

```text
RunStart:
Bank = 0.
Root.
GOTOABS[GermRunHookup]  {off to initialize the Mesa world},
```
getClass:
(upon entry, ollow must contain the oop whose class is desired, there must be a pending XDisp to allow us to test SmallInterness, and L2 contains the return linkage. At exit, temp3Low will contain the class oop. If the class is NOT SmallInteger, temp1High/Low points at the base of the object. L1 may be smashed by this routine!! At exit, temp3Low contains the class of the object)
BRANCH[classListIsNotSmallInt, classIsSmallInt, 0e], c1;
classIsSmallInt:
  temp3Low = classSmallIntegerPointer, L2Disp,
  RET[getClass-return], c2;
classIsNotSmallInt:
  L1 = gettingClass,
  CALL[otMap], c3;
classAlreadyHaveBase:
  temp3Low = temp3Low + classFieldOffset,
  Noop, c1.
  at[getClass, 10, otMap-return],
  Noop, c2;
  MAR + [temp1High, temp1Low + 0] (read the class field),
  temp3Low = temp3Low + classFieldOffset, L2Disp,
  temp3Low = MD, RET[getClass-return], c3;

getNewMethodHeader:
(upon entry, temp1High/Low must be high address of new compiledMethod. L2 contains the return linkage. At exit, temp3Low and uNewMethodHeader are both the new method header and uNewMethodHigh/Low are correct)
  uNewMethodLow = temp3Low, c2;
  temp3Low = temp3Low + objectHeaderSize, c3;
  MAR + [temp1High, temp1Low + 0], (start read of method header),
  uNewMethodHigh = 0 (finish saving CompiledMethod start address),
  temp3Low = MD (method header), L2Disp,
  uNewMethodHeader + temp3Low, RET[getNewMethodHeader-return] c1;

getDeltaWord:
(upon entry temp2High/Low must be pointing at the delta word of an object. L1 is the linkage register. The word is returned in temp3Low)
  MAR + [temp2High, temp2Low + 0], c1;
  L1Disp,
  temp3Low = MD, RET[getDeltaWord-return], c2;

returnTopOfStack:
(upon entry, there must be a pending ZoroDr [non-zero implies smash the top of stack, zero implies do not smash it]. L1 is the return linkage register, and the top of stack is returned in otlow)
  MAR + [stackHigh, stackLow + 0] (read top of stack), BRANCH[smashTOS, dontSmashTOS], c1;
  smashTOS:
  MD = nilPointer, c2;
  otlow + MD, c3;
  stacklow = stackLow - 1, c1;
  L1Disp, c2;
RET[returnTopOfStack-return],
dontSmashTOS:
  L1Disp,
  atlow + MD, RET[returnTopOfStack-return],

pushTemp3LowAndDispatch:
  MAR = [stackHigh, stackLow + 0],
  MDR = temp3Low, NextBytecode,
  DISP[bytecodes], 1plow + 1plow + PC16,

notYetInvented:
  Noop,
  temp1ow + 1 (mark notYetInvented), GOTO[saveSmalltalkState],

bytecodeFailed:
  Noop,
  temp1ow + 2 (mark bytecode failure), GOTO[saveSmalltalkState],

transferWords:
  (upon entry, temp1High/low must be the destination start address, temp2Low is the low 16 bits of the source limit address, and stackHigh/low is the source start address. L1 is the return linkage register. Q is smashed. Because we are moving between two volatile contexts, no reference counting is performed)
  temp2Low = temp1ow + 1 (adjust limit to one past the last word),

transferWordsCheck:
  [ ] = stackLow - temp2Low, ZeroBr,
  BRANCH[$, returnFromTransferWords],
  MAR = [stackHigh, stackLow + 0] {read and smash receiver or argument}, CANCELBR[$, OF], c1;
  MDR = atPointer,
  Q = MD,
  MAR = [temp1High, temp1ow + 0] {put into new context},
  MDR = Q,
  temp1ow = temp1ow + 1 (up destination index),
  stackLow = stackLow + 1, GOTO[transferWordsCheck],

returnFromTransferWords:
  stackLow = stackLow - 1 (the last word actually moved),
  temp2Low = temp2Low - 1, L1Disp,
  RET[transferWords-return].
Make all free chunk lists empty by storing notAnObject (8001) at the head of each list.

```
toCompress:
  runHigh + uRunRecordHigh,          c1;  (point to first free chunk list)
  runLow + uRunRecordLow,            c2;
  runLow + runLow + freelistsOffset, c3;
  temp3Low = largestFreeChunkSize,   c1;  (set list counter)
  temp2Low = runLow + runLow + 1,    c2;  (create list terminator = 8001)
  temp2Low + temp2Low = RNot1,       c3;
emptyFreeChunkLoop:
  MAR + [runHigh, runLow + 0],       c1;  (mark each list)
  MDR + temp2Low, runLow + runLow + 1, c2;
  Noop,                               c3;
  Noop,                               c1;
  tempLow = temp3Low - 1, NegBr,      c2;
  BRANCH [emptyFreeChunkLoop, $].     c3;
```

Make real object relocatable by storing its oop in its class field and its class in its offset field. Mark free chunks and reclaim free oops. Test purpose bits in the OT entry and do one of the following.

00: Offset field = class, class field = oop.
01: Fail1.
10: Fail1.
11: Add oop to free oop list. If refc < 0 (free object), make link odd (class field = 1).

```
reverseOT:
  otlow - 2,                           c1;  (start at bottom of OT)
  Noop,                               c2;
  runLow + uRunRecordLow,              c3;
MAR + [runHigh, runLow + objectTableHighOffset], c1;  (omap oop)
  CANCELBR [$, 2],                    c2;
  othigh + MD,                        c3;
reverseOTLoop:
  We can get by with the -1 because a page cross is impossible --
  the OT entry must be at an even address, and only incrementing an
  odd address can cause a page fault)
  MAR + [othigh, otlow + 1],          c1;
  CANCELBR [$, 2],                    c2;
  Q = objectHigh + MD, XdwDisp,       c3;  (dispatch on purpose bits)
  MAR + [othigh, otlow + 0]. DISP2 [purposeTable], c1;
impossiblePurposeBits1:
  GOTO [bailout3].                     c2, at [1, 4, purposeTable];
impossiblePurposeBits2:
  GOTO [bailout3].                     c2, at [2, 4, purposeTable];
normalObject:
  (class field + oop, offset + class)
  Noop,                               c2, at [0, 4, purposeTable];
optLow + optLow + classFieldOffset,   c3;
  Noop,                               c4;
  Noop,                               c5;
  MAR + [objectHigh, objectLow + 0],  c1;
  MDR + optLow,                       c2;  (object = oop)
  temp3Low + MD, Xdisp,               c3;
  MAR + [othigh, otlow + 0]. BRANCH [$, impossibleClass, OE], c1;
  MDR + temp3Low, GOTO [nextOop],     c2;  (OT = class)
impossibleClass:
  GOTO [bailout3].                     c2;  (class field can't be odd)
freeOopOrObject:
  (do nothing to free oop; add free object's oop to free oop list and mark free object)
  [ ] = 0, NegBr,                     c2, at [3, 4, purposeTable];
  objectLow + MD, BRANCH [freeOop, $]. c3;
  MAR + [runHigh, runLow + freePointersOopOffset], c1;
  MDR + otlow, CANCELBR [$, 2], LOOPHOLE [wok], c2;  (add new free oop)
  temp3Low + MD,                       c3;
  MAR + [othigh, otlow + 0],          c1;
  MDR + temp3Low,                      c2;  (fix link field)
optLow + optLow + classFieldOffset,   c3;
  MAR + [othigh, otlow + 1],          c1;
  MDR + 80, CANCELBR [$, 2], LOOPHOLE [wok], c2;  (fix flag field)
  Noop,                               c3;
  MAR + [objectHigh, objectLow + 0],  c1;
```
MDR = 1. GOTO [nextOop], c2; (mark object free)

nextOop:
Woop, c3;

freeOop:
Woop, c1;
oLow = oLow + 2, CarryBr, c2;
BRANCH [reverseOTloop, $], c3;

{ Move all objects as low in object space as possible without straddling a bank boundary.
    oHigh = OT base high (object memory limit)
    sourceHigh = sourceLow + object source
    destHigh = destLow = object destination
    runHigh, runLow = run record base
}

compress:
{initialize the pointers}
MAR = [runHigh, runLow + objectMemoryHighOffset], c1;
CANCELBKR [$. 2], c2;
sourceLow = sourceHigh + MD, c3;

MAR = [runHigh, runLow + objectMemoryLowOffset], c1;
destHigh = sourceLow LRot8, CANCELBKR [$. 2], c2;
sourceLow = MD, c3;
destLow = sourceLow, c1;

compressLoop:
sourceLow = sourceLow + classFieldOffset, c2; (sourceLow + class field)
Woop, c3;

MAR = [sourceHigh, sourceLow + 0], c1;
sourceLow = sourceLow - offsetFromClassFieldToClassField, c2;
oLow = MD, XDisp, c3;

MAR = [sourceHigh, sourceLow + 0], BRANCH [. skipFreeChunk, 0E], c1;
sourceLow = sourceLow - sizeFieldOffset, c2;
size = MD, c3;
(will it fit at destination?)
Q = destLow + size, ZeroBr, c1;
Q = destLow + size, CarryBr, BRANCH [. perfectFit], c2;
tempLow = oLow, BRANCH [. $, mdbr], c3;
Q = Q - objectHeaderSize, CarryBr, c1;
BRANCH [unreverseOTEntry, newDestinationBank], c2;

perfectFit:
sourceLow = sourceLow + classFieldOffset, CANCELBKR [woe2, 1], c3;

skipFreeChunk:
sourceLow = sourceLow - sizeFieldOffset, c2; (sourceLow + tfirst field)
size = MD, c3;
[ ] + size = objectHeaderSize, CarryBr, BRANCH [. impossibleSize1, $.], c1;
sourceLow = sourceLow + size, CarryBr, GOTO [newSourceBank], c2;

impossibleSize1:
GOTO [bailout1], c3; (hang if (0 <= size < objectHeaderSize))

impossibleSize2:
GOTO [bailout1], c3; (hang if (0 <= size < objectHeaderSize))

(newDestinationBank)
tempLow = oLow, GOTO [mdbr2], c3;

mdb2:
temp2Low = -destLow, c1;
L1 = remainderFree, c2;
CALL [newFreeChunk], c3;
oLow = runLow, c1, at [remainderFree, 10, addToFreeChunkList-return];
destLow = destHigh, c2;
set new bank
destHigh = destLow + 1 LRot10, LOPHOLE [nib1Timing], c3;
destLow = 0, c1;
runLow + uRunRecordLow, GOTO [unreverseOTEntry], c2;

unreverseOTEntry:
sourceLow = sourceLow + classFieldOffset, c3;

woe2:
MAR = [oHigh, oLow + 0], c1; (set offset)
MDR = destLow, c2;

compactor.mc  2-Aug-84 11:30:08 PDT
temp2Low = NO;

MAR = [sourceHigh, sourceLow + 0],
MDR = temp2Low,
sourceLow + classFieldOffset,
MAR = [artHigh, artLow + 1],
temp3Low = 0, CANCELBR [S, 2],
temp2Low = NO and temp3Low,

Q = destHigh, BRANCH [ImpossibleSize2, $],
Noop.
MAR = [artHigh, artLow + 1],
MDR = temp2Low or 0, CANCELBR [S, 2], LOOPHOLE [wok],
[3] = Q xor sourceHigh, ZeroBr,
[3] = destLow xor sourceLow, ZeroBr, BRANCH [moveObject1, $],
BRANCH [$, noMove],
GOTO [no10],

moveObject1,
CANCELBR [$, 1],
GOTO [no10],

noMove:
sourceLow = sourceLow + size, ZeroBr,
BRANCH [$, nn2],
GOTO [nn4],

mm2:
destHigh = Q + 1 LRot0, LOOPHOLE [nte1Timing],

mm4:
destLow = sourceLow, ZeroBr, GOTO [newSourceBank],

moveObjectLoop:
Noop.

mo10:
MAR = [sourceHigh, sourceLow + 0],
sourceLow = sourceLow + 1,
temp2Low = NO,

MAR = [destHigh, destLow + 0],
MDR = temp2Low, destLow + destLow + 1, ZeroBr,
Q = destHigh, BRANCH [mo12, $],

destHigh + Q + 1 LRot0, LOOPHOLE [nte1Timing],
[3] = size - 1, ZeroBr,
BRANCH [bankStraddle, $],
size = size - 1, ZeroBr, GOTO [no14],

mo12:
size = size - 1, ZeroBr,

mo14:
BRANCH [moveObjectLoop, $],
[3] = sourceLow, ZeroBr, GOTO [newSourceBank],

newSourceBank:
Q = sourceHigh, BRANCH [compressLoop3, $],
sourceHigh = Q + 0 + 1 LRot0, LOOPHOLE [nte1Timing],
Q = artHigh xor 0, ZeroBr,
BRANCH [compressLoop2, carveRemainingFreeSpace],

bankStraddle:
GOTO [bailout2],

compressLoop3:
sourceLow = sourceLow + classFieldOffset, GOTO [c12],

compressLoop2:
sourceLow = sourceLow + classFieldOffset, GOTO [c12],

carveRemainingFreeSpace:
temp2Low = -destLow,
Q = temp2Low + objectHeaderSize, CarryBr.
BRANCH [$, carveFreeBank],
temp2Low = temp2Low + 1, L1 = carveFree1,
CALL [newFreeChunk],
runLow = uRunRecordLow,
Q = destHigh,
destHigh = Q + 0 + 1 LRot0, LOOPHOLE [nte1Timing],
Noop,
The code segment appears to be a part of a program that manipulates memory chunks. Here's a breakdown of the code:

```assembly
Q = Q xor otHigh, ZeroBr,
destLow = 0, BRANCH [carveFreeBankLoop1, finis1],
carveFreeBank:
    tempLow = objectHeaderSize, L1 = carveFree2, GOTO [cfb14], c2;
    (remainder size is too big for one chunk)
carveFreeBankLoop2:
    tempLow = objectHeaderSize, L1 = carveFree2, GOTO [cfb12], c1;
carveFreeBankLoop1:
    tempLow = objectHeaderSize, L1 = carveFree2, GOTO [cfb12], c1;
cfb12:
    Noop, c2;
cfb14:
    size = temp2Low, CALL [newFreeChunk], c3;
destLow = destLow + size, Noop, Noop, c1;
    at [carveFree2, 10, addToFreeChunkList-return]; c2;
    set new bank c3;
    runLow = uRunRecordLow, temp2Low = destLow, L1 = carveFree3, c1;
    CALL [newFreeChunk], c2;
    runLow = uRunRecordLow, Q = destHigh, c3;
    destHigh = Q = Q + 1 LRot0, LOOPHOLE [nbitTiming], c2;
    Noop, c2;
    Q = Q xor otHigh, ZeroBr, c2;
    destLow = 0, BRANCH [carveFreeBankLoop2, finis2], c1;
finis1:
    Noop, c1;
finis3:
    Noop, GOTO [oopsLeft], c2;
    c3;
finis2:
    GOTO [finis3], c1;

{ Subroutine to create a new free chunk on the appropriate list. }
inputs: run points to run record dest points to free chunk base temp2Low is size of chunk
smashes: Q, temp2, temp3, runLow
saves: input temp2Low in runLow }
newFreeChunk:
    Noop, c1;
    Noop, c2;
    destLow = destLow + sizeFieldOffset, c3;
    MAR = [destHigh, destLow + 0], c1;
    (set free chunk size)
    MDR = destLow - sizeFieldOffset, c2;
    CANCELBR [$, 2], c3;
    oLow + MDR, c2;
    MAR = [runHigh, runLow + FreePointersOopOffset], c1;
    (get free oop)
    CANCELBR [$, 2], c2;
    oLow + MDR, c3;
    MAR = [orHigh, oLow + 0], c1;
    MDR = destLow, c2;
    temp2Low = MDR, c3;
    (set free chunk offset)
    MAR = [orHigh, oLow + 1], c1;
    MDR = destHigh, CANCELBR [$, 2], LOOPHOLE [wok], c2;
    (get free oop link)
    Noop, c3;
    MAR = [runHigh, runLow + FreePointersOopOffset], c1;
    MDR = temp2Low, CANCELBR [$, 2], LOOPHOLE [wok], c2;
    CANCELBR [$, 2], c3;
    (set free chunk bank)
    runLow + tempLow, GOTO [returnTopo], c2;
    (add chunk to free list)

{ Count the oops on the free oop list. Then count the objects on the free chunk lists. }
oopsLeft:
    MAR = [runHigh, runLow + FreePointersOopOffset], c2;
    stackLow = 0, CANCELBR [$, 2], c3;
    oLow + MDR, c2;
    oLow + MDR, c1;
    (terminator is odd)
    MAR = [orHigh, oLow + 0], c3;
    MDR = stackLow + 1, c2;
    CANCELBR [$, countFreeChunks, 0F], c3;
    Noop, c3;
    MAR = [orHigh, oLow + 0], c1;
    stackLow = stackLow + 1, c2;
    oLow + MDR, GOTO [oopCountLoop], c3;
    oopCountLoop:
        [] = oLow LRot0, XDisp, c1;
        BRANCH [$, countFreeChunks, 0F], c2;
        (terminator is odd)
    c3;
countFreeChunks:
    GOTO [wordsLeft], c3;

{ }
```
Count all the free space on the free chunk lists.
}

wordsizeleft:
  rnumLow + rnumLow + freelistOffset,
temp3High = 0, temp3Low = 0,
temp3Low = largestFreeChunkSize,
c1:
c2:
c3:

wordCountOuterLoop:
  MAR = [rnumHigh, rnumLow + 0],
  Noop,
  otlow + MD,
c1:
c2:
c3:

wordCountInnerLoop:
  [()% + otlow LRot0, XDisp,
  Q = temp3High, BRANCH [$, nextlist, OE],
  stackLow = stackLow + 1,
  MAR = [atHigh, otlow + 0],
  Q = Q + 1,
  objectLow + MD,
  MAR = [otHigh, otlow + 1],
  objectLow + objectLow + sizeFieldOffset, CANCELBR [$. 2],
  objectHigh + MD,
  MAR = [objectHigh, objectLow + 0],
  objectLow + objectLow + offsetFromSizeFieldToClassField,
  size + MD,
  temp3Low + temp3Low + size, CarryBr,
  BRANCH [$, wci12],
  GOTO [wci14],

wci12:
  temp3High = Q LRot0,
c3:

wci14:
  MAR = [objectHigh, objectLow + 0],
  Noop,
  otlow + MD, GOTO [wordCountInnerLoop],
c1:
c2:
c3:

nextlist:
  Noop,

  Noop,

  temp3Low + temp3Low - 1, NegBr,
  rnumLow + rnumLow + 1, BRANCH [wordCountOuterLoop, $.],
c1:
c2:
c3:

setTextLevel:
  rnumLow = uNumRecordLow,
  Noop,
  Noop,
c1:
c2:
c3:

MAR = [rnumHigh, rnumLow + wordLevelLowOffset],
MDR = temp3Low, CANCELBR [$. 2], LOOPHOLE [wok],
Q = MD,
MAR = [rnumHigh, rnumLow + wordLevelHighOffset],
MDR = temp3High, CANCELBR [$. 2], LOOPHOLE [wok],
Q = MD,
c1:
c2:
c3:
c4:
c5:
c6:

defSetClass:
  MAR = [rnumHigh, rnumLow + oopLevelLowOffset],
  MDR = stackLow, CANCELBR [$. 2], LOOPHOLE [wok],
  Q = MD,
  MAR = [rnumHigh, rnumLow + oopLevelHighOffset],
  MDR = 0, CANCELBR [$. 2], LOOPHOLE [wok],
  Q = MD, GOTO [restoreHeapState],
c1:
c2:
c3:
c4:
c5:
c6:

compactor.mc 2-Aug-84 11:30:08 PDT
{ 13-Mar-84 18:29:29 }

{ The following register equates are used only to initialize Rm and to save and restore the Mesa emulator state: }

RegDef[StateL, U, 30];
RegDef[StateG, U, 48];
RegDef[StatePC, U, 50];
RegDef[StatePC16, U, 52];
RegDef[StateIB, U, 63];
(Reg[StateIB] gets saved in the STK)
RegDef[StateRNDS, U, 65];
RegDef[StateRL, U, 33];
RegDef[StateRNG, U, 46];
RegDef[StateRHP, U, 61];

{ The following equates are copied from Dandelion.df and are only used to save and restore the Mesa emulator state. }

RegDef[TOS, R, 0];
RegDef[RNDS, RH, 0];
RegDef[L, R, 3];
RegDef[RL, RH, 3];
RegDef[G, R, 4];
RegDef[RNG, RH, 4];
RegDef[PC, R, 6];
RegDef[RHPC, RH, 6];

(todo --- we can probably do better saving the RH registers)

(fromMesa:

STK + TOS, push.  
MesaStateL + L.  
MesaStateG + G.  
MesaStatePC = PC,  
PC = LShift3 PC, SE = pc16,  
MesaStatePC16 = PC,  

(now save the RH registers--somewhat messy since you can't write U reg directly from an RH register)

G = RNDS.  
MesaStateRNDS + G.  
L = RL.  
MesaStateRL + L.  
G = RNG.  
MesaStateRNG + G.  
Pc = RHPC.  
MesaStateRHPC + PC.  
Pc = ErrlnBPsp.  
PC = PC LRot12.  
PC = -PC. (Get it back to its non-negated state)  
MesaStateIBptr + PC, YDisp.  
DISPA[drainIB, 0c].  
GOTO[drain]. (state = empty, nothing to do)  
PC = 1b, (state = byte, grab it)  
MesaStateIB = PC,  
Moop.  
GOTO[drain].

IBspin:  
GOTO[IBspin]. (state = full, cannot happen...).  
PC = 1b. (state = word, get first byte)  
PC = PC LRot8.  
L + 1b. (get second byte)  

}
drained:
  Noop,
  LDDisp,
  DISP2[doBytecodesOrStabilization],
  GOTO[getSmalltalkState],
  GOTO[timeToStabilize],
  GOTO[timeToCompress], c2;

restoreMesaState:

{restore the RH registers}
  RHDS = MesaStateRHDS,
  rhl = MesaStateRhl,
  rhG = MesaStateRhG,
  rhPC = MesaStateRhPC, c1;

{and now the R registers}

{restoring the PC16 register for Mesa is tricky. Can’t assign to it directly, so we must read it to determine it’s current state, but reading it toggles the state......}
  PC = MesaStatePC16, XDisp,
  X2CpcDisp, BRANCH[pcl16wasZero, pcl16wasOne, 0e],
  pcl16wasZero:
  BRANCH[$, wasZeroIsOk, 0e],
  Cin = pcl16, GOTO[okPc],
  wasZeroIsOk:
  GOTO[okPc],
  pcl16wasOne:
  BRANCH[wasOneIsOk, $, 0e],
  Cin = pcl16, GOTO [okPc],
  wasOneIsOk:
  GOTO[okPc],

okPc: {now restore the IB state}
  IBPtr = 1, {drain any smalltalk bytecodes},
  PC = IB,
  Noop,
  PC = MesaStateIBptr, XDisp,
  DISP2[restoreIB, 0c],
  GOTO[IBdone], {state was empty},
  PC = MesaStateIB,
  IB = PC lrot0,
  IBPtr = 1, {state was byte},
  IB = MesaStateIB, {state was word}, GOTO[IBdone],
  IBdone:
  PC = MesaStatePC,
  L = MesaStateL,
  G = MesaStateG,
  Noop,

punt:
  Bank = 0,
  Noop,
  GOTOABS[returnFromRunBank],
RunInitialization:
    uRunRecordHigh + TOS,
    Q + STK, pop,
    uRunRecordLow + Q,
    Bank + 0,
    Noop,
    GOTOABS[returnFromRunBank],

                        c1; c2; c3;

            timeToStabilize:
        LO = mesaRequestedStabilization,
    osHigh + 0b,
    CALL[stabilize],
    Noop,
    stabilize-return];
    Noop.
    GOTO[restoreMesaState],

                        c1; c2; c3;

{Jump

jump1:  temp3low + 2, GOTO[jumpUnconditionally].

jump2:  temp3low + 3, GOTO[jumpUnconditionally].

jump3:  temp3low + 4, GOTO[jumpUnconditionally].

jump4:  temp3low + 5, GOTO[jumpUnconditionally].

jump5:  temp3low + 6, GOTO[jumpUnconditionally].

jump6:  temp3low + 7, GOTO[jumpUnconditionally].

jump7:  temp3low + 8, GOTO[jumpUnconditionally].

jump8:  temp3low + 9, GOTO[jumpUnconditionally].

} c1, bytecode[90];

(Pop and Jump on False)

popAndJump0OnFalse:  temp3low + 2, GOTO[popAndJump0OnFalse].

popAndJump1OnFalse:  temp3low + 3, GOTO[popAndJump1OnFalse].

popAndJump2OnFalse:  temp3low + 4, GOTO[popAndJump2OnFalse].

popAndJump3OnFalse:  temp3low + 5, GOTO[popAndJump3OnFalse].

popAndJump4OnFalse:  temp3low + 6, GOTO[popAndJump4OnFalse].

popAndJump5OnFalse:  temp3low + 7, GOTO[popAndJump5OnFalse].

popAndJump6OnFalse:  temp3low + 8, GOTO[popAndJump6OnFalse].

popAndJump7OnFalse:  temp3low + 9, GOTO[popAndJump7OnFalse].

popAndJump0OnFalse:  temp3low + uCurrentMethodIdLow, backupIs0Bytes, L1 + branchIfFalse, GOTO[jump-getTopOfStack]

} c2;
c3;

(Extended Jump)

extendedJump0:  temp3low + 0fc, GOTO[extendedJump].

extendedJump1:  temp3low + 0fd, GOTO[extendedJump].

extendedJump2:  temp3low + 0fe, GOTO[extendedJump].

extendedJump3:  temp3low + 0ff, GOTO[extendedJump].

extendedJump4:  temp3low + 00, GOTO[extendedJump].

extendedJump5:  temp3low + 01, GOTO[extendedJump].

extendedJump6:  temp3low + 02, GOTO[extendedJump].

} c1, bytecode[0a0];
c1, bytecode[0a1];
c1, bytecode[0a2];
c1, bytecode[0a3];
c1, bytecode[0a4];
c1, bytecode[0a5];
c1, bytecode[0a6];
extendedJump7:
  temp3Low = 03, GOTO[extendedJump].
  c1, bytecode[0a7];

extendedJump:
  temp3Low = temp3Low LRot8 (get the left byte correct),
  temp3Low = temp3Low + 1b + 1,
  iflow + iflow + PC16 (account for the extension byte), GOTO[jumpUnconditionally], c1;

{Extended Jump on True}

extendedJumpOnTrue0:
  temp3Low = 0, l1 = branchIfTrue, GOTO[extendedJumpTrue].
  c1, bytecode[0a8];

extendedJumpOnTrue1:
  temp3Low = 1, l1 = branchIfTrue, GOTO[extendedJumpTrue].
  c1, bytecode[0a9];

extendedJumpOnTrue2:
  temp3Low = 2, l1 = branchIfTrue, GOTO[extendedJumpTrue].
  c1, bytecode[0a8];

extendedJumpOnTrue3:
  temp3Low = 3, l1 = branchIfTrue, GOTO[extendedJumpTrue].
  c1, bytecode[0a8];

extendedJumpTrue:
  temp3Low = temp3Low LRot8 (get high bits of displacement), GOTO[extendedJumpOnCondition], c2;

{Extended Jump on False}

extendedJumpOnFalse0:
  temp3Low = 0, l1 = branchIfFalse, GOTO[extendedJumpFalse].
  c1, bytecode[0a8];

extendedJumpOnFalse1:
  temp3Low = 1, l1 = branchIfFalse, GOTO[extendedJumpFalse].
  c1, bytecode[0a9];

extendedJumpOnFalse2:
  temp3Low = 2, l1 = branchIfFalse, GOTO[extendedJumpFalse].
  c1, bytecode[0a8];

extendedJumpOnFalse3:
  temp3Low = 3, l1 = branchIfFalse, GOTO[extendedJumpFalse].
  c1, bytecode[0a8];

extendedJumpFalse:
  temp3Low = temp3Low LRot8 (get high bits of displacement), GOTO[extendedJumpOnCondition], c2;

extendedJumpOnCondition:
  temp3Low = temp3Low + 1b (and low bits) + 1 (and low bits), backupIs[Byte].
  c3;
  iflow + iflow + PC16, (account for extension byte)
  temp3Low = uCurrentMethodLow (point at base of method), GOTO[jump-getTopOfStack].
  c1;
  c2;
  c3;

jump-getTopOfStack:
  (upon entry, temp3Low must contain the number of bytes by which to adjust the instructionPointer. temp3Low must point at
  the base of the object header. pc16 and temp3Low must both be correct. l1 contains the constant branchIfTrue or
  branchIfFalse)
  MAR = {stackHigh, stackLow = 0} (read top of stack),
  MDR = nilPointer, (and smash it)
  temp3Low = MO,
  c1;
  c2;
  c3;
  [0] = temp3Low xor falsePointer, ZeroBr,
  [1] = temp3Low xor truePointer, ZeroBr, BRANCH[$, toIsFalse],
  BRANCH[tosIsNotBoolean, toIsTrue].
  c1;
  c2;
  c3;
tosIsFalse:
  CANCELBR[$, 1], stackLow = stackLow - 1, l1Disp, (do we want to branch)
  c3;
  DISP2[falseTos],
  NextBytecode, (no branch)
  DISP4[bytecodes], iplow + iplow + PC16,
  c1;
  c2, at[jumpIfTrue, 4, falseTos]:
  c3;
  Jump.mc  16-Oct-83 18:23:38 PDT
Jump.mc  16-Oct-83 18:23:38 PDT

iplow + iplow - temp2Low,
GOTO[alterIp] (take branch).

tosIsTrue:
stacklow + stacklow - 1, L1Disp, {do we want to branch}
DISP[trueTos],
1plow + iplow - temp2Low, GOTO[alterIp] (take branch),
Noop (no branch),
Noop,
NextBytecode,
DISP[bytecodes], iplow + iplow + PC16,
c2, at[branchIfFalse, 4, falseTos];
c3; c1;
c2; c3; c1;
c2; c3;

tosIsNotBoolean:
{rats, put the stack back}
{todo -- more of this when send is implemented}
MAR + [stackHigh, stacklow + 0],
MHR + temp1Low,
Noop,
GOTO[bytecodeFailed],
c1; c2; c3;

jumpUnconditionally:
temp2Low = uCurrentMethodLow,
iplow + iplow - temp2Low,
c2; c3;

alterIp:
{upon entry, temp2Low must contain the new displacement into the object including the object header, temp2Low must be the base of the current method header. iplow must be the current word offset into the compiledMethod exclusive of the header. pc16 must be correct}
1plow = LShift1 iplow, SE + pc16, {yields current byte offset into compiled method including object header},
temp3Low = temp2Low + iplow {calculate the new location},
iplow = temp2Low (point at base of object header), IBPtr = 1 (drain all bytecodes from instruction buffer), c3;
Noop,
Noop,
Ybus + 4b, GOTO[fixupInstructionPointer],
c1; c2; c3;
LoRes.mc 27-Sep-83 12:00:00 PDT

{filed on: LoRes.mc
 by col: 27-Sep-83 12:00:48
}

Reserve[1.7F];
    { E N D }
{  
  (For each of these three entry points, uClassToInstantiate must be the oop of the class, temp3Low is the size in words or bytes, temp3High is the return linkage register!!!)

  createInstanceWithPointers:
  temp2Low = n1Pointer,  
  temp1Low = n1Pointer, GOTO[createInstance], c1;  
  temp1Low = n1Pointer, GOTO[createInstance], c2;

  createInstanceWithBytes:
  temp2Low = 0,  
  [] = temp3Low Left 0, XDisp,  
  temp3Low = temp3Low + 1, BRANCH[byteCountIsEven, byteCountIsOdd, 0e]., c1;  
  byteCountIsEven:
  temp1Low = 0, GOTO[byteShift], c1;  
  byteCountIsOdd:
  temp1Low = 1, GOTO[byteShift], c1;  
  byteShift:
  temp3Low = RShift1 temp3Low, SE = 0, GOTO[createInstance], c2;

  createInstanceWithWords:
  temp2Low = 0,  
  temp1Low = 0, GOTO[createInstance], c1;

  createLargePositiveInteger:
  temp1Low = classLargePositiveIntegerPointer,  
  uClassToInstantiate = temp1Low,  
  GOTO[createInstanceWithBytes], c1;  
  uClassToInstantiate = temp1Low, GOTO[createInstanceWithBytes], c2;

  createInstance:
  (temp3Low is number of words in object less objectHeaderSize,  
  temp1Low is the value of the low two bits of the delta word,  
  temp2Low is the default value with which to initialize the object)
  uFieldtype = temp1Low  
  (save these for initializing the object and its ot entry), c3;  
  uDefault = temp2Low,  
  temp3Low = temp3Low + objectHeaderSize, CarryBr,  
  0 = objectSizeTestLim1, BRANCH[2, massiveSeparation], c2;  
  [] = temp3Low + 0, CarryBr,  
  BRANCH[$, requestedSizeTooBig], c1;  
  temp2High = uNumRecordHigh,  
  temp2Low = uNumRecordLow,  
  temp1Low = largestFreeChunkSize, c1;  
  [] = temp3Low = temp1Low, CarryBr, c2;

  uRequestedSize + temp3Low, BRANCH[trySpecificList, useBigFreeList], c1;  

  trySpecificList:
  temp2Low = temp2Low + freeListsOffset,  
  Nop,  
  MAR = [temp2High, temp2Low + temp3Low],  
  CANCELBR[2, 2], c1;  
  otLow = MB, XDisp, L2 = creatingInstance  
  (set up for the nextFreeChunk call), c2;  
  uNewObject + otLow, BRANCH[gotOne, tryBigList, 0e], c1;  

  gotOne:
  L1 = getNextFreeChunk (needed for otMap2 call inside nextFreeChunk), CALL[nextFreeChunk], c1;  

  MAR = [temp2High, temp2Low + temp3Low] (update free list head)  
  at[creatingInstance, 10,  
  nextFreeChunk-return], c1;  
  NDR = 0, CANCELBR[2, 2], LOOPHOLE[wok], c2;  
  (save new object's address)  
  0 = temp1High, c3;  
  uNewObjectHigh = 0, c1;  
  uNewObjectLow = temp1Low, c2;  
  GOTO[allocate], c3;

  tryBigList:
  temp2Low = uNumRecordLow, GOTO[useBigFreeList], c1;  

  useBigFreeList:
MemoryMangement.mc 2-Aug-84 9:24:20 PDT

(upon entry, temp2High/low contains the run record address, usRequiredSize is valid, temp3Low is the requested size)

Noop.

c2:

useBigFreeList:
upPredecessor = -oltLow xor oltow (yields -1),
L1 = gettingNextFreeChunk (for stmp call in nextFreeChunk),
MAR = [temp2High, tempLow + bigFreeListOffset],
L2 = consideringBigChunks, temp3Low = temp3low + objectHeaderSize, CarryBr, (yields minimum splitable block size) CANCELBR[2],
oftow = MD (current free chunk), XDisp (test if more),
BRANCH[$, massiveSentlity4].
c1:
c2:
c3:

considerNextBigFreeChunk:
unNewObject = oltow, BRANCH[$, outOfChucks, 0e],
unCurrentFreeChunkTop = oltow, CALL[nextFreeChunk].
c1:
c2:

unNextFreeChunk = 0 (remember next free chunk),
nextFreeChunk-return;
tempow - tempow + sizeofFieldOffset, Noop.
c2:
c3:

MAR = [temp1High, temp1Low + 0],
Noop.
c1:
c2:
0 + MD (size of current free chunk).
c3:

[1] = 0 xor usRequiredSize, ZeroBr.
[2] = 0 - temp3Low, CarryBr, BRANCH[$, exactFit],
BRANCH[iterate, canSubdivide].
c1:
c2:
c3:

iterate:
upPredecessor = oltow,
Noop.
c1:
c2:
oftow = unNextFreeChunk, XDisp, GOTO[considerNextBigFreeChunk].
c3:

exactFit:
tempow - tempow + sizeofFieldOffset, CANCELBR[1],
Q = temp1High, unNewObjectLow = tempLow,
unNewObjectHigh = 0, GOTO[splice].
c1:
c3:

canSubdivide:
temp3Low = usRequiredSize,
temp3Low = 0 (current chunk size) - temp3Low (requested size) (yields excess size),
Q = temp1High (part of new objects address),
MAR = [temp2High, temp2Low + freePointersDopOffset],
unNewObjectHigh = 0, CANCELBR[2],
oftow = MD (first free oop), XDisp,
BRANCH[$, outOfOops, 0e],
Noop.
c1:
c2:
Noop.
c3:

(write the new size of the current free chunk (temp1High/Low still pointing at its size field))
MAR = [temp1High, temp1Low + 0],
MDR = temp3Low,
tempow - tempow + sizeofFieldOffset (point at base of the current free chunk),
MAR = [cotHigh, oltow + 1],
MDR = temp1High, CANCELBR[2], LOOPHOLE[wok],
Q = tempow (base of free chunk we split) + temp3Low (excessSize) (yields base of new object),
MAR = [cotHigh, oltow + 0],
unNewObjectLow = MDR + 0, Q = MD + MD (link to next free oop),
MAR = [temp2High, temp2Low + freePointersDopOffset],
MDR = Q (update free oop list head), CANCELBR[2], LOOPHOLE[wok],
unNewObject = oltow.
c1:
c2:
c3:

(should we move the current free chunk to a small free chunk list?)
tempow = largestFreeChunkSizelessOne, temp3Low = temp3Low (excess size) - temp2Low, CarryBr,
oltow = unCurrentFreeChunkDop, BRANCH[$, itsFineWhereItIs],
L1 = movingFromBigToSmall.
c1:
c2:
Noop.
c3:
CALL[addToFreeChunkList] (this call returns directly to splice),
splice:
oftow = upPredecessor, XDisp, L1 = splicingBigFreeList.
c1:
c2:
c3:

MemoryManagement.mc 2-Aug-84 9:24:20 PDT
Adjust the memory and oop levels and signal Mesa if either is below its alert level and Mesa has not yet been signalled.

{ (((wordLevel < wordAlertLevel) or: [oopLevel < oopAlertLevel]) and: [alreadyAlerted = 0]) ifTrue: [signalAlert = 1. MesaIntReq] }

allocate:
  temp2High = uRunRecordHigh,
  temp2Low = uRunRecordLow,
  Noop,
  MAR = [temp2High, temp2Low + oopLevelLowOffset],
  CANCELBR [$, 2].
  temp3Low = MD,
  MAR = [temp2High, temp2Low + oopLevelLowOffset],
  MDR = temp3Low + temp3Low - 1, CANCELBR [$, 2].
  LOOPHOLE [wok].
  Noop.
  Q = MD.
  MAR = [temp2High, temp2Low + oopAlertLevelLowOffset],
  CANCELBR [$, 2].
  Q = MD.
  Q = temp3Low - Q. CarryBr.
  BRANCH [I. decreaseWordLevel].
  MAR = [temp2High, temp2Low + alreadyAlertedOffset],
  CANCELBR [$, 2].
  Q = MD.
  [I] = Q. ZeroBr.
MemoryMangement.mc 2-Aug-84 9:24:20 PDT

BRANCH [decreaseWordLevel2, $].
Noop. c2;

MAR + [temp2High, tempLow + signalAlertOffset], c1;
MDR + 1, CANCELBR [$, 2], LOOPHOLE [wok], c2;
MessIntRq, GOTO [decreaseWordLevel2]. c3;

decreaseWordLevel2:
Noop. c3;

decreaseWordLevel:
MAR + [temp2High, tempLow + wordLevelLowOffset], c1;
temp3Low = uRequestedSize, CANCELBR [$, 2]. c2;
Q = MD, c3;
MAR + [temp2High, tempLow + wordLevelLowOffset], c1;
MDR + temp3Low = Q - temp3Low, CANCELBR [$, 2], c2;
LOOPHOLE [wok], CarryBr, c3;
BRANCH [wordLevelBorrow, lowMemoryTestHighGetData]. c4;

wordLevelBorrow:
MAR + [temp2High, tempLow + wordLevelHighOffset], c1;
CANCELBR [$, 2], c2;
Q = MD, c3;
MAR + [temp2High, tempLow + wordLevelHighOffset], c1;
MDR + Q - 1, CANCELBR [$, 2], LOOPHOLE [wok], c2;
temp3Low = Q - 1, GOTO [lowMemoryTestHighGetData]. c3;

lowMemoryTestHighGetData:
MAR + [temp2High, tempLow + wordAlertLevelHighOffset], c1;
CANCELBR [$, 2], c2;
temp3Low = MD, c3;

lowMemoryTestHighGetData2:
MAR + [temp2High, tempLow + wordAlertLevelLowOffset], c1;
CANCELBR [$, 2], c2;
Q = MD, c3;
[ ] = Q, zeroBr, BRANCH [reallyAllocate1, $], c1;
BRANCH [$, lowMemoryTestLow]. c2;
MAR + [temp2High, tempLow + alreadyAlertedOffset], c1;
CANCELBR [$, 2], c2;
Q = MD, c3;
[ ] = Q, zeroBr, c1;
BRANCH [lowMemoryTestLow2, $]. c2;
Noop. c3;

lowMemoryTestLow2:
Noop. c3;

lowMemoryTestLow:
MAR + [temp2High, tempLow + wordAlertLevelLowOffset], c1;
CANCELBR [$, 2], c2;
Q = MD, c3;
Noop. c1;
Q = temp3Low - Q, CarryBr, c2;
BRANCH [$, reallyAllocate3]. c3;
MAR + [temp2High, tempLow + alreadyAlertedOffset], c1;
CANCELBR [$, 2], c2;
Q = MD, c3;
[ ] = Q, zeroBr, c1;
BRANCH [reallyAllocate2, $], c2;
Noop. c3;
MAR + [temp2High, tempLow + signalAlertOffset], c1;
MDR + 1, CANCELBR [$, 2], LOOPHOLE [wok], c2;
MessIntRq, GOTO [reallyAllocate3]. c3;

reallyAllocate1:
CANCELBR [reallyAllocate1, 2]. c3;

reallyAllocate2:
Noop. c3;

reallyAllocate3:
Noop. c1;
temp3High = uNewObjectHigh, c2;
temp3Low = uNewObjectLow. c3;
temp3Low = uRequestedSize. c1;
temp3Low = temp3Low + temp3Low (low 16 bits of address of one word past the end of the object). c2;
temp3Low = temp3Low + deltaWordOffset. c3;

{initialize the object header now}
MAR + [tempHigh, tempLow + 0].
"the following MAR sets the delta ref count, the unused field, the clean
field, the containsLambda field, and the isVolatile field all to zero.
It also sets the oddBytes and hasPointers field appropriately"
MAR + utf16Type,
IFEqual [Waffle, 1, SkipTo[waffleOnly]], []
tempLow = tempLow + offsetFromDeltaWordToDiskWord,
IFEqual [Waffle, 0, SkipTo[endWaffleOnly]], []

waffleOnly:
tempLow = tempLow + offsetFromDeltaWordToDiskWord,
MAR + [tempHigh, tempLow + 0],
MAR + tempLow + offsetFromDiskWordToSizeField,
endWaffleOnly:
MAR + [tempHigh, tempLow + 0],
MAR + uRequestedSize,
tempLow = tempLow + offsetFromSizeFieldToClassField,
MAR + [tempHigh, tempLow + 0],
MAR + uClassToInstantiate,
(\(ok, object header is done, now zap the object body\)
tempLow = tempLow + offsetFromClassFieldToFirstChild,

initializeObjectBody:
Noop,
[] = tempLow - tempLow, ZeroBR,
BRANCH[\$, allZapped],
MAR + [tempHigh, tempLow + 0],
MAR + uDefault,
tempLow = tempLow + 1, GOTO[initializeObjectBody],

allZapped:
(the object header and object body are now completely initialized, now fix up the object table entry)
MAR + [ohigh, oentry = 1] \(\{\text{read second word of o entry}\}\)
CANCELLER[\$, 2],
Q = MD,
(the only thing worth preserving in the o entry is the segment number.
clobber everything else, turn on the untouched bit and write o entry)
MAR + [ohigh, oentry = 1] \(\{\text{read second word of o entry}\}\),
Q = MD + Q and 0,
CANCELLER[\$, 2], LOOPHOLE[\$, w1k],
tempLow = tempHigh\(\{\text{retrieve return link cause addToZeroCountTable will smash it}\\)
Lz = creatingAnInstance,
tempLow = tempLow (save return link), CALL[addToZeroCountTable],
Noop,
addToZeroCountTableReturn,
L1 = upClassAtInstantiation,
olow = uClassToInstantiates, XDisp, CALL[ref1],
olow = uNewObject,
Xbus = tempLow LBus, XDisp,
RET[createInstance-return],

nextFreeChunk:
CALL[otMap2],
tempLow = tempLow + chunkLinkOffset,
Noop,
Noop,
MAR + [tempHigh, tempLow + 0],
tempLow = tempLow + chunkLinkOffset, LDisp,
Q = MD, RET[nextFreeChunk-return],

nextFreeChunk:
CALL[otMap2],
tempLow = tempLow +chunkLinkOffset,
Noop,
Noop,
addToFreeChunkList:

(upon entry, temp2High/low must be the base of the object to add to the free list, otoLow must be the oop of the free chunk to put on the list and temp3Low must be the freelist index (0 -- 46) to put the oop on. L1 is the return linkage register, smashes temp2High/low, O)

temp2High = uRumRecordHigh,
temp2Low = uRumRecordLow,
temp2Low = temp2Low + freelistOffset,
MAR = [temp2High, temp2Low + temp3Low],
temp3Low = temp3Low + chunkLinkOffset, CANCELBR[$, 2],
0 = MD (current free list head),
MAR = [temp3High, temp3Low + 0],
MDR = 0,
MOp,
MAR = [temp2High, temp2Low + temp3Low],
MDR = otoLow, CANCELBR[$, 2], LOOPHOLE[wok], LIDisp,
temp3Low = temp3Low + chunkLinkOffset, RE[addToFreeChunkList-return], c3;
refl: (upon entry, ollow must indicate the oop to be refl’d. SmallIntegers acceptable. There must be a pending XDisp to test for SmallIntegers, and li is the return linkage register. smashes Q and tempilow)

```plaintext
[@ 6 - ollow, CarryBr, BRANCH[$, isSmallRef], 0a],
LIDisp, BRANCH[doRef], $],
RET[reflReturn], c2;
```

c1;
c2;
c3;
doRef: CANCELBR, 0f],
c1;
MAR = [otHigh, otlow + 1], c1;
tempilow = 4 (for adding "one" to ref count), CANCELBR[$, 2], c2;
Q = MD, XDisp [first part of test for stuck ref count], c3;
tempilow + tempilow LRot8, BRANCH[notStuckRef, maybeStuckRef], 2], c1;
notStuckRef: (sign is positive, thus not stuck and can not become stuck)
Q = Q + tempilow (up ref count), c2;
GOTO[updateDtrRef], c3;
maybeStuckRef: (sign is negative, can get stuck, may already be stuck)
Q = Q + tempilow (up ref count), CarryBr (carry implies already stuck), c2;
[ ] = Q + tempilow, CarryBr (carry implies just got stuck), BRANCH[updateDtrRef, stuckRef], c3;
updateDtrRef: MAR = [otHigh, otlow + 1], BRANCH[$, justGotStuckRef], c1;
MD = Q, LOOPHOLE [mdro], LOOPHOLE[wok], CANCELBR[returnFromRef], 3], LIDisp, c2;
justGotStuckRef: (loop: need to call Loom here for newly stuck refl)
MD = Q [write updated ref count], LOOPHOLE[mdro], LOOPHOLE[wok], CANCELBR[returnFromRef], 3], LIDisp, c2;

```plaintext
stuckRef: CANCELBR[$, 3],
LIDisp, GOTO[returnFromRef], c1;
c2;
```

c3;
isSmallRef: LIDisp, CANCELBR[returnFromRef], 3],
c2;
returnFromRef: RET[reflReturn], c3;

```plaintext
refd: (upon entry, ollow must be the oop to be refd’d. SmallIntegers acceptable. There must be a pending XDisp to test for SmallIntegers, and li is the return linkage register. smashes Q and tempilowHigh/low and tempilowHigh/low and L2)

[@ 6 - ollow, CarryBr, BRANCH[$, isSmallRef], 0a],
LIDisp, BRANCH[doRef], $],
RET[refdReturn], c1;
c2;
c3;
doRef: CANCELBR[$, 0f],
c3;
MAR = [otHigh, otlow + 1],
c1;
tempilow = 0fc (for "subtracting one" from the reference count), CANCELBR[$, 2], c2;
Q = MD, XDisp [first part of stuck ref count test], c3;
tempilow + tempilow LRot8, BRANCH[positiveRefCount, negativeRefCount], 2], c1;
positiveRefCount: (not stuck but could go to zero)
Q = Q + tempilow (subtract 1), CarryBr (carry implies already zero, an error), L2 = doingRefd, c2;
[ ] = Q + tempilow (subtract again), CarryBr (no carry implies just went to zero), BRANCH[triedToRefdZeroCountObject, updateDtrRef], c3;
negativeRefCount: (could be stuck but cannot go to zero)
tempilow = 4, c2;
tempilow = tempilow LRot8, c3;
[ ] = Q + tempilow, CarryBr (carry implies stuck ref count), c1;
Q = Q + tempilow (subtract one from ref count), BRANCH[notStuckRef, stuckRef], c2;
notStuckRef: Xlus + 1, XDisp (makes the branch at updateDtrRef happy!), c3;
updateDtrRef: MAR = [otHigh, otlow + 1][write updated refcount], BRANCH[addToZeroCountTable, notZeroRefd], c1;
```
GOTO[isSmallRefd],
notZeroRefd:
    MDR + Q, LOOPHOLE[wok], L2Disp, CANCELBR[returnFromRefd, 3],
stuckRefd:
    Noop,
    GOTO[isSmallRefd],
isSmallRefd:
    L2Disp, CANCELBR[returnFromRefd, 3],
returnFromRefd:
    NET[refdReturn],
    (the following mess is the result of addressing constraints and branching limitations)
    triedToRefdZeroCountObject:
        BRANCH[triedToRefdZeroCountObjectA, triedToRefdZeroCountObjectB],
    triedToRefdZeroCountObjectA:
        GOTO[bailoutA],
    triedToRefdZeroCountObjectB:
        GOTO[bailoutA],

addToZeroCountTable:
    {Loom: Loom may want to get involved here--but I don't think so}
    {upon entry, otlow is the op to put into the zct, and Q is the second word of the OT entry for otlow. L2 is the return
     Transaction register. Stamp high/low and temp3 high/low and Q, we turn on the inZct bit in the OT & write the new OT
     entry. see if op is already in the zct(by looking at former OT entry). if not we need to put the op into the ZCT}
    tempLow = 2, CANCELBR[$, 2],
    tempLow = tempLow LRot18, [the inzct bit]
    MAR = [otHigh, otlow + 1],
    MDR = 0 or tempLow (turn on zct bit), LOOPHOLE[wok], CANCELBR[$, 2],
    tempLow = MD, {former OT entry so we can test previous Inzct bit,}
    Q = uHmemRecordHigh {retrieve the Run Record address}, L2Disp, BRANCH[returnFromAddToZeroCountTable, needToPutInZct], c2;

needToPutInZct:
    {this could be sped up a bunch by keeping the address of the zct table in u registers...}
    tempHigh = 0 LRot0, CANCELBR[$, 0],
    tempLow = uHmemRecordLow,
    Noop,
    Noop,
    MAR = [tempHigh, tempLow + zctIndexOffset]{read current index},
    CANCELBR[$, 2],
    Q + MD, LOOPHOLE[mdxk],
    MAR = [tempHigh, tempLow + zctLowOffset]{get zct address},
    CANCELBR[$, 2],
    temp3Low = MD,
    MAR = [tempHigh, tempLow + zctHighOffset],
    temp3Low = temp3Low + Q, CANCELBR[$, 2],
    temp3High = MD,
    (note: the Molasses zeroCountTable is one-relative, not zero-relative. So, while Molasses bumps the index before putting
     something in the zct, we put it in, then bump.)
    MAR = [temp3High, temp3Low + 0],
    MDR = otLow,
    Noop,
    MAR = [tempHigh, tempLow + zctIndexOffset]{write updated index},
    MDR = 0 + Q + 1, CANCELBR[$, 2], LOOPHOLE[wok],
    Noop,
    MAR = [tempHigh, tempLow + stabilizationLimitOffset],
    CANCELBR[$, 2],
    temp3Low = MD,
    Noop,
    [1] = temp3Low - 0, NegBr,
    BRANCH[zctIndexOA, stabilizationNeeded],
    stabilizationNeeded:
    MAR = [tempHigh, tempLow + stabilizationFlagOffset],
    MDR = 1, CANCELBR[$, 2], LOOPHOLE[wok],
    c2,
uTimeToStabilize + stackLow xor stackLow.

zctIndexOk:
  Noop,
  L2Disp,
  returnFromAddToZeroCountTable;
  RET[addToZeroCountTableReturn].

  c3:
  c1:
  c2:
  c3:
makeVolatile:
( upon entry, otlow is the oop to make volatile, uMakeVolatile:linkage is the return linkage register; if it is odd, each object referred to by the object will be ref'd; it is even, the object is marked volatile, but no ref'd occurs, smashes temp3Low, Q, L1, L2, leaves base of object in uMakeVolatile:High/Low, and in temp3High/Low, leaves uLastPointer set up)

(see if we're trying to make nil volatile -- this happens when the least context oop is nil, check should probably be moved to the place where volatileization is done after stabilization...)

[] = otlow xor nil:Pointer, ZeroBr,
BRANCH[$, nil:MakeVolatile:]. c2;

uMakeVolatile:Op = otlow,
L1 = makingVolatile, CALL[otMap] (get address of base of object),
Q = temp3High (save object base)
UMakeVolatile:High = Q,
UMakeVolatile:Low = temp3low,

temp3low = temp3low + deltaWordOffset,
Noop,
Noop,
MAR = [temp3High, temp3low + 0],
Noop,
Q = MD (delta word),
[] = Q and 4, ZeroBr (volatile bit),
Ybus = UMakeVolatile:linkage, XDisp, BRANCH[alreadyVolatile, doMakeVolatile:], c2;

alreadyVolatile:

temp3low = temp3low - deltaWordOffset, RET[makeVolatile:return], c3;

doMakeVolatile:
O = Q or 4 (volatile bit), CANCELBR[$, 0f],
MAR = [temp3High, temp3low + 0],
MDR = Q (delta word with volatile bit set),
Ybus = UMakeVolatile:linkage, XDisp, (should we ref'd the referents or not?), c3;

temp3low = temp3low - deltaWordOffset, BRANCH[returnFromMakeVolatile, doRefdFields, 0a], c1;

doRefdFields:

temp3low = temp3low - sizeFieldOffset,
Noop,
MAR = [temp3High, temp3low + 0],
temp3low = temp3low - sizeFieldOffset,
temp3low = MD (size field),
temp3low = temp3low + temp3Low,
temp3low = temp3low - 1 (low 16 bits of last pointer of context object),
uLastPointer = temp3low,

( now, sweep the object decrementing reference counts of all pointer fields)

temp3Low = temp3low + firstPointerFieldOffset,
O = temp3High,
temp2High = O LRot0,

makeVolatile:Loop:
MAR = [temp2High, temp3Low + 0], L1 = lnMakeVolatile,
Noop,
otlow = MD, XDisp, CALL[ref],
temp3Low = uLastPointer,
[] = temp3Low - temp3Low, ZeroBr,
temp2Low = temp3Low + 1, BRANCH[makeVolatile:Loop, ],

otlow = uMakeVolatile:OOp,
Noop,
Noop,
MAR = [otHigh, otLow + 1],
CANCELBR[$, 2],
Q = MD (second word of o entry of object we are volatileizing), L2 = lnMakeVolatile, c3;

CALL[addToZeroCountTable],
temp3High = uMakeVolatile:High,
addToZeroCountTable:Return;
temp3low = uMakeVolatile:Low,
Noop,

returnFromMakeVolatile:
Ybus = uMakeVolatile:linkage, XDisp,

returningFromMakeVolatile:
RET[makeVolatile:return], c3;
pfMakeVolatile:
GOTO[returnFromMakeVolatile].

cl1:
lastPointerOf:
  (upon entry, tempHigh/low must point at the base of the object of interest, Q must be the delta word of the object, L2 is the return linkage register. returns the low 16 bits of the ADDRESS of the last pointer in uLastPointer and in temp3low, smashes Q)

[] + Q and 1 (pointer bit), 0โอBr, c1;
Q = classCompiledMethodDop, BRANCH[doesHavePointers, doesNotHavePointers], c2;

doesHavePointers:
  (is pure pointer object -- last pointer is size of object)
tempLow = tempLow + sizeFieldOffset, c3;
MAR = [tempHigh, tempLow + 0][start read of length field], c1;
tempLow = tempLow + sizeFieldOffset (again point at base of object), c2;
temp3Low = HD, GOTO[returnFromLastPointerOF], c3;

doesNotHavePointers:
  (no pointers, might be compiledMethod -- need to check class)
tempLow = tempLow + classFieldOffset, c3;
MAR = [tempHigh, tempLow + 0], c1;
tempLow = tempLow + classFieldOffset (again point at base of object), c2;
temp3Low = HD (the class of the object), c3;
[] + temp3Low xor Q (compiledMethodClass oop), 0โอBr, c1;
BRANCH[notCompiledMethod, isCompiledMethod], c2;

notCompiledMethod:
temp3Low = objectHeaderSize, GOTO[returnFromLastPointerOF], c3;

isCompiledMethod:
  (need to get number of literals from the method header)
tempLow = tempLow + objectHeaderSize (point at method header), c3;
MAR = [tempHigh, tempLow + 0], c1;
tempLow = tempLow + objectHeaderSize (again point at base of object) c2;
temp3Low = HD (the method header), c3;
temp3Low = (RShift1 temp3low and ?){get literal count of compiledMethod}, SE = 0, c1;
temp3Low = temp3Low + literalStart, c2;
temp3Low = temp3Low + objectHeaderSize, c2;

returnFromLastPointerOF:
temp3Low = temp3Low - 1, c1;
temp3Low = temp3Low + tempLow, L2Dsp, c2;
uLastPointer + temp3Low, RET[LastPointerOF-return], c3;
{
    Test the memory and oop levels and reset already alerted if both are above their alert levels.

    \(((\text{wordLevel} > \text{wordAlertLevel}) \text{ and} \ [\text{oopLevel} > \text{oopAlertLevel}]) \text{ ifTrue:} \ [\text{alreadyAlerted} += 0])\}

    \text{stabilize:}
    \\{\\begin{align*}
    &\text{[linkage register is L0 -- runs only between bytecodes]} \\
    &\text{tempHigh} + \text{umRecordHigh}, \quad \text{c1}; \\
    &\text{tempLow} + \text{umRecordLow}, \quad \text{c2}; \\
    &\text{uTimeToStabilize} += 0, \quad \text{c3}; \\
    \end{align*}\\}

    \text{testOopLevel:}
    \begin{align*}
    &\text{MAR} = \text{[tempHigh, tempLow + oopAlertLevelOffset]}, \quad \text{c1}; \\
    &\text{CANCELBR} [\$, 2], \quad \text{c2}; \\
    &\text{tempLow} + \text{MD}, \quad \text{c3}; \\
    &\text{MAR} = \text{[tempHigh, tempLow + oopAlertLevelOffset]}, \quad \text{c1}; \\
    &\text{CANCELBR} [\$, 2], \quad \text{c2}; \\
    &\text{Q} = \text{MD}, \quad \text{c3}; \\
    &\text{Q + tempLow} = \text{Q, CarryBr,} \quad \text{c1}; \\
    &\text{[+} = \text{Q, zeroBr, BRANCH [stabilize3, $]}, \quad \text{c2}; \\
    &\text{BRANCH [resetAlreadyAlerted, testWordLevelLow]}, \quad \text{c3}; \\
    \end{align*}

    \text{testWordLevelHigh:}
    \begin{align*}
    &\text{MAR} = \text{[tempHigh, tempLow + wordLevelHighOffset]}, \quad \text{c1}; \\
    &\text{CANCELBR} [\$, 2], \quad \text{c2}; \\
    &\text{tempLow} + \text{MD}, \quad \text{c3}; \\
    &\text{MAR} = \text{[tempHigh, tempLow + wordAlertLevelHighOffset]}, \quad \text{c1}; \\
    &\text{CANCELBR} [\$, 2], \quad \text{c2}; \\
    &\text{Q} = \text{MD}, \quad \text{c3}; \\
    &\text{Q + tempLow} = \text{Q, CarryBr,} \quad \text{c1}; \\
    &\text{[+} = \text{Q, zeroBr, BRANCH [stabilize3, $]}, \quad \text{c2}; \\
    &\text{BRANCH [resetAlreadyAlerted, testWordLevelLow]}, \quad \text{c3}; \\
    \end{align*}

    \text{testWordLevelLow:}
    \begin{align*}
    &\text{MAR} = \text{[tempHigh, tempLow + wordLevelLowOffset]}, \quad \text{c1}; \\
    &\text{CANCELBR} [\$, 2], \quad \text{c2}; \\
    &\text{tempLow} + \text{MD}, \quad \text{c3}; \\
    &\text{MAR} = \text{[tempHigh, tempLow + wordAlertLevelLowOffset]}, \quad \text{c1}; \\
    &\text{CANCELBR} [\$, 2], \quad \text{c2}; \\
    &\text{Q} = \text{MD}, \quad \text{c3}; \\
    &\text{Q + tempLow} = \text{Q, CarryBr,} \quad \text{c1}; \\
    &\text{BRANCH [stabilize3, $]}, \quad \text{c2}; \\
    &\text{Noop,} \quad \text{c3}; \\
    \end{align*}

    \text{resetAlreadyAlerted:}
    \begin{align*}
    &\text{MAR} = \text{[tempHigh, tempLow + alreadyAlertedOffset]}, \quad \text{c1}; \\
    &\text{MDR} = \text{0, CANCELBR} [\$, 2], \quad \text{c2}; \\
    &\text{LOOPHOLE [wok]}, \quad \text{c3}; \\
    &\text{GOTO [reallyStabilize]}, \quad \text{c3}; \\
    \end{align*}

    \text{stabilize1:}
    \begin{align*}
    &\text{CANCELBR [reallyStabilize, 1]}, \quad \text{c3}; \\
    \end{align*}

    \text{stabilize2:}
    \begin{align*}
    &\text{GOTO [reallyStabilize]}, \quad \text{c3}; \\
    \end{align*}

    \text{stabilize3:}
    \begin{align*}
    &\text{GOTO [reallyStabilize]}, \quad \text{c3}; \\
    \end{align*}

    \text{reallyStabilize:}
    \begin{align*}
    &\text{[get address of the zct]} \\
    &\text{MAR} = \text{[tempHigh, tempLow + zctLowOffset]}, \quad \text{c1}; \\
    &\text{CANCELBR} [\$, 2], \quad \text{c2}; \\
    &\text{tempLow} + \text{MD}, \quad \text{c3}; \\
    &\text{MAR} = \text{[tempHigh, tempLow + zctHighOffset]}, \quad \text{c1}; \\
    &\text{uZctBaseLow + tempLow, CANCELBR} [\$, 2], \quad \text{c2}; \\
    &\text{temp2High + MD}, \quad \text{c3}; \\
    &\text{get. then smash the zct index from the um record} \\
    &\text{MAR} = \text{[tempHigh, tempLow + zctIndexOffset]}, \quad \text{c1}; \\
    &\text{MDR} = \text{0, CANCELBR} [\$, 2], \quad \text{c2}; \\
    &\text{LOOPHOLE [wok]}, \quad \text{c3}; \\
    &\text{tempLow} + \text{MD}, \quad \text{c3}; \\
    \end{align*}

    \text{[reset the stabilization flag]}
    \begin{align*}
    &\text{MAR} = \text{[tempHigh, tempLow + stabilizationFlagOffset]}, \quad \text{c1}; \\
    &\text{MDR} = \text{0, CANCELBR} [\$, 2], \quad \text{c2}; \\
    &\text{LOOPHOLE [wok]}, \quad \text{c3}; \\
    &\text{tempLow} + \text{tempLow} \text{ (yields low 16 bits of one word past the last valid oop in the zct), c3}; \\
    &\text{uZctLimit = tempLow,} \quad \text{c3}; \\
    &\text{uQueueHead = -oLow xor otlow,} \quad \text{c2}; \\
    &\text{uCurrentObject = -otlow xor otlow,} \quad \text{c3}; \\
    \end{align*}

    \text{(sweep the zct. for each oop marked (in its ot entry) as volatile, reset the isVolatil bit, and increase the reference counts of all of its referents. recall that the zct index is one greater than the number of valid entries in the zct)}

    \text{stabilizationLoop:}
temp2low xor u2ctlimit, ZeroBR {are we there yet?},
      c1;
temp3low = Off. BRANCH[2], countAreNowCorrect, c2;
temp3low = temp3low LRot8, c3;
MAR + [temp2High, temp2Low + 0], c1;
temp2low = temp2Low + 1, L1 = stabilizing, c2;
oilow = MD, CALL[otMapZ] {so we can get its delta word}, c3;
temp1low = temp1low + deltaWordOffset, at[stabilizing, 10, otMapZ-return],
temp3low = temp3low or 0fb {yields ff0b, for turning off the isVolatile bit}, c2;
Noop, c3;
MAR + [temp1High, temp1Low + 0] {read delta word} c1;
Noop, c2;
0 = MD, XDisp {to test isVolatile bit}, c3;
MAR + [temp1High, temp1Low + 0], BRANCH[oopIsNotVolatile, $, 0b], c1;
0 = MD + Q and temp2Low {not volatile anymore}, L2 = stabilizingContext, c2;
temp1low = temp1low - deltaWordOffset, CALL[lastPointerOf], c3;
{need to move the temp1 regs into temp3 to keep refi from smashing them...}
0 = temp1high, c1, at[stabilizingContext, 10,
lastPointerOf-return],
temp3High = Q LRot8, c2;
temp3low = temp1low + classFieldOffset, c3;
{swEEP over the volatile object, updating the reference counters of its references}
upReferents:
      MAR + [temp3High, temp3Low + 0], L1 = correcting, c1;
      Noop, c2;
oilow = MD, XDisp, CALL[refi], c3;
[] = temp3low xor u1astPointer, ZeroBR, c1, at[correcting, 10, refi-return],
temp3low = temp3Low + 1, BRANCH[1, thisOneIsStable], c2;
GOTO[upReferents], c3;
oopIsNotVolatile:
      Noop, c2;
thisOneIsStable:
      GOTO[stabilizationLoop], c3;

countAreNowCorrect:
{at this point, all contexts have been stabilized, and all reference counts are correct. swell over the zct again: any
object in the zct whose reference count is zero is garbage!}
(temp2High is still valid despite the CALLs. restore temp2low)
temp2low = u2ctBaseLow, c3;
sweepAndDeallocateLoop:
      [] = temp2low xor u2ctLimit, ZeroBR {are we there yet?}, c1;
      BRANCH[2], returnFromStabilize, c2;
tempilow = Off, c3;
MAR + [temp2High, temp2Low + 0] {get oop from zct} c1,
temp1low = temp1low LRot8, c2;
oilow = MD, c3;
MAR + [otHigh, oilow + 1] {get its ot entry} c1,
tempilow = tempilow or Off {yields ff0f for turning off fnZct bit}, CANCELBR[$, 2], c2;
tempilow = MD, c3;
MAR + [otHigh, oilow + 1] {rewrite its ot entry} c1;
MDR = temp3Low and temp1low {turns off the fnZct bit}, CANCELBR[$, 2], LOOPHOLE[wok], c2;
tempilow = temp1low LRot8, c3;
tempilow = 0fc {mask is reference count bits}, c1;
[] = tempilow and temp1low, ZeroBR, c2;
temp2low = temp2Low + 1, BRANCH[sweepAndDeallocateLoop, needToDeallocate], c3;

needToDeallocate:
{save our current state}
0 = temp2High, u2ctSweepHigh = 0, L3 = fromStabilize, c1;
u2ctSweepLow = temp1low, CALL[deallocate], c2;
{recover our state}
temp2High = u2ctSweepHigh, c1, at[FromStabilize, 10, deallocate-return],
tempilow = u2ctSweepLow, c2;
GOTO[sweepAndDeallocateLoop], c3;

returnFromStabilize:
      Noop, c3;
      Noop, c1;
      LDisp, c2;
      RET[stabilizer-return], c3;
deallocate:
  (allocate is the oop to deallocate -- it has already been determined that
  its reference count is 0, L3 is the return linkage register)
  L2 = startingDeallocate,              c1;
  Noop,                                 c2;
  [j] = ollow LRot0, XDisp, CALL[getClass],  c3;
  tempLow = tempLow + deltaWordOffset,  c1, at[startingDeallocate, 10, getClass-return];
  Q = classCompiledMethodGop,           c2;
  [j] = tempLow xor Q, ZeroBr,           c3;
  (get delta word to see if object has pointers)
  MAR = [tempHigh, tempLow + 0], BRANCH[$, deallocatingACompiledMethod], c1;
  Noop,                                 c2;
  0 = MD, XLDisp,                        c3;
  BRANCH[deallocateWithNoPointers, deallocateWithPointers, 2].        c1;
deallocateWithNoPointers:
  tempLow = tempLow - deltaWordOffset,  L1 = freeNonPointerObject,      c2;
  uClass = tempLow, CALL[adjustLevelsAndReturnToPool],                c3;
  tempLow = uClass, GOTO[nowDoObjectsClass],                           c1, at[freeNonPointerObject, 10, addToFreeChunkList-return];
deallocatingACompiledMethod:
  GOTO[deallocateWithPointersA].                       c2;
deallocateWithPointers:
  Noop,                                              c2;
deallocatingWithPointersA:
  tempLow = tempLow + offsetFromDeltaWordToClassField,         c3;
  (enqueue this object for deallocaction)
  MAR = [tempHigh, tempLow + 0],                       c1;
  NDR = uQueueHead,                                    c2;
  uQueueHead = ollow,                                   c3;
  Noop,                                                 c1;
nowDoObjectsClass:
  Noop,                                               c2;
  ollow = tempLow LRot0, XDisp, GOTO[specialRefd].       c3;
more:
  (if there is a current object, continue with it. if not, if there is a queued object, start it. otherwise we are done)
  ollow = uCurrentObject, XDisp,                      c1, at[nowDoneWithObject, 10, addToFreeChunkList-return];
  BRANCH[continueWithCurrentObject, checkQueueHead, 0e]. c2;
checkQueueHead:
  ollow = uQueueHead, XDisp,                          c3;
  BRANCH[startWithQueueHead, noMore, 0e].             c1;
noMore:
  (all recursive freeing is now complete)
  L3Disp,                                             c2;
  RET[deallocate-return].                              c3;
startWithQueueHead:
  uCurrentObject = ollow, L1 = sweepingObject,        c2;
  CALL[otMap2].                                        c3;
  tempLow = tempLow + deltaWordOffset,               c1, at[sweepingObject, 10, otMap2-return];
  Q = tempHigh,                                       c2;
  uSofAmHigh = Q,                                     c3;
  MAR = [tempHigh, tempLow + 0],                     c1;
  tempLow = tempLow - deltaWordOffset,               c2;
  L2 = getobjectEndForFreeing,                        c3;
  0 = MD, XLDisp,                                      c3;
  BRANCH[doingACompiledMethod, notDoingACompiledMethod, 0e]. c1;  
  (both isCompiledMethod and doesHavePointers live in the lastPointerOf routine)
  doingACompiledMethod:
  CALL[isCompiledMethod].                            c2;
notDoingACompiledMethod:
    CALL[doesHavePointers],
    uCurrentObjectBaseLow = tempilow,  
c1: @getObjectEndForFreesing, 10,
    lastPointer0f=return,
    temp1ow = temp1ow + class_fieldoffset,
    Noop,
    MAR = [temp3High, temp1ow + 0],
    Noop,  
c2: temp3ow = temp3ow + 0 (link to next object on queue),
    uQueueHead = temp3ow,  
c3: Noop,
    GOTO[areWeThereYet].
areWeThereYet:
    MAR = [temp3High, temp3ow + 0],
    Noop,  
c1: otlow = oDHigh, XDisp, GOTO[specialRefd].
    temp3ow = uDSoFarHigh,
    temp3ow = uDSoFarHigh,  
c2: uDSoFar = temp3ow, GOTO[doAnotherField].
    Noop.  
c3:
doAnotherField:
    MAR = [temp3High, temp3ow + 0],
    Noop,  
c1: otlow = oDHigh, XDisp, GOTO[specialRefd].
doneWithObject:
    Noop,  
c3:
    otlow = uCurrentObjectBaseLow, l1 = nowDoneWithObject,
    uCurrentObject = otlow xor otlow, CALL[adjustLevelsAndReturnToPool].
specialRefd:
    (upon entry, otlow must be the oop to be specialRefd'd and there must be a pending XDisp to test for smallIntegerness,
    (smashes Q and temp3High/Low and temp3High/Low and l2)
    [ ] = 6. otlow, CarryBr, BRANCH[$, specialSmall, 0w],
    BRANCH[doSpecial[], $],  
c1: GOTO[more].
doSpecial:
    CANCELBR[$, 0f].
    MAR = [otHigh, otLow + 1],
    temp3ow = 0fc (for "subtracting one" from the reference count), CANCELBR[$, 2],  
c2: Q = oDHigh, XDisp (first part of stuck ref count test),
    temp3ow = temp3ow LRot6, BRANCH[specialPositiveRefCount, specialNegativeRefCount, 2],  
c3: specialPositiveRefCount: (not stuck but could go to zero)
    Q = Q + temp3ow (subtract 1). CarryBr (carry implies already zero, an error),  
c4: temp3ow = temp3ow (subtract again). CarryBr (no carry implies just went to zero),
    BRANCH[tryDoSpecialRefdZeroCountObject, specialUpdate0tRefd],  
c5: specialNegativeRefCount: (could be stuck but cannot go to zero)
    temp3ow = 4,  
c6: temp3ow = temp3ow LRot8,
    [ ] = 0 + temp3ow, CarryBr (carry implies stuck ref count),  
c7: Q = Q + temp3ow (subtract one from ref count). BRANCH[specialNotStuckRefd, specialStuckRefd],
specialNotStuckRefd:
    XDisp = 1. XDisp (makes the branch at specialUpdate0tRefd happy!).
specialStuckRefd:
    MAR = [otHigh, otLow + 1] (write updated refCount). BRANCH[specialNeedsDeallocation, $],  
c8: NoD = 0. LOOPHOLE[ok], CANCELBR[$, 2],
specialUpdate0tRefd:
    MAR = [otHigh, otLow + 1],
GOTO[more].
specialStackRef:
  GOTO[more].
specialNeedsDeallocation:
  MDR = 0, LOOPHOLE[wok], CANCELBR[$, 2],
tempLow = 1.
  (this objects ref count went to zero. deallocate it, but only if it’s inZct bit is off -- if it’s on, the zct processing
  will take care of it in a little while)
  tempLow = tempLow Low68,
  [ ] = 0 and tempLow, ZeroBr.
  BRANCH[$, deallocate],
  Noop,
  Noop,
  GOTO[more].
specialSmall:
  CANCELBR[$, 1],
  GOTO[more].

triedToSpecialRefZeroCountObject:
  BRANCH[triedToSpecialRefZeroCountObjectA, triedToSpecialRefZeroCountObjectB].
triedToSpecialRefZeroCountObjectA:
  GOTO[ba1out3].
triedToSpecialRefZeroCountObjectB:
  GOTO[ba1out3].

adjustLevelsAndReturnToPool:
  tempLow = tempLow - sizeFieldOffset,
  temp2High = uNumRecordHigh,
  temp2Low = uNumRecordLow,
  IncrementOpLevel:
  MAR = [temp2High, temp2Low + oopLevelLowOffset],
  CANCELBR [$, 2],
  temp3Low = MD,
  MAR = [temp2High, temp2Low + oopLevelLowOffset],
  MDR = temp3Low - temp3Low = 1,
  CANCELBR [$, 2], LOOPHOLE [wok], CarryBr,
  BRANCH [IncreaseWordLevelLow, impossibleOpLevel],
  impossibleOpLevel:
  GOTO [ba1out2].

IncreaseWordLevelLow:
  MAR = [temp2High, temp2Low + 0],
  temp3Low = temp2Low - sizeFieldOffset,
  temp3Low = MD,
  MAR = [temp2High, temp2Low + wordLevelLowOffset],
  CANCELBR [$, 2],
  Q = MD,
  MAR = [temp2High, temp2Low + wordLevelLowOffset],
  MDR = Q + temp3Low, CANCELBR [$, 2], LOOPHOLE [wok], CarryBr,
  BRANCH [returnToPool, wordLevelCarry],

wordLevelCarry:
  MAR = [temp2High, temp2Low + wordLevelHighOffset],
  CANCELBR [$, 2],
  temp3Low = MD,
  MAR = [temp2High, temp2Low + wordLevelHighOffset],
  MDR = temp3Low + temp3Low + 1, LOOPHOLE [wok],
  GOTO [returnToPool].

returnToPool:
{build a mask to set all ref count bits on and to set purpose bits to free (11)}
  temp3Low = 0F8,
  temp3Low = temp3Low LRead,
  temp3Low = temp3Low or 60,
  MAR = [otiHigh, otLow + 1],
  CANCELBR[$, 2],
  Q = MD,
  MAR = [otiHigh, otLow + 1],
  MDR = Q or temp3Low, CANCELBR[$, 2], LOOPHOLE[wok],
  GOTO[addToProperFreeChunkList].
addToProperFreeChunkList:

(Upon entry, otlow is the top of the object to add to the free list, tempHigh/Low must be that object's base. Calls addToFreeChunkList thus smashing temp2High/Low and 0, smashes temp3Low)

```plaintext
tempLow = tempLow + sizeofFieldOffset,
Q = largestFreeChunkSize,
Noop,
MAR = [tempHigh, tempLow + 0],
tempLow = tempLow - sizeofFieldOffset,
temp3Low = HD (object's size),
[ ] = temp3Low - 0, CarryBr,
BRANCH[selectRegularList, selectBigFreeList],
```

selectRegularList:
GOTO[addToFreeChunkList],
slectBigFreeList:
temp3Low = 0, GOTO[addToFreeChunkList].
```plaintext
c1; c2; c3; c4; c5; c6; c7; c8;
```
otmap.mc  1-Aug-84 18:27:16 PDT

{

1-Aug-84 18:27:13
}

{
inputs - otLow is the oop whose mapping is desired -- better not supply a SmallInteger --, li is the return linkage register
output - high portion of address is returned in tempHigh, low portion in tempLow
}

{ (todo --- put in a check and trap for trying to otMap lambda!)
  otMap:
  IfEqual[otMapDebug, debug, SkipTo[endOtMapDebug]]:
    {some debugging code to ensure that the oop is even}
    Ybus + otLow, YDisp,
    BRANCH[goodOop-otMap, badOop-otMap, 0a].
    badOop-otMap:
    GOTO[bs1out1],
    goodOop-otMap:
    Noop,
  endOtMapDebug:
    {the real code for otMap starts here}
    {we can get by with the + 1 because a page cross is impossible --
    the OT entry must be at an even address, and only incrementing an
    odd address can cause a page fault}
    MAR = [otHigh, otLow + 1],
    CANCELBS[$, 2],
    tempHigh + HD, XDisp (check if this object is a leaf).
    tempHigh + HD, RET[otMap-return].
    MAR = [otHigh, otLow - 0], BRANCH[$, isLeaf-otMap, 1].
    LIDisp (this is not a leaf).
    tempLow + MD, RET[otMap-return].
    {Loop: needs to punt to Mesa to handle this leaf}
    isLeaf-otMap:
    GOTO[bs1out3].
    c1:
    c2:
    c3:
    c1:
    c2:
    c3:
    c2:
}

{
inputs - otLow is the oop whose mapping is desired -- better not supply a SmallInteger --,
li is the return linkage register
output - high portion of address is returned in tempHigh, low portion in tempLow
}

{ (todo --- put in a check and trap for trying to otMap lambda!)
  otMap2:
  IfEqual[otMapDebug, debug, SkipTo[endOtMapDebug2]]:
    {some debugging code to ensure that the oop is even}
    Ybus + otLow, YDisp,
    BRANCH[goodOop-otMap2, badOop-otMap2, 0a].
    badOop-otMap2:
    GOTO[bs1out1],
    goodOop-otMap2:
    Noop,
  otmap.mc  1-Aug-84 18:27:16 PDT
endOtMapDebug2!

{the real code for otMap starts here}

(we can get by with the + 1 because a page cross is impossible --
the OT entry must be at an even address, and only incrementing an
odd address can cause a page fault)
MAR = [ctHigh, otlow + 1].
Noop, CANCEL[$, 2].
tempHigh += MD, XLDisp {check if this object is a leaf}.
MAR = [ctHigh, otlow + 0]. BRANCH[$, fslLeaf-otMap2-1].
L1Disp {this is not a leaf}.
tempLow += MD, RET[otMap2-return].

{Loom: needs to punt to Mesa to handle this leaf}

fslLeaf-otMap2:
GOTO[settOut3].

fslLeaf-otMap2:

(Pop and Store Temporary Location bytecodes)

popAndStoreTemporaryLocation0:
    homeLow = homeLow + temp0, GOTO[popAndStoreTemporary].
    c1, bytecode[88];

popAndStoreTemporaryLocation1:
    homeLow = homeLow + temp1, GOTO[popAndStoreTemporary].
    c1, bytecode[89];

popAndStoreTemporaryLocation2:
    homeLow = homeLow + temp2, GOTO[popAndStoreTemporary].
    c1, bytecode[8a];

popAndStoreTemporaryLocation3:
    homeLow = homeLow + temp3, GOTO[popAndStoreTemporary].
    c1, bytecode[8b];

popAndStoreTemporaryLocation4:
    homeLow = homeLow + temp4, GOTO[popAndStoreTemporary].
    c1, bytecode[8c];

popAndStoreTemporaryLocation5:
    homeLow = homeLow + temp5, GOTO[popAndStoreTemporary].
    c1, bytecode[8d];

popAndStoreTemporaryLocation6:
    homeLow = homeLow + temp6, GOTO[popAndStoreTemporary].
    c1, bytecode[8e];

popAndStoreTemporaryLocation7:
    homeLow = homeLow + temp7, GOTO[popAndStoreTemporary].
    c1, bytecode[8f];

popAndStoreTemporary:
    uSmashTos = ~temp2Low xor temp2Low (-1), L1 = storeTemporary, GOTO[storeTemporary], c2;

storeTemporary:
    (upon entry, homeHigh/low must indicate the field to be written. L1 must contain the value "storeTemporary". and uSmashTos must be -1 to smash top of stack. 0 otherwise)
    [] = uSmashTos, Zero$BR, CALL[returnTopOfStack].
    c3;
    MAR = [homeHigh, homeLow + 0], (start write of context field)
    c1, at[storeTemporary, 10];
    returnTopOfStack: return;
    c3;
    NDR = atLw, (write the top of stack),
    c2;
    homeLow = homeLow, (restore the home base register)
    homeLow = homeLow + temp6, GOTO[popAndStoreTemporary].
    homeLow = homeLow + temp6, GOTO[popAndStoreTemporary].
    c3;
    c3;

(Extended Store (Receiver Variable, Temporary Location, Illegal, Literal Variable))

extendedStore:
    uSmashTos = 0, GOTO[commonExtendedStore].
    c1, bytecode[81];

(Extended Pop and Store (Receiver Variable, Temporary Location, Illegal, Literal Variable))

extendedPopAndStore:
    uSmashTos = ~temp2Low xor temp2Low (-1), GOTO[commonExtendedStore].
    c1, bytecode[82];

commonExtendedStore:
    temp2Low = lb [get the extension byte],
    temp3Low = temp2Low and 3f, (get only the offset)
    c2;
    temp3Low + temp3Low + objectHeaderSize, (and bump to the object body)
    c1;
    temp2Low = RShift1 temp2Low, (shift for convenient dispatch)
    c2;
    [] + temp2Low LAnd0, XwDisp, (dispatch on the type of extended push) c3;
    DISP2[extendedStoreTarget], iPLow + iPLow + PC16, (account for the extension byte, and take off) c1;

extendedStoreReceiver:
    GOTO[storeReceiverVariable], backupIsByte (in case of refi/refd needs to call loom), c2, at[0, 4, extendedStoreTarget];

extendedStoreTemporary:
    temp2Low + temp3Low + tempFrameStart,
    homeLow = homeLow + temp3Low, L1 = storeTemporary,
    c2, at[1, 4, extendedStoreTarget];
    Woop,
    c3;
    c1;

Pop.mc 1-Aug-84 18:28:52 PDT
illegalExtendedStore:
GOTO[illegalExtendedStore], c2;

extendedStoreLiteralVariable:
    temp3Low = temp3Low + literalStart, backupIs1Byte {in case ref/refs overflow or lambda in association}, c2, at[3, 4, extendedStoreTarget],
    temp2Low = uCurrentMethodLow, c3;
    tempHigh = uCurrentMethodHigh, temp2Low + temp3Low + temp2Low, {and add in offset to the appropriate association} c1; Neop, c2;
    Neop, c3;
    MAR = [temp3High, temp2Low + 0], {read the association oop} c1;
    t1 = storingLiteralVariable, c2;
    otlow = MO, CALL[otMap] {otMap the association}, c3;
    O = tempHigh, temp2Low + temp1Low = associationValueIndex {add in offset to value}, c2;
    temp2High = 0, temp1Low = 0, c3;
    MAR = [temp2High, temp2Low + 0], c1;
    Neop, c2;
    otlow = HD {get the value of the association}, c3;
    [] = otlow, ZeroBr {test for lambda}, t1 = storingLitVar, c1; BRANCH[valuesIsNotLambda-storeLiteralVariable, valueIsLambda-storeLiteralVariable], c2;

valueIsLambda-storeLiteralVariable:
    {Loop: need to call Loom here} GOTO[bailout], c3;

valueIsNotLambda-storeLiteralVariable:
    [] = otlow LRot0, XDisp, CALL[refs], c3;
    MAR = [stackHigh, stacklow + 0], c1, at[storingLitVar, 10, refdReturn];
    Neop, c2;
    otlow = HD, XDisp, CALL[refi], c3;
    Neop, c2;
    Neop, c3;
    [] = uSmashPos, ZeroBr, CALL[returnTopOfStack], c1, at[storingLitVar, 10, refiReturn];
    Neop, c2;
    Neop, c3;
    MAR = [temp2High, temp2Low + 0], c1, at[storingLitVar, 10, returnTopOfStack-return];
    returnTopOfStack-return; HD = otlow, NextBytecode, c2;
    DISPLI[bytecodes], 1plow + 1plow + PC16, c3;

(Pop Stack Top)

popStackTop:
    MAR = [stackHigh, stacklow + 0], c1, bytecode[87];
    MDR = n1Pointer {smash top of stack}, c2;
    Neop, c3;
    stackLow = stackLow - 1, c1;
    NextBytecode, c2;
    DISPLI[bytecodes], 1plow + 1plow + PC16, c3;
primitiveIndexNotZero: 
  uPrimitiveNumber = temp2low (save in case of primitive failure). CANCELBE[0, 0f]. c1:
  Noop. 
  [] = temp2low LRot12, XDisp (dispatch on high 4 bits of primitive #). c3:
  [] = temp2low LRot0, XDisp (dispatch on low 4 bits). DISP4[primitiveBank]. c1:
  DISP4[bank0].
  DISP4[bank1].
  DISP4[bank2].
  DISP4[bank3].
  DISP4[bank4].
  DISP4[bank5].
  DISP4[bank6].
  DISP4[bank7].
  DISP4[bank8].
  DISP4[bank9].
  DISP4[bank10].
  DISP4[bank11].
  DISP4[bank12].
  DISP4[bank13].
  DISP4[bank14].
  DISP4[bank15].
  c2. at[0, 10, primitiveBank].
  c2. at[1, 10, primitiveBank].
  c2. at[2, 10, primitiveBank].
  c2. at[3, 10, primitiveBank].
  c2. at[4, 10, primitiveBank].
  c2. at[5, 10, primitiveBank].
  c2. at[6, 10, primitiveBank].
  c2. at[7, 10, primitiveBank].
  c2. at[8, 10, primitiveBank].
  c2. at[9, 10, primitiveBank].
  c2. at[10, 10, primitiveBank].
  c2. at[11, 10, primitiveBank].
  c2. at[12, 10, primitiveBank].
  c2. at[13, 10, primitiveBank].
  c2. at[14, 10, primitiveBank].
  c2. at[15, 10, primitiveBank].
  somethingIsDefinitelyWrong: 
  GOTO [somethingIsDefinitelyWrong].
  GOTO [sendArithmeticMessage].
  GOTO [sendArithmeticMessage].
  GOTO [sendArithmeticMessage].
  GOTO [sendArithmeticMessage].
  GOTO [sendArithmeticMessage].
  GOTO [sendArithmeticMessage].
  GOTO [sendArithmeticMessage].
  GOTO [sendArithmeticMessage].
  GOTO [sendArithmeticMessage].
  c3. at[0, 10, bank0].
  c3. at[1, 10, bank0].
  c3. at[2, 10, bank0].
  c3. at[3, 10, bank0].
  c3. at[4, 10, bank0].
  c3. at[5, 10, bank0].
  c3. at[6, 10, bank0].
  c3. at[7, 10, bank0].
  c3. at[8, 10, bank0].
  c3. at[9, 10, bank0].
  c3. at[10, 10, bank0].
  c3. at[11, 10, bank0].
  c3. at[12, 10, bank0].
  c3. at[13, 10, bank0].
  c3. at[14, 10, bank0].
  c3. at[15, 10, bank0].
  c3. at[16, 10, bank0].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  c3. at[0, 10, bank1].
  c3. at[1, 10, bank1].
  c3. at[2, 10, bank1].
  c3. at[3, 10, bank1].
  c3. at[4, 10, bank1].
  c3. at[5, 10, bank1].
  c3. at[6, 10, bank1].
  c3. at[7, 10, bank1].
  c3. at[8, 10, bank1].
  c3. at[9, 10, bank1].
  c3. at[10, 10, bank1].
  c3. at[11, 10, bank1].
  c3. at[12, 10, bank1].
  c3. at[13, 10, bank1].
  c3. at[14, 10, bank1].
  c3. at[15, 10, bank1].
  c3. at[16, 10, bank1].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  c3. at[0, 10, bank2].
  c3. at[1, 10, bank2].
  c3. at[2, 10, bank2].
  c3. at[3, 10, bank2].
  c3. at[4, 10, bank2].
  c3. at[5, 10, bank2].
  c3. at[6, 10, bank2].
  c3. at[7, 10, bank2].
  c3. at[8, 10, bank2].
  c3. at[9, 10, bank2].
  c3. at[10, 10, bank2].
  c3. at[11, 10, bank2].
  c3. at[12, 10, bank2].
  c3. at[13, 10, bank2].
  c3. at[14, 10, bank2].
  c3. at[15, 10, bank2].
  c3. at[16, 10, bank2].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  GOTO [noSuchPrimitive1 Microrcode].
  c3. at[0, 10, bank3].
  c3. at[1, 10, bank3].
  c3. at[2, 10, bank3].
  c3. at[3, 10, bank3].
  c3. at[4, 10, bank3].
  c3. at[5, 10, bank3].
  c3. at[6, 10, bank3].
  c3. at[7, 10, bank3].
  c3. at[8, 10, bank3].
  1
GOTO[noSuchPrimitiveInMicrocode].
GOTO[noSuchPrimitiveInMicrocode].
GOTO[noSuchPrimitiveInMicrocode].

c3. at[0, 10, bank3];
c3. at[0a, 10, bank3];
c3. at[0b, 0, bank3];

primitiveAt:
{_unixReceiverHigh/Low must be valid, receiver must not be smallInt}
Xbus = _unixReceiver, XDisp = _atPrim,
MAR = [stackHigh, stackLow + 0] {read index}. BRANCH[$, primAtFa1A, 0e], c1:
temp3High = primitiveAt,
temp3Low = MD, XDisp, CALL[positive16BitValueOf].

Q = Q + objectHeaderSize, BRANCH[$, atFailedA],
positive16BitValueOf-return;
ultIndex = Q, CALL[getobjectSize] (which establishes Q as the size of the object), c2:
templow = temp1low + classFieldOffset;
c1. at[atPrim, 10, getobjectSize-return];

commonAt:
L2 = classNameAt,
Utlow = _unixReceiversClass, CALL[fixedFieldsOf].

c2;
c3;

c3;

c3;

commonAtAlreadyHaveFixedFields:
Ybus = temp3low LHot4, XDisp, {test words bit of instanceSpace},
temp1low = ultIndex, BRANCH[getIndexingBytes, indexingWords, 0g].
c1. at[classNameAt, 10, fixedFieldsOf-return];
c3.

indexingBytes:
L2 = _atPrim,
temp1low = temp1low + objectHeaderSize, L3Disp,
ultIndex = temp1low, BRANCH[$, returnToAtPut, 0e].
c2;
c3;

c3;

c1;

c1;

c1;

temp1High = _unixReceiverHigh, c1:

getloclow = _unixReceiverlow, CALL[getloclowOrAddress].
temp3low = LShift1 temp3low, SE = 1, GOTO[atSmash].
c1. at[atPrim, 10, getloclowOrAddress-return];

indexingWords:
temp1low = temp1low (index + objectHeaderSize) + temp2low {fixed fields}, c1:
[1] = Q - temp1low, Carry78r (valid index?),
temp2low = _unixReceiverslow, BRANCH[atRangeError, $].
c1;
c3;

c3;

c3;

c1. at[atPrim, 10, commonAt-return];

c1;

c1;

atSmash:
MAR = [temp1High, temp1low + 0],
Ybus = temp3low LHot4, XDisp,
temp3low = MD, BRANCH[isWordObject, atSmash, 7], c1:
c2;
c3;

isWordObject:
GOTO[bytecodeFailed].
c1;

primAtFa1A:

Moop,
GOTO[activateNewMethod].
c2;
c3;

atRangeError:
{if this is a non-pointer object, and the subscript is out of range, it might be the current display or cursor bitmap.
if so, punt to Moop. If not, fail the primitive and run the method}
temp2High = _unixRecordHigh,
temp2Low = _unixRecordLow,
Moop.
c1;
c2;
c3;

MAR = [temp2High, temp2low + displayBitmapTopOffset],
CANCELBR[$, 2],
Q = MD,
c1;
c2;
c3;

MAR = [temp2High, temp2low + cursorBitmapTopOffset],
[1] = Q xor _unixReceiver, ZeroBr, CANCELBR[$, 2],
Q = MD, BRANCH[$, atFailedC].
c1;
c2;
c3;

[1] = Q xor _unixReceiver, ZeroBr,
GOTO[activateNewMethod].

c3;

atFailedA:
{zero subscripts are always invalid!}
Moop,
GOTO[activateNewMethod].
c2;
c3;

atFailedC:
AtFailedC:
GOTO[bytecodeFailed];

AtFailedD:
GOTO[AtFailedC];

GetObjectSize:

tempHigh = uNewReceiverHigh,
tempLow = uNewReceiverLow,
tempLow = tempLow + sizeFieldOffset.

MAR = [tempHigh, tempLow + 0],
tempLow = uNewReceiverLow, LDIsp, Q + HD {size}, RET[GetObjectSize-return].

PrimitiveAtPut:
(maxlength/low must be valid, receiver must not be SmallInt)
tempHigh + primitiveAtPut.

MAR = [stackHigh, stackLow + 0], L3 + atPutPrim,
stackLow = stackLow - 1.
tempLow = HD {the value}.

MAR = [stackHigh, stackLow + 0],
stackLow = stackLow - 1,
tempLow = HD {the index}, XDisp, CALL[positive16BitValueOf],

uAtPutValue = tempLow, BRANCH[$, atPutFail],
positive16BitValueOf-return];
Q = Q + objectHeaderSize,
uAtIndex = Q,
XBus = uNewReceiver, XDisp, {test for SmallInt}
BRANCH[CALL] getObjectSize, primAtPutFallA, 0e].

tempLow = tempLow + classFieldOffset, CALL[commonAt],

Noop.
Noop.

returnToAtPut:
YBus = tempLow LROT4, XDisp,
uAtPutLow = tempLow, DISP[atPutTable, 3];

{storing a pointer}
tempLow = uNewReceiverLow.

tempLow = tempLow + deltaWordOffset,

Noop.
Noop.

MAR = [tempHigh, tempLow + 0] {read delta word to determine volatility},
templow = uAtPutLow,
YBus = HD, XDisp.

MAR = [tempHigh, tempLow + 0] {get old contents, and refer to it},
BRANCH[$, atPutIsVolatile, 0b],

Noop.

otLow = HD, XDisp, CALL[refer], LI + viaPrimitiveAtPut,

tempHigh = uNewReceiverHigh,
tempLow = uAtPutLow,

Noop.

MAR = [tempHigh, tempLow + 0],

MDR = uAtPutValue,

otLow + uAtPutValue, XDisp, CALL[refer].

AtPutIsVolatile:

MDR = uAtPutValue, LOOPHOLE[medrok].

Primitives.mc 9-Jul-84 18:12:02 PDT
atPutWrapup:

    Noop.
    MAR = [stackHigh, stackLow + 0] {smash value on stack},
    MDR = nilPointer,
    stackLow = stackLow - 1,
    temp3Low = uAtPutValue,
    Noop,
    GOTO[atSmash].

improperInstanceSpec:
    GOTO[improperInstanceSpec].
    {storing a word}
    Noop,
    GOTO[bytecodeFailed],
    {storing a byte}
    Noop,
    GOTO[bytecodeFailed],

atPutFail:
    Noop,
    prmAtPutFailA:
    GOTO[activateNewMethod].

primitiveSize:

    Xbus = uNewReceiver, XDisp,
    BRANCH[$, sizeFast1C, 0],
    L3 = sizePrim,
    atLow = uNewReceiversClass, L2 = forSizePrim, CALL[fixedFieldsOf],
    CALL[getSizeObject].
    [return]
    temp2Low = XDisp, [on the instance spec],
    temp3Low = Q - objectHeaderSize, DISP4[sizeTable, 3],
    {pointers}
    temp3Low = temp3Low (size = objectHeader) - temp2Low (fixed fields),
    NegBr, GOTO[tryForSmall].
    {invalid}
    badInstanceSpec:
    GOTO[badInstanceSpec].
    {words - could be current display bitmap}
    temp2High = uRunRecordHigh,
    temp2Low = uRunRecordLow,
    Noop,
    Noop.
    MAR = [temp2High, temp2Low + displayBitmapOffset],
    CANCELBR[$, 2],
    Q = MD,
    [return]
    Q = Q xor uNewReceiver, ZeroBr,
    BRANCH[$, sizeFast1A],
    [return]
    temp3Low, NegBr, GOTO[tryForSmall].

sizeFast1B:
    CANCELBR[$, 3],
sizeFast1A:
    Noop,
    GOTO[bytecodeFailed],
sizeFast1C:
    Noop.
GOTO[activateNewMethod], c3;
(bytes)
tempLow + tempLow + deltaWordOffset, c3, at[3, 10, sizeTable];
MAR + [tempHigh, tempLow + O]. c1;
tempLow + tempLow + tempLow. PgCrVDisp, c2;
Ybus + MD, XDisp (test odd bit). BRANCH[$,龇sizeFall, 2], c8;
BRANCH[sizeEven, sizeOdd, 6d]. c1;
sizeEven:
GOTO[byteSizeOk], c2;
sizeOdd:
tempLow + tempLow + 1, GOTO[byteSizeOk], c2;
byteSizeOk:
Noop, c3;
tryForSmall:
    tempLow + tempLow + tempLow + 1. PgCrVDisp, BRANCH[$, sizeFall1B], c1;
    BRANCH[sizeOk, sizeFall1, 2], c2;
sizeOk:
    GOTO[pushTempLowAndDispatch], c3;
sizeFall1:
    Noop, c3;
byteSizeFall1:
    CANCEL[bytecodeFailed, 0f]. c1;

primitiveStringAt:
{unicodeReceiverHigh/Low must be valid}
Noop, c3, at[0f, 10, bank3];
MAR + [stackHigh, stacklow + O] (read index), L3 + stringAtPrim, c1;
temp3High = primitiveStringAt, c2;
temp3Low + ND, XDisp, CALL[positive16BitValueOf], c3;
Q = Q + twiceObjectHeaderSize, BRANCH[$, stringAtFall1A], L2 + stringAtPrim, c1, at[primitiveStringAt, 10,
positive16BitValueOf-return];
uintIndex = Q, CALL[getObjecSize], c2;
{the call on getObjecSize returns directly to getByteOrAddress, which in turn returns directly to
the following instruction:}
Noop, getByteOrAddress-return];
    Noop, c3, at[stringAtPrim, 10,
Noop, c1;
L1 + primStringAt, c2;
o1ow + characterTable10p, CALL[0Map], c3;
temp1ow + temp1ow + objectHeaderSize, c1, at[primStringAt, 10, 0Map-return];
temp1ow = temp1ow + temp3low, c2;
Noop, c3;
MAR + [temp1High, temp1low + 0], c1;
Noop, c2;
temp3low + ND, GOTO[atSmash], c3;

getByteOrAddress:
temp3low = LShift1 Q (object size). SE + 0, {yields object size in bytes}. c1, at[stringAtPrim, 10,
getObjecSize-return];
Q = uintIndex, c2;
temp1ow = temp1ow + deltaWordoffset, c3;
MAR + [temp1High, temp1low + O] (read the instances delta word), c1;
temp1ow = unicodeReceiverLow, c2;
Ybus + MD, XDisp (and dispatch so that we can test its odd bit), c3;
Q = Q - 1, (zero adjust), BRANCH[toStringLengthIsEven, toStringLengthIsOdd, 0], cl;

    toStringLengthIsEven: GOTO[toStringOnward], c2;
    toStringLengthIsOdd: temp3low = temp3low - 1, GOTO[toStringOnward], c2;
    toStringOnward: temp2low = uAtIndex [valid index?], c3;
                    [ ] = temp3low + temp2low, CarryBr, c1;
                    temp2low = RShift1 temp2low, SE = 0, GOTO[toStringAtFallB, $], c2;
                    temp2low = temp3low + temp2low, LDisp, c3;
                    MAR = [temp3High, temp2low + 0] (read correct word from object), BRANCH[$, getByteOrAddress, 0], LDisp, cl;
                    yBase = 0, CANCELBR[0, 0], YDisp [which byte do we want], c2;
                    temp3low = MD, BRANCH[fetchLeftByte, fetchRightByte, 0], c3;
        fetchLeftByte: temp3low = temp3low | Rot8, LDisp, GOTO[getByteOrAddress], c1;
        fetchRightByte: GOTO[getByteOrAddress], LDisp, c1;
        getByteOrAddress: RET[getByteOrAddress-return], c2;
    getStringAtFallA: Noop, c2;
    getStringAtFallB: GOTO[activateNewMethod], c3;

primitiveStringAtPut: {uNewReceiverHigh/Low must be valid)
    temp3High = primitiveStringAtPut, c3, at[0, 10, bank4];
    MAR = [stackHigh, stacklow + 0],
    LDisp = MD [the value], XDisp, CALL[getTimeClass], cl;
    temp3Low = temp3Low xor 0, ZeroBr,
    uAtPutValue = atLow, BRANCH[primitiveStringAtPutFallA, $], c2;
    temp3low = temp3Low + firstFieldOfObject, Noop, c1;
    Noop, cl;
    MAR = [temp3High, temp3Low + 0],
    stackLow = stackLow - 1,
    temp3Low = MD [the ascii value of the character], cl;
    MAR = [stackHigh, stackLow + 0] (read index), LDisp = stringAtPutPrim,
    stackLow = stackLow + 1, c2;
    temp3Low = MD, XDisp, CALL[positive16BitValueOF], c3;
    Q + 0 = twiceObjectHeaderSize, BRANCH[$, stringAtPutFallB],
    positive16BitValueOF-return;
    uAtIndex = 0, c1;
    temp2Low = RShift1 temp2low, SE = 0, c2;
    LDisp = stringAtPutPrim, c1;
    uAtPutLow = temp2Low, CALL[getObjectSize], c2;
    Noop, c1;
    at[stringAtPutPrim, 10, getByteOrAddress-return];
    Noop, c2;
    CALL[getObjectSize], c3;
    temp2Low = uAtPutLow, getByteOrAddress-return];
    Noop, c2;
    at[stringAtPutPrim, 10, c3;
MAR = [temp1High, temp3Low + 0] (read correct word).
Ybus = 0, YDisp (which byte?),
temp3Low + MD, BRANCH[keepRightByte, keepLeftByte, 0e].

keepLeftByte:
temp3Low + temp3Low LRot8, GOTO[stringAtPutAnd].

keepRightByte:
GOTO[stringAtPutAnd].

stringAtPutAnd:
temp3Low + temp3Low and 0f,
temp2Low + temp2Low LRot8,
Ybus = 0, YDisp (need to rotate?).
temp3Low or temp2Low, BRANCH[atPutOK, atPutRotate, 0e].

atPutOK:
GOTO[stringAtPutWrite].

atPutRotate:
temp3Low + temp3Low LRot8, GOTO[stringAtPutWrite].

stringAtPutWrite:
MAR = [temp1High, temp1Low + 0],
MDR + temp3Low, GOTO[atPutWrapup].

stringAtPutFlag:
GOTO[stringAtPutFlag].

stringAtPutFlag:
Noop,
GOTO[activateNewMethod].

GOTO[noSuchPrimitiveInMicrocode],
GOTO[noSuchPrimitiveInMicrocode],
GOTO[noSuchPrimitiveInMicrocode].

primitiveObjectAt:
Xbus = uNewReceiver, XDisp (small ints are a no-no)
MAR = [stackHigh, stackLow + 0], BRANCH[$, objectAtFlagA, 0e],
temp3High = primitiveObjectAt,
temp3Low = MD, XDisp, CALL[positive16BitValueOf],
Q = 0 + objectHeaderSize, BRANCH[$, objectAtFlagB],
positive16BitValueOf-return;
putAtIndex = Q,
temp3High = uNewReceiveOffset,
temp3Low = uNewReceiveValue,
templow = temp3Low + objectHeaderSize,
templow = 0 (no fixed fields for commonAt),
MAR = [temp1High, temp1Low + 0],
temp3Low = RShift1 (temp3Low and 7f), SE = 0, (extract literal count),
Q = temp3Low + objectHeaderSize + 1(yields max legal index including header), L3 = atPrim (tell commonAt to fall thru, not return), c2;
templow = uAtIndex (argument + objectHeaderSize), GOTO[ indexingWords].

at[1, 10, bank4];
at[2, 10, bank4];
at[3, 10, bank4];
at[4, 10, bank4];

objectAtFallA:
GOTO[objectAtFallA],
c2;

objectAtFallB:
Noop,
c2;

objectAtFallC:
GOTO[activateNewMethod],
c3;
GOTO[noSuchPrimitiveInMicrocode],
c3, at[6, 10, bank4];

primitiveNew:
L2 = newPrimitive,
c3, at[6, 10, bank4];
temp3High = viaPrimitiveNew {return linkage for instantiation},
c1;
L3 = 0 {flag = new with no argument},
c2;
L3Disp, GOTO[primitiveNewCommon],
c3;

primitiveNewWithArg:
Noop,
c3, at[7, 10, bank4];
MAR = [stackHigh, stacklow + 0] {read argument},
L3 = 1 {flag = new with argument},
c1;
temp3High = primitiveNew,
c2;
temp3Low = HD, XDisp, CALL[positive16bitValueOf],
c3;
stacklow = stacklow - 1 {point at class to instantiate},
CANCELBR[8, 1], c1, at[primitiveNew, 10,
positive16bitValueOf-return];
L2 = newPrimitve,
c2;
L3Disp,
c3;

primitiveNewCommon:
{read class pop from stack, then, if new with arg, readjust stackLow to point at tos, L2 must already be set to
newPrimitive} MAR = [stackHigh, stacklow + 0],
BRANCH[vanillaNew, argumentativeNew, 0e], c1;

vanillaNew:
GOTO[newOnward],
c2;

argumentativeNew:
stacklow = stackLow + 1 {point at tos},
GOTO[newOnward],
c2;

newOnward:
olow = HD, CALL[fixedFieldsOf],
c3;
(new use the InstanceSpec twice – once to test for indexability, once for which kind of object to create)
temp3Low = temp3Low Ldut4 {rotate InstanceSpec for dispatch},
LDisp {args/args flag},
c2, at[newPrimitive, 10,
fixedFieldsOf-return];
[ ] = temp3Low, YDisp {to test indexable flag},
BRANCH[newNoArgs, newArgs, 0e], c3;

newNoArgs:
[ ] = temp3Low, YDisp {for type of object to create},
BRANCH[newOke, newFalla, 0d], c1;

newArgs:
BRANCH[newFallb, newOdb, 0d],
c1;

newOke: {new without argument}
temp3Low = temp3Low {the fixed size},
DISP[whichFlavor, 3],
c2;

newOka: {new with argument}
temp3High = viaPrimitiveNew {return linkage for instantiation},
Noop,
c2;
[ ] = temp3Low, YDisp {for type of object to create},
temp3Low = temp3Low {fixed size + 0 {variable size},
DISP[whichFlavor, 3], c2;

uClassToInstantiate = olow, CALL[createInstanceWithBytes],
c3, at[3, 10, whichFlavor];
uClassToInstantiate = olow, CALL[createInstanceWithWords],
c3, at[7, 10, whichFlavor];
uClassToInstantiate = olow, CALL[createInstanceWithPointers],
c3, at[0f, 10, whichFlavor];

uClassToInstantiate = olow {oop of the new object},
L3Disp, createInstance-return;
BRANCH[newWithout, newWith, 0e],
c1, at[viaPrimitiveNew, 10,
createInstance-return];
newWithout: {replace class to instantiate with new object}
GOTO[pushTempLowAndDispatch].
newWith: (all out requested size, then replace class to instantiate with new object)
    Noop.
    MAR = [stackHigh, stackLow + 0].
    MDR = allPointer,
    stackLow = stackLow - 1 (point at class field). GOTO[pushTempLowAndDispatch].

newFailed:
    CANCELBR[newFailed, 0F].
newFailed1:
    Noop.
newFailed:
    (failed because of argument problems. run the method)
    GOTO[activateNewMethod].

GOTO[nosuchPrimitiveInMicrocode],

primitiveInstVarAt:
    Xbus = uNewReceiver, XDisp, L3 = InstVarAtPrim,
    MAR = [stackHigh, stackLow + 0]. BRANCH[$, InstVarAtFail1A, 0a],
    tempHigh = primitiveInstVarAt,
    tempLow = MD, XDisp, CALL[positive16BitValueOf].
    Q = Q + objectHeaderSize, BRANCH[$, InstVarAtFail1B],
    positive16BitValueOf-return;
    utIndex = Q, CALL[getObjectSize],
    tempLow = tempLow + classfieldNameoffset,
    L2 = forInstVarAtPrim,
    oLow = uNewReceiversClass, CALL[fixedFieldsOf].
    temp2Low = 0 (do not consider fixed fields for InstVarAt), L3 = atPrim (so that we fall into the at code rather than returning),
    c2, at[forInstVarAtPrim, 10, fixedFieldsOf-return];
    Noop.
    GOTO[commonAtAlreadyHaveFixedFields].

InstVarAtFail1A:
    GOTO[InstVarAtFail1],
InstVarAtFail1B:
    Noop,
InstVarAtFail1:
    Noop.
    GOTO[bytecodeFailed].

GOTO[nosuchPrimitiveInMicrocode],
GOTO[nosuchPrimitiveInMicrocode],
GOTO[nosuchPrimitiveInMicrocode].

primitiveFirstChild:
    temp2Low = uNewReceiver (get the class we are looking for instances of), c3, at[0d, 10, bank4],
    oLow = 0 (search entire ot),
    findOne:
Noop.

considerNextA:
  otlow = otlow + 2 (next ot entry), CarryBr.
  MAR = [othigh, otlow + 1] (read object table entry), BRANCH[3, noMoreInstances], CANCELBR[2, 2],
temp1High = MD, XwdDisp,
  MAR = [othigh, otlow + 0], DISP2[findInstance],
  {in use}
  L2 = lookingForInstances,
  temp1Low = MD, CALL[getClassAlreadyHaveBase],

  [] = temp2Low xor temp3Low, Zerodr (correct class?),
  BRANCH[considerNextA, 3],
temp3Low = otlow, GOTO[pushTemp3LowAndDispatch] (yes, return oop),
  {forwarding block}
forwardingBlock:
  GOTO[forwardingBlock],

beingRecursivelyFreed:
  GOTO[forwardingBlock],
  GOTO[considerNextA].

noMoreInstances:
  CANCELBR[3, 2],
  GOTO[activatesNewMethod].

primitiveNextInstance:
  temp2Low = uNewReceiver->class,
  otlow = uNewReceiver, GOTO[findOne].

GOTO[noSuchPrimitiveInMicrocode].

primitiveBlockCopy:
  Noop.
  MAR = [stackHigh, stacklow + 0],
  stacklow = stacklow - 1, temp1Low = MD (argument count),
  temp1High = uNewReceiver->high,
  temp1Low = uNewReceiver->low,
  otlow = uNewReceiver (context oop),

primitiveBlockCopyViaDirectDispatch:
  uSaveHome = otlow,
  temp1Low = temp1Low + methodFieldOffset,
  stacklow = stacklow + 1 {in case of instantiation failure},
  MAR = [temp1High, temp1Low + 0],
  temp1Low = temp1Low - methodFieldOffset,
  Ybus = MD, XDisp,
  temp3High = viaBlockCopy, BRANCH[moreBlockCopy, blockCopyOfBlockContext, 0e], c1;

blockCopyOfBlockContext:
  temp1Low = temp1Low + homeFieldOffset,
  Noop.
  MAR = [temp1High, temp1Low + 0], L1 = blockCopyOfBlock,
  Noop,
  otlow = MD {oop of home context}, CALL[otMap2],
  uSaveHome = otlow,

moreBlockCopy:
  temp1Low = temp1Low + sizeFieldOffset,
  otlow = blockContextClassOop,
MAR = [temp1High, temp1Low + 0],
  uClassToInstance = 0low,
  temp3Low = MD (size of new block context),
  temp3Low = temp3Low - objectHeaderSize,
  Noop,
  CALL[createInstanceWithPointers],
  Noop,
  temp3Low = blockCopying,
  uMakeVolatileLinkage = temp3Low, CALL[makeVolatile]
  temp3Low = uCurrentMethodLow,
  temp3Low = f1ow - temp3Low, (word relative plus headerSize),
  temp3Low = temp3Low - objectHeaderSize (word relative),
  temp3Low = LShift1 temp3Low, SE = pc16 (byte offset into compiledMethod, note that the SE+pc16 has toggled pc16, so we will toggle it again soon),
  temp3Low = temp3Low + 4,
  temp3Low = LShift1 temp3Low, SE = 1, (yields SmallInt),
  temp3Low = temp3Low + instructionPointerFieldOffset,
  temp3Low = 0 + PC16 (toggle again),
  Noop,
  MAR = [temp1High, temp1Low + 0],
  MDR = temp3Low,
  temp3Low = temp3Low - offsetFromInstructionPointerToStackPointer,
  MAR = [temp1High, temp1Low + 0],
  MDR = 1 (zero as SmallInt),
  temp3Low = temp3Low - offsetFromStackPointerToArgCount,
  MAR = [temp1High, temp1Low + 0],
  MDR = uArgumentCount,
  temp3Low = temp3Low - offsetFromArgCountToInitialIP,
  MAR = [temp1High, temp1Low + 0],
  MDR = temp3Low,
  temp3Low = temp3Low - offsetFromInitialIPToHome,
  MAR = [temp1High, temp1Low + 0],
  MDR = uSaveHome,
  Noop,
  MAR = [stackHigh, stackLow + 0],
  MDR = nilPointer,
  stackLow = stackLow - 1,
  temp3Low = uNewObject,
  Noop,
  GOTO[pushTemp3LowAndDispatch].

primitiveValue:

{handles value, value:, value:value:, and so on. The receiver must already have been verified to be an instance of BlockContext and uArgumentCount must be correct}
  ollow = uNewReceiver,
  c3, at[1, 10, bank6],
  uMakeVolatileLinkage = 0,
  Noop,
  uNewContext0op = ollow, CALL[makeVolatile],
  temp3Low = temp3Low + senderFieldOffset,
  temp3Low = uArgumentCount,
  temp3Low = LShift1 temp3Low, SE = 1, (shift for easy compare),
  MAR = [temp1High, temp1Low + 0],
  temp3Low = temp3Low + offsetFromSenderToBlockArgCount,
  MAR = [temp1High, temp1Low + 0],
  temp3Low = Q xor nilPointer, ZeroBr,
  temp3Low = (block argument count, a SmallInt), BRANCH[blockAlreadyActive, $],
  c3;
  Q = temp2Low xor temp3Low, Zer0Br (verify that arg counts match),
  temp3Low = uArgumentCount, BRANCH[valueArgCountMismatch, $],
  stackLow = stackLow - temp2Low (point at blockContext oop),
  MAR = [stackHigh, stackLow + 0] (hack block context oop in active context),
  MDR = nilPointer,
  temp3Low = stackLow + temp2Low (last word to move),
  stackLow = stackLow - 1 (first word to move),
  temp3Low = temp3Low + offsetFromBlockArgumentCountToFirstTemp (first destination word),
  L1 = primitiveValue, c2,
  c3, at[primitiveValue, 10, transferWords-return],
  Noop,
tempLow = uMakeVolatileLow,
tempLow = tempLow + InitialInstructionPointerOffset,
Noop. c1;
tempHigh = tempHigh + [temp1High, tempLow + 0] (read initial ip),
tempLow = tempLow + offsetFromInitialIpToLip, c2;
Q = MD (the initial instruction pointer), c3;
MAR = [temp1High, tempLow + 0] (write ip), c1;
MDR = Q, c2;
tempLow = tempLow + offsetFromIpToStackPointer, c3;
MAR = [temp1High, tempLow + 0] (write stackPointer), c1;
MDR = tempLow, c2;
tempLow = tempLow + offsetFromSenderToStackPointer, c3;
MAR = [temp1High, tempLow + 0] (write caller), c1;
MDR = uActiveContextDop, c2;
temp2High = uRunRecordHigh, c3;
temp2Low = uRunRecordLow, c1;
Noop, c2;
MAR = [temp2High, temp2Low + leafContextDopOffset], c3;
CANCEL[$, 2], c1;
temp3Low = MD (oop of leaf context), c2;
Noop. c3;
[] = temp3Low xor uActiveContextDop, ZeroBr, c1;
BRANCH[valueNotLeaf, $], c2;
MAR = [temp2High, temp2Low + leafContextDopOffset], c3;
MDR = nilPointer, L1 = viaPrimitiveValue, CANCEL[$, 2], c1;
LOOPHOLE[wok], c2;
otLow = temp3Low Lnot0, XDisp, CALL[refd], c3;

valueNotLeaf:

Noop, c1;
GOTO[newActiveContext], c2;
st[viaPrimitiveValue, 10, refdReturn], c3;

blockAlreadyActive:

GOTO[bytecodeFailed], c1;

valueArgCountMismatches:

GOTO[blockAlreadyActive], c3;

GOTO[noSuchPrimitiveInMicrocode], c3;
st[$, 10, bank6], c1;

primitivePerform:

Q = uArgumentCount, c3;
Q = uArgumentCount,
temp2High = uActiveContextHigh (point at perform receiver), c1;
temp2Low = stackLow - Q, c2;
O = Q - 1, c3;
MAR = [temp2High, temp2Low + 0] (read perform receiver), L2 = primitivePerform, c1;
temp2Low = temp2Low + 1, c2;
notLow = MD, XDisp, CALL[getClass], c3;
MAR = [temp2High, temp2Low + 0] (read selector to be performed), L3 = 1 (this is a lookup for perform), c1;
notArg = uArgumentCount - Q, c2;
Q = MD, CALL[startMethodLookup], c3;

[now determine the arg count of method we are to perform]
[] = tempLow (methodHeader) Lnot4, XDisp, c1;
O = uArgumentCount, DISP[performFlagTable, 1], c2;
(flag = 0 .. 6) c3;
GOTO[noArgs], c3;
GOTO[performArgCountCheck], c3;
temp3Low = 1, GOTO[performArgCountCheck], c1;

GOTO[1, 10, performOrExecute-return], c2;
GOTO[1, 10, performFlagTable], c3;
GOTO[1, 10, performFlagTable], c3;
tempLow = 2, GOTO[performArgCountCheck],
tempLow = 3, GOTO[performArgCountCheck],
tempLow = 4, GOTO[performArgCountCheck],
tempLow = 0, GOTO[performArgCountCheck],
tempLow = 0, GOTO[performArgCountCheck],
tempLow = RShift1 (tempLow and 7F), SE = 0, {extract literal count}
          gotLow = uNewMethodLow,
          gotHigh = uNewMethodHigh,
          tempLow = tempLow + gotLow,
          tempLow = tempLow + objectHeaderSize,
          tempLow = tempLow - 1,
          Noop,
          MAR = [tempHigh, tempLow = 0] {read method extension},
          Noop,
          tempLow = WD,
          tempLow = LRot8,
          tempLow = RShift1 (tempLow and off), SE = 0,
          GOTO[performArgCountCheck],
performArgCountCheck:
           [] = tempLow xor O, ZeroBr,
           tempHigh = uActiveContextHigh, BRANCH[performFat1A, $],
           tempLow = stackLow - 0, {tempHigh/low is now destination high/low}
           uPrimitiveNumber = tempLow, ZeroBr, LI + performPrim,
           tempLow = uPrimitiveNumber {source limit}, BRANCH[performFat1C, $],
           stackLow = stackLow + 1 {source start}, CALL[transferWords],
           stackLow = stackLow - 1,
           tempLow = uNewMethodHeader, GOTO[executeNowMethodViaPrimitivePerform], c3;

noArgs:
           [] = 0, ZeroBr, {is send argcnt also 0?}
           uPrimitiveNumber = tempLow, BRANCH[performFat1B, $],
           Noop,
           MAR = [stackHigh, stackLow = 0] {smash perform selector},
           HDR = nilPointer,
           Noop,
           stackLow = stackLow - 1,
           tempLow = uPrimitiveNumber,
           tempLow = uNewMethodHeader, GOTO[executeNowMethodViaPrimitivePerform], c3;
performFat1A:
          Noop,
          GOTO[bytecodeFailed], c3;
performFat1B:
          Noop,
          GOTO[bytecodeFailed], c3;
performFat1C:
          Noop,
          GOTO[bytecodeFailed], c3;
GOTO[noSuchPrimitiveInMicrocode], c3, at[4, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode], c3, at[6, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode], c3, at[8, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode], c3, at[0a, 10, bank6];
GOTO noSuchPrimitiveInMicrocode.
GOTO noSuchPrimitiveInMicrocode.
GOTO noSuchPrimitiveInMicrocode.
GOTO noSuchPrimitiveInMicrocode.
GOTO noSuchPrimitiveInMicrocode.
GOTO noSuchPrimitiveInMicrocode.
GOTO noSuchPrimitiveInMicrocode.
GOTO noSuchPrimitiveInMicrocode.
GOTO noSuchPrimitiveInMicrocode.
GOTO noSuchPrimitiveInMicrocode.
c3. at[0, 10, bankc].
c3. at[1, 10, bankc].
c3. at[2, 10, bankc].
c3. at[3, 10, bankc].
c3. at[4, 10, bankc].
c3. at[5, 10, bankc].
c3. at[6, 10, bankc].
c3. at[7, 10, bankc].
c3. at[9, 10, bankc].
c3. at[0a, 10, bankc].
c3. at[0b, 10, bankc].
c3. at[0c, 10, bankc].
c3. at[0d, 10, bankc].
c3. at[0e, 10, bankc].
c3. at[0f, 10, bankc].
c3. at[0, 10, bankd].
c3. at[1, 10, bankd].
c3. at[2, 10, bankd].
c3. at[3, 10, bankd].
c3. at[4, 10, bankd].
c3. at[5, 10, bankd].
c3. at[6, 10, bankd].
c3. at[7, 10, bankd].
c3. at[9, 10, bankd].
c3. at[0a, 10, bankd].
c3. at[0b, 10, bankd].
c3. at[0c, 10, bankd].
c3. at[0d, 10, bankd].
c3. at[0e, 10, bankd].
c3. at[0f, 10, bankd].
c3. at[0, 10, banke].
c3. at[1, 10, banke].
c3. at[2, 10, banke].
c3. at[3, 10, banke].
c3. at[4, 10, banke].
c3. at[5, 10, banke].
c3. at[6, 10, banke].
c3. at[7, 10, banke].
c3. at[9, 10, banke].
c3. at[0a, 10, banke].
c3. at[0b, 10, banke].
c3. at[0c, 10, banke].
c3. at[0d, 10, banke].
c3. at[0e, 10, banke].
c3. at[0f, 10, banke].

noSuchPrimitiveInMicrocode:

(see if molasses implements this primitive)
tempHigh = uUnRecordHigh,
tempLow = uUnRecordLow,
temp2Low = uPrimitiveNumber.

(get the address of the primitive map)
RAR = [temp3High, temp3Low + primitiveMapLowOffset].
CANCELINES[2],
temp3Low = MD.

Primitives.mc 9-Jul-84 18:12:02 PDT
fixedFieldsOf:

upon entry, otlow must be the oop of the class. at exit, temp2Low contains the number of fixed fields, and temp3Low contains the instanceSpec. L2 is the linkage register. sets temp1High/low to be the base of the class. smashes l1)

(build mask for extracting the fixed size of the object from instance spec)

\[
\begin{align*}
temp2Low & = \text{off}, \text{l1 = instanceSpec}, \quad c1; \\
temp2Low & = \text{temp2Low LRot8}, \quad c2; \\
temp2Low & = \text{temp2Low or off, CALL (otMap2)}, \quad c3; \\
temp3Low & = \text{temp3Low + instanceSpecificationFieldOffset}, \quad c1; \\
& \text{at [instanceSpec, 10, otMap2-return];} \quad c2; \\
& \text{Noop,} \quad c3; \\
\text{Noop.} & \quad c3; \\
\text{MAR} & = \text{[tempHigh, temp1Low + 0]}, \quad c1; \quad \text{c4;}
\end{align*}
\]

(temp2Low = RShift1 (temp3Low and temp2Low), SE = 0, {yields fixed size}, RET [fixedFieldsOf-return], c1;)

positive16BitValueOf:

todo -- need to fail primitive if smallint and negative--watch out for the 12)
(upon entry, temp2Low is the oop or SmallInteger. There is a pending XDisp to test for SmallIntegers. temp3High is the return linkage register. smashes otlow, temp3Low and temp1High/low, and may smash L2. result returned to Q)

(use pending dispatch to see if SmallInteger or whether further work is needed)

\[
\begin{align*}
temp3Low & = \text{RShift1 temp3Low, SE = 0, BRANCH [pos16NotSmall, pos16IsSmall, 0E]}; \quad c1; \\
pos16IsSmall: & \quad \text{(just turn it into a useful number and return)} \\
& \text{t1 = 0 + tempHigh, XDisp, GOTO[returnFromPositive16BitValueof], c2;}
\quad \text{pos16NotSmall:} \\
& \text{0 + class LargePositiveIntegerPointer, L2 = pos16Bit,} \\
& \text{otlow = LShift1 temp3Low, CALL [getClass],} \\
& \text{[] = temp3Low xor 0, ZeroBr (is class LargePositiveInteger?)} \\
& \text{BRANCH [pos16Faila, 3].} \\
\text{temp3Low} & = \text{temp3Low + sizeFieldOffset}, \quad c3; \\
\text{MAR} & = \text{[tempHigh, temp1Low + 0]}, \quad c1; \\
\text{0 + sizeOfLargeIntegerForPositive16BitValueof,} \\
\text{temp3Low = MD {size of this object}.} \\
& \text{[] = temp3Low - 0, ZeroBr} \\
\text{temp3Low} & = \text{temp3Low - sizeFieldOffset, BRANCH [pos16Failb, 3].} \\
\text{temp3Low} & = \text{temp3Low + firstFieldOfObject,} \\
\text{MAR} & = \text{[tempHigh, temp1Low + 0],} \\
\text{Noop,} \\
\text{temp3Low} & = \text{MD,} \\
\end{align*}
\]
temp3low = temp3low LAST8,
Ybus = temp3High, XDisp,
returnFromPositive16BitValueOf;
Q = temp3Low, ZeroBr, RET[positive16BitValueOf-return],
pos16Fa1a:
  GOTO[activateNowMethod],
pos16Fa1b:
  GOTO[activateNowMethod].
bytecodes:

{Push Receiver Variable bytecodes}

pushReceiverVariable0:
  temp3Low = field0, GOTO[pushReceiverVariable],
  c1. bytecode[6];
pushReceiverVariable1:
  temp3Low = field1, GOTO[pushReceiverVariable],
  c1. bytecode[1];
pushReceiverVariable2:
  temp3Low = field2, GOTO[pushReceiverVariable],
  c1. bytecode[2];
pushReceiverVariable3:
  temp3Low = field3, GOTO[pushReceiverVariable],
  c1. bytecode[3];
pushReceiverVariable4:
  temp3Low = field4, GOTO[pushReceiverVariable],
  c1. bytecode[4];
pushReceiverVariable5:
  temp3Low = field5, GOTO[pushReceiverVariable],
  c1. bytecode[5];
pushReceiverVariable6:
  temp3Low = field6, GOTO[pushReceiverVariable],
  c1. bytecode[6];
pushReceiverVariable7:
  temp3Low = field7, GOTO[pushReceiverVariable],
  c1. bytecode[7];
pushReceiverVariable8:
  temp3Low = field8, GOTO[pushReceiverVariable],
  c1. bytecode[8];
pushReceiverVariable9:
  temp3Low = field9, GOTO[pushReceiverVariable],
  c1. bytecode[9];
pushReceiverVariable10:
  temp3Low = field10, GOTO[pushReceiverVariable],
  c1. bytecode[10];
pushReceiverVariable11:
  temp3Low = field11, GOTO[pushReceiverVariable],
  c1. bytecode[11];
pushReceiverVariable12:
  temp3Low = field12, GOTO[pushReceiverVariable],
  c1. bytecode[12];
pushReceiverVariable13:
  temp3Low = field13, GOTO[pushReceiverVariable],
  c1. bytecode[13];
pushReceiverVariable14:
  temp3Low = field14, GOTO[pushReceiverVariable],
  c1. bytecode[14];
pushReceiverVariable15:
  temp3Low = field15, GOTO[pushReceiverVariable],
  c1. bytecode[15];

(pushReceiverVariable:
  (upon entry, temp3Low must indicate the receiver field to be pushed. Including the size of the object header. Since
  the receiver has no lambdas, we need not check for them.)
  temp2Low = uReceiverLow,
  temp2High = uReceiverHigh, temp2Low = temp3Low + temp2Low,
  top = top + 0) [read the receiver's field].
  stackLow = stackLow + 1.
  temp3Low = MD, GOTO[pushTemp3LowAndDispatch].
  c2;  c3;

{Push Temporary Location bytecodes}

pushTemporaryLocation0:
  homelow = homelow + temp0, GOTO[pushTemporary],
  c1. bytecode[10];
pushTemporaryLocation1:  
  homeLow + temp1, GOTO[pushTemporary].

c1, bytecode[11];

pushTemporaryLocation2:  
  homeLow + homeLow + temp2, GOTO[pushTemporary].

c1, bytecode[12];

pushTemporaryLocation3:  
  homeLow + homeLow + temp3, GOTO[pushTemporary].

c1, bytecode[13];

pushTemporaryLocation4:  
  homeLow + homeLow + temp4, GOTO[pushTemporary].

c1, bytecode[14];

pushTemporaryLocation5:  
  homeLow + homeLow + temp5, GOTO[pushTemporary].

c1, bytecode[15];

pushTemporaryLocation6:  
  homeLow + homeLow + temp6, GOTO[pushTemporary].

c1, bytecode[16];

pushTemporaryLocation7:  
  homeLow + homeLow + temp7, GOTO[pushTemporary].

c1, bytecode[17];

pushTemporaryLocation8:  
  homeLow + homeLow + temp8, GOTO[pushTemporary].

c1, bytecode[18];

pushTemporaryLocation9:  
  homeLow + homeLow + temp9, GOTO[pushTemporary].

c1, bytecode[19];

pushTemporaryLocation10:  
  homeLow + homeLow + temp10, GOTO[pushTemporary].

c1, bytecode[20];

pushTemporaryLocation11:  
  homeLow + homeLow + temp11, GOTO[pushTemporary].

c1, bytecode[21];

pushTemporaryLocation12:  
  homeLow + homeLow + temp12, GOTO[pushTemporary].

c1, bytecode[22];

pushTemporaryLocation13:  
  homeLow + homeLow + temp13, GOTO[pushTemporary].

c1, bytecode[23];

pushTemporaryLocation14:  
  homeLow + homeLow + temp14, GOTO[pushTemporary].

c1, bytecode[24];

pushTemporaryLocation15:  
  homeLow + homeLow + temp15, GOTO[pushTemporary].

c1, bytecode[25];

pushTemporary:
  Noop.

c2;

pushTemporaryViaExtendedPush:
  (upon entry, homeLow must contain the low-order 16-bits of the absolute address of the temporary field to be loaded; no lambda checking is performed. At exit, homeLow is properly restored.)

  Noop.

c3;

  MAR = [homeHigh, homeLow + 0] (read the temporary).
  stackLow = stackLow + 1.
  temp3Low = MD,

  MAR = [stackHigh, stackLow + 0],
  NDR = temp3Low, NextBytecode, homeLow + uhomeLow,
  DISP32[bytecodes], ploW + ploW + PC16,

  (push Literal Constant bytecodes)

  pushLiteralConstant10:
    temp3Low = literalField0, GOTO[pushLiteralConstant].

  c1, bytecode[26];

  pushLiteralConstant11:
    temp3Low = literalField1, GOTO[pushLiteralConstant].

  c1, bytecode[27];

  pushLiteralConstant12:
    temp3Low = literalField2, GOTO[pushLiteralConstant].

  c1, bytecode[28];

  pushLiteralConstant13:
    temp3Low = literalField3, GOTO[pushLiteralConstant].

  c1, bytecode[29];

  pushLiteralConstant14:
    temp3Low = literalField4, GOTO[pushLiteralConstant].

  c1, bytecode[30];

  pushLiteralConstant15:
    temp3Low = literalField5, GOTO[pushLiteralConstant].

  c1, bytecode[31];

  pushLiteralConstant16:
    temp3Low = literalField6, GOTO[pushLiteralConstant].

  c1, bytecode[32];

  pushLiteralConstant17:
    temp3Low = literalField7, GOTO[pushLiteralConstant].

  c1, bytecode[33];

  pushLiteralConstant18:
    temp3Low = literalField8, GOTO[pushLiteralConstant].

  c1, bytecode[34];

  pushLiteralConstant19:
    temp3Low = literalField9, GOTO[pushLiteralConstant].

  c1, bytecode[35];
pushLITERALConstant10:
    temp3Low = literalField10, GOTO[pushLITERALConstant].
    cl. bytecode[7a];
pushLITERALConstant11:
    temp3Low = literalField11, GOTO[pushLITERALConstant].
    cl. bytecode[7b];
pushLITERALConstant12:
    temp3Low = literalField12, GOTO[pushLITERALConstant].
    cl. bytecode[7c];
pushLITERALConstant13:
    temp3Low = literalField13, GOTO[pushLITERALConstant].
    cl. bytecode[7d];
pushLITERALConstant14:
    temp3Low = literalField14, GOTO[pushLITERALConstant].
    cl. bytecode[7e];
pushLITERALConstant16:
    temp3Low = literalField16, GOTO[pushLITERALConstant].
    cl. bytecode[7f];
pushLITERALConstant18:
    temp3Low = literalField18, GOTO[pushLITERALConstant].
    cl. bytecode[30];
pushLITERALConstant17:
    temp3Low = literalField17, GOTO[pushLITERALConstant].
    cl. bytecode[31];
pushLITERALConstant18:
    temp3Low = literalField18, GOTO[pushLITERALConstant].
    cl. bytecode[32];
pushLITERALConstant19:
    temp3Low = literalField19, GOTO[pushLITERALConstant].
    cl. bytecode[33];
pushLITERALConstant20:
    temp3Low = literalField20, GOTO[pushLITERALConstant].
    cl. bytecode[34];
pushLITERALConstant21:
    temp3Low = literalField21, GOTO[pushLITERALConstant].
    cl. bytecode[35];
pushLITERALConstant22:
    temp3Low = literalField22, GOTO[pushLITERALConstant].
    cl. bytecode[36];
pushLITERALConstant23:
    temp3Low = literalField23, GOTO[pushLITERALConstant].
    cl. bytecode[37];
pushLITERALConstant24:
    temp3Low = literalField24, GOTO[pushLITERALConstant].
    cl. bytecode[38];
pushLITERALConstant25:
    temp3Low = literalField25, GOTO[pushLITERALConstant].
    cl. bytecode[39];
pushLITERALConstant26:
    temp3Low = literalField26, GOTO[pushLITERALConstant].
    cl. bytecode[40];
pushLITERALConstant27:
    temp3Low = literalField27, GOTO[pushLITERALConstant].
    cl. bytecode[41];
pushLITERALConstant28:
    temp3Low = literalField28, GOTO[pushLITERALConstant].
    cl. bytecode[42];
pushLITERALConstant29:
    temp3Low = literalField29, GOTO[pushLITERALConstant].
    cl. bytecode[43];
pushLITERALConstant30:
    temp3Low = literalField30, GOTO[pushLITERALConstant].
    cl. bytecode[44];
pushLITERALConstant31:
    temp3Low = literalField31, GOTO[pushLITERALConstant].
    cl. bytecode[45];

pushLITERALConstant:
    (upon entry, temp3Low must be the offset to the literal constant, including the object header and literalStart. Since
the current method has no lambdas, we need not check for them.)
    temp2Low = uCurrentMethodLow, (get the address of the current method) c2;
    temp2High = uCurrentMethodHigh, temp2Low = temp3Low + temp2Low, (and add in offset to appropriate literal) c3;
    MAR = [temp2High, temp2Low + 0], (read the literal oop) c1;
    stackLow = stackLow + 1, c2;
    temp3Low = MD, GOTO[pushTemp3LowAndDispatch]. c3;

{Push Literal Variable bytecodes}

pushLiteralVariable0:
    temp3Low = literalField0, backupIs0Bytes, GOTO[pushLiteralVariable].
    cl. bytecode[46];
pushLiteralVariable1:
    temp3Low = literalField1, backupIs0Bytes, GOTO[pushLiteralVariable].
    cl. bytecode[47];
pushLiteralVariable2:
    temp3Low = literalField2, backupIs0Bytes, GOTO[pushLiteralVariable].
    cl. bytecode[48];
pushLiteralVariable3:
  temp3Low = literalField3, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[43];
pushLiteralVariable4:
  temp3Low = literalField4, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[44];
pushLiteralVariable5:
  temp3Low = literalField5, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[45];
pushLiteralVariable6:
  temp3Low = literalField6, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[46];
pushLiteralVariable7:
  temp3Low = literalField7, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[47];
pushLiteralVariable8:
  temp3Low = literalField8, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[48];
pushLiteralVariable9:
  temp3Low = literalField9, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[49];
pushLiteralVariable10:
  temp3Low = literalField10, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[50];
pushLiteralVariable11:
  temp3Low = literalField11, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[51];
pushLiteralVariable12:
  temp3Low = literalField12, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[52];
pushLiteralVariable13:
  temp3Low = literalField13, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[53];
pushLiteralVariable14:
  temp3Low = literalField14, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[54];
pushLiteralVariable15:
  temp3Low = literalField15, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[55];
pushLiteralVariable16:
  temp3Low = literalField16, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[56];
pushLiteralVariable17:
  temp3Low = literalField17, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[57];
pushLiteralVariable18:
  temp3Low = literalField18, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[58];
pushLiteralVariable19:
  temp3Low = literalField19, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[59];
pushLiteralVariable20:
  temp3Low = literalField20, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[60];
pushLiteralVariable21:
  temp3Low = literalField21, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[61];
pushLiteralVariable22:
  temp3Low = literalField22, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[62];
pushLiteralVariable23:
  temp3Low = literalField23, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[63];
pushLiteralVariable24:
  temp3Low = literalField24, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[64];
pushLiteralVariable25:
  temp3Low = literalField25, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[65];
pushLiteralVariable26:
  temp3Low = literalField26, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[66];
pushLiteralVariable27:
  temp3Low = literalField27, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[67];
pushLiteralVariable28:
  temp3Low = literalField28, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[68];
pushLiteralVariable29:
  temp3Low = literalField29, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[69];
pushLiteralVariable30:
  temp3Low = literalField30, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[70];
pushLiteralVariable31:
  temp3Low = literalField31, backupIs0Bytes, GOTO[pushLiteralVariable], c1, bytecode[71];
pushLiteralVariable:
  (upon entry, temp3Low must be the offset to the op of the association, including the object header and literalStart. Since the current method contains no lambdas, we do not check for them. However, the association might have a lambda in it, so we do check after it is fetched)
temp3Low = uCurrentMethodHigh, (get the address of the current method) c2;
temp2High = uCurrentMethodHigh, temp3Low + temp3Low + temp3Low, (and add in offset to the appropriate association) c3;
MAR = [temp2High, temp2Low + 0], [read the association oop] c1;

push.mc 1-Aug-84 16:30:17 PDT
LI + pushingLiteralVariable, c2;
oflag + MD, CALL[otMap] (othMap the association), c3;
temp1Low + temp1Low + associationValueIndex (add in offset to value), c1, at[pushingLiteralVariable, 10, othMap-return];
Noop, c2;
Noop, c3;
MAR - [temp1High, temp1Low + 0], c1;
Noop, c2;
temp1Low - MD (get the value of the association), c3;
[ ] + temp1Low, ZeroRr, (test for lambda) c1;
BRANCH[valueIsNotLambda.pushLiteralVariable, valueIsLambda.pushLiteralVariable], c2;
valueIsNotLambda.pushLiteralVariable; c3;
stacklow + stacklow + 1 (not a lambda, ok to push), GOTO[pushTemp3LowAndDispatch], c3;
valueIsLambda.pushLiteralVariable;
GOTO[break1];
(Loom: need to call Loom to map association lambda) c3;

{Push (Receiver, true, false, nil, -1, 0, 1, 2)}

pushReceiver:
temp3Low = uReceiverOop, GOTO[pushConstantOop], c1, bytecode[70];
pushTrue:
temp3Low = truePointer, GOTO[pushConstantOop], c1, bytecode[71];
pushFalse:
temp3Low = falsePointer, GOTO[pushConstantOop], c1, bytecode[72];
pushNil:
temp3Low = nilPointer, GOTO[pushConstantOop], c1, bytecode[73];
pushMinusOne:
temp3Low = --temp3Low xor temp3Low, GOTO[pushConstantOop], c1, bytecode[74];
pushZero:
temp3Low = zeroPointer, GOTO[pushConstantOop], c1, bytecode[75];
pushOne:
temp3Low = onePointer, GOTO[pushConstantOop], c1, bytecode[76];
pushTwo:
temp3Low = twoPointer, GOTO[pushConstantOop], c1, bytecode[77];
pushConstantOop:
Noop, stackLow = stackLow + 1, GOTO[pushTemp3LowAndDispatch], c2;
{Extended Push (Receiver Variable, Temporary Location, Literal Constant, Literal Variable)}

extendedPush:
temp3Low + 1b (get the extension byte), c1, bytecode[80];
temp3Low + temp3Low and 3f. (and save only the offset) c2;
temp3Low + temp3Low + objectHeaderSize, (and bump to the object body) c3;
temp3Low + RShl1 temp3Low, (shift for convenient dispatch) c1;
[ ] + temp3Low lbTo, XwDisp. (dispatch on the type of extended push) c2;
[DISP][extendedPushTarget], licow + 1low + PC16, (account for the extension byte, and take off) c3;
[todo --- when it runs faster, check out that pushExtendedTemp/LitConst/LitVar work] GOTO[pushReceiverVariable],
c1, at[0, 4, extendedPushTarget];
temp3Low = temp3Low + tempFrameStart, c1, at[1, 4, extendedPushTarget];
homeLow = homeLow + temp3Low, GOTO[pushTemporaryViaExtendedPush], c2;
temp3Low = temp3Low + literalStart, GOTO[pushLiteralVariable], c1, at[2, 4, extendedPushTarget];
backupIsByte (in case a lambda is found), c1, at[3, 4, extendedPushTarget];
Noop, c2;
Noop, c3;
temp3Low = temp3Low + literalStart, GOTO[pushLiteralVariable], c1;
(Duplicate Stack Top)

duplicateStackTop:
  MAR = [stackHigh, stackLow + 0] {start read of stackTop},
  stackLow = stackLow + 1,
  temp3Low = NO, GOTO[pushTemp3LowAndDispatch],

(Push Active Context)

pushActiveContext:
  Q = uRunRecordHigh {retrieve the Run Record address},
  temp2High = Q LRot0,
  temp2Low = uRunRecordLow,
  MAR = [temp2High, temp2Low + leafContextDopOffset],
  CANCEL[R$, 2],
  temp1Low = HO {oop of the leafContext},
  otlow = uActiveContextDop,
  l1 = pushingActiveContext,
  [ ] = temp1Low - otlow, ZeroBr,
  BRANCH[leafContextIsNotActiveContext, leafContextIsActiveContext],

leafContextIsActiveContext:
  MAR = [temp2High, temp2Low + leafContextDopOffset],
  MDR = nilPointer, CANCEL[R$, 2], LOOPHOLE[wok],
  [ ] = otlow LRot0, XDisp (not really necessary—it better be an oop), CALL[refd],

leafContextIsNotActiveContext:
  temp3Low = otlow,
  stackLow = stackLow + 1,
  GOTO[pushTemp3LowAndDispatch],
(Refill and Trap microcode for the Smalltalk Virtual Machine
Introduction to the Refill microcode:

As each Smalltalk bytecode implementation nears completion, an IBDisp is executed. The action that then occurs is summarized in the following table:

<table>
<thead>
<tr>
<th>IB state</th>
<th>Messa Interrupt Pending</th>
<th>No Messa Interrupt Pending</th>
</tr>
</thead>
<tbody>
<tr>
<td>full</td>
<td>trap to location 600</td>
<td>branch to next macroinstruction interpreter</td>
</tr>
<tr>
<td>empty</td>
<td>trap to location 600</td>
<td>trap to location 400</td>
</tr>
<tr>
<td>not empty</td>
<td>trap to location 700</td>
<td>trap to location 600</td>
</tr>
</tbody>
</table>

If a Messa Interrupt is pending (locations 600 and 700), we pack up the Smalltalk interpreter state and punt to Molasses. Otherwise, we refill the Instruction Buffer and continue interpreting bytecodes. The code at location 400 (empty Instruction Buffer) reads a word and loads it into the Instruction Buffer, then starts a read of the next word while simultaneously starting a dispatch on the first byte of the Instruction Buffer. If we have trapped to location 600 (non-empty Instruction Buffer) we must read one additional word and load it into the Instruction Buffer.

The reader should note that we unconditionally refill the Instruction Buffer, even if the subsequent bytes are not needed. Thus, if the buffer is empty, we read two words even though we may never execute the last 3 bytes. We take this approach for simplicity and speed, but it does mean that we can potentially read a word beyond the end of the Object Space. This is not a problem if the Object Space is not at the end of the Smalltalk memory (e.g., you could place the Object Table after the Object Space, or simply make the last word of the Object Space unavailable for allocation).

MacroDef[AlwaysIBDisp. (IBDisp. IBPtr + 1)] (same definition as in Mese.df):

stEmpty:
{ when we arrive here, the buffer is empty, and the ip is pointing at the word to be fetched}
MAR = [spHigh, ipLow+0],
ipLow = ipLow + 1,
IB = MD, GOTO[nextWord-stNotEmpty],

stNotEmpty:
{ when we arrive here, the buffer is not empty, and the ip is one word short of pointing at the word to be fetched}
ipLow = ipLow + 1,
Ybus = uTimeToStabilize. ZeroBr,
BRANCH[stabilizeNow, nextWord-stNotEmpty],

stabilizeNow:
LO = uCodeRequestedStabilization,
Noop,
CALL[stabilize],

otlow = uActiveContextDop,
stabilize-return];
temp3Low = activeAfterStabilize,
Noop,

uMakeVolatileLinkage + temp3Low, CALL[makeVolatile],
temp2High = uRunRecordHigh,
makeVolatile-return];
temp2Low = uRunRecordLow,
Noop,

MAR = [temp2High, temp2Low = homeContextDopOffset].

temp3Low = homeAfterStabilize, CANCELBR[3, 2], c2; c3;
4thLow = HD, c3;
5MakeVolatileLinkage = temp3Low, CALL[makeVolatile], c1;
temp2High = uNumRecordHigh, makeVolatile-return], c1, at[homeAfterStabilize, 10.
temp2Low = uNumRecordLow, c2;
5oop,
c3;
MAR = [temp2High, temp2Low + 1, leafContextOopOffset], c1;
temp3Low = leafAfterStabilize, CANCELBR[3, 2], c2;
4thLow = MD, c3;
5MakeVolatileLinkage = temp3Low, CALL[makeVolatile], c1;
MAR = [ipHigh, ipLow+0], GOTO[nextWord-notNotEmpty-c2], c1, at[leafAfterStabilize, 10.
makeVolatile-return];

nextWord-notNotEmpty:
MAR = [ipHigh, ipLow+0], c1;

nextWord-notNotEmpty-c2:
ipLow = ipLow - 1, (adjust to point at the proper word), AlwaysIBDisp, L3 = 0, c2;
IB = MD, OTSPHI[bytecodes], c3;

stEmptyOrFullAndMesaInterrupt;
GOTO[messaInterrupt], c1, at[600];
stNotEmptyAndMesaInterrupt;
5oop,
c1, at[700];
MesaInterrupt;
5oop,
c2;
tempLow = 0 (mark Mesa interrupt), GOTO[saveSmalltalkState], c3;

{
Introduction to the Trap microcode:
Certain conditions (control store parity errors, emulator memory errors, Mesa stackPointer overflow or underflow, and IB-empty errors) cause traps to location 0. Currently we just hang for any of these errors; future implementations probably want to take a more official action. The IB-empty error is useful in the Mesa emulator (where it is utilized to detect and handle page crossings), but in Smalltalk land it means that a coding error has been made and too many bytes have been fetched from the Instruction Buffer.
}

FatalError;
0 = ErrnIBStkp, c1, at[0];
FatalErrorSpin;
GOTO[bailout3], c2;
{  
    returnReceiver:  
        temp3Low = uReceiverOop, GOTO[ETPhoneHome].  
    returnTrue:  
        temp3Low = truePointer, GOTO[ETPhoneHome].  
    returnFalse:  
        temp3Low = falsePointer, GOTO[ETPhoneHome].  
    returnNil:  
        temp3Low = nilPointer, GOTO[ETPhoneHome].  

    (Return Stack Top From (Message, Block))  

    returnStackTopFromMessage:  
        MAR = [stackHigh, stackLow + 0], HMR = nilPointer,  
            temp3Low = MD,  
            stackLow = stackLow - 1.  
        ETPhoneHome:  
            (set up temp2 so that we can get the sender oop from the home context)  
            Q = homeHigh, backupIsOBytes,  
                temp2High = Q LRot6,  
                temp2Low = homeLow, GOTO[getSender].  

    returnStackTopFromBlock:  
        MAR = [stackHigh, stackLow + 0], backupIsOBytes,  
            HMR = nilPointer,  
            temp3Low = MD,  
            stackLow = stackLow - 1,  
            (set up temp2 so that we can get the sender oop from the active context)  
                temp2High = uActiveContextHigh,  
                temp2Low = uActiveContextLow,  
            loop.  

    getSender:  
        temp2Low = temp2Low + senderFieldOffset,  
            uReturnValue = temp3Low,  
            MAR = [temp2High, temp2Low + 0], L1 = getSenderBase,  
                temp3High = uRunRecordHigh,  
                otoLow = MD (oop of sender), CALL[otMap2] (to get the base of the sender context), c3;  

    returnValue:  
        (upon entry the oop is the oop of the context to return to, temp1High/low point at its objectHeader, and uReturnValue is the return value)  
        (if the sender oop is nil, we cannot return)  
            [] = otoLow xor nilPointer, ZeroBr,  
                temp2Low = temp2Low + instructionPointerFieldOffset, BRANCH[$, cannotReturn], c2;  
            loop.  
        MAR = [temp1High, temp1Low + 0] (read instruction pointer),  
            temp3Low = uRunRecordLow,  
                Q = MD (instruction pointer of context to return to), c3;  

        (if the instruction pointer is nil, also cannot return)  
            [] = Q xor nilPointer, ZeroBr,  
                temp2Low = uActiveContextLow, BRANCH[$, cannotReturn], c2;  

    returnToSenderContext:  
        (ok, we can return. save return contexts base address, then up reference count of target context and see if activeContext is leaf, or if returning from block, or non-leaf return)  
        Q = temp1High, c3;  
        lpHigh = Q LRot6, c1;  
        lpLow = temp1Low, c2;  
        homeLow = otoLow, c3;  

    (get the oop of the leaf)  
}
MAR = [tempHigh, tempLow + 1, leafContextDopOffset].
    temp2Low = temp2Low + senderFieldOffset, CANCELBR[$, 2].
    temp3Low = MD (leaf dop).

    (and see if active context is the leaf)
    [] = uActiveContextDop xor temp2Low, ZeroBr.
    temp2High = uActiveContextHigh, BRANCH[notReturningFromLeaf, returningFromLeaf].
    (by the way, the above assignment to temp2High provides the first time that temp2 is guaranteed to be pointing at the active context)

returningFromLeaf:

    (If activeContext is the current leaf, then clean out its inside so that it can be recycled, temp2High/Low is pointing at the sender field of the active context)
    [] = stackLow - temp2Low, NegBr. (are we done yet?)
    MAR = [temp2High, tempLow + 0], BRANCH[$, returnWrapup]
    HDR = nilPointer, (smash a field)
    temp2Low = temp2Low + 1,
    Noop.
    GOTO[returningFromLeaf].

notReturningFromLeaf:

    (see if the active context is a block context, by looking in the method field, temp2Low is pointing at the sender field of the active context)
    temp2Low = temp2Low + differenceBetweenSenderAndMethodFields.
    MAR = [temp2High, tempLow + 0], (read method field)
    temp2Low = uActiveContextLow,
    temp2Low = MD, XDisp,
    BRANCH[smashTwo, $, 0e].
    (this is a blockContext, so we look up the sender/caller chain and see if we find the context we are returning to)
    temp3High = uActiveContextHigh,
    temp2Low = uActiveContextLow.

contextChainChase:
    [] = oLow xor nilPointer, ZeroBr.
    [] = oLow xor homeLow, ZeroBr. BRANCH[$, isOnChain],
    temp2Low = temp2Low + senderFieldOffset, BRANCH[$, isOnChain].
    MAR = [temp3High, temp2Low + 0] (read sender from this context),
    LI = chasingContextChain,
    oLow = MD (next sender field), CALL[otMap2],
    Noop.
    isOnChain:
    oLow = uActiveContextDop.
    invalidateContexts:
    LI = invalidatingContext, CALL[otMap2],
    temp2Low = temp2Low + instructionPointerFieldOffset,
    Noop,
    Noop.
    MAR = [temp3High, temp2Low + 0] (smash inst ptr field),
    HDR = nilPointer,
    temp2Low = temp2Low - differenceBetweenSenderAndInstructionPointerFields.
    MAR = [temp3High, temp2Low + 0] (smash sender field),
    HDR = nilPointer,
    oLow = MD (get the next sender field),
    temp2Low = temp2Low - senderFieldOffset,
    temp2Low = temp2Low + deltaWordOffset,
    Noop.
    MAR = [temp3High, temp2Low + 0],
    Noop,
    temp3Low = MD (delta word of current context), XDisp,
    BRANCH[refdNextContext, dontRefdNextContext, Ob].
    refdNextContext:
    LI = nextContext,
    [] = oLow LR0, XDisp, CALL[refd],
    Noop.
    dontRefdNextContext:
    [] = oLow xor homeLow, ZeroBr. (are we done yet)
    BRANCH[$, wrapupReturn].
    GOTO[invalidateContexts].

notOnChain:
    CANCELBR[$, 1]..
    Noop.
    Noop.

Return.mc  3-Apr-84 21:20:26 PST
smashTwo:
  temp2Low = temp2Low + senderFieldOffset,
  nop, c2:
  MAR = [temp2High, temp2Low + 0], c1:
  HDR = v11Pointer, c2:
  temp2Low = temp2Low + differenceBetweenSenderAndInstructionPointerFields, c3;
  MAR = [temp2High, temp2Low + 0], c1:
  HDR = v11Pointer, c2:
  nop, c3:
wrapupReturn:
  nop, c1:
returnWrapup:
  L1 = downOldContextOnReturn,
  o1low = uActiveContextOop, XDisp, CALL[refd], c2:
  uNewContextOop = 0 or homeLow, CALL[fetchContextRegistersAndMakeContextVolatile], L0 = returningToAContext, c1,
  at[downOldContextOnReturn, 10, refdReturn]; c3, at[returningToAContext, 10,
  stackLow = stackLow + 1,
  fetchContextRegisters-return]; c3:
  MAR = [stackHigh, stackLow + 0],
  HDR = uReturnValue, c1:
  goto[fixupInstructionPointer], c2:
cannotReturn:
  goto[zot], c3:
cannotReturnB:
  goto[zot], c3:

zot:
  goto[bytecodeFailed], c1;
{Send Literal Selector}

sendLiteralSelectorWith3BitsOfArguments:
  temp2Low = temp2Low + 1b (get extension byte),
  t1Low = t1Low + PC16. backupIs1Byte, (account for extension byte)
  temp3Low = temp2Low and 1f (get literal index).
  c1, bytecode[83]:
  c2:
  c3:
  temp3Low = temp3Low + objectHeaderSize + literalStart,
  temp1Low = temp2Low LHrs12 (start getting argument count),
  temp1Low = RShift1 (temp1Low and 0e), SE = 0,
  c1:
  c2:
  c3:
  GOTO[getSelector],
  c1:
  c2:

{Send Literal Selector to Superclass}

sendLiteralSelectorToSuperclassWith3BitsOfArguments:
  temp2Low = temp2Low + 1b (get extension byte),
  t1Low = t1Low + PC16. backupIs1Byte, (account for extension byte)
  temp3Low = temp2Low and 1f (get literal index).
  c1, bytecode[83]:
  c2:
  c3:
  temp3Low = temp3Low + objectHeaderSize + literalStart,
  temp1Low = temp2Low LHrs12 (start getting argument count),
  temp1Low = RShift1 (temp1Low and 0e), SE = 0, GOTO[getSuperSelector],
  c3:

sendLiteralSelectorToSuperclassWith8BitsOfArguments:
  temp2Low = temp2Low + 1b (get extension byte),
  t1Low = t1Low + PC16. backupIs1Byte, (account for extension byte)
  temp3Low = temp2Long (get literal index).
  c1, bytecode[83]:
  c2:
  c3:
  temp3Low = temp3Low + objectHeaderSize + literalStart,
  GOTO[getSuperSelector],
  c3:

getSuperSelector:

( upon entry, temp3Low must contain the offset to the literal selector including the objectHeaderSize, temp1Low must contain the number of arguments)

  temp2Low = uCurrentMethodLow (get the current method address),
  c1:
  temp2High = uCurrentMethodHigh, temp2Low = temp2Low + temp3Low (add in offset to literal selector),
  temp3Low must contain the number of arguments.
  c2:
  stackLow = stackLow - temp1Low (point at the new receiver).
  stackLow = stackLow - temp1Low (point at the new receiver),
  c3:
  c3:
  MAR = [temp2High, temp2Low + 0] (read the literal selector),
  MAR = [temp2High, temp2Low + 0] (read the literal selector),
  c1:
  uArgumentCount = temp1Low (may need this later, so save it),
  uArgumentCount = temp1Low (may need this later, so save it),
  c2:
  MAR = [StackHigh, stackLow + 0].
  MAR = [StackHigh, stackLow + 0].
  c2:
  stockLow = stackLow + temp1Low (point at tos again),
  stockLow = stackLow + temp1Low (point at tos again),
  L2 = superclassReceiver, c2:
  ollow = MD (the new receiver),
  XDisp, CALL[getClass],
  c3:
  c3:
  uNewReceiverHigh = temp1High,
  uNewReceiverHigh = temp1High,
  c1, at[superclassReceiver, 10, getClass-return],
  c2:
  uNewReceiverHigh = temp1High,
  c3:

{ we save the class (contained in temp3Low) in a few clicks }

{ in order to do a send to super, we need to get the superclass of the current method. The oop of the class of the current method is kept in an association, whose oop is the last literal of the current compiled method. To get at the last literal, we need to know how many literals exist in the current compiled method. We do this by checking a little and
calling the getNewMethodHeader routine with it's arguments pointing at the current method. Then we fetch the
association oop, check it's value field for a lambda, then get the classes superclass and return it in temp3Low.)

temp1High = uCurrentMethodHigh,
temp1Low = uCurrentMethodLow,
uSelector = 0, L2 = gettingSuperClass,

CALL[getNewMethodHeader] (thus getting the current method's header), c1:
temp1Low = [RSHFLI temp1Low and 7F], GE = 0 {literal count of current method}.c2, at[gettingSuperclass, 10, 
getNewMethodHeader-return];
temp1Low = uCurrentMethodLow,

temp1Low = temp1Low + temp2Low,
temp1Low = temp1Low + objectHeaderSize,

uNewReceiver = otlow (save newReceiver -- next read smashes it), c3;

MAR = [temp1High, temp1Low + 0] {read last literal in current method}, L1 = getMethodClass, c1;
uNewReceiversClass = temp1Low,

otlow = MD, {the association oop}, CALL[otMap], c2;

temp1Low = temp1Low + associationValueIndex,

Noop.

Noop.

MAR = [temp1High, temp1Low + 0] {read value field of association}, c1:

Noop.

otlow = MD {the oop of the class of the current method}, c2;

]+ = otlow, ZeroBR, L1 = getSuperClass,

BRANCH[classListNotLambda, classListLambda]

classListNotLambda: {Loom: need to call Loom here)

GOTO[breakout], c2;

classListLambda: {Loom: need to call Loom here)

GOTO[breakout], c3;

(Send Arithmetic Messages)

sendArithmeticMessage0: {SmallInteger +}

MAR = [stackHigh, stackLow + 0] {start read of argument}. L1 = smallAdd, c1, bytecode[0B0];

stackLow = stackLow - 1 {point at receiver}, CALL[primitiveNeeds2SmallIntegers], backupIs8Bytes, c2;

temp3Low = temp2Low + temp3Low, GOTO[pushArithmeticResult], c2, at[smallAdd, 10, arithmeticPrimitives];

sendArithmeticMessage1: {SmallInteger -}.

MAR = [stackHigh, stackLow + 0] {start read of argument}. L1 = smallSubtract, c1, bytecode[0B1];

stackLow = stackLow - 1 {point at receiver}, CALL[primitiveNeeds2SmallIntegers], backupIs8Bytes, c2;

temp3Low = temp2Low + temp3Low, GOTO[pushArithmeticResult], c2, at[smallSubtract, 10, arithmeticPrimitives];

sendArithmeticMessage2: {SmallInteger ^}

MAR = [stackHigh, stackLow + 0] {start read of top of stack}. L1 = smallLess, c1, bytecode[0B2];

stackLow = stackLow - 1 {point at receiver}, CALL[primitiveNeeds2SmallIntegers], backupIs8Bytes, c2;

]+ = temp2Low = temp3Low, NegBR,

BRANCH[pushFalseInPrimitiveRelation0, pushTrueInPrimitiveRelation0], c2, at[smallLess, 10, arithmeticPrimitives];

sendArithmeticMessage3: {SmallInteger >

MAR = [stackHigh, stackLow + 0] {start read of top of stack}. L1 = smallGreater, c1, bytecode[0B3];

stackLow = stackLow - 1 {point at receiver}, CALL[primitiveNeeds2SmallIntegers], backupIs8Bytes, c2;

]+ = temp3Low = temp2Low, NegBR,

BRANCH[pushFalseInPrimitiveRelation0, pushTrueInPrimitiveRelation0], c2, at[smallGreater, 10, arithmeticPrimitives];

sendArithmeticMessage4: {SmallInteger <}

MAR = [stackHigh, stackLow + 0] {start read of top of stack}. L1 = smallLessOrEqual, c1, bytecode[0B4];

stackLow = stackLow - 1 {point at receiver}, CALL[primitiveNeeds2SmallIntegers], backupIs8Bytes, c2;

Noop., c2, at[smallLessOrEqual, 10, arithmeticPrimitives];
arithmeticPrimitives;
        Noop, c3;
        [] = tempLow - tempLow, NegBr, BRANCH[1, tsNeg], c1;
        [] = tempLow - tempLow, ZeroBr, BRANCH[$, tsNeg], c2;
        BRANCH[pushFalseInPrimitiveRelational, pushTrueInPrimitiveRelational], c3;
    tsNeg;
    CANCEL[pushTrueInPrimitiveRelational, 1], c3;
    sendArithmeticMessage: (SmallInteger ->)
        MAR = [stackHigh, stackLow + 0] (start read of top of stack), L1 = smallGreaterEqual, c1, bytecode[0bb];
        stackLow = stackLow - 1 (point at receiver), CALL[primitiveNeeds2SmallIntegers], backupIs0Bytes, c2;
        [] = tempLow - tempLow, NegBr, c2, at[smallGreaterEqual, 10, arithmeticPrimitives];
        BRANCH[pushFalseInPrimitiveRelational, pushTrueInPrimitiveRelational], c3;
    sendArithmeticMessage: (SmallInteger ->)
        MAR = [stackHigh, stackLow + 0] (start read of top of stack), L1 = smallEqual, c1, bytecode[0bb];
        stackLow = stackLow - 1 (point at receiver), CALL[primitiveNeeds2SmallIntegers], backupIs0Bytes, c2;
        [] = tempLow - tempLow, ZeroBr, c2, at[smallEqual, 10, arithmeticPrimitives];
        BRANCH[pushFalseInPrimitiveRelational, pushTrueInPrimitiveRelational], c3;
    sendArithmeticMessage: (SmallInteger ->)
        MAR = [stackHigh, stackLow + 0] (start read of top of stack), L1 = smallNotEqual, c1, bytecode[0bb];
        stackLow = stackLow - 1 (point at receiver), CALL[primitiveNeeds2SmallIntegers], backupIs0Bytes, c2;
        [] = tempLow - tempLow, NotZeroBr, c2, at[smallNotEqual, 10, arithmeticPrimitives];
        BRANCH[pushFalseInPrimitiveRelational, pushTrueInPrimitiveRelational], c3;
    sendArithmeticMessage: (SmallInteger *)
        backupIs0Bytes, Q = 0bb, GOTO[notYetInvented], c1, bytecode[0bb];
    sendArithmeticMessage: (SmallInteger divide)
        backupIs0Bytes, Q = 0bb, GOTO[notYetInvented], c1, bytecode[0bb];
    sendArithmeticMessage: (SmallInteger mod)
        backupIs0Bytes, Q = 0oa, GOTO[notYetInvented], c1, bytecode[0ba];
    sendArithmeticMessage: (SmallInteger makePoint)
        otlow = classPointPointer, uclassToInstance = otlow, c1, bytecode[0bb];
        tempHigh = viaPrimitiveMakePoint, c2;
        tempLow = 2 (number of fields in a Point), backupIs0Bytes, c1;
        Noop, c2;
        CALL[createInstanceWithPointers], c3;
        tempHigh = uNewObjectHigh, c1, at[viaPrimitiveMakePoint, 10, createInstance-return];
        tempLow = uNewObjectLow, L1 = makePoint, c2;
        tempLow = tempLow + yFieldOffsetInPoint, CALL[getSmashTos], c3;
    MAR = [tempHigh, tempLow + 0],
    MDR = temp3Low, otlow = temp3Low {save y field for ref!},
    temp3Low = temp3Low - offsetFromFieldToYField, CALL[getTos], c2;
    MAR = [tempHigh, tempLow + 0],
    MDR = temp3Low, L2 = upY, c1, at[makePoint, 10, getSmashTos-return];
    CALL[primRefi], (up y field) c2;
    otlow = temp3Low, L2 = upX, CALL[primRefi] {up x field},
    temp3Low = uNewObject, GOTO[pushTemp3LowAndDispatch], c3, at[upY, 10, primRefi-return];
    temp3Low = uNewObject, GOTO[pushTemp3LowAndDispatch], c3, at[upX, 10, primRefi-return];

    primRefi:
        Noop, c1;
        L1 = primitiveRefi, c2;
        [] = otlow LNot0, XDisp, CALL[refi], c3;
    L2Disp, c1, at[primitiveRefi, 10, refiReturn],
    REI[primRefi-return], c2;

    getTos:
        (return the tos value in temp3Low without changing stackPointer or nilling tos)
        MAR = [stackHigh, stackLow + 0], c1;
        L2Disp, c2;
        temp3Low = MDR, RET[getTos-return], c3;

    getSmashTos:
        (return the tos value in temp3Low, replacing tos with nil, and decrementing the stack pointer)
        MAR = [stackHigh, stackLow + 0], c1;
        MDR = nilPointer, c2;
Send.mc 1-Aug-84 18:34:47 PDT

```plaintext
temp3Low = MO,
Noop,
L1Disp,
stackLow = stackLow - 1, RET[getSmashTop-return],

sendArithmeticMessage12: (SmallInteger bitShift)
MAR = [stackHigh, stackLow = 0] (start read of argument),
L1 = smallBitShift,
stackLow = stackLow - 1 (point at receiver),
CALL[primitiveNeeds2SmallIntegers], backupIs0Bytes,
c1, bytecode[0bc];
c2;
c3;
c4;
c5;
c6;

bitShift:
[·] = temp3Low, NegBr,
[·] = temp2Low, ZeroBr, BRANCH [left, right],
c2, at [smallBitShift, 10, arithmeticPrimitives];
c3;

left:
Q = OD - temp3Low, CarryBr, BRANCH [§, returnZeroLeft],
BRANCH [failLeftShift, §],
[·] = temp3Low, ZeroBr,
c1;
c2;
c3;

leftLoop:
BRANCH [§, leftShiftDone],
 temp2Low = temp2Low + temp3Low, PgCr0Disp,
 temp3Low = temp3Low - 1, ZeroBr, BRANCH [leftLoop, leftOverflow, 2],
c1;
c2;
c3;
c4;
c5;

leftShiftDone:
 temp3Low = temp2Low, GOTO [pushArithmeticResult],
c2;

returnZeroLeft:
 temp3Low = temp2Low, CANCELBr [pushArithmeticResult, 1],
c2;

right:
Q = -0F, BRANCH [§, returnZeroRight],
[·] = temp3Low, NegBr,
 temp3Low = temp3Low or Q, ZeroBr, BRANCH [posRightLoop, negRightLoop],
c1;
c2;
c3;

posRightLoop:
BRANCH [§, posRightShiftDone],
 temp2Low = RShift1 temp3Low, SE = 0,
 temp3Low = temp3Low + 1, ZeroBr, GOTO [posRightLoop],
c1;
c2;
c3;
c4;

posRightShiftDone:
 temp3Low = temp2Low, GOTO [pushArithmeticResult],
c2;

negRightLoop:
BRANCH [§, negRightShiftDone],
 temp2Low = RShift1 temp3Low, SE = 1,
 temp3Low = temp3Low + 1, ZeroBr, GOTO [negRightLoop],
c1;
c2;
c3;

negRightShiftDone:
 temp3Low = temp2Low, GOTO [pushArithmeticResult],
c2;

returnZeroRight:
 temp3Low = temp2Low, CANCELBr [pushArithmeticResult, 1],
c2;

failLeftShift:
Noop,
c3;

leftOverflow:
CANCELBr [§, 1],
GOTO [arithmeticPrimitiveFailedC3],
c1;
c2;

sendArithmeticMessage13: (SmallInteger div)
backupIs0Bytes, 0 = Odb,
GOTO[notYetInvented],
c1, bytecode[0bd];

sendArithmeticMessage14: (SmallInteger bitAnd)
MAR = [stackHigh, stackLow = 0] (start read of argument), L1 = smallBitAnd, c1, bytecode[0be];
stackLow = stackLow - 1 (point at receiver), CALL[primitiveNeeds2SmallIntegers], backupIs0Bytes, c2;
temp3Low = temp2Low and temp3Low, GOTO[pushArithmeticResult],
c2, at [smallBitAnd, 10, arithmeticPrimitives];

sendArithmeticMessage16: (SmallInteger bitOr)
MAR = [stackHigh, stackLow = 0] (start read of argument), L1 = smallBitOr, c1, bytecode[0bf];
stackLow = stackLow - 1 (point at receiver), CALL[primitiveNeeds2SmallIntegers], backupIs0Bytes, c2;
temp3Low = temp2Low or temp3Low, GOTO[pushArithmeticResult],
c2, at [smallBitOr, 10, arithmeticPrimitives];

primitiveNeeds2SmallIntegers:
{upon entry, stackLow points at receiver, there is a pending read on the argument, and L1 contains the return linkage. At
Send.mc 1-Aug-84 18:34:47 PDT 4

```
exit, stackLow still points at the receiver. temp3Low contains the raw (hardware-oriented) argument, temp2Low contains the raw receiver.

temp3Low = MD (get argument). LOOPHELLO(undo), XDisp (to test for SmallInteger). c3;
MAR = [stackHigh, stackLow = 0]. BRANCH[arithmeticPrimitiveFailed, $, 0e], c1; }

[ ] = temp3Low LRot0, XDisp (determine sign of argument). c2;
temp2Low = MD, BRANCH [$, argNegative, 2], XDisp (so we can test receiver for being a SmallInteger). c3;
temp3Low = RShift temp3Low, SE = 0, (make a useful number) BRANCH[receiverNotSmall, adjustReceiver, 0e], c1;
argNegative:
temp3Low = RShift temp3Low, SE = 1, (make a useful negative number) BRANCH[receiverNotSmall, adjustReceiver, 0e], c1;
adjustReceiver:
[ ] = temp3Low LRot0, XDisp (determine sign of receiver). c2;
BRANCH[$, receiverNegative, 2], LDisp. c3;
temp2Low = RShift temp2Low, SE = 0, RET[arithmeticPrimitives]. c1;
receiverNegative:
temp2Low = RShift temp2Low, SE = 1, RET[arithmeticPrimitives]. c1;

receiverNotSmall:
GOTO[arithmeticPrimitiveFailedC3], c2;

pushArithmeticResult:
{upon entry temp3Low is the result of the primitive arithmetic operation, stackLow is pointing at the receiver. We make temp3Low a SmallInteger, and test for overflow to see if all is ok. If not, fail the primitive}
temp3Low = temp3Low RShift0, SE = 0, RBr. c3;
temp3Low = temp3Low LRot0, BRANCH [arithmeticPrimitiveFailedC1, $]. c3;
MAR = [stackHigh, stackLow = 0] (start writing resultant SmallInteger), c1;
BRANCH[$, largeIntegerTest, 2]. c1;
MDR = temp3Low, NextBytecode, c2;
DISPN[,bytecodes]. iplow + iplow + PC16, c3;
largeIntegerTest:
{If it's negative now, it's really positive}
temp3Low = temp3Low RShift0, SE = 0, RBr. c2;
temp3Low = temp3Low LRot0, BRANCH [arithmeticPrimitiveFailedC1, $]. c3;
ulargeIntegerValueLow = temp3Low. c1;
temp3Low = 2, (2 byte large integer) c2;
temp3High = twoByteLargeInteger, CALL [createLargePositiveInteger]. c3;
temp3Low = ulargeIntegerValueLow, createInstance-return]; c1. at [twoByteLargeInteger, 10, c2;
L1 = largeIntegerResult, CALL [otMap2]]. c3;
tempLow = temp1Low + objectHeaderSize, c1. at [largeIntegerResult, 10, otMap2-return];
L0, c2;
MAR = [tempHigh, tempLow = 0]. c3;
MDR = temp3Low, c1;
MAR = [stackHigh, stackLow = 0]. c2;
MDR = otLow, NextBytecode, c3;
DISPN[,bytecodes]. iplow + iplow + PC16, c2;
arithmeticPrimitiveFailedC1:
Noop. c1;
GOTO[arithmeticPrimitiveFailedC3]. c2;
arithmeticPrimitiveFailed:
CANCELBR[$, 0f], c2;
arithmeticPrimitiveFailedC3:
stackLow = stackLow + 1 (adjust to point at argument). LDisp (are we here because a directly dispatched primitive failed or because a locked up primitive failed). c3;
DISP[whichWayShouldIGo], c1;
Noop. [primitive found thru lookup failed, run method]. GOTO[activateNewMethod]. c2, at[1, 4, whichWayShouldIGo];
(directly dispatched primitive failed, so do lookup) Noop. c3;
LDisp. c2, at[0, 4, whichWayShouldIGo];
DISP[arithmeticPrimitive]. c3;
temp3Low = 0, GOTO[getSelectorAndArgs]. c1;
temp3Low = 2, GOTO[getSelectorAndArgs]. c2;
temp3Low = 4, GOTO[getSelectorAndArgs]. c3;
temp3Low = 6, GOTO[getSelectorAndArgs]. c2.
temp3Low = 8, GOTO[getSelectorAndArgs].
temp3Low = 0a, GOTO[getSelectorAndArgs].
temp3Low = 0c, GOTO[getSelectorAndArgs].
temp3Low = 0a, GOTO[getSelectorAndArgs].
temp3Low = 10, GOTO[getSelectorAndArgs].
temp3Low = 12, GOTO[getSelectorAndArgs].
temp3Low = 14, GOTO[getSelectorAndArgs].
temp3Low = 16, GOTO[getSelectorAndArgs].
temp3Low = 18, GOTO[getSelectorAndArgs].
temp3Low = 1a, GOTO[getSelectorAndArgs].
temp3Low = 1c, GOTO[getSelectorAndArgs].
temp3Low = 1e, GOTO[getSelectorAndArgs].

specialLookup;
  Noop.

getSelectorAndArgs:
  (upon entry temp3Low must be the (index*2)+1 into the special selectors object)
  otlow = specialSelectorsOop. CALL[otMap2]. L1 = gettingSpecialSelectors.
  c1: at[gettingSpecialSelectors, 10, arithPrimitive];
  c2: at[4, 10, arithPrimitive];
  c2: at[5, 10, arithPrimitive];
  c2: at[6, 10, arithPrimitive];
  c2: at[7, 10, arithPrimitive];
  c2: at[8, 10, arithPrimitive];
  c2: at[9, 10, arithPrimitive];
  c2: at[0a, 10, arithPrimitive];
  c2: at[0b, 10, arithPrimitive];
  c2: at[0c, 10, arithPrimitive];
  c2: at[0d, 10, arithPrimitive];
  c2: at[0e, 10, arithPrimitive];
  c2: at[0f, 10, arithPrimitive];
  Noop.
  c2:

  getMap2 = return;
  temp3Low = temp3Low + temp3Low.
  c1: at[gettingSpecialSelectors, 10, arithPrimitive];
  c2: at[4, 10, arithPrimitive];
  c2: at[5, 10, arithPrimitive];
  c2: at[6, 10, arithPrimitive];
  c2: at[7, 10, arithPrimitive];
  c2: at[8, 10, arithPrimitive];
  c2: at[9, 10, arithPrimitive];
  c2: at[0a, 10, arithPrimitive];
  c2: at[0b, 10, arithPrimitive];
  c2: at[0c, 10, arithPrimitive];
  c2: at[0d, 10, arithPrimitive];
  c2: at[0e, 10, arithPrimitive];
  c2: at[0f, 10, arithPrimitive];
  Noop.
  c2:

  temp3Low = temp3Low + objectHeaderSize.
  c1: at[gettingSpecialSelectors, 10, arithPrimitive];
  c2: at[4, 10, arithPrimitive];
  c2: at[5, 10, arithPrimitive];
  c2: at[6, 10, arithPrimitive];
  c2: at[7, 10, arithPrimitive];
  c2: at[8, 10, arithPrimitive];
  c2: at[9, 10, arithPrimitive];
  c2: at[0a, 10, arithPrimitive];
  c2: at[0b, 10, arithPrimitive];
  c2: at[0c, 10, arithPrimitive];
  c2: at[0d, 10, arithPrimitive];
  c2: at[0e, 10, arithPrimitive];
  c2: at[0f, 10, arithPrimitive];
  Noop.
  c2:

  MAR = [temp3High, temp3Low + 0],
  temp3Low = temp3Low + 1,
  0 = MD (the selector).
  c1: at[gettingSpecialSelectors, 10, arithPrimitive];
  c2: at[4, 10, arithPrimitive];
  c2: at[5, 10, arithPrimitive];
  c2: at[6, 10, arithPrimitive];
  c2: at[7, 10, arithPrimitive];
  c2: at[8, 10, arithPrimitive];
  c2: at[9, 10, arithPrimitive];
  c2: at[0a, 10, arithPrimitive];
  c2: at[0b, 10, arithPrimitive];
  c2: at[0c, 10, arithPrimitive];
  c2: at[0d, 10, arithPrimitive];
  c2: at[0e, 10, arithPrimitive];
  c2: at[0f, 10, arithPrimitive];
  Noop.
  c2:

  temp3Low = MD (the argument count, a SmallInteger).
  c1: at[gettingSpecialSelectors, 10, arithPrimitive];
  c2: at[4, 10, arithPrimitive];
  c2: at[5, 10, arithPrimitive];
  c2: at[6, 10, arithPrimitive];
  c2: at[7, 10, arithPrimitive];
  c2: at[8, 10, arithPrimitive];
  c2: at[9, 10, arithPrimitive];
  c2: at[0a, 10, arithPrimitive];
  c2: at[0b, 10, arithPrimitive];
  c2: at[0c, 10, arithPrimitive];
  c2: at[0d, 10, arithPrimitive];
  c2: at[0e, 10, arithPrimitive];
  c2: at[0f, 10, arithPrimitive];
  temp3Low = RShift temp3Low, SE = 0,
  argumentCount = temp3Low.
  c1: at[gettingSpecialSelectors, 10, arithPrimitive];
  c2: at[4, 10, arithPrimitive];
  c2: at[5, 10, arithPrimitive];
  c2: at[6, 10, arithPrimitive];
  c2: at[7, 10, arithPrimitive];
  c2: at[8, 10, arithPrimitive];
  c2: at[9, 10, arithPrimitive];
  c2: at[0a, 10, arithPrimitive];
  c2: at[0b, 10, arithPrimitive];
  c2: at[0c, 10, arithPrimitive];
  c2: at[0d, 10, arithPrimitive];
  c2: at[0e, 10, arithPrimitive];
  c2: at[0f, 10, arithPrimitive];
  stackLow = stackLow - temp3Low (point at receiver). GOTO[getReceiver].
  c1: at[gettingSpecialSelectors, 10, arithPrimitive];
  c2: at[4, 10, arithPrimitive];
  c2: at[5, 10, arithPrimitive];
  c2: at[6, 10, arithPrimitive];
  c2: at[7, 10, arithPrimitive];
  c2: at[8, 10, arithPrimitive];
  c2: at[9, 10, arithPrimitive];
  c2: at[0a, 10, arithPrimitive];
  c2: at[0b, 10, arithPrimitive];
  c2: at[0c, 10, arithPrimitive];
  c2: at[0d, 10, arithPrimitive];
  c2: at[0e, 10, arithPrimitive];
  c2: at[0f, 10, arithPrimitive];
  Noop.
  c2:

pushFalseInPrimitiveRelational:
  temp3Low = falsePointer, GOTO[pushRelationalResult].
  c1:

pushTrueInPrimitiveRelational:
  temp3Low = truePointer, GOTO[pushRelationalResult].
  c1:

pushRelationalResult:
  Noop.
  GOTO[pushTemp3LowAndDispatch].
  c2;

(Send Special Message)
  sendSpecialMessage0: (at:)
  temp3Low = 32'd, GOTO[specialLookup], backupIs0Bytes.
  c1: bytecode[0c0];
  c1: bytecode[0c1];
  c1: bytecode[0c2];
  c1: bytecode[0c3];
  temp3Low = 34'd, GOTO[specialLookup], backupIs0Bytes.
  sendSpecialMessage1: (at:put:)
  temp3Low = 34'd, GOTO[specialLookup], backupIs0Bytes.
  sendSpecialMessage2: (size)
  temp3Low = 36'd, GOTO[specialLookup], backupIs0Bytes.
  sendSpecialMessage3: (next)
  temp3Low = 38'd, GOTO[specialLookup], backupIs0Bytes.
sendSpecialMessage4: (nextPut:)  
tempSlow = 40'd. GOTO[specialLookup]. backupIs8Bytes,  
c1. bytecode(0c4);

sendSpecialMessage5: (atEnd)  
tempSlow = 42'd. GOTO[specialLookup]. backupIs8Bytes,  
c1. bytecode(0c6);

sendSpecialMessage6: ( ** )  
MAR = [stackHigh, stackLow + 0] (start read of argument).  
NDR = nilPointer (smash argument).  
tempSlow = MD (get argument).  
c1. bytecode(0c8);

stackLow = stackLow - 1,  
Noop,  
c1;

Noop,  
c2;

MAR = [stackHigh, stackLow + 0] (start read of receiver).  
Noop,  
tempSlow = MD,  
c1. bytecode(0c8);

Noop,  
c2;

[ ] = tempSlow - tempLow, ZeroBr.  
BRANCH[pushFalseInPrimitiveRelational, pushTrueInPrimitiveRelational].  
c1;

sendSpecialMessage7: (class)  
MAR = [stackHigh, stackLow + 0].  
L2 = primitiveClass (the getClass call proceeds directly to pushTempSlowAndDispatch), c2;  
otlow = MD, XDisp (to test for SmallInteger), CALL[getClass].  
c1. bytecode(0c7);

sendSpecialMessage8: (blockCopy:)  
MAR = [stackHigh, stackLow + 0] (read arg count), backupIs8Bytes,  
stackLow = stackLow - 1,  
tempSlow = MD (arg count as SmallInteger).  
c1. bytecode(0c8);

MAR = [stackHigh, stackLow + 0],  
0 = methodContextOfClassDop, L2 = directBlockCopy,  
otlow = MD (context oop), XDisp, CALL[getClass].  
c1;

[ ] = temp3Low xor 0, ZeroBr.  
utargumentCount = temp2Low, BRANCH[$, blockCopyOk].  
0 = blockContextOfClassDop,  
[c1;

[ ] = temp3Low xor 0, ZeroBr.  
BRANCH[$, blockCopyOkA].  
temp3Low = 48'd,  
c1. bytecode(0c8);

stackLow = stackLow + 1, GOTO[specialLookup].  
c1;

blockCopyOk:  
GOTO[primitiveBlockCopyViaDirectDispatch].  
c1;

blockCopyOkA:  
GOTO[primitiveBlockCopyViaDirectDispatch].  
c2;

sendSpecialMessage9: (value)  
MAR = [stackHigh, stackLow + 0]. backupIs8Bytes,  
utargumentCount = 0,  
valueGettingContext:  
otlow = MD, XDisp, L2 = directValue, CALL[getClass].  
c1. bytecode(0c9);

temp2Low = blockContextOfClassDop,  
[ ] = temp3Low xor temp2Low, ZeroBr.  
BRANCH[$, primitiveValueViaDirectDispatch], c3;

not a blockContext, do special lookup  
[ ] = utargumentCount, ZeroBr.  
BRANCH[wasValueColor, wasValue],  
c1;

wasValue:  
temp3Low = 60'd. GOTO[valueLookup].  
c1;

wasValueColor:  
temp3Low = 62'd, GOTO[valueLookup].  
c2;

c1. bytecode(0c2);

valueLookup:  
GOTO[specialLookup].  
c1;

sendSpecialMessage10: (value:)  
stackLow = stackLow - 1. backupIs8Bytes,  
temp3Low = 1,  
1 = argCount + temp3Low,  
MAR = [stackHigh, stackLow + 0].  
c1;

stackLow = stackLow + 1 (in case of failure), GOTO[valueGettingContext].  
c2;
Send mc 1-Aug-84 18:34:47 PDT

sendSpecialMessage1: (do:
  temp32Low = 64'd, GOTO[specialLookup], backupIs0Bytes,
  c1, bytecode[0cb];
sendSpecialMessage2: (new
  temp32Low = 66'd, GOTO[specialLookup], backupIs0Bytes,
  c1, bytecode[0cc];
sendSpecialMessage3: (new
  temp32Low = 68'd, GOTO[specialLookup], backupIs0Bytes,
  c1, bytecode[0cd];
sendSpecialMessage4: (x
  temp32Low = 60'd, GOTO[specialLookup], backupIs0Bytes,
  c1, bytecode[0ce];
sendSpecialMessage5: (y
  temp32Low = 62'd, GOTO[specialLookup], backupIs0Bytes,
  c1, bytecode[0cf];

(Send Literal Selector With No Arguments)

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField0, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0d0];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField1, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0d1];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField2, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0d2];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField3, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0d3];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField4, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0d4];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField5, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0d5];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField6, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0d6];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField7, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0d7];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField8, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0d8];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField9, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0d9];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField10, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0da];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField11, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0db];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField12, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0dc];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField13, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0dd];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField14, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0de];

sendLiteralSelectorWithNoArguments:
  temp32Low = literalField16, backupIs0Bytes, GOTO[getSelectorZeroArguments], c1, bytecode[0df];

getSelectorZeroArguments:
  temp32Low = 0, GOTO[getSelector], c2;

(Send Literal Selector With 1 Argument)

sendLiteralSelectorWith1Argument:
  temp32Low = literalField0, backupIs0Bytes, GOTO[getSelectorOneArgument], c1, bytecode[0e0];

sendLiteralSelectorWith1Argument:
  temp32Low = literalField1, backupIs0Bytes, GOTO[getSelectorOneArgument], c1, bytecode[0e1];

sendLiteralSelectorWith1Argument:
  temp32Low = literalField2, backupIs0Bytes, GOTO[getSelectorOneArgument], c1, bytecode[0e2];
sendLiteralSelector2WithArgument:
temp3low = literalField0, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0f0];

sendLiteralSelector2WithArgument:
temp3low = literalField1, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0f1];

sendLiteralSelector2WithArgument:
temp3low = literalField2, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0f2];

sendLiteralSelector2WithArgument:
temp3low = literalField3, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0f3];

sendLiteralSelector2WithArgument:
temp3low = literalField4, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0f4];

sendLiteralSelector2WithArgument:
temp3low = literalField5, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0f5];

sendLiteralSelector2WithArgument:
temp3low = literalField6, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0f6];

sendLiteralSelector2WithArgument:
temp3low = literalField7, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0f7];

sendLiteralSelector2WithArgument:
temp3low = literalField8, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0f8];

sendLiteralSelector2WithArgument:
temp3low = literalField9, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0f9];

sendLiteralSelector2WithArgument:
temp3low = literalField10, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0fa];

sendLiteralSelector2WithArgument:
temp3low = literalField11, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0fb];

sendLiteralSelector2WithArgument:
temp3low = literalField12, backupIs0Bytes, GOTO[getSelectorTwoArguments], cl, bytecode[0fc];
Send.mc 1-Aug-84 18:34:47 PDT

sendLiteralSelectorWith2Arguments:
    temp3Low + literalField10, backupIs0Bytes, GOTO[getSelectorTwoArguments], c1, bytecode[0Fb];
sendLiteralSelectorWith2Arguments:
    temp3Low + literalField14, backupIs0Bytes, GOTO[getSelectorTwoArguments], c1, bytecode[0Fb];
sendLiteralSelectorWith2Arguments:
    temp3Low + literalField16, backupIs0Bytes, GOTO[getSelectorTwoArguments], c1, bytecode[0Ff];

getSelectorTwoArguments:
    temp1low = 2, GOTO[getSelector], c2;

getSelector:
    {upon entry, temp3Low must contain the offset to the literal selector including the objectHeaderSize, temp1Low must contain the number of arguments}
    temp2Low = uCurrentMethodLow {get the current method address}, c3;
    temp2High = uCurrentMethodHigh, temp2Low = temp2Low + temp1Low {add in offset to literal selector}, c1;
    Noop, c2;
    stackLow = stackHigh - temp1Low {point at the new receiver}, c3;
    MAR = [temp2High, temp2Low = 0] {read the literal selector}, c1;
    uArgumentCount = temp1Low {may need this later, so save it}, c2;
    0 + MD {the selector}, c3;

getReceiver:
    MAR = [stackHigh, stackLow = 0], L3 = 0 {this is a lookup for execution}, c1;
    stackLow = stackHigh - temp1Low {point at the new receiver}, L2 = gettingNowReceiversClass, c2;
    oltlow = MD {the new receiver}, XDisp. CALL[getClass] c3;
{after the getClass call --}
    Q is the selector
    temp1High/Low are the base of the new receiver
    temp2High is the high address of the current CompiledMethod
    temp2Low is low address of literal in the current CompiledMethod
    temp3Low is the class of the receiver
    oltlow is still the new receiver
}

startMethodLookup:
    {upon entry, oltlow must be oop of new receiver, temp1High/Low is the base of new receiver, temp1Low must be oop of the class in which to start the search, Q must be selector}
    uNewReceivers + oltlow, c1, st[gettingNewReceiversClass, 10,
    getClass-return];
    uNewReceiversClass + temp3Low, c2;
    uSelector = 0, c3;
    uNewReceiversLow + temp1Low, c1;
    temp1Low = temp1High, c2;
    uNewReceiversHigh = temp1Low, c3;

startMethodLookupViaSuperSend:
    {first, we look in the method cache}
    temp1Low = Q and temp3Low {hash selector and class together}, c1;
    temp1Low = LShift1 temp1Low, SE = 0 {cache entries are 4 words }, c2;
    temp1Low = LShift1 temp1Low, SE = 0, c3;
    uHash = temp1Low {and save in case cache update is needed}, c1;
    temp1Low = uMethodCacheLow {get the method cache address}, c2;
    temp1High = uMethodCacheHigh, temp1low = temp1Low(hash index) + temp1Low, c1;
    uStartLookup = temp3Low, c2;
    Noop, c3;
{at this point, temp1Low is the hash index, temp1Low is abs address into cache}
    MAR = [temp1High, temp1low = 0] {read word of cache entry}, c1;
    temp1low = temp1Low + 1, c2;
    temp1Low = MD {the selector field from cache entry}, c3;
    [] = temp1Low - 0, ZeroFSR {test for selector match}, L1 = methodSearch, c5;
    BRANCH[notInCache, 3], c2;
    Noop, c3;

Send.mc 1-Aug-84 18:34:47 PDT
MAR = [tempHigh, tempLow + 0] (next word from cache entry), c1;
tempLow = tempLow + 1, c2;
tempLow = MD (the class field of cache entry), c3;
[] + tempLow - temp2Low, ZeroBr (test for class match), c1;
BRANCH[notInCache], c1;
Noop, c3;

(we have a cache hit)
MAR = [tempHigh, tempLow + 0] (cache hit, get method oop)
tempLow = tempLow + 1, c2;
outLow = MD, (and save it for oMapMapping)
MAR = [tempHigh, tempLow + 0] (finally, get the primitive flag)
unKnownMethodOop = outLow, l1 = gettingMethodBase, c2;
tempLow = MD, CALL[otMap] (to get the CompiledMethods address), c3;

(at the otMap call, temp2Low is the primitive flag
temp2Low is also the new receiver's class
tempLow is also the new receiver's class
outLow is the oop of the method

after the otMap call, tempHigh/Low are the compiledMethod's start address)

Q = tempHigh (start saving CompiledMethod's address). L2 = foundViaCache (the getNewMethodHeader call will return
directly to executeNewMethod). CALL[newMethodHeader], c1, at[getNewMethodHeader], 10, otMap-return;

(If we get to either notInCache or notInCache, tempLow must be the oop of the new receiver's class and Q must be the
selector we're looking for)

notInCache:
outLow = temp3Low (get oop of class), CALL[otMap] (to get address of class), c3;

notInCache:
outLow = temp3Low (get oop of class), CALL[otMap] (to get address of class), c3;
temp3Low = RShift1 Q (apply simple hash function to selector), SC=0, c1, at[messageSearch], 10, otMap-return;
unHashedSelector + temp3Low, c2;
Noop, c3;
unNewClassLow + temp2Low, tempLow = temp3High,
unNewClassHigh + tempLow (save class address in case we need to look in superclass), c3;
tempLow = unNewClassLow (not enough registers...),
tempLow = temp2Low + messageDictionaryOffset, c3;
Noop, c3;
MAR = [tempHigh, tempLow + 0] (read the messageDictionary oop),
L1 = messageDictionary,
outLow = MD, CALL[otMap] (and otMap it), c3;

tryThisDictionary:
temp3Low = temp2Low + SelectorStartPlusObjectHeaderSize,
unSelectedStartInDictionary = temp2Low (remember address of first oop in dictionary), c2;
tempLow = temp2Low + sizeFieldOffset, c3;
MAR = [tempHigh, tempLow + 0] (get the size field), c1;
tempLow = tempLow - sizeFieldOffset (low address of dictionary object), c2;
temp3Low = MD (dictionary length), c3;
tempLow = temp2Low,
temp2Low = temp2Low (dict low address) + temp3Low (dictionary length), c2;
tempLow = temp3Low - 1 (yielding address of last selector in dictionary), c3;
temp3Low = temp2Low (length) - SelectorStartPlusObjectHeaderSize - 1 (yielding the "mask"), c1;
temp3Low = temp2Low (mask) and unHashedSelector, c2;
temp3Low = temp2Low + SelectorStartPlusObjectHeaderSize, c3;

(and add the dictionary relative offset to the dictionary base)
temp2Low = temp2Low + temp3Low,
unWrap = 0 (probing in this dictionary has not yet wrapped), c1;
Noop, c3;

probe:
MAR = [tempHigh, tempLow + 0], (read oop from methodDictionary)
Noop, c3;
temp2Low = MD, c3;
[ ] + temp2Low xor nilPointer, ZeroBr,
[ ] + temp2Low xor 0 (selector), ZeroBr, BRANCH[$, nilFound],
BRANCH[checkForEndOfDictionary (might still be in this dictionary if we wrap), foundIt], c3;

checkForEndOfDictionary:
Noop, c1;
[ ] + temp2Low (where we are) = temp2Low (end of dictionary), ZeroBr, c2;
temp2Low = temp2Low + 1, BRANCH[probe], c3;

(need to check for wrapping)
[ ] = unWrap, ZeroBr,

unwrap = -tempLow xor tempLow [mark that we've wrapped]. BRANCH[trySuperclass, $], c2; tempLow = uSelectorsStartInDictionary, GOTO[probe].

\[n1!\]Found: (not in this dictionary, try superclass)
  tempHigh = uNewClassHigh, CANCELBR[getAndCheckSuperclass, 1], c3;
trySuperclass:
  tempHigh = uNewClassHigh,
getAndCheckSuperclass:
  tempLow = uNewClassLow,
  tempLow = tempLow + superclassOffset, c1;
  Noop, c2;
  MAR = [tempHigh, tempLow + 0] [get superclass oop], c3;
  Noop, c2;
  oLow = MD, c3;
  Noop, c1;
  [ ] = oLow xor n1Pointer, ZeroBr, L1 = saveSuperclass (if we oMap, we return to just before tryThisDictionary). BRANCH[CALL] oMap [not nil], doesNotUnderstand, c2;
  GOTO[bytecodeFailed], c1;

\[n1\!\]Found:
  {we have found the selector in the methodDictionary. we need to get the method oop from the parallel array, and then update the cache}
  tempLow = uSelectorsStartInDictionary, \(Q =\) tempLow [location of bit in methodDictionary] - tempLow, \(Q(\) yielding relative location of selector\()\), c2;
  tempLow = tempLow + 1 [point at methodArray field], c3;
  MAR = [tempHigh, tempLow + 0] [get the oop of the methodArray], c1;
  0 = 0 + objectHeaderSize, L1 = methodArray, c2;
  oLow = MD, CALL[oMap] {get address of method array}, c3;
  tempLow = tempLow + 0 [point at appropriate entry of methodArray], \(Q =\) tempHigh [point at methodArray], \(L2 =\) foundViaLookup, c1;
  L1 = methodBaseAfterLookup, c2;
  oLow = MD [oop of method], CALL[GetMap] {to get address of new compiledMethod}, c3;
  tempLow = RRot1 tempLow [start extracting flag bits from method header], c2;
  att[foundViaLookup, 10, getNewMethodHeader-return], c3;
  temp2Low = temp2Low LRot4, c1;
  temp2Low = temp2Low and 7, c1;
  [ ] = temp2Low xor 7, ZeroBr, \(\{\) could a primitive be specified\(\)}, c2;
  temp2Low = RShift1 temp2Low, c1;
  temp2Low = temp2Low and 0ff {the primitive specified}, c2;

\[\text{n0Primitive}\]
  temp2Low = 0, \(Q = 0\) no primitive, c1;
  Q = umask [more getting ready to update cache], c2;
  tempLow = tempLow + 0, GOTO[updateCache], c3;

\[\text{primitiveSpecified}\]
  {read the method header extension -- next to last literal}
  tempLow = uNewMethodLow, c1;
  tempLow = tempLow + tempLow [literal count], c2;
  tempLow = tempLow + objectHeaderSize, c3;
  tempLow = tempLow - 1, c1;
  0 = umask [more getting ready to update method cache], c2;
  tempLow = tempLow + 0 [and more getting ready to update method cache], c3;
  MAR = [tempHigh, tempLow + 0] [get method header extension], c1;
  tempLow = uNewMethodHeader, c2;
  temp2Low = MD, c3;
  temp2Low = RShift1 temp2Low, c1;
  temp2Low = temp2Low and 0ff {the primitive flag}, c2;
updateCache:
{upon entry, temp3High/low must be address of appropriate cache entry, uSelector, uNewReceiverClass must both be valid. temp2low must be primitive flag, and otlow must be oop of new method}
MAR = [temp3High, temp3low + 0] {selector field}, c1;
MDR = uSelector, c2;
temp3low = temp3low + 1, c3;
MAR = [temp3High, temp3low + 0], {class field}, c1;
MDR = uStartLookup, c2;
temp3low = temp3low + 1, c3;
MAR = [temp3High, temp3low + 0], {method field}, c1;
MDR = otlow, c2;
temp3low = temp3low + 1, c3;
MAR = [temp3High, temp3low + 0], {primitive flag}, c1;
MDR = temp3low, c2;
Woop, c3;
unNewMethodOop = otlow, GOTO[executeNewMethod], c1;

executeNewMethod:
{upon entry, temp2low is the primitiveFlag, temp1low is the methodHeader}
L3Disp [0 -> lookup for execution, 1 -> lookup for perform primitive], c2, at [foundViaCache, 10,
getNewMethodHeader-return];
RET[performOrExecute-return], c3;
executeNewMethodViaPrimitivePerform:
{test to see if this might be a primitive response, return self, or return of an instance variable}
Woop, c1, at [0, 10, performOrExecute-return];
[1 = temp2low, ZeroDB {any primitive specified?}, L3 = 1, c2;
YBus = temp1low LRot, XDisp, BRANCH[primitiveIndexNotZero {yes}, $], c3, c3;
DISP[flagTable, 1].
{flag = 0 - 4 and flag = 7}
GOTO[doActivate], c2, at [1, 10, flagTable], c3;
GOTO[doActivate], c2, at [3, 10, flagTable], c3;
GOTO[doActivate], c2, at [5, 10, flagTable], c3;
GOTO[doActivate], c2, at [7, 10, flagTable], c3;
GOTO[doActivate], c2, at [9, 10, flagTable], c3;
GOTO[doActivate], c2, at [11, 10, flagTable], c3;
{flag = 6, return self}
Next bytecode {easy result is already on stack, do nothing}, c2, at [0b, 10, flagTable], c3;
DISP[bytecodes], Iplow + 1plow + PIC16, c3;
{flag = 6, return instance var}
temp3low = uNewMethodHeader, c2, at [0d, 10, flagTable], c3;
temp3low = temp3low LRot8, c3;
temp3low = temp3low and if {get offset into object}, c1;
temp3High = uNewReceiverHigh, c2;
temp3low = uNewReceiverLow, c3;
temp1low = temp1low + temp3low {add in receiverField offset}, c1;
temp1low = temp1low + objectHeaderSize {account for object header}, c2;
Woop, c3;
MAR = [temp3High, temp1low + 0] {read the instance variable}, c1;
Woop, c2;
temp3low = MD, GOTO[pushTemp3LowAndDispatch], c3;
doActivate:
Woop, c3;
activateNewMethod:
{ok, we need to find a context to use for this send}
temp3High = uRumRecordHigh {get rum record address}, c1;
temp1low = uRumRecordLow, c2;
YBus = uNewMethodHeader, XDisp, c3;
BRANCH[needSmallContext, needLargeContext, 1], c1;
needLargeContext:
{In our schema, large contexts never become leaf (the reason being that they are profoundly unlikely to remain leaf).}
therefore don't grab for the leaf if a large context is needed)

temp3High = makeBigContext,

temp3Low = largeContextSizelessObjectHeader,
c2;
c3;

CALL[methodContextPlease],
c1;

temp2High = uNumRecordHigh,
c1, at[makeBigContext, 10,
createInstance-return];
temp2Low = uNumRecordLow,
c2;
Noop,
c3;

MAR = [temp2High, temp2Low + leafContextOpOffset],
c1;
CANCELBR[$, 2],
c2;
temp3Low = MD,
c3;
Noop,
c1;
[] = temp3Low xor uActiveContextOop, ZeroBr,
c2;
BRANCH[activeNotLeaf, activeIsLeaf],
c3;

activeIsLeaf:
MAR = [temp2High, temp2Low + leafContextOpOffset],
L1 = newLargeContextSmashLeaf, c1;
MDR = nilPointer, CANCELBR[$, 2], LOOPHOLE[wok],
c2;
ottow = MD, XDisp, CALL[refI],
c3;
ottow + uNewObject, GOTO[readyToTravel],
c1, at[newLargeContextSmashLeaf, 10, refdReturn];

needSmallContext:
{see if we can recycle the leaf context -- must not be the activeContext nor nil}

temp3Low = uActiveContextOop,
c2;
Noop,
c3;

MAR = [temp2High, temp2Low + leafContextOpOffset],
c1;
temp2High = makeSmallContext {return link if we instantiate}, CANCELBR[$, 2],
c2;
ottow = MD {top of leaf context},
c3;
[] = ottow - temp3Low, ZeroBr,
c1;
[] = ottow xor nilPointer, ZeroBr, BRANCH[$, needInstantiateSmallContext],
c2;
newContextOp = ottow, BRANCH[haveContextWillTravel {ok, we can recycle the leaf context}, instantiateSmallContext],
c3;

needInstantiateSmallContext:
CANCELBR[instantiateSmallContext, 1],
c3;

InstantiateSmallContext:

temp3Low = smallContextSizelessObjectHeader, CALL[methodContextPlease], c1;

temp2High = uNumRecordHigh,
c1, at[makeSmallContext, 10,
createInstance-return];
temp2Low = uNumRecordLow, L1 = upNewSmallContext,
c2;
[] = ottow LRot0, XDisp, CALL[refI],
c3;

MAR = [temp2High, temp2Low + leafContextOpOffset],
L1 = downOldLeafContext, c1, at[upNewSmallContext, 10, refdReturn];
MDR = ottow, CANCELBR[$, 2], LOOPHOLE[wok],
c2;
ottow = MD, XDisp, CALL[refI],
c3;
ottow + uNewObject, GOTO[readyToTravel],
c1, at[downOldLeafContext, 10, refdReturn];

activeNotLeaf:
Noop,
c1;

readyToTravel:
Noop,
c2;
newContextOp = ottow, GOTO[haveContextWillTravel],
c3;

haveContextWillTravel:

uMakeVolatileLinkage = makeNewContextVolatile, CALL[makeVolatile {for the new context}], c1:

{start filling in the fields of the new context}
temp3Low = temp3Low - senderFieldOffset,
c1, at[makeNewContextVolatile, 10,
movableVolatile-return];
{extract literal count to calculate initial pc}
temp3Low = uNewMethodHeader,
c2;
temp3Low = RShl1i [temp3Low and 7F], SE + 0,
c3;
MAR = [temp3High, temp3Low + 0] {write sender field},
c1;
MDR = uActiveContextOop,
c2;
temp1Low = uMakeVolatileLow,
c3;
temp3Low = temp3Low + literalStart,
c1;
temp3Low = LShl1i temp3Low, SE = 1, {LiteralCountOf[methodBase] + literalStart}*2 +1),
c2;
temp3Low = LShl1i temp3Low, SE = 1, {yields smallInteger},
c3;
temp3Low = temp3Low + InstructionPointerFieldOffset,
c1;
Noop,
c2;
Noop,
c3;
MAR + [tempHiHigh, tempLow + 0] {write pc field},
MAR + tempLow,
stackLow + uMakeVolatileLow,
{compute stackPointer from temporary count of method header}
tempLow + uUnMethodHeader,
tempLow + temp3Low Lr0, 
tempLow + LShift1 temp3Low and 1f, SE + 1, {yields smallInteger}
tempLow + tempLow + stackPointerFieldOffset,
then, 
then,
MAR + [tempHiHigh, tempLow + 0] {write stack pointer field},
MAR + temp3Low,
tempLow + uMakeVolatileLow,
tempLow + tempLow + methodFieldOffset,
{and start setting up for transferring the receiver & arguments from the sending context to the new context}
tempLow + uArgumentCount,
tempLow + stackLow, {source limit address},
MAR + [tempHiHigh, tempLow + 0] {write oop of method we are activating},
MAR + uNewMethodOop,
tempLow + uMakeVolatileLow,
{finish setting up for the receiver/arguments move}
stackLow + stackLow - temp3Low {low 16 bits of source address},
tempLow + tempLow + receiverFieldOffset {low 16 bits of destination address}, L1 + activatingMove,
{now, move the receiver and any arguments from active context to the new context, nulling out the corresponding words of the active context}
CALL [transferWords],
{now we need to adjust the stackPointer to reflect the absence of the transferred words}
tempLow + uArgumentCount,
stackLow + stackLow - tempLow - 1,

newActiveContext:
{well, by now it's clear that we are going to execute this send, so we may as well commit to it, and update the instruction pointer}
1pLow + 1pLow + PC16,
temp3Low + uCurrentMethodOop,
temp3Low - pLow - temp3Low, {word relative plus headerSize} 
temp3Low - temp3Low - objectHeaderSize {word relative},
temp3Low + LShift1 temp3Low, SE + pc16(byte offset into compiledMethod),
tempLow + tempLow + 1 {by the book...},
tempHiHigh + uActiveContextHi,
tempLow + uActiveContextLow,
tempLow + tempLow + instructionPointerFieldOffset,
temp3Low + LShift1 temp3Low, SE + 1, {yields smallInteger},
MAR + [tempHiHigh, tempLow + 0], {write current instruction pointer},
MAR + temp3Low,
tempLow + tempLow - instructionPointerFieldOffset {again point at base of context},
stackLow + stackLow - tempLow {relative stack offset},
stackLow + stackLow - stackPointerAdjustmentFactor,
stackLow + LShift1 stackLow, SE + 1 {yields smallInteger},
tempLow + tempLow + stackPointerFieldOffset,
1oLow + uActiveContextOop,
noop,
MAR + [tempHiHigh, tempLow + 0] {write current stack pointer}, L1 + changingActiveContext, 
MAR + stackLow, L0 + newContext,
{} + oLow Lr0, XDisp, CALL [refd] {decrease refs to activeContext},
CALL [fetchContextRegisters0FA1AlreadyVolatileContext],
GOTO [fixupInstructionPointer],
fetchContextRegisters-returns;

fetchContextRegistersAndMakeContextVolatile:
{upon entry, uNewContextOop must be the oop of the new context. L0 is the return linkage register}
temp2Low - fetchingContextRegisters,
oLow + uNewContextOop,
unMakeVolatileLinkage = temp2Low, CALL [makeVolatile] {the new context},
fetchContextRegisters0FA1AlreadyVolatileContext:
{upon entry, uNewContextOop must be the oop of the new context and that context must have already been made volatile.
thus leaving uMakeVolatileHigh/Low set up. L0 is the return linkage register

Noop,

stackHigh = uMakeVolatileHigh (get base of new context),
makeVolatile-return);
stackLow = uMakeVolatileLow, L1 = changingActiveContext,
ويلow = uNewContextDop, CALL[refi],
temp2High = uRunRecordHigh,
temp2Low = uRunRecordLow,
templow = uMakeVolatileLow,
MAR = [temp2High, temp2Low + activeContextDopOffset] (write new active context oop), c1;
activeContextDop = MAR = oillow, CANCELBL[4, 2], LOOPHOLE[wok], c2;
Q = tempHigh = uMakeVolatileHigh,

tempLow = tempLow + methodFieldOffset,
activeContextHigh = 0,
activeContextLow = stackLow,

(see if this is a method or block context, odd method field implies blockcontext)
MAR = [tempHigh, tempLow = 0], c1;
tempLow = tempLow - methodFieldOffset (again point at base of object), c2;
ipLow = MD, XDisp, c3;
IBPr = 1 (start draining any buffered bytecodes), BRANCH[isMethodContext, isBlockContext, 0x], c1:

isMethodContext:
(for methodContexts, the home is the active context)
Noop,
GOTO [saveHomeContextStuff],

isBlockContext:
(we need to know the blockContext's home. Get it and make it volatile. As a side effect of volatileization, we get the base of the home context)
tempLow = tempLow + homeFieldOffset,
Noop,
MAR = [tempHigh, tempLow + 0], c1;
tempLow = home,
otlow = MD (the block context's home), c3;

nameMakeVolatileLinkage = tempLow, CALL[makeVolatile],
tempLow = tempLow + methodFieldOffset,
Noop,
Noop,

MAR = [tempHigh, tempLow = 0] (read oop of method for homeContext) c1;
tempLow = tempLow - methodFieldOffset (again point at base of home context), c2;
ipLow = MD (oop of method), c3;

saveHomeContextStuff:

homeHigh = uMakeVolatileHigh,
homeLow = tempLow,
Q = tempLow,

uHomeLow = 0,
temp2High = uRunRecordHigh,
temp2Low = uRunRecordLow,
MAR = [temp2High, temp2Low + currentMethodDopOffset], c1;
MDR = ipLow (record the new context oop), CANCELER[4, 2], LOOPHOLE[wok], c2;
Yoos = ib (fetch draining buffered bytecodes), c3;
MAR = [temp2High, temp2Low + homeContextDopOffset], c1;
MDR = ipLow (record the new home oop), CANCELER[4, 2], LOOPHOLE[wok], c2;
tempLow = tempLow + receiverFieldOffset,
MAR = [tempHigh, tempLow = 0] (read receiver field), c1;
tempLow = homeLow,
otlow = MD (receiver oop), c3;

MAR = [temp2High, temp2Low + receiverDopOffset] (write receiver field in run record), c1;
MDR = iplow, CANCELER[4, 4], LOOPHOLE[wok],
[] = otlow LRet, XDisp, c3;
uReceiverOp + otLow, BRANCH[newReceiverIsDop, newReceiverIsSmall, 0x], c1;

newReceiverIsDop:
L1 = receiverDuringFetch,
CALL[otMap2],
Q = tempHigh,

uReceiverHigh = 0,

uReceiverLow = 0 or tempLow,
Noop,

newReceiverIsSmall:
(get relative stack pointer for this context)

stackLow = stackLow + stackPointerFieldOffset,
Noop,
MAR = [stackHigh, stackLow + 0],
stackLow = uActiveContextLow,
temp2Low = MD {relative stack pointer represented as smallInteger},
temp2Low = RShift1 temp2Low {de-smallIntegerize}, SE + 0,
Woop,
stackLow = stackLow + InstructionPointerFieldOffset,
MAR = [stackHigh, stackLow + 0] {get instruction pointer from context},
stackLow = uActiveContextLow,
temp2Low = MD {relative instr ptr represented as smallInteger},
stackLow = stackLow + temp2Low,
stackLow = stackLow + stackPointerAdjustmentFactor, L1 = methodDuringFetch,
otlow = ipLow {oop of method}, CALL[otMap2],
Q = temp1High,
uCurrentMethodHigh = Q,
uCurrentMethodLow = temp1Low,
temp1Low = RShift1 temp1Low, SE + 0 {de smallIntegerize},
temp1Low = temp1Low - 1 {by the book...},
1pHigh = Q LRot0,
1pLow = temp1Low + objectHeaderSize, LOD1sp,
RET[fetchContextRegisters-return],

methodContextPlease:
{upon entry, temp3High must be the return linkage for instance creation. temp3Low is the size context desired. create instance returns directly to methodContextPlease's caller}
otlow = methodContextFromClassOop,
uClassToInstantiate = otlow, CALL[createInstanceWithPointers],

d1: GOTO[bailout3]. at[3,10,getCurrentMethodHeader-return], c2;
d2: GOTO[bailout3]. at[4,10,getCurrentMethodHeader-return], c2;
d3: GOTO[bailout3]. at[5,10,getCurrentMethodHeader-return], c2;
d4: GOTO[bailout3]. at[6,10,getCurrentMethodHeader-return], c2;
d5: GOTO[bailout3]. at[7,10,getCurrentMethodHeader-return], c2;
d6: GOTO[bailout3]. at[8,10,getCurrentMethodHeader-return], c2;
d7: GOTO[bailout3]. at[9,10,getCurrentMethodHeader-return], c2;
d8: GOTO[bailout3]. at[0A,10,getCurrentMethodHeader-return], c2;
d9: GOTO[bailout3]. at[0B,10,getCurrentMethodHeader-return], c2;
d10: GOTO[bailout3]. at[0C,10,getCurrentMethodHeader-return], c2;
d11: GOTO[bailout3]. at[0D,10,getCurrentMethodHeader-return], c2;
d12: GOTO[bailout3]. at[0E,10,getCurrentMethodHeader-return], c2;
d13: GOTO[bailout3]. at[0F,10,getCurrentMethodHeader-return], c2;

d16: GOTO[bailout2]. at[1,10,getCurrentMethodHeader-return], c1;
d17: GOTO[bailout2]. at[2,10,getCurrentMethodHeader-return], c1;
d18: GOTO[bailout2]. at[3,10,getCurrentMethodHeader-return], c1;
d19: GOTO[bailout2]. at[4,10,getCurrentMethodHeader-return], c1;
d20: GOTO[bailout2]. at[5,10,getCurrentMethodHeader-return], c1;
d21: GOTO[bailout2]. at[6,10,getCurrentMethodHeader-return], c1;
d22: GOTO[bailout2]. at[7,10,getCurrentMethodHeader-return], c1;
d23: GOTO[bailout2]. at[8,10,getCurrentMethodHeader-return], c1;
d24: GOTO[bailout2]. at[9,10,getCurrentMethodHeader-return], c1;
d25: GOTO[bailout2]. at[0A,10,getCurrentMethodHeader-return], c1;
d26: GOTO[bailout2]. at[0B,10,getCurrentMethodHeader-return], c1;
d27: GOTO[bailout2]. at[0C,10,getCurrentMethodHeader-return], c1;
d28: GOTO[bailout2]. at[0D,10,getCurrentMethodHeader-return], c1;
d29: GOTO[bailout2]. at[0E,10,getCurrentMethodHeader-return], c1;

d42: GOTO[bailout2]. at[0a,10,ref1Return], c1;
d43: GOTO[bailout2]. at[0b,10,ref1Return], c1;
d44: GOTO[bailout2]. at[0c,10,ref1Return], c1;

d55: GOTO[bailout2]. at[6,10,ref1Return], c1;
d56: GOTO[bailout2]. at[7,10,ref1Return], c1;
d57: GOTO[bailout2]. at[8,10,ref1Return], c1;
d58: GOTO[bailout2]. at[9,10,ref1Return], c1;

d91: GOTO[bailout2]. at[0d,10,ref1Return], c1;
d92: GOTO[bailout2]. at[0e,10,ref1Return], c1;
d93: GOTO[bailout2]. at[0f,10,ref1Return], c1;

d100: GOTO[bailout3]. at[smalMultiply,10,arithmeticPrimitives], c2;
d101: GOTO[bailout3]. at[smalDivide,10,arithmeticPrimitives], c2;
{d104: GOTO[bailout3]. at[smallDivShift,10,arithmeticPrimitives], c2; }
d106: GOTO[bailout3]. at[smalDiv,10,arithmeticPrimitives], c2;

d88: GOTO[bailout2]. at[0a,10,getClass-return], c1;
d89: GOTO[bailout2]. at[0b,10,getClass-return], c1;
d90: GOTO[bailout2]. at[0c,10,getClass-return], c1;
d91: GOTO[bailout2]. at[0d,10,getClass-return], c1;
d92: GOTO[bailout2]. at[0e,10,getClass-return], c1;
d93: GOTO[bailout2]. at[0f,10,getClass-return], c1;

d202: GOTO[bailout2]. at[3,10,addToZeroCountTableReturn], c1;
d203: GOTO[bailout2]. at[4,10,addToZeroCountTableReturn], c1;
d204: GOTO[bailout2]. at[5,10,addToZeroCountTableReturn], c1;
d205: GOTO[bailout2]. at[6,10,addToZeroCountTableReturn], c1;
d206: GOTO[bailout2]. at[7,10,addToZeroCountTableReturn], c1;
d207: GOTO[bailout2]. at[8,10,addToZeroCountTableReturn], c1;
d208: GOTO[bailout2]. at[9,10,addToZeroCountTableReturn], c1;
d115e: GOTO[bailout1]. at[0e, 10, getByteOrAddress-return]. c3;
d115f: GOTO[bailout1]. at[0f, 10, getByteOrAddress-return]. c3;

d1163: GOTO[bailout2]. at[3, 10, commonAt-return]. c1;
d1164: GOTO[bailout2]. at[4, 10, commonAt-return]. c1;
d1165: GOTO[bailout2]. at[5, 10, commonAt-return]. c1;
d1166: GOTO[bailout2]. at[6, 10, commonAt-return]. c1;
d1167: GOTO[bailout2]. at[7, 10, commonAt-return]. c1;
d1168: GOTO[bailout2]. at[8, 10, commonAt-return]. c1;
d1169: GOTO[bailout2]. at[9, 10, commonAt-return]. c1;
d116a: GOTO[bailout2]. at[0a, 10, commonAt-return]. c1;
d116b: GOTO[bailout2]. at[0b, 10, commonAt-return]. c1;
d116c: GOTO[bailout2]. at[0c, 10, commonAt-return]. c1;
d116d: GOTO[bailout2]. at[0d, 10, commonAt-return]. c1;
d116e: GOTO[bailout2]. at[0e, 10, commonAt-return]. c1;
d116f: GOTO[bailout2]. at[0f, 10, commonAt-return]. c1;

d1172: GOTO[bailout2]. at[2, 10, performOrExecute-return]. c1;
d1173: GOTO[bailout2]. at[3, 10, performOrExecute-return]. c1;
d1174: GOTO[bailout2]. at[4, 10, performOrExecute-return]. c1;
d1175: GOTO[bailout2]. at[5, 10, performOrExecute-return]. c1;
d1176: GOTO[bailout2]. at[6, 10, performOrExecute-return]. c1;
d1177: GOTO[bailout2]. at[7, 10, performOrExecute-return]. c1;
d1178: GOTO[bailout2]. at[8, 10, performOrExecute-return]. c1;
d1179: GOTO[bailout2]. at[9, 10, performOrExecute-return]. c1;
d117a: GOTO[bailout2]. at[0a, 10, performOrExecute-return]. c1;
d117b: GOTO[bailout2]. at[0b, 10, performOrExecute-return]. c1;
d117c: GOTO[bailout2]. at[0c, 10, performOrExecute-return]. c1;
d117d: GOTO[bailout2]. at[0d, 10, performOrExecute-return]. c1;
d117e: GOTO[bailout2]. at[0e, 10, performOrExecute-return]. c1;
d117f: GOTO[bailout2]. at[0f, 10, performOrExecute-return]. c1;

bailout2:
    Noop, c2;

bailout3:
    Noop, c3;

bailout1:
    temp2High = uRumRecordHigh,
    temp2Low = uRumRecordLow,
    Noop, c1;
    MAR = [temp2High, temp2Low + directveOffset],
    MDR = 2, CANCELBR [5, 2]. LOOPTHOLE [wok],
    GOTO[restoreMessState], c2;
    MAR = 0, CANCELBR [5, 2]. NOOP, c2;
    GOTO[restoreMessState], c3;
getSmalltalkState:
  O = urunRecordHigh (retrieve the Run Record address),
  tempHigh = 0 LReg0,
  tempLow = urunRecordLow,
  (get the method cache address)
  MAR = [tempHigh, tempLow + methodCacheOffset],
  CANCELBR[7, 2], Noop, Q = MD, c1;
  CANCELBR[7, 2], uMethodCacheLow = Q, O = MD, c2;
  MAR = [tempHigh, tempLow + methodCacheOffset],
  CANCELBR[7, 2], uMethodCacheHigh = Q, O = MD, c3;
  (get the Object Table address)
  MAR = [tempHigh, tempLow + objectTableOffset],
  CANCELBR[7, 2], uMethodCacheHigh = Q, O = MD, c4;
  MAR = [tempHigh, tempLow + objectTableOffset],
  CANCELBR[7, 2], uMethodCacheHigh = Q, oTable = MD, c5;
  (get the op of the active context, omap it, save op and base address)
  MAR = [tempHigh, tempLow + activeContextOopOffset],
  LI = gettingActiveContextDuringInterpreterSwap, c1;
  CANCELBR[7, 2], Low = NO, CALL[omap2], c2;
  O = tempHigh, c3.
  10, omap2-return;
  uActiveContextHigh = Q, O = tempLow, c4;
  uActiveContextLow = Q, O = oTable, c5;
  uActiveContextOp = Q, c6;
  (get current Stack Pointer)
  MAR = [tempHigh, tempLow + stackPointersOffset],
  CANCELBR[7, 2], Noop, c7;
  stackLow = MD, c8;
  MAR = [tempHigh, tempLow + stackPointersOffset],
  CANCELBR[7, 2], Noop, c9;
  stackHigh = MD, c10;
  (get Home Context Oop, omap it, save its address)
  MAR = [tempHigh, tempLow + homeContextOopOffset],
  LI = getSmalltalkState, c11;
  CANCELBR[7, 2], c12;
  oTable = NO, CALL[omap], c13;
  uHomeLow = tempLow, c14.
  O = tempHigh, c15.
  homeHigh = Q LReg0, c16;
  (get the op of current receiver, save it and if it is not a SmallInteger, omap it and save its address)
  MAR = [tempHigh, tempLow + receiverOopOffset],
  CANCELBR[7, 2], homeLow = uHomeLow, c17;
  oTable = MD, c18;
  uTimeToStabilize = 0, c19;
  uReceiverOop = oTable, YDisp, c20;
  BRANCH[isOop-getSmalltalkState, isSmall-getSmalltalkState, 0e], c21;
isOop-getSmalltalkState:
  LI = isOopGettingSmalltalkState, c22;
  Noop, c23;
  CALL[omap], c24;
  O = tempHigh, c25.
  omap-return;
  uReceiverHigh = Q, c26;
  O = tempLow, c27.
  CANCELBR[7, 2], c28;
  (get the op of the current method, omap it, set up the machine’s instruction pointer registers)
  MAR = [tempHigh, tempLow + currentMethodOopOffset],
  LI = isSmallGettingSmalltalkState, c29;
  CANCELBR[7, 2], uCurrentMethodLow = 0, c30;
  oTable = MD, CALL[omap], c31;
  O = tempHigh, c32.
  omap-return;
  ipHigh = Q LReg0, c33;
  ipLow = tempLow + objectHeaderSize, c34;
  uCurrentMethodHigh = Q, c35;
  CANCELBR[7, 2],
fixupInstructionPointer:
(Upon entry, temp3low must contain the number of bytes by which to adjust the instruction pointer. iplow must be the base of the current compiled method. Either iplow must be bumped by objectHeaderSize or temp3low must account for the object header by being overstated by the amount objectHeaderSize*2)

temp2low = RShift1 temp3low (get word offset), SE = 0, X2tempDisp (see what state pc16 is--we want it zero). c1;
iplow = iplow + temp2low (add in word offset), BRANCH[flip, noFlip, 0]. c2;

flip:
Cin = pc16 (make it zero), GOTO[pc16isZeroNow]. c3;
noFlip:
GOTO[pc16isZeroNow]. c3;

pc16isZeroNow:
MAR = [iphigh, iplow > 0] (read a word of bytecodes).
P = temp3low (determine desired state of pc16). YDisp;
IB = MD (load up instruction buffer), BRANCH[leaveItBe, makeIt1, 0]. c3;
makeIt1:
Cin = pc16, IBptr = 1, GOTO[offToSeeTheWizard]. c1;
leaveItBe:
GOTO[offToSeeTheWizard]. c1;
offToSeeTheWizard:
(because the saving of the Mesa state left the Instruction Buffer empty and we then put in one or two bytes, the following IBDisp will cause a trap to the refill code at stNotEmpty which will refill the Instruction Buffer and execute another IBDisp that will take us to the interpreter for the current bytecode)

NextBytecode,
DISPN[bytecodes]. c2;

saveSmalltalkState:

{(we come here when Run finds an unpleasant bytecode, or when a Mesa interrupt has been set. Upon entry temp1low should be: 1 for a notYetInvented bytecode, 0 for a Mesa interrupt has occurred, and 2 for bytecode failure)

Q = urunRecordHigh (retrieve the Run Record address),
temp3High = Q  LRot8;
temp3Low = urunRecordLow,

(write the current stack pointer)
MAR = [temp3High, temp3Low + stackPointerLowOffset],
CANCELBR[$, 2], LOOPHOLE[wok], HDR = stackLow,
Noop,
MAR = [temp3High, temp3Low + stackPointerLowOffset],
CANCELBR[$, 2], LOOPHOLE[wok], HDR = stackHigh,

(adjust the instruction pointer to make Molasses happy, then write the Instruction Pointer)
iplow = iplow - objectHeaderSize,
temp2low = uCurrentMethodLow,
iplow = iplow - temp2low,
iplow = LShift1 iplow, SE = pc16,
[] + temp3low, YDisp,
DISP[smalltalkState], L0Disp (in case of bytecode failure),
CANCELBR[write1p, 0f],
CANCELBR[write1p, 0f],
temp1low = 1 (tell Molasses to execute this bytecode), DISPZ[ipAdjustment] (bytecode failed -- need to fix up inst ptr). c3, at[0, 10, smalltalkState];
c3, at[1, 10, smalltalkState];}
GOTO[ipAdjusted].
ipLow = ipLow - 1, GOTO[ipAdjusted]
ipLow = ipLow - 2, GOTO[ipAdjusted]
ipLow = ipLow - 3, GOTO[ipAdjusted]

{todo --- save current method oop}
Q = 7a, GOTO[badBytecode].
Q = 7f, GOTO[badBytecode].
Q = 8a, GOTO[badBytecode].
Q = 8b, GOTO[badBytecode].
Q = 8c, GOTO[badBytecode].
Q = 8d, GOTO[badBytecode].
Q = 8e, GOTO[badBytecode].
Q = 8f, GOTO[badBytecode].

c1, bytecode[7a];
c1, bytecode[7f];
c1, bytecode[8a];
c1, bytecode[8b];
c1, bytecode[8c];
c1, bytecode[8d];
c1, bytecode[8e];
c1, bytecode[8f];

badBytecode:
GOTO[bailout3].

c2;