Mesac.Mu - Jumps, Load/Store, Read/Write, Binary/Unary/Stack Operators

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

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

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

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

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

11,2,JZEQBne,JZEQBeq:

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

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)

11,2,JZNEBne,JZNEBeq:

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

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

11,2,JEQnB,JEQnA; shake IR= dispatch

JEQ2:  IR= sr0, L+T, :JEQnB; returns to J2
JEQ3:  IR= sr1, L+T, :JEQnB; returns to J3
JEQ4:  IR= sr2, L+T, :JEQnB; returns to J4
JEQ5:  IR= sr3, L+T, :JEQnB; returns to J5
JEQ6:  IR= sr4, L+T, :JEQnB; returns to J6
JEQ7:  IR= sr5, L+T, :JEQnB; returns to J7
JEQ8:  IR= sr6, L+T, :JEQnB; returns to J8
JEQ9:  IR= sr7, L+T, :JEQnB; returns to J9

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

JEQB:  IR= sr10, :JEQnA; returns to JB

; JEQ common code

11,2,JEQcom,JEQcom; 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)

11,2,JEQne,JEQeq;

JEQcom:  L=stk0-T-1, :JEQne; L: old stk0 - 2
JEQne:  SINK= temp, BUS, TASK, :Setstk; no jump, reset stkp
JEQeq:  stkp=L, IDISP, :JEQNExxx; jump, set stkp, then dispatch

; JEQJNE common code

; 17,1,JEQNEcom; appears above with JEQn
; 11,2,JEQcom,JEQcom; appears above with JEQB

JEQNEcom:  T=stk1;
           L=stk0-T, SH=0;
           T=0+1, SH=0, :JEQcom; dispatch EQ/NE
           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
;-----------------------------------------------------------------
11,2,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
11,2,JNENe,JNEnQ;
JNEcom:    L=stkp-T-1, :JNEn; L: old stkp - 2
JNEn:     stkp=L, IDISP, :JEQNExxx; jump, set stkp, then dispatch
JNEnQ:    SINK=temp, BUS, TASK, :Setstkp; no jump, reset stkp
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
- \( \text{JrB} \) 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: \( \text{IR}=10, \text{Jscale} \);
JLEB: \( \text{IR}=11, \text{Jscale} \);
JGB: \( \text{IR}=\text{ONE}, \text{Jscale} \);
JGEB: \( \text{IR}=0, \text{Jscale} \);
JULB: \( \text{IR}=10, \text{Jnoscale} \);
JULEB: \( \text{IR}=11, \text{Jnoscale} \);
JUGB: \( \text{IR}=\text{ONE}, \text{Jnoscale} \);
JUGEB: \( \text{IR}=0, \text{Jnoscale} \);

Comparison "subroutine":

\( 1 \), \( 2 \), \( \text{Jcz}, \text{Jco} \);
\( 1 \), \( 2 \), \( \text{Jnobz}, \text{Jbz} \);
\( 1 \), \( 2 \), \( \text{Jbo}, \text{Jnobo} \);

\( \text{Jscale} \): \( T^{77777}, \text{Jadjust} \);
\( \text{Jnoscale} \): \( T^{\text{ALLONES}}, \text{Jadjust} \);
\( \text{Jadjust} \): \( L^{\text{stk}1+T+1}; \)
\( \text{Jadc} \): \( L^{\text{temp}+T-1}; \)
\( \text{Jcommon} \): \( T^{\text{ONE}}; \)
\( \text{Jcz} \): \( \text{stk}p^{L}, \text{Jnobz} \);
\( \text{Jbo} \): \( \text{stk}p^{L}, \text{Jbo} \);
\( \text{Jbz} \): \( T^{ib}, \text{JBx} \);
\( \text{Jnobo} \): \( L^{mpc+1}, \text{TASK, :nextAput}; \)

- \( \text{adc, branch if carry one} \)
- \( \text{sub, branch if carry one} \)
- \( \text{sub, branch if carry zero} \)
- \( \text{adc, branch if carry zero} \)
- \( \text{perform add-complement} \)
- \( \text{perform subtract} \)
- \( \text{warning: not } T^0+1 \)
- \( \text{test } \text{ADC/SUB} \) outcome
- \( \text{dispatch on encoded bit 12} \)
- \( \text{carry is zero } (\text{stk}p^{\text{stk}-2}) \)
- \( \text{carry is one } (\text{stk}p^{\text{stk}-2}) \)
- \( \text{no jump, alignment=>nextAa} \)
- \( \text{jump} \)
- \( \text{no jump, alignment=>nextAa} \)
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;
SINK=stk0, BUSODD;
T=377, :JIBe;

fetch <<cp>+alphabeta+X/2>
test which byte we want
byte mask

JIBe: L=MD AND NOT T, TASK;
temp=L LCY 8;
L+temp-1, :Jbranchf;
L=MD AND T, :JnA;
save left byte
swap halves
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;
NOP;
L=MD-1, :Jbranchf;

fetch <<cp>+alphabeta+X>
L has offset now

JI common code
assumes only 2 callers

1,2,JIuge,JIul;

JIcom: L+stk-p-T-1, TASK, :JIcomx;
stkp-stkp-2
stkp=L;
T=stk0;
L=MAR+mpc+1;
mpc=L;
L=stk1-T-1;
ALUCY;
T=MD, :JIuge;

load alphabeta
do unsigned compare

JIuge: L=mpc+1, TASK, :nextAput;

JIul: L=cp+1, TASK;
temp=L;
T=temp, IDISP;
L=stk0, :JIBr;

out of bounds - to 'nextA'
; Loads

; Note: These instructions keep track of their parity
11,2,LLyB,LLyA; keep ball 1 in air
11,2,LLyB,LLyA; keep ball 1 in air
11,2,LLyB,LLyA; 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;
LLBr: T-nlpoffset+T+1, :LLyB;
returns to LLBr
returns to LLBr
to undiddle lp
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+stk+1, BUS, :LIOxB;
LI1:  L+stk+1, BUS, :pushT1B;
LI2:  T=2, :pushTB;
LI3:  T=3, :pushTB;
LI4:  T=4, :pushTB;
LI5:  T=5, :pushTB;
LI6:  T=6, :pushTB;
LIOxB:  stkp=L, L=0, TASK, :push0;
LIOxA:  stkp=L, BUS=0, L=0, TASK, :push0;  BUS=0 keeps branch pending

; Lin1 - push -1
;--------------------------------------------------------
LIN1:  T=ALLONES, :pushTB;

; 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
duplicates pushT1A, but because of overlapping return points, we can't use it
LIWr:  L+stk+1, BUS, :pushT1A;  ; ; ;
; Stores
;-----------------------------------------------------------------
; SLn - <<lp>+n>TOS (popped)
;-----------------------------------------------------------------
1,2,SLxB,SLxA; keep ball 1 in air

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

SLO: 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=n1poffset+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=n1poffset+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)

loop, SGBx;
    drop ball 1
    SGB:   IR=sr7, :Geta1pha;  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;
; Binary operations

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

37,40,ADDr,UBr,ANDr,ORr,XORr,MLr,DLr,SHFlr,EXFlr,MRFlr,SRFlr,WSBr,WSOr,WSFr,WFr,WSDBr,Fr;

; Binary operations common code

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

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

; dispatches on stkp-1, so Binpop1 = 1 mod 20B

17,20,Binpop,Binpop1,Binpop2,Binpop3,Binpop4,Binpop5,Binpop6,Binpop7,...........
11,2,BincomB,BincomA;
14,1,Bincomx;

shake IR+ in BincomA

value for dispatch into Binpop

L: value for push dispatch

stash briefly

make mACSOURCE produce 1

BincomB: L+ T+ stkp-1, :Bincomx;

Bincomx: stkp=L, L=T;

Bincomd: temp2=L, :Binpop;

BincomA: L=2000 OR T;

Binpop: IR=M, :BincomB;

Binpop1: T=stkl;

Binpop2: T=stkl, :Binend;

Binpop3: T=stkl, :Binend;

Binpop4: T=stkl, :Binend;

Binpop5: T=stkl, :Binend;

Binpop6: T=stkl, :Binend;

Binpop7: T=stkl, :Binend;

Binend: SINK=DISP, BUS;

SINK=temp2, BUS, :ADDr;

perform return dispatch

perform push dispatch
ADD - replace <TOS> with sum of top two stack elements

ADD: \( IR=T+ret0, :BincomB; \)
ADDr: \( L+M+T, mACSOURCE, TASK, :push0; \)

ADD01 - replace stk0 with \( stk0+stk1 \)

ADD01: \( T=stk1-1, \) ADD01x;
ADD01x: \( T=stk0+T+1, SH=0; \)
          \( L=stk-1, :pushT1B; \)

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

SUB: \( IR=T+ret1, :BincomB; \)
SUBr: \( L+M-T, mACSOURCE, TASK, :push0; \)

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

AND: \( IR=T+ret2, :BincomB; \)
ANDr: \( L+M AND T, mACSOURCE, TASK, :push0; \)

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

OR: \( IR=T+ret3, :BincomB; \)
ORr: \( L+M OR T, mACSOURCE, TASK, :push0; \)

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

XOR: \( IR=T+ret4, :BincomB; \)
XORr: \( L+M XOR T, mACSOURCE, TASK, :push0; \)
; MUL - replace <TOS> with product of top two stack elements
; high-order bits of product recoverable by PUSH
17,1,MULDIVcoma;
11,2,GoROMMUL,GoROMDIV;
17,2,MULx,DIVx;

shakes stack dispatch
also shakes bus dispatch

MUL: IR+T=ret5, :BincomB;
MULr: AC1=L, L+T, :MULDIVcoma;

MULDIVcoma: AC2=L, L+O, :MULx;

MULx: AC0=L, T+O+1, :MULDIVcomb;
DIVx: AC0=L, T+O, BUS=O, :MULDIVcomb;

MULDIVcomb: L=MULDIVretloc+T, SWMODE, :GoROMMUL;

GoROMMUL: PC=L, :ROMMUL;
GoROMDIV: PC=L, :ROMDIV;

MULDIVret: :MULDIVret1;
;
MULDIVret1: T=AC1;
L=stk+1;
L=T, SINK=M, BUS;
T=AC0, :dpush;
;

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

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

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

(TOS > 0) left shift, (TOS < 0) right shift

SHIFT:

SHIFTx:  IR=T=ret10, :BincomB;

SHIFTx:

SHIFT:

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

Rshift:  T+37, IR=37;

Shiftcom:

Shiftloop:

DoShift:

DoRight:

DoLeft:

Shiftdone:

Shiftdonex:

shifts stack dispatch

L: value, T: count

L: -count, T: count

IR= causes no branch

mask to reasonable size

equivalent to IR=msr0

mask to reasonable size

test for completion

dispatch to push result
Double-Precision Arithmetic

returns from DSUBsub

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

DADDX:
DADDBorrow, DADDCarry;

DADD:
T=stk2, :DADDx;
L=stkO+T;
stkO=L, ALUCY;
T=stk3, :DADDBorrow;
DADDCarry:
L=stk1+T, TASK, :DASStail;
DASStail:
T+2;
Dsetstkp:
L=stkO+T, TASK, :Setstkp;

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

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., stkO DSUB stk3., stk2) )

DCOMP:
IR=sr1, :DSUBsub;
DCOMPr:
L=stk1, BUS=0;
L=0+1, SHK0, :DCOMPnz;
DCOMPnz:
T+3, :DCOMPr;
DCOMPz:
L=stkO, BUS=0, :DCOMPnz;
DCOMPpos:
L=0+1, :DCOMPdone;
DCOMPdone:
stkO=L, :Dsetstkp;

Double-precision subtract subroutine

DSUB:
T=stk2, :DSUBx;
L=stkO-T;
stkO=L, ALUCY;
T=stk3, :DSUBborrow;
DSUB borrow, DSUBnoborrow;

DSUBx:
L=stkO-T;
stkO=L, ALUCY;
T=stk3, :DSUBborrow;
DSUBborrow:
L=stk1-T, IDISP, TASK, :DASStail;
L=high half of difference

stuff, test carry

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

high-order carry is lost
adjust stack pointer

positive
stash result
; Reads
;
; Note: RBr must be odd!

; Rn - TOS<<TOS>+n>

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

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

1,2,ReadB,ReadA; keep ball 1 in air

| RB: | IR=sr15, :Getalpha; 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 Opush |

; RDO - temp<<TOS>>, push <<temp>>, push <<temp>+1>

| RDO: | IR=sr32, :Popsub; returns to RDOr |
| RDOr: | L=0, :Opusha; |
; RILP - push <<lp>+alpha[0-3]+alpha[4-7]>
RILP:  L+retO, :Splitalpha;  get two 4-bit values
      T+lp, :RIPcom;  T:address of local 2

; RIGP - push <<gp>+alpha[0-3]+alpha[4-7]>
RIGP:  L+ret1, :Splitalpha;  shake IR- at WILPr
      T+gp+1, :RIPcom;  get two 4-bit values
      IR=msr0, :IPcom;  T:address of global 2
      IPcom:  T=3+T+1;  set up return to pushMD
      MAR=lefthalf+T;
      L=righthalf;
      IPcomx:  T+MD, IDISP;
      MAR=M+T, :pushMD;  T:address of local or global 0

; RILO - push <<lp>>>
RIL0:  MAR=lp-T-1, :RILxB;  start memory cycle
      RILxB:  IR=msrO, L+0, :IPcomx;  start fetch/store
      RILxA:  IR=srl, L=srl AND T, :IPcomx;  L:local/global value

; 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;
                righthalf+L, :RILPr;  L:TOS+alpha[4-7]
                now act like RILP
; Writes

; Wn - <<TOS> (popped)+n>+<<TOS> (popped)

11,2,WnB,WnA; keep ball 1 in air

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

WnB: IR=sr2, :Wsub;
WnA: IR=sr3, :Wsub;

; Write subroutine:

17,1,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;
WBr: :WnB;

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

17,1,WSBx; shake stack dispatch

WSB: IR=T+ret14, :BincomA;
WSBr: T=M, L~T, :WSBx;

WSBx: MAR=ib+T, :WScom;
WScom: temp=L;
WScoma: L=stkp-1;

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

17,1,WS0x; shake stack dispatch

WS0: IR=T+ret15, :BincomB;
WS0r: T=M, L~T, :WS0x;
WS0x: MAR=T, :WScom;
WILP: \[L \leftarrow \text{ret2}, :\text{Splitalpha};\] get halves of alpha
WILPr: \[\text{IR} \leftarrow \text{sr2};\] IPcom will exit to StoreB
T \leftarrow \text{lp}, :\text{IPcom}; prepare to undiddle

WXLP: \[L \leftarrow \text{ret4}, :\text{Splitalpha};\] get halves of alpha
WXLPra: \[\text{IR} \leftarrow \text{sr35}, :\text{Popsub};\] fetch TOS
WXLPrb: \[L \leftarrow \text{righthalf}+T, \text{TASK};\] L:TOS+alpha[4-7] now act like WILP

WOB: \[\text{temp} \leftarrow \text{alpha}+\text{TOS} \text{ (popped)}, \text{pop into} \text{ <temp>+1} \text{ and} \text{ <temp>}, \text{assuming:}\]
\[\text{WOB is A-aligned (also ensures no pending branch at entry)}\]

WOO: \[L \leftarrow \text{ret6}, \text{TASK}, :\text{Xpopsub};\] returns to Dpop
WOO: \[\text{L} \leftarrow \text{0}, :\text{Dpopa};\]

WSDB: \[\text{IR} \leftarrow \text{sr24}, :\text{Popsub};\] get low data word
WSDBra: \[\text{saveret} \leftarrow \text{L};\] stash it briefly
\[\text{IR} \leftarrow \text{ret20}, :\text{BincomA};\] alignment requires BincomA
WSDBrb: \[T \leftarrow \text{M}, L \leftarrow \text{WSDBx};\] start store of low data word
WSDBx: \[\text{MAR} \leftarrow \text{T+1}+\text{T}+1;\] temp:high data
temp \leftarrow L, L \leftarrow T;
temp2 \leftarrow L, \text{TASK}; temp2:updated address
MD \leftarrow \text{saveret}; stash low data word
MAR \leftarrow \text{temp2}+1, :\text{WScoma}; start store of high data word
; Unary operations

; INC - TOS + (TOS)+1

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

; NEG - TOS - (TOS)

NEG:     L=ret11, TASK, :Xpopsub;
NEGrr:   L=O-T, :Untail;

; DBL - TOS + 2*(TOS)

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

; Unary operation common code

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

; PUSH - add 1 to stack pointer
;-----------------------------------------------
11,1,PUSHx;

PUSH:   L=stkp+1, BUS=1, :PUSHx;
PUSHx:   SINK+ib, BUS=0, TASK=, :Setstkp;
         BUS checks for overflow
         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>
;-----------------------------------------------
11,1,DUPx;

DUP:     IR=sr2, :DUPx;
DUPx:     L=stkp, BUS=0, TASK=, :Popsuba;
         returns to pushTB
         don't pop stack

; EXCH - exchange top two stack elements
;-----------------------------------------------
11,1,EXCHx;

EXCH:    IR=ret11, :EXCHx;
EXCHx:    L=stkp-1;
        L=M+1, BUS=0, TASK=, :Bincomd;
        dispatch on stkp-1
        set temp2=stkp
EXCHr:   T=M, L=T, :dpush;
        Note: dispatch using temp2

; LADRB - push alpha+lp (undiddled)
;-----------------------------------------------
11,1,LADRBx;

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

; GADRB - push alpha+gp (undiddled)
;-----------------------------------------------
11,1,GADRBx;

GADRB:   IR=sr11, :Getalpha;
GADRBx:   T=gp+offset+T+1, :GADRBx;
GADRBx:   L=gp+T, :Untail;
         shake branch from Getalpha
         returns to GADRBx
; String Operations

17,1,STRsub; shake stack dispatch
11,2,STRsubA,STRsubB;
11,2,RSTRrx,WSTRrx;

STRsub: L=stkp-1; update stack pointer
stkp=L;
L=ib+T;
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
T=+count+T;
MAR=COUNT+T;
T=M;
SINK=DISP, BUSODD;
mask=L, SH<0, :RSTRrx;

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

11,2,STR, RSTRB,RSTRA;

RSTR: IR=T=ret12, :BincomB; stash string base address
RSTRr: temp=L, :STRsub;
RSTRrx: L=MD AND T, TASK, :RSTRB; isolate good bits
RSTRB: temp=L, :RSTRcom;
RSTRA: temp=L LCY 8, :RSTRcom;
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)

11,2,STR, WSTRA, WSTRB;

WSTR: IR=T=ret13, :BincomB; stash string base
WSTRr: temp=L, :STRsub;
WSTRrx: L=MD AND NOT T, :WSTRB; isolate good bits
WSTRB: temp2=L, L=ret0, TASK, :xpopsub;
WSTRA: temp2=L, L=ret0+1, TASK, :xpopsub;
WSTRrA: taskhole=L LCY 8;
WSTRrB: taskhole, :WSTRrB;

WSTRrB: T=mask.T;
L=temp2 OR T; retrieve string address
T=temp;
MAR=COUNT+T;
TASK;
MD=M, :nextA;
Field Instructions

; returns from Fieldsub
I1,2,RFrr,WFrr;
17,1,Fieldsub;
; 17,1,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;
RFrr:  L+=ret0, :Fieldsub;
RFrr:  T+=mask, :pushTA;

; WF - pop data in <TOS-1> into field specified by beta in word at <TOS> (popped) + alpha
;   if WF is A-aligned, B byte is irrelevant
;   alpha in B byte, beta in A byte of word after WF
WF:   IR+=ret17, :BincomB;
WFrr:  newfield+=L, L+=ret0+1, :Fieldsub;
WFrr:  T+=mask;
WFrr:  temp+=L;
WFrr:  T+=newfield.T;
WFrr:  L+=temp OR T, TASK;
WFrr:  CYCOUT+=L;
WFrr:  T+=index, BUS=0;
WFrr:  L+=WFretloc, :WFnzct;
WFnzct:  PC+=L;
WFnzct:  L+=20-T, SWMODE;
WFnzct:  T+=CYCOUT, :RAMCYCX;
WFret:  MAR+=frame;
WFret:  L+=stkp-1;
WFret:  MD+=CYCOUT, TASK, :JZNEBeq;

; WSF - like WF, but with top two stack elements reversed
;   if WSF is A-aligned, B byte is irrelevant
;   alpha in B byte, beta in A byte of word after WSF
WSF:  IR+=ret16, :BincomB;
WSFrr:  L+=T, T+=, :WFr;
RFS - like RF, but with a word containing alpha and beta on top of stack
if RFS is A-aligned, B byte is irrelevant

RFS: L+ret12, TASK, :Xpopsub; get alpha and beta
   temp=L; stash for WFSa
RFSra: L+ret13, TASK, :Xpopsub; T:address
RFSrb: L+ret10, BUS=0, :Fieldsub; returns quickly to WFSa

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

WFS: L+ret14, TASK, :Xpopsub; get alpha and beta
WFSra: temp=L; stash temporarily
WFSrb: IR=T+ret21, :BincomB; L:new data, T:address
WFSa: newfield=L, L+ret0+1, BUS=0, :Fieldsub; returns quickly to WFSa
      frame=L; stash address
      T=177400;
      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:
\begin{align*}
temp2 &+ L, L &+ T, IR + msr0, TASK, :Fieldsuba; \\
frame &+ L, :GetalphaA; \\
\end{align*}

Fieldsuba:
\begin{align*}
\text{stash return} \\
\text{stash base address} \\
T: \text{beta, } ib: \text{alpha} \\
\end{align*}

Fielddra:
\begin{align*}
L &+ ret5; \\
saveret &+ L, :Splitcomr; \\
\end{align*}

Fieldrdb:
\begin{align*}
T &+ righthalf; \\
MAR &+ MASKTAB + T; \\
T &+ lefthalf + T + 1; \\
L &+ 17 \text{ AND } T; \\
index &+ L; \\
L &+ MD, TASK; \\
mask &+ L; \\
T &+ frame; \\
L &+ MAR &+ ib + T; \\
frame &+ L; \\
L &+ Fielddretloc; \\
PC &+ L; \\
T &+ MD, SWMODE; \\
L &+ index, :RAMCYCX; \\
SINK &+ temp2, BUS; \\
L &+ T &+ CYCOUT, :RFrr; \\
\end{align*}

Fieldrc:
\begin{align*}
\text{stash return} \\
\text{stash base address} \\
T: \text{beta, } ib: \text{alpha} \\
\end{align*}

get two halves of beta
index for MASKTAB
start fetch of mask
L: left-cycle count
mask to 4 bits
stash position
L: mask for caller's use
stash mask
get base address
add alpha
stash updated address for WF
return location from RAMCYCX
data word into T for cycle
count to cycle, go do it
return dispatch
cycled data word in L and T