```
{
        14-Jan-84 14:16:32
}


        RumStart:

                Bank ← 0,                                                       c1;
                Noop,                                                           c2;
                GOTOABS[GermRumHookup]  {off to initialize the Mesa world},     c3;
```

```
{
        30-Mar-84 16:43:25
}


        getClass:
                {upon entry, otLow must contain the oop whose class is desired , there must be a pending XDisp to allow us to test
                SmallIntegerness, 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[classIsNotSmallInt, classIsSmallInt, 0e],                        c1;

          classIsSmallInt:
                temp3Low ← classSmallIntegerPointer, L2Disp,                            c2;
                RET[getClass-return],                                                   c3;

          classIsNotSmallInt:
                L1 ← gettingClass,                                                      c2;
                CALL[otMap],                                                            c3;

        getClassAlreadyHaveBase:
                temp1Low ← temp1Low + classFieldOffset,                                 c1, at[gettingClass, 10, otMap-return];
                Noop,                                                                   c2;
                Noop,                                                                   c3;

                MAR ← [temp1High, temp1Low + 0] {read the class field},                 c1;
                temp1Low ← temp1Low - classFieldOffset, L2Disp,                         c2;
                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, temp1Low and uNewMethodHeader are both the new method header and uNewMethodHigh/Low are correct}

                uNewMethodLow ← temp1Low,                                               c2;
                temp1Low ← temp1Low + objectHeaderSize,                                 c3;

                MAR ← [temp1High, temp1Low + 0], {start read of method header}          c1;
                uNewMethodHigh ← Q {finish saving CompiledMethod start address},        c2;
                temp1Low ← MD {method header}, L2Disp,                                  c3;

                uNewMethodHeader ← temp1Low, 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 temp1Low}
                MAR ← [temp2High, temp2Low + 0],                                        c1;
                L1Disp,                                                                 c2;
                temp1Low ← MD, RET[getDeltaWord-return],                                c3;




        returnTopOfStack:
                {upon entry, there must be a pending ZeroBr [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:
                MDR ← nilPointer,                                                       c2;
                otLow ← MD,                                                             c3;

                stackLow ← stackLow - 1,                                                c1;
                L1Disp,                                                                 c2;
```

```
        RET[returnTopOfStack-return],                                   c3;

dontSmashTOS:
        L1Disp,                                                         c2;
        otLow ← MD, RET[returnTopOfStack-return],                       c3;


pushTemp3LowAndDispatch:
        MAR ← [stackHigh, stackLow + 0],                                c1, at[primitiveClass, 10, getClass-return];
        MDR ← temp3Low, NextBytecode,                                   c2;
        DISPNI[bytecodes], ipLow ← ipLow + PC16,                        c3;


notYetInvented:
        Noop,                                                           c2;
        temp1Low ← 1 {mark notYetInvented}, GOTO[saveSmalltalkState],   c3;


bytecodeFailed:
        Noop,                                                           c2;
        temp1Low ← 2 {mark bytecode failure}, GOTO[saveSmalltalkState], c3;


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 ← temp2Low + 1 {adjust limit to one past the last word},   c1;

transferWordsCheck:
        [] ← stackLow - temp2Low, ZeroBr,                               c2;
        BRANCH[$, returnFromTransferWords],                             c3;

        MAR ← [stackHigh, stackLow + 0] {read and smash receiver or argument}, CANCELBR[$, 0f], c1;
        MDR ← nilPointer,                                               c2;
        Q ← MD,                                                         c3;

        MAR ← [temp1High, temp1Low + 0] {put into new context},         c1;
        MDR ← Q,                                                        c2;
        temp1Low ← temp1Low + 1 {up destination index},                 c3;

        stackLow ← stackLow + 1, GOTO[transferWordsCheck],              c1;

returnFromTransferWords:
        stackLow ← stackLow - 1 {the last word actually moved},         c1;
        temp2Low ← temp2Low - 1, L1Disp,                                c2;
        RET[transferWords-return],                                      c3;
```

```
{
 2-Aug-84 11:30:06
}

{
Make all free chunk lists empty by storing notAnObject (8001) at the head of each list.
}
timeToCompress:
        rumHigh ← uRumRecordHigh,                                       c1;     {point to first free chunk list}
        rumLow ← uRumRecordLow,                                         c2;
        rumLow ← rumLow + freeListsOffset,                              c3;

        temp3Low ← largestFreeChunkSize,                                c1;     {set list counter}
        temp2Low ← 3,                                                   c2;     {create list terminator = 8001}
        temp2Low ← temp2Low RRot1,                                      c3;

emptyFreeChunkLoop:
        MAR ← [rumHigh, rumLow + 0],                                    c1;     {mark each list}
        MDR ← temp2Low, rumLow ← rumLow + 1,                            c2;
        Noop,                                                           c3;

        Noop,                                                           c1;
        temp3Low ← temp3Low - 1, NegBr,                                 c2;
        BRANCH [emptyFreeChunkLoop, $],                                 c3;


{
Make each 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:     Fail.
        10:     Fail.
        11:     Add oop to free oop list.  If refct < 0 (free object), make link odd (class field ← 1).
}

reverseOT:
        otLow ← 2,                                                      c1;     {start at bottom of OT}
        Noop,                                                           c2;
        rumLow ← uRumRecordLow,                                         c3;

        MAR ← [rumHigh, rumLow + objectTableHighOffset],                c1;
        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;     {otmap oop}
        CANCELBR [$, 2],                                                c2;
        Q ← objectHigh ← MD, XwdDisp,                                   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];
        objectLow ← MD,                                                 c3;

        objectLow ← objectLow + classFieldOffset,                       c1;
        Noop,                                                           c2;
        Noop,                                                           c3;

        MAR ← [objectHigh, objectLow + 0],                              c1;
        MDR ← otLow,                                                    c2;     {object ← oop}
        temp3Low ← MD, XDisp,                                           c3;

        MAR ← [otHigh, otLow + 0], BRANCH [$, impossibleClass, 0E],     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}
        [] ← Q, NegBr,                                                  c2, at [3, 4, purposeTable];
        objectLow ← MD, BRANCH [freeOop, $],                            c3;

        MAR ← [rumHigh, rumLow + 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}
        objectLow ← objectLow + classFieldOffset,                       c3;

        MAR ← [otHigh, otLow + 1],                                      c1;
        MDR ← 60, CANCELBR [$, 2], LOOPHOLE [wok],                      c2;     {fix flag field}
        Noop,                                                           c3;

        MAR ← [objectHigh, objectLow + 0],                              c1;
```

```
        MDR ← 1, GOTO [nextOop],                                        c2;     {mark object free}

nextOop:
        Noop,                                                           c3;

freeOop:
        Noop,                                                           c1;
        otLow ← otLow + 2, CarryBr,                                     c2;
        BRANCH [reverseOTloop, $],                                      c3;


{
Move all objects as low in object space as possible without straddling a bank boundary.
        otHigh ← OT base high (object memory limit)
        sourceHigh, sourceLow ← object source
        destHigh, destLow ← object destination
        rumHigh, rumLow ← rum record base
}

compress:
        {initialize the pointers}
        MAR ← [rumHigh, rumLow + objectMemoryHighOffset],               c1;
        CANCELBR [$, 2],                                                c2;
        sourceLow ← sourceHigh ← MD,                                    c3;

        MAR ← [rumHigh, rumLow + objectMemoryLowOffset],                c1;
        destHigh ← sourceLow LRot0, CANCELBR [$, 2],                    c2;
        sourceLow ← MD,                                                 c3;

        destLow ← sourceLow,                                            c1;


compressLoop:
        sourceLow ← sourceLow + classFieldOffset,                       c2;     {sourceLow ← ↑class field}
c12:
        Noop,                                                           c3;

        MAR ← [sourceHigh, sourceLow + 0],                              c1;
        sourceLow ← sourceLow - offsetFromSizeFieldToClassField,        c2;     {sourceLow ← ↑size field}
        otLow ← MD, XDisp,                                              c3;     {otLow ← free chunk mark or oop}

        MAR ← [sourceHigh, sourceLow + 0], BRANCH [$, skipFreeChunk, 0E],  c1;  {skip free object if class field odd}
        sourceLow ← sourceLow - sizeFieldOffset,                        c2;     {sourceLow ← ↑first field}
        size ← MD,                                                      c3;

        {will it fit at destination?}
        Q ← destLow + size, ZeroBr,                                     c1;
        Q ← destLow + size, CarryBr, BRANCH [$, perfectFit],            c2;
        temp3Low ← otLow, BRANCH [$, ndb2],                             c3;

        Q ← Q + objectHeaderSize, CarryBr,                              c1;
        BRANCH [unreverseOtEntry, newDestinationBank],                  c2;


perfectFit:
        sourceLow ← sourceLow + classFieldOffset, CANCELBR [uoe2, 1],   c3;


skipFreeChunk:
        sourceLow ← sourceLow - sizeFieldOffset,                        c2;     {sourceLow ← ↑first field}
        size ← MD,                                                      c3;

        [] ← size - objectHeaderSize, CarryBr,                          c1;
        BRANCH [impossibleSize1, $],                                    c2;
        sourceLow ← sourceLow + size, CarryBr, GOTO [newSourceBank],    c3;


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

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


        {put the remainder of the destination bank on the appropriate free list}
newDestinationBank:
        temp3Low ← otLow, GOTO [ndb2],                                  c3;

ndb2:
        temp2Low ← -destLow,                                            c1;     {temp2Low ← size of free chunk}
        L1 ← remainderFree,                                             c2;
        CALL [newFreeChunk],                                            c3;

        otLow ← rumLow,                                                 c1, at [remainderFree, 10, addToFreeChunkList-return];
        destLow ← destHigh,                                             c2;     {set new bank}
        destHigh ← destLow + 1 LRot0, LOOPHOLE [niblTiming],            c3;

        destLow ← 0,                                                    c1;
        rumLow ← uRumRecordLow, GOTO [unreverseOtEntry],                c2;


unreverseOtEntry:
        sourceLow ← sourceLow + classFieldOffset,                       c3;

uoe2:
        MAR ← [otHigh, otLow + 0],                                      c1;     {set offset}
        MDR ← destLow,                                                  c2;
```

```
        temp2Low ← MD,                                                  c3;

        MAR ← [sourceHigh, sourceLow + 0],                              c1;     {restore class}
        MDR ← temp2Low,                                                 c2;
        sourceLow ← sourceLow - classFieldOffset,                       c3;

        MAR ← [otHigh, otLow + 1],                                      c1;     {set bank}
        temp3Low ← ~OF, CANCELBR [$, 2],                                c2;
        temp2Low ← MD and temp3Low,                                     c3;

        [] ← size - objectHeaderSize, CarryBr,                          c1;
        Q ← destHigh, BRANCH [impossibleSize2, $],                      c2;
        Noop,                                                           c3;

        MAR ← [otHigh, otLow + 1],                                      c1;
        MDR ← temp2Low or Q, CANCELBR [$, 2], LOOPHOLE [wok],           c2;
        [] ← Q xor sourceHigh, ZeroBr,                                  c3;     {is move necessary?}

        [] ← destLow xor sourceLow, ZeroBr, BRANCH [moveObject1, $],    c1;
        BRANCH [$, noMove],                                             c2;
        GOTO [mol0],                                                    c3;


moveObject1:
        CANCELBR [$, 1],                                                c2;
        GOTO [mol0],                                                    c3;


noMove:
        sourceLow ← sourceLow + size, ZeroBr,                           c3;

        BRANCH [$, nm2],                                                c1;
        GOTO [nm4],                                                     c2;
nm2:
        destHigh ← Q + 1 LRot0, LOOPHOLE [niblTiming],                  c2;
nm4:
        destLow ← sourceLow, ZeroBr, GOTO [newSourceBank],              c3;


moveObjectLoop:
        Noop,                                                           c3;

mol0:
        MAR ← [sourceHigh, sourceLow + 0],                              c1;
        sourceLow ← sourceLow + 1,                                      c2;
        temp2Low ← MD,                                                  c3;

        MAR ← [destHigh, destLow + 0],                                  c1;
        MDR ← temp2Low, destLow ← destLow + 1, ZeroBr,                  c2;
        Q ← destHigh, BRANCH [mol2, $],                                 c3;

        destHigh ← Q + 1 LRot0, LOOPHOLE [niblTiming],                  c1;
        [] ← size - 1, ZeroBr,                                          c2;
        BRANCH [bankStraddle, $],                                       c3;

        size ← size - 1, ZeroBr, GOTO [mol4],                           c1;

mol2:
        size ← size - 1, ZeroBr,                                        c1;
mol4:
        BRANCH [moveObjectLoop, $],                                     c2;
        [] ← sourceLow, ZeroBr, GOTO [newSourceBank],                   c3;


newSourceBank:
        Q ← sourceHigh, BRANCH [compressLoop3, $],                      c1;
        sourceHigh ← Q ← Q + 1 LRot0, LOOPHOLE [niblTiming],            c2;
        Q ← otHigh xor Q, ZeroBr,                                       c3;

        BRANCH [compressLoop2, carveRemainingFreeSpace],                c1;


bankStraddle:
        GOTO [bailout2],                                                c1;


compressLoop2:
        sourceLow ← sourceLow + classFieldOffset, GOTO [c12],           c2;     {sourceLow ← †class field}

compressLoop3:
        sourceLow ← sourceLow + classFieldOffset, GOTO [c12],           c2;     {sourceLow ← †class field}


carveRemainingFreeSpace:
        temp2Low ← ~destLow,                                            c2;     {temp2Low ← remainder size - 1}
        Q ← temp2Low + objectHeaderSize, CarryBr,                       c3;     {will it fit in one chunk?}

        BRANCH [$, carveFreeBank],                                      c1;
        temp2Low ← temp2Low + 1, L1 ← carveFree1,                       c2;     {make remainder a free chunk}
        CALL [newFreeChunk],                                            c3;

        rumLow ← uRumRecordLow,                                         c1, at [carveFree1, 10, addToFreeChunkList-return];
        Q ← destHigh,                                                   c2;     {set new bank}
        destHigh ← Q ← Q + 1 LRot0, LOOPHOLE [niblTiming],              c3;

        Noop,                                                           c1;
```

```
        Q ← Q xor otHigh, ZeroBr,                                   c2;
        destLow ← 0, BRANCH [carveFreeBankLoop1, finis1],           c3;

carveFreeBank:
        temp2Low ← objectHeaderSize, L1 ← carveFree2, GOTO [cfbl4], c2;     {remainder size is too big for one chunk}

carveFreeBankLoop2:
        temp2Low ← objectHeaderSize, L1 ← carveFree2, GOTO [cfbl2], c1;

carveFreeBankLoop1:
        temp2Low ← objectHeaderSize, L1 ← carveFree2, GOTO [cfbl2], c1;
cfbl2:
        Noop,                                                       c2;
cfbl4:
        size ← temp2Low, CALL [newFreeChunk],                       c3;

        destLow ← destLow + size,                                   c1, at [carveFree2, 10, addToFreeChunkList-return];
        Noop,                                                       c2;
        Noop,                                                       c3;

        rumLow ← uRumRecordLow,                                     c1;
        temp2Low ← -destLow, L1 ← carveFree3,                       c2;
        CALL [newFreeChunk],                                        c3;

        rumLow ← uRumRecordLow,                                     c1, at [carveFree3, 10, addToFreeChunkList-return];
        Q ← destHigh,                                               c2;     {set new bank}
        destHigh ← Q ← Q + 1 LRot0, LOOPHOLE [nibITiming],          c3;

        Noop,                                                       c1;
        Q ← Q xor otHigh, ZeroBr,                                   c2;
        destLow ← 0, BRANCH [carveFreeBankLoop2, finis2],           c3;

finis1:
        Noop,                                                       c1;
finis3:
        Noop,                                                       c2;
        GOTO [oopsLeft],                                            c3;

finis2:
        GOTO [finis3],                                              c1;


{
Subroutine to create a new free chunk on the appropriate list.
        inputs: rum points to rum record
                dest points to free chunk base
                temp2Low is size of chunk
        smashes: Q, temp2, temp3, rumLow
        saves:  input temp3Low in rumLow
}
newFreeChunk:
        Noop,                                                       c1;
        Noop,                                                       c2;
        destLow ← destLow + sizeFieldOffset,                        c3;

        MAR ← [destHigh, destLow + 0],                              c1;     {set free chunk size}
        MDR ← temp2Low,                                             c2;
        destLow ← destLow - sizeFieldOffset,                        c3;

        MAR ← [rumHigh, rumLow + freePointersOopOffset],            c1;     {get free oop}
        CANCELBR [$, 2],                                            c2;
        otLow ← MD,                                                 c3;

        MAR ← [otHigh, otLow + 0],                                  c1;
        MDR ← destLow,                                              c2;     {set free chunk offset}
        temp2Low ← MD,                                              c3;     {get free oop link}

        MAR ← [otHigh, otLow + 1],                                  c1;
        MDR ← destHigh, CANCELBR [$, 2], LOOPHOLE [wok],            c2;     {set free chunk bank}
        Noop,                                                       c3;

        MAR ← [rumHigh, rumLow + freePointersOopOffset],            c1;     {fix list head}
        MDR ← temp2Low, CANCELBR [$, 2], LOOPHOLE [wok],            c2;
        rumLow ← temp3Low, GOTO [returnToPool],                     c3;     {add chunk to free list}


{
Count the oops on the free oop list.  Then count the objects on the free chunk lists.
}
oopsLeft:
        MAR ← [rumHigh, rumLow + freePointersOopOffset],            c1;
        stackLow ← 0, CANCELBR [$, 2],                              c2;
        otLow ← MD,                                                 c3;

oopCountLoop:
        [] ← otLow LRot0, XDisp,                                    c1;
        BRANCH [$, countFreeChunks, 0E],                            c2;     {terminator is odd}
        Noop,                                                       c3;

        MAR ← [otHigh, otLow + 0],                                  c1;
        stackLow ← stackLow + 1,                                    c2;
        otLow ← MD, GOTO [oopCountLoop],                            c3;

countFreeChunks:
        GOTO [wordsLeft],                                           c3;

{
```

```
Count all the free space on the free chunk lists.
}
wordsLeft:
        rumLow ← rumLow + freeListsOffset,                          c1;
        temp1High ← 0, temp1Low ← 0,                                c2;
        temp3Low ← largestFreeChunkSize,                            c3;

wordCountOuterLoop:
        MAR ← [rumHigh, rumLow + 0],                                c1;
        Noop,                                                       c2;
        otLow ← MD,                                                 c3;

wordCountInnerLoop:
        [] ← otLow LRot0, XDisp,                                    c1;
        Q ← temp1High, BRANCH [$, nextList, 0E],                    c2;     {list terminator is odd}
        stackLow ← stackLow + 1,                                    c3;

        MAR ← [otHigh, otLow + 0],                                  c1;     {otmap}
        Q ← Q + 1,                                                  c2;
        objectLow ← MD,                                             c3;

        MAR ← [otHigh, otLow + 1],                                  c1;
        objectLow ← objectLow + sizeFieldOffset, CANCELBR [$, 2],   c2;
        objectHigh ← MD,                                            c3;

        MAR ← [objectHigh, objectLow + 0],                          c1;
        objectLow ← objectLow + offsetFromSizeFieldToClassField,    c2;
        size ← MD,                                                  c3;

        temp1Low ← temp1Low + size, CarryBr,                        c1;
        BRANCH [$, wci12],                                          c2;
        GOTO [wci14],                                               c3;

wci12:
        temp1High ← Q LRot0,                                        c3;

wci14:
        MAR ← [objectHigh, objectLow + 0],                          c1;
        Noop,                                                       c2;
        otLow ← MD, GOTO [wordCountInnerLoop],                      c3;

nextList:
        Noop,                                                       c3;

        Noop,                                                       c1;
        temp3Low ← temp3Low - 1, NegBr,                             c2;
        rumLow ← rumLow + 1, BRANCH [wordCountOuterLoop, $],        c3;

setWordLevel:
        rumLow ← uRumRecordLow,                                     c1;
        Noop,                                                       c2;
        Noop,                                                       c3;

        MAR ← [rumHigh, rumLow + wordLevelLowOffset],               c1;
        MDR ← temp1Low, CANCELBR [$, 2], LOOPHOLE [wok],            c2;
        Q ← MD,                                                     c3;

        MAR ← [rumHigh, rumLow + wordLevelHighOffset],              c1;
        MDR ← temp1High, CANCELBR [$, 2], LOOPHOLE [wok],           c2;
        Q ← MD,                                                     c3;

setOopLevel:
        MAR ← [rumHigh, rumLow + oopLevelLowOffset],                c1;
        MDR ← stackLow, CANCELBR [$, 2], LOOPHOLE [wok],            c2;
        Q ← MD,                                                     c3;

        MAR ← [rumHigh, rumLow + oopLevelHighOffset],               c1;
        MDR ← 0, CANCELBR [$, 2], LOOPHOLE [wok],                   c2;
        Q ← MD, GOTO [restoreMesaState],                            c3;
```

```
{
        13-Mar-84 18:29:29
}



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


        RegDef[MesaStateL, U, 30];
        RegDef[MesaStateG, U, 45];
        RegDef[MesaStatePC, U, 50];
        RegDef[MesaStatePC16, U, 52];
        RegDef[MesaStateIBPtr, U, 53];
        RegDef[MesaStateIB, U, 54];
        {TOS (register 0) gets saved in the STK}

        RegDef[MesaStateRhMDS, U, 66];
        RegDef[MesaStateRhL, U, 33];
        RegDef[MesaStateRhG, U, 46];
        RegDef[MesaStateRhPC, U, 51];


{
        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[rhMDS,   RH,     0];
        RegDef[L,       R,      3];
        RegDef[rhL,     RH,     3];
        RegDef[G,       R,      4];
        RegDef[rhG,     RH,     4];
        RegDef[PC,      R,      5];
        RegDef[rhPC,    RH,     5];


{todo --- we can probably do better saving the RH registers}

        fromMesa:

                STK ← TOS, push,                                        c1, at[doBytecodes];
                MesaStateL ← L,                                         c2;
                MesaStateG ← G,                                         c3;

                MesaStatePC ← PC,                                       c1;
                PC ← LShift1 PC, SE ← pc16,                             c2;
                MesaStatePC16 ← PC,                                     c3;



    {now save the RH registers--somewhat messy since you can't write U reg directly from an RH register}
                Q ← rhMDS,                                              c1;
                MesaStateRhMDS ← Q,                                     c2;
                L ← rhL,                                                c3;

                MesaStateRhL ← L,                                       c1;
                G ← rhG,                                                c2;
                MesaStateRhG ← G,                                       c3;

                PC ← rhPC,                                              c1;
                MesaStateRhPC ← PC,                                     c2;
                PC ← ErrnIBnStkp,                                       c3;

                PC ← PC LRot12,                                         c1;
                PC ←  ~PC, {get it back to its non-negated state}       c2;
                MesaStateIBPtr ← PC, YDisp,                             c3;

                DISP4[drainIB,0c],                                      c1;

                GOTO[drained],   {state = empty, nothing to do}         c2, at[0c,10,drainIB];

                PC ← ib, {state = byte, grab it}                        c2, at[0d,10,drainIB];

                MesaStateIB ← PC,                                       c3;

                Noop,                                                   c1;
                GOTO[drained],                                          c2;

        IBspin:
                GOTO[IBspin], {state = full, cannot happen...},         c*, at[0e,10,drainIB];

                PC ← ib,  {state = word, get first byte}                c2, at[0f,10,drainIB];

                PC ← PC LRot8,                                          c3;

                L ← ib, {get second byte}                               c1;
```

```
            MesaStateIB ← PC or L,  GOTO[drained],                          c2;


    drained:
            Noop,                                                           c3;
            L0Disp,                                                         c1;
            DISP2[doBytecodesOrStablilization],                             c2;


            GOTO[getSmalltalkState],                                        c3,at[0, 4, doBytecodesOrStablilization];

            GOTO[timeToStabilize],                                          c3,at[1, 4, doBytecodesOrStablilization];

            GOTO[timeToCompress],                                           c3, at[2,4,doBytecodesOrStablilization];









    restoreMesaState:

{restore the RH registers}
            rhMDS ← MesaStateRhMDS,                                         c1;
            rhL ← MesaStateRhL,                                             c2;
            rhG ← MesaStateRhG,                                             c3;

            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,                                     c2;
            XC2npcDisp, BRANCH[pc16wasZero, pc16wasOne, 0e],                c3;
    pc16wasZero:
            BRANCH[$, wasZeroIsOk, 0e],                                     c1;

            Cin ← pc16, GOTO[okPc],                                         c2;
    wasZeroIsOk:
            GOTO[okPc],                                                     c2;


    pc16wasOne:
            BRANCH[wasOneIsOk, $, 0e],                                      c1;

            Cin ← pc16, GOTO [okPc],                                        c2;
    wasOneIsOk:
            GOTO[okPc],                                                     c2;


    okPc:  {now restore the IB state}

            IBPtr ← 1, {drain any smalltalk bytecodes},                     c3;

            PC ← ib,                                                        c1;
            Noop,                                                           c2;
            PC ← MesaStateIBPtr, XDisp,                                     c3;

            DISP4[restoreIB, 0c],                                           c1;

            GOTO[IBdone], {state was empty},                                c2,at[0c,10,restoreIB];

            PC ← MesaStateIB,                                               c2, at[0d,10,restoreIB];
            IB ← PC LRot0,                  {state was byte},               c3;
            IBPtr ← 1,                                                      c1;

            GOTO[IBdone],                                                   c2;

            IB ← MesaStateIB, {state was word}, GOTO[IBdone],               c2,at[0f,10,restoreIB];

    IBdone:

            PC ← MesaStatePC,                                               c3;

            L ← MesaStateL,                                                 c1;
            G ← MesaStateG,                                                 c2;
            Noop,                                                           c3;

    punt:

            Bank ← 0,                                                       c1;
            Noop,                                                           c2;
            GOTOABS[returnFromRumBank],                                     c3;
```

```
RumInitialization:
        uRumRecordHigh ← TOS,                                   c1, at[doInitialization];
        Q ← STK, pop,                                           c2;
        uRumRecordLow ← Q,                                      c3;

        Bank ← 0,                                               c1;
        Noop,                                                   c2;
        GOTOABS[returnFromRumBank],                             c3;



timeToStabilize:
        L0 ← mesaRequestedStabilization,                        c1;
        otHigh ← 0b,                                            c2;
        CALL[stabilize],                                        c3;

        Noop,                                                   c1, at[mesaRequestedStabilization, 10,
        stabilize-return];
        Noop,                                                   c2;
        GOTO[restoreMesaState],                                 c3;
```

```
{
        16-Oct-83 18:23:28
}


    {Jump}

        jump1:
                temp3Low ← 2, GOTO[jumpUnconditionally],                    c1, bytecode[90];
        jump2:
                temp3Low ← 3, GOTO[jumpUnconditionally],                    c1, bytecode[91];
        jump3:
                temp3Low ← 4, GOTO[jumpUnconditionally],                    c1, bytecode[92];
        jump4:
                temp3Low ← 5, GOTO[jumpUnconditionally],                    c1, bytecode[93];
        jump5:
                temp3Low ← 6, GOTO[jumpUnconditionally],                    c1, bytecode[94];
        jump6:
                temp3Low ← 7, GOTO[jumpUnconditionally],                    c1, bytecode[95];
        jump7:
                temp3Low ← 8, GOTO[jumpUnconditionally],                    c1, bytecode[96];
        jump8:
                temp3Low ← 9, GOTO[jumpUnconditionally],                    c1, bytecode[97];



    {Pop and Jump on False}


        popAndJump1OnFalse:
                temp3Low ← 2, GOTO[popAndJumpOnFalse],                      c1, bytecode[98];
        popAndJump2OnFalse:
                temp3Low ← 3, GOTO[popAndJumpOnFalse],                      c1, bytecode[99];
        popAndJump3OnFalse:
                temp3Low ← 4, GOTO[popAndJumpOnFalse],                      c1, bytecode[9a];
        popAndJump4OnFalse:
                temp3Low ← 5, GOTO[popAndJumpOnFalse],                      c1, bytecode[9b];
        popAndJump5OnFalse:
                temp3Low ← 6, GOTO[popAndJumpOnFalse],                      c1, bytecode[9c];
        popAndJump6OnFalse:
                temp3Low ← 7, GOTO[popAndJumpOnFalse],                      c1, bytecode[9d];
        popAndJump7OnFalse:
                temp3Low ← 8, GOTO[popAndJumpOnFalse],                      c1, bytecode[9e];
        popAndJump8OnFalse:
                temp3Low ← 9, GOTO[popAndJumpOnFalse],                      c1, bytecode[9f];


        popAndJumpOnFalse:
                temp2Low ← uCurrentMethodLow, backupIs0Bytes,               c2;
                L1 ← branchIfFalse, GOTO[jump-getTopOfStack]                c3;

    {Extended Jump}


        extendedJump0:
                temp3Low ← 0fc, GOTO[extendedJump],                         c1, bytecode[0a0];
        extendedJump1:
                temp3Low ← 0fd, GOTO[extendedJump],                         c1, bytecode[0a1];
        extendedJump2:
                temp3Low ← 0fe, GOTO[extendedJump],                         c1, bytecode[0a2];
        extendedJump3:
                temp3Low ← 0ff, GOTO[extendedJump],                         c1, bytecode[0a3];
        extendedJump4:
                temp3Low ← 00,  GOTO[extendedJump],                         c1, bytecode[0a4];
        extendedJump5:
                temp3Low ← 01,  GOTO[extendedJump],                         c1, bytecode[0a5];
        extendedJump6:
                temp3Low ← 02,  GOTO[extendedJump],                         c1, bytecode[0a6];
```

```
extendedJump7:
        temp3Low ← 03,  GOTO[extendedJump],                                    c1, bytecode[0a7];

extendedJump:
        temp3Low ← temp3Low LRot8 {get the left byte correct},                 c2;
        temp3Low ← temp3Low + ib + 1,                                          c3;

        ipLow ← ipLow + 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[0aa];
extendedJumpOnTrue3:
        temp3Low ← 3, L1 ← branchIfTrue, GOTO[extendedJumpTrue],               c1, bytecode[0ab];

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[0ac];
extendedJumpOnFalse1:
        temp3Low ← 1, L1 ← branchIfFalse, GOTO[extendedJumpFalse],     c1, bytecode[0ad];
extendedJumpOnFalse2:
        temp3Low ← 2, L1 ← branchIfFalse, GOTO[extendedJumpFalse],     c1, bytecode[0ae];
extendedJumpOnFalse3:
        temp3Low ← 3, L1 ← branchIfFalse, GOTO[extendedJumpFalse],     c1, bytecode[0af];

extendedJumpFalse:
        temp3Low ← temp3Low LRot8 {get high bits of displacement}, GOTO[extendedJumpOnCondition], c2;



extendedJumpOnCondition:

        temp3Low ← temp3Low + ib {and low bits}  + 1{and low bits}, backupIs1Byte, c3;

        ipLow ← ipLow + PC16, {account for extension byte}                     c1;
        temp2Low ← uCurrentMethodLow {point at base of method},                c2;
        GOTO[jump-getTopOfStack],                                              c3;



jump-getTopOfStack:
        {upon entry, temp3Low must contain the number of bytes by which to adjust the instructionPointer. temp2Low must point at
        the base of the object header.  pc16 and temp1Low must both be correct. L1 contains the constant branchIfTrue or
        branchIfFalse}

        MAR ← [stackHigh, stackLow+ 0] {read top of stack},                    c1;
        MDR ← nilPointer, {and smash it}                                       c2;
        temp1Low ← MD,                                                         c3;

        [] ← temp1Low xor falsePointer, ZeroBr,                                c1;
        [] ← temp1Low xor truePointer, ZeroBr, BRANCH[$, tosIsFalse],          c2;
        BRANCH[tosIsNotBoolean, tosIsTrue],                                    c3;

tosIsFalse:
        CANCELBR[$, 1], stackLow ← stackLow - 1, L1Disp, {do we want to branch} c3;

        DISP2[falseTos],                                                       c1;

        NextBytecode,   {no branch}                                            c2, at[branchIfTrue, 4, falseTos];
        DISPNI[bytecodes], ipLow ← ipLow + PC16,                               c3;
```

```
        ipLow ← ipLow - temp2Low,                                   c2, at[branchIfFalse, 4, falseTos];
        GOTO[alterIp] {take branch},                                c3;

tosIsTrue:
        stackLow ← stackLow - 1, L1Disp,{do we want to branch}      c1;

        DISP2[trueTos],                                             c2;

        ipLow ← ipLow - temp2Low, GOTO[alterIp] {take branch},      c3, at[branchIfTrue, 4, trueTos];

        Noop {no branch},                                           c3, at[branchIfFalse, 4, trueTos];
        Noop,                                                       c1;
        NextBytecode,                                               c2;
        DISPNI[bytecodes], ipLow ← ipLow + PC16,                    c3;


tosIsNotBoolean:

        {rats. put the stack back}
        {todo -- more of this when send is implemented}

        MAR ← [stackHigh, stackLow + 0],                            c1;
        MDR ← temp1Low,                                             c2;
        Noop,                                                       c3;

        GOTO[bytecodeFailed],                                       c1;



jumpUnconditionally:
        temp2Low ← uCurrentMethodLow,                               c2;
        ipLow ← ipLow - temp2Low,                                   c3;

alterIp:
        {upon entry, temp3Low 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}

        ipLow ← LShift1 ipLow, SE ← pc16, {yields current byte offset into compiled method including object header}, c1;
        temp3Low ← temp3Low + ipLow {calculate the new location},   c2;
        ipLow ← temp2Low {point at base of object header}, IBPtr ← 1 {drain all bytecodes from instruction buffer}, c3;

        Noop,                                                       c1;
        Noop,                                                       c2;
        Ybus ← ib, GOTO[fixupInstructionPointer],                   c3;
```

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

Reserve[1,7F];

        { E N D }
```

```
{

 2-Aug-84  9:24:16
}

{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 ← nilPointer,                                                  c1;
        temp1Low ← hasPointers, GOTO[createInstance],                           c2;

createInstanceWithBytes:
        temp2Low ← 0,                                                           c1;
        [] ← temp3Low LRot0, XDisp,                                             c2;
        temp3Low ← temp3Low + 1, BRANCH[byteCountIsEven, byteCountIsOdd, 0e],   c3;

   byteCountIsEven:
        temp1Low ← 0, GOTO[byteShift],                                          c1;

   byteCountIsOdd:
        temp1Low ← isOdd, GOTO[byteShift],                                      c1;

   byteShift:
        temp3Low ← RShift1 temp3Low, SE ← 0, GOTO[createInstance],              c2;

createInstanceWithWords:
        temp2Low ← 0,                                                           c1;
        temp1Low ← 0, GOTO[createInstance],                                     c2;


createLargePositiveInteger:
        temp1Low ← classLargePositiveIntegerPointer,                            c1;
        uClassToInstantiate ← temp1Low,                                         c2;
        GOTO [createInstanceWithBytes]                                          c3;


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,                                                    c1;
        temp3Low ← temp3Low + objectHeaderSize, CarryBr,                        c2;
        Q ← objectSizeTestLimit, BRANCH[$, massiveSenility2],                   c3;

        [] ← temp3Low + Q, CarryBr,                                             c1;
        BRANCH[$, requestedSizeTooBig],                                         c2;
        temp2High ← uRumRecordHigh,                                             c3;

        temp2Low ← uRumRecordLow,                                               c1;
        temp1Low ← largestFreeChunkSize,                                        c2;
        [] ← temp3Low - temp1Low, CarryBr,                                      c3;

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


trySpecificList:
        temp2Low ← temp2Low + freeListsOffset,                                  c2;
        Noop,                                                                   c3;

        MAR ← [temp2High, temp2Low + temp3Low],                                 c1;
        CANCELBR[$, 2],                                                         c2;
        otLow ← MD, XDisp, L2 ← creatingInstance
                {set up for the nextFreeChunk call},                            c3;

        uNewObject ← otLow, BRANCH[gotOne, tryBigList, 0e],                     c1;


gotOne:
        L1 ← gettingNextFreeChunk {needed for otMap2 call inside nextFreeChunk}, CALL[nextFreeChunk], c2;




        MAR ← [temp2High, temp2Low + temp3Low] {update free list head}          c1, at[creatingInstance, 10,
        nextFreeChunk-return];
        MDR ← Q, CANCELBR[$, 2], LOOPHOLE[wok],                                 c2;
        {save new object's address}
        Q ← temp1High,                                                          c3;

        uNewObjectHigh ← Q,                                                     c1;
        uNewObjectLow ← temp1Low,                                               c2;
        GOTO[allocate],                                                         c3;



tryBigList:
        temp2Low ← uRumRecordLow, GOTO[useBigFreeListA],                        c2;

useBigFreeList:
```

```
        {upon entry, temp2High/Low contains the rum record address.
                uRequestedSize is valid. temp3Low is the requested size}
        Noop,                                                                   c2;

useBigFreeListA:
        uPredecessor ← ~otLow xor otLow {yields -1},
                L1 ← gettingNextFreeChunk {for otmap call in nextFreeChunk},    c3;

        MAR ← [temp2High, temp2Low + bigFreeListOffset],
                L2 ← consideringBigChunks,                                      c1;
        temp3Low ← temp3Low + objectHeaderSize, CarryBr,
                {yields minimum splittable block size} CANCELBR[$, 2],          c2;
        otLow ← MD {current free chunk}, XDisp {test if more},
                BRANCH[$, massiveSenility4],                                    c3;

considerNextBigFreeChunk:
        uNewObject ← otLow, BRANCH[$, outOfChunks, 0e],                         c1;
        uCurrentFreeChunkOop ← otLow, CALL[nextFreeChunk],                      c2;

        uNextFreeChunk ← Q {remember next free chunk},                          c1, at[consideringBigChunks, 10,
        nextFreeChunk-return];
        temp1Low ← temp1Low + sizeFieldOffset,                                  c2;
        Noop,                                                                   c3;

        MAR ← [temp1High, temp1Low + 0],                                        c1;
        Noop,                                                                   c2;
        Q ← MD {size of current free chunk},                                    c3;

        [] ← Q xor uRequestedSize, ZeroBr,                                      c1;
        [] ← Q - temp3Low, CarryBr, BRANCH[$, exactFit],                        c2;
        BRANCH[iterate, canSubdivide],                                          c3;


iterate:
        uPredecessor ← otLow,                                                   c1;
        Noop,                                                                   c2;
        otLow ← uNextFreeChunk, XDisp, GOTO[considerNextBigFreeChunk],          c3;


exactFit:
        temp1Low ← temp1Low - sizeFieldOffset, CANCELBR[$, 1],                  c3;

        Q ← temp1High,                                                          c1;
        uNewObjectLow ← temp1Low,                                               c2;
        uNewObjectHigh ← Q, GOTO[splice],                                       c3;


canSubdivide:
        temp3Low ← uRequestedSize,                                              c1;
        temp3Low ← Q {current chunk size} - temp3Low {requested size} {yields excess size}, c2;
        Q ← temp1High {part of new objects address},                            c3;

        MAR ← [temp2High, temp2Low + freePointersOopOffset],                    c1;
        uNewObjectHigh ← Q, CANCELBR[$, 2],                                     c2;
        otLow ← MD {first free oop}, XDisp,                                     c3;

        BRANCH[$, outOfOops, 0e],                                               c1;
        Noop,                                                                   c2;
        Noop,                                                                   c3;

                {write the new size of the current free chunk (temp1High/Low still pointing at its size field)}
        MAR ← [temp1High, temp1Low + 0],                                        c1;
        MDR ← temp3Low,                                                         c2;
        temp1Low ← temp1Low - sizeFieldOffset {point at base of the current free chunk}, c3;

        {write address of new object into its ot entry}
        MAR ← [otHigh, otLow + 1],                                              c1;
        MDR ← temp1High, CANCELBR[$, 2], LOOPHOLE[wok],                         c2;
        Q ← temp1Low {base of free chunk we split} + temp3Low {excessSize} {yields base of new object}, c3;

        MAR ← [otHigh, otLow + 0],                                              c1;
        uNewObjectLow ← MDR ← Q,                                                c2;
        Q ← MD {link to next free oop},                                         c3;

        MAR ← [temp2High, temp2Low + freePointersOopOffset],                    c1;
        MDR ← Q {update free oop list head}, CANCELBR[$, 2], LOOPHOLE[wok],     c2;
        uNewObject ← otLow,                                                     c3;

        {should we move the current free chunk to a small free chunk list?}
        temp2Low ← largestFreeChunkSizeLessOne,                                 c1;
        [] ← temp3Low  {excess size} - temp2Low, CarryBr,                       c2;
        otLow ← uCurrentFreeChunkOop, BRANCH[$, itsFineWhereItIs],              c3;

        L1 ← movingFromBigToSmall,                                              c1;
        Noop,                                                                   c2;
        CALL[addToFreeChunkList] {this call returns directly to splice},        c3;


splice:
        otLow ← uPredecessor, XDisp, L1 ← splicingBigFreeList,                  c1, at[movingFromBigToSmall, 10,
```

```
            addToFreeChunkList-return];
            temp2High ← uRumRecordHigh, BRANCH[didHavePredecessor, noPredecessor, 0e], c2;

    noPredecessor:
            temp2Low ← uRumRecordLow,                                                   c3;

            MAR ← [temp2High, temp2Low + bigFreeListOffset],GOTO[linkNextChunk],         c1;


    didHavePredecessor:
            CALL[otMap2],                                                               c3;

            temp1Low ← temp1Low + chunkLinkOffset,                                      c1, at[splicingBigFreeList, 10, otMap2-return];
            Noop,                                                                       c2;
            Noop,                                                                       c3;

            MAR ← [temp1High, temp1Low + 0], GOTO[linkNextChunk],                       c1;


    linkNextChunk:
            MDR ← uNextFreeChunk, CANCELBR[bigListWrapup, 2], LOOPHOLE[wok],            c2;









    itsFineWhereItIs:
            Noop,                                                                       c1;
            Noop,                                                                       c2;
    bigListWrapup:
            otLow ← uNewObject, GOTO[allocate],                                         c3;









    outOfChunks:
            GOTO[massiveSenility],                                                      c2;
    outOfOops:
            Noop,                                                                       c2;
    massiveSenility:
            Noop,                                                                       c3;

    massiveSenility2:
            GOTO[bytecodeFailed],                                                       c1;
    massiveSenility4:
            CANCELBR[bytecodeFailed, 0F],                                               c1;

    requestedSizeTooBig:
            GOTO[massiveSenility2],                                                     c3;
```

```
{       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. MesaIntRq]
}
```

```
    allocate:
            temp2High ← uRumRecordHigh,                                                 c1;
            temp2Low ← uRumRecordLow,                                                   c2;
            Noop,                                                                       c3;

            MAR ← [temp2High, temp2Low + oopLevelLowOffset],                            c1;
            CANCELBR [$, 2],                                                            c2;
            temp3Low ← MD,                                                              c3;

            MAR ← [temp2High, temp2Low + oopLevelLowOffset],                            c1;
            MDR ← temp3Low ← temp3Low - 1, CANCELBR [$, 2], LOOPHOLE [wok],             c2;
            Noop,                                                                       c3;

    lowOopTest:
            MAR ← [temp2High, temp2Low + oopAlertLevelLowOffset],                       c1;
            CANCELBR [$, 2],                                                            c2;
            Q ← MD,                                                                     c3;

            Noop,                                                                       c1;
            Q ← temp3Low - Q, CarryBr,                                                  c2;
            BRANCH [$, decreaseWordLevel],                                              c3;

            MAR ← [temp2High, temp2Low + alreadyAlertedOffset],                         c1;
            CANCELBR [$, 2],                                                            c2;
            Q ← MD,                                                                     c3;

            [] ← Q, ZeroBr,                                                             c1;
```

```
            BRANCH [decreaseWordLevel2, $],                                  c2;
            Noop,                                                            c3;

            MAR ← [temp2High, temp2Low + signalAlertOffset],                 c1;
            MDR ← 1, CANCELBR [$, 2], LOOPHOLE [wok],                        c2;
            MesaIntRq, GOTO [decreaseWordLevel],                             c3;

    decreaseWordLevel2:
            Noop,                                                            c3;

    decreaseWordLevel:
            MAR ← [temp2High, temp2Low + wordLevelLowOffset],                c1;
            temp3Low ← uRequestedSize, CANCELBR [$, 2],                      c2;
            Q ← MD,                                                          c3;

            MAR ← [temp2High, temp2Low + wordLevelLowOffset],                c1;
            MDR ← temp3Low ← Q - temp3Low, CANCELBR [$, 2],
                  LOOPHOLE [wok], CarryBr,                                   c2;
            BRANCH [wordLevelBorrow, lowMemoryTestHighGetData],              c3;

    wordLevelBorrow:
            MAR ← [temp2High, temp2Low + wordLevelHighOffset],               c1;
            CANCELBR [$, 2],                                                 c2;
            Q ← MD,                                                          c3;

            MAR ← [temp2High, temp2Low + wordLevelHighOffset],               c1;
            MDR ← Q - 1, CANCELBR [$, 2], LOOPHOLE [wok],                    c2;
            temp1Low ← Q - 1, GOTO [lowMemoryTestHighHaveData],              c3;

    lowMemoryTestHighGetData:
            MAR ← [temp2High, temp2Low + wordLevelHighOffset],               c1;
            CANCELBR [$, 2],                                                 c2;
            temp1Low ← MD,                                                   c3;

    lowMemoryTestHighHaveData:
            MAR ← [temp2High, temp2Low + wordAlertLevelHighOffset],          c1;
            CANCELBR [$, 2],                                                 c2;
            Q ← MD,                                                          c3;

            Q ← Q - temp1Low, CarryBr,                                       c1;
            [] ← Q, ZeroBr, BRANCH [reallyAllocate1, $],                     c2;
            BRANCH [$, lowMemoryTestLow],                                    c3;

            MAR ← [temp2High, temp2Low + alreadyAlertedOffset],              c1;
            CANCELBR [$, 2],                                                 c2;
            Q ← MD,                                                          c3;

            [] ← Q, ZeroBr,                                                  c1;
            BRANCH [lowMemoryTestLow2, $],                                   c2;
            Noop,                                                            c3;

            MAR ← [temp2High, temp2Low + signalAlertOffset],                 c1;
            MDR ← 1, CANCELBR [$, 2], LOOPHOLE [wok],                        c2;
            MesaIntRq, GOTO [reallyAllocate3],                               c3;

    lowMemoryTestLow2:
            Noop,                                                            c3;

    lowMemoryTestLow:
            MAR ← [temp2High, temp2Low + wordAlertLevelLowOffset],           c1;
            CANCELBR [$, 2],                                                 c2;
            Q ← MD,                                                          c3;

            Noop,                                                            c1;
            Q ← temp3Low - Q, CarryBr,                                       c2;
            BRANCH [$, reallyAllocate3],                                     c3;

            MAR ← [temp2High, temp2Low + alreadyAlertedOffset],              c1;
            CANCELBR [$, 2],                                                 c2;
            Q ← MD,                                                          c3;

            [] ← Q, ZeroBr,                                                  c1;
            BRANCH [reallyAllocate2, $],                                     c2;
            Noop,                                                            c3;

            MAR ← [temp2High, temp2Low + signalAlertOffset],                 c1;
            MDR ← 1, CANCELBR [$, 2], LOOPHOLE [wok],                        c2;
            MesaIntRq, GOTO [reallyAllocate3],                               c3;

    reallyAllocate1:
            CANCELBR [reallyAllocate3, 1],                                   c3;

    reallyAllocate2:
            Noop,                                                            c3;

    reallyAllocate3:
            Noop,                                                            c1;
            temp1High ← uNewObjectHigh,                                      c2;
            temp1Low ← uNewObjectLow,                                        c3;

            temp3Low ← uRequestedSize,                                       c1;
            temp3Low ← temp1Low + temp3Low
                    {low 16 bits of address of one word
                    past the end of the object},                             c2;
            temp1Low ← temp1Low + deltaWordOffset,                           c3;

            {initialize the object header now}
```

```
        MAR ← [temp1High, temp1Low + 0],                                    c1;
        {the following MDR← 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}
        MDR ← uFieldType,                                                   c2;
        IfEqual [Waffle, 1, SkipTo[waffleOnly], ];
        temp1Low ← temp1Low + offsetFromDeltaWordToSizeField,               c3;
        IfEqual [Waffle, 0, SkipTo[endWaffleOnly], ];

waffleOnly!
        temp1Low ← temp1Low + offsetFromDeltaWordToDiskWord,                c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        MDR ← 0,                                                            c2;
        temp1Low ← temp1Low + offsetFromDiskWordToSizeField,                c3;

endWaffleOnly!
        MAR ← [temp1High, temp1Low + 0],                                    c1;
        MDR ← uRequestedSize,                                               c2;
        temp1Low ← temp1Low + offsetFromSizeFieldToClassField,              c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        MDR ← uClassToInstantiate,                                          c2;

        {ok, object header is done, now zap the object body}
        temp1Low ← temp1Low + offsetFromClassFieldToFirstField,             c3;

initializeObjectBody:
        Noop,                                                               c1;
        [] ← temp1Low - temp3Low, ZeroBr,                                   c2;
        BRANCH[$, allZapped],                                               c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        MDR ← uDefault,                                                     c2;
        temp1Low ← temp1Low + 1, COTO[initializeObjectBody],                c3;


allZapped:

        {the object header and object body are now completely initialized. now fix up the object table entry}
        MAR ← [otHigh, otLow + 1] {read second word of ot entry},           c1;
        CANCELBR[$, 2],                                                     c2;
        Q ← MD,                                                             c3;

        {the only thing worth preserving in the ot entry is the segment number.
                clobber everything else, turn on the untouched bit and write ot entry}

        MAR ← [otHigh, otLow + 1] {read second word of ot entry},           c1;
        Q ← MDR ← Q and 0f,     CANCELBR[$, 2], LOOPHOLE[wok],              c2;
        temp3Low ← temp3High
                {retrieve return link cause addToZeroCountTable will smash it},
                L2 ← creatingAnInstance,                                    c3;

        temp2Low ← temp3Low {save return link}, CALL[addToZeroCountTable],  c1;

        Noop,                                                               c1, at[creatingAnInstance, 10,
        addToZeroCountTableReturn];
        L1 ← upClassAtInstantiation,                                        c2;
        otLow ← uClassToInstantiate, XDisp, CALL[refi],                     c3;

        otLow ← uNewObject,                                                 c1, at[upClassAtInstantiation, 10, refiReturn];
        Xbus ← temp2Low LRot0, XDisp,                                       c2;
        RET[createInstance-return],                                         c3;




nextFreeChunk:
        CALL[otMap2],                                                       c3;

        temp1Low ← temp1Low + chunkLinkOffset,                              c1, at[gettingNextFreeChunk, 10, otMap2-return];
        Noop,                                                               c2;
        Noop,                                                               c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        temp1Low ← temp1Low - chunkLinkOffset, L2Disp,                      c2;
        Q ← MD, RET[nextFreeChunk-return],                                  c3;
```

```
addToFreeChunkList:
        {upon entry, temp1High/Low must be the base of the object to add to the free list. otLow must be the oop of the free
        chunk to put on the list and temp3Low must be the freelist index (0 -- 45) to put the oop on.  L1 is the return linkage
        register. smashes temp2High/Low, Q}

        temp2High ← uRumRecordHigh,                                          c1;
        temp2Low ← uRumRecordLow,                                            c2;
        temp2Low ← temp2Low + freeListsOffset,                               c3;

        MAR ← [temp2High, temp2Low + temp3Low],                              c1;
        temp1Low ← temp1Low + chunkLinkOffset, CANCELBR[$, 2],               c2;
        Q ← MD {current free list head},                                     c3;

        MAR ← [temp1High, temp1Low + 0],                                     c1;
        MDR ← Q,                                                             c2;
        Noop,                                                                c3;

        MAR ← [temp2High, temp2Low + temp3Low],                              c1;
        MDR ← otLow, CANCELBR[$, 2], LOOPHOLE[wok], L1Disp,                  c2;
        temp1Low ← temp1Low - chunkLinkOffset, RET[addToFreeChunkList-return],  c3;
```

```
refi:
        {upon entry, otLow must indicate the oop to be refi'd. SmallIntegers acceptable. There must be a pending XDisp to test
        for SmallIntegers, and L1 is the return linkage register. smashes Q and temp1Low}

        [] ← 6 - otLow, CarryBr, BRANCH[$, isSmallRefi, 0e],                    c1;
        L1Disp, BRANCH[doRefi, $],                                              c2;
        RET[refiReturn],                                                        c3;

doRefi:
        CANCELBR[$, 0f],                                                        c3;

        MAR ← [otHigh, otLow + 1],                                              c1;
        temp1Low ← 4 {for adding "one" to ref count}, CANCELBR[$, 2],           c2;
        Q ← MD,  XHDisp {first part of test for stuck ref count},               c3;

        temp1Low ← temp1Low LRot8, BRANCH[notStuckRefi, maybeStuckRefi, 2],     c1;

notStuckRefi:  {sign is positive, thus not stuck and can not become stuck}
        Q ← Q + temp1Low {up ref count},                                        c2;
        GOTO[updateOtRefi],                                                     c3;

maybeStuckRefi: {sign is negative, can get stuck, may already be stuck}
        Q ← Q + temp1Low {up ref count}, CarryBr {carry implies already stuck}, c2;
        [] ← Q + temp1Low, CarryBr {carry implies just got stuck}, BRANCH[updateOtRefi, stuckRefi], c3;

updateOtRefi:
        MAR ← [otHigh, otLow + 1], BRANCH[$, justGotStuckRefi],                 c1;
        MDR ← Q, LOOPHOLE [mdrok], LOOPHOLE[wok], CANCELBR[returnFromRefi, 3], L1Disp,  c2;


justGotStuckRefi:
        {Loom:  need to call Loom here for newly stuck refi}

        MDR ← Q {write updated ref count}, LOOPHOLE[mdrok], LOOPHOLE[wok], CANCELBR[returnFromRefi, 3], L1Disp, c2;


stuckRefi:
        CANCELBR[$, 3],                                                         c1;
        L1Disp, GOTO[returnFromRefi],                                           c2;



isSmallRefi:
        L1Disp, CANCELBR[returnFromRefi, 3],                                    c2;

returnFromRefi:
        RET[refiReturn],                                                        c3;



refd:
        {upon entry, otLow must be the oop to be refd'd. SmallIntegers acceptable.  There must be a pending XDisp to test for
        SmallIntegers, and L1 is the return linkage register.  smashes Q and temp3High/Low and temp1High/Low and L2}


        [] ← 6 - otLow, CarryBr, BRANCH[$, isSmallRefd, 0e],                    c1;
        L1Disp, BRANCH[doRefd, $],                                              c2;
        RET[refdReturn],                                                        c3;

doRefd:
        CANCELBR[$, 0f],                                                        c3;

        MAR ← [otHigh, otLow + 1],                                              c1;
        temp1Low ← 0fc {for "subtracting one" from the reference count}, CANCELBR[$, 2], c2;
        Q ← MD, XHDisp {first part of stuck ref count test},                    c3;

        temp1Low ← temp1Low LRot8, BRANCH[positiveRefCount, negativeRefCount, 2],c1;

positiveRefCount: {not stuck but could go to zero}
        Q ← Q + temp1Low {subtract 1}, CarryBr {carry implies already zero, an error}, L2 ← doingRefd,  c2;
        [] ← Q + temp1Low {subtract again}, CarryBr {no carry implies just went tozero}, BRANCH[triedToRefdZeroCountObject,
        updateOtRefd],  c3;

negativeRefCount: {could be stuck but cannot go to zero}
        temp3Low ← 4,                                                           c2;
        temp3Low ← temp3Low LRot8,                                              c3;

        [] ← Q + temp3Low, CarryBr {carry implies stuck ref count},             c1;
        Q ← Q + temp1Low {subtract one from ref count}, BRANCH[notStuckRefd, stuckRefd], c2;

notStuckRefd:
        Xbus ← 1, XDisp {makes the branch at updateOtRefd happy!},              c3;

updateOtRefd:
        MAR ← [otHigh, otLow + 1]{write updated refcount}, BRANCH[addToZeroCountTable, notZeroRefd], c1;
```

```
        GOTO[isSmallRefd],                                              c1, at[doingRefd, 10, addToZeroCountTableReturn];

notZeroRefd:
        MDR ← Q,  LOOPHOLE[wok], L1Disp, CANCELBR[returnFromRefd, 3],   c2;

stuckRefd:
        Noop,                                                           c3;

        GOTO[isSmallRefd],                                              c1;

isSmallRefd:
        L1Disp, CANCELBR[returnFromRefd, 3],                            c2;


returnFromRefd:
        RET[refdReturn],                                                c3;

            {the following mess is the result of addressing constraints and branching limitations}
triedToRefdZeroCountObject:
        BRANCH[triedToRefdZeroCountObjectA, triedToRefdZeroCountObjectB],   c1;
triedToRefdZeroCountObjectA:
        GOTO[bailout3],                                                 c2;
triedToRefdZeroCountObjectB:
        GOTO[bailout3],                                                 c2;




addToZeroCountTable:
        {Loom: Loom may want to get involved here--but I don't think so}

        {upon entry, otLow is the oop to put into the zct, and Q is the second word of the OT entry for otlow.  L2 is the return
        linkage register. smashes temp1High/Low and temp3High/Low and Q. we turn on the inZCT bit in the OT & write the new OT
        entry. see if oop is already in the zct(by looking at former OT entry). if not we need to put the oop into the ZCT}

        temp1Low ← 2, CANCELBR[$, 2],                                   c2;
        temp1Low ← temp1Low LRot8, {the inzct bit}                      c3;

        MAR ← [otHigh, otLow + 1],                                      c1;
        MDR ← Q or temp1Low {turn on zct bit}, LOOPHOLE[wok], CANCELBR[$, 2],   c2;
        temp3Low ← MD {former OT entry so we can test previous inZct bit},      c3;

        [] ← temp3Low and temp1Low, ZeroBr,                             c1;
        Q ← uRumRecordHigh {retreive the Rum 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...}

        temp1High ← Q LRot0, CANCELBR[$, 0f],                           c3;

        temp1Low ← uRumRecordLow,                                       c1;
        Noop,                                                           c2;
        Noop,                                                           c3;

        MAR ← [temp1High, temp1Low + zctIndexOffset]{read current index},       c1;
        CANCELBR[$, 2],                                                 c2;
        Q ← MD,  LOOPHOLE[mdok],                                        c3;

        MAR ← [temp1High, temp1Low + zctLowOffset]{get zct address},    c1;
        CANCELBR[$, 2],                                                 c2;
        temp3Low ← MD,                                                  c3;

        MAR ← [temp1High, temp1Low + zctHighOffset],                    c1;
        temp3Low ← temp3Low + Q, CANCELBR[$, 2],                        c2;
        temp3High ← MD,                                                 c3;

        {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],                                c1;
        MDR ← otLow,                                                    c2;
        Noop,                                                           c3;

        MAR ← [temp1High, temp1Low + zctIndexOffset]{write updated index},      c1;
        MDR ← Q ← Q + 1, CANCELBR[ $,2], LOOPHOLE[wok],                 c2;
        Noop,                                                           c3;

        MAR ← [temp1High, temp1Low + stabilizationLimitOffset],         c1;
        CANCELBR[$, 2],                                                 c2;
        temp3Low ← MD,                                                  c3;

        Noop,                                                           c1;
        [] ← temp3Low - Q, NegBr,                                       c2;
        BRANCH[zctIndexOk, stabilizationNeeded],                        c3;

stabilizationNeeded:
        MAR ← [temp1High, temp1Low + stabilizationFlagOffset],          c1;
        MDR ← 1, CANCELBR[$, 2], LOOPHOLE[wok],                         c2;
```

```
        uTimeToStabilize ← ~stackLow xor stackLow,                    c3;

zctIndexOk:
        Noop,                                                         c1;
        L2Disp,                                                       c2;

returnFromAddToZeroCountTable:
        RET[addToZeroCountTableReturn],                               c3;
```

```
makeVolatile:
        {upon entry, otLow is the oop to make volatile,  uMakeVolatileLinkage is the return linkage register: if it is odd, each
        object referred to by the object will be refd'd; it it is even, the object is marked volatile, but no refding occurs.
        smashes temp3Low, Q, L1, L2. leaves base of object in uMakeVolatileHigh/Low, and in temp1High/Low. leaves uLastPointer
        set up}

        {see if we're trying to make nil volatile -- this happens when the leaf context oop is nil.  check should probably be
        moved to the place where volatilization is done after stabilization...}

        [] ← otLow xor nilPointer, ZeroBr,                                  c2;
        BRANCH[$, nilMakeVolatile],                                         c3;

        uMakeVolatileOop ← otLow,                                           c1;
        L1 ← makingVolatile,                                                c2;
        CALL[otMap] {get address of base of object},                        c3;

        Q ← temp1High {save object base}                                    c1, at[makingVolatile, 10, otMap-return];
        uMakeVolatileHigh ← Q,                                              c2;
        uMakeVolatileLow ← temp1Low,                                        c3;

        temp1Low ← temp1Low + deltaWordOffset,                              c1;
        Noop,                                                               c2;
        Noop,                                                               c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        Noop,                                                               c2;
        Q ← MD {delta word},                                                c3;

        [] ← Q and 4, ZeroBr {volatile bit},                                c1;
        Ybus ← uMakeVolatileLinkage, XDisp, BRANCH[alreadyVolatile, doMakeVolatile], c2;

alreadyVolatile:
        temp1Low ← temp1Low - deltaWordOffset, RET[makeVolatile-return],    c3;

doMakeVolatile:
        Q ← Q or 4 {volatile bit}, CANCELBR[$, 0f],                         c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        MDR ← Q {delta word with volatile bit set},                         c2;
        Ybus ← uMakeVolatileLinkage, XDisp, {should we refd the referents or not?}, c3;

        temp1Low ← temp1Low - deltaWordOffset, BRANCH[returnFromMakeVolatile, doRefdFields, 0e], c1;


doRefdFields:
        temp1Low ← temp1Low + sizeFieldOffset,                              c2;
        Noop,                                                               c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        temp1Low ← temp1Low - sizeFieldOffset,                              c2;
        temp3Low ← MD {size field},                                         c3;

        temp3Low ← temp3Low + temp1Low,                                     c1;
        temp3Low ← temp3Low - 1 {low 16 bits of last pointer of context object},c2;
        uLastPointer ← temp3Low,                                            c3;

        {now, sweep the object decrementing reference counts of all pointer fields}

        temp2Low ← temp1Low + firstPointerFieldOfObject,                    c1;
        Q ← temp1High,                                                      c2;
        temp2High ← Q LRot0,                                                c3;

makeVolatileLoop:
        MAR ← [temp2High, temp2Low + 0], L1 ← inMakeVolatile,               c1;
        Noop,                                                               c2;
        otLow ← MD, XDisp, CALL[refd],                                      c3;

        temp3Low ← uLastPointer,                                            c1, at[inMakeVolatile, 10, refdReturn];
        [] ←  temp2Low - temp3Low, ZeroBr,                                  c2;
        temp2Low ← temp2Low + 1, BRANCH[makeVolatileLoop, $],               c3;

        otLow ← uMakeVolatileOop,                                           c1;
        Noop,                                                               c2;
        Noop,                                                               c3;

        MAR ← [otHigh, otLow + 1],                                          c1;
        CANCELBR[$, 2],                                                     c2;
        Q ← MD {second word of ot entry of object we are volatilizing}, L2 ← inMakeVolatile, c3;

        CALL[addToZeroCountTable],                                          c1;

        temp1High ← uMakeVolatileHigh,                                      c1, at[inMakeVolatile, 10,
        addToZeroCountTableReturn];
        temp1Low ← uMakeVolatileLow,                                        c2;
        Noop,                                                               c3;


        Noop,                                                               c1;

returnFromMakeVolatile:
        Ybus ← uMakeVolatileLinkage, XDisp,                                 c2;

returningFromMakeVolatile:
        RET[makeVolatile-return],                                           c3;
```

```
        nilMakeVolatile:
                GOTO[returnFromMakeVolatile],                    c1;
```

```
lastPointerOf:
        {upon entry,  temp1High/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}, ZeroBr,                                          c1;
        Q ← classCompiledMethodOop, BRANCH[doesHavePointers, doesNotHavePointers], c2;

doesHavePointers:
        {is pure pointer object -- last pointer is size of object}
        temp1Low ← temp1Low + sizeFieldOffset,                                       c3;

        MAR ← [temp1High, temp1Low + 0]{start read of length field},                 c1;
        temp1Low ← temp1Low - sizeFieldOffset {again point at base of object},       c2;
        temp3Low ← MD, GOTO[returnFromLastPointerOf],                                c3;

doesNotHavePointers:
        {no pointers, might be compiledMethod -- need to check class}
        temp1Low ← temp1Low + classFieldOffset,                                      c3;

        MAR ← [temp1High, temp1Low + 0],                                             c1;
        temp1Low ← temp1Low - classFieldOffset{again point at base of object},       c2;
        temp3Low ← MD {the class of the object},                                     c3;

        [] ← temp3Low xor Q {compiledMethodClass oop}, ZeroBr,                       c1;
        BRANCH[notCompiledMethod, isCompiledMethod],                                 c2;

notCompiledMethod:
        temp3Low ← objectHeaderSize, GOTO[returnFromLastPointerOf],                  c3;

isCompiledMethod:
        {need to get number of literals from the method header}
        temp1Low ← temp1Low + objectHeaderSize {point at method header},             c3;

        MAR ← [temp1High, temp1Low + 0],                                             c1;
        temp1Low ← temp1Low - objectHeaderSize, {again point at base of object}      c2;
        temp3Low ← MD {the method header},                                           c3;

        temp3Low ← (RShift1 temp3Low and 7f){get literal count of compiledMethod}, SE ← 0, c1;
        temp3Low ← temp3Low + literalStart,                                          c2;
        temp3Low ← temp3Low + objectHeaderSize,                                      c3;

returnFromLastPointerOf:
        temp3Low ← temp3Low - 1,                                                     c1;
        temp3Low ← temp3Low + temp1Low, L2Disp,                                      c2;
        uLastPointer ← temp3Low, RET[lastPointerOf-return],                          c3;
```

```
{       Test the memory and oop levels and reset alreadyAlerted if both are above their alert levels.

        ((wordLevel >= wordAlertLevel) and: [oopLevel >= oopAlertLevel]) ifTrue: [alreadyAlerted ← 0]
}

        stabilize:
                {linkage register is L0 -- runs only between bytecodes!}
                temp1High ← uRumRecordHigh,                                     c1;
                temp1Low  ← uRumRecordLow,                                      c2;
                uTimeToStabilize ← 0,                                           c3;

        testOopLevel:
                MAR ← [temp1High, temp1Low + oopLevelLowOffset],                 c1;
                CANCELBR [$, 2],                                                c2;
                temp3Low ← MD,                                                  c3;

                MAR ← [temp1High, temp1Low + oopAlertLevelLowOffset],            c1;
                CANCELBR [$, 2],                                                c2;
                Q ← MD,                                                         c3;

                Q ← temp3Low - Q, CarryBr,                                      c1;
                BRANCH [stabilize2, $],                                         c2;
                Noop,                                                           c3;

        testWordLevelHigh:
                MAR ← [temp1High, temp1Low + wordLevelHighOffset],              c1;
                CANCELBR [$, 2],                                                c2;
                temp3Low ← MD,                                                  c3;

                MAR ← [temp1High, temp1Low + wordAlertLevelHighOffset],         c1;
                CANCELBR [$, 2],                                                c2;
                Q ← MD,                                                         c3;

                Q ← temp3Low - Q, CarryBr,                                      c1;
                [] ← Q, ZeroBr, BRANCH [stabilize1, $],                         c2;
                BRANCH [resetAlreadyAlerted, testWordLevelLow],                 c3;

        testWordLevelLow:
                MAR ← [temp1High, temp1Low + wordLevelLowOffset],               c1;
                CANCELBR [$, 2],                                                c2;
                temp3Low ← MD,                                                  c3;

                MAR ← [temp1High, temp1Low + wordAlertLevelLowOffset],          c1;
                CANCELBR [$, 2],                                                c2;
                Q ← MD,                                                         c3;

                Q ← temp3Low - Q, CarryBr,                                      c1;
                BRANCH [stabilize3, $],                                         c2;
                Noop,                                                           c3;

        resetAlreadyAlerted:
                MAR ← [temp1High, temp1Low + alreadyAlertedOffset],             c1;
                MDR ← 0, CANCELBR [$, 2], LOOPHOLE [wok],                       c2;
                GOTO [reallyStabilize],                                         c3;

        stabilize1:
                CANCELBR [reallyStabilize, 1],                                  c3;

        stabilize2:
                GOTO [reallyStabilize],                                         c3;

        stabilize3:
                GOTO [reallyStabilize],                                         c3;

        reallyStabilize:
                {get address of the zct}
                MAR ← [temp1High, temp1Low + zctLowOffset],                     c1;
                CANCELBR[$, 2],                                                 c2;
                temp2Low ← MD,                                                  c3;

                MAR ← [temp1High, temp1Low + zctHighOffset],                    c1;
                uZctBaseLow ← temp2Low, CANCELBR[$, 2],                         c2;
                temp2High ← MD,                                                 c3;

                {get, then smash the zct index from the Rum record}
                MAR ← [temp1High, temp1Low + zctIndexOffset],                   c1;
                MDR ← 0, CANCELBR[$, 2], LOOPHOLE[wok],                         c2;
                temp3Low ← MD,                                                  c3;

                {reset the stabilization flag}
                MAR ← [temp1High, temp1Low +  stabilizationFlagOffset],         c1;
                MDR ← 0, CANCELBR[$, 2], LOOPHOLE[wok],                         c2;
                temp3Low ← temp3Low + temp2Low {yields low 16 bits of one word past the last valid oop in the zct}, c3;

                uZctLimit ← temp3Low,                                           c1;
                uQueueHead ← ~otLow xor otLow,                                  c2;
                uCurrentObject ← ~otLow xor otLow,                              c3;

                {sweep the zct. for each oop marked (in its ot entry) as volatile, reset the isVolatile 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}

        stabilizationLoop:
```

```
            [] ← temp2Low xor uZctLimit, ZeroBr {are we there yet?},          c1;
            temp3Low ← 0ff, BRANCH[$, countsAreNowCorrect],                   c2;
            temp3Low ← temp3Low LRot8,                                        c3;

            MAR ← [temp2High, temp2Low + 0],                                  c1;
            temp2Low ← temp2Low + 1, L1 ← stabilizing,                        c2;
            otLow ← MD, CALL[otMap2] {so we can get its delta word},          c3;

            temp1Low ← temp1Low + deltaWordOffset,                            c1, at[stabilizing, 10, otMap2-return];
            temp3Low ← temp3Low or 0fb {yields fffb, for turning off the isVolatile bit}, c2;
            Noop,                                                             c3;

            MAR ← [temp1High, temp1Low + 0] {read delta word}                 c1;
            Noop,                                                             c2;
            Q ← MD, XDisp {to test isVolatile bit},                           c3;

            MAR ← [temp1High, temp1Low + 0], BRANCH[oopIsNotVolatile, $, 0b], c1;
            Q ← MDR ← Q and temp3Low {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...}
            Q ← temp1High,                                                    c1, at[stabilizingContext, 10,
            lastPointerOf-return];
            temp3High ← Q LRot0,                                              c2;
            temp3Low ← temp1Low + classFieldOffset,                           c3;

            {sweep over the volatile object, upping the reference counts of its referents}

        upReferents:
            MAR ← [temp3High, temp3Low + 0], L1 ← correcting,                 c1;
            Noop,                                                             c2;
            otLow ← MD, XDisp, CALL[refi],                                    c3;

            [] ← temp3Low xor uLastPointer, ZeroBr,                           c1, at[correcting, 10, refiReturn];
            temp3Low ← temp3Low + 1, BRANCH[$, thisOneIsStable],              c2;
            GOTO[upReferents],                                                c3;

        oopIsNotVolatile:
            Noop,                                                             c2;
        thisOneIsStable:
            GOTO[stabilizationLoop],                                          c3;




        countsAreNowCorrect:
            {at this point, all contexts have been stabilized, and all reference counts are correct.  sweep 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 ←  uZctBaseLow,                                      c3;

        sweepAndDeallocateLoop:
            [] ← temp2Low xor uZctLimit, ZeroBr {are we there yet?},          c1;
            BRANCH[$, returnFromStabilize],                                   c2;
            temp1Low ← 0fd,                                                   c3;

            MAR ← [temp2High, temp2Low + 0] {get oop from zct}                c1;
            temp1Low ← temp1Low LRot8,                                        c2;
            otLow ← MD,                                                       c3;

            MAR ← [otHigh, otLow + 1] {get its ot entry}                      c1;
            temp1Low ← temp1Low or 0ff {yields fdff for turning off inZct bit}, CANCELBR[$, 2], c2;
            temp3Low ← MD,                                                    c3;


            MAR ← [otHigh, otLow + 1] {rewrite its ot entry}                  c1;
            MDR ← temp3Low and temp1Low {turns off the inZct bit},  CANCELBR[$, 2], LOOPHOLE[wok], c2;
            temp3Low ← temp3Low LRot8,                                        c3;

            temp1Low ← 0fc {mask is reference count bits},                    c1;
            [] ← temp3Low and temp1Low, ZeroBr,                               c2;
            temp2Low ← temp2Low + 1, BRANCH[sweepAndDeallocateLoop, needToDeallocate], c3;

        needToDeallocate:
            {save our current state}
            Q ← temp2High,                                                    c1;
            uZctSweepHigh ← Q, L3 ← fromStabilize,                            c2;
            uZctSweepLow ← temp2Low, CALL[deallocate],                        c3;

            {recover our state}
            temp2High ← uZctSweepHigh,                                        c1, at[fromStabilize, 10, deallocate-return];
            temp2Low ← uZctSweepLow,                                          c2;
            GOTO[sweepAndDeallocateLoop],                                     c3;


        returnFromStabilize:
            Noop,                                                             c3;

            Noop,                                                             c1;
            LODisp,                                                           c2;
            RET[stabilize-return],                                            c3;
```

```
deallocate:
        {otLow 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;
        [] ← otLow LRot0, XDisp, CALL[getClass],                        c3;

        temp1Low ← temp1Low + deltaWordOffset,                          c1, at[startingDeallocate, 10, getClass-return];
        Q ← classCompiledMethodOop,                                     c2;
        [] ← temp3Low xor Q, ZeroBr,                                    c3;

        {get delta word to see if object has pointers}
        MAR ← [temp1High, temp1Low + 0], BRANCH[$, deallocatingACompiledMethod],c1;
        Noop,                                                           c2;
        Q ← MD, XLDisp,                                                 c3;

        BRANCH[deallocateWithNoPointers, deallocateWithPointers, 2],    c1;

deallocateWithNoPointers:
        temp1Low ← temp1Low - deltaWordOffset, L1 ← freeNonPointerObject,       c2;
        uClass ← temp3Low, CALL[adjustLevelsAndReturnToPool],           c3;

        temp3Low ← uClass, GOTO[nowDoObjectsClass],                     c1, at[freeNonPointerObject, 10,
        addToFreeChunkList-return];

deallocatingACompiledMethod:
        GOTO[deallocateWithPointersA],                                  c2;

deallocateWithPointers:
        Noop,                                                           c2;

deallocateWithPointersA:
        temp1Low ← temp1Low + offsetFromDeltaWordToClassField,          c3;

        {enqueue this object for deallocation}
        MAR ← [temp1High, temp1Low + 0],                                c1;
        MDR ← uQueueHead,                                               c2;
        uQueueHead ← otLow,                                             c3;

        Noop,                                                           c1;

nowDoObjectsClass:
        Noop,                                                           c2;
        otLow ← temp3Low 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}

        otLow ← uCurrentObject, XDisp,                                  c1, at[nowDoneWithObject, 10,
        addToFreeChunkList-return];
        BRANCH[continueWithCurrentObject, checkQueueHead, 0e],          c2;


checkQueueHead:
        otLow ← uQueueHead, XDisp,                                      c3;

        BRANCH[startWithQueueHead, noMore, 0e],                         c1;

noMore:
        {all recursive freeing is now complete}
        L3Disp,                                                         c2;
        RET[deallocate-return],                                         c3;


startWithQueueHead:
        uCurrentObject ← otLow, L1 ← sweepingObject,                    c2;
        CALL[otMap2],                                                   c3;

        temp1Low ← temp1Low + deltaWordOffset,                          c1, at[sweepingObject, 10, otMap2-return];
        Q ← temp1High,                                                  c2;
        uSoFarHigh ← Q,                                                 c3;

        MAR ← [temp1High, temp1Low + 0],                                c1;
        temp1Low ← temp1Low - deltaWordOffset,  L2 ← getObjectEndForFreeing,    c2;
        Q ← MD, XDisp,                                                  c3;

        BRANCH[doingACompiledMethod, notDoingACompiledMethod, 0e],      c1;

        {both isCompiledMethod and doesHavePointers live in the lastPointerOf routine}
doingACompiledMethod:
        CALL[isCompiledMethod],                                         c2;
```

```
notDoingACompiledMethod:
        CALL[doesHavePointers],                                                 c2;

        uCurrentObjectBaseLow ← temp1Low,                                       c1, at[getObjectEndForFreeing, 10,
        lastPointerOf-return];
        temp1Low ← temp1Low + classFieldOffset,                                 c2;
        Noop,                                                                   c3;

        MAR ← [temp1High, temp1Low + 0],                                        c1;
        Noop,                                                                   c2;
        temp2Low ← MD {link to next object on queue},                           c3;

        uQueueHead ← temp2Low,                                                  c1;
        Noop,                                                                   c2;
        GOTO[areWeThereYet],                                                    c3;


doAnotherField:
        MAR ← [temp1High, temp1Low + 0],                                        c1;
        Noop,                                                                   c2;
        otLow ← MD,  XDisp, GOTO[specialRefd],                                  c3;


continueWithCurrentObject:
        temp1Low ← uSoFar,                                                      c3;

        temp1High ← uSoFarHigh,                                                 c1;
        temp3Low ← uLastPointer,                                                c2;

        Noop,                                                                   c3;

areWeThereYet:
        [] ← temp1Low xor temp3Low, ZeroBr,                                     c1;
        temp1Low ← temp1Low + 1, BRANCH[$, doneWithObject],                     c2;
        uSoFar ← temp1Low, GOTO[doAnotherField],                                c3;









doneWithObject:
        Noop,                                                                   c3;

        otLow ← uCurrentObject,                                                 c1;
        temp1Low ← uCurrentObjectBaseLow, L1 ← nowDoneWithObject,               c2;
        uCurrentObject ← ~otLow xor otLow, CALL[adjustLevelsAndReturnToPool],   c3;









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 temp1High/Low and L2}

        [] ← 6 - otLow, CarryBr, BRANCH[$, specialSmall, 0e],                   c1;
        BRANCH[doSpecial, $],                                                   c2;
        GOTO[more],                                                             c3;

doSpecial:
        CANCELBR[$, 0f],                                                        c3;


        MAR ← [otHigh, otLow + 1],                                              c1;
        temp1Low ← 0fc {for "subtracting one" from the reference count}, CANCELBR[$, 2], c2;
        Q ← MD, XHDisp {first part of stuck ref count test},                    c3;

        temp1Low ← temp1Low LRot8, BRANCH[specialPositiveRefCount, specialNegativeRefCount, 2], c1;

specialPositiveRefCount: {not stuck but could go to zero}
        Q ← Q + temp1Low {subtract 1}, CarryBr {carry implies already zero, an error}, c2;
        [] ← Q + temp1Low {subtract again}, CarryBr {no carry implies just went tozero},
        BRANCH[triedToSpeicalRefdZeroCountObject, specialUpdateOtRefd], c3;

specialNegativeRefCount: {could be stuck but cannot go to zero}
        temp3Low ← 4,                                                           c2;
        temp3Low ← temp3Low LRot8,                                              c3;

        [] ← Q + temp3Low, CarryBr {carry implies stuck ref count},             c1;
        Q ← Q + temp1Low {subtract one from ref count}, BRANCH[specialNotStuckRefd, specialStuckRefd], c2;

specialNotStuckRefd:
        Xbus ← 1, XDisp {makes the branch at specialUpdateOtRefd happy!},       c3;

specialUpdateOtRefd:
        MAR ← [otHigh, otLow + 1]{write updated refcount}, BRANCH[specialNeedsDeallocation, $], c1;
        MDR ← Q,  LOOPHOLE[wok], CANCELBR[$, 2],                                c2;
```

```
        GOTO[more],                                                     c3;

specialStuckRefd:
        GOTO[more],                                                     c3;

specialNeedsDeallocation:
        MDR ← Q, LOOPHOLE[wok], CANCELBR[$, 2],                         c2;
        temp1Low ← 2,                                                   c3;

        {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}
        temp1Low ← temp1Low LRot8,                                      c1;
        [] ← Q and temp1Low, ZeroBr,                                    c2;
        BRANCH[$, deallocate],                                          c3;

        Noop,                                                           c1;
        Noop,                                                           c2;
        GOTO[more],                                                     c3;

specialSmall:
        CANCELBR[$, 1],                                                 c2;
        GOTO[more],                                                     c3;


triedToSpeicalRefdZeroCountObject:
        BRANCH[triedToSpecialRefdZeroCountObjectA,
                triedToSpecialRefdZeroCountObjectB],                    c1;
triedToSpecialRefdZeroCountObjectA:
        GOTO[bailout3],                                                 c2;
triedToSpecialRefdZeroCountObjectB:
        GOTO[bailout3],                                                 c2;



adjustLevelsAndReturnToPool:
        temp1Low ← temp1Low + sizeFieldOffset,                          c1;
        temp2High ← uRumRecordHigh,                                     c2;
        temp2Low ← uRumRecordLow,                                       c3;

incrementOopLevel:
        MAR ← [temp2High, temp2Low + oopLevelLowOffset],                c1;
        CANCELBR [$, 2],                                                c2;
        temp3Low ← MD,                                                  c3;

        MAR ← [temp2High, temp2Low + oopLevelLowOffset],                c1;
        MDR ← temp3Low ← temp3Low + 1,
              CANCELBR [$, 2], LOOPHOLE [wok], CarryBr,                 c2;
        BRANCH [increaseWordLevelLow, impossibleOopLevel],              c3;

impossibleOopLevel:
        GOTO [bailout2],                                                c1;

increaseWordLevelLow:
        MAR ← [temp1High, temp1Low + 0],                                c1;
        temp1Low ← temp1Low - sizeFieldOffset,                          c2;
        temp3Low ← MD,                                                  c3;

        MAR ← [temp2High, temp2Low + wordLevelLowOffset],               c1;
        CANCELBR [$, 2],                                                c2;
        Q ← MD,                                                         c3;

        MAR ← [temp2High, temp2Low + wordLevelLowOffset],               c1;
        MDR ← Q + temp3Low, CANCELBR [$, 2], LOOPHOLE [wok], CarryBr,   c2;
        BRANCH [returnToPool, wordLevelCarry],                          c3;

wordLevelCarry:
        MAR ← [temp2High, temp2Low + wordLevelHighOffset],              c1;
        CANCELBR [$, 2],                                                c2;
        temp3Low ← MD,                                                  c3;

        MAR ← [temp2High, temp2Low + wordLevelHighOffset],              c1;
        MDR ← temp3Low ← temp3Low + 1, CANCELBR [$, 2], LOOPHOLE [wok], c2;
        GOTO [returnToPool],                                            c3;

returnToPool:
        {build a mask to set all ref count bits on and to set purpose bits to free (11)}
        temp3Low ← 0fc,                                                 c1;
        temp3Low ← temp3Low LRot8,                                      c2;
        temp3Low ← temp3Low or 60,                                      c3;

        MAR ← [otHigh, otLow + 1],                                      c1;
        CANCELBR[$, 2],                                                 c2;
        Q ← MD,                                                         c3;

        MAR ← [otHigh, otLow + 1],                                      c1;
        MDR ← Q or temp3Low, CANCELBR[$, 2], LOOPHOLE[wok],             c2;
        GOTO[addToProperFreeChunkList],                                 c3;
```

```
addToProperFreeChunkList:
        {upon entry, otLow is the oop of the object to add to the free list, temp1High/Low must be that object's base. calls
        addToFreeChunkList thus smashing temp2High/Low and Q. smashes temp3Low}

        temp1Low ← temp1Low + sizeFieldOffset,                          c1;
        Q ← largestFreeChunkSize,                                       c2;
        Noop, .                                                         c3;

        MAR ← [temp1High, temp1Low + 0],                                c1;
        temp1Low ← temp1Low - sizeFieldOffset,                          c2;
        temp3Low ← MD {object's size},                                  c3;

        [] ← temp3Low - Q, CarryBr,                                     c1;
        BRANCH[selectRegularList, selectBigFreeList],                   c2;

selectRegularList:
        GOTO[addToFreeChunkList],                                       c3;

selectBigFreeList:
        temp3Low ← Q, GOTO[addToFreeChunkList],                         c3;
```

```
{
        1-Aug-84 18:27:13
}


{
 inputs - otLow is the oop whose mapping is desired -- better not supply a SmallInteger --,
          L1 is the return linkage register
 output - high portion of address is returned in temp1High, low portion in temp1Low
}


{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,                                          c1;
              BRANCH[goodOop-otMap, badOop-otMap, 0e],                      c2;

      badOop-otMap:
              GOTO[bailout1],                                               c3;

      goodOop-otMap:
              Noop,
 endOtMapDebug!                                                             c3;

              {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],                                    c1;
              CANCELBR[$,2],                                                c2;
              temp1High ← MD, XLDisp {check if this object is a leaf},      c3;

              MAR ← [otHigh, otLow + 0], BRANCH[$,isLeaf-otMap,1],          c1;
              L1Disp {this is not a leaf},                                  c2;
              temp1Low ← MD, RET[otMap-return],                             c3;

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

     isLeaf-otMap:
              GOTO[bailout3],                                               c2;
```

```
{
 inputs - otLow is the oop whose mapping is desired -- better not supply a SmallInteger --,
          L1 is the return linkage register
 output - high portion of address is returned in temp1High, low portion in temp1Low
}


{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,                                          c1;
              BRANCH[goodOop-otMap2, badOop-otMap2, 0e],                    c2;

      badOop-otMap2:
              GOTO[bailout1],                                               c3;

      goodOop-otMap2:
              Noop,
```

```
endOtMapDebug2!                                                          c3;

              {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],                                 c1;
              Noop, CANCELBR[$,2],                                       c2;
              temp1High ← MD, XLDisp {check if this object is a leaf},   c3;

              MAR ← [otHigh, otLow + 0], BRANCH[$,isLeaf-otMap2,1],      c1;
              L1Disp {this is not a leaf},                               c2;
              temp1Low ← MD, RET[otMap2-return],                         c3;


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

     isLeaf-otMap2:
              GOTO[bailout3],                                            c2;
```

```
{
        1-Aug-84 18:28:49
}


    {Pop and Store Receiver Variable bytecodes}


        popAndStoreReceiverVariable0:
                temp3Low ← field0, backupIs0Bytes, GOTO[popAndStoreReceiverVariable],   c1, bytecode[60];

        popAndStoreReceiverVariable1:
                temp3Low ← field1, backupIs0Bytes, GOTO[popAndStoreReceiverVariable],   c1, bytecode[61];

        popAndStoreReceiverVariable2:
                temp3Low ← field2, backupIs0Bytes, GOTO[popAndStoreReceiverVariable],   c1, bytecode[62];

        popAndStoreReceiverVariable3:
                temp3Low ← field3, backupIs0Bytes, GOTO[popAndStoreReceiverVariable],   c1, bytecode[63];

        popAndStoreReceiverVariable4:
                temp3Low ← field4, backupIs0Bytes, GOTO[popAndStoreReceiverVariable],   c1, bytecode[64];

        popAndStoreReceiverVariable5:
                temp3Low ← field5, backupIs0Bytes, GOTO[popAndStoreReceiverVariable],   c1, bytecode[65];

        popAndStoreReceiverVariable6:
                temp3Low ← field6, backupIs0Bytes, GOTO[popAndStoreReceiverVariable],   c1, bytecode[66];

        popAndStoreReceiverVariable7:
                temp3Low ← field7, backupIs0Bytes, GOTO[popAndStoreReceiverVariable],   c1, bytecode[67];



        popAndStoreReceiverVariable:
                uSmashTos ← ~temp2Low xor temp2Low { -1 },  GOTO[storeReceiverVariable],c2;

        storeReceiverVariable:

                {upon entry temp3Low must indicate the receiver field to store into, including the size of the object header and
                uSmashTos must be 0 for no pop, -1 for popandsmash.  Since the receiver has no lambdas, no checks are made for them}

                Noop,                                                                   c3;

                Noop,                                                                   c1;
                temp2Low ← uReceiverLow, L1 ← popAndStoreRecVariable,                   c2;
                temp2High ← uReceiverHigh, temp2Low ← temp2Low + deltaWordOffset, CALL[getDeltaWord], c3;

                [] ← temp1Low, YDisp,                                                   c1, at[popAndStoreRecVariable, 10,
                getDeltaWord-return];
                temp2Low ← uReceiverLow, BRANCH[receiverNotVolatile, receiverVolatile, 0b], c2;

        receiverVolatile:
                {volatile implies no reference counting needed}
                temp2Low ← temp2Low + temp3Low {get to the receiver field}, GOTO[getTosStoreRecVariable], c3;

        receiverNotVolatile:
                {non-volatile -- need to refd old contents, refi new contents.  because of possible Loom calls on the refi/refd calls, we
                do not know that the bytecode will complete until after the calls are complete, and cannot change the stack or receiver
                till then}

                temp2Low ← temp2Low + temp3Low {get to the receiver field},             c3;

                MAR ← [temp2High, temp2Low + 0] {read value from field},                c1;
                Noop,                                                                   c2;
                otLow ← MD, XDisp, CALL[refd] {and refd it},                            c3;

                MAR ← [stackHigh, stackLow + 0],                                        c1, at[popAndStoreRecVariable, 10, refdReturn];
                Noop,                                                                   c2;
                otLow ← MD, XDisp, CALL[refi] {which returns to getTosStoreRecVariable:},      c3;


        getTosStoreRecVariable:

                Noop,                                                                   c1, at[popAndStoreRecVariable, 10, refiReturn];
                Noop,                                                                   c2;
                [] ← uSmashTos, ZeroBr, CALL[returnTopOfStack],                         c3;



        putIntoReceiver:

                MAR ← [temp2High, temp2Low + 0],                                        c1, at[popAndStoreRecVariable,10,
                returnTopOfStack-return];
                MDR ← otLow {store into receiver}, NextBytecode,                        c2;
                DISPNI[bytecodes], ipLow ← ipLow + PC16,                                c3;
```

```
{Pop and Store Temporary Location bytecodes}


    popAndStoreTemporaryLocation0:
        homeLow ← homeLow + temp0, GOTO[popAndStoreTemporary],                  c1, bytecode[68];

    popAndStoreTemporaryLocation1:
        homeLow ← homeLow + temp1, GOTO[popAndStoreTemporary],                  c1, bytecode[69];

    popAndStoreTemporaryLocation2:
        homeLow ← homeLow + temp2, GOTO[popAndStoreTemporary],                  c1, bytecode[6a];

    popAndStoreTemporaryLocation3:
        homeLow ← homeLow + temp3, GOTO[popAndStoreTemporary],                  c1, bytecode[6b];

    popAndStoreTemporaryLocation4:
        homeLow ← homeLow + temp4, GOTO[popAndStoreTemporary],                  c1, bytecode[6c];

    popAndStoreTemporaryLocation5:
        homeLow ← homeLow + temp5, GOTO[popAndStoreTemporary],                  c1, bytecode[6d];

    popAndStoreTemporaryLocation6:
        homeLow ← homeLow + temp6, GOTO[popAndStoreTemporary],                  c1, bytecode[6e];

    popAndStoreTemporaryLocation7:
        homeLow ← homeLow + temp7, GOTO[popAndStoreTemporary],                  c1, bytecode[6f];

    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, ZeroBr, CALL[returnTopOfStack],                         c3;

        MAR ← [homeHigh, homeLow + 0], {start write of context field}           c1, at[storeTemporary, 10,
        returnTopOfStack-return];
        MDR ← otLow,    {write the top of stack},                               c2;
        homeLow ← uHomeLow, {restore the home base register}                    c3;

        Noop,                                                                   c1;
        NextBytecode,                                                           c2;
        DISPNI[bytecodes], ipLow ← ipLow + PC16,                                c3;




{Extended Store (Receiver Variable, Temorary 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 ← ib {get the extension byte},                                 c2;
        temp3Low ← temp2Low and 3f, {get only the offset}                       c3;

        temp3Low ← temp3Low + objectHeaderSize, {and bump to the object body}   c1;
        temp2Low ← RShift1 temp2Low, {shift for convienient dispatch}           c2;
        [] ← temp2Low LRot0, XwdDisp,   {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], backupIs1Byte {in case of refi/refd needs to call Loom}, c2, at[0, 4, extendedStoreTarget];

     extendedStoreTemporary:
        temp3Low ← temp3Low + tempFrameStart,                                   c2, at[1, 4, extendedStoreTarget];
        homeLow ← homeLow + temp3Low, L1 ← storeTemporary,                      c3;

        Noop,                                                                   c1;
```

```
        GOTO[storeTemporary],                                                   c2;


illegalExtendedStore:
        GOTO[illegalExtendedStore],                                             c*, at[2, 4, extendedStoreTarget];

  extendedStoreLiteralVariable:
        temp3Low ← temp3Low + literalStart, backupIs1Byte {in case refi/refd overflow or lambda in association}, c2, at[3, 4,
        extendedStoreTarget];
        temp2Low ← uCurrentMethodLow,                                           c3;

        temp2High ← uCurrentMethodHigh, temp2Low ← temp3Low + temp2Low,  {and add in offset to the appropriate association} c1;
        Noop,                                                                   c2;
        Noop,                                                                   c3;

        MAR ← [temp2High, temp2Low + 0], {read the association oop}             c1;
        L1 ← storingLiteralVariable,                                            c2;
        otLow ← MD, CALL[otMap] {otMap the association},                        c3;

        Q ← temp1High,                                                          c1, at[storingLiteralVariable, 10, otMap-return];
        temp2Low ← temp1Low + associationValueIndex {add in offset to value},   c2;
        temp2High ← Q LRot0,                                                    c3;

        MAR ← [temp2High, temp2Low + 0],                                        c1;
        Noop,                                                                   c2;
        otLow ← MD {get the value of the association},                          c3;

        [] ← otLow, ZeroBr {test for lambda}, L1 ← storingLitVar,               c1;
        BRANCH[valueIsNotLambda-storeLiteralVariable, valueIsLambda-storeLiteralVariable], c2;

valueIsLambda-storeLiteralVariable:
        {Loom: need to call Loom here}
        GOTO[bailout1],                                                         c3;


valueIsNotLambda-storeLiteralVariable:
        [] ← otLow LRot0, XDisp, CALL[refd],                                    c3;

        MAR ← [stackHigh, stackLow + 0],                                        c1, at[storingLitVar, 10, refdReturn];
        Noop,                                                                   c2;
        otLow ← MD, XDisp, CALL[refi],                                          c3;

        Noop,                                                                   c1, at[storingLitVar, 10, refiReturn];
        Noop,                                                                   c2;
        [] ← uSmashTos, ZeroBr, CALL[returnTopOfStack],                         c3;

        MAR ← [temp2High, temp2Low + 0],                                        c1, at[storingLitVar, 10,
        returnTopOfStack-return];
        MDR ← otLow, NextBytecode,                                              c2;
        DISPNI[bytecodes], ipLow ← ipLow + PC16,                                c3;




   {Pop Stack Top}

popStackTop:
        MAR ← [stackHigh, stackLow + 0],                                        c1, bytecode[87];
        MDR ← nilPointer {smash top of stack},                                  c2;
        Noop,                                                                   c3;

        stackLow ← stackLow - 1,                                                c1;
        NextBytecode,                                                           c2;
        DISPNI[bytecodes], ipLow ← ipLow + PC16,                                c3;
```

```
{
        9-Jul-84 18:12:00
}


    primitiveIndexNotZero:
        uPrimitiveNumber ← temp2Low {save in case of primitive failure}, CANCELBR[$, 0f], c1;
        Noop,                                                                          c2;
        [] ← temp2Low LRot12, XDisp {dispatch on high 4 bits of primitive #},          c3;

        [] ← temp2Low LRot0, XDisp {dispatch on low 4 bits}, DISP4[primitiveBank], c1;

        DISP4[bank0],                                                                  c2, at[0, 10, primitiveBank];
        DISP4[bank1],                                                                  c2, at[1, 10, primitiveBank];
        DISP4[bank2],                                                                  c2, at[2, 10, primitiveBank];
        DISP4[bank3],                                                                  c2, at[3, 10, primitiveBank];
        DISP4[bank4],                                                                  c2, at[4, 10, primitiveBank];
        DISP4[bank5],                                                                  c2, at[5, 10, primitiveBank];
        DISP4[bank6],                                                                  c2, at[6, 10, primitiveBank];
        DISP4[bank7],                                                                  c2, at[7, 10, primitiveBank];
        DISP4[bank8],                                                                  c2, at[8, 10, primitiveBank];
        DISP4[bank9],                                                                  c2, at[9, 10, primitiveBank];
        DISP4[banka],                                                                  c2, at[0a, 10, primitiveBank];
        DISP4[bankb],                                                                  c2, at[0b, 10, primitiveBank];
        DISP4[bankc],                                                                  c2, at[0c, 10, primitiveBank];
        DISP4[bankd],                                                                  c2, at[0d, 10, primitiveBank];
        DISP4[banke],                                                                  c2, at[0e, 10, primitiveBank];
        DISP4[bankf],                                                                  c2, at[0f, 10, primitiveBank];

    somethingIsDefinitelyWrong:
        GOTO[somethingIsDefinitelyWrong],                                              c*, at[0, 10, bank0];
        GOTO[sendArithmeticMessage0],                                                  c3, at[1, 10, bank0];
        GOTO[sendArithmeticMessage1],                                                  c3, at[2, 10, bank0];
        GOTO[sendArithmeticMessage2],                                                  c3, at[3, 10, bank0];
        GOTO[sendArithmeticMessage3],                                                  c3, at[4, 10, bank0];
        GOTO[sendArithmeticMessage4],                                                  c3, at[5, 10, bank0];
        GOTO[sendArithmeticMessage5],                                                  c3, at[6, 10, bank0];
        GOTO[sendArithmeticMessage6],                                                  c3, at[7, 10, bank0];
        GOTO[sendArithmeticMessage7],                                                  c3, at[8, 10, bank0];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[9, 10, bank0];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0a, 10, bank0];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0b, 10, bank0];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0c, 10, bank0];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0d, 10, bank0];
        GOTO[sendArithmeticMessage14],                                                 c3, at[0e, 10, bank0];
        GOTO[sendArithmeticMessage15],                                                 c3, at[0f, 10, bank0];

        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[1, 10, bank1];
        GOTO[sendArithmeticMessage11],                                                 c3, at[2, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[3, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[4, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[5, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[6, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[7, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[8, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[9, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0a, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0b, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0c, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0d, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0e, 10, bank1];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0f, 10, bank1];

        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[1, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[2, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[3, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[4, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[5, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[6, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[7, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[8, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[9, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0a, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0b, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0c, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0d, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0e, 10, bank2];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0f, 10, bank2];

        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[0, 10, bank3];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[1, 10, bank3];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[2, 10, bank3];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[3, 10, bank3];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[4, 10, bank3];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[5, 10, bank3];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[6, 10, bank3];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[7, 10, bank3];
        GOTO[noSuchPrimitiveInMicrocode],                                              c3, at[8, 10, bank3];
```

```
        GOTO[noSuchPrimitiveInMicrocode],                                   c3, at[9, 10, bank3];
        GOTO[noSuchPrimitiveInMicrocode],                                   c3, at[0a, 10, bank3];
        GOTO[noSuchPrimitiveInMicrocode],                                   c3, at[0b, 10, bank3];

primitiveAt:
        {uNewReceiverHigh/Low must be valid, receiver must not be smallInt}

        Xbus ← uNewReceiver, XDisp, L3 ← atPrim,                            c3, at[0c, 10, bank3];

        MAR ← [stackHigh, stackLow + 0] {read index}, BRANCH[$, primAtFailA, 0e], c1;
        temp3High ← primitiveAt,                                            c2;
        temp3Low ← MD, XDisp, CALL[positive16BitValueOf],                   c3;

        Q ← Q + objectHeaderSize, BRANCH[$, atFailedA],                     c1, at[primitiveAt, 10,
        positive16BitValueOf-return];
        uAtIndex ← Q, CALL[getObjectSize] {which establishes Q as the size of the object}, c2;

        temp1Low ← temp1Low + classFieldOffset,                             c1, at[atPrim, 10, getObjectSize-return];


commonAt:
        L2 ← classInPrimAt,                                                 c2;
        otLow ← uNewReceiversClass, CALL[fixedFieldsOf],                    c3;

commonAtAlreadyHaveFixedFields:
        Ybus ← temp3Low LRot4, XDisp, {test words bit of instanceSpec},     c2, at[classInPrimAt, 10, fixedFieldsOf-return];
        temp1Low ← uAtIndex, BRANCH[indexingBytes, indexingWords, 0b],      c3;

indexingBytes:
        L2 ← atPrim,                                                        c1;
        temp1Low ← temp1Low + objectHeaderSize, L3Disp,                     c2;
        uAtIndex ← temp1Low, BRANCH[$, returnToAtPut, 0e],                  c3;

        temp1High ← uNewReceiverHigh, c1;
        Noop,                                                               c2;
        temp1Low ← uNewReceiverLow, CALL[getByteOrAddress],                 c3;

        temp3Low ← LShift1 temp3Low, SE ← 1, GOTO[atSmash],                 c3, at[atPrim, 10, getByteOrAddress-return];

indexingWords:
        temp1Low ← temp1Low {index + objectheadersize} + temp2Low {fixed fields}, c1;
        [] ← Q - temp1Low, CarryBr {valid index?},                          c2;
        temp2Low ← uNewReceiverLow, BRANCH[atRangeError, $],                c3;

        temp1High ← uNewReceiverHigh,                                       c1;
        temp1Low ← temp1Low + temp2Low, L3Disp,                             c2;
        temp1Low ← temp1Low - 1, RET[commonAt-return],                      c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1, at[atPrim, 10, commonAt-return];
        Ybus ← temp3Low LRot4, XDisp,                                       c2;
        temp3Low ← MD, BRANCH[isWordObject, atSmash, 7],                    c3;

atSmash:
        MAR ← [stackHigh, stackLow + 0],                                    c1;
        MDR ← nilPointer,                                                   c2;
        stackLow ← stackLow - 1, GOTO[pushTemp3LowAndDispatch],             c3;


isWordObject:
        GOTO[bytecodeFailed],                                               c1;



primAtFailA:
        Noop,                                                               c2;
        GOTO[activateNewMethod],                                            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 Molasses. if not, fail the primitive and run the method}

        temp2High ← uRumRecordHigh,                                         c1;
        temp2Low ← uRumRecordLow,                                           c2;
        Noop,                                                               c3;

        MAR ← [temp2High, temp2Low + displayBitmapOopOffset],               c1;
        CANCELBR[$, 2],                                                     c2;
        Q ← MD,                                                             c3;

        MAR ← [temp2High, temp2Low + cursorBitmapOopOffset],                c1;
        [] ← Q xor uNewReceiver, ZeroBr, CANCELBR[$, 2],                    c2;
        Q ← MD, BRANCH[$, atFailedC],                                       c3;

        [] ← Q xor uNewReceiver, ZeroBr,                                    c1;


        Q ← uAtIndex, BRANCH[$, atFailedD],                                 c2;
        GOTO[activateNewMethod],                                            c3;

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

```
atFailedC:
        GOTO[bytecodeFailed];                                                   c1;

atFailedD:
        GOTO[atFailedC],                                                        c3;



getObjectSize:

        temp1High ← uNewReceiverHigh,                                           c3;

        temp1Low ← uNewReceiverLow,                                             c1;
        temp1Low ← temp1Low + sizeFieldOffset,                                  c2;
        Noop,                                                                   c3;

        MAR ← [temp1High, temp1Low + 0],                                        c1;
        temp1Low ← uNewReceiverLow, L3Disp,                                     c2;
        Q ← MD {size},  RET[getObjectSize-return],                              c3;



primitiveAtPut:
        {uNewReciever High/low must be valid, receiver must not be smallInt}
        temp3High ← primitiveAtPut,                                  c3, at[0d, 10, bank3];


        MAR ← [stackHigh, stackLow + 0], L3 ← atPutPrim,                        c1;
        stackLow ← stackLow - 1,                                                c2;
        temp2Low ← MD {the value},                                              c3;

        MAR ← [stackHigh, stackLow + 0],                                        c1;
        stackLow ← stackLow + 1,                                                c2;
        temp3Low ← MD {the index}, XDisp, CALL[positive16BitValueOf],           c3;

        uAtPutValue ← temp2Low, BRANCH[$, atPutFail],                           c1, at[primitiveAtPut, 10,
        positive16BitValueOf-return];
        Q ← Q + objectHeaderSize,                                               c2;
        uAtIndex ← Q,                                                           c3;

        Xbus ← uNewReceiver, XDisp,{test for smallInt}                          c1;
        BRANCH[ {CALL} getObjectSize, primAtPutFailA, 0e],                      c2;


        temp1Low ← temp1Low + classFieldOffset, CALL[commonAt],                 c1, at[atPutPrim, 10, getObjectSize-return];

        Noop,                                                                   c1, at[atPutPrim, 10, commonAt-return];
        Noop,                                                                   c2;
        Noop,                                                                   c3;

returnToAtPut:
        Ybus ← temp3Low LRot4, XDisp,                                           c1;
        uAtPutLow ← temp1Low, DISP4[atPutTable, 3],                             c2;


        {storing a pointer}
        temp1Low ← uNewReceiverLow,                                             c3, at[0f, 10, atPutTable];

        temp1Low ← temp1Low + deltaWordOffset,                                  c1;
        Noop,                                                                   c2;
        Noop,                                                                   c3;

        MAR ← [temp1High, temp1Low + 0] {read delta word to determine volatility}, c1;
        temp1Low ← uAtPutLow,                                                   c2;
        Ybus ← MD, XDisp,                                                       c3;

        MAR ← [temp1High, temp1Low + 0] {get old contents, and refd it}, BRANCH[$, atPutIsVolatile, 0b], c1;
        Noop,                                                                   c2;
        otLow ← MD, XDisp, CALL[refd],  L1 ← viaPrimitiveAtPut,                 c3;

        temp1High ← uNewReceiverHigh,                                           c1, at[viaPrimitiveAtPut, 10, refdReturn];
        temp1Low ← uAtPutLow,                                                   c2;
        Noop,                                                                   c3;

        MAR ← [temp1High, temp1Low + 0],                                        c1;
        MDR ← uAtPutValue,                                                      c2;
        otLow ← uAtPutValue, XDisp, CALL[refi],                                 c3;



atPutIsVolatile:
        MDR ← uAtPutValue, LOOPHOLE[mdrok],                                     c2;
```

```
atPutWrapup:

        Noop,                                                                   c3;

        MAR ← [stackHigh, stackLow + 0] {smash value on stack},                 c1, at[viaPrimitiveAtPut, 10, refiReturn];
        MDR ← nilPointer,                                                       c2;
        stackLow ← stackLow - 1,                                                c3;

        temp3Low ← uAtPutValue,                                                 c1;
        Noop,                                                                   c2;
        GOTO[atSmash],                                                          c3;


improperInstanceSpec:
        GOTO[improperInstanceSpec],                                             c*, at[0b, 10, atPutTable];

        {storing a word}
        Noop,                                                                   c3, at[ 7, 10, atPutTable];

        GOTO[bytecodeFailed],                                                   c1;

        {storing a byte}
        Noop,                                                                   c3, at[ 3, 10, atPutTable];

        GOTO[bytecodeFailed],                                                   c1;


atPutFail:
        Noop,                                                                   c2;
primAtPutFailA:

        GOTO[activateNewMethod],                                                c3;




primitiveSize:

        Xbus ← uNewReceiver, XDisp,                                             c3, at[0e, 10, bank3];

        BRANCH[$,sizeFailC, 0e],                                                c1;
        L3 ← sizePrim,                                                          c2;
        otLow ← uNewReceiversClass, L2 ← forSizePrim, CALL[fixedFieldsOf],      c3;

        CALL[getObjectSize],                                                    c2, at[forSizePrim, 10, fixedFieldsOf-return];

        [] ← temp3Low LRot4, XDisp, {on the instance spec},                     c1, at[sizePrim, 10, getObjectSize-return];
        temp3Low ← Q - objectHeaderSize, DISP4[sizeTable, 3],                   c2;

        {pointers}
        temp3Low ← temp3Low {size - objectHeader} - temp2Low {fixed fields},
                NegBr, GOTO[tryForSmall],                                       c3, at[0f, 10, sizeTable];

        {invalid}
badInstanceSpec:
        GOTO[badInstanceSpec],                                                  c*, at[0b, 10, sizeTable];

        {words--could be current display bitmap}
        temp2High ← uRumRecordHigh,                                             c3, at[7, 10, sizeTable];

        temp2Low ← uRumRecordLow,                                               c1;
        Noop,                                                                   c2;
        Noop,                                                                   c3;

        MAR ← [temp2High, temp2Low + displayBitmapOopOffset],                   c1;
        CANCELBR[$, 2],                                                         c2;
        Q ← MD,                                                                 c3;

        [] ← Q xor uNewReceiver, ZeroBr,                                        c1;
        BRANCH[$, sizeFailA],                                                   c2;
        [] ← temp3Low, NegBr, GOTO[tryForSmall],                                c3;


sizeFailB:
        CANCELBR[$, 3],                                                         c2;
sizeFailA:
        Noop,                                                                   c3;

        GOTO[bytecodeFailed],                                                   c1;

sizeFailC:
        Noop,                                                                   c2;
```

```
        GOTO[activateNewMethod],                                        c3;

        {bytes}
        temp1Low ← temp1Low + deltaWordOffset,                          c3, at[3, 10, sizeTable];

        MAR ← [temp1High, temp1Low + 0],                                c1;
        temp3Low ← temp3Low + temp3Low,  PgCrOvDisp,                    c2;
        Ybus ← MD, XDisp {test odd bit}, BRANCH[$, byteSizeFail, 2],    c3;

        BRANCH[sizeEven, sizeOdd, 0d],                                  c1;

sizeEven:
        GOTO[byteSizeOk],                                               c2;

sizeOdd:
        temp3Low ← temp3Low - 1, GOTO[byteSizeOk],                      c2;

byteSizeOk:
        Noop,                                                           c3;

tryForSmall:

        temp3Low ← temp3Low + temp3Low + 1, PgCrOvDisp, BRANCH[$, sizeFailB],   c1;
        BRANCH[sizeOk, sizeFail, 2],                                    c2;
sizeOk:
        GOTO[pushTemp3LowAndDispatch],                                  c3;

sizeFail:

        Noop,                                                           c3;

byteSizeFail:
        CANCELBR[bytecodeFailed, 0f],                                   c1;
```

```
primitiveStringAt:
        {uNewReceiverHigh/Low must be valid}

        Noop,                                                           c3, at[0f, 10, bank3];


        MAR ← [stackHigh, stackLow + 0] {read index}, L3 ← stringAtPrim,        c1;
        temp3High ← primitiveStringAt,                                  c2;
        temp3Low ← MD, XDisp, CALL[positive16BitValueOf],               c3;

        Q ← Q + twiceObjectHeaderSize, BRANCH[$, stringAtFailA], L2 ← stringAtPrim, c1, at[primitiveStringAt, 10,
        positive16BitValueOf-return];
        uAtIndex ← Q, CALL[getObjectSize],                              c2;
        {the call on getObjectSize returns directly to getByteOrAddress, which in turn returns directly to
         the following instruction:}

         Noop,                                                          c3, at[stringAtPrim, 10,
         getByteOrAddress-return];

         Noop,                                                          c1;
         L1 ←  primStringAt,                                            c2;
         otLow ← characterTableOop, CALL[otMap],                        c3;

         temp1Low ← temp1Low + objectHeaderSize,                        c1, at[primStringAt, 10, otMap-return];
         temp1Low ← temp1Low + temp3Low,                                c2;
         Noop,                                                          c3;

         MAR ← [temp1High, temp1Low + 0],                               c1;
         Noop,                                                          c2;
         temp3Low ← MD, GOTO[atSmash],                                  c3;
```

```
getByteOrAddress:
        temp3Low ← LShift1 Q {object size}, SE ← 0, {yields object size in bytes}, c1, at[stringAtPrim, 10,
        getObjectSize-return];
        Q ← uAtIndex,                                                   c2;
        temp1Low ← temp1Low + deltaWordOffset,                          c3;

        MAR ← [temp1High, temp1Low + 0] {read the instances delta word},        c1;
        temp1Low ← uNewReceiverLow,                                     c2;
        Ybus ← MD, XDisp {and dispatch so that we can test its odd bit},        c3;
```

```
        Q ← Q - 1, {zero adjust}, BRANCH[stringLengthIsEven, stringLengthIsOdd, 0d], c1;

stringLengthIsEven:
        GOTO[stringOnward],                                                    c2;

stringLengthIsOdd:
        temp3Low ← temp3Low - 1, GOTO[stringOnward],                           c2;

stringOnward:
        temp2Low ← uAtIndex {valid index?},                                    c3;

        [] ← temp3Low - temp2Low, CarryBr,                                     c1;
        temp2Low ← RShift1 Q, SE ← 0 {yields word offset}, BRANCH[stringAtFailB, $], c2;
        temp1Low← temp1Low + temp2Low, L2Disp,                                 c3;

        MAR ← [temp1High, temp1Low + 0] {read correct word from object}, BRANCH[$, gettingByteAddress, 0e], L2Disp, c1;
        Ybus ← Q, CANCELBR[$, 0f], YDisp {which byte do we want},              c2;
        temp3Low ← MD, BRANCH[fetchLeftByte, fetchRightByte, 0e],              c3;

fetchLeftByte:
        temp3Low ← temp3Low LRot8, L2Disp, GOTO[stringAnd],                    c1;

fetchRightByte:
        GOTO[stringAnd], L2Disp,                                               c1;

stringAnd:
        temp3Low ← temp3Low and 0ff, RET[getByteOrAddress-return],             c2;


gettingByteAddress:
        RET[getByteOrAddress-return],                                          c2;




stringAtFailA:
        Noop,                                                                  c2;

stringAtFailB:
        GOTO[activateNewMethod],                                               c3;




primitiveStringAtPut:
        {uNewReceiverHigh/Low must be valid}

        temp3High ← primitiveStringAtPut,                                      c3, at[0, 10, bank4];

        MAR ← [stackHigh, stackLow + 0],                                       c1;
        L2 ← primStringAtPut,                                                  c2;
        otLow ← MD {the value}, XDisp, CALL[getClass],                         c3;

        Q ← classCharacterOop,                                                 c1, at[primStringAtPut, 10, getClass-return];
        [] ← temp3Low xor Q, ZeroBr,                                           c2;
        uAtPutValue ← otLow, BRANCH[stringAtPutFailA, $],                      c3;

        temp1Low ← temp1Low + firstFieldOfObject,                              c1;
        Noop,                                                                  c2;
        Noop,                                                                  c3;

        MAR ← [temp1High, temp1Low + 0],                                       c1;
        stackLow ← stackLow - 1,                                               c2;
        temp2Low ← MD {the ascii value of the character},                      c3;

        MAR ← [stackHigh, stackLow + 0] {read index}, L3 ← stringAtPutPrim,    c1;
        stackLow ← stackLow + 1,                                               c2;
        temp3Low ← MD, XDisp, CALL[positive16BitValueOf],                      c3;

        Q ← Q + twiceObjectHeaderSize, BRANCH[$, stringAtPutFailB],            c1, at[primitiveStringAtPut, 10,
        positive16BitValueOf-return];
        uAtIndex ← Q,                                                          c2;
        temp2Low ← RShift1 temp2Low, SE ← 0,                                   c3;

        L2 ← stringAtPutPrim,                                                  c1;
        uAtPutLow ← temp2Low, CALL[getObjectSize],                             c2;

        Noop,                                                                  c1, at[stringAtPutPrim, 10,
        getObjectSize-return];
        Noop,                                                                  c2;
        CALL[getByteOrAddress],                                                c3;

        temp2Low ← uAtPutLow,                                                  c3, at[stringAtPutPrim, 10,
        getByteOrAddress-return];
```

```
        MAR ← [temp1High, temp1Low + 0] {read correct word},                    c1;
        Ybus ← Q, YDisp {which byte?},                                          c2;
        temp3Low ← MD, BRANCH[keepRightByte, keepLeftByte, 0e],                 c3;

keepLeftByte:
        temp3Low ← temp3Low LRot8, GOTO[stringAtPutAnd],                        c1;

keepRightByte:
        GOTO[stringAtPutAnd],                                                   c1;

stringAtPutAnd:
        temp3Low ← temp3Low and 0ff,                                            c2;
        temp2Low ← temp2Low LRot8,                                              c3;

        Ybus ← Q, YDisp {need to rotate?},                                      c1;
        temp3Low ← temp3Low or temp2Low, BRANCH[atPutOk, atPutRotate, 0e],      c2;

atPutOk:
        GOTO[stringAtPutWrite],                                                 c3;

atPutRotate:

        temp3Low ← temp3Low LRot8, GOTO[stringAtPutWrite],                      c3;


stringAtPutWrite:
        MAR ← [temp1High, temp1Low + 0],                                        c1;
        MDR ← temp3Low, GOTO[atPutWrapup],                                      c2;




stringAtPutFail1A:
        GOTO[stringAtPutFail1B],                                                c1;

stringAtPutFail1B:
        Noop,                                                                   c2;
        GOTO[activateNewMethod],                                                c3;









        GOTO[noSuchPrimitiveInMicrocode],                                       c3, at[1, 10, bank4];
        GOTO[noSuchPrimitiveInMicrocode],                                       c3, at[2, 10, bank4];
        GOTO[noSuchPrimitiveInMicrocode],                                       c3, at[3, 10, bank4];




primitiveObjectAt:
        Xbus ← uNewReceiver, XDisp, {smallints are a no-no}                     c3, at[4, 10, bank4];

        MAR ← [stackHigh, stackLow + 0], BRANCH[$, objectAtFail1A, 0e],         c1;
        temp3High ← primitiveObjectAt,                                          c2;
        temp3Low ← MD, XDisp, CALL[positive16BitValueOf],                       c3;

        Q ← Q + objectHeaderSize, BRANCH[$, objectAtFail1B],                    c1, at[primitiveObjectAt, 10,
        positive16BitValueOf-return];
        uAtIndex ← Q,                                                           c2;
        temp1High ← uNewReceiverHigh,                                           c3;

        temp1Low ← uNewReceiverLow,                                             c1;
        temp1Low ← temp1Low + objectHeaderSize,                                 c2;
        temp2Low ← 0 {no fixed fields for commonAt},                            c3;

        MAR ← [temp1High, temp1Low + 0],                                        c1;
        temp3Low ← RRot1 (1) {tell commonAt code that we're dealing with pointers}, c2;
        temp1Low ← MD,                                                          c3;

        temp1Low ← RShift1 (temp1Low and 7f), SE ← 0, {extract literal count},  c1;
        Q ← temp1Low + objectHeaderSize + 1{yields max legal index including header}, L3 ← atPrim {tell commonAt to fall thru,
        not return}, c2;
        temp1Low ← uAtIndex {argument + objectheadersize},  GOTO[indexingWords],c3;
```

```
objectAtFailA:
        GOTO[objectAtFail],                                                     c2;

objectAtFailB:
        Noop,                                                                   c2;

objectAtFail:
        GOTO[activateNewMethod],                                                c3;

        GOTO[noSuchPrimitiveInMicrocode],                                       c3, at[5, 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 ← MD, XDisp, CALL[positive16BitValueOf],                       c3;

        stackLow ← stackLow - 1 {point at class to instantiate}, CANCELBR[$, 1], c1, at[primitiveNew, 10,
        positive16BitValueOf-return];
        L2 ← newPrimitive,                                                      c2;
        L3Disp,                                                                 c3;

primitiveNewCommon:
        {read class oop 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:
        otLow ← MD, CALL[fixedFieldsOf],                                        c3;

        {now use the instanceSpec twice -- once to test for indexability, once for which kind of object to create}
        temp3Low ← temp3Low LRot4 {rotate instanceSpec for dispatch}, L3Disp {args/noargs 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[newOka, newFaila, 0d], c1;

newArgs:
        BRANCH[newFailb, newOkb, 0d],                                           c1;

newOka: {new without argument}
        temp3Low ← temp2Low {the fixed size}, DISP4[whichFlavor, 3],            c2;

newOkb: {new with argument}
        temp3High ← viaPrimitiveNew {return linkage for instantiation},         c2;
        Noop,                                                                   c3;

        [] ← temp3Low, YDisp {for type of object to create},                    c1;
        temp3Low ← temp2Low {fixed size} + Q {variable size}, DISP4[whichFlavor, 3], c2;


        uClassToInstantiate ← otLow, CALL[createInstanceWithBytes],             c3, at[3, 10, whichFlavor];

        uClassToInstantiate ← otLow, CALL[createInstanceWithWords],             c3, at[7, 10, whichFlavor];

        uClassToInstantiate ← otLow, CALL[createInstanceWithPointers],          c3, at[0f, 10, whichFlavor];


        temp3Low ← otLow {oop of the new object}, L3Disp,                       c1, at[viaPrimitiveNew, 10,
        createInstance-return];
        BRANCH[newWithout, newWith, 0e],                                        c2;

newWithout: {replace class to instantiate with new object}
```

```
            GOTO[pushTemp3LowAndDispatch],                                  c3;

        newWith: {nil out requested size, then replace class to instantiate with new object}
            Noop,                                                           c3;

            MAR ← [stackHigh, stackLow + 0],                                c1;
            MDR ← nilPointer,                                               c2;
            stackLow ← stackLow - 1 {point at class field}, GOTO[pushTemp3LowAndDispatch], c3;


        newFail1a:
            CANCELBR[newFailed, 0f],                                        c2;

        newFail1b:
            Noop,                                                           c2;

        newFailed:
            {failed because of argument problems.  run the method}
            GOTO[activateNewMethod],                                        c3;



            GOTO[noSuchPrimitiveInMicrocode],                               c3, at[8, 10, bank4];







        primitiveInstVarAt:
            Xbus ← uNewReceiver, XDisp, L3 ← instVarAtPrim,                 c3, at[9, 10, bank4];

            MAR ← [stackHigh, stackLow + 0], BRANCH[$, instVarAtFail1A, 0e], c1;
            temp3High ← primitiveInstVarAt,                                 c2;
            temp3Low ← MD, XDisp, CALL[positive16BitValueOf],               c3;

            Q ← Q + objectHeaderSize, BRANCH[$, instVarAtFail1B],           c1, at[primitiveInstVarAt, 10,
            positive16BitValueOf-return];
            uAtIndex ← Q, CALL[getObjectSize],                              c2;

            temp1Low ← temp1Low + classFieldOffset,                         c1, at[instVarAtPrim, 10, getObjectSize-return];
            L2 ← forInstVarAtPrim,                                          c2;
            otLow ← uNewReceiversClass, CALL[fixedFieldsOf],                c3;

            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,                                                           c3;

            GOTO[commonAtAlreadyHaveFixedFields],                           c1;


        instVarAtFail1A:
            GOTO[instVarAtFail],                                            c2;

        instVarAtFail1B:
            Noop,                                                           c2;

        instVarAtFail:
            Noop,                                                           c3;

            GOTO[bytecodeFailed],                                           c1;









            GOTO[noSuchPrimitiveInMicrocode],                               c3, at[0a, 10, bank4];
            GOTO[noSuchPrimitiveInMicrocode],                               c3, at[0b, 10, bank4];
            GOTO[noSuchPrimitiveInMicrocode],                               c3, at[0c, 10, bank4];









        primitiveFirstInstance:
            temp2Low ← uNewReceiver {get the class we are looking for instances of}, c3, at[0d, 10, bank4];

            otLow ← 0 {search entire ot},                                   c1;

        findOne:
```

```
            Noop,                                                                   c2;

        considerNextA:
            otLow ← otLow + 2 {next ot entry}, CarryBr,                             c3;

            MAR ← [otHigh, otLow + 1] {read object table entry}, BRANCH[$, noMoreInstances],    c1;
            CANCELBR[$, 2],                                                         c2;
            temp1High ← MD, XwdDisp,                                                c3;

            MAR ← [otHigh, otLow + 0], DISP2[findInstance],                         c1;

            {in use}
            L2 ← lookingForInstances,                                               c2, at[0, 4, findInstance];
            temp1Low ← MD, CALL[getClassAlreadyHaveBase],                           c3;


            [] ← temp2Low xor temp3Low, ZeroBr {correct class?},                    c1, at[lookingForInstances, 10, getClass-return];
            BRANCH[considerNextA, $],                                               c2;
            temp3Low ← otLow, GOTO[pushTemp3LowAndDispatch] {yes, return oop},      c3;

            {forwarding block}
        forwardingBlock:
            GOTO[forwardingBlock],                                                  c*, at[1, 4, findInstance];

        beingRecursivelyFreed:
            GOTO[forwardingBlock],                                                  c*, at[2, 4, findInstance];

            GOTO[considerNextA],                                                    c2, at[3, 4, findInstance];



        noMoreInstances:
            CANCELBR[$, 2],                                                         c2;
            GOTO[activateNewMethod],                                                c3;




        primitiveNextInstance:
            temp2Low ← uNewReceiversClass,                                          c3, at[0e, 10, bank4];

            otLow ← uNewReceiver, GOTO[findOne],                                    c1;




            GOTO[noSuchPrimitiveInMicrocode],                                       c3, at[0f, 10, bank4];

        primitiveBlockCopy:
            Noop,                                                                   c3, at[0, 10, bank5];
            MAR ← [stackHigh, stackLow + 0],                                        c1;
            stackLow ← stackLow - 1,                                                c2;
            temp1Low ← MD {argument count},                                         c3;

            temp1High ← uNewReceiverHigh,                                           c1;
            temp1Low ← uNewReceiverLow,                                             c2;
            otLow ← uNewReceiver {context oop},                                     c3;

        primitiveBlockCopyViaDirectDispatch:

            uSaveHome ← otLow,                                                      c1;
            temp1Low ← temp1Low + methodFieldOffset,                                c2;
            stackLow ← stackLow + 1 {in case of instantiation failure},             c3;

            MAR ← [temp1High, temp1Low + 0],                                        c1;
            temp1Low ← temp1Low - methodFieldOffset,                                c2;
            Ybus ← MD, XDisp,                                                       c3;

            temp3High ← viaBlockCopy, BRANCH[moreBlockCopy, blockCopyOfBlockContext, 0e], c1;

        blockCopyOfBlockContext:
            temp1Low ← temp1Low + homeFieldOffset,                                  c2;
            Noop,                                                                   c3;

            MAR ← [temp1High, temp1Low + 0], L1 ← blockCopyOfBlock,                 c1;
            Noop,                                                                   c2;
            otLow ← MD {oop of home context}, CALL[otMap2],                         c3;

            uSaveHome ← otLow,                                                      c1, at[blockCopyOfBlock, 10, otMap2-return];

        moreBlockCopy:
            temp1Low ← temp1Low + sizeFieldOffset,                                  c2;
            otLow  ← blockContextClassOop,                                          c3;
```

```
        MAR ← [temp1High, temp1Low + 0],                                    c1;
        uClassToInstantiate ← otLow,                                        c2;
        temp3Low ← MD {size of new block context},                          c3;

        temp3Low ← temp3Low - objectHeaderSize,                             c1;
        Noop,                                                               c2;
        CALL[createInstanceWithPointers],                                   c3;

        Noop,                                                               c1, at[viaBlockCopy, 10, createInstance-return];
        Noop,                                                               c2;
        temp3Low ← blockCopying,                                            c3;

        uMakeVolatileLinkage ← temp3Low, CALL[makeVolatile]                 c1;

        temp3Low ← uCurrentMethodLow,                                       c1, at[blockCopying, 10, makeVolatile-return];
        temp3Low ← ipLow - temp3Low, {word relative plus headerSize}        c2;
        temp3Low ← temp3Low - objectHeaderSize {word relative},             c3;

        temp3Low ← LShift1 temp3Low, SE ← pc16{byte offset into compiledMethod. note that the SE←pc16 has toggled pc16, so we
        will toggele it again soon},c1;
        temp3Low ← temp3Low + 4,                                            c2;
        temp3Low ← LShift1 temp3Low, SE ← 1, {yields SmallInt},             c3;

        temp1Low ← temp1Low + instructionPointerFieldOffset,                c1;
        temp2Low ← 0 + PC16 {toggle again},                                 c2;
        Noop,                                                               c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        MDR ← temp3Low,                                                     c2;
        temp1Low ← temp1Low + offsetFromInstructionPointerToStackPointer,   c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        MDR ← 1 {zero as SmallInt},                                         c2;
        temp1Low ← temp1Low + offsetFromStackPointerToArgCount,             c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        MDR ← uArgumentCount,                                               c2;
        temp1Low ← temp1Low + offsetFromArgCountToInitialIP,                c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        MDR ← temp3Low,                                                     c2;
        temp1Low ← temp1Low + offsetFromInitialIPToHome,                    c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        MDR ← uSaveHome,                                                    c2;
        Noop,                                                               c3;

        MAR ← [stackHigh, stackLow + 0],                                    c1;
        MDR ← nilPointer,                                                   c2;
        stackLow ← stackLow - 1,                                            c3;

        temp3Low ← uNewObject,                                              c1;
        Noop,                                                               c2;
        GOTO[pushTemp3LowAndDispatch],                                      c3;


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}
        otLow ← uNewReceiver,                                               c3, at[1, 10, bank6];

primitiveValueViaDirectDispatch:
        Q ← primValue,                                                      c1;
        uMakeVolatileLinkage ← Q,                                           c2;
        Noop,                                                               c3;

        uNewContextOop ← otLow, CALL[makeVolatile],                         c1;

        temp1Low ← temp1Low + senderFieldOffset,                            c1, at[primValue, 10, makeVolatile-return];
        temp3Low ← uArgumentCount,                                          c2;
        temp3Low ← LShift1 temp3Low, SE← 1, {shift for easy compare},       c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        temp1Low ← temp1Low + offsetFromSenderToBlockArgCount,              c2;
        Q ← MD {sender field, it better be nil},                            c3;

        MAR ← [temp1High, temp1Low + 0],                                    c1;
        [] ← Q xor nilPointer, ZeroBr,                                      c2;
        temp2Low ←  MD {block argument count, a SmallInt}, BRANCH[blockAlreadyActive, $], c3;

        [] ← temp2Low xor temp3Low, ZeroBr {verify that arg counts match},  c1;
        temp2Low ← uArgumentCount, BRANCH[valueArgCountMismatch, $],        c2;
        stackLow ← stackLow - temp2Low {point at blockContext oop},         c3;

        MAR ← [stackHigh, stackLow + 0] {smash block context oop in active context}, c1;
        MDR ← nilPointer,                                                   c2;
        temp2Low ← stackLow + temp2Low {last word to move},                 c3;

        stackLow ← stackLow + 1 {first word to move},                       c1;
        temp1Low ← temp1Low + offsetFromBlockArgumentCountToFirstTemp {first destination word}, L1 ← primitiveValue, c2;
        CALL[transferWords],                                                c3;

        Q ← uArgumentCount,                                                 c1, at[primitiveValue, 10, transferWords-return];
        stackLow ← stackLow - Q - 1,                                        c2;
        Noop,                                                               c3;
```

```
            temp1Low ← uMakeVolatileLow,                                                    c1;
            temp1Low ← temp1Low + initialInstructionPointerOffset,                          c2;
            Noop,                                                                           c3;

            MAR ← [temp1High, temp1Low + 0] {read initial ip},                              c1;
            temp1Low ← temp1Low - offsetFromInitialIpToIp,                                  c2;
            Q ← MD {the initial instruction pointer},                                       c3;

            MAR ← [temp1High, temp1Low + 0] {write ip},                                     c1;
            MDR ← Q,                                                                        c2;
            temp1Low ← temp1Low + offsetFromIpToStackPointer,                               c3;

            MAR ← [temp1High, temp1Low + 0] {write stackPointer},                           c1;
            MDR ← temp3Low,                                                                 c2;
            temp1Low ← temp1Low - offsetFromSenderToStackPointer,                           c3;

            MAR ← [temp1High, temp1Low + 0] {write caller},                                 c1;
            MDR ← uActiveContextOop,                                                        c2;
            temp2High ← uRumRecordHigh,                                                     c3;

            temp2Low ← uRumRecordLow,                                                       c1;
            Noop,                                                                           c2;
            Noop,                                                                           c3;

            MAR ← [temp2High, temp2Low + leafContextOopOffset],                             c1;
            CANCELBR[$, 2],                                                                 c2;
            temp3Low ← MD {oop of leaf context},                                            c3;


            Noop,                                                                           c1;
            [] ← temp3Low xor uActiveContextOop, ZeroBr,                                    c2;
            BRANCH[valueNotLeaf, $],                                                        c3;

            MAR ← [temp2High, temp2Low + leafContextOopOffset],                             c1;
            MDR ← nilPointer, L1 ← viaPrimitiveValue, CANCELBR[$, 2], LOOPHOLE[wok],        c2;
            otLow ← temp3Low LRot0, XDisp, CALL[refd],                                      c3;

    valueNotLeaf:
            Noop,
            GOTO[newActiveContext],                                                         c1, at[viaPrimitiveValue, 10, refdReturn];
                                                                                            c2;



    blockAlreadyActive:
            GOTO[bytecodeFailed],                                                           c1;

    valueArgCountMismatch:
            GOTO[blockAlreadyActive],                                                       c3;









            GOTO[noSuchPrimitiveInMicrocode],                                               c3, at[2, 10, bank5];




    primitivePerform:
            Q ← uArgumentCount,                                                             c3, at[3, 10, bank5];

            temp2High ← uActiveContextHigh {point at perform receiver},                     c1;
            temp2Low ← stackLow - Q,                                                        c2;
            Q ← Q - 1,                                                                      c3;

            MAR ← [temp2High, temp2Low + 0] {read perform receiver}, L2 ← primitivePerform, c1;
            temp2Low ← temp2Low + 1,                                                        c2;
            otLow ← MD, XDisp, CALL[getClass],                                              c3;

            MAR ← [temp2High, temp2Low + 0] {read selector to be performed}, L3 ← 1 {this is a lookup for perform}, c1,
            at[primitivePerform, 10, getClass-return];
            uArgumentCount ← Q,                                                             c2;
            Q ← MD, CALL[startMethodLookup],                                                c3;


            {now determine the arg count of method we are to perform}
            [] ← temp1Low {methodHeader} LRot4, XDisp,                                      c1,at[ 1, 10, performOrExecute-return];
            Q ← uArgumentCount, DISP4[performFlagTable,1],                                  c2;

            {flag = 0 .. 6}

            GOTO[noArgs],                                                                   c3, at[1, 10, performFlagTable];

            temp3Low ← 1, GOTO[performArgCountCheck],                                       c3, at[3, 10, performFlagTable];
```

```
            temp3Low ← 2, GOTO[performArgCountCheck],                          c3, at[5, 10, performFlagTable];

            temp3Low ← 3, GOTO[performArgCountCheck],                          c3, at[7, 10, performFlagTable];

            temp3Low ← 4, GOTO[performArgCountCheck],                          c3, at[9, 10, performFlagTable];

            temp3Low ← 0, GOTO[performArgCountCheck],                          c3, at[0b, 10, performFlagTable];

            temp3Low ← 0, GOTO[performArgCountCheck],                          c3, at[0d, 10, performFlagTable];

            temp3Low ← RShift1 (temp1Low and 7f), SE ← 0, {extract literal count}  c3, at[0f, 10, performFlagTable];

            otLow ← uNewMethodLow,                                             c1;
            temp3High ← uNewMethodHigh,                                        c2;
            temp3Low ← temp3Low + otLow,                                       c3;

            temp3Low ← temp3Low + objectHeaderSize,                            c1;
            temp3Low ← temp3Low - 1,                                           c2;
            Noop,                                                              c3;

            MAR ← [temp3High, temp3Low + 0] {read method header extension},    c1;
            Noop,                                                              c2;
            temp3Low ← MD,                                                     c3;

            temp3Low ← temp3Low LRot8,                                         c1;
            temp3Low ← RShift1 (temp3Low and 0ff), SE ← 0,                     c2;
            GOTO[performArgCountCheck],                                        c3;

    performArgCountCheck:
            [] ← temp3Low xor Q, ZeroBr,                                       c1;
            temp1High ← uActiveContextHigh, BRANCH[performFailA, $],           c2;
            temp1Low ← stackLow - Q, {temp1High/Low is now destination high/low}  c3;

            uPrimitiveNumber ← temp2Low, ZeroBr, L1 ← performPrim,             c1;
            temp2Low ← stackLow {source limit}, BRANCH[performFailC, $],       c2;
            stackLow ← temp1Low + 1 {source start}, CALL[transferWords],       c3;



    noArgs:
            [] ← Q, ZeroBr, {is send argcount also 0?}                         c1;
            uPrimitiveNumber ← temp2Low, BRANCH[performFailB, $],              c2;
            Noop,                                                              c3;

            MAR ← [stackHigh, stackLow + 0] {smash perform selector},          c1;
            MDR ← nilPointer,                                                  c2;
            Noop,                                                              c3;


            stackLow ← stackLow - 1,                                           c1, at[performPrim, 10, transferWords-return];
            temp2Low ← uPrimitiveNumber,                                       c2;
            temp1Low ← uNewMethodHeader, GOTO[executeNewMethodViaPrimitivePerform], c3;



    performFailA:
            Noop,                                                              c3;

            GOTO[bytecodeFailed],                                              c1;

    performFailB:
            Noop,                                                              c3;

            GOTO[bytecodeFailed],                                              c1;

    performFailC:
            Noop,                                                              c3;

            GOTO[bytecodeFailed],                                              c1;









            GOTO[noSuchPrimitiveInMicrocode],                                  c3, at[4, 10, bank5];
            GOTO[noSuchPrimitiveInMicrocode],                                  c3, at[5, 10, bank5];
            GOTO[noSuchPrimitiveInMicrocode],                                  c3, at[6, 10, bank5];
            GOTO[noSuchPrimitiveInMicrocode],                                  c3, at[7, 10, bank5];
            GOTO[noSuchPrimitiveInMicrocode],                                  c3, at[8, 10, bank5];
            GOTO[noSuchPrimitiveInMicrocode],                                  c3, at[9, 10, bank5];
            GOTO[noSuchPrimitiveInMicrocode],                                  c3, at[0a, 10, bank5];
            GOTO[noSuchPrimitiveInMicrocode],                                  c3, at[0b, 10, bank5];
            GOTO[noSuchPrimitiveInMicrocode],                                  c3, at[0c, 10, bank5];
            GOTO[noSuchPrimitiveInMicrocode],                                  c3, at[0d, 10, bank5];
            GOTO[noSuchPrimitiveInMicrocode],                                  c3, at[0e, 10, bank5];
            GOTO[noSuchPrimitiveInMicrocode],                                  c3, at[0f, 10, bank5];
```

```
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[1, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[2, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[3, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[4, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[5, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[6, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[7, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[8, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[9, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0a, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0b, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0c, 10, bank6];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0d, 10, bank6];

GOTO[sendSpecialMessage6] { == },                c3, at[0e, 10, bank6];

GOTO[noSuchPrimitiveInMicrocode],                c3, at[0f, 10, bank6];

GOTO[noSuchPrimitiveInMicrocode],                c3, at[0, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[1, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[2, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[3, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[4, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[5, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[6, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[7, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[8, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[9, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0a, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0b, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0c, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0d, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0e, 10, bank7];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0f, 10, bank7];


GOTO[noSuchPrimitiveInMicrocode],                c3, at[0, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[1, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[2, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[3, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[4, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[5, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[6, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[7, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[8, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[9, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0a, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0b, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0c, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0d, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0e, 10, bank8];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0f, 10, bank8];

GOTO[noSuchPrimitiveInMicrocode],                c3, at[0, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[1, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[2, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[3, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[4, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[5, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[6, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[7, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[8, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[9, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0a, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0b, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0c, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0d, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0e, 10, bank9];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0f, 10, bank9];

GOTO[noSuchPrimitiveInMicrocode],                c3, at[0, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[1, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[2, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[3, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[4, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[5, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[6, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[7, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[8, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[9, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0a, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0b, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0c, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0d, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0e, 10, banka];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[0f, 10, banka];

GOTO[noSuchPrimitiveInMicrocode],                c3, at[0, 10, bankb];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[1, 10, bankb];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[2, 10, bankb];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[3, 10, bankb];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[4, 10, bankb];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[5, 10, bankb];
GOTO[noSuchPrimitiveInMicrocode],                c3, at[6, 10, bankb];
```

```
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[7, 10, bankb];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[8, 10, bankb];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[9, 10, bankb];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0a, 10, bankb];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0b, 10, bankb];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0c, 10, bankb];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0d, 10, bankb];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0e, 10, bankb];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0f, 10, bankb];


        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[1, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[2, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[3, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[4, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[5, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[6, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[7, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[8, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[9, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0a, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0b, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0c, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0d, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0e, 10, bankc];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0f, 10, bankc];

        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[1, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[2, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[3, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[4, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[5, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[6, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[7, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[8, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[9, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0a, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0b, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0c, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0d, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0e, 10, bankd];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0f, 10, bankd];

        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[1, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[2, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[3, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[4, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[5, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[6, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[7, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[8, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[9, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0a, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0b, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0c, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0d, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0e, 10, banke];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0f, 10, banke];


        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[1, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[2, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[3, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[4, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[5, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[6, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[7, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[8, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[9, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0a, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0b, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0c, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0d, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0e, 10, bankf];
        GOTO[noSuchPrimitiveInMicrocode],                    c3, at[0f, 10, bankf];


noSuchPrimitiveInMicrocode:

        {see if molasses implements this primitive}
        temp1High ← uRumRecordHigh,                          c1;
        temp1Low ← uRumRecordLow,                            c2;
        temp2Low ← uPrimitiveNumber,                         c3;

        {get the address of the primitive map}
        MAR ← [temp1High, temp1Low + primitiveMapLowOffset], c1;
        CANCELBR[$, 2],                                      c2;
        temp3Low ← MD,                                       c3;
```

```
            MAR ← [temp1High, temp1Low + primitiveMapHighOffset],                    c1;
            temp3Low ← temp3Low + temp2Low {add in primitive number}, CANCELBR[$, 2], c2;
            temp3High ← MD,                                                          c3;

            {read primitive map entry}
            MAR ← [temp3High, temp3Low + 0],                                         c1;
            Noop,                                                                    c2;
            temp1Low ← MD,                                                           c3;

            [] ← temp1Low, ZeroBr,                                                   c1;
            BRANCH[molassesDoesImplement, molassesDoesNotImplement]                  c2;

    molassesDoesNotImplement:
            GOTO[activateNewMethod],                                                 c3;

    molassesDoesImplement:
            Noop,                                                                    c3;

            GOTO[bytecodeFailed],                                                    c1;




    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}

            temp2Low ← 0f, L1 ← instanceSpec,                                        c1;
            temp2Low ← temp2Low LRot8,                                               c2;
            temp2Low ← temp2Low or 0ff, CALL[otMap2],                                c3;

            temp1Low ← temp1Low +  instanceSpecificationFieldOffset,                 c1, at[instanceSpec, 10, otMap2-return];
            Noop,                                                                    c2;
            Noop,                                                                    c3;

            MAR ← [temp1High, temp1Low + 0],                                         c1;
            temp1Low ← temp1Low - instanceSpecificationFieldOffset,                  c2;
            temp3Low ← MD {the instanceSpec}, L2Disp,                                c3;

            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,  temp3Low 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 in Q}

            {use pending dispatch to see if SmallInteger or whether furthur work is needed}

            temp3Low ← RShift1 temp3Low, SE ← 0, BRANCH[pos16NotSmall, pos16IsSmall, 0e], c1;

    pos16IsSmall:
            {just turn it into a useful number and return}
            Ybus ← temp3High, XDisp, GOTO[returnFromPositive16BitValueOf],           c2;

    pos16NotSmall:
            Q ← classLargePositiveIntegerPointer, L2 ← pos16Bit,                     c2;
            otLow ← LShift1 temp3Low, CALL[getClass],                                c3;

            [] ← temp3Low xor Q, ZeroBr {is class LargePositiveInteger?},            c1, at[pos16Bit, 10, getClass-return];
            BRANCH[pos16Failа, $],                                                   c2;

            temp1Low ← temp1Low +  sizeFieldOffset,                                  c3;

            MAR ← [temp1High, temp1Low + 0],                                         c1;
            Q ← sizeOfLargeIntegerForPositive16BitValueOf,                           c2;
            temp3Low ← MD {size of this object},                                     c3;

            [] ← temp3Low  - Q, ZeroBr,                                              c1;
            temp1Low ← temp1Low - sizeFieldOffset,  BRANCH[pos16Failb, $],           c2;

            temp1Low ← temp1Low + firstFieldOfObject,                                c3;

            MAR ← [temp1High, temp1Low + 0],                                         c1;
            Noop,                                                                    c2;
            temp3Low ← MD,                                                           c3;
```

```
            temp3Low ← temp3Low LRot8,                                  c1;
            Ybus ← temp3High, XDisp,                                    c2;

    returnFromPositive16BitValueOf:
            Q ← temp3Low, ZeroBr, RET[positive16BitValueOf-return],     c3;



    pos16Fail1a:
            GOTO[activateNewMethod],                                    c3;
    pos16Fail1b:
            GOTO[activateNewMethod],                                    c3;
```

{

1-Aug-84 18:30:14

}

bytecodes:

{Push Receiver Variable bytecodes}

```
pushReceiverVariable0:
        temp3Low ← field0, GOTO[pushReceiverVariable],                    c1, bytecode[0];

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[0a];

pushReceiverVariable11:
        temp3Low ← field11, GOTO[pushReceiverVariable],                   c1, bytecode[0b];

pushReceiverVariable12:
        temp3Low ← field12, GOTO[pushReceiverVariable],                   c1, bytecode[0c];

pushReceiverVariable13:
        temp3Low ← field13, GOTO[pushReceiverVariable],                   c1, bytecode[0d];

pushReceiverVariable14:
        temp3Low ← field14, GOTO[pushReceiverVariable],                   c1, bytecode[0e];

pushReceiverVariable15:
        temp3Low ← field15, GOTO[pushReceiverVariable],                   c1, bytecode[0f];

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,                                          c2;
        temp2High ← uReceiverHigh, temp2Low ← temp3Low + temp2Low,        c3;

        MAR ← [temp2High, temp2Low + 0] {read the reciever's field},      c1;
        stackLow ← stackLow + 1,                                          c2;
        temp3Low ← MD, GOTO[pushTemp3LowAndDispatch],                     c3;
```

{Push Temporary Location bytecodes}

```
pushTemporaryLocation0:
        homeLow ← homeLow + temp0, GOTO[pushTemporary],                   c1, bytecode[10];
```

```
        pushTemporaryLocation1:
                homeLow ← 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[1a];

        pushTemporaryLocation11:
                homeLow ← homeLow + temp11, GOTO[pushTemporary],                c1, bytecode[1b];

        pushTemporaryLocation12:
                homeLow ← homeLow + temp12, GOTO[pushTemporary],                c1, bytecode[1c];

        pushTemporaryLocation13:
                homeLow ← homeLow + temp13, GOTO[pushTemporary],                c1, bytecode[1d];

        pushTemporaryLocation14:
                homeLow ← homeLow + temp14, GOTO[pushTemporary],                c1, bytecode[1e];

        pushTemporaryLocation15:
                homeLow ← homeLow + temp15, GOTO[pushTemporary],                c1, bytecode[1f];

        pushTemporary:
                 Noop,                                                          c2;

        pushTemporaryViaExtendedPush:
                {upon entry, homeLow must contain the low-order 16-bits of the absoulute address of the temporary field to be
                loaded.
                no lambda checking is performed. At exit, homeLow is properly restored.}                    Since we're push

                Noop,                                                           c3;

                MAR ← [homeHigh, homeLow + 0] {read the temporary},             c1;
                stackLow ← stackLow + 1,                                        c2;
                temp3Low ← MD,                                                  c3;

                MAR ← [stackHigh, stackLow + 0],                                c1;
                MDR ← temp3Low, NextBytecode, homeLow ← uHomeLow,               c2;
                DISPNI[bytecodes], ipLow ← ipLow + PC16,                        c3;

    {push Literal Constant bytecodes}

        pushLiteralConstant0:
                temp3Low ← literalField0, GOTO[pushLiteralConstant],            c1, bytecode[20];

        pushLiteralConstant1:
                temp3Low ← literalField1, GOTO[pushLiteralConstant],            c1, bytecode[21];

        pushLiteralConstant2:
                temp3Low ← literalField2, GOTO[pushLiteralConstant],            c1, bytecode[22];

        pushLiteralConstant3:
                temp3Low ← literalField3, GOTO[pushLiteralConstant],            c1, bytecode[23];

        pushLiteralConstant4:
                temp3Low ← literalField4, GOTO[pushLiteralConstant],            c1, bytecode[24];

        pushLiteralConstant5:
                temp3Low ← literalField5, GOTO[pushLiteralConstant],            c1, bytecode[25];

        pushLiteralConstant6:
                temp3Low ← literalField6, GOTO[pushLiteralConstant],            c1, bytecode[26];

        pushLiteralConstant7:
                temp3Low ← literalField7, GOTO[pushLiteralConstant],            c1, bytecode[27];

        pushLiteralConstant8:
                temp3Low ← literalField8, GOTO[pushLiteralConstant],            c1, bytecode[28];

        pushLiteralConstant9:
                temp3Low ← literalField9, GOTO[pushLiteralConstant],            c1, bytecode[29];
```

```
        pushLiteralConstant10:
                temp3Low ← literalField10, GOTO[pushLiteralConstant],                   c1, bytecode[2a];

        pushLiteralConstant11:
                temp3Low ← literalField11, GOTO[pushLiteralConstant],                   c1, bytecode[2b];

        pushLiteralConstant12:
                temp3Low ← literalField12, GOTO[pushLiteralConstant],                   c1, bytecode[2c];

        pushLiteralConstant13:
                temp3Low ← literalField13, GOTO[pushLiteralConstant],                   c1, bytecode[2d];

        pushLiteralConstant14:
                temp3Low ← literalField14, GOTO[pushLiteralConstant],                   c1, bytecode[2e];

        pushLiteralConstant15:
                temp3Low ← literalField15, GOTO[pushLiteralConstant],                   c1, bytecode[2f];

        pushLiteralConstant16:
                temp3Low ← literalField16, GOTO[pushLiteralConstant],                   c1, bytecode[30];

        pushLiteralConstant17:
                temp3Low ← literalField17, GOTO[pushLiteralConstant],                   c1, bytecode[31];

        pushLiteralConstant18:
                temp3Low ← literalField18, GOTO[pushLiteralConstant],                   c1, bytecode[32];

        pushLiteralConstant19:
                temp3Low ← literalField19, GOTO[pushLiteralConstant],                   c1, bytecode[33];

        pushLiteralConstant20:
                temp3Low ← literalField20, GOTO[pushLiteralConstant],                   c1, bytecode[34];

        pushLiteralConstant21:
                temp3Low ← literalField21, GOTO[pushLiteralConstant],                   c1, bytecode[35];

        pushLiteralConstant22:
                temp3Low ← literalField22, GOTO[pushLiteralConstant],                   c1, bytecode[36];

        pushLiteralConstant23:
                temp3Low ← literalField23, GOTO[pushLiteralConstant],                   c1, bytecode[37];

        pushLiteralConstant24:
                temp3Low ← literalField24, GOTO[pushLiteralConstant],                   c1, bytecode[38];

        pushLiteralConstant25:
                temp3Low ← literalField25, GOTO[pushLiteralConstant],                   c1, bytecode[39];

        pushLiteralConstant26:
                temp3Low ← literalField26, GOTO[pushLiteralConstant],                   c1, bytecode[3a];

        pushLiteralConstant27:
                temp3Low ← literalField27, GOTO[pushLiteralConstant],                   c1, bytecode[3b];

        pushLiteralConstant28:
                temp3Low ← literalField28, GOTO[pushLiteralConstant],                   c1, bytecode[3c];

        pushLiteralConstant29:
                temp3Low ← literalField29, GOTO[pushLiteralConstant],                   c1, bytecode[3d];

        pushLiteralConstant30:
                temp3Low ← literalField30, GOTO[pushLiteralConstant],                   c1, bytecode[3e];

        pushLiteralConstant31:
                temp3Low ← literalField31, GOTO[pushLiteralConstant],                   c1, bytecode[3f];


        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],    c1, bytecode[40];

        pushLiteralVariable1:
                temp3Low ← literalField1, backupIs0Bytes, GOTO[pushLiteralVariable],    c1, bytecode[41];

        pushLiteralVariable2:
                temp3Low ← literalField2, backupIs0Bytes, GOTO[pushLiteralVariable],    c1, bytecode[42];
```

```
pushLiteralVariable3:
        temp3Low ← literalField3, backupIsOBytes, GOTO[pushLiteralVariable],    c1, bytecode[43];

pushLiteralVariable4:
        temp3Low ← literalField4, backupIsOBytes, GOTO[pushLiteralVariable],    c1, bytecode[44];

pushLiteralVariable5:
        temp3Low ← literalField5, backupIsOBytes, GOTO[pushLiteralVariable],    c1, bytecode[45];

pushLiteralVariable6:
        temp3Low ← literalField6, backupIsOBytes, GOTO[pushLiteralVariable],    c1, bytecode[46];

pushLiteralVariable7:
        temp3Low ← literalField7, backupIsOBytes, GOTO[pushLiteralVariable],    c1, bytecode[47];

pushLiteralVariable8:
        temp3Low ← literalField8, backupIsOBytes, GOTO[pushLiteralVariable],    c1, bytecode[48];

pushLiteralVariable9:
        temp3Low ← literalField9, backupIsOBytes, GOTO[pushLiteralVariable],    c1, bytecode[49];

pushLiteralVariable10:
        temp3Low ← literalField10, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[4a];

pushLiteralVariable11:
        temp3Low ← literalField11, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[4b];

pushLiteralVariable12:
        temp3Low ← literalField12, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[4c];

pushLiteralVariable13:
        temp3Low ← literalField13, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[4d];

pushLiteralVariable14:
        temp3Low ← literalField14, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[4e];

pushLiteralVariable15:
        temp3Low ← literalField15, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[4f];

pushLiteralVariable16:
        temp3Low ← literalField16, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[50];

pushLiteralVariable17:
        temp3Low ← literalField17, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[51];

pushLiteralVariable18:
        temp3Low ← literalField18, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[52];

pushLiteralVariable19:
        temp3Low ← literalField19, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[53];

pushLiteralVariable20:
        temp3Low ← literalField20, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[54];

pushLiteralVariable21:
        temp3Low ← literalField21, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[55];

pushLiteralVariable22:
        temp3Low ← literalField22, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[56];

pushLiteralVariable23:
        temp3Low ← literalField23, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[57];

pushLiteralVariable24:
        temp3Low ← literalField24, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[58];

pushLiteralVariable25:
        temp3Low ← literalField25, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[59];

pushLiteralVariable26:
        temp3Low ← literalField26, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[5a];

pushLiteralVariable27:
        temp3Low ← literalField27, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[5b];

pushLiteralVariable28:
        temp3Low ← literalField28, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[5c];

pushLiteralVariable29:
        temp3Low ← literalField29, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[5d];

pushLiteralVariable30:
        temp3Low ← literalField30, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[5e];

pushLiteralVariable31:
        temp3Low ← literalField31, backupIsOBytes, GOTO[pushLiteralVariable],   c1, bytecode[5f];


pushLiteralVariable:
          {upon entry, temp3Low must be the offset to the oop 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}
        temp2Low ← uCurrentMethodLow, {get the address of the current method}   c2;
        temp2High ← uCurrentMethodHigh, temp2Low ← temp3Low + temp2Low,  {and add in offset to the appropriate   association} c3;

        MAR ← [temp2High, temp2Low + 0], {read the association oop}             c1;
```

```
            L1 ← pushingLiteralVariable,                                        c2;
            otLow ← MD, CALL[otMap] {otMap the association},                    c3;

            temp1Low ← temp1Low + associationValueIndex {add in offset to value},  c1, at[pushingLiteralVariable, 10, otMap-return];
            Noop,                                                               c2;
            Noop,                                                               c3;

            MAR ← [temp1High, temp1Low + 0],                                    c1;
            Noop,                                                               c2;
            temp3Low ← MD {get the value of the association},                   c3;

            [] ← temp3Low, ZeroBr,  {test for lambda}                           c1;
            BRANCH[valueIsNotLambda-pushLiteralVariable, valueIsLambda-pushLiteralVariable],      c2;

    valueIsNotLambda-pushLiteralVariable:
            stackLow ← stackLow + 1 {not a lambda, ok to push}, GOTO[pushTemp3LowAndDispatch], c3;


    valueIsLambda-pushLiteralVariable:
            GOTO[bailout1],                                                     c3;
            {Loom: need to call loom to map association lambda}
```

{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,                                                               c2;
            stackLow ← stackLow + 1, GOTO[pushTemp3LowAndDispatch],             c3;
```

{Extended Push (Receiver Variable, Temporary Location, Literal Constant, Literal Variable)}

```
    extendedPush:
            temp2Low ← ib {get the extension byte},                             c1, bytecode[80];
            temp3Low ← temp2Low and 3f, {and save only the offset}              c2;
            temp3Low ← temp3Low + objectHeaderSize, {and bump to the object body}  c3;

            temp2Low ← RShift1 temp2Low, {shift for convienient dispatch}       c1;
            [] ← temp2Low LRot0, XwdDisp,   {dispatch on the type of extended push} c2;
            DISP2[extendedPushTarget], ipLow ← ipLow + 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[pushLiteralConstant],      c1, at[2, 4, extendedPushTarget];

            backupIs1Byte {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},          c1, bytecode[88];
        stackLow ← stackLow + 1,                                           c2;
        temp3Low ← MD, GOTO[pushTemp3LowAndDispatch],                      c3;


{Push Active Context}

    pushActiveContext:
        Q ← uRumRecordHigh {retreive the Rum Record address},              c1, bytecode[89];
        temp2High ← Q LRot0,                                               c2;
        temp2Low ← uRumRecordLow,                                          c3;

        MAR ← [temp2High, temp2Low + leafContextOopOffset],                c1;
        CANCELBR[$, 2],                                                    c2;
        temp1Low ← MD {oop of the leafContext},                            c3;

        otLow ← uActiveContextOop,      L1 ← pushingActiveContext,         c1;
        [] ← temp1Low - otLow, ZeroBr,                                     c2;
        BRANCH[leafContextIsNotActiveContext, leafContextIsActiveContext], c3;

    leafContextIsActiveContext:
        MAR ← [temp2High, temp2Low + leafContextOopOffset],                c1;
        MDR ← nilPointer, CANCELBR[$, 2], LOOPHOLE[wok],                   c2;
        [] ← otLow LRot0, XDisp {not really necessary--it better be an oop}, CALL[refd], c3;

    leafContextIsNotActiveContext:
        temp3Low ← otLow,                                                  c1, at[pushingActiveContext, 10, refdReturn];
        stackLow ← stackLow + 1,                                           c2;
        GOTO[pushTemp3LowAndDispatch],                                     c3;
```

```
{
        1-Aug-84 18:31:51
}
```

{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:

IB state	Mesa Interrupt Pending	No Mesa Interrupt Pending
full	trap to location 600	branch to next macroinstruction interpreter
empty	trap to location 600	trap to location 400
not empty	trap to location 700	trap to location 500

If a Mesa 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 500 (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 Mesa.df};

stEmpty:
        { when we arrive here, the buffer is empty, and the ip is pointing at the word to be fetched}
        MAR ← [ipHigh, ipLow+0],                                        c1, at[400];
        ipLow ← ipLow + 1,                                              c2;
        IB ← MD, GOTO[nextWord-stNotEmpty],                                     c3;

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,                                              c1, at[500];
        Ybus ← uTimeToStabilize, ZeroBr,                                c2;
        BRANCH[stabilizeNow, nextWord-stNotEmpty],                      c3;

stabilizeNow:
        L0 ← uCodeRequestedStabilization,                               c1;
        Noop,                                                           c2;
        CALL[stabilize],                                                c3;

        otLow ← uActiveContextOop,                                      c1, at[uCodeRequestedStabilization, 10,
        stabilize-return];
        temp3Low ← activeAfterStabilize,                                c2;
        Noop,                                                           c3;

        uMakeVolatileLinkage ← temp3Low, CALL[makeVolatile],            c1;

        temp2High ← uRumRecordHigh,                                     c1, at[activeAfterStabilize, 10,
        makeVolatile-return];
        temp2Low ← uRumRecordLow,                                       c2;
        Noop,                                                           c3;

        MAR ← [temp2High, temp2Low + homeContextOopOffset],             c1;
```

```
            temp3Low ← homeAfterStabilize,  CANCELBR[$, 2],                    c2;
            otLow ← MD,                                                        c3;

            uMakeVolatileLinkage ← temp3Low, CALL[makeVolatile],               c1;

            temp2High ← uRumRecordHigh,                                        c1, at[homeAfterStabilize, 10,
            makeVolatile-return];
            temp2Low ← uRumRecordLow,                                          c2;
            Noop,                                                              c3;

            MAR ← [temp2High, temp2Low + leafContextOopOffset],                c1;
            temp3Low ← leafAfterStabilize, CANCELBR[$, 2],                     c2;
            otLow ← MD,                                                        c3;

            uMakeVolatileLinkage ← temp3Low, CALL[makeVolatile],               c1;

            MAR ← [ipHigh, ipLow+0], GOTO[nextWord-stNotEmpty-c2],             c1, at[leafAfterStabilize, 10,
            makeVolatile-return];


    nextWord-stNotEmpty:
            MAR ← [ipHigh, ipLow+0],                                           c1;

    nextWord-stNotEmpty-c2:
            ipLow ← ipLow - 1, {adjust to point at the proper word}, AlwaysIBDisp, L3 ← 0, c2;
            IB ← MD, DISPNI[bytecodes],                                        c3;



    stEmptyOrFullAndMesaInterrupt:
            GOTO[mesaInterrupt],                                               c1, at[600];

    stNotEmptyAndMesaInterrupt:
            Noop,                                                              c1, at[700];
    mesaInterrupt:
            Noop,                                                              c2;
            temp1Low ← 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:
                Q ←  ErrnIBnStkp,                                              c1, at[0];
        FatalErrorSpin:
                GOTO[bailout3],                                                c2;
```

```
{
        3-Apr-84 21:20:25
}


    {Return (Receiver, true, false, nil)}

        returnReceiver:
                temp3Low ← uReceiverOop, GOTO[ETPhoneHome],                          c1, bytecode[78];
        returnTrue:
                temp3Low ← truePointer, GOTO[ETPhoneHome],                           c1, bytecode[79];
        returnFalse:
                temp3Low ← falsePointer, GOTO[ETPhoneHome],                          c1, bytecode[7a];,
        returnNil:
                temp3Low ← nilPointer,GOTO[ETPhoneHome]                              c1, bytecode[7b];


    {Return Stack Top From (Message, Block)}


        returnStackTopFromMessage:
                MAR ← [stackHigh, stackLow + 0],                                     c1, bytecode[7c];
                MDR ← nilPointer,                                                    c2;
                temp3Low ← MD,                                                       c3;

                stackLow ← stackLow - 1,                                             c1;

        ETPhoneHome:
                {set up temp2 so that we can get the sender oop from the home context}
                Q ← homeHigh,   backupIs0Bytes,                                      c2;
                temp2High ← Q LRot0,                                                 c3;

                temp2Low ← homeLow, GOTO[getSender]                                  c1;


        returnStackTopFromBlock:
                MAR ← [stackHigh, stackLow + 0], backupIs0Bytes,                     c1, bytecode[7d];
                MDR ← nilPointer,                                                    c2;
                temp3Low ← MD,                                                       c3;

                stackLow ← stackLow - 1,                                             c1;
                {set up temp2 so that we can get the sender oop from the active context}
                temp2High ← uActiveContextHigh,                                      c2;
                temp2Low ← uActiveContextLow,                                        c3;

                Noop,                                                                c1;

        getSender:
                temp2Low ← temp2Low + senderFieldOffset,                             c2;
                uReturnValue ← temp3Low,                                             c3;

                MAR ← [temp2High, temp2Low + 0], L1 ← getSenderBase,                 c1;
                temp3High ← uRumRecordHigh,                                          c2;
                otLow ← MD {oop of sender}, CALL[otMap2] {to get the base of the sender context}, c3;


        returnValue:
                {upon entry otLow 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}
                [] ← otLow xor nilPointer, ZeroBr,                                   c1, at[getSenderBase, 10, otMap2-return];
                temp1Low ← temp1Low + instructionPointerFieldOffset, BRANCH[$, cannotReturn],   c2;
                Noop,                                                                c3;

                MAR ← [temp1High, temp1Low + 0] {read instruction pointer},          c1;
                temp3Low ← uRumRecordLow,                                            c2;
                Q ← MD {instruction pointer of context to return to},                c3;

                {if the instruction pointer is nil, also cannot return}
                [] ← Q xor nilPointer, ZeroBr,                                       c1;
                temp2Low ← uActiveContextLow, BRANCH[$, cannotReturnB],              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;

                ipHigh ← Q LRot0,                                                    c1;
                ipLow ← temp1Low,                                                    c2;

                homeLow ← otLow,                                                     c3;

                {get the oop of the leaf}
```

```
            MAR ← [temp3High, temp3Low + leafContextOopOffset],                    c1;
            temp2Low ← temp2Low + senderFieldOffset, CANCELBR[$, 2],               c2;
            temp3Low ← MD {leaf oop},                                              c3;

            {and see if active context is the leaf}
            [] ← uActiveContextOop xor temp3Low, ZeroBr,                           c1;
            temp2High ← uActiveContextHigh, BRANCH[notReturningFromLeaf, returningFromLeaf], c2;
            {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 insides 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?}                    c3;

            MAR ← [temp2High, temp2Low + 0], BRANCH[$, returnWrapup]               c1;
            MDR ← nilPointer, {smash a field}                                      c2;
            temp2Low ← temp2Low + 1,                                               c3;

            Noop,                                                                  c1;
            GOTO[returningFromLeaf],                                               c2;

    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 activeContext}

            temp2Low ← temp2Low + differenceBetweenSenderAndMethodFields,          c3;

            MAR ← [temp2High, temp2Low + 0], {read method field}                   c1;
            temp2Low ← uActiveContextLow,                                          c2;
            temp3Low ← MD, XDisp,                                                  c3;

            BRANCH[smashTwo, $, 0e],                                               c1;

            {this is a blockContext, so we look up the sender/caller chain and see if we find the context we are returning to}
            temp1High ← uActiveContextHigh,                                        c2;
            temp1Low ← uActiveContextLow,                                          c3;

    contextChainChase:
            [] ← otLow xor nilPointer, ZeroBr,                                     c1, at[chasingContextChain, 10, otMap2-return];
            [] ← otLow xor homeLow, ZeroBr, BRANCH[$, notOnChain],                 c2;
            temp1Low ← temp1Low + senderFieldOffset, BRANCH[$, isOnChain],         c3;

            MAR ← [temp1High, temp1Low + 0] {read sender from this context},       c1;
            L1 ← chasingContextChain,                                              c2;
            otLow ← MD {next sender field}, CALL[otMap2],                          c3;

    isOnChain:
            otLow ← uActiveContextOop,                                             c1;

    invalidateContexts:
            L1 ← invalidatingContext,                                              c2;
            CALL[otMap2],                                                          c3;

            temp1Low ← temp1Low + instructionPointerFieldOffset,                   c1, at[invalidatingContext, 10, otMap2-return];
            Noop,                                                                  c2;
            Noop,                                                                  c3;

            MAR ← [temp1High, temp1Low + 0] {smash inst ptr field},                c1;
            MDR ← nilPointer,                                                      c2;
            temp1Low ← temp1Low - differenceBetweenSenderAndInstructionPointerFields, c3;

            MAR ← [temp1High, temp1Low + 0] {smash sender field},                  c1;
            MDR ← nilPointer,                                                      c2;
            otLow ← MD {get the next sender field},                                c3;

            temp1Low ← temp1Low - senderFieldOffset,                               c1;
            temp1Low ← temp1Low + deltaWordOffset,                                 c2;
            Noop,                                                                  c3;

            MAR ← [temp1High, temp1Low + 0],                                       c1;
            Noop,                                                                  c2;
            temp3Low ← MD {delta word of current context}, XDisp,                  c3;

            BRANCH[refdNextContext, dontRefdNextContext, 0b],                      c1;

    refdNextContext:
            L1 ← nextContext,                                                      c2;
            [] ← otLow LRot0, XDisp, CALL[refd],                                   c3;

            Noop,                                                                  c1, at[nextContext, 10, refdReturn];

    dontRefdNextContext:
            [] ← otLow xor homeLow, ZeroBr, {are we done yet}                      c2;
            BRANCH[$, wrapupReturn],                                               c3;

            GOTO[invalidateContexts],                                              c1;



    notOnChain:
            CANCELBR[ $,1],                                                        c3;

            Noop,                                                                  c1;
```

```
smashTwo:
        temp2Low ← temp2Low + senderFieldOffset,                                    c2;
        Noop,                                                                       c3;

        MAR ← [temp2High, temp2Low + 0],                                            c1;
        MDR ← nilPointer,                                                           c2;
        temp2Low ← temp2Low + differenceBetweenSenderAndInstructionPointerFields, c3;

        MAR ← [temp2High, temp2Low + 0],                                            c1;
        MDR ← nilPointer,                                                           c2;
        Noop,                                                                       c3;

wrapupReturn:
        Noop,                                                                       c1;

returnWrapup:

        L1 ← downOldContextOnReturn,                                                c2;
        otLow ← uActiveContextOop, XDisp, CALL[refd],                               c3;

        uNewContextOop ← 0 or homeLow,CALL[fetchContextRegistersAndMakeContextVolatile], L0 ← returningToAContext, c1,
        at[downOldContextOnReturn, 10, refdReturn];

        stackLow ← stackLow + 1,                                                    c3, at[returningToAContext, 10,
        fetchContextRegisters-return];

        MAR ← [stackHigh, stackLow + 0],                                            c1;
        MDR ← uReturnValue,                                                         c2;
        GOTO[fixupInstructionPointer],                                              c3;



cannotReturn:
        GOTO[zot], c3;

cannotReturnB:
        GOTO[zot], c3;


zot:
        GOTO[bytecodeFailed], c1;
```

```
{
        1-Aug-84 18:34:44
}


    {Send Literal Selector}

        sendLiteralSelectorWith3BitsOfArguments:
                temp2Low ← ib {get extension byte},                                  c1, bytecode[83];
                ipLow ← ipLow + PC16, backupIs1Byte, {account for extension byte}    c2;
                temp3Low ← temp2Low and 1f {get literal index},                      c3;

                temp3Low ← temp3Low + objectHeaderSize + literalStart,               c1;
                temp1Low ← temp2Low LRot12 {start getting argument count},           c2;
                temp1Low ← RShift1 (temp1Low and 0e), SE ← 0,                        c3;

                Noop,                                                                c1;
                GOTO[getSelector],                                                   c2;



        sendLiteralSelectorWith8BitsOfArguments:
                temp1Low ← ib {get argument count},                                  c1, bytecode[84];
                temp3Low ← ib {get literal index},                                   c2;
                ipLow ← ipLow + 1, backupIs2Bytes, {account for extension bytes},    c3;

                temp3Low ← temp3Low + objectHeaderSize + literalStart,               c1;
                GOTO[getSelector],                                                   c2;



    {Send Literal Selector to Superclass}

        sendLiteralSelectorToSuperclassWith3BitsOfArguments:

                temp2Low ← ib {get extension byte},                                  c1, bytecode[85];
                ipLow ← ipLow + PC16, backupIs1Byte, {account for extension byte}    c2;
                temp3Low ← temp2Low and 1f {get literal index},                      c3;

                temp3Low ← temp3Low + objectHeaderSize + literalStart,               c1;
                temp1Low ← temp2Low LRot12 {start getting argument count},           c2;
                temp1Low ← RShift1 (temp1Low and 0e), SE ← 0, GOTO[getSuperSelector], c3;



        sendLiteralSelectorToSuperclassWith8BitsOfArguments:

                temp1Low ← ib {get argument count},                                  c1, bytecode[86];
                temp3Low ← ib {get literal index},                                   c2;
                ipLow ← ipLow + 1, backupIs2Bytes, {account for extension bytes},    c3;

                temp3Low ← temp3Low + objectHeaderSize + literalStart,               c1;
                Noop,                                                                c2;
                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}, c2;
                stackLow ← stackLow - 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;
                Q ← MD {the selector},                                               c3;

                MAR ← [stackHigh, stackLow + 0],                                     c1;
                stackLow ← stackLow + temp1Low {point at tos again}, L2 ← superclassReceiver, c2;
                otLow ← MD {the new receiver}, XDisp, CALL[getClass],                c3;

                uNewReceiverLow ← temp1Low,                                          c1, at[superclassReceiver, 10, getClass-return];
                temp1Low ← temp1High,                                                c2;
                uNewReceiverHigh ← temp1Low,                                         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 cheating 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,                                                 c1;
        temp1Low ← uCurrentMethodLow,                                                   c2;
        uSelector ← 0, L2 ← gettingSuperclass,                                          c3;

        CALL[getNewMethodHeader] {thus getting the current method's header},            c1;
        temp2Low ← (RShift1 temp1Low and 7f), SE ← 0 {literal count of current method},c2, at[gettingSuperclass, 10,
        getNewMethodHeader-return];
        temp1Low ← uCurrentMethodLow,                                                   c3;

        temp1Low ← temp1Low + temp2Low,                                                 c1;
        temp1Low ← temp1Low + objectHeaderSize,                                         c2;
        uNewReceiver ← otLow {save newReceiver -- next read smashes it},                c3;

        MAR ← [temp1High, temp1Low + 0] {read last literal in current method}, L1 ← getMethodClass, c1;
        uNewReceiversClass ← temp3Low,                                                  c2;
        otLow ← MD, {the association oop}, CALL[otMap],                                 c3;

        temp1Low ← temp1Low + associationValueIndex,                                    c1, at[getMethodClass, 10, otMap-return];
        Noop,                                                                           c2;
        Noop,                                                                           c3;

        MAR ← [temp1High, temp1Low + 0] {read value field of association},              c1;
        Noop,                                                                           c2;
        otLow ← MD {the oop of the class of the current method},                        c3;

        [] ← otLow, ZeroBr, L1 ← getSuperclass,                                         c1;
        BRANCH[classIsNotLambda, classIsLambda],                                        c2;

classIsNotLambda:
        CALL[otMap] {to map the class of the current method},                           c3;

        temp1Low ← temp1Low + superclassOffset,                                         c1, at[getSuperclass, 10, otMap-return];
        Noop,                                                                           c2;
        Noop,                                                                           c3;

        MAR ← [temp1High, temp1Low + 0] {get superclass oop},                           c1;
        otLow ← uNewReceiver {restore this for startMethodLookup},                      c2;
        temp3Low ← MD,                                                                  c3;

        [] ← temp3Low, ZeroBr {make sure superclass is non-lambda}, L3 ← 0 {this is a lookup for execution}, c1;
        uStartLookup ← temp3Low, BRANCH[$, superclassIsLambda],                         c2;
        GOTO[startMethodLookupViaSuperSend],                                            c3;




classIsLambda: {Loom:  need to call Loom here}
        GOTO[bailout1],                                                                 c3;


superclassIsLambda: {Loom:  need to call Loom here}
        GOTO[bailout1],                                                                 c3;


{Send Arithmetic Message}


sendArithmeticMessage0: {SmallInteger +}
        MAR ← [stackHigh, stackLow + 0] {start read of argument}, L1 ← smallAdd, c1, bytecode[0b0];
        stackLow ← stackLow - 1 {point at receiver}, CALL[primitiveNeeds2SmallIntegers], backupIs0Bytes, 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], backupIs0Bytes,  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], backupIs0Bytes,  c2;

        [] ← temp2Low - temp3Low, NegBr,                                                c2, at[smallLess, 10, arithmeticPrimitives];
        BRANCH[pushFalseInPrimitiveRelational, pushTrueInPrimitiveRelational],  c3;

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], backupIs0Bytes,  c2;

        [] ← temp3Low - temp2Low, NegBr,                                                c2, at[smallGreater, 10, arithmeticPrimitives];
        BRANCH[pushFalseInPrimitiveRelational, pushTrueInPrimitiveRelational],  c3;

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], backupIs0Bytes,  c2;

        Noop,                                                                           c2, at[smallLessOrEqual, 10,
```

```
            arithmeticPrimitives];
            Noop,                                                                   c3;

            [] ← temp2Low - temp3Low, NegBr,                                        c1;
            [] ← temp2Low - temp3Low, ZeroBr, BRANCH[$, isNeg],                     c2;
            BRANCH[pushFalseInPrimitiveRelational, pushTrueInPrimitiveRelational],  c3;

        isNeg:
            CANCELBR[pushTrueInPrimitiveRelational, 1],                             c3;

    sendArithmeticMessage5: {SmallInteger >=}
            MAR ← [stackHigh, stackLow + 0] {start read of top of stack}, L1 ← smallGreaterEqual, c1, bytecode[0b5];
            stackLow ← stackLow - 1 {point at receiver}, CALL[primitiveNeeds2SmallIntegers], backupIs0Bytes,  c2;

            [] ← temp2Low - temp3Low, NegBr,                                        c2, at[smallGreaterEqual, 10,
            arithmeticPrimitives];
            BRANCH[pushTrueInPrimitiveRelationala, pushFalseInPrimitiveRelationala],c3;

    sendArithmeticMessage6: {SmallInteger =}
            MAR ← [stackHigh, stackLow + 0] {start read of top of stack}, L1 ← smallEqual, c1, bytecode[0b6];
            stackLow ← stackLow - 1 {point at receiver}, CALL[primitiveNeeds2SmallIntegers], backupIs0Bytes,  c2;

            [] ← temp2Low - temp3Low, ZeroBr,                                       c2, at[smallEqual, 10, arithmeticPrimitives];
            BRANCH[pushFalseInPrimitiveRelational, pushTrueInPrimitiveRelational],  c3;

    sendArithmeticMessage7: {SmallInteger ~=}
            MAR ← [stackHigh, stackLow + 0] {start read of top of stack}, L1 ← smallNotEqual, c1, bytecode[0b7];
            stackLow ← stackLow - 1 {point at reeiver}, CALL[primitiveNeeds2SmallIntegers], backupIs0Bytes,  c2;

            [] ← temp2Low - temp3Low, NZeroBr,                                      c2, at[smallNotEqual, 10, arithmeticPrimitives];
            BRANCH[pushFalseInPrimitiveRelational, pushTrueInPrimitiveRelational],  c3;

    sendArithmeticMessage8: {SmallInteger *}
            backupIs0Bytes, Q ← 0b8,        GOTO[notYetInvented],                                   c1, bytecode[0b8];

    sendArithmeticMessage9: {SmallInteger divide}
            backupIs0Bytes, Q ← 0b9,        GOTO[notYetInvented],                                   c1, bytecode[0b9];

    sendArithmeticMessage10: {SmallInteger mod}
            backupIs0Bytes, Q ← 0ba,        GOTO[notYetInvented],                                   c1, bytecode[0ba];

    sendArithmeticMessage11: {SmallInteger makePoint}
            otLow ← classPointPointer,                                              c1, bytecode[0bb];
            uClassToInstantiate ← otLow,                                            c2;
            temp3High ← viaPrimitiveMakePoint,                                      c3;

            temp3Low ← 2 {number of fields in a Point}, backupIs0Bytes,             c1;
            Noop,                                                                   c2;
            CALL[createInstanceWithPointers],                                       c3;


            temp1High ← uNewObjectHigh,                                             c1, at[viaPrimitiveMakePoint, 10,
            createInstance-return];
            temp1Low ← uNewObjectLow,       L1 ← makePoint,                         c2;
            temp1Low ← temp1Low + yFieldOffsetInPoint, CALL[getSmashTos],           c3;


            MAR ← [temp1High, temp1Low + 0],                                        c1, at[makePoint, 10, getSmashTos-return];
            MDR ← temp3Low, otLow ← temp3Low {save y field for refi},               c2;
            temp1Low ← temp1Low - offsetFromXFieldToYField, CALL[getTos],           c3;

            MAR ← [temp1High, temp1Low + 0],                                        c1, at[makePoint, 10, getTos-return];
            MDR ← temp3Low, L2 ← upY,                                               c2;
            CALL[primRefi], {up y field}                                            c3;

            otLow ← temp3Low, L2 ← upX, CALL[primRefi] {up x field},                c3, at[upY, 10, primRefi-return];

            temp3Low ← uNewObject, GOTO[pushTemp3LowAndDispatch],                   c3, at[upX, 10, primRefi-return];




    primRefi:
            Noop,                                                                   c1;
            L1 ← primitiveRefi,                                                     c2;
            [] ← otLow LRot0, XDisp, CALL[refi],                                    c3;

            L2Disp,                                                                 c1, at[primitiveRefi, 10, refiReturn];
            RET[primRefi-return],                                                   c2;




    getTos:
            {return the tos value in temp3Low without changing stackPointer or nilling tos}
            MAR ← [stackHigh, stackLow + 0],                                        c1;
            L1Disp,                                                                 c2;
            temp3Low ← MD, 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;
```

```
                temp3Low ← MD,                                                  c3;

                Noop,                                                           c1;
                L1Disp,                                                         c2;
                stackLow ← stackLow - 1, RET[getSmashTos-return],               c3;


        sendArithmeticMessage12: {SmallInteger bitShift}
                MAR ← [stackHigh, stackLow + 0] {start read of argument},
                        L1 ← smallBitShift,                                     c1, bytecode[0bc];
                stackLow ← stackLow - 1 {point at receiver},
                        CALL[primitiveNeeds2SmallIntegers], backupIs0Bytes,     c2;


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


left:
        Q ← 0D - temp3Low, CarryBr, BRANCH [$, returnZeroLeft],                 c1;
        BRANCH [failLeftShift, $],                                              c2;
        [] ← temp3Low, ZeroBr,                                                  c3;

leftLoop:
        BRANCH [$, leftShiftDone],                                              c1;
        temp2Low ← temp2Low + temp2Low, PgCrOvDisp,                             c2;
        temp3Low ← temp3Low - 1, ZeroBr, BRANCH [leftLoop, leftOverflow, 2],    c3;

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

returnZeroLeft:
        temp3Low ← temp2Low, CANCELBR [pushArithmeticResult, 1],                c2;


right:
        Q ← ~0F, BRANCH [$, returnZeroRight],                                   c1;
        [] ← temp2Low, NegBr,                                                   c2;
        temp3Low ← temp3Low or Q, ZeroBr, BRANCH [posRightLoop, negRightLoop],  c3;

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

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

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

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

returnZeroRight:
        temp3Low ← temp2Low, CANCELBR [pushArithmeticResult, 1],                c2;


failLeftShift:
        Noop,                                                                   c3;

leftOverflow:
        CANCELBR [$, 1],                                                        c1;
        GOTO [arithmeticPrimitiveFailedC3],                                     c2;


        sendArithmeticMessage13: {SmallInteger div}
                backupIs0Bytes, Q ← 0bd,        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];


        sendArithmeticMessage15: {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
```

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

        temp3Low ← MD {get argument}, LOOPHOLE[mdok], XDisp {to test for SmallInteger}, c3;

        MAR  ← [stackHigh, stackLow + 0], BRANCH[arithmeticPrimitiveFailed, $, 0e], c1;

        [] ← temp3Low LRot0,  XHDisp {determine sign of argument},                    c2;
        temp2Low ← MD, BRANCH [$, argNegative, 2], XDisp {so we can test receiver for being a SmallInteger},c3;

        temp3Low ← RShift1 temp3Low, SE ← 0, {make a useful number} BRANCH[receiverNotSmall, adjustReceiver, 0e], c1;
argNegative:
        temp3Low ← RShift1 temp3Low, SE ← 1, {make a useful negative number} BRANCH[receiverNotSmall, adjustReceiver, 0e], c1;
adjustReceiver:
        [] ← temp2Low LRot0,  XHDisp {determine sign of receiver},                    c2;
        BRANCH[$, receiverNegative, 2], L1Disp,                                       c3;
        temp2Low ← RShift1 temp2Low, SE ← 0, RET[arithmeticPrimitives],               c1;

receiverNegative:
        temp2Low ← RShift1 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 + temp3Low + 1 {make result a SmallInteger},
                PgCrOvDisp,                                                           c3;

        MAR ← [stackHigh, stackLow + 0] {start writing resultant SmallInteger},
                BRANCH[$, largeIntegerTest, 2],                                       c1;
        MDR ← temp3Low, NextBytecode,                                                 c2;
        DISPNI[bytecodes], ipLow ← ipLow + PC16,                                      c3;

largeIntegerTest:
          {if it's negative now, it's really positive}
        temp3Low ← temp3Low RShift1, SE ← 0, NegBr,                                   c2;
        temp3Low ← temp3Low LRot8, BRANCH [arithmeticPrimitiveFailedC1, $],           c3;

        uLargeIntegerValueLow ← temp3Low,                                             c1;
        temp3Low ← 2, {2 byte large integer}                                          c2;
        temp3High ← twoByteLargeInteger, CALL [createLargePositiveInteger],           c3;

        temp3Low ← uLargeIntegerValueLow,                                             c1, at [twoByteLargeInteger, 10,
        createInstance-return];
        L1 ← largeIntegerResult,                                                      c2;
        CALL [otMap2],                                                                c3;

        temp1Low ← temp1Low + objectHeaderSize,                                       c1, at [largeIntegerResult, 10, otMap2-return];
        Noop,                                                                         c2;
        Noop,                                                                         c3;

        MAR ← [temp1High, temp1Low + 0],                                              c1;
        MDR ← temp3Low,                                                               c2;
        Noop,                                                                         c3;

        MAR ← [stackHigh, stackLow + 0],                                              c1;
        MDR ← otLow, NextBytecode,                                                    c2;
        DISPNI[bytecodes], ipLow ← ipLow + PC16,                                      c3;

arithmeticPrimitiveFailedC1:
        Noop,                                                                         c1;
        GOTO[arithmeticPrimitiveFailedC3],                                            c2;     {can't create a LargeNegativeInteger}

arithmeticPrimitiveFailed:
        CANCELBR[$,0f],                                                               c2;

arithmeticPrimitiveFailedC3:
        stackLow ← stackLow + 1 {adjust to point at argument},
                L3Disp {are we here because a directly dispatched primitive
                failed or because a looked up primitive failed?},                     c3;

        DISP2[whichWayShouldIGo],                                                     c1;

        Noop, {primitive found thru lookup failed, run method},                       c2, at[1, 4, whichWayShouldIGo];
        GOTO[activateNewMethod],                                                      c3;

        {directly dispatched primitive failed, so do lookup}
        Noop,                                                                         c2, at[0, 4, whichWayShouldIGo];
        L1Disp,                                                                       c3;

        DISP4[arithPrimitive],                                                        c1;

        temp3Low ← 0, GOTO[getSelectorAndArgs],                                       c2, at[0, 10, arithPrimitive];

        temp3Low ← 2, GOTO[getSelectorAndArgs],                                       c2, at[1, 10, arithPrimitive];

        temp3Low ← 4, GOTO[getSelectorAndArgs],                                       c2, at[2, 10, arithPrimitive];

        temp3Low ← 6, GOTO[getSelectorAndArgs],                                       c2, at[3, 10, arithPrimitive];
```

```
                temp3Low ← 8, GOTO[getSelectorAndArgs],                          c2, at[4, 10, arithPrimitive];
                temp3Low ← 0a, GOTO[getSelectorAndArgs],                         c2, at[5, 10, arithPrimitive];
                temp3Low ← 0c, GOTO[getSelectorAndArgs],                         c2, at[6, 10, arithPrimitive];
                temp3Low ← 0e, GOTO[getSelectorAndArgs],                         c2, at[7, 10, arithPrimitive];
                temp3Low ← 10, GOTO[getSelectorAndArgs],                         c2, at[8, 10, arithPrimitive];
                temp3Low ← 12, GOTO[getSelectorAndArgs],                         c2, at[9, 10, arithPrimitive];
                temp3Low ← 14, GOTO[getSelectorAndArgs],                         c2, at[0a, 10, arithPrimitive];
                temp3Low ← 16, GOTO[getSelectorAndArgs],                         c2, at[0b, 10, arithPrimitive];
                temp3Low ← 18, GOTO[getSelectorAndArgs],                         c2, at[0c, 10, arithPrimitive];
                temp3Low ← 1a, GOTO[getSelectorAndArgs],                         c2, at[0d, 10, arithPrimitive];
                temp3Low ← 1c, GOTO[getSelectorAndArgs],                         c2, at[0e, 10, arithPrimitive];
                temp3Low ← 1e, GOTO[getSelectorAndArgs],                         c2, at[0f, 10, arithPrimitive];
        specialLookup:
                Noop,                                                            c2;
        getSelectorAndArgs:
                {upon entry temp3Low must be the (index*2)+1 into the special selectors object}
                otLow ← specialSelectorsOop, CALL[otMap2], L1 ← gettingSpecialSelectors,c3;
                temp1Low ← temp1Low + temp3Low,                                  c1, at[gettingSpecialSelectors, 10,
                otMap2-return];
                temp1Low ← temp1Low + objectHeaderSize,                          c2;
                Noop,                                                            c3;
                MAR ← [temp1High, temp1Low + 0],                                 c1;
                temp1Low ← temp1Low + 1,                                         c2;
                Q ← MD {the selector},                                           c3;
                MAR ← [temp1High, temp1Low + 0],                                 c1;
                Noop,                                                            c2;
                temp1Low ← MD {the argument count, a SmallInteger},              c3;
                temp1Low ← RShift1 temp1Low, SE ← 0,                             c1;
                uArgumentCount ← temp1Low,                                       c2;
                stackLow ← stackLow - temp1Low {point at receiver}, GOTO[getReceiver], c3;
        pushFalseInPrimitiveRelational:
                temp3Low ← falsePointer, GOTO[pushRelationalResult],             c1;
        pushFalseInPrimitiveRelationala:
                temp3Low ← falsePointer, GOTO[pushRelationalResult],             c1;
        pushTrueInPrimitiveRelational:
                temp3Low ← truePointer, GOTO[pushRelationalResult],              c1;
        pushTrueInPrimitiveRelationala:
                temp3Low ← truePointer, GOTO[pushRelationalResult],              c1;
        pushRelationalResult:
                Noop,                                                            c2;
                GOTO[pushTemp3LowAndDispatch],                                   c3;
    {Send Special Message}
        sendSpecialMessage0: {at:}
                temp3Low ← 32'd, GOTO[specialLookup], backupIs0Bytes,            c1, bytecode[0c0];
        sendSpecialMessage1: {at:put:}
                temp3Low ← 34'd, GOTO[specialLookup], backupIs0Bytes,            c1, bytecode[0c1];
        sendSpecialMessage2: {size}
                temp3Low ← 36'd, GOTO[specialLookup], backupIs0Bytes,            c1, bytecode[0c2];
        sendSpecialMessage3: {next}
                temp3Low ← 38'd, GOTO[specialLookup], backupIs0Bytes,            c1, bytecode[0c3];
```

```
sendSpecialMessage4: {nextPut:}
        temp3Low ← 40'd, GOTO[specialLookup], backupIsOBytes,                    c1, bytecode[0c4];

sendSpecialMessage5: {atEnd}
        temp3Low ← 42'd, GOTO[specialLookup], backupIsOBytes,                    c1, bytecode[0c5];

sendSpecialMessage6: { == }
        MAR ← [stackHigh, stackLow + 0] {start read of argument},                c1, bytecode[0c6];
        MDR ← nilPointer {smash argument},                                       c2;
        temp3Low ← MD {get argument},                                            c3;

        stackLow ← stackLow - 1,                                                 c1;
        Noop,                                                                    c2;
        Noop,                                                                    c3;

        MAR ← [stackHigh, stackLow + 0] {start read of receiver},                c1;
        Noop,                                                                    c2;
        temp2Low ← MD,                                                           c3;

        Noop,                                                                    c1;
        [] ← temp3Low - temp2Low, ZeroBr,                                        c2;
        BRANCH[pushFalseInPrimitiveRelational, pushTrueInPrimitiveRelational],   c3;

sendSpecialMessage7: {class}
        MAR ← [stackHigh, stackLow + 0],                                         c1, bytecode[0c7];
        L2 ← primitiveClass {the getClass call proceeds directly to pushTemp3LowAndDispatch}, c2;
        otLow ← MD, XDisp {to test for SmallInteger}, CALL[getClass],            c3;


sendSpecialMessage8: {blockCopy:}
        MAR ← [stackHigh, stackLow + 0] {read arg count}, backupIsOBytes,        c1, bytecode[0c8];
        stackLow ← stackLow - 1,                                                 c2;
        temp2Low ← MD {arg count as smallInteger},                               c3;

        MAR ← [stackHigh, stackLow + 0],                                         c1;
        Q ← methodContextClassOop, L2 ← directBlockCopy,                         c2;
        otLow ← MD {context oop}, XDisp, CALL[getClass],                         c3;

        [] ← temp3Low xor Q, ZeroBr,                                             c1, at[directBlockCopy, 10, getClass-return];
        uArgumentCount ← temp2Low, BRANCH[$, blockCopyOk],                       c2;
        Q ← blockContextClassOop,                                                c3;

        [] ← temp3Low xor Q, ZeroBr,                                             c1;
        BRANCH[$, blockCopyOkA],                                                 c2;
        temp3Low ← 48'd,                                                         c3;

        stackLow ← stackLow + 1, GOTO[specialLookup],                            c1;


blockCopyOk:
        GOTO[primitiveBlockCopyViaDirectDispatch],                               c3;

blockCopyOkA:
        GOTO[primitiveBlockCopyViaDirectDispatch],                               c3;


sendSpecialMessage9: {value}
        MAR ← [stackHigh, stackLow + 0], backupIsOBytes,                         c1, bytecode[0c9];
        uArgumentCount ← 0,                                                      c2;

valueGettingContext:
        otLow ← MD, XDisp, L2 ← directValue, CALL[getClass],                     c3;

        temp2Low ← blockContextClassOop,                                         c1, at[directValue, 10, getClass-return];
        [] ← temp3Low xor temp2Low, ZeroBr,                                      c2;
        BRANCH[$, primitiveValueViaDirectDispatch], c3;

        {not a blockContext, do special lookup}
        [] ← uArgumentCount, ZeroBr,                                             c1;
        BRANCH[wasValueColon, wasValue],                                         c2;

wasValue:
        temp3Low ← 50'd, GOTO[valueLookup],                                      c3;

wasValueColon:
        temp3Low ← 52'd, GOTO[valueLookup],                                      c3;

valueLookup:
        GOTO[specialLookup],                                                     c1;


sendSpecialMessage10: {value:}
        stackLow ← stackLow - 1, backupIsOBytes,                                 c1, bytecode[0ca];
        temp3Low ← 1,                                                            c2;
        uArgumentCount ← temp3Low,                                               c3;

        MAR ← [stackHigh, stackLow + 0],                                         c1;
        stackLow ← stackLow + 1 {in case of failure}, GOTO[valueGettingContext], c2;
```

```
    sendSpecialMessage11: {do:}
            temp3Low ← 64'd, GOTO[specialLookup], backupIs0Bytes,                    c1, bytecode[0cb];

    sendSpecialMessage12: {new}
            temp3Low ← 66'd, GOTO[specialLookup], backupIs0Bytes,                    c1, bytecode[0cc];

    sendSpecialMessage13: {new:}
            temp3Low ← 68'd, GOTO[specialLookup], backupIs0Bytes,                    c1, bytecode[0cd];

    sendSpecialMessage14: {x}
            temp3Low ← 60'd, GOTO[specialLookup], backupIs0Bytes,                    c1, bytecode[0ce];

    sendSpecialMessage15: {y}
            temp3Low ← 62'd, GOTO[specialLookup], backupIs0Bytes,                    c1, bytecode[0cf];


{Send Literal Selector With No Arguments}

    sendLiteralSelector0WithNoArguments:
            temp3Low ← literalField0, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0d0];

    sendLiteralSelector1WithNoArguments:
            temp3Low ← literalField1, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0d1];

    sendLiteralSelector2WithNoArguments:
            temp3Low ← literalField2, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0d2];

    sendLiteralSelector3WithNoArguments:
            temp3Low ← literalField3, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0d3];

    sendLiteralSelector4WithNoArguments:
            temp3Low ← literalField4, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0d4];

    sendLiteralSelector5WithNoArguments:
            temp3Low ← literalField5, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0d5];

    sendLiteralSelector6WithNoArguments:
            temp3Low ← literalField6, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0d6];

    sendLiteralSelector7WithNoArguments:
            temp3Low ← literalField7, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0d7];

    sendLiteralSelector8WithNoArguments:
            temp3Low ← literalField8, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0d8];

    sendLiteralSelector9WithNoArguments:
            temp3Low ← literalField9, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0d9];

    sendLiteralSelector10WithNoArguments:
            temp3Low ← literalField10, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0da];

    sendLiteralSelector11WithNoArguments:
            temp3Low ← literalField11, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0db];

    sendLiteralSelector12WithNoArguments:
            temp3Low ← literalField12, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0dc];

    sendLiteralSelector13WithNoArguments:
            temp3Low ← literalField13, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0dd];

    sendLiteralSelector14WithNoArguments:
            temp3Low ← literalField14, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0de];

    sendLiteralSelector15WithNoArguments:
            temp3Low ← literalField15, backupIs0Bytes, GOTO[getSelectorZeroArguments],c1, bytecode[0df];


    getSelectorZeroArguments:
            temp1Low ← 0, GOTO[getSelector],                                         c2;


{Send Literal Selector With 1 Argument}

    sendLiteralSelector0With1Argument:
            temp3Low ← literalField0, backupIs0Bytes, GOTO[getSelectorOneArgument], c1, bytecode[0e0];

    sendLiteralSelector1With1Argument:
            temp3Low ← literalField1, backupIs0Bytes, GOTO[getSelectorOneArgument], c1, bytecode[0e1];

    sendLiteralSelector2With1Argument:
            temp3Low ← literalField2, backupIs0Bytes, GOTO[getSelectorOneArgument], c1, bytecode[0e2];
```

```
sendLiteralSelector3With1Argument:
        temp3Low ← literalField3, backupIsOBytes, GOTO[getSelectorOneArgument], c1, bytecode[0e3];

sendLiteralSelector4With1Argument:
        temp3Low ← literalField4, backupIsOBytes, GOTO[getSelectorOneArgument], c1, bytecode[0e4];

sendLiteralSelector5With1Argument:
        temp3Low ← literalField5, backupIsOBytes, GOTO[getSelectorOneArgument], c1, bytecode[0e5];

sendLiteralSelector6With1Argument:
        temp3Low ← literalField6, backupIsOBytes, GOTO[getSelectorOneArgument], c1, bytecode[0e6];

sendLiteralSelector7With1Argument:
        temp3Low ← literalField7, backupIsOBytes, GOTO[getSelectorOneArgument], c1, bytecode[0e7];

sendLiteralSelector8With1Argument:
        temp3Low ← literalField8, backupIsOBytes, GOTO[getSelectorOneArgument], c1, bytecode[0e8];

sendLiteralSelector9With1Argument:
        temp3Low ← literalField9, backupIsOBytes, GOTO[getSelectorOneArgument], c1, bytecode[0e9];

sendLiteralSelector10With1Argument:
        temp3Low ← literalField10, backupIsOBytes, GOTO[getSelectorOneArgument],c1, bytecode[0ea];

sendLiteralSelector11With1Argument:
        temp3Low ← literalField11, backupIsOBytes, GOTO[getSelectorOneArgument],c1, bytecode[0eb];

sendLiteralSelector12With1Argument:
        temp3Low ← literalField12, backupIsOBytes, GOTO[getSelectorOneArgument],c1, bytecode[0ec];

sendLiteralSelector13With1Argument:
        temp3Low ← literalField13, backupIsOBytes, GOTO[getSelectorOneArgument],c1, bytecode[0ed];

sendLiteralSelector14With1Argument:
        temp3Low ← literalField14, backupIsOBytes, GOTO[getSelectorOneArgument],c1, bytecode[0ee];

sendLiteralSelector15With1Argument:
        temp3Low ← literalField15, backupIsOBytes, GOTO[getSelectorOneArgument],c1, bytecode[0ef];


getSelectorOneArgument:
        temp1Low ← 1, GOTO[getSelector],                                    c2;
```

{Send Literal Selector With 2 Arguments}

```
sendLiteralSelector0With2Arguments:
        temp3Low ← literalField0, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0f0];

sendLiteralSelector1With2Arguments:
        temp3Low ← literalField1, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0f1];

sendLiteralSelector2With2Arguments:
        temp3Low ← literalField2, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0f2];

sendLiteralSelector3With2Arguments:
        temp3Low ← literalField3, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0f3];

sendLiteralSelector4With2Arguments:
        temp3Low ← literalField4, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0f4];

sendLiteralSelector5With2Arguments:
        temp3Low ← literalField5, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0f5];

sendLiteralSelector6With2Arguments:
        temp3Low ← literalField6, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0f6];

sendLiteralSelector7With2Arguments:
        temp3Low ← literalField7, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0f7];

sendLiteralSelector8With2Arguments:
        temp3Low ← literalField8, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0f8];

sendLiteralSelector9With2Arguments:
        temp3Low ← literalField9, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0f9];

sendLiteralSelector10With2Arguments:
        temp3Low ← literalField10, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0fa];

sendLiteralSelector11With2Arguments:
        temp3Low ← literalField11, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0fb];

sendLiteralSelector12With2Arguments:
        temp3Low ← literalField12, backupIsOBytes, GOTO[getSelectorTwoArguments],c1, bytecode[0fc];
```

```
sendLiteralSelector13With2Arguments:
        temp3Low ← literalField13, backupIs0Bytes, GOTO[getSelectorTwoArguments],c1, bytecode[0fd];

sendLiteralSelector14With2Arguments:
        temp3Low ← literalField14, backupIs0Bytes, GOTO[getSelectorTwoArguments],c1, bytecode[0fe];

sendLiteralSelector15With2Arguments:
        temp3Low ← literalField15, 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 + temp3Low {add in offset to literal selector}, c1;
        Noop,                                                                   c2;
        stackLow ← stackLow - 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;
        Q ← MD {the selector},                                                  c3;

getReceiver:
        MAR ← [stackHigh, stackLow + 0], L3 ← 0 {this is a lookup for execution}, c1;
        stackLow ← stackLow + temp1Low {point at tos again}, L2 ← gettingNewReceiversClass, c2;
        otLow ← 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
                        otLow is still the new receiver
        }


startMethodLookup:

        {upon entry, otLow must be oop of new receiver, temp1High/Low is the base of new receiver, temp3Low must oop of the class
        in which to start the search, Q must be selector}

        uNewReceiver ← otLow,                                                   c1, at[gettingNewReceiversClass, 10,
        getClass-return];
        uNewReceiversClass ← temp3Low,                                          c2;
        uSelector ← Q,                                                          c3;

        uNewReceiverLow ← temp1Low,                                             c1;
        temp1Low ← temp1High,                                                   c2;
        uNewReceiverHigh ← temp1Low,                                            c3;

startMethodLookupViaSuperSend:

        {first, we look in the method cache}

        temp2Low ← Q and temp3Low {mash selector and class together},           c1;
        temp2Low ← temp2Low and 0ff {256 entries in message cache},             c2;
        temp2Low ← LShift1 temp2Low, SE ← 0 {cache entries are 4 words },       c3;

        temp2Low ← LShift1 temp2Low, SE ← 0,                                    c1;
        uHash ← temp2Low {and save in case cache update is needed},             c2;
        temp1Low ← uMethodCacheLow {get the method cache address},              c3;

        temp1High ← uMethodCacheHigh, temp1Low ← temp2Low{hash index} + temp1Low, c1;
        uStartLookup ← temp3Low,                                                c2;
        Noop,                                                                   c3;

        {at this point, temp2Low is the hash index, temp1Low is abs address into cache}

        MAR ← [temp1High, temp1Low + 0] {read word of cache entry},             c1;
        temp1Low ← temp1Low + 1,                                                c2;
        temp2Low ← MD {the selector field from cache entry},                    c3;

        [] ← temp2Low - Q, ZeroBr {test for selector match}, L1 ← methodSearch, c1;
        BRANCH[notInCache, $],                                                  c2;
        Noop,                                                                   c3;
```

```
        MAR ← [temp1High, temp1Low + 0] {next word from cache entry},          c1;
        temp1Low ← temp1Low + 1,                                               c2;
        temp2Low ← MD {the class field of cache entry},                        c3;

        [] ← temp3Low - temp2Low, ZeroBr {test for class match},               c1;
        BRANCH[notInCachex, $],                                                c2;
        Noop,                                                                  c3;

        {we have a cache hit!}
        MAR ← [temp1High, temp1Low + 0] {cache hit, get method oop}            c1;
        temp1Low ← temp1Low + 1,                                               c2;
        otLow ← MD, {and save it for otMapping}                                c3;

        MAR ← [temp1High, temp1Low + 0] {finally, get the primitive flag}      c1;
        uNewMethodOop ← otLow, L1 ← gettingMethodBase,                         c2;
        temp2Low ← MD, CALL[otMap] {to get the CompiledMethods address},       c3;


        {at the otMap call, temp2Low is the primitive flag
                temp2Low is the new receiver's class
                temp3Low is also the new receiver's class
                otLow is the oop of the method

                after the otMap call, temp1High/Low are the compiledMethod's start address
        }


        Q ← temp1High {start saving CompiledMethod's address}, L2 ← foundViaCache {the getNewMethodHeader call will return
        directly to executeNewMethod}, CALL[getNewMethodHeader], c1, at[gettingMethodBase, 10, otMap-return];



        {if we get to either notInCachex or notInCache, temp3Low must be the oop of the new receiver's class and Q must be the
        selector we're looking for}

notInCachex:
        otLow ← temp3Low {get oop of class}, CALL[otMap] {to get address of class}, c3;


notInCache:
        otLow ← temp3Low {get oop of class}, CALL[otMap] {to get address of class}, c3;

        temp3Low ← RShift1 Q {apply simple hash function to selector}, SE←0,   c1, at[methodSearch, 10, otMap-return];
        uHashedSelector ← temp3Low,                                            c2;
        Noop,                                                                  c3;

        uNewClassLow ← temp1Low,                                               c1, at[saveSuperclass, 10, otMap-return];
        temp1Low ← temp1High,                                                  c2;
        uNewClassHigh ← temp1Low {save class address in case we need to look in superclass}, c3;

        temp1Low ← uNewClassLow {not enuf registers ...},                      c1;
        temp1Low ← temp1Low + messageDictionaryOffset,                         c2;
        Noop,                                                                  c3;

        MAR ← [temp1High, temp1Low + 0]{read the messageDictionary oop},       c1;
        L1 ← messageDictionary,                                                c2;
        otLow ← MD, CALL[otMap] {and otMap it},                                c3;

tryThisDictionary:
        temp2Low ← temp1Low + SelectorStartPlusObjectHeaderSize,               c1, at[messageDictionary, 10, otMap-return];
        uSelectorsStartInDictionary ← temp2Low {remember address of first oop in dictionary}, c2;
        temp1Low ← temp1Low + sizeFieldOffset,                                 c3;

        MAR ← [temp1High, temp1Low + 0] {get the size field},                  c1;
        temp1Low ← temp1Low - sizeFieldOffset {low address of dictionary object}, c2;
        temp3Low ← MD {dictionary length},                                     c3;

        temp2Low ← temp1Low,                                                   c1;
        temp2Low ← temp2Low {dict low address} + temp3Low {dictinary length}, c2;
        temp2Low ← temp2Low - 1 {yielding address of last selector in dictionary}, c3;

        temp3Low ← temp3Low {length}- SelectorStartPlusObjectHeaderSize - 1 {yielding the "mask"}, c1;
        temp3Low ← temp3Low{mask} and uHashedSelector,                         c2;
        temp3Low ← temp3Low + SelectorStartPlusObjectHeaderSize,               c3;

                {and add the dictionary relative offset to the dictionary base}
        temp1Low ← temp1Low + temp3Low,                                        c1;
        uWrap ← 0 {probing in this dictionary has not yet wrapped},            c2;
        Noop,                                                                  c3;

probe:
        MAR ← [temp1High, temp1Low + 0], {read oop from methodDictionary}      c1;
        Noop,                                                                  c2;
        temp3Low ← MD,                                                         c3;

        [] ← temp3Low xor nilPointer, ZeroBr,                                  c1;
        [] ← temp3Low xor Q {selector}, ZeroBr, BRANCH[$, nilFound],           c2;
        BRANCH[checkForEndOfDictionary {might still be in this dictionary if we wrap}, foundIt], c3;

checkForEndOfDictionary:
        Noop,                                                                  c1;
        [] ← temp1Low {where we are} - temp2Low {end of dictionary}, ZeroBr,   c2;
        temp1Low ← temp1Low + 1, BRANCH[probe, $],                             c3;

        {need to check for wrapping}
        [] ← uWrap, ZeroBr,                                                    c1;
```

```
        uWrap ← ~temp1Low xor temp1Low {mark that we've wrapped}, BRANCH[trySuperclass, $], c2;
        temp1Low ← uSelectorsStartInDictionary, GOTO[probe],                    c3;


nilFound:
        {not in this dictionary, try superclass}
        temp1High ← uNewClassHigh, CANCELBR[getAndCheckSuperclass, 1],          c3;

trySuperclass:
        temp1High ← uNewClassHigh,                                              c3;

getAndCheckSuperclass:
        temp1Low ← uNewClassLow,                                                c1;
        temp1Low ← temp1Low + superclassOffset,                                 c2;
        Noop,                                                                   c3;

        MAR ← [temp1High, temp1Low + 0] {get superclass oop},                   c1;
        Noop,                                                                   c2;
        otLow ← MD,                                                             c3;

        Noop,                                                                   c1;
        [] ← otLow xor nilPointer, ZeroBr, L1 ← saveSuperclass {if we otMap, we return to just before tryThisDictionary}, c2;
        BRANCH[{CALL} otMap {not nil}, doesNotUnderstand],                              c3;


doesNotUnderstand:
        GOTO[bytecodeFailed],                                                   c1;




foundIt:
        {we have found the selector in the methodDictionary.  we need to get the method oop from the parallel array, and then
        update the cache}
        temp2Low ← uSelectorsStartInDictionary,                                 c1;
        Q ← temp1Low {location of hit in methodDictionary} - temp2Low, {yielding relative location of selector} c2;
        temp1Low ← temp2Low - 1 {point at methodArray field},                   c3;

        MAR ← [temp1High, temp1Low + 0], {get the oop of the methodArray}       c1;
        Q ← Q + objectHeaderSize, L1 ← methodArray,                             c2;
        otLow ← MD, CALL[otMap] {get address of method array},                  c3;

        temp1Low ← temp1Low + Q {point at appropriate entry of methodArray},    c1, at[methodArray, 10, otMap-return];
        temp3High ← uMethodCacheHigh {temp3High/Low used later to update cache},c2;
        temp3Low ← uMethodCacheLow,                                             c3;

        MAR ← [temp1High, temp1Low + 0] {read oop from methodArray}, L2 ← foundViaLookup, c1;
        L1 ← methodBaseAfterLookup,                                             c2;
        otLow ← MD {oop of method!}, CALL[otMap] {to get address of new compiledMethod}, c3;

        Q ← temp1High, CALL[getNewMethodHeader],                                c1, at[methodBaseAfterLookup, 10, otMap-return];

        temp2Low ← RRot1 temp1Low {start extracting flag bits from method header}, c2, at[foundViaLookup, 10,
        getNewMethodHeader-return];
        temp2Low ← temp2Low LRot4,                                              c3;

        temp2Low ← temp2Low and 7,                                              c1;
        [] ← temp2Low xor 7, ZeroBr, {could a primitive be specified?}          c2;
        temp2Low ← RShift1 (temp1Low and 7f) {extract the literal bits from method header}, SE ← 0, BRANCH[noPrimitive,
        primitiveSpecified],    c3;



noPrimitive:
        temp2Low ← 0, {0 = no primitive}                                        c1;
        Q ← uHash {more getting ready to update cache},                         c2;
        temp3Low ← temp3Low + Q, GOTO[updateCache],                             c3;


primitiveSpecified:
        {read the method header extenstion -- next to last literal}
        temp1Low ← uNewMethodLow,                                               c1;
        temp1Low ← temp1Low + temp2Low {literal count},                         c2;
        temp1Low ← temp1Low + objectHeaderSize,                                 c3;

        temp1Low ← temp1Low - 1,                                                c1;
        Q ← uHash {more getting ready to update method cache}                   c2;
        temp3Low ← temp3Low + Q, {and more getting ready to update method cache}c3;

        MAR ← [temp1High, temp1Low + 0] {get method header extension},          c1;
        temp1Low ← uNewMethodHeader,                                            c2;
        temp2Low ← MD,                                                          c3;

        temp2Low ← RShift1 temp2Low,                                            c1;
        temp2Low ← temp2Low and 0ff {the primitive flag},                       c2;
```

```
        Noop,                                                                   c3;

    updateCache:
        {upon entry, temp3High/Low must be address of appropriate cache entry, uSelector, uNewReceiversClass 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 ← temp2Low,                                                         c2;
        Noop,                                                                   c3;

        uNewMethodOop ← 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}
        Noop,                                                                   c1, at[0, 10, performOrExecute-return];
        [] ← temp2Low, ZeroBr {any primitive specified?}, L3 ← 1,               c2;
        Ybus ← temp1Low LRot4, XDisp, BRANCH[primitiveIndexNotZero {yes}, $],   c3;

        DISP4[flagTable, 1],                                                    c1;

        {flag = 0 - 4 and flag = 7}

        GOTO[doActivate],                                                       c2, at[1, 10, flagTable];
        GOTO[doActivate],                                                       c2, at[3, 10, flagTable];
        GOTO[doActivate],                                                       c2, at[5, 10, flagTable];
        GOTO[doActivate],                                                       c2, at[7, 10, flagTable];
        GOTO[doActivate],                                                       c2, at[9, 10, flagTable];
        GOTO[doActivate],                                                       c2, at[0f, 10, flagTable];


        {flag = 5, return self}
        NextBytecode {easy, result is already on stack, do nothing},           c2, at[0b, 10, flagTable];
        DISPNI[bytecodes], ipLow ← ipLow + PC16,                                c3;

        {flag = 6, return instance var}
        temp3Low ← uNewMethodHeader,                                            c2, at[0d, 10, flagTable];
        temp3Low ← temp3Low LRot8,                                              c3;

        temp3Low ← temp3Low and 1f {get offset into object},                    c1;
        temp1High ← uNewReceiverHigh,                                           c2;
        temp1Low ← uNewReceiverLow,                                             c3;

        temp1Low ← temp1Low + temp3Low {add in receiver field offset},          c1;
        temp1Low ← temp1Low + objectHeaderSize {account for object header},     c2;
        Noop,                                                                   c3;

        MAR ← [temp1High, temp1Low + 0] {read the instance variable},           c1;
        Noop,                                                                   c2;
        temp3Low ← MD, GOTO[pushTemp3LowAndDispatch],                           c3;



    doActivate:
        Noop,                                                                   c3;


    activateNewMethod:
        {ok, we need to find a context to use for this send}

        temp1High ← uRumRecordHigh {get Rum record address},                    c1;
        temp1Low ← uRumRecordLow,                                               c2;
        Xbus ← uNewMethodHeader, XLDisp,                                        c3;

        BRANCH[needSmallContext, needLargeContext, 1],                          c1;

    needLargeContext:
        {In our scheme, 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,                                          c2;
            temp3Low ← largeContextSizeLessObjectHeader,                         c3;

            CALL[methodContextPlease],                                           c1;

            temp2High ← uRumRecordHigh,                                          c1, at[makeBigContext, 10,
            createInstance-return];
            temp2Low ← uRumRecordLow,                                            c2;
            Noop,                                                                c3;

            MAR ← [temp2High, temp2Low + leafContextOopOffset],                  c1;
            CANCELBR[$, 2],                                                      c2;
            temp3Low ← MD,                                                       c3;

            Noop,                                                                c1;
            [] ← temp3Low xor uActiveContextOop, ZeroBr,                         c2;
            BRANCH[activeNotLeaf, activeIsLeaf],                                 c3;

      activeIsLeaf:
            MAR ← [temp2High, temp2Low + leafContextOopOffset], L1 ← newLargeContextSmashLeaf, c1;
            MDR ← nilPointer, CANCELBR[$, 2], LOOPHOLE[wok],                     c2;
            otLow ← MD, XDisp, CALL[refd],                                       c3;

            otLow ← 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 ← [temp1High, temp1Low + leafContextOopOffset],                  c1;
            temp3High ← makeSmallContext {return link if we instantiate}, CANCELBR[$, 2], c2;
            otLow ← MD {oop of leaf context},                                    c3;

            [] ← otLow - temp3Low, ZeroBr,                                       c1;
            [] ← otLow xor nilPointer, ZeroBr, BRANCH[$, needInstantiateSmallContext], c2;
            uNewContextOop ← otLow, BRANCH[haveContextWillTravel {ok, we can recycle the leaf context}, instantiateSmallContext], c3;


      needInstantiateSmallContext:
            CANCELBR[instantiateSmallContext, 1],                                c3;


      instantiateSmallContext:
            temp3Low ← smallContextSizeLessObjectHeader, CALL[methodContextPlease], c1;

            temp2High ← uRumRecordHigh,                                          c1, at[makeSmallContext, 10,
            createInstance-return];
            temp2Low ← uRumRecordLow, L1 ← upNewSmallContext,                    c2;
            [] ← otLow LRot0, XDisp, CALL[refi],                                 c3;

            MAR ← [temp2High, temp2Low + leafContextOopOffset], L1 ← downOldLeafContext, c1, at[upNewSmallContext, 10, refiReturn];
            MDR ← otLow, CANCELBR[$, 2], LOOPHOLE[wok],                          c2;
            otLow ← MD, XDisp, CALL[refd],                                       c3;

            otLow ← uNewObject, GOTO[readyToTravel],                             c1, at[downOldLeafContext, 10, refdReturn];

      activeNotLeaf:
            Noop,                                                                c1;

      readyToTravel:
            Noop,                                                                c2;
            uNewContextOop ← otLow, GOTO[haveContextWillTravel],                 c3;




      haveContextWillTravel:

            uMakeVolatileLinkage ← makeNewContextVolatile, CALL[makeVolatile {for the new context}], c1;

            {start filling in the fields of the new context}
            temp1Low ← temp1Low + senderFieldOffset,                             c1, at[makeNewContextVolatile, 10,
            makeVolatile-return];
            {and extract literal count to calculate initial pc}
            temp3Low ← uNewMethodHeader,                                         c2;
            temp3Low ← RShift1 (temp3Low and 7f), SE ← 0,                        c3;

            MAR ← [temp1High, temp1Low + 0] {write sender field},                c1;
            MDR ← uActiveContextOop,                                             c2;
            temp1Low ← uMakeVolatileLow,                                         c3;

            temp3Low ←  temp3Low + literalStart,                                 c1;
            temp3Low ← LShift1 temp3Low, SE ← 1, {LiteralCountOf[methodBase] + literalStart)*2 +1}, c2;
            temp3Low ← LShift1 temp3Low, SE ← 1, {yields smallInteger}           c3;

            temp1Low ← temp1Low + instructionPointerFieldOffset,                 c1;
            Noop,                                                                c2;
            Noop,                                                                c3;
```

```
        MAR ← [temp1High, temp1Low + 0] {write pc field},                    c1;
        MDR ← temp3Low,                                                      c2;
        temp1Low ← uMakeVolatileLow,                                         c3;

        {compute stackPointer from temporary count of method header}
        temp3Low ← uNewMethodHeader,                                         c1;
        temp3Low ← temp3Low LRot8,                                           c2;
        temp3Low ← LShift1 (temp3Low and 1f), SE ← 1, {yields smallInteger}  c3;

        temp1Low ← temp1Low + stackPointerFieldOffset,                       c1;
        Noop,                                                                c2;
        Noop,                                                                c3;

        MAR ← [temp1High, temp1Low + 0] {write stack pointer field},         c1;
        MDR ← temp3Low,                                                      c2;
        temp1Low ← uMakeVolatileLow,                                         c3;

        temp1Low ← temp1Low + methodFieldOffset,                             c1;
        {and start setting up for transferring the receiver & arguments from the sending context to the new context}
        temp3Low ← uArgumentCount,                                           c2;
        temp2Low ← stackLow, {source limit address},                         c3;

        MAR ← [temp1High, temp1Low + 0] {write oop of method we are activating},c1;
        MDR ← uNewMethodOop,                                                 c2;
        temp1Low ← uMakeVolatileLow,                                         c3;

        {finish setting up for the receiver/arguments move}
        stackLow ← stackLow - temp3Low {low 16 bits of source address},      c1;
        temp1Low ← temp1Low + receiverFieldOffset  {low 16 bits of destination address}, L1 ← activatingMove, c2;

        {now, move the receiver and any arguments from active context to the new context, nilling out the corresponding words of
        the active context}
        CALL[transferWords],                                                          c3;


        {now we need to adjust the stackPointer to reflect the absence of the transferred words}
        temp1Low ← uArgumentCount,                                           c1, at[activatingMove, 10, transferWords-return];
        stackLow ← stackLow - temp1Low - 1,                                  c2;



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}
        ipLow ← ipLow + PC16,                                                c3;

        temp3Low ← uCurrentMethodLow,                                        c1;
        temp3Low ← ipLow - temp3Low, {word relative plus headerSize}         c2;
        temp3Low ← temp3Low - objectHeaderSize {word relative},              c3;

        temp3Low ← LShift1 temp3Low, SE ← pc16{byte offset into compiledMethod},c1;
        temp3Low ← temp3Low + 1 {by the book...},                            c2;
        temp1High ← uActiveContextHigh,                                      c3;

        temp1Low ← uActiveContextLow,                                        c1;
        temp1Low ← temp1Low + instructionPointerFieldOffset,                 c2;
        temp3Low ← LShift1 temp3Low, SE ← 1, {yields smallInteger},          c3;

        MAR ← [temp1High, temp1Low + 0], {write current instruction pointer} c1;
        MDR ← temp3Low,                                                      c2;
        temp1Low ← temp1Low - instructionPointerFieldOffset {again point at base of context}, c3;

        stackLow ← stackLow - temp1Low {relative stack offset},              c1;
        stackLow ← stackLow - stackPointerAdjustmentFactor,                  c2;
        stackLow ← LShift1 stackLow, SE ← 1 {yields smallInteger},           c3;

        temp1Low ← temp1Low + stackPointerFieldOffset,                       c1;
        otLow ← uActiveContextOop,                                           c2;
        Noop,                                                                c3;

        MAR ← [temp1High, temp1Low + 0] {write current stack pointer}, L1 ← changingActiveContext, c1;
        MDR ← stackLow, L0 ← newContext,                                     c2;
        [] ← otLow LRot0, XDisp, CALL[refd] {decrease refs to activeContext},  c3;

        CALL[fetchContextRegistersOfAlreadyVolatileContext],                 c1, at[changingActiveContext, 10, refdReturn];

        GOTO[fixupInstructionPointer],                                       c3, at[newContext, 10,
        fetchContextRegisters-return];



fetchContextRegistersAndMakeContextVolatile:
        {upon entry, uNewContextOop must be the oop of the new context.    L0 is the return linkage register}

        temp2Low ← fetchingContextRegisters,                                 c2;
        otLow ← uNewContextOop,                                              c3;

        uMakeVolatileLinkage ← temp2Low, CALL[makeVolatile] {the new context}, c1;


fetchContextRegistersOfAlreadyVolatileContext:
        {upon entry, uNewContextOop must be the oop of the new context and that context must have already been made volatile,
```

```
        thus leaving uMakeVoltileHigh/Low set up.  L0 is the return linkage register}

        Noop,                                                                   c2;
        Noop,                                                                   c3;

        stackHigh ← uMakeVolatileHigh {get base of new context},                c1, at[fetchingContextRegisters, 10,
        makeVolatile-return];
        stackLow ← uMakeVolatileLow, L1 ← changingActiveContext,                c2;
        otLow ← uNewContextOop, CALL[refi],                                     c3;

        temp2High ← uRumRecordHigh,                                             c1, at[changingActiveContext, 10, refiReturn];
        temp2Low ← uRumRecordLow,                                               c2;
        temp1Low ← uMakeVolatileLow,                                            c3;

        MAR ← [temp2High, temp2Low + activeContextOopOffset] {write new active context oop}, c1;
        uActiveContextOop ← MDR ← otLow, CANCELBR[$, 2], LOOPHOLE[wok],         c2;
        Q ← temp1High ← uMakeVolatileHigh,                                      c3;

        temp1Low ← temp1Low + methodFieldOffset,                                c1;
        uActiveContextHigh ← Q,                                                 c2;
        uActiveContextLow ← stackLow,                                           c3;

        {see if this is a method or block context. odd method field implies blockcontext}
        MAR ← [temp1High, temp1Low + 0],                                        c1;
        temp1Low ← temp1Low - methodFieldOffset {again point at base of object},c2;
        ipLow ← MD, XDisp,                                                      c3;

        IBPtr ← 1 {start draining any buffered bytecodes}, BRANCH[isMethodContext, isBlockContext, 0e], c1;

isMethodContext:
        {for methodContexts, the home is the active context}
        Noop,                                                                   c2;
        GOTO[saveHomeContextStuff],                                             c3;

isBlockContext:
        {we need to know the blockContext's home. get it and make it volatile. As a side effect of volatilization, we get the
        base of the home context}
        temp1Low ← temp1Low + homeFieldOffset,                                  c2;
        Noop,                                                                   c3;

        MAR ← [temp1High, temp1Low + 0],                                        c1;
        temp1Low ← home,                                                        c2;
        otLow ← MD {the block context's home},                                  c3;

        uMakeVolatileLinkage ← temp1Low, CALL[makeVolatile],                    c1;

        temp1Low ← temp1Low + methodFieldOffset,                                c1, at[home, 10, makeVolatile-return];
        Noop,                                                                   c2;
        Noop,                                                                   c3;

        MAR ← [temp1High, temp1Low + 0] {read oop of method for homeContext}    c1;
        temp1Low ← temp1Low - methodFieldOffset {again point at base of home context}, c2;
        ipLow ← MD {oop of method},                                             c3;

saveHomeContextStuff:
        homeHigh ← uMakeVolatileHigh,                                           c1;
        homeLow ← temp1Low,                                                     c2;
        Q ← temp1Low,                                                           c3;

        uHomeLow ← Q,                                                           c1;
        temp2High ← uRumRecordHigh,                                             c2;
        temp2Low ← uRumRecordLow,                                               c3;

        MAR ← [temp2High, temp2Low + currentMethodOopOffset],                   c1;
        MDR ← ipLow {record the new context oop},CANCELBR[$, 2], LOOPHOLE[wok], c2;
        Ybus ← ib {finish draining buffered bytecodes},                         c3;

        MAR ← [temp2High, temp2Low + homeContextOopOffset],                     c1;
        MDR ← otLow {record the new home oop},CANCELBR[$, 2], LOOPHOLE[wok],    c2;
        temp1Low ← temp1Low + receiverFieldOffset,                              c3;

        MAR ← [temp1High, temp1Low + 0] {read receiver field}                   c1;
        temp1Low ← homeLow,                                                     c2;
        otLow ← MD {receiver oop},                                              c3;

        MAR ← [temp2High, temp2Low + receiverOopOffset] {write receiver field in Rum record},   c1;
        MDR ← otLow, CANCELBR [$, 2], LOOPHOLE[wok],                            c2;
        [] ← otLow LRot0, XDisp,                                                c3;

        uReceiverOop ← otLow, BRANCH[newReceiverIsOop, newReceiverIsSmall, 0e], c1;

newReceiverIsOop:
        L1 ← receiverDuringFetch,                                               c2;
        CALL[otMap2],                                                           c3;

        Q ← temp1High,                                                          c1, at[receiverDuringFetch, 10, otMap2-return];
        uReceiverHigh ← Q,                                                      c2;
        uReceiverLow ← 0 or temp1Low,                                           c3;

        Noop,                                                                   c1;

newReceiverIsSmall:
        {get relative stack pointer for this context}

        stackLow ← stackLow + stackPointerFieldOffset,                          c2;
        Noop,                                                                   c3;
```

```
        MAR ← [stackHigh, stackLow + 0],                                        c1;
        stackLow ← uActiveContextLow,                                           c2;
        temp2Low ← MD {relative stack pointer represented as smallInteger},     c3;

        temp2Low ← RShift1 temp2Low {de-smallintegerize}, SE ← 0,               c1;
        Noop,                                                                   c2;
        stackLow ← stackLow + instructionPointerFieldOffset,                    c3;

        MAR ← [stackHigh, stackLow + 0] {get instruction pointer from context}, c1;
        stackLow ← uActiveContextLow,                                           c2;
        temp3Low ← MD {relative instr ptr represented as smallInteger},         c3;

        stackLow ← stackLow + temp2Low,                                         c1;
        stackLow ← stackLow + stackPointerAdjustmentFactor, L1 ← methodDuringFetch,     c2;
        otLow ← ipLow {oop of method}, CALL[otMap2],                            c3;

        Q ← temp1High,                                                          c1, at[methodDuringFetch, 10, otMap2-return];
        uCurrentMethodHigh ← Q,                                                 c2;
        uCurrentMethodLow ← temp1Low,                                           c3;

        temp3Low ← RShift1 temp3Low, SE ← 0 {de smallIntegerize},               c1;
        temp3Low ← temp3Low - 1 {by the book...},                               c2;
        ipHigh ← Q LRot0,                                                       c3;

        ipLow ← temp1Low + objectHeaderSize, L0Disp,                            c1;
        RET[fetchContextRegisters-return],                                      c2;
```

```
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 ← methodContextClassOop,                                          c2;
        uClassToInstantiate ← otLow, CALL[createInstanceWithPointers],          c3;
```

```
{
        1-Aug-84 13:48:27
}


d1: GOTO[bailout3], at[3,10,getNewMethodHeader-return], c2;
d2: GOTO[bailout3], at[4,10,getNewMethodHeader-return], c2;
d3: GOTO[bailout3], at[5,10,getNewMethodHeader-return], c2;
d4: GOTO[bailout3], at[6,10,getNewMethodHeader-return], c2;
d5: GOTO[bailout3], at[7,10,getNewMethodHeader-return], c2;
d6: GOTO[bailout3], at[8,10,getNewMethodHeader-return], c2;
d7: GOTO[bailout3], at[9,10,getNewMethodHeader-return], c2;
d8: GOTO[bailout3], at[0a,10,getNewMethodHeader-return], c2;
d9: GOTO[bailout3], at[0b,10,getNewMethodHeader-return], c2;
d10: GOTO[bailout3], at[0c,10,getNewMethodHeader-return], c2;
d11: GOTO[bailout3], at[0d,10,getNewMethodHeader-return], c2;
d12: GOTO[bailout3], at[0e,10,getNewMethodHeader-return], c2;
d13: GOTO[bailout3], at[0f,10,getNewMethodHeader-return], c2;



d15: GOTO[bailout2], at[1,10,getDeltaWord-return], c1;
d16: GOTO[bailout2], at[2,10,getDeltaWord-return], c1;
d17: GOTO[bailout2], at[3,10,getDeltaWord-return], c1;
d18: GOTO[bailout2], at[4,10,getDeltaWord-return], c1;
d19: GOTO[bailout2], at[5,10,getDeltaWord-return], c1;
d20: GOTO[bailout2], at[6,10,getDeltaWord-return], c1;
d21: GOTO[bailout2], at[7,10,getDeltaWord-return], c1;
d22: GOTO[bailout2], at[8,10,getDeltaWord-return], c1;
d23: GOTO[bailout2], at[9,10,getDeltaWord-return], c1;
d24: GOTO[bailout2], at[0a,10,getDeltaWord-return], c1;
d25: GOTO[bailout2], at[0b,10,getDeltaWord-return], c1;
d26: GOTO[bailout2], at[0c,10,getDeltaWord-return], c1;
d27: GOTO[bailout2], at[0d,10,getDeltaWord-return], c1;
d28: GOTO[bailout2], at[0e,10,getDeltaWord-return], c1;
d29: GOTO[bailout2], at[0f,10,getDeltaWord-return], c1;



d41: GOTO[bailout2], at[0d,10,refdReturn], c1;
d42: GOTO[bailout2], at[0e,10,refdReturn], c1;
d43: GOTO[bailout2], at[0f,10,refdReturn], c1;



d56: GOTO[bailout2], at[8,10,refiReturn], c1;
d57: GOTO[bailout2], at[9,10,refiReturn], c1;
d58: GOTO[bailout2], at[0a,10,refiReturn], c1;
d59: GOTO[bailout2], at[0b,10,refiReturn], c1;

d61: GOTO[bailout2], at[0d,10,refiReturn], c1;
d62: GOTO[bailout2], at[0e,10,refiReturn], c1;
d63: GOTO[bailout2], at[0f,10,refiReturn], c1;



d100: GOTO[bailout3], at[smallMultiply,10,arithmeticPrimitives], c2;
d101: GOTO[bailout3], at[smallDivide,10,arithmeticPrimitives], c2;
d102: GOTO[bailout3], at[smallMod,10,arithmeticPrimitives], c2;

{d104: GOTO[bailout3], at[smallBitShift,10,arithmeticPrimitives], c2;   }
d105: GOTO[bailout3], at[smallDiv,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;
```

```
d209: GOTO[bailout2], at[0a, 10, addToZeroCountTableReturn], c1;
d210: GOTO[bailout2], at[0b, 10, addToZeroCountTableReturn], c1;
d211: GOTO[bailout2], at[0c, 10, addToZeroCountTableReturn], c1;
d212: GOTO[bailout2], at[0d, 10, addToZeroCountTableReturn], c1;
d213: GOTO[bailout2], at[0e, 10, addToZeroCountTableReturn], c1;
d214: GOTO[bailout2], at[0f, 10, addToZeroCountTableReturn], c1;


d302: GOTO[bailout1], at[3, 10, smalltalkState], c3;
d303: GOTO[bailout1], at[4, 10, smalltalkState], c3;
d304: GOTO[bailout1], at[5, 10, smalltalkState], c3;
d305: GOTO[bailout1], at[6, 10, smalltalkState], c3;
d306: GOTO[bailout1], at[7, 10, smalltalkState], c3;
d307: GOTO[bailout1], at[8, 10, smalltalkState], c3;
d308: GOTO[bailout1], at[9, 10, smalltalkState], c3;
d309: GOTO[bailout1], at[0a, 10, smalltalkState], c3;
d310: GOTO[bailout1], at[0b, 10, smalltalkState], c3;
d311: GOTO[bailout1], at[0c, 10, smalltalkState], c3;
d312: GOTO[bailout1], at[0d, 10, smalltalkState], c3;
d313: GOTO[bailout1], at[0e, 10, smalltalkState], c3;
d314: GOTO[bailout1], at[0f, 10, smalltalkState], c3;




d329a: GOTO[bailout2], at[1,10,returnTopOfStack-return], c1;
d330: GOTO[bailout2], at[2,10,returnTopOfStack-return], c1;
d331: GOTO[bailout2], at[3,10,returnTopOfStack-return], c1;
d332: GOTO[bailout2], at[4,10,returnTopOfStack-return], c1;
d333: GOTO[bailout2], at[5,10,returnTopOfStack-return], c1;

d335: GOTO[bailout2], at[7,10,returnTopOfStack-return], c1;
d336: GOTO[bailout2], at[8,10,returnTopOfStack-return], c1;
d337: GOTO[bailout2], at[9,10,returnTopOfStack-return], c1;
d338: GOTO[bailout2], at[0a,10,returnTopOfStack-return], c1;
d339: GOTO[bailout2], at[0b,10,returnTopOfStack-return], c1;
d340: GOTO[bailout2], at[0c,10,returnTopOfStack-return], c1;
d341: GOTO[bailout2], at[0d,10,returnTopOfStack-return], c1;
d342: GOTO[bailout2], at[0e,10,returnTopOfStack-return], c1;



d404: GOTO[bailout2], at[4, 10, makeVolatile-return], c1;
d406: GOTO[bailout2], at[6, 10, makeVolatile-return], c1;
d408: GOTO[bailout2], at[8, 10, makeVolatile-return], c1;
d40a: GOTO[bailout2], at[0a, 10, makeVolatile-return], c1;
d412: GOTO[bailout2], at[0c, 10, makeVolatile-return], c1;
d413: GOTO[bailout2], at[0d, 10, makeVolatile-return], c1;
d414: GOTO[bailout2], at[0e, 10, makeVolatile-return], c1;
d415: GOTO[bailout2], at[0f, 10, makeVolatile-return], c1;




d503: GOTO[bailout2], at[3; 10, transferWords-return], c1;
d504: GOTO[bailout2], at[4, 10, transferWords-return], c1;
d505: GOTO[bailout2], at[5, 10, transferWords-return], c1;
d506: GOTO[bailout2], at[6, 10, transferWords-return], c1;
d507: GOTO[bailout2], at[7, 10, transferWords-return], c1;
d508: GOTO[bailout2], at[8, 10, transferWords-return], c1;
d509: GOTO[bailout2], at[9, 10, transferWords-return], c1;
d510: GOTO[bailout2], at[0a, 10, transferWords-return], c1;
d511: GOTO[bailout2], at[0b, 10, transferWords-return], c1;
d512: GOTO[bailout2], at[0c, 10, transferWords-return], c1;
d513: GOTO[bailout2], at[0d, 10, transferWords-return], c1;
d514: GOTO[bailout2], at[0e, 10, transferWords-return], c1;
d515: GOTO[bailout2], at[0f, 10, transferWords-return], c1;


{d61c: GOTO[bailout2], at[0c, 10, otMap2-return], c1;}
d61e: GOTO[bailout2], at[0e, 10, otMap2-return], c1;
d615: GOTO[bailout2], at[0f, 10, otMap2-return], c1;



d700: GOTO[bailout2], at[0, 10, lastPointerOf-return], c1;
d703: GOTO[bailout2], at[3, 10, lastPointerOf-return], c1;
d704: GOTO[bailout2], at[4, 10, lastPointerOf-return], c1;
d705: GOTO[bailout2], at[5, 10, lastPointerOf-return], c1;
d706: GOTO[bailout2], at[6, 10, lastPointerOf-return], c1;
d707: GOTO[bailout2], at[7, 10, lastPointerOf-return], c1;
d708: GOTO[bailout2], at[8, 10, lastPointerOf-return], c1;
d709: GOTO[bailout2], at[9, 10, lastPointerOf-return], c1;
d710: GOTO[bailout2], at[0a, 10, lastPointerOf-return], c1;
d711: GOTO[bailout2], at[0b, 10, lastPointerOf-return], c1;
d712: GOTO[bailout2], at[0c, 10, lastPointerOf-return], c1;
d713: GOTO[bailout2], at[0d, 10, lastPointerOf-return], c1;
d714: GOTO[bailout2], at[0e, 10, lastPointerOf-return], c1;
d715: GOTO[bailout2], at[0f, 10, lastPointerOf-return], c1;



d802: GOTO[bailout1], at[2, 10, fetchContextRegisters-return], c3;
```

```
d803: GOTO[bailout1], at[3, 10, fetchContextRegisters-return], c3;
d804: GOTO[bailout1], at[4, 10, fetchContextRegisters-return], c3;
d805: GOTO[bailout1], at[5, 10, fetchContextRegisters-return], c3;
d806: GOTO[bailout1], at[6, 10, fetchContextRegisters-return], c3;
d807: GOTO[bailout1], at[7, 10, fetchContextRegisters-return], c3;
d808: GOTO[bailout1], at[8, 10, fetchContextRegisters-return], c3;
d809: GOTO[bailout1], at[9, 10, fetchContextRegisters-return], c3;
d810: GOTO[bailout1], at[0a, 10, fetchContextRegisters-return], c3;
d811: GOTO[bailout1], at[0b, 10, fetchContextRegisters-return], c3;
d812: GOTO[bailout1], at[0c, 10, fetchContextRegisters-return], c3;
d813: GOTO[bailout1], at[0d, 10, fetchContextRegisters-return], c3;
d814: GOTO[bailout1], at[0e, 10, fetchContextRegisters-return], c3;
d815: GOTO[bailout1], at[0f, 10, fetchContextRegisters-return], c3;



d903: GOTO[bailout2], at[3, 10, nextFreeChunk-return], c1;
d904: GOTO[bailout2], at[4, 10, nextFreeChunk-return], c1;
d905: GOTO[bailout2], at[5, 10, nextFreeChunk-return], c1;
d906: GOTO[bailout2], at[6, 10, nextFreeChunk-return], c1;
d907: GOTO[bailout2], at[7, 10, nextFreeChunk-return], c1;
d908: GOTO[bailout2], at[8, 10, nextFreeChunk-return], c1;
d909: GOTO[bailout2], at[9, 10, nextFreeChunk-return], c1;
d910: GOTO[bailout2], at[0a, 10, nextFreeChunk-return], c1;
d911: GOTO[bailout2], at[0b, 10, nextFreeChunk-return], c1;
d912: GOTO[bailout2], at[0c, 10, nextFreeChunk-return], c1;
d913: GOTO[bailout2], at[0d, 10, nextFreeChunk-return], c1;
d914: GOTO[bailout2], at[0e, 10, nextFreeChunk-return], c1;
d915: GOTO[bailout2], at[0f, 10, nextFreeChunk-return], c1;



d956: GOTO[bailout2], at[6, 10, createInstance-return], c1;
d957: GOTO[bailout2], at[7, 10, createInstance-return], c1;
d958: GOTO[bailout2], at[8, 10, createInstance-return], c1;
d959: GOTO[bailout2], at[9, 10, createInstance-return], c1;
d960: GOTO[bailout2], at[0a, 10, createInstance-return], c1;
d961: GOTO[bailout2], at[0b, 10, createInstance-return], c1;
d962: GOTO[bailout2], at[0c, 10, createInstance-return], c1;
d963: GOTO[bailout2], at[0d, 10, createInstance-return], c1;
d964: GOTO[bailout2], at[0e, 10, createInstance-return], c1;
d965: GOTO[bailout2], at[0f, 10, createInstance-return], c1;



d977: GOTO[bailout2], at[7, 10, positive16BitValueOf-return], c1;
d978: GOTO[bailout2], at[8, 10, positive16BitValueOf-return], c1;
d979: GOTO[bailout2], at[9, 10, positive16BitValueOf-return], c1;
d980: GOTO[bailout2], at[0a, 10, positive16BitValueOf-return], c1;
d981: GOTO[bailout2], at[0b, 10, positive16BitValueOf-return], c1;
d982: GOTO[bailout2], at[0c, 10, positive16BitValueOf-return], c1;
d983: GOTO[bailout2], at[0d, 10, positive16BitValueOf-return], c1;
d984: GOTO[bailout2], at[0e, 10, positive16BitValueOf-return], c1;
d986: GOTO[bailout2], at[0f, 10, positive16BitValueOf-return], c1;



d1004: GOTO[bailout3], at[4, 10, fixedFieldsOf-return], c2;
d1005: GOTO[bailout3], at[5, 10, fixedFieldsOf-return], c2;
d1006: GOTO[bailout3], at[6, 10, fixedFieldsOf-return], c2;
d1007: GOTO[bailout3], at[7, 10, fixedFieldsOf-return], c2;
d1008: GOTO[bailout3], at[8, 10, fixedFieldsOf-return], c2;
d1009: GOTO[bailout3], at[9, 10, fixedFieldsOf-return], c2;
d1010: GOTO[bailout3], at[0a, 10, fixedFieldsOf-return], c2;
d1011: GOTO[bailout3], at[0b, 10, fixedFieldsOf-return], c2;
d1012: GOTO[bailout3], at[0c, 10, fixedFieldsOf-return], c2;
d1013: GOTO[bailout3], at[0d, 10, fixedFieldsOf-return], c2;
d1014: GOTO[bailout3], at[0e, 10, fixedFieldsOf-return], c2;
d1015: GOTO[bailout3], at[0f, 10, fixedFieldsOf-return], c2;



d1021: GOTO[bailout2], at[1, 10, getTos-return], c1;
d1022: GOTO[bailout2], at[2, 10, getTos-return], c1;
d1023: GOTO[bailout2], at[3, 10, getTos-return], c1;
d1024: GOTO[bailout2], at[4, 10, getTos-return], c1;
d1025: GOTO[bailout2], at[5, 10, getTos-return], c1;
d1026: GOTO[bailout2], at[6, 10, getTos-return], c1;
d1027: GOTO[bailout2], at[7, 10, getTos-return], c1;
d1028: GOTO[bailout2], at[8, 10, getTos-return], c1;
d1029: GOTO[bailout2], at[9, 10, getTos-return], c1;
d1030: GOTO[bailout2], at[0a, 10, getTos-return], c1;
d1031: GOTO[bailout2], at[0b, 10, getTos-return], c1;
d1032: GOTO[bailout2], at[0c, 10, getTos-return], c1;
d1033: GOTO[bailout2], at[0d, 10, getTos-return], c1;
d1034: GOTO[bailout2], at[0e, 10, getTos-return], c1;
d1035: GOTO[bailout2], at[0f, 10, getTos-return], c1;
```

```
d1041: GOTO[bailout2], at[1, 10, getSmashTos-return], c1;
d1042: GOTO[bailout2], at[2, 10, getSmashTos-return], c1;
d1043: GOTO[bailout2], at[3, 10, getSmashTos-return], c1;
d1044: GOTO[bailout2], at[4, 10, getSmashTos-return], c1;
d1045: GOTO[bailout2], at[5, 10, getSmashTos-return], c1;
d1046: GOTO[bailout2], at[6, 10, getSmashTos-return], c1;
d1047: GOTO[bailout2], at[7, 10, getSmashTos-return], c1;
d1048: GOTO[bailout2], at[8, 10, getSmashTos-return], c1;
d1049: GOTO[bailout2], at[9, 10, getSmashTos-return], c1;
d1050: GOTO[bailout2], at[0a, 10, getSmashTos-return], c1;
d1051: GOTO[bailout2], at[0b, 10, getSmashTos-return], c1;
d1052: GOTO[bailout2], at[0c, 10, getSmashTos-return], c1;
d1053: GOTO[bailout2], at[0d, 10, getSmashTos-return], c1;
d1054: GOTO[bailout2], at[0e, 10, getSmashTos-return], c1;
d1056: GOTO[bailout2], at[0f, 10, getSmashTos-return], c1;


d1067: GOTO[bailout2], at[7, 10, addToFreeChunkList-return], c1;
d1068: GOTO[bailout2], at[8, 10, addToFreeChunkList-return], c1;
d1069: GOTO[bailout2], at[9, 10, addToFreeChunkList-return], c1;
d1080: GOTO[bailout2], at[0a, 10, addToFreeChunkList-return], c1;
d1081: GOTO[bailout2], at[0b, 10, addToFreeChunkList-return], c1;
d1082: GOTO[bailout2], at[0c, 10, addToFreeChunkList-return], c1;
d1083: GOTO[bailout2], at[0d, 10, addToFreeChunkList-return], c1;
d1084: GOTO[bailout2], at[0e, 10, addToFreeChunkList-return], c1;
d1086: GOTO[bailout2], at[0f, 10, addToFreeChunkList-return], c1;


d1101: GOTO[bailout2], at[1, 10, deallocate-return], c1;
d1102: GOTO[bailout2], at[2, 10, deallocate-return], c1;
d1103: GOTO[bailout2], at[3, 10, deallocate-return], c1;
d1104: GOTO[bailout2], at[4, 10, deallocate-return], c1;
d1105: GOTO[bailout2], at[5, 10, deallocate-return], c1;
d1106: GOTO[bailout2], at[6, 10, deallocate-return], c1;
d1107: GOTO[bailout2], at[7, 10, deallocate-return], c1;
d1111: GOTO[bailout2], at[8, 10, deallocate-return], c1;
d1109: GOTO[bailout2], at[9, 10, deallocate-return], c1;
d1110: GOTO[bailout2], at[0a, 10, deallocate-return], c1;
d1121: GOTO[bailout2], at[0b, 10, deallocate-return], c1;
d1122: GOTO[bailout2], at[0c, 10, deallocate-return], c1;
d1123: GOTO[bailout2], at[0d, 10, deallocate-return], c1;
d1124: GOTO[bailout2], at[0e, 10, deallocate-return], c1;
d1126: GOTO[bailout2], at[0f, 10, deallocate-return], c1;


d1132: GOTO[bailout2], at[2, 10, stabilize-return], c1;
d1133: GOTO[bailout2], at[3, 10, stabilize-return], c1;
d1134: GOTO[bailout2], at[4, 10, stabilize-return], c1;
d1135: GOTO[bailout2], at[5, 10, stabilize-return], c1;
d1136: GOTO[bailout2], at[6, 10, stabilize-return], c1;
d1137: GOTO[bailout2], at[7, 10, stabilize-return], c1;
d1138: GOTO[bailout2], at[8, 10, stabilize-return], c1;
d1139: GOTO[bailout2], at[9, 10, stabilize-return], c1;
d113a: GOTO[bailout2], at[0a, 10, stabilize-return], c1;
d113b: GOTO[bailout2], at[0b, 10, stabilize-return], c1;
d113c: GOTO[bailout2], at[0c, 10, stabilize-return], c1;
d113d: GOTO[bailout2], at[0d, 10, stabilize-return], c1;
d113e: GOTO[bailout2], at[0e, 10, stabilize-return], c1;
d113f: GOTO[bailout2], at[0f, 10, stabilize-return], c1;


d1146: GOTO[bailout2], at[6, 10, getObjectSize-return], c1;
d1147: GOTO[bailout2], at[7, 10, getObjectSize-return], c1;
d1148: GOTO[bailout2], at[8, 10, getObjectSize-return], c1;
d1149: GOTO[bailout2], at[9, 10, getObjectSize-return], c1;
d114a: GOTO[bailout2], at[0a, 10, getObjectSize-return], c1;
d114b: GOTO[bailout2], at[0b, 10, getObjectSize-return], c1;
d114c: GOTO[bailout2], at[0c, 10, getObjectSize-return], c1;
d114d: GOTO[bailout2], at[0d, 10, getObjectSize-return], c1;
d144e: GOTO[bailout2], at[0e, 10, getObjectSize-return], c1;
d114f: GOTO[bailout2], at[0f, 10, getObjectSize-return], c1;


d1151: GOTO[bailout1], at[1, 10, getByteOrAddress-return], c3;

d1154: GOTO[bailout1], at[4, 10, getByteOrAddress-return], c3;
d1155: GOTO[bailout1], at[5, 10, getByteOrAddress-return], c3;
d1156: GOTO[bailout1], at[6, 10, getByteOrAddress-return], c3;
d1157: GOTO[bailout1], at[7, 10, getByteOrAddress-return], c3;
d1158: GOTO[bailout1], at[8, 10, getByteOrAddress-return], c3;
d1159: GOTO[bailout1], at[9, 10, getByteOrAddress-return], c3;
d115a: GOTO[bailout1], at[0a, 10, getByteOrAddress-return], c3;
d115b: GOTO[bailout1], at[0b, 10, getByteOrAddress-return], c3;
d115c: GOTO[bailout1], at[0c, 10, getByteOrAddress-return], c3;
d115d: GOTO[bailout1], at[0d, 10, getByteOrAddress-return], c3;
```

```
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,                              c1;
        temp2Low ← uRumRecordLow,                                c2;
        Noop,                                                    c3;

        MAR ← [temp2High, temp2Low + directiveOffset],           c1;
        MDR ← 2, CANCELBR [$, 2], LOOPHOLE [wok],                c2;
        GOTO[restoreMesaState],                                  c3;
```

```
{
        18-Jul-84 15:53:41
}


    getSmalltalkState:
            Q ← uRumRecordHigh {retrieve the Rum Record address},                c1;
            temp3High ← Q LRot0,                                                 c2;
            temp3Low ← uRumRecordLow,                                            c3;

                        {get the method cache address}
            MAR ← [temp3High, temp3Low + methodCacheLowOffset],                  c1;
            CANCELBR[$, 2], Noop,                                                c2;
            Q ← MD,                                                              c3;

            MAR ← [temp3High, temp3Low + methodCacheHighOffset],                 c1;
            CANCELBR[$, 2], uMethodCacheLow ← Q,                                 c2;
            Q ← MD,                                                              c3;

            {get the Object Table address}
            MAR ← [temp3High, temp3Low + objectTableHighOffset],                 c1;
            CANCELBR[$, 2], uMethodCacheHigh ← Q,                                c2;
            otHigh ← MD,                                                         c3;

            {get the oop of the active context, otmap it, save oop and base address}
            MAR ← [temp3High, temp3Low + activeContextOopOffset],
                  L1 ← gettingActiveContextDuringInterpreterSwap,                c1;
            CANCELBR[$, 2],                                                      c2;
            otLow ← MD, CALL[otMap2],                                            c3;

            Q ← temp1High,                                                       c1, at[gettingActiveContextDuringInterpreterSwap,
            10, otMap2-return];
            uActiveContextHigh ← Q,                                              c2;
            Q ← temp1Low,                                                        c3;

            uActiveContextLow ← Q,                                               c1;
            Q ← otLow,                                                           c2;
            uActiveContextOop ← Q,                                               c3;

            {get current Stack Pointer}
            MAR ← [temp3High, temp3Low + stackPointerLowOffset],                 c1;
            CANCELBR[$, 2], Noop,                                                c2;
            stackLow ← MD,                                                       c3;

            MAR ← [temp3High, temp3Low + stackPointerHighOffset],                c1;
            CANCELBR[$, 2], Noop,                                                c2;
            stackHigh ← MD,                                                      c3;

            {get Home Context Oop, otMap it, save its address}
            MAR ← [temp3High, temp3Low + homeContextOopOffset], L1 ← gettingSmalltalkState, c1;
            CANCELBR[$, 2],                                                      c2;
            otLow ← MD, CALL[otMap],                                             c3;

            uHomeLow ← temp1Low,                                                 c1, at[gettingSmalltalkState, 10, otMap-return];
            Q ← temp1High,                                                       c2;
            homeHigh  ← Q LRot0,                                                 c3;

            {get the oop of current receiver, save it and if it is not a SmallInteger, otMap it and save its address }
            MAR ← [temp3High, temp3Low + receiverOopOffset],                     c1;
            CANCELBR[$, 2],  homeLow ← uHomeLow,                                 c2;
            otLow ← MD,                                                          c3;

            uTimeToStabilize ← 0,                                                c1;
            uReceiverOop ← otLow, YDisp,                                         c2;
            BRANCH[isOop-getSmalltalkState, isSmall-getSmalltalkState, 0e],      c3;

    isOop-getSmalltalkState:
            L1 ← isOopGettingSmalltalkState,                                     c1;
            Noop,                                                                c2;
            CALL[otMap],                                                         c3;

            Q ← temp1High,                                                       c1, at[isOopGettingSmalltalkState, 10,
            otMap-return];
            uReceiverHigh ← Q,                                                   c2;
            Q ← temp1Low,                                                        c3;

    isSmall-getSmalltalkState:

      {get the oop of the current method, otMap it, set up the machine's instruction pointer registers}
            MAR ← [temp3High, temp3Low + currentMethodOopOffset], L1 ← isSmallGettingSmalltalkState,c1;
            CANCELBR[$, 2],  uReceiverLow ← Q,                                   c2;
            otLow ← MD,  CALL[otMap],                                            c3;


            Q ← temp1High,                                                       c1, at[isSmallGettingSmalltalkState , 10,
            otMap-return];
            ipHigh ← Q LRot0,                                                    c2;
            ipLow ← temp1Low + objectHeaderSize,                                 c3;

            uCurrentMethodHigh ← Q,                                              c1;
```

```
        Q ← temp1Low,                                                   c2;
        uCurrentMethodLow ← Q,                                          c3;

        MAR ← [temp3High, temp3Low + instructionPointerOffset],         c1;
        CANCELBR[$, 2], Noop,                                           c2;
        temp3Low ← MD,                                                  c3;

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, XC2npcDisp {see what state pc16 is--we want it zero},c1;
        ipLow ← ipLow + temp2Low {add in word offset}, BRANCH[flip, noFlip, 0e],c2;

flip:
        Cin ← pc16 {make it zero}, GOTO[pc16isZeroNow],                 c3;
noFlip:
        GOTO[pc16isZeroNow],                                            c3;

pc16isZeroNow:
        MAR ← [ipHigh, ipLow+ 0] {read a word of bytecodes},            c1;
        [] ← temp3Low{determine desired state of pc16}, YDisp,          c2;
        IB ← MD {load up instruction buffer}, BRANCH[leaveItBe, makeIt1, 0e],  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,                                                   c2;
        DISPNI[bytecodes],                                              c3;

saveSmalltalkState:

        {we come here when Rum 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 ← uRumRecordHigh {retrieve the Rum Record address},           c1;
        temp3High ← Q LRot0,                                            c2;
        temp3Low ← uRumRecordLow,                                       c3;

                {write the current stack pointer}
        MAR ← [temp3High, temp3Low + stackPointerLowOffset],            c1;
        CANCELBR[$, 2], LOOPHOLE[wok], MDR ← stackLow,                  c2;
        Noop,                                                           c3;

        MAR ← [temp3High, temp3Low + stackPointerHighOffset],           c1;
        CANCELBR[$, 2], LOOPHOLE[wok], MDR ← stackHigh,                 c2;

        {adjust the instruction pointer to make Molasses happy, then write the instructionPointer}

        ipLow ← ipLow - objectHeaderSize,                               c3;

        temp2Low ← uCurrentMethodLow,                                   c1;
        ipLow ← ipLow - temp2Low,                                       c2;
        ipLow ← LShift1 ipLow,  SE ← pc16,                              c3;

        [] ← temp1Low, YDisp,                                           c1;
        DISP4[smalltalkState],   LODisp {in case of bytecode failure},  c2;

        CANCELBR[writeIp, 0f],                                          c3, at[0, 10, smalltalkState];

        CANCELBR[writeIp, 0f],                                          c3, at[1, 10, smalltalkState];

        temp1Low ← 1 {tell Molasses to execute this bytecode}, DISP2[ipAdjustment] {bytecode failed -- need to fix up inst
        ptr},                                                           c3, at[2
```

```
                GOTO[ipAdjusted],                                       c1, at[0, 4, ipAdjustment];

                ipLow ← ipLow - 1, GOTO[ipAdjusted]                     c1, at[1, 4, ipAdjustment];

                ipLow ← ipLow - 2, GOTO[ipAdjusted]                     c1, at[2, 4, ipAdjustment];

                ipLow ← ipLow - 3, GOTO[ipAdjusted]                     c1, at[3, 4, ipAdjustment];
        ipAdjusted:
                Noop,                                                   c2;
                Noop,                                                   c3;

        writeIp:
                MAR ← [temp3High , temp3Low + instructionPointerOffset], c1;
                CANCELBR[$, 2], LOOPHOLE[wok], MDR ← ipLow ,            c2;
                Noop,                                                   c3;

                MAR ← [temp3High, temp3Low + directiveOffset],          c1;
                CANCELBR[$, 2], LOOPHOLE[wok], MDR ← temp1Low,          c2;
                GOTO[restoreMesaState],                                 c3;

{todo --- save current method oop}
```

```
{
        1-Aug-84 18:35:43
}

        Q ← 7e,         GOTO[badBytecode],      c1, bytecode[7e];
        Q ← 7f,         GOTO[badBytecode],      c1, bytecode[7f];
        Q ← 8a,         GOTO[badBytecode],      c1, bytecode[8a];
        Q ← 8b,         GOTO[badBytecode],      c1, bytecode[8b];
        Q ← 8c,         GOTO[badBytecode],      c1, bytecode[8c];
        Q ← 8d,         GOTO[badBytecode],      c1, bytecode[8d];
        Q ← 8e,         GOTO[badBytecode],      c1, bytecode[8e];
        Q ← 8f,         GOTO[badBytecode],      c1, bytecode[8f];

    badBytecode:
            GOTO[bailout3],                     c2;
```