```
;-----------------------------------------------------------------;
;               M E S A   M I C R O C O D E                       ;
;                       Version 34                                ;
;-----------------------------------------------------------------;

; Mesa.Mu - Instruction fetch and general subroutines
; Last modified by Levin - August 1, 1978  11:27 AM

;Completely rewritten by Roy Levin, Sept-Oct. 1977
;Modified by Johnsson; July 25, 1977  10:20 AM
;First version assembled 5 June 1976.
;Developed from Lampson's MESA.U of 21 March 1976.


;-----------------------------------------------------------------
; 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.


;-----------------------------------------------------------------
; Get definitions for ALTO and MESA
;-----------------------------------------------------------------

#Mesab.mu;
```

```
;-----------------------------------------------------------------
; 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.

%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 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.
;-----------------------------------------------------------------


;-----------------------------------------------------------------
; 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 = 177675B
;         ROM1 extant and emulator in RAM  => MULDIVretloc = 177162B
;-----------------------------------------------------------------
$ROMMUL         $L004120,0,0;                   MUL routine address (120B) in ROM0
$ROMDIV         $L004121,0,0;                   DIV routine address (121B) in ROM0

$MULDIVretloc   $177675;

; The first value in the following pre-def must be:
;       (MULDIVretloc AND 777B)+1               (yes, 'plus', not 'OR').

!676,2,MULDIVret,MULDIVret1;                    return addresses from MUL/DIV in ROM0


;-----------------------------------------------------------------
; BITBLT linkage:
;       An additional constraint peculiar to the BITBLT 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 => BITBLTret = 177714B
;         ROM1 extant and emulator in RAM  => BITBLTret = 177175B
;-----------------------------------------------------------------
$ROMBITBLT      $L004124,0,0;                   BITBLT routine address (124B) in ROM0

$BITBLTret      $177714;

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

!714,2,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 = 605B
;         ROM1 extant and emulator in RAM  => Fieldretloc = 335B, WFretloc = 203B
;-----------------------------------------------------------------
$RAMCYCX        $L004022,0,0;                   CYCLE routine address (22B) in ROM0

$Fieldretloc    $452;                           RAMCYCX return to Fieldsub
$WFretloc       $605;                           RAMCYCX return to WF

; The first value in the following pre-def must be the same as 'Fieldretloc' above.

!452,1,Fieldrc;                                 return address from RAMCYCX to Fieldsub

; The first value in the following pre-def must be the same as 'WFretloc' above.

!605,2, 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,LG0,LG1,LG2,LG3,LG4,LG5,LG6,LG7;                         010-017
%7,1777,1420,LGB,LGDB,SG0,SG1,SG2,SG3,SGB,SGDB;                       020-027
%7,1777,1430,LL0,LL1,LL2,LL3,LL4,LL5,LL6,LL7;                         030-037
%7,1777,1440,LLB,LLDB,SL0,SL1,SL2,SL3,SL4,SL5;                        040-047
%7,1777,1450,SL6,SL7,SLB,SLDB,LI0,LI1,LI2,LI3;                        050-057
%7,1777,1460,LI4,LI5,LI6,LIN1,LIB,LIW,LINB,;                          060-067
%7,1777,1470,,,,,,,,;                                                 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,,,,,,,,;                                                 130-137
%7,1777,1540,,WS0,WSB,WSF,WSDB,RFC,RFS,WFS;                           140-147
%7,1777,1550,PUSH,POP,EXCH,PUSHX,DUP,,,;                              150-157
%7,1777,1560,J2,J3,J4,J5,J6,J7,J8,J9;                                 160-167
%7,1777,1570,JB,JW,JEQ2,JEQ3,JEQ4,JEQ5,JEQ6,JEQ7;                     170-177
%7,1777,1600,JEQ8,JEQ9,JEQB,JNE2,JNE3,JNE4,JNE5,JNE6;                 200-207
%7,1777,1610,JNE7,JNE8,JNE9,JNEB,JLB,JGEB,JGB,JLEB;                   210-217
%7,1777,1620,JULB,JUGEB,JUGB,JULEB,JZEQB,JZNEB,JIB,JIW;               220-227
%7,1777,1630,,,,,,,,;                                                 230-237
%7,1777,1640,,,,,,,,;                                                 240-247
%7,1777,1650,,,,,DESCB,DESCBS,,;                                      250-257
%7,1777,1660,ADD,SUB,MUL,DBL,DIV,LDIV,NEG,INC;                        260-267
%7,1777,1670,AND,OR,XOR,SHIFT,DADD,DSUB,DCOMP,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,LFC9,LFC10,LFC11,LFC12,LFC13,LFC14,LFC15;           330-337
%7,1777,1740,LFC16,LFCB,SFC,RET,LLKB,PORTO,PORTI,KFCB;                340-347
%7,1777,1750,LADRB,GADRB,BLT,ALLOC,FREE,IWDC,DWDC,BLTC;               350-357
%7,1777,1760,STOP,CATCH,,,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,,;

; Returns for Xpopsub only

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


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

Dpush:          L←T←ib+T+1;
                IR←0, :Dpushb;
Dpusha:         L←T←M+T+1;
                IR←ib, :Dpushb;
Dpushb:         MAR←L←M-1;
                temp←L, T←0+1;
                L←stkp+T+1;                               stkp←stkp+2
                stkp←L;
                L←MD, TASK;
                taskhole←L;
                SINK←DISP, BUS=0;
                MAR←temp+1, :DpA;

DpA:            IR←0, :Dpushc;                            (IR← causes no dispatch,
;                                                         but subsequent mACSOURCE
;                                                         will produce 0.)
DpB:            IR←2000, :Dpushc;                         mACSOURCE will produce 1

Dpushc:         L←taskhole, :Dpushx;
Dpushx:         SINK←stkp, BUS;                           dispatch on new stkp

dpush:          T←MD, :dpush;

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'.
;-----------------------------------------------------------------

!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;
                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;
```