```
;---------------------------------------------------------------
; Mesac.Mu - Jumps, Load/Store, Read/Write, Binary/Unary/Stack Operators
; Last modified by Johnsson - July 26, 1978  1:57 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,JZEQBne,JZEQBeq;

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

JZEQBne:        stkp←L, :nextA;                                 no branch, alignment => nextA
JZEQBeq:        stkp←L, :JB;                                    branch, pick up alpha


;---------------------------------------------------------------
; 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,Jcz,Jco;
!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
```

```
;---------------------------------------------------------------
; JIB - see Principles of Operation for description
;   assumes:
;       stack contains precisely two elements
;       if JIB is A-aligned, B byte is irrelevant
;       alpha in B byte, beta in A byte of word after JIB
;---------------------------------------------------------------
!1,1,JIcom;

JIB:            IR←msr0, :JIcom;

!1,2,JIBr,JIWr;
!1,2,JIBe,JIBo;

JIBr:           temp←L RSH 1;
                MAR←temp+T;                                     fetch <<cp>+alphabeta+X/2>
                SINK←stk0, BUSODD;                              test which byte we want
                T←377, :JIBe;                                   byte mask

JIBe:           L←MD AND NOT T, TASK;                           save left byte
                temp←L LCY 8;                                   swap halves
                L←temp-1, :Jbranchf;
JIBo:           L←MD AND T, :JnA;                               save right byte


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

JIW:            IR←sr1, :JIcom;

JIWr:           MAR←M+T;                                        fetch <<cp>+alphabeta+X>
                NOP;
                L←MD-1, :Jbranchf;                              L has offset now


;---------------------------------------------------------------
;  JI common code
;       assumes only 2 callers
;---------------------------------------------------------------
!1,2,JIuge,JIul;

JIcom:          L←stkp-T-1, TASK, :JIcomx;                      stkp←stkp-2
JIcomx:         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;
                temp←L;
                T←temp, IDISP;
                L←stk0, :JIBr;
```

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

; Note: These instructions keep track of their parity
!1,2,LLyB,LLyA;                                          keep ball 1 in air
!1,2,LGyB,LGyA;                                          keep ball 1 in air
!1,2,LI0xB,LI0xA;                                        keep ball 1 in air


;---------------------------------------------------------------
; LLn - push <<lp>+n>
;---------------------------------------------------------------

; 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+1, :pushMD;
LL4:            MAR←lp+T+1, :pushMD;
LL5:            T←3, :LLyB;
LL6:            T←4, :LLyB;
LL7:            T←5, :LLyB;

LLyB:           MAR←lp+T, :pushMD;
LLyA:           MAR←lp+T, :pushMDA;


;---------------------------------------------------------------
; LLB - push <<lp>+alpha>
;---------------------------------------------------------------

LLB:            IR←sr4, :Getalpha;                       returns to LLBr
LLBr:           T←nlpoffset+T+1, :LLyB;                  to undiddle lp


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

LLDB:           T←lp;
                T←nlpoffset+T+1, :Dpush;
```

```
;---------------------------------------------------------------
; LGn - push <<gp>+n>
;---------------------------------------------------------------

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

LG0:            MAR←gp-1, :pushMD;
LG1:            MAR←gp, :pushMD;
LG2:            MAR←gp+1, :pushMD;
LG3:            MAR←gp+T+1, :pushMD;
LG4:            T←3, :LGyB;
LG5:            T←4, :LGyB;
LG6:            T←5, :LGyB;
LG7:            T←6, :LGyB;

LGyB:           MAR←gp+T, :pushMD;
LGyA:           MAR←gp+T, :pushMDA;


;---------------------------------------------------------------
; LGB - push <<gp>+alpha>
;---------------------------------------------------------------

LGB:            IR←sr5, :Getalpha;                              returns to LGBr
LGBr:           T←ngpoffset+T+1, :LGyB;                         to undiddle gp


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

LGDB:           T←gp;
                T←ngpoffset+T+1, :Dpush;
```

```
;-------------------------------------------------------------
; LIn - push n
;-------------------------------------------------------------

; 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;


;-------------------------------------------------------------
; 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)
;------------------------------------------------------------------
!1,2,SLxB,SLxA;                                          keep ball 1 in air

; 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+1, :StoreB;
SL4:            MAR←lp+T+1, :StoreB;
SL5:            T←3, :SLxB;
SL6:            T←4, :SLxB;
SL7:            T←5, :SLxB;

SLxB:           MAR←lp+T, :StoreB;
SLxA:           MAR←lp+T, :StoreA;


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

SLB:            IR←sr6, :Getalpha;                       returns to SLBr
SLBr:           T←nlpoffset+T+1, :SLxB;


;------------------------------------------------------------------
; 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;
                T←nlpoffset+T+1, :Dpop;
```

```
;-----------------------------------------------------------------
; SGn - <<gp>+n>←TOS (popped)
;-----------------------------------------------------------------

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

SG0:            MAR←gp-1, :StoreB;
SG1:            MAR←gp, :StoreB;
SG2:            MAR←gp+1, :StoreB;
SG3:            MAR←gp+T+1, :StoreB;


;-----------------------------------------------------------------
; SGB - <<gp>+alpha>←TOS (popped)
;-----------------------------------------------------------------
!1,1,SGBx;                                                      drop ball 1

SGB:            IR←sr7, :Getalpha;                              returns to SGBr
SGBr:           T←ngpoffset+T+1, :SGBx;
SGBx:           MAR←gp+T, :StoreB;


;-----------------------------------------------------------------
; 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←ngpoffset+T+1, :Dpop;
```

```
;---------------------------------------------------------------
; 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,,,,,,,,,,,,,;


;---------------------------------------------------------------
; 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+1, :MULDIVcomb;              AC0←0 keeps ROM happy
DIVx:           AC0←L, T←0, BUS=0, :MULDIVcomb;         BUS=0  => GoROMDIV

MULDIVcomb:     L←MULDIVretloc+T, 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
;---------------------------------------------------------------

!1,2,DAStail,DCOMPr;                                    returns from DSUBsub

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

DASCtail:       stk1←L, :DAStail;                       high-order carry is lost
DAStail:        T←2;                                    adjust stack pointer
Dsetstkp:       L←stkp-T, TASK, :Setstkp;

;---------------------------------------------------------------
; DSUB - subtract two double-word quantities, assuming:
;       stack contains precisely 4 elements
;---------------------------------------------------------------
!1,1,DSUBsub;                                           shake B/A dispatch

DSUB:           IR←msr0, :DSUBsub;


;---------------------------------------------------------------
; DCOMP - compare two double-word quantities, 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) )
;---------------------------------------------------------------
!1,2,DCOMPnz,DCOMPz;
!1,2,DCOMPpos,DCOMPdone;

DCOMP:          IR←sr1, :DSUBsub;                       perform subtract first
DCOMPr:         L←stk1, BUS=0;                          test high word = 0
                L←0-1, SH<0, :DCOMPnz;                  test high word < 0

DCOMPnz:        T←3, :DCOMPpos;                         non-zero, pending branch +, -
DCOMPz:         L←stk0, BUS=0, :DCOMPnz;                zero, test low word = 0

DCOMPpos:       L←0+1, :DCOMPdone;                      positive
DCOMPdone:      stk0←L, :Dsetstkp;                      stash result

;---------------------------------------------------------------
; Double-precision subtract subroutine
;---------------------------------------------------------------
!1,2,DSUBborrow,DSUBnoborrow;

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, TASK, :DASCtail;     L:high half of difference
DSUBnoborrow:   L←stk1-T, IDISP, TASK, :DASCtail;       L:high half of difference
```

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