !1,1,MiscBig;
```

```
FRCEOv: NOP,TASK,:FPTrap;


;---------------------------------------------------------------
;  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←L,SH<0,:FixEPlus;    !1,2,FixEPlus,FixENeg;

FixEPlus:       L←T←M1,:FixEOK; E1 must be < 31 decimal
FixShift:       L←T←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←T←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,:FPdpush;

FixENeg:        L←0;    !1,1,FixENeg1;
FixENeg1:       Arg1←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 < 15 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;       FixCExponentOverflow (trap)
FixCVNeg:       NOP,TASK,:FPTrap;       FixCValueNegative (trap)

;---------------------------------------------------------------
;  FixI  Fix to INTEGER
;---------------------------------------------------------------
!1,2,FixIEPlus,FixIENeg;
```

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

FFixI:  L←5,TASK,:SavePVA;      (see FFix)
LoadRet5:       L←T←E1;
        L←17-T-1,SH<0;  E1 must be positive
        E1←L,SH<0,: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;        !1,2,FixINeg,FixIPos;

FixIPos:        NOP,TASK,:FixCDone;
FixINeg:        L←0-T,TASK,:FixCDone;

FixIENeg:       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
FixIEOv:        NOP,TASK,:FPTrap;       FixIExponentOverflow (trap)

;-------------------------------------------------------------
;Mul: floating point multiply
;-------------------------------------------------------------
!1,2,MulNZero,MulZero;
!1,2,MulNZero1,MulZero1;
!1,2,FMNoCy,FMCy;
!1,2,MulNoCry,MulCry;
!1,2,FMNoCy1,FMCy1;
!1,2,FMNoCy2,FMCy2;
!1,2,FMLL,FMNoLL;
!1,2,MulNorm,MulNoNorm;
!1,2,MulNZero2,MulZero2;

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,:MulNZero;       !1,2,MulNZero,MulZero;
MulNZero:       L←N2;
        Arg1←L,L←0,TASK,:MulZeroa;
MulZero:        L←0,:LowZero1;  return 0
MulZeroa:       Arg0←L;
        IR←sr0,:ramMul;
MulRet: L←Arg0,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,:MulNZero1;      !1,2,MulNZero1,MulZero1;
MulNZero1:      L←N1;   second multiply: other high*other low
        Arg1←L,L←0,TASK,:MulZero1a;
MulZero1:       L←0,:LowZero1;
MulZero1a:      Arg0←L;
        IR←sr1,:ramMul;
MulRet1:        T←Arg1;
```

```
        L←E2+T; add results, set carry if overflow
        E2←L,ALUCY;
        T←Arg0,:FMNoCy; !1,2,FMNoCy,FMCy;
FMNoCy: L←S2+T,:FMCyS;
FMCy:   L←S2+T+1,:FMCyS;
FMCyS:  S2←L,ALUCY;
        L←0,:MulNoCry;  !1,2,MulNoCry,MulCry;
MulCry: L←ONE,TASK;

; Use SubRet to hold carry bit
MulNoCry:       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:  S2←L,ALUCY;
FMNoCy1:        L←SubRet+1,:FMNoCy2;    !1,2,FMNoCy2,FMCy2;
FMCy2:  SubRet←L;
FMNoCy2:        L←S2,TASK;
        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,FMNoLL;
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;      9
        N1←L LSH 1,:MulNZero2;  10  !1,2,MulNZero2,MulZero2;
MulNZero2:      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,DivL0NoCy,DivL0Cy;
!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←T←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 msw result

        L←30+1,TASK;    set E2 to NumLoops-1
        E2←L;
        T←N2,:DivSub;

DivAdd: L←Arg1+T;
        Arg1←L,ALUCY;
        T←M2,:DAddNoCy; !1,2,DAddNoCy,DAddCy;
DAddNoCy:       L←Arg0+T,:DOpCom;
DAddCy: L←Arg0+T+1,:DOpCom;

DivSub: L←Arg1-T;
        Arg1←L,ALUCY;
        T←M2,:DSubNoCy; !1,2,DSubNoCy,DSubCy;
DSubNoCy:       L←Arg0-T-1,:DOpCom;
DSubCy: 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←T←N1,:DivRes0;        !1,2,DivRes0,DivRes1;
DivRes1:        L←N1+T+1;
        N1←L,:DResCom;
DivRes0:        N1←L LSH 1,:DResCom;

DResCom:        L←M1,TASK;
        M1←L MLSH 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←T←Arg1;
        Arg1←L LSH 1;
        L←Arg0,TASK;
        Arg0←L MLSH 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;
        L←M1;
        mSAD←L LSH 1,SH<0;
        M1←L MLSH 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,,Arg1=0 Then EXACT
; If the last bit of result was a 0 AND Arg0,,Arg1=-M2,,N2 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;
```

```
DivLast0:       L←N2+T;
        NOP,ALUCY;
        T←Arg0,:DivL0NoCy;      !1,2,DivL0NoCy,DivL0Cy;
DivL0NoCy:      L←M2+T,SH=0,:DivLCom;
DivL0Cy:        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;
Sh:     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;
NoFix:  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←M1,: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←L 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←0+1;
        L←N2 OR T,TASK;
        N2←L,:Sh2NStik;

Fix1:   L←0;
        M2←L,L←0+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,A1NAddS,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 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;  !1,2,A1NoCry,A1Cry;
A1Cry:  L←M2+T+1,:A1Store;
A1NoCry:        L←M2+T;
A1Store:        M1←L,ALUCY,TASK;
        NOP,:A1xNoCry;  !1,2,A1xNoCry,A1xCry;
A1xCry: T←L←M1; post shift
        M1←L RSH 1;
        L←N1,TASK,BUSODD;
        N1←L MRSH 1,:A1NAddS;   !1,2,A1NAddS,A1AddS;
A1AddS: T←0+1;
        L←N1 OR T,TASK;
        N1←L;
A1NAddS:        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←0-T,:LowNZero;        !1,2,LowNZero,LowZero;
LowNZero:       N1←L,T←0-1;
        L←M1 XOR T,:A2Cx;       complement
LowZero:        T←M1;
        L←0-T;  negate (note that N1 is already 0, so no need to update it)
A2Cx:   M1←L,T←0-1;
        L←S1 XOR T,TASK;        complement sign
        S1←L;

A2NoSign:       L←0,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 LSH 1,T←0,:A2NoCryL;       !1,2,A2NoCryL,A2CryL;
A2CryL: T←ALLONES;
A2NoCryL:       L←M1;
        M1←L MLSH 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←0,:LowNZero1;    !1,2,LowNZero1,LowZero1;
LowNZero1:      N1←L;   zero out low order
        L←20,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;
LNZPos: T←0,:LNZNeg;
LNZNeg: Arg1←L,L←T,TASK;
        Arg0←L,:FPdpush;

;---------------------------------------------------------------
;  Compare
;---------------------------------------------------------------

!1,2,FCANZ,FCAZ;
!1,2,FCANZBNZ,FCANZBZ;
!1,2,FCSD,FCSS;
!1,2,FCEDiff,FCESame;
!1,2,FCSgnA,FCSgnB;
!1,1,FCESame1;
!1,2,FCMDiff,FCMSame;
!1,1,FCMSame1;
!1,2,FCNDiff,FCNSame;
;!1,1,FCRet;    used farther up
!1,2,FCAZBNZ,FCAZBZ;
!1,2,FCAlsB,FCAgrB;
!1,2,FCAgrB1,FCAlsB1;

FComp:  L←6,TASK,:LoadArgs;
LoadRet6:       SINK←M1,BUS=0;
        SINK←M2,BUS=0,TASK,:FCANZ;      !1,2,FCANZ,FCAZ;
FCANZ:  NOP,:FCANZBNZ;  !1,2,FCANZBNZ,FCANZBZ;
FCANZBZ:        SINK←S1,BUS=0,TASK,:FCAST;      Return according to sign of A
FCANZBNZ:       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;
        T←M1,SH=0;
```

```
        L←M2-T,SH<0,:FCEDiff;   !1,2,FCEDiff,FCESame;

FCEDiff:        NOP,:FCSgnA;    !1,2,FCSgnA,FCSgnB;
FCSgnA: SINK←S1,BUS=0,TASK,:FCAST;
FCSgnB: SINK←S2,BUS=0,TASK,:FCBST;

; In what follows, we do unsigned compares.  ALUCY will branch if
; a >= b on execution of a-b
FCESame:        T←N1,SH=0,:FCESame1;    !1,1,FCESame1;
FCESame1:       L←N2-T,ALUCY,:FCMDiff;  !1,2,FCMDiff,FCMSame;

FCMDiff:        NOP,:FCSgnA;    (!1,2,FCSgnA,FCSgnB;)

FCMSame:        NOP,SH=0,:FCMSame1;     !1,1,FCMSame1;
FCMSame1:       NOP,ALUCY,:FCNDiff;     !1,2,FCNDiff,FCNSame;
FCNDiff:        NOP,:FCSgnA;    (!1,2,FCSgnA,FCSgnB;)
FCNSame:        L←0,TASK,:FCRet;        !1,1,FCRet;

FCAZ:   NOP,:FCAZBNZ;   !1,2,FCAZBNZ,FCAZBZ;
FCAZBZ: L←0,TASK,:FCRet;
FCAZBNZ:        SINK←S2,BUS=0,TASK,:FCBST;      return according to sign of B

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

; BUS=0 in instruction calling FCAST, will branch if op. 2 is plus
FCBST:  NOP,:FCAgrB1;   !1,2,FCAgrB1,FCAlsB1;
FCAgrB1:        L←0+1,TASK,:FCRet;
FCAlsB1:        L←0-1,TASK,:FCRet;

FCRet:  Arg0←L,:ShortRet;       called from FSticky also

;---------------------------------------------------------------
;  Sticky (microcode copy of sticky flags)
;---------------------------------------------------------------

!1,2,FSErr,FSOk;

FSticky:        L←stkp-1,TASK;
        stkp←L;
        T←Sticky;
        SINK←stkp,BUS=0;
        L←stk0,:FSErr;  !1,2,FSErr,FSOk;
FSOk:   Sticky←L,L←T,TASK,:FCRet;
FSErr:  NOP,TASK,:RamStkErr;
```

```
;-------------------------------------------------------------;
;              X M E S A   M I C R O C O D E                  ;
;                       Version 41                            ;
;-------------------------------------------------------------;

; MesaROM.Mu - Instruction fetch and general subroutines
; Last modified by Johnsson - December 21, 1979  11:02 AM


;-------------------------------------------------------------
; 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:
;  uCodeVersion   Microcode version   Mesa release
;       0              34              4.1
;       1              39              5.0
;       2              41              6.0

$uCodeVersion   $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
;       sktp=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.
```

```
;-------------------------------------------------------------
; Location-specific Definitions
;-------------------------------------------------------------


; 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, ROM0), 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 "RAM1" and "ROM1" is systematically replaced by "RAM0".

%1,1777,0,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.
;-------------------------------------------------------------

%1,1777,420,Mgo;                                Normal entry to Mesa Emulator - load state
;                                               of process specified by AC0.

%1,1777,400,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.

%1,1777,776,DSTr1,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
;-------------------------------------------------------------

%1,1777,406,Intstop;                                            must correspond to romIntstop
%1,1777,407,Untail;                                             must correspond to romUntail
%7,1777,430,XferGT,Xfer,Mstopr,PORTOpc,LSTr,ALLOCrfr;           Xfer must agree with romXfer
```

```
;-------------------------------------------------------------
; Linkage from Mesa emulator to ROM0
;       The Mesa emulator uses a number of subroutines that reside in ROM0.  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 ROM0, 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 ROM0, 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)
;-------------------------------------------------------------
$ROMMUL         $L004120,0,0;                   MUL routine address (120B) in ROM0
$ROMDIV         $L004121,0,0;                   DIV routine address (121B) in ROM0

$MULDIVretloc   $177576;                        (may be even or odd)

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

%1,1777,574,MULDIVret,MULDIVret1;               return addresses from MUL/DIV in ROM0

;-------------------------------------------------------------
; 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 ROM0

$Fieldretloc    $452;                           RAMCYCX return to Fieldsub (even or odd)
$WFretloc       $523;                           RAMCYCX return to WF (must be odd)

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

%1,1777,452,Fieldrc;                            return address from RAMCYCX to Fieldsub

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

%1,1777,522,WFnzct,WFret;                       return address from RAMCYCX to WF
```

```
;-------------------------------------------------------------
; I n s t r u c t i o n   f e t c h
;
; 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;                  000-007
%7,1777,1410,LL0,LL1,LL2,LL3,LL4,LL5,LL6,LL7;                          010-017
%7,1777,1420,LLB,LLDB,SL0,SL1,SL2,SL3,SL4,SL5;                         020-027
%7,1777,1430,SL6,SL7,SLB,PL0,PL1,PL2,PL3,LG0;                          030-037
%7,1777,1440,LG1,LG2,LG3,LG4,LG5,LG6,LG7,LGB;                          040-047
%7,1777,1450,LGDB,SG0,SG1,SG2,SG3,SGB,LI0,LI1;                         050-057
%7,1777,1460,LI2,LI3,LI4,LI5,LI6,LIN1,LINI,LIB;                        060-067
%7,1777,1470,LIW,LINB,LADRB,GADRB,,,,;                                 070-077
%7,1777,1500,R0,R1,R2,R3,R4,RB,W0,W1;                                  100-107
%7,1777,1510,W2,WB,RF,WF,RDB,RD0,WDB,WD0;                              110-117
%7,1777,1520,RSTR,WSTR,RXLP,WXLP,RILP,RIGP,WILP,RIL0;                  120-127
%7,1777,1530,WS0,WSB,WSF,WSDB,RFC,RFS,WFS,RBL;                         130-137
%7,1777,1540,WBL,,,,,,,;                                               140-147
%7,1777,1550,,,,,,,,;                                                  150-157
%7,1777,1560,,,SLDB,SGDB,PUSH,POP,EXCH,LINKB;                          160-167
%7,1777,1570,DUP,NILCK,,BNDCK,,,,;                                     170-177
%7,1777,1600,J2,J3,J4,J5,J6,J7,J8,J9;                                  200-207
%7,1777,1610,JB,JW,JEQ2,JEQ3,JEQ4,JEQ5,JEQ6,JEQ7;                      210-217
%7,1777,1620,JEQ8,JEQ9,JEQB,JNE2,JNE3,JNE4,JNE5,JNE6;                  220-227
%7,1777,1630,JNE7,JNE8,JNE9,JNEB,JLB,JGEB,JGB,JLEB;                    230-237
%7,1777,1640,JULB,JUGEB,JUGB,JULEB,JZEQB,JZNEB,,JIW;                   240-247
%7,1777,1650,ADD,SUB,MUL,DBL,DIV,LDIV,NEG,INC;                         250-257
%7,1777,1660,AND,OR,XOR,SHIFT,DADD,DSUB,DCOMP,DUCOMP;                  260-267
%7,1777,1670,ADD01,,,,,,,;                                             270-277
%7,1777,1700,EFC0,EFC1,EFC2,EFC3,EFC4,EFC5,EFC6,EFC7;                  300-307
%7,1777,1710,EFC8,EFC9,EFC10,EFC11,EFC12,EFC13,EFC14,EFC15;            310-317
%7,1777,1720,EFCB,LFC1,LFC2,LFC3,LFC4,LFC5,LFC6,LFC7;                  320-327
%7,1777,1730,LFC8,,,,,,,;                                              330-337
%7,1777,1740,,LFCB,SFC,RET,LLKB,PORTO,PORTI,KFCB;                      340-347
%7,1777,1750,DESCB,DESCBS,BLT,BLTL,BLTC,,ALLOC,FREE;                   350-357
%7,1777,1760,IWDC,DWDC,STOP,CATCH,MISC,BITBLT,STARTIO,JRAM;            360-367
%7,1777,1770,DST,LST,LSTF,,WR,RR,BRK,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;                                   dispatch on ib
                ib←L, T←0+1, BUS=0, :NOOP;                      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←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
;

nextAdeaf:      L←XMAR←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;                                          updated pc
                SINK←NWW, BUS=0;                                check pending interrupts
nextAcomx:      T←177400, :nextAi;

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

nextAni:        L←MD AND T, BUS, :nextAgo;                      L←"B"↑8, dispatch on "A"
nextAgo:        ib←L LCY 8, L←T←0+1, :NOOP;                     establish exit state

;
;  Interrupt pending - check if enabled.
;

nextAi:         L←MD;
                SINK←wdc, BUS=0;                                check wakeup counter
                T←M.T, :nextAini;                               isolate left byte
nextAini:       SINK←M, L←T, BUS, :nextAgo;                     dispatch even byte

;
;  Interrupt pending and enabled.
;
!1,2,nextXBini,nextXBii;

nextAii:        L←mpc-1;                                        back up mpc for Savpcinframe
                mpc←L, L←0, :nextXBii;
```

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

nextXB:         L←XMAR←mpc+T;
                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, :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.
;

nextXBi:        SINK←wdc, BUS=0, :nextXBni;                     check wakeup counter

;
;  Interrupt pending and enabled.
;

nextXBii:       ib←L, :Intstop;                                 ib = 0 for even, ~= 0 for odd
```

```
;-------------------------------------------------------------
; S u b r o u t i n e s
;-------------------------------------------------------------

;
;  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,RWBLra;

; Returns for Xpopsub only

!17,20,WSTRrB,WSTRrA,JRAMr,WRr,STARTIOr,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,Tpop0,Tpop1,Tpop2,Tpop3,Tpop4,Tpop5,Tpop6,Tpop7,,,,,,,;

Popsub:         L←stkp-1, BUS, TASK, :Popsuba;
Popsuba:        stkp←L, :Tpop;                                  old stkp > 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;                               returns to Xpopret
;                                                               Note:  putting Tpop here makes
;                                                               stack underflow logic work if
;                                                               stkp=0

Xpopret:        SINK←saveret, BUS;
                :WSTRrB;
```

```
;-------------------------------------------------------------
; 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,Getalphax;                                         shake IR← dispatch
!7,1,GetalphaAx;                                        shake IR← dispatch

Getalpha:       T←ib, IDISP;
Getalphax:      ib←L RSH 1, L←0, BUS=0, :Fieldra;       ib←0, set branch 1 pending

GetalphaA:      L←XMAR←mpc+1;                           initiate fetch
GetalphaAx:     mpc←L;
                T←177400;                               mask for new ib
                L←MD AND T, T←MD;                       L: new ib, T: whole word
Getalphab:      T←377.T, IDISP;                         T now has alpha
                ib←L LCY 8, L←0+1, :Fieldra;            return: no branch pending


;-------------------------------------------------------------
; FetchAB subroutine:
;       Entry conditions:  none
;       Exit conditions:
;         T: <<mpc>+1>
;         ib: unchanged (caller must ensure return to 'nextA')
;-------------------------------------------------------------
!1,1,FetchAB;                                           drops ball 1
!7,1,FetchABx;                                          shakes IR← dispatch
!7,10,LIWr,JWr,,,,,;                                    return points

FetchAB:        L←XMAR←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 Splitcomr 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;                                                    drop ball 1
%160,377,217,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;                    L←1 for Getalpha
SplitalphaB:    saveret←L, L←0, BUS=0, :Splitcom;               (keep ball 1 in air)

Splitcom:       IR←sr33, :Getalpha;                             T:alpha[0-7]
Splitcomr:      L←17 AND T, :Splitx;                            L:alpha[4-7]
Splitx:         righthalf←L, L←T, TASK;                         L:alpha, righthalf:alpha[4-7]
                temp←L;                                         temp:alpha
                L←temp, BUS;                                    dispatch on alpha[1-3]
                temp←L LCY 8, SH<0, :Split0;                    dispatch on alpha[0]

Split0:         L←T←0, :Splitout0;                              L,T:alpha[1-3]
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;                             L:alpha[0-3]

Splitout0:      SINK←saveret, BUS, TASK;                        dispatch return
                lefthalf←L, :RILPr;                             lefthalf:alpha[0-3]
```

```
;-------------------------------------------------------------
; D i s p a t c h e s
;-------------------------------------------------------------

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

pushMD:         L←stkp+1, IR←stkp;                              (IR← causes no branch)
                stkp←L, T←0+1, :pushMDa;

pushMDA:        L←stkp+1, IR←stkp;                              (IR← causes no branch)
                stkp←L, T←0, :pushMDa;

pushMDa:        SINK←DISP, L←T, BUS;                            dispatch on old stkp value
                L←MD, SH=0, TASK, :push0;

;-------------------------------------------------------------
; 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'.
;-------------------------------------------------------------
!1,2,pushT1B,pushT1A;                                           keep ball 1 in air

pushTB:         L←stkp+1, BUS, :pushT1B;
pushTA:         L←stkp+1, BUS, :pushT1A;

pushT1B:        stkp←L, L←T, TASK, :push0;
pushT1A:        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

!17,20,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/LDIV logic.
;       returns to 'nextA' <=> ib = 0  or entry at 'Dpush'
;-------------------------------------------------------------
!1,2,DpA,DpB;
!4,1,Dpushx;                                                    shakes IR←2000 dispatch

Dpush:          MAR←L←ib+T, :DpB;                               L: address of low half

Dpusha:         SINK←ib, BUS=0;
                MAR←L←M+T, :DpA;

DpA:            IR←0, :Dpushb;                                  mACSOURCE will produce 0
DpB:            IR←2000, :Dpushb;                               mACSOURCE will produce 1

Dpushb:         temp←L, :Dpushx;                                temp: address of low half
Dpushx:         L←MD, TASK, :Dpushc;
Dpushc:         taskhole←L;                                     taskhole: low half bits
                T←0+1;
                L←stkp+T+1;
                MAR←temp+1;                                     fetch high half
                stkp←L;                                         stkp ← stkp+2
                L←taskhole;                                     L: low half bits
                SINK←stkp, BUS, :dpush;                         dispatch on new stkp
dpush:          T←MD, :dpush;                                   T: high half bits

dpush1:         stk0←L, L←T, TASK, mACSOURCE, :push1;           stack cells are S-registers,
dpush2:         stk1←L, L←T, TASK, mACSOURCE, :push2;           so mACSOURCE does not affect
dpush3:         stk2←L, L←T, TASK, mACSOURCE, :push3;           addressing.
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, 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 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,MDpopuf,MDpop0,MDpop1,MDpop2,MDpop3,MDpop4,MDpop5,MDpop6,MDpop7,,,,,,,;

StoreB:         L←stkp-1, BUS;
StoreBa:        stkp←L, TASK, :MDpopuf;
StoreA:         L←stkp-1, BUS;
StoreAa:        stkp←L, BUS=0, TASK, :MDpopuf;                  keep branch (1) alive

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,dpopuf2,dpopuf1,dpop1,dpop2,dpop3,dpop4,dpop5,dpop6,dpop7,,,,,,,;
!1,1,Dpopb;                                                     required by placement of
;                                                               MDpopuf only.

Dpop:           L←T←ib+T+1;
MDpopuf:        IR←0, :Dpopb;                                   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.
Dpopa:          L←T←M+T+1;
                IR←ib, :Dpopb;
Dpopb:          MAR←T, temp←L;
dpopuf2:        L←stkp-1, BUS;
                stkp←L, TASK, :dpopuf2;

dpopuf1:        :StkUf;                                         stack underflow, honor TASK
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;
```

```
;-------------------------------------------------------------
; Get operation-specific code from other files
;-------------------------------------------------------------

#MesacROM.mu;
#MesadROM.mu;
```

```
;---------------------------------------------------------------
; MesacROM.Mu - Jumps, Load/Store, Read/Write, Binary/Unary/Stack Operators
; Last modified by Levin - November 5, 1979  4:34 PM
;---------------------------------------------------------------


;---------------------------------------------------------------
; J u m p s
;---------------------------------------------------------------

; 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;                                                   must be odd (shakes IR← below)

JB:             T←ib, :JBx;
JBx:            L←400 OR T;                                     ←DISP will do sign extension
                IR←M;                                           400 above causes branch (1)
                L←DISP-1, :Jbranch;                             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 JWr
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;                                    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)
;-----------------------------------------------------------------
!1,2,Jcz,Jco;

JZEQB:          SINK←stk0, BUS=0;                               test TOS = 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;                               test TOS = 0
                L←stkp-1, TASK, :JZNEBne;

JZNEBne:        stkp←L, :JB;                                    branch, pick up alpha
JZNEBeq:        stkp←L, :nextA;                                 no branch, alignment => nextA
```

```
;---------------------------------------------------------------
; JEQn - if TOS (popped) = TOS (popped), jump PC-relative by n, assuming:
;       stack has precisely two elements
;---------------------------------------------------------------
!1,2,JEQnB,JEQnA;
!7,1,JEQNEcom;                                          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,JNEcom;                                     return points from JEQNEcom

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

;
;       JEQ/JNE common code
;
; !7,1,JEQNEcom;        appears above with JEQn
; !1,2,JEQcom,JNEcom;   appears above with JEQB

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

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
;-------------------------------------------------------------


JNEnB:          temp←L RSH 1, L←0, :JEQNEcom;           temp:0, L:0
JNEnA:          temp←L, L←0, :JEQNEcom;                 temp:1, L:0

!1,2,JNEne,JNEeq;

JNEcom:         L←stkp-T-1, :JNEne;                     L: old stkp - 2

JNEne:          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←ALLONES, :Jadjust;

Jadjust:        L←stk1+T+1;                                     L:stk1 + (0 or 100000)
                temp←L;
                SINK←DISP, BUSODD;                              dispatch ADC/SUB
                T←stk0+T+1, :Jadc;

Jadc:           L←temp-T-1, :Jcommon;                           perform add complement
Jsub:           L←temp-T, :Jcommon;                             perform subtract

Jcommon:        T←ONE;                                          warning: not T←0+1
                L←stkp-T-1, ALUCY;                              test ADC/SUB outcome
                SINK←DISP, SINK←lgm10, BUS=0, TASK, :Jcz;       dispatch on encoded bit 12

Jcz:            stkp←L, :Jnobz;                                 carry is zero (stkp←stkp-2)
Jco:            stkp←L, :Jbo;                                   carry is one (stkp←stkp-2)

Jnobz:          L←mpc+1, TASK, :nextAput;                       no jump, alignment=>nextAa
Jbz:            T←ib, :JBx;                                     jump
Jbo:            T←ib, :JBx;                                     jump
Jnobo:          L←mpc+1, TASK, :nextAput;                       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,JIul;
!1,1,JIWx;

JIW:            L←stkp-T-1, TASK, :JIWx;                    stkp←stkp-2
JIWx:           stkp←L;
                T←stk0;
                L←XMAR←mpc+1;                               load alphabeta
                mpc←L;
                L←stk1-T-1;                                 do unsigned compare
                ALUCY;
                T←MD, :JIuge;

JIuge:          L←mpc+1, TASK, :nextAput;                   out of bounds - to 'nextA'
JIul:           L←cp+T, TASK;                               (removing this TASK saves a
                taskhole←L;                                 word, but leaves a run of
                T←taskhole;                                 15 instructions)
                XMAR←stk0+T;                                fetch <<cp>+alphabeta+X>
                NOP;
                L←MD-1, :Jbranch;                           L: offset
```

```
;------------------------------------------------------------------
; L o a d s
;------------------------------------------------------------------

; 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←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

LG0:            MAR←gp-1, :pushMD;
LG1:            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 - push <<gp>+alpha>
;-------------------------------------------------------------

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


;-------------------------------------------------------------
; LGDB - push <<gp>+alpha>, push <<gp>+alpha+1>
;       LGDB is A-aligned (also ensures no pending branch at entry)
;-------------------------------------------------------------

LGDB:           T←gp+T+1, :LDcommon;                        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, :pushT1B;
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;


;---------------------------------------------------------------
; 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 377B8)
;---------------------------------------------------------------

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
```

```
;-------------------------------------------------------------
; S t o r e s
;-------------------------------------------------------------


;-------------------------------------------------------------
; 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;
```

```
;---------------------------------------------------------------
; 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+T, :StoreB;
SG3:            MAR←gp+T+1, :StoreB;


;---------------------------------------------------------------
; SGB - <<gp>+alpha>←TOS (popped)
;---------------------------------------------------------------

SGB:            IR←sr7, :Getalpha;                              returns to SGBr
SGBr:           T←ngpoffset+T+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+T+1, :SDcommon;                            T: gp-gpoffset+lpoffset
```

```
;-------------------------------------------------------------
; P u t 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

PL0:            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
```

```
;-------------------------------------------------------------
; B i n a r y   o p e r a t i o n s
;-------------------------------------------------------------

; 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,WS0r,WSFr,WFr,
       WSDBrb,WFSrb,BNDCKr,RWBLrb,WBLrb,,,,,,,,,,;


;-------------------------------------------------------------
; 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 push0) 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 20B

!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
Bincomd:        temp2←L, :Binpop;                               stash briefly

BincomA:        L←2000 OR T;                                    make mACSOURCE produce 1
Binpop:         IR←M, :BincomB;

Binpop1:        T←stk1;
                L←stk0, :Binend;
Binpop2:        T←stk2;
                L←stk1, :Binend;
Binpop3:        T←stk3;
                L←stk2, :Binend;
Binpop4:        T←stk4;
                L←stk3, :Binend;
Binpop5:        T←stk5;
                L←stk4, :Binend;
Binpop6:        T←stk6;
                L←stk5, :Binend;
Binpop7:        T←stk7;
                L←stk6, :Binend;

Binend:         SINK←DISP, BUS;                                 perform return dispatch
                SINK←temp2, BUS, :ADDr;                         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;                               pick up ball 1
                L←stkp-1, :pushT1B;                             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
;-------------------------------------------------------------
!7,1,MULDIVcoma;                                        shakes stack dispatch
!1,2,GoROMMUL,GoROMDIV;
!7,2,MULx,DIVx;                                         also shakes bus dispatch

MUL:            IR←T←ret5, :BincomB;
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
;                                                       perhaps, but garbage now.
MULDIVret1:     T←AC1;                                  Normal return
                L←stkp+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;
DIVr:           AC1←L, L←T, BUS=0, :MULDIVcoma;         BUS=0 => DIVx


;-------------------------------------------------------------
; LDIV - push quotient of <TOS-1>,,<TOS-2>/<TOS> (all popped)
;   remainder recoverable by PUSH
;-------------------------------------------------------------

LDIV:           IR←sr27, :Popsub;                       get divisor
LDIVf:          AC2←L;                                  stash it
                IR←T←ret7, :BincomB;                    L:low bits, T:high bits
LDIVr:          AC1←L, L←T, IR←0, :DIVx;                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;                                                    shakes stack dispatch
!1,2,Lshift,Rshift;
!1,2,DoShift,Shiftdone;
!1,2,DoRight,DoLeft;
!1,1,Shiftdonex;

SHIFT:          IR←T←ret10, :BincomB;
SHIFTr:         temp←L, L←T, TASK, :SHIFTx;                     L: value, T: count
SHIFTx:         count←L;
                L←T←count;
                L←0-T, SH<0;                                    L: -count, T: count
                IR←sr1, :Lshift;                                IR← causes no branch

Lshift:         L←37 AND T, TASK, :Shiftcom;                    mask to reasonable size

Rshift:         T←37, IR←37;                                    equivalent to IR←msr0
                L←M AND T, TASK, :Shiftcom;                     mask to reasonable size

Shiftcom:       count←L, :Shiftloop;

Shiftloop:      L←count-1, BUS=0;                               test for completion
                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;                   dispatch to push result
Shiftdonex:     L←temp, TASK, :push0;
```

```
;-------------------------------------------------------------
; D o u b l e - P r e c i s i o n   A r i t h m e t i c
;-------------------------------------------------------------


;-------------------------------------------------------------
; DADD - add two double-word quantities, assuming:
;       stack contains precisely 4 elements
;-------------------------------------------------------------
!1,1,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
```

```
;-------------------------------------------------------------
; R a n g e   C h e c k i n g
;-------------------------------------------------------------


;-------------------------------------------------------------
; NILCK - check TOS for NIL (0), trap if so
;-------------------------------------------------------------
!1,2,InRange,OutOfRange;

NILCK:          L←ret17, :Xpopsub;                          returns to NILCKr
NILCKr:         T←ONE, SH=0, :NILCKpush;                    test TOS=0

NILCKpush:      L←stkp+T, :InRange;

InRange:        SINK←ib, BUS=0, TASK, :Setstkp;             pick up ball 1
OutOfRange:     T←sBoundsFaultm1+T+1, :KFCr;                T:SD index; go trap


;-------------------------------------------------------------
; BNDCK - check subrange inclusion
;       if TOS-1 ~IN [0..TOS) then trap (test is unsigned)
;       only TOS is popped off
;-------------------------------------------------------------
!7,1,BNDCKx;                                                shake push dispatch

BNDCK:          IR←T←ret22, :BincomB;                       returns to BNDCKr
BNDCKr:         L←M-T, :BNDCKx;                             L: value, T: limit
BNDCKx:         T←0, ALUCY, :NILCKpush;
```

```
;---------------------------------------------------------------
; R e a d s
;---------------------------------------------------------------

; 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;                     returns to RBr
RBr:            L←stkp-1, BUS, :ReadB;

ReadB:          stkp←L, :MAStkT;                        to pushMD
ReadA:          stkp←L, BUS=0, :MAStkT;                 to pushMDA


;---------------------------------------------------------------
; 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;                       returns to Dpush


;---------------------------------------------------------------
; RD0 - temp←<TOS>, push <<temp>>, push <<temp>+1>
;---------------------------------------------------------------

RD0:            IR←sr32, :Popsub;                       returns to RD0r
RD0r:           L←0, :Dpusha;
```

```
;-----------------------------------------------------------------
; RILP - push <<<lp>+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]>
;-----------------------------------------------------------------
!3,1,IPcom;                                                     shake IR← at WILPr

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←lefthalf+T;                                 start memory cycle
                L←righthalf;
IPcomx:         T←MD, IDISP;                                    T:local/global value
                MAR←M+T, :pushMD;                               start fetch/store


;-----------------------------------------------------------------
; RIL0 - push <<<lp>>>
;-----------------------------------------------------------------
!1,2,RILxB,RILxA;

RIL0:           MAR←lp-T-1, :RILxB;                             fetch local 0

RILxB:          IR←msr0, L←0, :IPcomx;                          to pushMD
RILxA:          IR←sr1, L←sr1 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;                               fetch TOS
RXLPrb:         L←righthalf+T, TASK;                            L:TOS+alpha[4-7]
                righthalf←L, :RILPr;                            now act like RILP
```

```
;-------------------------------------------------------------
; W r i t e s
;-------------------------------------------------------------


;-------------------------------------------------------------
; Wn - <<TOS> (popped)+n>←<TOS> (popped)
;-------------------------------------------------------------
!1,2,WnB,WnA;                                                  keep ball 1 in air

W0:             T←0, :WnB;
W1:             T←ONE, :WnB;
W2:             T←2, :WnB;

WnB:            IR←sr2, :Wsub;                                 returns to StoreB

WnA:            IR←sr3, :Wsub;                                 returns to StoreA

;-------------------------------------------------------------
; Write subroutine:
;-------------------------------------------------------------
!7,1,Wsubx;                                                    shake IR← dispatch

Wsub:           L←stkp-1, BUS, :Wsubx;
Wsubx:          stkp←L, IDISP, :MAStkT;


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

WB:             IR←sr16, :Getalpha;                            returns to WBr
WBr:            :WnB;                                          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;                                                     shake stack dispatch

WSB:            IR←T←ret14, :BincomA;                          alignment requires BincomA
WSBr:           T←M, L←T, :WSBx;

WSBx:           MAR←ib+T, :WScom;
WScom:          temp←L;
WScoma:         L←stkp-1;
                MD←temp;
                mACSOURCE, TASK, :Setstkp;


;-------------------------------------------------------------
; WS0 - act like WSB but with alpha value of zero
;-------------------------------------------------------------
!7,1,WS0x;                                                     shake stack dispatch

WS0:            IR←T←ret15, :BincomB;
WS0r:           T←M, L←T, :WS0x;

WS0x:           MAR←T, :WScom;
```

```
;---------------------------------------------------------------
; WILP - <<lp>+alpha[0-3]>+alpha[4-7] ← <TOS> (popped)
;---------------------------------------------------------------

WILP:           L←ret2, :Splitalpha;                        get halves of alpha
WILPr:          IR←sr2;                                     IPcom will exit to StoreB
                T←lp, :IPcom;                               prepare to undiddle


;---------------------------------------------------------------
; WXLP - <TOS>+<<lp>+alpha[0-3]>+alpha[4-7] ← <TOS-1> (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]
                righthalf←L, :WILPr;                        now act like WILP


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

WDB:            IR←sr31, :Popsub;                           returns to Dpop


;---------------------------------------------------------------
; WD0 - temp←<TOS> (popped), pop into <temp>+1 and <temp>
;---------------------------------------------------------------

WD0:            L←ret6, TASK, :Xpopsub;                     returns to WD0r
WD0r:           L←0, :Dpopa;


;---------------------------------------------------------------
; WSDB - like WDB but with address below data words, assuming:
;       WSDB is A-aligned (also ensures no pending branch at entry)
;---------------------------------------------------------------
!7,1,WSDBx;

WSDB:           IR←sr24, :Popsub;                           get low data word
WSDBra:         saveret←L;                                  stash it briefly
                IR←T←ret20, :BincomA;                       alignment requires BincomA
WSDBrb:         T←M, L←T, :WSDBx;                           L:high data, T:address
WSDBx:          MAR←T←ib+T+1;                               start store of low data word
                temp←L, L←T;                                temp:high data
                temp2←L, TASK;                              temp2:updated address
                MD←saveret;                                 stash low data word
                MAR←temp2-1, :WScoma;                       start store of high data word
```

```
;-------------------------------------------------------------
; L o n g   P o i n t e r   o p e r a t i o n s
;-------------------------------------------------------------

!1,1,RWBLcom;                                                   drop ball 1

;-------------------------------------------------------------
; 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;                                                    drop ball 1
!7,1,RWBLxb;                                                    shake stkp dispatch
!7,1,WBLx;                                                      shake stkp dispatch
!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;                                       T: low half, L: high half
                temp←L;                                         temp: high pointer
                L←alpha+T;                                      L: low pointer+alpha
                T←MD;                                           T: bank register to save
                MAR←BankReg;                                    reaccess bank register
                frame←L, L←T;                                   frame: pointer
                taskhole←L, TASK;                               taskhole: old bank register
                MD←temp, :WBLx;                                 set new alternate bank value

WBLx:           XMAR←frame;                                     start memory access
                L←entry+1, BUS;                                 dispatch RBL/WBL
                entry←L, L←T, :RBLra;                           (L←T for WBLrc only)

RBLra:          T←MD, :RWBLtail;                                T: data from memory
WBLra:          IR←ret24, :BincomB;                             returns to WBLrb
WBLrb:          T←M, :WBLx;                                     T: data to write

WBLrc:          MD←M, :RWBLtail;                                stash data in memory

RWBLtail:       MAR←BankReg;
                SINK←entry, BUS;                                dispatch return
RWBLdone:       MD←taskhole, :RWBLdone;                         restore bank register

RBLdone:        L←temp2+1, BUS, :pushT1B;                       temp2: original stkp-2
WBLdone:        L←temp2, TASK, :Setstkp;                        temp2: original stkp-3
```

```
;-------------------------------------------------------------------
; U n a r y  o p e r a t i o n s
;-------------------------------------------------------------------

; XMESA Note:  Untail is wired down by a pre-def in MesaROM.mu


;-------------------------------------------------------------------
; INC - TOS ← <TOS>+1
;-------------------------------------------------------------------

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


;-------------------------------------------------------------------
; NEG - TOS ← -<TOS>
;-------------------------------------------------------------------

NEG:            L←ret11, TASK, :Xpopsub;
NEGr:           L←0-T, :Untail;


;-------------------------------------------------------------------
; DBL - TOS ← 2*<TOS>
;-------------------------------------------------------------------

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


;-------------------------------------------------------------------
; Unary operation common code
;-------------------------------------------------------------------

Untail:         T←M, :pushTB;
```

```
;-----------------------------------------------------------------
; S t a c k   a n d   M i s c e l l a n e o u s   O p e r a t i o n s
;-----------------------------------------------------------------


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

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


;-----------------------------------------------------------------
; POP - subtract 1 from stack pointer
;-----------------------------------------------------------------

POP:            L←stkp-1, SH=0, TASK, :Setstkp;                 L=0 <=> branch 1 pending
;                                                               need not check stkp=0


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

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


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

EXCH:           IR←ret11, :EXCHx;
EXCHx:          L←stkp-1;                                       dispatch on stkp-1
                L←M+1, BUS, TASK, :Bincomd;                     set temp2←stkp
EXCHr:          T←M, L←T, :dpush;                               Note: dispatch using temp2

;-----------------------------------------------------------------
; LADRB - push alpha+lp (undiddled)
;-----------------------------------------------------------------
!1,1,LADRBx;                                                    shake branch from Getalpha

LADRB:          IR←sr10, :Getalpha;                             returns to LADRBr
LADRBr:         T←nlpoffset+T+1, :LADRBx;
LADRBx:         L←lp+T, :Untail;


;-----------------------------------------------------------------
; GADRB - push alpha+gp (undiddled)
;-----------------------------------------------------------------
!1,1,GADRBx;                                                    shake branch from Getalpha

GADRB:          IR←sr11, :Getalpha;                             returns to GADRBr
GADRBr:         T←ngpoffset+T+1, :GADRBx;
GADRBx:         L←gp+T, :Untail;
```

```
;-----------------------------------------------------------------
; S t r i n g   O p e r a t i o n s
;-----------------------------------------------------------------
!7,1,STRsub;                                                    shake stack dispatch
!1,2,STRsubA,STRsubB;
!1,2,RSTRrx,WSTRrx;

STRsub:         L←stkp-1;                                       update stack pointer
                stkp←L;
                L←ib+T;                                         compute index and offset
                SINK←M, BUSODD, TASK;
                count←L RSH 1, :STRsubA;

STRsubA:        L←177400, :STRsubcom;                           left byte
STRsubB:        L←377, :STRsubcom;                              right byte

STRsubcom:      T←temp;                                         get string address
                MAR←count+T;                                    start fetch of word
                T←M;                                            move mask to more useful place
                SINK←DISP, BUSODD;                              dispatch to caller
                mask←L, SH<0, :RSTRrx;                          dispatch B/A, mask for WSTR


;-----------------------------------------------------------------
; 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;                                stash string base address
RSTRrx:         L←MD AND T, TASK, :RSTRB;                       isolate good bits

RSTRB:          temp←L, :RSTRcom;
RSTRA:          temp←L LCY 8, :RSTRcom;                         right-justify byte

RSTRcom:        T←temp, :pushTA;                                go push result byte


;-----------------------------------------------------------------
; 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;
WSTRr:          temp←L, :STRsub;                                stash string base
WSTRrx:         L←MD AND NOT T, :WSTRB;                         isolate good bits

WSTRB:          temp2←L, L←ret0, TASK, :Xpopsub;                stash them, return to WSTRrB
WSTRA:          temp2←L, L←ret0+1, TASK, :Xpopsub;              stash them, return to WSTRrA

WSTRrA:         taskhole←L LCY 8;                               move new data to odd byte
                T←taskhole, :WSTRrB;

WSTRrB:         T←mask.T;
                L←temp2 OR T;
                T←temp;                                         retrieve string address
                MAR←count+T;
                TASK;
                MD←M, :nextA;
```

```
;-------------------------------------------------------------
; 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

%1,3,2,RFrr,WFrr;                                       returns from Fieldsub
!7,1,Fieldsub;                                          shakes stack dispatch
; !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
;-------------------------------------------------------------

RF:             IR←sr12, :Popsub;
RFr:            L←ret0, :Fieldsub;
RFrr:           T←mask.T,  :pushTA;                     alignment requires 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
;-------------------------------------------------------------
; !1,2,WFnzct,WFret; - see location-specific definitions

WF:             IR←T←ret17, :BincomB;                   L:new data, T:address
WFr:            newfield←L, L←ret0+1, :Fieldsub;        (actually, L←ret1)

WFrr:           T←mask;
                L←M AND NOT T;                          set old field bits to zero
                temp←L;                                 stash result
                T←newfield.T;                           save new field bits
                L←temp OR T, TASK;                      merge old and new
                CYCOUT←L;                               stash briefly
                T←index, BUS=0;                         get position, test for zero
                L←WFretloc, :WFnzct;                    get return address from ROM

WFnzct:         PC←L;                                   stash return
                L←20-T, SWMODE;                         L:remaining count to cycle
                T←CYCOUT, :RAMCYCX;                     go cycle remaining amount
WFret:          MAR←frame;                              start memory
                L←stkp-1;                               pop remaining word
                MD←CYCOUT, TASK, :JZNEBeq;              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
;-------------------------------------------------------------

WSF:            IR←T←ret16, :BincomB;                   L:address, T:new data
WSFr:           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
RFSra:          temp←L;                                     stash for WFSa
                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
;-------------------------------------------------------------
!1,2,Fieldsuba,WFSa;

WFS:            L←ret14, TASK, :Xpopsub;                    get alpha and beta
WFSra:          temp←L;                                     stash temporarily
                IR←T←ret21, :BincomB;                       L:new data, T:address
WFSrb:          newfield←L, L←ret0+1, BUS=0, :Fieldsub;     returns quickly to WFSa
WFSa:           frame←L;                                    stash address
                T←177400;                                   to separate alpha and beta
                L←temp AND T, T←temp, :Getalphab;           L:alpha, T:both
;                                                           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;                                        T:address
                L←ret2, :Fieldsub;                          returns to RFrr
```

```
;-----------------------------------------------------------------
; 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
;                                                               T: beta, ib: alpha
Fieldra:        L←ret5;
                saveret←L, :Splitcomr;                          get two halves of beta
Fieldrb:        T←righthalf;                                    index for MASKTAB
                MAR←MASKTAB+T;                                  start fetch of mask
                T←lefthalf+T+1;                                 L:left-cycle count
                L←17 AND T;                                     mask to 4 bits
                index←L;                                        stash position
                L←MD, TASK;                                     L:mask for caller's use
                mask←L;                                         stash mask
                SINK←temp2, BUS;                                temp2=2 <=> RFC
                T←frame, :NotCodeSeg;                           get base address
NotCodeSeg:     L←MAR←ib+T, :StashFieldLoc;                     add alpha
IsCodeSeg:      XMAR←ib+T, :DoCycle;                            add alpha

StashFieldLoc:  frame←L, :DoCycle;                              stash updated address for WF
DoCycle:        L←Fieldretloc;                                  return location from RAMCYCX
                PC←L;
                T←MD, SWMODE;                                   data word into T for cycle
                L←index, :RAMCYCX;                              count to cycle, go do it
Fieldrc:        SINK←temp2, BUSODD;                             return dispatch
                L←T←CYCOUT, :RFrr;                              cycled data word in L and T
```

```
;------------------------------------------------------------
; MesadROM.Mu - Xfer, State switching, process support, Nova interface
; Last modified by Levin - December 21, 1979  11:14 AM
;------------------------------------------------------------


;------------------------------------------------------------
; F r a m e   A l l o c a t i o n
;------------------------------------------------------------


;------------------------------------------------------------
; 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;                                             subroutine returns
!1,2,ALLOCrf,XferGrf;                                           failure returns
!3,4,Alloc0,Alloc1,Alloc2,Alloc3;                               dispatch on pointer flag
;       if more than 2 callers, un-comment the following pre-definition:
; !17,1,Allocx;                                                 shake IR← dispatch

AllocSub:       L←avm1+T+1, TASK, :Allocx;                      fetch av entry

Allocx:         entry←L;                                        save av entry address
                L←MAR←entry;
                T←3;                                            mask for pointer flags
                L←MD AND T, T←MD;                               (L←MD AND 3, T←MD)
                temp←L, L←MAR←T;                                start reading pointer
                SINK←temp, BUS;                                 branch on bits 14:15
                frame←L, :Alloc0;

;
; Bits 14:15 = 00, a frame of the right index is queued for allocation
;

Alloc0:         L←MD, TASK;                                     new entry for frame vector
                temp←L;                                         new value of vector entry
                MAR←entry;                                      update frame vector
                L←T←frame, IDISP;                               establish exit conditions
                MD←temp, :ALLOCr;                               update and return

;
; Bits 14:15 = 01, allocation list empty:  restore argument, take failure return
;

Alloc1:         L←temp2, IDISP, TASK;                           restore parameter
                temp2←L LSH 1, :ALLOCrf;                        allocation failed

;
; Bits 14:15 = 10, a pointer to an alternate list to use
;

Alloc2:         temp←L RSH 1, :Allocp;                          indirection: index←index/4

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)
;-------------------------------------------------------------

!3,4,RETr,FREEr,LSTFr,;                                 FreeSub returns
!17,1,Freex;                                            shake IR← dispatch

FreeSub:        MAR←frame-1;                            start read of fsi word
Freex:          NOP;                                    wait for memory
                T←MD;                                   T←index
                L←MAR←avm1+T+1;                         fetch av entry
                entry←L;                                save av entry address
                L←MD;                                   read current pointer
                MAR←frame;                              write it into current frame
                temp←L, TASK;
                MD←temp;                                write!
                MAR←entry;                              entry points at frame
                IDISP, TASK;
                MD←frame, :RETr;                        free
```

```
;-------------------------------------------------------------
; ALLOC - allocate a frame whose fsi is specified by <TOS> (popped)
;-------------------------------------------------------------
!1,1,Savpcinframe;                                          (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,XferGfz;                                   used by XferGrf

ALLOC:          L←ret7, TASK, :Xpopsub;                     returns to ALLOCrx
ALLOCrx:        temp2←L LSH 1, IR←msr0, :AllocSub;          L,T: fsi
ALLOCr:         L←stkp+1, BUS, :pushT1B;                    duplicates pushTB

;
;  Allocation failed - save mpc, undiddle lp, push fsi*4 on stack, then trap
;

ALLOCrf:        IR←sr5, :Savpcinframe;                      failure because lists empty
ALLOCrfr:       L←temp2, TASK, :doAllocTrap;                pick up trap parameter

;
;  Inform software that allocation failed
;

doAllocTrap:    ATPreg←L;                                   store param. to trap proc.
                T←sAllocTrap, :Mtrap;                       go trap to software


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

FREE:           L←ret10, TASK, :Xpopsub;                    returns to FREErx
FREErx:         frame←L, TASK;
                IR←sr1, :FreeSub;
FREEr:          :next;
```

```
;-------------------------------------------------------------
; D e s c r i p t o r   I n s t r u c t i o n s
;-------------------------------------------------------------


;-------------------------------------------------------------
; 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;                     T:address of frame

DESCBcom:       MAR←gfioffset+T;                                start fetch of gfi word
                T←gfimask;                                      mask to isolate gfi bits
                T←MD.T;                                         T:gfi
                L←ib+T, T←ib;                                   L:gfi+alpha, T:alpha
                T←M+T+1, :pushTA;                               pushTA because A-aligned


;-------------------------------------------------------------
; 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
```

```
;-------------------------------------------------------------
; T r a n s f e r   O p e r a t i o n s
;-------------------------------------------------------------


;-------------------------------------------------------------
; 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)
;-------------------------------------------------------------
; !1,1,Savpcinframe;                                        required by PORTO
; !7,10,XferGT,Xfer,Mstopr,PORTOpc,LSTr,ALLOCrfr,,;         returns (appear with ALLOC)
!7,1,Savpcx;                                                shake IR← dispatch
!1,2,Spcodd,Spceven;                                        pc odd or even

Savpcinframe:   T←cp, :Savpcx;                              code segment base
Savpcx:         L←mpc-T;                                    L is code-relative pc
                SINK←ib, BUS=0;                             check for odd or even pc
                T←M, :Spcodd;                               pick up pc word addr

Spcodd:         L←0-T, TASK, :Spcopc;                       - pc => odd, this word
Spceven:        L←0+T+1, TASK, :Spcopc;                     + pc => even, next word

Spcopc:         taskhole←L;                                 pc value to save
                L←0;                                        (can't merge above - TASK)
                T←npcoffset;                                offset to pc stash
                MAR←lp-T, T←lp;                             (MAR←lp-npcoffset, T←lp)
                ib←L;                                       clear ib for XferG
                L←nlpoffset+T+1;                            L:undiddled lp
                MD←taskhole;                                stash pc in frame+pcoffset
                my←L, IDISP, TASK;                          store undiddled lp
                lp←L, :XferGT;
```

```
;-------------------------------------------------------------
; 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,Xfer0r,Xfer1r;                                             return points
!1,2,Loadgc,LoadgcTrap;
!1,2,LoadgcOK,LoadgcNull;                                       good global frame or null
!1,2,LoadgcIn,LoadgcSwap;                                       in-core or swapped out
!1,2,LoadgcDiv2,LoadgcDiv4;                                     first/second shift
!1,2,LoadgcNoXM,LoadgcIsXM;                                     short/long codebase
!1,2,DoLoad,NoLoad;                                             changing G or not

Loadgc:         L←lpoffset+T;                                   diddle (presumed) lp
                lp←L;                                           (only correct if frame ptr)
                T←MD;                                           global frame address
                L←MD;                                           2nd word (PC or EV offset)
                MAR←cpoffset+T;                                 read code pointer
                mpc←L, L←T;                                     copy g to L for null test
                L←cpoffset+T+1, SH=0;                           test gf=0
                taskhole←L, :LoadgcOK;                          taskhole:addr of hi code base

LoadgcOK:       L←MD, BUSODD, TASK;                             L: low bits of code base
                cp←L, :LoadgcIn;                                stash low bits, branch if odd

LoadgcIn:       MAR←BankReg;                                    access bank register
                T←taskhole+1;                                   T:addr of global 0
                L←gp-T-1;                                       compare with current gp
                T←14, SH=0;                                     mask to save primary bank
                L←MD AND T, :DoLoad;                            L: primary bank *4
DoLoad:         temp2←L, :LoadgcShift;                          temp2: primary bank *4
LoadgcShift:    newfield←L RSH 1, L←0-T, :LoadgcDiv2;           newfield: bank*2, L: negative
LoadgcDiv2:     L←newfield, SH<0, TASK, :LoadgcShift;           SH<0 forces branch, TASK safe
LoadgcDiv4:     MAR←T←taskhole;                                 fetch high bits of code base
                L←gpcpoffset+T;                                 diddle gp
                gp←L;
                T←177400;                                       mask for high bits
                L←MD AND T, T←MD;
                T←3.T, SH=0;                                    T: bank if long codebase
                MAR←BankReg, :LoadgcNoXM;                       initiate store

LoadgcNoXM:     T←newfield, :LoadgcIsXM;                        T: MDS bank
LoadgcIsXM:     L←temp2 OR T, TASK;                             L: new bank registers
                MD←M;                                           stash bank

NoLoad:         XMAR←cp;                                        access first cseg word
                IDISP, TASK;                                    dispatch return
                MD←ONE, :Xfer0r;

;
;       picked up global frame of zero somewhere, call it unbound
;
!1,1,Stashmx;
LoadgcNull:     T←sUnbound, :Stashmx;                           BUSODD may be pending

;
;       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.
;-------------------------------------------------------------
!3,4,Xfers,XferG,RETxr,;                                    returns from CheckXferTrap
!1,2,NoXferTrap,DoXferTrap;
!3,1,DoXferTrapx;

CheckXferTrap:  L←XTSreg, BUSODD;                           XTSreg[15]=1 => trap
                SINK←DISP, BUS, :NoXferTrap;                dispatch (possible) return

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;                        L:trap parameter
                XTPreg←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'.
;-------------------------------------------------------------

!3,4,Xfer0,Xfer1,Xfer2,Xfer3;                                   destination link type

Xfer:           T←mx;                                           mx[14:15] is dest link type
                IR←0, :CheckXferTrap;
Xfers:          L←3 AND T;                                      extract type bits
                SINK←M, L←T, BUS;                               L:dest link, branch on type
                SH=0, MAR←T, :Xfer0;                            check for link = 0. Memory
;                                                               data is used only if link
;                                                               is frame pointer or indirect

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

Xfer0:          IR←msr0, :Loadgc;                               to LoadgcNull if dest link = 0
Xfer0r:         L←T←mpc;                                        offset from cp: - odd, + even

;
; 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
;
!1,2,Xdobreak,Xnobreak;
!1,2,Xfer0B,Xfer0A;
!1,2,XbrkB,XbrkA;
!1,2,XbrkBgo,XbrkAgo;

                SINK←brkbyte, BUS=0;                            set up by Loadstate
                SH<0, L←0, :Xdobreak;                           dispatch even/odd pc

;
; Not proceeding from a breakpoint - simply pick up next instruction
;

Xnobreak:       :Xfer0B;

Xfer0B:         L←XMAR←cp+T, :nextAdeafa;                       fetch word, pc even
Xfer0A:         L←XMAR←cp-T;                                    fetch word, pc odd
                mpc←L, :nextXBni;

;
; Proceeding from a breakpoint - dispatch brkbyte and clear it
;

Xdobreak:       ib←L, :XbrkB;                                   clear ib for XbrkA

XbrkB:          IR←sr20;                                        here if BRK at even byte
                L←XMAR←cp+T, :GetalphaAx;                       set up ib (return to XbrkBr)

XbrkA:          L←cp-T;                                         here if BRK at odd byte
                mpc←L, L←0, BUS=0, :XbrkBr;                     ib already zero (to XbrkAgo)

XbrkBr:         SINK←brkbyte, BUS, :XbrkBgo;                    dispatch brkbyte

XbrkBgo:        brkbyte←L RSH 1, T←0+1, :NOOP;                  clear brkbyte, act like nextA
XbrkAgo:        brkbyte←L, T←0+1, BUS=0, :NOOP;                 clear brkbyte, act like next
```

```
;-------------------------------------------------------------
; mx[14-15] = 01
;       Destination link is procedure descriptor:
;         mx[0-8]: GFT index (gfi)
;         mx[9-13]: EV bias, or entry number (en)
;-------------------------------------------------------------

Xfer1:          temp←L RSH 1;                                   temp:ep*2+garbage
                count←L MLSH 1;                                 since L=T, count←L LCY 1;
                L←count, TASK;                                  gfi now in 0-7 and 15
                count←L LCY 8;                                  count:gfi w/high bits garbage
                L←count, TASK;
                count←L LSH 1;                                  count:gfi*2 w/high garbage
                T←count;
                T←1777.T;                                       T:gfi*2
                MAR←gftm1+T+1;                                  fetch GFT[T]
                IR←sr1, :Loadgc;                                pick up two word entry into
;                                                               gp and mpc
Xfer1r:         L←temp, TASK;                                   L:en*2+high bits of garbage
                count←L RSH 1;                                  count:en+high garbage
                T←count;
                T←enmask.T;                                     T:en
                L←mpc+T+1, TASK;                                (mpc has EV base in code seg)
                count←L LSH 1, :XferG;                          count:ep*2


;-------------------------------------------------------------
; mx[14-15] = 10
;       Destination link is indirect:
;         mx[0-15]: address of location holding destination link
;-------------------------------------------------------------

Xfer2:          NOP;                                            wait for memory
                T←MD, :Xfers;

;-------------------------------------------------------------
; mx[14-15] = 11
;       Destination link is unbound:
;         mx[0-15]: passed to trap handler
;-------------------------------------------------------------

Xfer3:          T←sUnbound, :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;                                        parameter to CheckXferTrap
                IR←ONE, :CheckXferTrap;
XferG:          T←count;                                        index into entry vector
                XMAR←cp+T;                                      fetch of new pc and fsi
                T←cp-1;                                         point just before bytes
;                                                               (main loop increments mpc)
                IR←sr1;                                         note: does not cause branch
                L←MD+T;                                         relocate pc from cseg base
                T←MD;                                           second word contains fsi
                mpc←L;                                          new pc setup, ib already 0
                T←377.T, :AllocSub;                             mask for size index
;
; Stash source link in new frame, establishing dynamic link
;
XferGr:         MAR←retlinkoffset+T;                            T has new frame base
                L←lpoffset+T;                                   diddle new lp
                lp←L;                                           install diddled lp
                MD←my;                                          source link to new frame

;
; Stash new global pointer in new frame (same for local call)
;
                MAR←T;                                          write gp to word 0 of frame
                T←gpoffset;                                     offset to point at gf base
                L←gp-T, TASK;                                   subtract off offset
                MD←M, :nextAdeaf;                               global pointer stashed, GO!

;
;  Frame allocation failed - push destination link, then trap
;
; !1,2,doAllocTrap,XferGfz;                                     (appears with ALLOC)

XferGrf:        L←mx, BUS=0;                                    pick up destination, test = 0
                T←count-1, :doAllocTrap;                        T:2*ep+1

;       if destination link is zero (i.e. local procedure call), we must first
;       fabricate the destination link

XferGfz:        L←T, T←ngfioffset;                              offset from gp to gfi word
                MAR←gp-T;                                       start fetch of gfi word
                count←L LSH 1;                                  count:4*ep+2
                L←count-1;                                      L:4*ep+1
                T←gfimask;                                      mask to save gfi only
                T←MD.T;                                         T:gfi
                L←M+T, :doAllocTrap;                            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
;-------------------------------------------------------------
!1,2,EFCgetr,LLKBr;                                     return points
!1,2,framelink,codelink;
!7,1,Fetchlink;                                         shake IR← in KFCB

Getlink:        T←gp;                                   diddled frame address
                MAR←T←ngpoffset+T+1;                    fetch word 0 of global frame
                L←temp+T, T←temp;                       L:address of link in frame
                taskhole←L;                             stash it
                L←cp+T;                                 L:address of link in code
                SINK←MD, BUSODD, TASK;                  test bit 15 of word zero
                temp2←L, :framelink;                    stash code link address

framelink:      MAR←taskhole, :Fetchlink;               fetch link from frame
codelink:       XMAR←temp2, :Fetchlink;                 fetch link from code

Fetchlink:      SINK←DISP, BUS=0;                       dispatch to caller
                L←T←MD, :EFCgetr;
```

```
;-------------------------------------------------------------
; EFCn - perform XFER to destination specified by external link n
;-------------------------------------------------------------
; !1,1,EFCr; implicit in EFCr's return number (23B)

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;


;-------------------------------------------------------------
; EFCB - perform XFER to destination specified by external link 'alpha'
;-------------------------------------------------------------
!1,1,EFCdoGetlink;                                          shake B/A dispatch (Getalpha)

EFCB:           IR←sr23, :Getalpha;                         fetch link number
EFCr:           L←0-T-1, TASK, :EFCdoGetlink;               L:-(link number+1)

EFCdoGetlink:   temp←L, :Getlink;                           stash index for Getlink
EFCgetr:        IR←sr1, :SFCr;                              for Savpcinframe; no branch


;-------------------------------------------------------------
; SFC - Stack Function Call (using descriptor on top of stack)
;-------------------------------------------------------------

SFC:            IR←sr1, :Popsub;                            get dest link for xfer
;                                                           now assume IR still has sr1
SFCr:           mx←L, :Savpcinframe;                        set dest link, return to Xfer
```

```
;-------------------------------------------------------------
; 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;                 fetch alpha
KFCr:           IR←avm1, T←avm1+T+1, :KFCx;         DISP must be non zero
KFCx:           MAR←sdoffset+T, :Fetchlink;         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;                               can't TASK, T has trap index
                OTPreg←L, :Mtrap;

Mtrap:          T←avm1+T+1;
                MAR←sdoffset+T;                     fetch dest link for trap
                NOP;
Mtrapa:         L←MD, TASK;                         (enter here from PORTO)
                mx←L, :Xfer;
```

```
;-------------------------------------------------------------
; LFCn - call local procedure n (i.e. within same global frame)
;-------------------------------------------------------------
!1,1,LFCx;                                                    shake B/A dispatch

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.
;-------------------------------------------------------------
!1,1,RETx;                                                  shake B/A branch

RET:            T←lp, :RETx;                                local pointer

RETx:           IR←2, :CheckXferTrap;
RETxr:          MAR←nretlinkoffset+T;                       get previous local frame
                L←nlpoffset+T+1;
                frame←L;                                    stash for 'Free'
                L←MD;                                       pick up prev frame pointer
                mx←L, L←0, IR←msr0, TASK;                   mx points to caller
                my←L, :FreeSub;                             clear my and go free frame
RETr:           T←mx, :Xfers;                               xfer back to caller


;-------------------------------------------------------------
; 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;                                 address of local 0
                T←ib;
                L←mx-T, TASK;                               L: mx-alpha
                MD←M, :nextA;                               local 0 ← 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
```

```
;-------------------------------------------------------------
; P o r t   O p e r a t i o n s
;-------------------------------------------------------------


;-------------------------------------------------------------
; 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;                                      frame addr to word 0 of PORT
                MAR←M+1;                                    second word of PORT
                my←L, :Mtrapa;                              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;                                   store it as second word
                TASK, :PORTIz;
PORTIz:         MD←my, :next;                               store my or zero
```

```
;-------------------------------------------------------------
; S t a t e   S w i t c h i n g
;-------------------------------------------------------------


;-------------------------------------------------------------
; 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;                                    i.e. T←-11
                L←lp, :Savsuba;
Sav11r:         L←stkp, :Savsub;
Sav10r:         T←stkp+1;
                L←-7+T;                                     check if stkp > 5 or negative
                L←0+T+1, ALUCY;                             L:stkp+2
                temp2←L, L←0-T, :Savok;                     L:-stkp-1

Savmax:         T←-7;                                       stkp > 5  => save all
                L←stk7, :Savsuba;

Savok:          SINK←temp2, BUS;                            stkp < 6 => save to stkp+2
                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;                             return to caller
                T←-12, :DSTr1;                              (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;                             dispatch on pos. value
                SINK←M, BUS, TASK;
                MD←temp2, :Sav0r;
```

```
;-------------------------------------------------------------
; 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[stkp+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.
;-------------------------------------------------------------
!17,20,Lsr0,Lsr1,Lsr2,Lsr3,Lsr4,Lsr5,Lsr6,Lsr7,Lsr10,Lsr11,Lsr12,,,,,;
!1,2,Lsmax,Ldsuba;

Loadstate:      temp←L, IR←msr0, :NovaIntrOn;                   stash pointer
Lsr:            T←12, :Ldsuba;
Lsr12:          my←L, :Ldsub;
Lsr11:          mx←L, :Ldsub;
Lsr10:          stkp←L;
                T←stkp;                                         check for BRK resumption
                L←177400 AND T;                                 (i.e. bytecode in stkp)
                brkbyte←L LCY 8;                                stash for Xfer
                L←T←17.T;                                       mask to 4 bits
                L←-7+T;                                         check stkp > 6
                L←T, SH<0;
                stkp←L, T←0+T+1, :Lsmax;                        T:stkp+1
Lsmax:          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;
Ldsuba:         MAR←temp+T;
                L←ALLONES+T;                                    decr count for next time
                count←L, L←T;                                   use old value for dispatch
                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;                                       get alpha
                T←lp+T+1;
                L←nlpoffset1+T+1, TASK;                     L:lp-lpoffset+alpha
                temp←L, IR←ret0, :Savestatea;
DSTr1:          L←my, :Savsuba;                             save my too!
DSTr:           temp←L, L←0, TASK, BUS=0, :Setstkp;         zap stkp, return to 'nextA'


;-------------------------------------------------------------
; 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;                                       make Savpcinframe happy
                IR←sr4, :Savpcinframe;                      returns to LSTr
LSTr:           T←temp;                                     get alpha back
                L←lp+T, TASK, :Loadstate;                   lp already undiddled


;-------------------------------------------------------------
; 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←lpoffset;
                L←lp-T, TASK;                               compute frame base
                frame←L;
                IR←sr2, :FreeSub;

LSTFr:          T←frame;                                    set up by FreeSub
                L←ib+T, TASK, :Loadstate;                   get state from dead frame
```

```
;-------------------------------------------------------------
; E m u l a t o r   A c c e s s
;-------------------------------------------------------------


;-------------------------------------------------------------
; 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;                                          shake B/A dispatch (BLTL)

RR:             L←0, SWMODE, :DoRamRWB;
DoRamRWB:       SINK←M, BUS, L←T, :ramOverflow;         L←T for WR


;-------------------------------------------------------------
; WR - emulator register alpha ← <TOS> (popped), where:
;       WR is A-aligned (also ensures no pending branch at entry)
;       alpha: 1 => wdc, 2 => XTSreg
;-------------------------------------------------------------

WR:             L←ret3, TASK, :Xpopsub;
WRr:            L←2, SWMODE, :DoRamRWB;


;-------------------------------------------------------------
; JRAM - JMPRAM for Mesa programs (when emulator is in ROM1)
;-------------------------------------------------------------

JRAM:           L←ret2, TASK, :Xpopsub;
JRAMr:          SINK←M, BUS, SWMODE, :next;             BUS applied to 'nextBa' (=0)
```

```
;-------------------------------------------------------------
; P r o c e s s / M o n i t o r  S u p p o r t
;-------------------------------------------------------------
!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,FastMRExx;                                         drop ball 1
!1,1,FastEExx;                                          drop ball 1
!7,1,FastEExx;                                          shake IR←isME/isMXD
!1,2,MXDr,MEr;
!7,1,FastEExxx;                                         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;                                  IDISP:1, DISP:1, mACSOURCE:1
$isMRE          $65403;                                 IDISP:13, DISP:3, mACSOURCE:16
$isMXD          $402;                                   IDISP:0, DISP:2, mACSOURCE:0

ME:             IR←isME, :FastEEx;                      indicate ME instruction
MXD:            IR←isMXD, :FastEEx;                     indicate MXD instruction

MRE:            MAR←HardMRE, :FastMREx;                 <HardMRE> ~= 0 => do Nova code
FastMREx:       IR←isMRE, :MXDr;                        indicate MRE instruction

FastEEx:        MAR←stk0, IDISP, :FastEExx;             fetch monitor lock
FastEExx:       T←100000, :MXDr;                        value of unlocked monitor lock

MXDr:           L←MD, mACSOURCE, :FastEExxx;            L:0 if locked (or queue empty)
MEr:            L←MD-T, mACSOURCE, :FastEExxx;          L:0 if unlocked

FastEExxx:      MAR←stk0, SH=0, :MXDrr;                 start store, test lock state

; 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;                 L:100000, T:0 (stkp value)
MErr:           T←0+1, :FastEEtrap1;                    L:0, T:1 (stkp value)
MRErr:          L←0+1, TASK, :FastEEtrap2;              L:1 (stkp value)

MEXDdone:       MD←M, L←T, TASK, :Setstkp;

MREdone:        stkp←L, :ME;                            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;                               if you uncomment this, don't
;               AC3←L;                                      forget the pre-def above!
MoveParms3:     L←stk2, TASK;
FastEEtrap2:    AC2←L;                                      (enter here from MRE)
MoveParms2:     L←stk1, TASK;
FastEEtrap1:    AC1←L;                                      (enter here from ME/MXD)
MoveParms1:     L←stk0, TASK;
                AC0←L;

                L←0, TASK;                                  indicate stack empty
                stkp←L;
                T←DISP+1, :STOP;
```

```
;-------------------------------------------------------------
; M i s c e l l a n e o u s   O p e r a t i o n s
;-------------------------------------------------------------


;-------------------------------------------------------------
; 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←ret4, TASK, :Xpopsub;                         get argument in L
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;                             get argument in T
MISCr:          L←11-T, :MISCx;                                 test for RCLK
MISCx:          L←CLOCKLOC-1, SH=0;
                temp←L, IR←0, :RamMisc;
RCLK:           L←clockreg, :Dpushc;                            don't TASK here!
RamMisc:        L←3, SWMODE, :DoRamRWB;                         dispatch alpha#11 to RAM
```

```
;-------------------------------------------------------------
; 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←sr1, :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;                       save even/odd across ROM call
BITBLTx:        L←10, SWMODE, :DoRamRWB;
```

```
;-------------------------------------------------------------
; M e s a / N o v a   C o m m u n i c a t i o n
;-------------------------------------------------------------


;-------------------------------------------------------------
; Subroutines to Enable/Disable Nova Interrupts
;-------------------------------------------------------------
; currently each subroutine has only one caller
!7,1,NovaIntrOffx;                                          shake IR← dispatch

NovaIntrOff:    T←100000;                                   disable bit
NovaIntrOffx:   L←NWW OR T, TASK, IDISP;                    turn it on, dispatch return
                NWW←L, :Mstop;

NovaIntrOn:     T←100000;                                   disable bit
                L←NWW AND NOT T, IDISP;                     turn it off, dispatch return
                NWW←L, L←0, :Lsr;


;-------------------------------------------------------------
; IWDC - Increment Wakeup Disable Counter (disable interrupts)
;-------------------------------------------------------------
!1,2,IDnz,IDz;

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


;-------------------------------------------------------------
; DWDC - Decrement Wakeup Disable Counter (enable interrupts)
;-------------------------------------------------------------
!1,1,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, :nextAdeaf;


;-------------------------------------------------------------
; Entry to Mesa Emulation
;       AC0 holds address of current process state block
;       Location 'PSBloc' is assumed to hold the same value
;-------------------------------------------------------------

Mgo:            L←AC0, :Loadstate;
```

```
;-------------------------------------------------------------
;  N o v a   I n t e r f a c e
;-------------------------------------------------------------
$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 ROM0 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, :Savpcinframe;                          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 ...
```

```
;-----------------------------------------------------------------
; XMesaOverflow.Mu - Driver for XMesaRAM.mu
; Last modified by Johnsson - September 22, 1980  9:11 AM
;-----------------------------------------------------------------


;       Get Alto Definitions  (Mesab.mu included internally by XMesaRAM.mu)

#AltoConsts23.mu;

;
;       Reserve locations 0-17 of RAM for device tasks (silent boot)
;

;%17,1777,0,Task0,Task1,Task2,Task3,Task4,Task5,Task6,Task7,
;         Task10,Task11,Task12,Task13,Task14,Task15,Task16,Task17;
;
;Task0:          TASK, :Task0;
;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
;
%1,1777,550,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;                                             must match ramMgo

%1,1777,402,BLTintpend,BLTloop;                                 BLTloop must match ramBLTloop
%3,1777,404,BLTnoint,BLTint,BLTLnoint,BLTLint;                  BLTint must match ramBLTint

%1,1777,410,Overflow;                                           must correspond to ramOverflow

%1,1777,411,JramBITBLT;                                         advertized place to do BITBLT

;-----------------------------------------------------------------
; 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;              return addresses from BITBLT in ROM0


;-----------------------------------------------------------------
; Overflow instruction dispatch
;-----------------------------------------------------------------
!17,20,RR,BLTL,WR,MISC,DADD,DSUB,DCOMP,DUCOMP,BITBLT,,,,,,,;    dispatched in ROM1

GoToROM:        L←ONE, SWMODE;                                  smash G to disable
                gp←L, :romMgo;                                  optimization on initial entry
```

```
;-------------------------------------------------------------
; D o u b l e - P r e c i s i o n   A r i t h m e t i c
;-------------------------------------------------------------

; !1,1,DSUBsub;                                          shake B/A dispatch
!3,4,DAStail,,,DCOMPr;                                   returns from DSUBsub
!1,1,Dsetstkp;                                           shake ALUCY dispatch
!1,1,Setstkp;                                            shake IDISP from BLTnoint

Overflow:       :RR;                                     dispatch pending
;                                                        TASK pending for doubles

;-------------------------------------------------------------
; DADD - add two double-word quantities, assuming:
;       stack contains precisely 4 elements
;-------------------------------------------------------------
; !1,1,DADDx;                                            shake B/A dispatch
!1,2,DADDnocarry,DADDcarry;

DADD:           T←stk2, :DADDx;                          T:low bits of right operand
DADDx:          L←stk0+T;                                L:low half of sum
                stk0←L, ALUCY;                           stash, test carry
                T←stk3, :DADDnocarry;                    T:high bits of right operand

DADDnocarry:    L←stk1+T, :DASCtail;                     L:high half of sum
DADDcarry:      L←stk1+T+1, :DASCtail;                   L:high half of sum


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

DSUB:           IR←msr0, :DSUBsub;


;-------------------------------------------------------------
; Double-precision subtract subroutine
;-------------------------------------------------------------
!1,2,DSUBborrow,DSUBnoborrow;
!7,1,DSUBx;                                              shake IR← dispatch

DSUBsub:        T←stk2, :DSUBx;                          T:low bits of right operand
DSUBx:          L←stk0-T;                                L:low half of difference
                stk0←L, ALUCY;                           borrow = ~carry
                T←stk3, :DSUBborrow;                     T:high bits of right operand

DSUBborrow:     L←stk1-T-1, IDISP, :DASCtail;            L:high half of difference
DSUBnoborrow:   L←stk1-T, IDISP, :DASCtail;              L:high half of difference


;-------------------------------------------------------------
; Common exit code
;-------------------------------------------------------------

DASCtail:       stk1←L, ALUCY, :DAStail;                 carry used by double compares
DAStail:        T←2, :Dsetstkp;                          adjust stack pointer

Dsetstkp:       L←stkp-T, SWMODE, :Setstkp;
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)
;-------------------------------------------------------------
; !1,1,DCOMPxa;                                          shake B/A dispatch
!10,1,DCOMPxb;                                          shake IR← dispatch
!1,2,DCOMPnocarry,DCOMPcarry;
!1,2,DCOMPgtr,DCOMPequal;

DCOMP:          IR←T←100000, :DCOMPxa;                  IR←msr0, must shake dispatch
DCOMPxa:        L←stk1+T, :DCOMPxb;                     scale left operand
DCOMPxb:        stk1←L;
                L←stk3+T, TASK;                         scale right operand
                stk3←L, :DSUBsub;                       do DSUB, return to DCOMPr

DCOMPr:         T←stk0, :DCOMPnocarry;                  L: stk1, ALUCY pending

DCOMPnocarry:   L←0-1, BUS=0, :DCOMPsetT;               left opnd < right opnd
DCOMPcarry:     L←M OR T;                               L: stk0 OR stk1
                SH=0;
DCOMPsetT:      T←3, :DCOMPgtr;                         T: amount to adjust stack

DCOMPgtr:       L←0+1, :DCOMPequal;                     left opnd > right opnd
DCOMPequal:     stk0←L, :Dsetstkp;                      stash result


;-------------------------------------------------------------
; 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:         IR←sr3, :DSUBsub;                       returns to DCOMPr
```

```
;-------------------------------------------------------------
; E m u l a t o r   A c c e s s
;-------------------------------------------------------------


;-------------------------------------------------------------
; 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:            L←0, SWMODE, :RR0;                              (so SH=0 below will branch)

RR1:            L←wdc, SH=0, :romUntail;                        will go to pushTA
RR2:            L←XTSreg, SH=0, :romUntail;                     will go to pushTA
RR3:            L←XTPreg, SH=0, :romUntail;                     will go to pushTA
RR4:            L←ATPreg, SH=0, :romUntail;                     will go to pushTA
RR5:            L←OTPreg, SH=0, :romUntail;                     will go to 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;                         performed in ROM

WR:             SINK←ib, BUS;                                   dispatch on alpha
WR0:            SWMODE, :WR0;

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,BLTintx;                                           shake branch from BLTloop

;  Entry sequence in ROM1; actual entry is at BLTloop

;BLT:           stk7←L, SWMODE, :BLTx;                  stk7=0 <=> branch pending
;BLTx:          IR←msr0, :ramBLTloop;                   IR← is harmless


BLTloop:        L←T←stk1-1, BUS=0, :BLTnoint;
BLTnoint:       stk1←L, L←BUS AND ~T, IDISP, :BLTmore;  L←0 on last iteration (value
;                                                       on bus is irrelevant, since T
;                                                       will be -1).  IDISP on last
;                                                       cycle requires that Setstkp
;                                                       be odd.

BLTmore:        T←cp, :BLTsource;

BLTsource:      MAR←stk0, :BLTupdate;                   start data source fetch
BLTCsource:     XMAR←stk0+T, :BLTupdate;                start code source fetch

BLTupdate:      L←stk0+1;
                stk0←L;                                 update source pointer
                L←stk2+1;
                T←MD;                                   source data
                MAR←stk2;                               start dest. write
                stk2←L, L←T;                            update dest. pointer
                SINK←NWW, BUS=0, TASK;                  check pending interrupts
                MD←M, :BLTintpend;                      loop or check further

BLTintpend:     SINK←wdc, BUS=0, :BLTloop;              check if interrupts enabled

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

BLTint:         SINK←stk7, BUS=0, :BLTintx;             test even/odd pc
BLTintx:        L←mpc-1, :BLTeven;                      prepare to back up

BLTeven:        mpc←L, L←0, :BLTodd;                    even - back up pc, clear ib
BLTodd:         ib←L, SWMODE;                           odd - set ib non-zero
                :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;                                         shake BUS=0 and IR←
!7,1,BLTLsetBRx;                                        shake IR←
!7,10,BLTLret0,BLTLret1,BLTLret2,BBret3,BBret4,,,;
!1,2,BLTLintpend,BLTLloop;
!1,2,BLTLmore,BLTLdone;
;!1,2,BLTLnoint,BLTLint;                                 appears above

; Note:  ROM1 code does stk7←L


BLTL:           MAR←BankReg;                                    access bank register
                T←stk1;                                         high source bits
                L←stk1+T;                                       L: high source *2
                temp←L LSH 1, IR←msr0;                          temp: high source *4;
                L←MD, TASK;                                     L: old bank register
                stk6←L;                                 stk6: stashed register
                T←stk4;                                         T: high dest bits
                T←3.T, :BLTLsetBR;                              (would like to avoid this)
;                                                               returns to BLTLret0

BLTLloop:       L←T←stk2-1, BUS=0, :BLTLnoint;                  decrement count, test done
BLTLnoint:      stk2←L, :BLTLmore;                              T: -1 the last time

BLTLmore:       MAR←stk0;                                       fetch source word
                L←stk0+1;                                       bump source pointer
                stk0←L;
                L←stk3+1;                                       bump destination pointer
                T←MD;
                XMAR←stk3;                                      initiate store
                stk3←L, L←T;                                    L: data
                SINK←NWW, BUS=0, TASK;                          check for possible interrupt
BLTLret0:       MD←M, :BLTLintpend;                             stash data

BLTLintpend:    SINK←wdc, BUS=0, :BLTLloop;                     check if enabled

BLTLint:        IR←sr2, :BLTLsetBR;                             restore bank before interrupt
BLTLret2:       MD←stk6, :BLTint;                               BLTint shakes branch

BLTLdone:       IR←sr1, :BLTLsetBR;                             restore bank before exit
BLTLret1:       MD←stk6, L←stk6 AND NOT T, :BLTdone;            BLTdone shakes branch, L←0


BLTLsetBR:      MAR←BankReg, :BLTLsetBRx;
BLTLsetBRx:     L←temp OR T, IDISP;                             (used by BLTLret0 only)
                SINK←0, BUS=0, :BLTLret0;                       force branch for BLTLret0
;                                                               others must shake
```

```
;-------------------------------------------------------------
; BITBLT - do BITBLT using ROM0 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;               save even/odd across ROM call
;BITBLTx:       L←10, SWMODE, :DoRamRWB;



!1,2,IntOff,TestLong;
!1,2,LongBB,DoBITBLT;                                  also shake SetBR branch
%2,3,1,BBshortDone,BBlongDone;

JramBITBLT:     L←ib, TASK;                            save alignment for interrupts
                stk7←L;
                L←3, TASK;                             set stkp for interrupts
                stkp←L, :BITBLT;

BITBLT:         L←stk0;
                AC2←L, L←0, TASK;
                stk5←L;
                SINK←wdc, BUS=0;                       check if Mesa interrupts off
                T←100000, :IntOff;
IntOff:         L←NWW OR T;                            if so, shut off Nova's
                NWW←L, :TestLong;

TestLong:       MAR←stk0+1;                            fetch word for long check
                L←stk1;
                AC1←L;                                 stash intermediate state
                L←MD, BUS=0, TASK;
                temp←L, :LongBB;                       BR word

!7,1,BBx;                               shake IR←sr3

LongBB:         MAR←BankReg;
                L←2;
                stk5←L;
                IR←sr3;
                L←MD, TASK;                            old Bank reg
BBx:            stk6←L, :BLTLsetBR;
BBret3:         MD←temp, :DoBITBLT;

DoBITBLT:       L←BITBLTret, SWMODE;                   get return address
                PC←L, L←0, :ROMBITBLT;                 L←0 for Alto II ROM0 "feature"

BITBLTdone:     T←100000;
                L←NWW AND NOT T;
                SINK←stk5, BUS;
                NWW←L, L←T←0, :BBshortDone;

BBlongDone:     IR←sr4, :BLTLsetBR;
BBret4:         MD←stk6, L←T, :BBshortDone;

BBshortDone:    brkbyte←L, BUS=0, SWMODE, :Setstkp;    don't bother to validate stkp

BITBLTintr:L←AC1;                                      pick up intermediate state
                SINK←stk5, BUS;
                stk1←L, :BLTint;                       stash intermediate state
```