```
**Dorado Smalltalk Microcode--A Semi-Literal translation from Byterp.Mu
**Last Edited:    October 12, 1978  10:15 AM

** By convention, labels starting with '.' are used only within the immediate vicinity
** (same subroutine or other local piece of code).

Title[Dsmall.Mc];

Top Level;                  *Can smash Link--when calling two+ levels deep, be sure
                            *to change the level appropriately

*Define the Macros
M[DontKnowR, KnowRBase[BBRegs]];      *i.e. force use of FF on stores

*Set up dispatch addresses in IM
Set[Dispatch1, 4700];       *20-way dispatch (OpCodeDispatch)
Set[Dispatch2, 4600];       *10-way dispatch (OpsDispatch)
Set[Dispatch3, 6400];       *20-way dispatch on bits 8:11 (ByteTypeD)
Set[Dispatch4, 4100];       *20-way dispatch (PrimOpsDispatch)
Set[Dispatch5, 4000];       *10-way dispatch (ArithDispatch)
Set[Dispatch6, 3700];       *20-way dispatch (Omsgs)
Set[Dispatch7, 3600];       *10-way dispatch (DeltaDispatch)
Set[Dispatch8, 6000];       *10-way dispatch (Returns)
Set[Dispatch9, 5100];       *was 3400
```

```
*ByteCode Interpreter--Dorado microcode translated from Byterp.Mu[Alto]

   KnowRbase[AEMRegs];          *Make sure that the assembler knows RBase=AemRegs

* EMStart is the entry point to the Alto emulator

EMStart:
   MemBase ← AemMemBase;
   RBase ← RBase[AemRegs];
   Branch[Start];               *Start is the emulator's actual entry point

*EMTrap is the entry point to the Emulator trap

EMTrap:
   MemBase ← AemMemBase;
   RBase ← RBase[AemRegs];
   Branch[Trap];                *Go to emulator trap

* Entry from Alto emulator on trapped Nova instructions to microcode

TrapX:
   KnowRBase[AEMRegs];          *Just came from emulator
   T ← Ldf[Ireg, 4, 10];        *Ireg has actual instruction//load low 4 bits
   B ← T, BigBDispatch;
   Branch[OpCodeDispatch];

OpCodeDispatch:
   Branch[Extract],     At[Dispatch1, 0];     *700xx
   Branch[Inject],      At[Dispatch1, 1];     *704xx
   Branch[RefCt],       At[Dispatch1, 2];     *710xx
   Branch[Trap],        At[Dispatch1, 3];     *714xx
   Branch[Sundry],      At[Dispatch1, 4];     *720xx
   Branch[Sundry2],     At[Dispatch1, 5];     *724xx
   Branch[Trap],        At[Dispatch1, 6];
   Branch[Trap],        At[Dispatch1, 7];
   Branch[Trap],        At[Dispatch1, 10];
   Branch[Trap],        At[Dispatch1, 11];
   Branch[Byterp],      At[Dispatch1, 12];    *750xx
   Branch[Trap],        At[Dispatch1, 13];
   Branch[Trap],        At[Dispatch1, 14];
   Branch[Trap],        At[Dispatch1, 15];
   Branch[NovaRet],     At[Dispatch1, 16];    *770xx
   Branch[Trap],        At[Dispatch1, 17];

Sundry:
   T ← Ldf[Ireg, 3, 0];         *want low three bits (ignore 3 higher bits)
   B ← T, BDispatch;            *72000 ops dispatch here, i.e. Nova special ops
   Branch[OpsDispatch];

OpsDispatch:
   Branch[IvalTP],      At[Dispatch2, 0];
   Branch[HashT],       At[Dispatch2, 1];
   Branch[Snat],        At[Dispatch2, 4];

Sundry2:
   A ← Ireg, DblBranch[WriteR, ReadR, R odd];     *72400=Readr, 72401=WriteR
```

***NovaCall and NovaRet--Call to Nova from microcode and return**

```
NovaCall:
   RBase ← RBase[AemRegs];
   MemBase ← AemMemBase;
   T ← T + (400c);                  *Load trap op from T
   Xreg ← T + (130c);
   T ← Ireg;
   SavDisp ← T;
   Fetch ← Xreg;
   Pc ← Md, Branch[EmStart];        *jump into trap vector (to Nova Instrs.)


NovaRet:
   KnowRbase[AEMRegs];
   T ← Ldf[Ireg, 4, 0];             *Get low 4 bits
   Xreg ← T, RBase ← RBase[SavDisp];
   T ← SavDisp, RBase ← RBase[AemRegs];
   Ireg ← T;
   B ← Xreg, BDispatch;
   Branch[NovaReturns];

NovaReturns:
   Branch[OvRet],          At[Dispatch8, 0];
   Branch[FiRet],          At[Dispatch8, 1];
   Branch[FltRet],         At[Dispatch8, 2];
   Branch[AllocRet],       At[Dispatch8, 3];
   Branch[PrimFail],       At[Dispatch8, 4];
   Branch[SndMsg],         At[Dispatch8, 5];
   Branch[SupRet],         At[Dispatch8, 6];
   Branch[PrimRet],        At[Dispatch8, 7];


FetchTPc:
   DontKnowR;                       *Force use of FF
   Fetch ← T;
   Pc ← Md, Branch[EMStart];        *uses FF
```

***ReadR/WriteR: Read an R register, with argument in Ac0--returns in/stores Ac1**

```
ReadR:
   KnowRBase[AemRegs];
   B ← Ac0, BigBDispatch;
   Xreg ← T - T, Branch[.RRegs];    *0 means read
```

*WriteR: Write an R register, with R# in Ac0, data in Ac1

```
WriteR:
   KnowRBase[AemRegs];
   T ← Ac1;
   B ← Ac0, BigBDispatch;
   Xreg ← (1c), Branch[.RRegs];     *1 means write

.RRegs:
   Nop,                    At[Dispatch9, 0];     *to make dispatch work
   Branch[.RegCoreBase],   At[Dispatch9, 14];
```

```
    Branch[.RegPcb],            At[Dispatch9, 15];
    Branch[.RegAOop],           At[Dispatch9, 17];
    Branch[.RegName],           At[Dispatch9, 44];
    Branch[.RegACore],          At[Dispatch9, 52];
    Branch[.RegFather],         At[Dispatch9, 53];
    Branch[.RegMinAt],          At[Dispatch9, 54];
    Branch[.RegPmBase],         At[Dispatch9, 55];
    Branch[.RegRotBase],        At[Dispatch9, 56];
    Branch[.RegArec],           At[Dispatch9, 60];
    Branch[.RegSavDisp],        At[Dispatch9, 62];
    Branch[.RegBCore],          At[Dispatch9, 63];
    Branch[.RegBOop],           At[Dispatch9, 65];
    Branch[.RegLocFrame],       At[Dispatch9, 66];
    Branch[.RegStackP],         At[Dispatch9, 67];
    Branch[.RegTop],            At[Dispatch9, 70];
    Branch[.RegCaddr],          At[Dispatch9, 71];
    Branch[.RegTFrame],         At[Dispatch9, 72];
    Branch[.RegSelf],           At[Dispatch9, 73];
    Branch[.RegSupMod],         At[Dispatch9, 74];
    Branch[.RegCtxt],           At[Dispatch9, 76];

.RegCoreBase:
    KnowRBase[AEMRegs];
    MemBase ← CoreBaseBr, A ← Xreg, DblBranch[.RBr, .WBr, R Even];


.RegPcb:
    KnowRBase[AEMRegs];
    A ← XReg, DblBranch[.R1, .W1, R Even];
.W1:
    Pcb ← T, Branch[.JumpOut];
.R1:
    RBase ← RBase[Pcb];
    T ← Pcb, Branch[.ReadOut];


.RegAOop:
    KnowRBase[AEMRegs];
    A ← XReg, DblBranch[.R2, .W2, R Even];
.W2:
    AOop ← T, Branch[.JumpOut];
.R2:
    RBase ← RBase[AOop];
    T ← AOop, Branch[.ReadOut];


.RegName:
    KnowRBase[AEMRegs];
    A ← XReg, DblBranch[.R3, .W3, R Even];
.W3:
    Name ← T, Branch[.JumpOut];
.R3:
    RBase ← RBase[Name];
    T ← Name, Branch[.ReadOut];


.RegAcore:
```

```
    KnowRBase[AEMRegs];
    MemBase ← AcoreBr, A ← XReg, DblBranch[.RBr, .WBr, R Even];

.RegFather:
    KnowRBase[AEMRegs];
    A ← XReg, DblBranch[.R5, .W5, R Even];
.W5:
    Father ← T, Branch[.JumpOut];
.R5:
    RBase ← RBase[Father];
    T ← Father, Branch[.ReadOut];

.RegMinAt:
    KnowRBase[AEMRegs];
    A ← XReg, DblBranch[.R6, .W6, R Even];
.W6:
    MinAt ← T, Branch[.JumpOut];
.R6:
    RBase ← RBase[MinAt];
    T ← MinAt, Branch[.ReadOut];

.RegPmBase:
    KnowRBase[AEMRegs];
    MemBase ← PmBaseBr, A ← XReg, DblBranch[.RBr, .WBr, R Even];

.RegRotBase:
    KnowRBase[AEMRegs];
    MemBase ← RotBaseBr, A ← XReg, DblBranch[.RBr, .WBr, R Even];

.RegArec:
    KnowRBase[AEMRegs];
    MemBase ← ArecBr, A ← XReg, DblBranch[.RBr, .WBr, R Even];

.RegSavDisp:
    KnowRBase[AEMRegs];
    A ← XReg, DblBranch[.R10, .W10, R Even];
.W10:
    SavDisp ← T, Branch[.JumpOut];
.R10:
    RBase ← RBase[SavDisp];
    T ← SavDisp, Branch[.ReadOut];

.RegBcore:
    KnowRBase[AEMRegs];
    MemBase ← BcoreBr, A ← XReg, DblBranch[.RBr, .WBr, R Even];

.RegBOop:
    KnowRBase[AEMRegs];
    A ← XReg, DblBranch[.R12, .W12, R Even];
.W12:
    BOop ← T, Branch[.JumpOut];
.R12:
    RBase ← RBase[BOop];
    T ← BOop, Branch[.ReadOut];


.RegLocFrame:
    KnowRBase[AEMRegs];
```

```
   MemBase ← LocFrameBr, A ← XReg, DblBranch[.RBr, .WBr, R Even];

.RegStackP:
   KnowRBase[AEMRegs];
   A ← XReg, DblBranch[.R14, .W14, R Even];
.W14:
   StackP ← T, Branch[.JumpOut];
.R14:
   RBase ← RBase[StackP];
   T ← StackP, Branch[.ReadOut];

.RegTop:
   KnowRBase[AEMRegs];
   A ← XReg, DblBranch[.R15, .W15, R Even];
.W15:
   Top ← T, Branch[.JumpOut];
.R15:
   RBase ← RBase[Top];
   T ← Top, Branch[.ReadOut];


.RegCaddr:
   KnowRBase[AEMRegs];
   A ← XReg, DblBranch[.R16, .W16, R Even];
.W16:
   Caddr ← T, Branch[.JumpOut];
.R16:
   RBase ← RBase[Caddr];
   T ← Caddr, Branch[.ReadOut];


.RegTFrame:
   KnowRBase[AEMRegs];
   MemBase ← TFrameBr, A ← XReg, DblBranch[.RBr, .WBr, R Even];

.RegSelf:
   KnowRBase[AEMRegs];
   MemBase ← SelfBr, A ← XReg, DblBranch[.RBr, .WBr, R Even];

.RegSupMod:
   KnowRBase[AEMRegs];
   A ← XReg, DblBranch[.R19, .W19, R Even];
.W19:
   SupMod ← T, Branch[.JumpOut];
.R19:
   RBase ← RBase[SupMod];
   T ← SupMod, Branch[.ReadOut];


.RegCtxt:
   KnowRBase[AEMRegs];
   A ← XReg, DblBranch[.R20, .W20, R Even];
.W20:
   Ctxt ← T, Branch[.Initialize];
.R20:
   RBase ← RBase[Ctxt];
   T ← Ctxt, Branch[.ReadOut];
```

```
.RBr:
   KnowRBase[AEMRegs];
   MyTemp ← T - T, TaskingOff;
   B ← Md;              *So it wont HOLD on me
   DummyRef ← MyTemp;
   Ac0 ← Pipe1;
   TaskingOn, Branch[.JumpOut];
.WBr:
   BrLo ← Ac1, Branch[.JumpOut];

.ReadOut:
   RBase ← RBase[AemRegs];        *Back to the emulator registers (No need for FF)
   Ac0 ← T;

.JumpOut:
   T ← Ac3;                       *Jmp @.return (emulator regs still in RBase)
   Pc ← T, Branch[EMStart];       *Return to Alto Emulator
```

***Initialize: set all base register high bytes to zero (i.e., use low 64K)**

```
.Initialize:
   T ← T - T;
   MemBase ← CoreBaseBr,      Call[.LoadBrHi];
   MemBase ← PmBaseBr,        Call[.LoadBrHi];
   MemBase ← RotBaseBr,       Call[.LoadBrHi];
   MemBase ← LocFrameBr,      Call[.LoadBrHi];
   MemBase ← ArecBr,          Call[.LoadBrHi];
   MemBase ← TFrameBr,        Call[.LoadBrHi];
   MemBase ← SelfBr,          Call[.LoadBrHi];
   MemBase ← ACoreBr,         Call[.LoadBrHi];
   MemBase ← BCoreBr,         Call[.LoadBrHi];
   Branch[.JumpOut];

.LoadBrHi:
   BrHi ← T, Return;
```

***Nova Hash trap**

```
HashT:
   KnowRBase[AEMRegs];
   T ← Pc, RBase ← RBase[SavPc];
   SavPc ← T, Call[Hash];
      AT[NHRetLoc!],          *So failure will go to HFail0
   RBase ← RBase[SavPc];
   SavPc ← (SavPc) + 1;
   RBase ← RBase[Residue];
   T ← Residue;
```

***Load the AC's for the Nova**

```
LoadAcs:
   RBase ← RBase[AemRegs];
   Ac3 ← T;                   *Ac3 ← old Rot0 for hits, Residue for misses
   RBase ← RBase[Rpc];
   T ← Rpc, RBase ← RBase[AemRegs];
   Ac1 ← T, TaskingOff;       *So that DummyRef ← works
   RBase ← RBase[RotA];
```

```
   MemBase ← RotBaseBr;              *Must add RotBase to RotAddr.
   B ← Md;                           *So that it won't HOLD on me
   DummyRef ← RotA;
   RBase ← RBase[AemRegs];
   Ac2 ← Pipe1;
   TaskingOn;
   RBase ← RBase[SavPc];
   T ← SavPc, RBase ← RBase[AemRegs];          *EmStart restores Emulator BR

Spc:
   Pc ← T, Branch[EMStart];

HFail0:
   RBase ← RBase[Residue];
   T ← Rsh[Residue, 13], Branch[LoadAcs];      *Ac3 ← Res for miss (Insert)
```

***Nova RefCt trap**

```
RefCt:
   RBase ← RBase[Pc];
   T ← Pc;
   A ← Ireg, RBase ← RBase[SavSp], Branch[.NRefD, R Odd];
   Nop;                              *Placement
   SavSp ← T, Call[RefCkInc];        *Ireg even, do RefI
   Branch[RsSp];

.NRefD:
   SavSp ← T, Call[RefCkDec];        *Ireg odd, do RefD

RsSp:
   RBase ← RBase[SavSp];
   T ← SavSp, RBase ← RBase[AemRegs], Branch[Spc];
```

***Nova Ival trap**

```
IvalTp:                              *Assumes RBase = AemRegs
   KnowRBase[AemRegs];
   T ← Pc, RBase ← RBase[SavSp];
   SavSp ← T, Call[Ival];
      AT[NIRetLoc!],                 *So failure in Ival will go to PrimFail
   RBase ← RBase[AemRegs];
   Ac0 ← T, Branch[RsSp];
```

***Skip not Atom**

```
Snat:
   T ← Ac0, RBase ← RBase[MinAt];
   Lu ← T - (MinAt);
   RBase ← RBase[AemRegs], Branch[EMStart, Carry];

   Pc ← (Pc) + 1, Branch[EMStart];
```

***Extract // 70000 Disp = < Width > < Shift >**

```
Extract:
   Branch[EMTrap];          *Emulator's trap
```

*Inject // 70400 Disp = < Width > < Shift >

```
Inject:
    Branch[EMTrap];          *Emulator's trap
```

```
* The Bytecode Interpreter proper

Byterp:
   DontKnowR;                  *force use of FF
   T ← T - T - 1;              *T ← (-1c)
   SupMod ← T;
   RBase ← RBase[State];       *Free up FF in next two instructions
   AOop ← T;
   BOop ← T;
   DontKnowR;                  *Force use of FF
   Mode ← T - T, Branch[NextByte];

*Interpret next bytecode

NextByte:
   MemBase ← AemMemBase;                                  *Absolute addressing
   RBase ← RBase[Nww];
   Lu ← Nww, RBase ← RBase[Caddr], Branch[.NoInt1, R < 0]; *Interrupts disabled?
   Fetch ← Caddr, Branch[.DoInt, Alu#0];                  *Wakeups waiting?

NextNoInt:                     * Smash ops enter here.  Fetch ← Caddr pending
   Pcb ← (Pcb) + 1, T ← Md, DblBranch[.LByte, .RByte, R Even];   *Test which byte

.NoInt1:
   Fetch ← Caddr, Branch[NextNoInt];

.DoInt:
   T ← Md, RBase ← RBase[AemRegs];        *finish memory reference
   Pc ← (2c), Branch[EMStart];            *Back to Alto emulator for interrupt

.LByte:
   RBase ← RBase[AemRegs];                *wait for T to load
   Ireg ← Rsh[T, 10], Branch[Dispatch];   *Push left byte over

   KnowRbase[State];                      *Hope that State's region is in control now

.RByte:
   Caddr ← (Caddr) + 1;
   RBase ← RBase[AemRegs];
   Ireg ← T and (377c);                   *use right byte

Dispatch:
   Nop;                                   *So that pipelining won't smash T
   RBase ← RBase[Ireg];                   *in case it came from a superclass call
   T ← Rsh[Ireg, 4];
   B ← T, BigBIDispatch;                  *Dispatch on high 4 bits of low byte
   T ← (Ireg) and (17c), Branch[ByteTypeD]; *T is low four bits

ByteTypeD:
   Branch[Ivars],      At[Dispatch3, 0];
   Branch[Tvars],      At[Dispatch3, 1];
   Branch[Lvars],      At[Dispatch3, 2];
   Branch[Lvars2],     At[Dispatch3, 3];
   Branch[IlVars],     At[Dispatch3, 4];
   Branch[IlVars2],    At[Dispatch3, 5];
   Branch[IlVars3],    At[Dispatch3, 6];
   Branch[Avars],      At[Dispatch3, 7];
   Branch[PrimOps],    At[Dispatch3, 10];
```

```
   Branch[ShortJumps],     At[Dispatch3, 11];
   Branch[LongJumps],      At[Dispatch3, 12];
   Branch[ATrapMsgs],      At[Dispatch3, 13];
   Branch[OTrapMsgs],      At[Dispatch3, 14];
   Branch[LitMsgs],        At[Dispatch3, 15];
   Branch[LitMsgs2],       At[Dispatch3, 16];
   Branch[LitMsgs3],       At[Dispatch3, 17];

*Load data for stack

Ivars:
   MemBase ← SelfBr, Branch[Lmem];          *Load relative to self

Tvars:
   MemBase ← TFrameBr, Branch[Lmem];        *Load relative to tempframe

Lvars2:
   T ← (T) + (20c);       *Offset by 20c
Lvars:
   MemBase ← LocFrameBr, Branch[Lmem];      *Load relative to localframe

IlVars2:
   T ← (T) + (20c), Branch[Ilvars];
IlVars3:
   T ← (T) + (40c);
Ilvars:
   MemBase ← LocFrameBr, MyTemp ← T;
   Fetch ← MyTemp;                          *load @ localframe (Fetch ← RM doesn't use FF)
   Arg1 ← Md, Call[Hash];                   *Hash has argument in Arg1
   Call[Dirty];                             *Hash indirect literal object reference, dirty
   MemBase ← AemMemBase, Branch[Lmem];

Avars:
   Lu ← (Ireg) and (10c);                   *Load relative to Arec
   Branch[.Lmem1, Alu=0];
   T ← T + (Casemc.c);
   MemBase ← AemMemBase, Branch[Lmem]; *System constant, not Arec

.Lmem1:
   MemBase ← ArecBr;

Lmem:
   RBase ← RBase[Mode];
   A ← Mode, DblBranch[Push, Pull, R Even];

Push:                                       *Stack to Data
   Fetch ← T;
   RBase ← RBase[StackP];
   Nop;
   StackP ← (StackP) + 1, T ← Md;

Shove:                                      *Replace top of stack with T, same as pop and push
   RBase ← RBase[Top];
   Top ← T, Call[RefCkLInc];

RefX4:
   RBase ← RBase[StackP];                   *Came from region with Top
   T ← StackP;
```

```
*Stor:
*T = new stack pointer
*Top = New oop
*Tempframe[Stackp ← T] ← Top

Stor:
    RBase ← RBase[StackP];
    StackP ← T;
    MemBase ← TFrameBr;
    Fetch ← StackP;
    RBase ← RBase[AemRegs];
    Arg1 ← Md;
    RBase ← RBase[Top];
    Store ← T, Md ← Top, Call[RefCkDec];
    Branch[Byterp];

*Dispatch on the primitive operations like smash, pop, etc.

PrimOps:
    B ← T, BigBDispatch;     *T = Ireg and (17c)
    RBase ← RBase[State], Branch[PrimOpsDispatch];

PrimOpsDispatch:
    Branch[SmashPop],        At[Dispatch4, 0];
    Branch[Smash],           At[Dispatch4, 1];
    Branch[Pop],             At[Dispatch4, 2];
    Branch[Rtrn],            At[Dispatch4, 3];   * Can't call it "Return"
    Branch[Rend],            At[Dispatch4, 4];
    Branch[Current],         At[Dispatch4, 5];
    Branch[Super],           At[Dispatch4, 6];
    Branch[SendAgain],       At[Dispatch4, 7];
    Branch[XIVars],          At[Dispatch4, 10];
    Branch[XTVars],          At[Dispatch4, 11];
    Branch[XLVars],          At[Dispatch4, 12];
    Branch[XILVars],         At[Dispatch4, 13];
    Branch[XLitMsgs],        At[Dispatch4, 14];

*SmashPop:
*Top = Data to be stored
*Store T into location described by next byte

SmashPop:
    RBase ← RBase[StackP];
    StackP ← (StackP) - 1, RBase ← RBase[Mode];

Smash:
    Mode ← (1c);
    MemBase ← AemMemBase;
    T ← (Fetch ← Caddr), Branch[NextNoInt];        *Fetch instruction word

*Pull Stack to Data--T+MemBase=Addr. of data, Top=new oop to be stored in data

Pull:
    Fetch ← T;
    RBase ← RBase[Arg1];
    Arg1 ← Md;
    RBase ← RBase[Top];
```

```
   Store ← T, Md ← Top, Call[RefCkDec];      *RefD old oop
   RBase ← RBase[Top];
   T ← Top, Call[RefCkLInc];                 *Do it again, RefI new oop
   RBase ← RBase[Model;
   A ← Mode, Mode ← (0c), Branch[NextByte, R Odd];
   Branch[RiSubEnd];                         *because of placement constraint


Pop:
   RBase ← RBase[StackP];
   StackP ← (StackP) - 1, Branch[NextByte];

*Return to sender

Rtrn:
   MemBase ← ArecBr;
   T ← (Senderf.c);
   Fetch ← T, Branch[DoReturn];      *Address sender

*Return control to caller (Return from Eval)

Rend:
   RBase ← RBase[StackP];
   T ← (StackP) - 1, MemBase ← TFrameBr;
   Fetch ← T, Branch[DoReturn];

*DoReturn--perform an actual control return
*T and memory pending = address of context to return to

DoReturn:
   RBase ← RBase[Temp1];
   Temp1 ← T, RBase ← RBase[Ctxt];
   T ← Ctxt, Ctxt ← Md;                      *Ctxt ← My sender
   RBase ← RBase[Name];
   Name ← T, RBase ← RBase[Temp1];           *Name ← Current context
   Store ← Temp1, Md ← AllOnes.c;            *Nil my sender
   T ← StackP;
   MemBase ← TFrameBr;
   MyTemp ← AllOnes.c;
→  Store ← T, Md ← MyTemp;                   *Nil ref to value (RefI)
   RBase ← RBase[AemRegs];
   A ← Ireg, DblBranch[EndEnd, ReturnEnd, R Even];

EndEnd:
   RBase ← RBase[AemRegs];
   Nop;                                      *Placement
   RBase ← RBase[StackP];
   T ← StackP ← (StackP) - (2c);             *Can't do in 1 instr. because of placement conflict
   Call[Stash];                              *Argument in T
   Branch[ReturnEnd];

ReturnEnd:
   RBase ← RBase[Ctxt];
   T ← Ctxt, Call[HashL];                    *Hash current context, dirty...
   Call[Dirty];                              *Only legal after hash call
   MemBase ← Area;
   BrLo ← T                                  *T ← Core address of current activation
```

```
      * Restore Pcb from arec

   MyTemp ← Pcf.c;
   Fetch ← MyTemp, T ← MOop00.c;        *MOop00.c is -(Oop00.c)
   Pcb ← T + (Md);

      * Restore StackP similarly

   MyTemp ← StackPf.c;
   Fetch ← MyTemp;
   StackP ← T + (Md);

      * Restore instance

   MyTemp ← (1c);
   Fetch ← MyTemp;
   RBase ← RBase[MinAt];
   T ← Md;
   Lu ← (T) - (MinAt), RBase ← RBase[Arg1];
   Arg1 ← Md, DblBranch[.NoHash, .DoHash, Carry];

.NoHash:
   T ← T - T - 1, Branch[.GotSelf];

.DoHash:
   Call[Hash];                          *T ← C.A.[Arec[Inst]]
   Call[Dirty];

.GotSelf:
   MemBase ← SelfBr;
   BrLo ← T;
   T ← (2c), Call[GetTpo];              *GetTpo wants T as an argument

   Call[MapCode];                       *Hash Method.  No Dirty
   Nop;                                 *Placement
   T ← (3c), Call[GetTpo];              *Argument is in T
   Call[Dirty];
   MemBase ← TFrameBr;                  *Referencing TFrame
   BrLo ← T;                            *T ← C.A.[Arec[Tframe]]
   RBase ← RBase[Name];
   T ← Name, Call[RefCkLDec];           *Zap me
   RBase ← RBase[StackP];
   T ← (StackP) + 1, Branch[Stor];      *And push top of old stack on new

*Current--Push current context

Current:
   RBase ← RBase[StackP];
   StackP ← (StackP) + 1, RBase ← RBase[Ctxt];
   T ← Ctxt, Branch[Shove];             *Shove takes argument in T

Super:
   T ← (33c), Branch[NovaCall];         *NovaCall 33

SupRet:
   RBase ← RBase[Caddr];
   MemBase ← AemMemBase;                *May not be necessary
   Fetch ← Caddr, Branch[NextNoInt];    *Fetch Instr word
```

```
SendAgain:
   Branch[SndMsg];

*Extended Loads

XiVars:
   Call[NxtByte];
   MemBase ← SelfBr, Branch[Lmem];        *NxtByte returns in T

XtVars:
   Call[NxtByte];
   MemBase ← TFrameBr, Branch[Lmem];

XlVars:
   Call[NxtByte];
   MemBase ← LocFrameBr;
   MyTemp ← T - T, Branch[Lmem];

XilVars:
   Call[NxtByte];
   Branch[IlVars];

XLitMsgs:
   RBase ← RBase[AcmRegs];
   IReg ← (207c), Call[NxtByte];
   MemBase ← LocFrameBr, Branch[FetchMsg];
```

***ShortJumps-- Control transfer operation of one to eight**

```
ShortJumps:
   KnowRBase[AEMRegs];
   T ← (IReg) and (7c);
   T ← (T) + 1, Branch[.JmpCmn];
```

***LongJumps-- Control transfer operations of -1024 to 1023**

```
LongJumps:
   Call[NxtByte];
   RBase ← RBase[AemRegs];
   Arg1 ← T;
   T ← Lsh[Ireg, 10];              *T[5-7]:  Bias Index
   T ← T and (3400c);
                                   *(256*B.I. = True Bias+02000)
   T ← (T) - (2000c);              *Offset for positive and negative jumps
   T ← T + (Arg1);                 *Arg1 = Ac0, an EmReg

.JmpCmn:
   KnowRBase[AEMRegs];
   Lu ← (Ireg) and (10c);          *Arg1 ← T = number of bytes to be skipped
   Arg1 ← T, Branch[.DoJmp, Alu=0];
```

*** Conditional jump//BFP**

```
   RBase ← RBase[Top];
   Lu ← (Top) - (FalseOop.c);      *Pop the stack, then see what top of stack was
   StackP ← (StackP) - 1, Branch[.NoJmp, Alu#0];      *See if Tos is false

   RBase ← RBase[Pcb];
   Pcb ← (Pcb) + (T), Branch[.DoJmp1];      *because of placement constraint

.NoJmp:
   Branch[NextByte];
```

*** Unconditional jump**

```
.DoJmp:
   RBase ← RBase[Pcb];
   Pcb ← (Pcb) + (T);
.DoJmp1:
   RBase ← RBase[AemRegs];
   T ← (Arg1) arsh 1;                      *right shift 1 for # of words skipped
   RBase ← RBase[Pcb];
   A ← Pcb, RBase ← RBase[AemRegs], Branch[.IncAddr, R Odd];

   A ← Arg1, Branch[.OddInc, R Odd];       *test bytes skipped is odd or even

   Branch[.IncAddr];                       *placement constraint

.OddInc:
   T ← T + 1;                              *Left byte and odd skip, inc Caddr

.IncAddr:
   RBase ← RBase[Caddr];
   Caddr ← (Caddr) + (T), Branch[NextByte];  *Caddr ← core address of next byte, go again
```

***The sixteen trapped arithmetic messages**

```
ATrapMsgs:
   RBase ← RBase[StackP];
   T ← StackP, RBase ← RBase[AemRegs];
   Lu ← (Ireg) and (10c);
   SavSp ← T, Branch[.Aimp, Alu=0];        *Save Stack pointer
   Branch[Apply];      *extra instr because of placement constraints
```

******Temp2 top of stack(i.e. Receiver), Temp3 is second--both arguments**

```
.Aimp:
   RBase ← RBase[Top];
   T ← Top, Call[Ivall];
   RBase ← RBase[Temp2];
   Temp2 ← T, Call[PopTop];
   Call[Ivall];
   RBase ← RBase[Temp3];
   Temp3 ← T;
   T ← (Temp2) xor T;
   Temp4 ← T, RBase ← RBase[AemRegs];
   T ← (IReg) and (17c);
   RBase ← RBase[Temp3];
   B ← T, BigBDispatch;
   T ← Temp3, Branch[ArithDispatch];

ArithDispatch:
   Branch[Addarith],     At[Dispatch5, 0];
   Branch[Subarith],     At[Dispatch5, 1];
   Branch[Les],          At[Dispatch5, 2];
   Branch[Gtr],          At[Dispatch5, 3];
   Branch[Leq],          At[Dispatch5, 4];
   Branch[Geq],          At[Dispatch5, 5];
   Branch[Eql],          At[Dispatch5, 6];
   Branch[Neq],          At[Dispatch5, 7];

*Temp4 = Temp2 xor T  (Temp2 xor Temp3)

Addarith:
   Lu ← Temp4, Branch[.CkAddOf, R > =0];      *branch if operand signs same
   T ← (Temp2) + T, Branch[ArithOut];

.CkAddOf:
   T ← (Temp2) + T;
.CkOf:
   Lu ← (Temp2) xor (T);
   Branch[.YesOf, Alu < 0];                   *branch if overflow

ArithOut:
   Nop;          *Placement
   Call[Intn];
   RBase ← RBase[Top];
   Top ← T, Branch[RefX4];

.YesOf:
   Branch[Apply];

Subarith:
```

```
    KnowRBase[StackAndTemps];
    Lu ← Temp4, Branch[.CkSubOf, R < 0];          *branch if operand signs different
    T ← (Temp2) - T, Branch[ArithOut];

.CkSubOf:
    T ← (Temp2) - T, Branch[.CkOf];

Les:
    T ← (Temp2) - T;
.Lrel:
    A ← Temp4, Lu ← T, Branch[.LDiff, R < 0];
    DblBranch[TTrue, FFalse, Alu < 0];

Gtr:
    T ← T - (Temp2);
.Grel:
    A ← Temp4, Lu ← T, Branch[.GDiff, R < 0];
    DblBranch[TTrue, FFalse, Alu < 0];

Eql:
    Lu ← Temp4;
    DblBranch[TTrue, FFalse, Alu=0];

Neq:
    Lu ← Temp4;
    DblBranch[TFalse, FTrue, Alu=0];

Leq:
    T ← (Temp2) - T - 1, Branch[.Lrel];

Geq:
    T ← T - (Temp2) - 1, Branch[.Grel];

.LDiff:
    A ← Temp2, DblBranch[TTrue, FFalse, R < 0];

.GDiff:
    A ← Temp3, DblBranch[TTrue, FFalse, R < 0];
```

*** Two sets of true/false are needed because of branching constraints--checking signs in compares**

```
TTrue:
    T ← Top, Branch[Shove];              *Put receiver on top of stack

FFalse:
    T ← FalseOop.c, Branch[Shove];       *Put False on top of stack

FTrue:
    T ← Top, Branch[Shove];              *Put receiver on top of stack

TFalse:
    T ← FalseOop.c, Branch[Shove];       *Put False on top of stack
```

*****Other messages trapped by the microcode interpreter

```
OTrapMsgs:
   RBase ← RBase[StackP];
   Q ← StackP;
   RBase ← RBase[AcmRegs];
   T ← (Ireg) and (17c);
   B ← T, BigBDispatch;
   SavSp ← Q, Branch[Omsgs];

Omsgs:
   Branch[Subscript],          At[Dispatch6, 0];     *subscript
   Branch[SubscriptGets],      At[Dispatch6, 1];     *subscript←
   Branch[Next],               At[Dispatch6, 2];     *next
   Branch[NextGets],           At[Dispatch6, 3];     *next←
   Branch[Len],                At[Dispatch6, 4];     *length
   Branch[Eq],                 At[Dispatch6, 5];     *identity
   Branch[Apply],              At[Dispatch6, 6];
   Branch[Apply],              At[Dispatch6, 7];
   Branch[Apply],              At[Dispatch6, 10];    *class
   Branch[Apply],              At[Dispatch6, 11];    *and:
   Branch[Apply],              At[Dispatch6, 12];    *or:
   Branch[Apply],              At[Dispatch6, 13];    *new
   Branch[Apply],              At[Dispatch6, 14];    *new:
   Branch[Apply],              At[Dispatch6, 15];    *to:
   Branch[Apply],              At[Dispatch6, 16];
   Branch[Apply],              At[Dispatch6, 17];    *asStream

SubscriptGets:              *Right bracket (subscripted store)
   Call[PopTop];          *pop top off
   RBase ← RBase[SavR1];
   SavR1 ← T;     *Fetch new val for Dot←

Subscript:                  *Left bracket (subscripted load)
   Call[GtopCls];          *Get class of top of stack
   RBase ← RBase[Name];
   Name ← T, Call[TestVecStr];        *Check to make sure arg is vec or str
   RBase ← RBase[Name];
   T ← Name, Call[Ilong];
   RBase ← RBase[Temp4];
   Temp4 ← T, RBase ← RBase[AOop];
   AOop ← AllOnes.c, Call[PopTop];
   Call[Ivall];
   Branch[IvalX3];

NextGets:                                   *store next into stream
   Call[PopTop];                            *Fetch new val for Next←
   RBase ← RBase[SavR1];
   SavR1 ← T;

Next:
   Call[GtopCls];
   Lu ← T - (StmCls.c);
   Branch[.NotStm, Alu#0];                  *Not a stream
   Call[Hash];                              *T ← Core Add of stream
   MemBase ← AcmMemBase, MyTemp ← (T) + 1;
   MyTemp ← (Fetch ← MyTemp) - 1;           *Save core address
   MemBase ← ACoreBr;                       *Referencing ACore
```

```
   BrLo ← MyTemp;
   DontKnowR;
   Temp3 ← Md;                          *Temp3 ← stream index
   RBase ← RBase[Top];
   Q ← B ← Top;
   AOop ← Q;                            *All this so that we can do AOop ← Top  (Uses FF)
   MyTemp ← T - T;
   MyTemp ← (Fetch ← MyTemp) + (2c);    *Fetch stream contents
   Top ← Md, Fetch ← MyTemp;            *Fetch stream length
   T ← Md, Call[Ivall];
   DontKnowR;
   Temp4 ← T;        *(Uses FF)
   RBase ← RBase[Top];
   T ← Top, Call[GclassL];
   DontKnowR;
   Name ← T;
   RBase ← RBase[Temp3];
   T ← Temp3, Call[Ivall];
   T ← (T) + 1;

IvalX3:
   RBase ← RBase[Temp4];
   Lu ← (Temp4) - (T);                  *Alu ← val of length-val of index
   Temp2 ← T - 1, Branch[.IToBg, Alu < 0];  *check index > = length
   T ← Top, Branch[.INeg, Alu < 0];     *Check index < = 0
   BOop ← T;
   Arg1 ← T, Call[Hash];
   Call[Dirty];                         *dirty strector being indexed
   MemBase ← BCoreBr;                   *referencing BCore
   BrLo ← T;
   Nop;                                 *Placement
   RBase ← RBase[Name];
   T ← Name, Call[TestVecStrInt];       *returns type code in T and Alu
   Lu ← T - 1, Branch[Vec, Alu=0];
   DblBranch[Str, Int, Alu=0];

.NotStm:
   Branch[Apply];

.IToBg:
   Branch[Apply];

.INeg:
   Branch[Apply];

Int:
   RBase ← RBase[AemRegs];
   A ← Ireg, Branch[.IntNextGets, R Odd];   *check for Next or Next←
   RBase ← RBase[Temp3];
   Temp3 ← Temp2;                       *Temp3 ← Index of interval      (Uses FF)
   T ← (2c), Call[Gti];
   Temp4 ← T;                           *Temp4 ← value of Int step      (Uses FF)
   T ← T - T, Call[Gti];                *T ← interval start (= 0)

MullP:
   RBase ← RBase[Temp4];
   Temp4 ← (Temp4) rsh 1, Branch[.DoAdd, R Odd];    *test low bit of multiplier
.Dbl:
```

```
   Temp3 ← (Temp3) + (Temp3), DblBranch[IntNStor, MullP, Alu=0];

.DoAdd:
   T ← (Temp3) + T;                          *add to sum
   Lu ← Temp4, Branch[.Dbl];                 * for zero test

IntNStor:
   Call[Intn];                               *T ← Intrned sum (T)
   DontKnowR;                                *Force use of FF in next instr
   Top ← T, Branch[RefX12];

.IntNextGets:
   Branch[Apply];

Str:
   RBase ← RBase[Temp3];
   T ← Temp2;
   Temp3 ← T ← (T) rsh 1;                    *T ← word index in string
   MemBase ← BCoreBr;
   RBase ← RBase[AemRegs];
   Fetch ← T, B ← Ireg, DblBranch[.StLd, .StSt, R Even];          *Load or store?

.StLd:
   RBase ← RBase[Temp2];
   A ← Temp2, T ← Md, DblBranch[.StLdEv, .StLdOd, R Even];        *T ← word with byte

.StLdEv:
   Nop;              * wait for T to load
   T ← Rsh[T, 10];   *Left byte

.StLdOd:
   Nop;        *Placement
   T ← T and (377c), Branch[IntnStor];       *Right byte

.StSt:
   T ← Md, RBase ← RBase[SavR1];             *Can't abandon Md
   T ← SavR1;
   Top ← T, Call[Ivall];                     *Save new byte on Top, T ← value of new byte (Uses FF)
   RBase ← RBase[State];
   MyTemp ← T and (377c);                    *MyTemp ← masked value of new byte
   RBase ← RBase[Temp3];
   T ← Temp3, MemBase ← BCoreBr;
   Fetch ← T, B ← Temp2, DblBranch[.StStEv, .StStOd, R Even];

.StStEv:
   RBase ← RBase[State];
   MyTemp ← DPF[MyTemp, 10, 10, Md], Branch[.StStCm];   *Deposit in left byte

.StStOd:
   RBase ← RBase[State];
   MyTemp ← DPF[MyTemp, 10, 0, Md];                     *Deposit in right byte

.StStCm:
   Store ← T, Md ← MyTemp, Branch[RiSubEnd];

Vec:
   MemBase ← BCoreBr;
   RBase ← RBase[AemRegs];                   * Address element of vector
```

```
   A ← Ireg, DblBranch[.VcLd, .VcSt, R Even];          *Test load or store?

.VcLd:
   RBase ← RBase[Temp2];
   Fetch ← Temp2;
   Top ← Md, Branch[RiSubEnd];

.VcSt:
   RBase ← RBase[SavR1];
   T ← SavR1;          *New oop
   RBase ← RBase[Top];
   Top ← T;            *FF free
   RBase ← RBase[Temp2];
   T ← Temp2, Branch[Pull];                            *Top ← Oop in vec

RiSubEnd:
   RBase ← RBase[Top];
   T ← Top, Call[RefCkLInc];

RefX12:
   MemBase ← ACoreBr;
   T ← (1c);
   RBase ← RBase[AOop];
   Lu ← (AOop) + 1;
   Fetch ← T, DblBranch[.ImDone, .PostStm, Alu=0];     *Check for stream op

.PostStm:
   DontKnowR;
   Arg1 ← Md;
   Name ← Md, Call[Ival];
   T ← (T) + 1, Call[Intn];
   MemBase ← ACoreBr;
   MyTemp ← (1c);
   Store ← MyTemp, Md ← T;             *Address stream index
   RBase ← RBase[Name];
   T ← Name, Call[RefCkLDec];
   Branch[RefX4];

.ImDone:
   T ← Md, Branch[RefX4];              *Can't abandon Md (not used)

Apply:
   RBase ← RBase[SavSp];
   T ← SavSp;
   StackP ← T;         *Uses FF
   MemBase ← TFrameBr;
   Fetch ← T;
   RBase ← RBase[AcmRegs];             *Recover top of stack
   T ← (Ireg) and (377c);
   T ← T - (244c);
   MemBase ← AcmMemBase;               *absolute addr
   RBase ← RBase[Top];
   Top ← Md, Branch[FetchMsg]; * -244 = diff between "+" byte code and addr of "+" atom

Len:
   Call[GTopCls];
   Call[TestVecStr];              *Arg1 loaded by GTopCls, not smashed by TestVecStr
   RBase ← RBase[AcmRegs];
```

```
    T ← Arg1, Call[Ilong];
    Branch[ArithOut];

Eq:
    Call[PopTop];
    RBase ← RBase[Top];
    Lu ← (Top) - (T);
    T ← TrueOopm1.c, Branch[.IsEq, Alu=0];

    T ← FalseOop.c, Branch[Shove];

.IsEq:
    T ← T + 1, Branch[Shove];
```

```
************************************************************

*Litmsgs----Non-Trapped messages

Litmsgs:
   MemBase ← LocFrameBr;

FetchMsg:                    *enter here with T+MemBase = addr of message, and correct BR
   Fetch ← T;
   Name ← Md;

SndMsg:
   Call[GTopCls];            *T ← class of top of stack
   DontKnowR;                *Don't make any assumptions about RBase
   SupMod ← T, Call[HashL];  *Hash class of target--no dirty
   RBase ← RBase[State];
   MyTemp ← T + (MDictF.c);
   MemBase ← AcmMemBase;     *Relative to zero
   Fetch ← MyTemp;
   Lu ← Md + 1;
   BOop ← Md, DblBranch[.Nid1, .DictOk, Alu=0];      *Null MDict fails

.DictOk:
   T ← Md;
   Call[HashL];                                      *Hash MDict of target, no dirty (arg. in T)
   MemBase ← AcmMemBase, MyTemp ← T;
   Fetch ← MyTemp;
   MemBase ← BCoreBr;
   BrLo ← T, T ← Md;                                 *Referencing BCore
   Call[Ilong];
   RBase ← RBase[Temp2];
   Temp2 ← T, Call[Hash];                            *Temp2=Length//Hash left side of dictionary
   RBase ← RBase[Names];
   Names ← T, RBase ← RBase[Temp2];
   T ← (Temp2) - 1, RBase ← RBase[Name];
   T ← (Name) and (T), Branch[.Nid2, Alu=0];         *Zero MDict size fails; initial hash=name and len-1
   RBase ← RBase[Temp1];
   Temp1 ← T, MemBase ← AcmMemBase;
   Cnt ← -1s;                                        *Count wraparounds

.DluLp:
   T ← Temp1, RBase ← RBase[Names];
   T ← (T) + (Names);
   T ← (Fetch ← T) - (Names);                        *Restore T to Temp1
   RBase ← RBase[Temp2];
   Lu ← (Temp2) - (T) - 1;
   DblBranch[.Wrap, .IncIx, Alu=0];                  *Look at Names[index]

.IncIx:
   Temp1 ← (Temp1) + 1, Branch[.Dlul1];              *Index ← Index + 1

.Wrap:
   Temp1 ← T - T, Branch[Wrap2, Cnt=0&+1];           *First wrap leaves 0 in Cnt

.DluL1:
   RBase ← RBase[Name];
   Lu ← (Md) + 1;                                    *Empty slot?
   T ← (Name) - (Md), RBase ← RBase[Temp2], DblBranch[.Nid3, .NotNil, Alu=0];
```

```
.NotNil:
   T ← (Temp1) - 1, DblBranch[.GotIt, .DluLp, Alu=0];      *Test for hit, otherwise loop

.GotIt:
   DblBranch[.UnWrap, .IxOk, Alu < 0];                     *Un-inc hashi

.UnWrap:
   KnowRBase[StackAndTemps];
   T ← (Temp2) - 1;

.IxOk:
   Temp2 ← T;
   MemBase ← BCoreBr;
   T ← (1c);
   Fetch ← T;
   RBase ← RBase[Arg1];
   Arg1 ← Md, Call[Hash];                                  *found name---get value
   RBase ← RBase[Temp2];
   T ← (Temp2) + (T), MemBase ← AemMemBase;                *real core address (more or less)
   Fetch ← T;

RProg:
   Nop;
   RBase ← RBase[BOop];                                    *Frees up FF in next instr
   BOop ← Md;
   Arg1 ← Md, Call[Hash];                                  *BOop ← MDict[Literal Operator]
   MemBase ← AemMemBase, MyTemp ← T;
   Fetch ← MyTemp;
   MemBase ← BCoreBr;
   BrLo ← T;
   T ← T - T, RBase ← RBase[Temp3];                        *since under BCoreBr
   Temp3 ← Md, T ← T + 1;                                  *increment to fetch 2nd method word
   Lu ← Temp3;
   Lu ← (Temp3) - 1, Branch[Bytes, Alu=0];                 *No primitive
   Temp1 ← T - T, Branch[Byterp, Alu=0];                   *Primitive 1=NoOp=Branch[Byterp]

      * Do a primitive.  Start by transferring the args into the fixed communication area

   Fetch ← T;                                              *T was incremented at RProg+3
   RBase ← RBase[StackP];
   T ← (StackP) - 1;
   MemBase ← TFrameBr;
   RBase ← RBase[Temp2];
   Temp2 ← T, T ← Md;
   T ← T and (7c);
   Temp1 ← T;
   RBase ← RBase[Top];
   Q ← Top;
   T ← T - T - 1;                                          *Nil T for store of Self
   RBase ← RBase[Temp1];

.NcLoop:
   T ← T + (SelfLoc.c) + 1;
   MemBase ← AemMemBase;
   Store ← T, Md ← Q;                                      *First time, store Self--then args
   T ← (Temp1) - 1;
   Temp1 ← T, MemBase ← TFrameBr, Branch[.NcDone, Alu < 0];
```

```
    Temp2 ← (Fetch ← Temp2) - 1;
    Q ← Md, Branch[.NcLoop];

.NcDone:
    MemBase ← AemMemBase;
    MyTemp ← (PrimTabLoc.c);
    Fetch ← MyTemp;
    T ← (Temp3) + (Md);                    *Add primitive number to primitive table
addr
    Fetch ← T;
    T ← Md, Branch[FetchTPc];

PrimRet:
    KnowRBase[AEMRegs];
    T ← Arg1;
    Top ← T;
    MemBase ← BCoreBr;
    MyTemp ← (1c);
    Fetch ← MyTemp;
    T ← (7c);
    T ← T and (Md);
    RBase ← RBase[StackP];
    T ← (StackP) - T, Branch[Stor];

PrimFail:
    RBase ← RBase[BOop];
    T ← BOop, Call[RefCkLInc];
    Branch[.DoBytes];
```

***Do a Smalltalk-coded method**

```
Bytes:
    Call[RefLastInc];                      *Bump refct of method

.DoBytes:
    Call[ACtxt];
    RBase ← RBase[AOop];
    AOop ← T, RBase ← RBase[Temp1];        *T has oop of new arec
    MemBase ← ACoreBr;                     *Temp1 has core address
    BrLo ← Temp1;
    T ← T - T, RBase ← RBase[Ctxt];
    T ← (Store ← T) + 1, Md ← Ctxt;        *New AR[Sender] ← retiring activation
    RBase ← RBase[Top];
    T ← (Store ← T) + 1, Md ← Top;         *New AR[Inst] ← top of old stack

    MyTemp ← T - T - 1;                    *-1
    T ← (Store ← T) + 1, Md ← MyTemp;      *New AR[Class] ← Nil
    RBase ← RBase[BOop];
    Store ← T, Md ← BOop;                  *New AR[Code]← MDict[Literal Op]
    MemBase ← BCoreBr;
    T ← (1c);
    Fetch ← T;                             *Address second word of code
    T ← Md;
    RBase ← RBase[AemRegs];                *wait for T to load
    Arg1 ← Rsh[T, 10];                     *Arg1 ← TSize (Byte #2 of code)
    RBase ← RBase[Name];
    Name ← T ← T and (377c);               *Name ← NArgs (Byte #3 of code)
    RBase ← RBase[StackP];
```

```
   StackP ← (StackP) - (T) - 1, RBase ← RBase[AemRegs];  *Pop the object and its args
   T ← Arg1;
   Temp4 ← T, Call[AVec];                                *Allocate vector, Temp4 ← TSize (Uses FF)
   MemBase ← ACoreBr;
   T ← (4c);
   RBase ← RBase[AemRegs];
   Store ← T, Md ← Arg1;                                 *New AR[Tempframe] ← that vector

*Loop to store args and nil stack and temp part of tframe
*RetN2 is the offset in Tframe which points at the last slot filled,
*starts out as size of tframe

   RBase ← RBase[Name];
   T ← Name, RBase ← RBase[Temp4];
   T ← T - (Temp4);                                      *T ← - # of temps to nil
   Cnt ← T;
   MyTemp ← AllOnes.c;                                   *prepare nil--MyTemp is in
RMRegion[Temps]
   T ← (Temp1) - 1, MemBase ← AemMemBase;                *Since relative to zero
   T ← (Temp4) + T, Branch[.NoTemps, Cnt=0&+1];          *prepare pointer to slot in new tempframe

.PutLp:
   T ← (Store ← T) - 1, Md ← MyTemp, Branch[.PutLp, Cnt#0&+1];   *Address current element

.NoTemps:
   RBase ← RBase[StackP];
   T ← (StackP) + 1, TaskingOff;                         *T ← Old StackP
   MemBase ← TFrameBr;
   B ← Md;                    *Necessary after a store and before a DummyRef so won't hold
   DummyRef ← T;
   T ← Pipe1;
   Nop;
   BrLo ← T;
   TaskingOn;
   RBase ← RBase[Name];
   T ← Name, RBase ← RBase[Temp1];                       *T has offset Nargs
   Temp1 ← (Temp1) + T;                                  *Temp1 has absolute addr. of new stack
   Branch[.NilTop];                                      *Add. of lowest arg in old stack

.StorIt:
   Fetch ← T;                                            *fetch next arg from old stack
   Q ← Md;
   MemBase ← AemMemBase;
   Store ← Temp1, Md ← Q;                                *store arg on new stack, absolute addressing

.NilTop:
   MemBase ← TFrameBr;
   Store ← T, Md ← MyTemp;                               *Nil arg on old stack (for refct)
   T ← (T) - 1;          *Decrement
   Temp1 ← (Temp1) - 1, DblBranch[.DoneIt, .StorIt, Alu < 0];  *Decrement absolute addr.

.DoneIt:
   Nop;                                                  *In the next instruction, Temp1 is restored
   Temp1 ← (Temp1) + 1, Call[Stash];                     *Puts pc and stackp in new arec
   MemBase ← BCoreBr;
   T ← (2c);
   Fetch ← T;                                            *Addressing third word
   RBase ← RBase[AOop];
```

```
    Q ← Md;
    T ← AOop;                          *Wait for memory
    Ctxt ← T;
    MemBase ← ACoreBr;
    MyTemp ← T - T, TaskingOff;
    DummyRef ← MyTemp;
    T ← Pipe1;
    TaskingOn;
    MemBase ← ArecBr;
    BrLo ← T;
    T ← Q;                             *T ← old Md
    RBase ← RBase[Temp1];              *Store into Arec
    Q ← Temp1;
    MyTemp ← Q;
    MemBase ← TFrameBr;
    BrLo ← MyTemp;                     *Storing into TFrame
    RBase ← RBase[StackP];
    StackP ← Rsh[T, 10];               *offset of 1st stack loc (byte #4)
    StackP ← (StackP) - 1, RBase ← RBase[Pcb];
    Pcb ← T and (377c);                *initial pcb (byte #5)
    MemBase ← BCoreBr;
    MyTemp ← T - T, TaskingOff;
    DummyRef ← MyTemp;
    T ← Pipe1;
    TaskingOn, Call[MapCode];          *T ← Core address of code, from BCore
    RBase ← RBase[MinAt];
    T ← MinAt, RBase ← RBase[Top];
    Lu ← (Top) - (T);
    T ← Top, DblBranch[.NilSelf, .HashSelf, Carry];

.NilSelf:
    T ← AllOnes.c, Branch[.LoadMe];    *Get ready to nil self

.HashSelf:
    Arg1 ← T, Call[Hash];              *T ← core address of new self
    Call[Dirty];

.LoadMe:
    MemBase ← SelfBr;
    BrLo ← T, Branch[Byterp];          *Heigh Ho Silver, Away!
                                       *"Who was that masked man?"


Wrap2:
    T ← Md, Branch[.Nid1];             *Can't abandon Md (Not used)
.Nid3:
    Branch[.Nid1];
.Nid2:
    Branch[.Nid1];
.Nid1:                                 *Comes here from failure:
    RBase ← RBase[SupMod];             *Zero, Null MDict, NotInd
    T ← SupMod, Call[HashL];           *Hash current MClass, no dirty
    RBase ← RBase[SupMod];
    MemBase ← AcmMemBase;
    MyTemp ← T + (SprClsF.c);
    Fetch ← MyTemp;
    SupMod ← Md;
    T ← (Md) + 1;
```

```
   Dbl Branch[NoMsg, Dispatch, Alu=0];

NoMsg:
   T ← ErrPrg.c;                          *Remember, Emulator BR in control
   Fetch ← T, Branch[RProg];

LitMsgs2:
   T ← (T) + (20c), Branch[LitMsgs];      *Address literal operator

LitMsgs3:
   T ← (T) + (40c), Branch[LitMsgs];      *Address literal operator
```

```
*Recursive Freer----Enter with refct=1
*some of the calls may change Link, or the saved return SavR1--
*Be sure to check that it does not smash important returns, and Calls and
*Branches are used in the right places

Subroutine;

Recuf:
   MemBase ← AemMemBase;
   RBase ← RBase[Father];
   T ← (Father) + 1, RBase ← RBase[RefCtLink];          *Father=-1 Iff top-level entry
   T ← RefCtLink, DblBranch[SavRs, Gpcl, Alu=0];        *Gpcl->no save for recursive
   Top Level;

SavRs:
   RBase ← RBase[SavR1];
   SavR1 ← T;                                           *Save Link for top-level entry

Gpcl:
   RBase ← RBase[AemRegs];
   T ← Arg1, RBase ← RBase[Temp3];
   Temp3 ← T;
   Call[PClassMap];                                     *Smashes Link
   MemBase ← AemMemBase;
   T ← Md;
   Lu ← T and (CptMsk.c);                               *extract cpt
   Lu ← T and (IscMsk.c), DblBranch[QFinst, IsPt, Alu=0];  *extract isc

IsPt:
   DblBranch[QFinst0, DoFld0, Alu=0];                   *Branch if size zero

DoFld0:
   Nop;                                                 *Placement
   Call[Hash];
   Call[Dirty];
   RBase ← RBase[Temp3];
   MemBase ← AemMemBase;
   Temp3 ← (Fetch ← T);                                 *Fetch from core address of object
   RBase ← RBase[AemRegs];                              * ref'd by oop
   MyTemp ← Md, RBase ← RBase[Father];                  *Get the data, and save it for swap
   Store ← T, Md ← Father;
   RBase ← RBase[AemRegs];
   T ← Arg1;
   Arg1 ← MyTemp;
   RBase ← RBase[Father];
   Father ← T, RBase ← RBase[Rot0];                     *Father ← oop
   T ← (Rot0) + (Rct1Bit.c);
   MemBase ← RotBaseBr;                                 *RotBase needed for RotA
   Store ← RotA, Md ← T;                                *Bump refct to 2, flag first time

   Call[RefCkDec];                                      *Decrement refct

ReRet:
   RBase ← RBase[Father];
   T ← Father;

.Here:
```

```
    Nop;                                          *Placement
    Arg1 ← T, Call[Hash];                         *Hash object being freed--dirty
    Call[Dirty];
    T ← (T) + 1, RBase ← RBase[Temp3];            *Core address of field 1
    MemBase ← AemMemBase;
    Temp3 ← (Fetch ← T);
    Temp2 ← Md, RBase ← RBase[Rot0];
    Lu ← (Rot0) and (Rct1Bit.c);                  *Test even rct (means first time)
    Branch[FTime, Alu=0];                         *field 1 holds field offset

DoNxt:
    RBase ← RBase[Temp2];
    T ← Temp2 ← (Temp2) - 1;                      *Look at the next field
.DoNxt1:
    T ← (Temp3) + (T), Branch[Out, Alu=0];        *test if done
    Fetch ← T;
    RBase ← RBase[MinAt];
    T ← Md;
    Lu ← (T) - (MinAt);                           *Atom?

Gtf1:
    DontKnowR;
    Arg1 ← T, Branch[.Gtf2, NoCarry];             *Branch if not atom (Uses FF)
    RBase ← RBase[Temp2];
    T ← Temp2 ← (Temp2) - 1, Branch[.DoNxt1];     *Look at the next field

.Gtf2:
    RBase ← RBase[Temp2];
    T ← Temp2;
    Store ← Temp3, Md ← T;
    Call[RefIDec];                                *Decrement ref count
    MemBase ← AemMemBase, Branch[ReRet];

FTime:
    KnowRBase[Hash];
    T ← (Rot0) - (Rct1Bit.c);                     *Done field zero
    MemBase ← RotBaseBr;
    Store ← RotA, Md ← T;                         *Unbump refct to 1, not first time
    RBase ← RBase[Arg1];
    T ← Arg1, Call[Ilong];                        *obj. length//Arg1←father at Here:
    MemBase ← AemMemBase;
    RBase ← RBase[Temp2];
    MyTemp ← (T) - 1;
    T ← Temp2;
    Temp2 ← MyTemp;                               *Routed through Alu//must be in same
region
    DblBranch[.Out0, NotOne, Alu=0];              *Test if only had 1 field

NotOne:
    RBase ← RBase[MinAt];
    Lu ← (T) - (MinAt), Branch[Gtf1];             *Jump into inner loop

Out:
    RBase ← RBase[Temp3];
    T ← (Temp3) - 1, Branch[.Out1];
.Out0:
    KnowRBase[StackAndTemps];
    T ← (Temp3) - 1, Branch[.Out1];
```

```
.Out1:
   RBase ← RBase[Father];
   T ← Father, Fetch ← T;                          *Load father stashed in field 0
   Temp3 ← T, T ← Md;

O2A:
   Father ← T, Branch[QFinst];                     *Free the instance
```

***QFinst: Quick Finst for exact size objects----**

```
QFinst0:                                           *duplicate--placement constraint
   RBase ← RBase[Temp3];
   T ← Temp3, Call[PClassMap];
   MemBase ← AemMemBase;
   Temp2 ← Md, Branch[.QF1];
QFinst:
   RBase ← RBase[Temp3];
   T ← Temp3, Call[PClassMap];
   MemBase ← AemMemBase;
   Temp2 ← Md, Branch[.QF1];
.QF1:
   T ← (Temp2) and (RciMsk.c);
   RBase ← RBase[AemRegs];
   Arg1 ← Rsh[T, 7];                               *Mytemp and Arg1 in same region
   T ← VarClsm400.c;
   T ← T + (400c);
   Lu ← (Arg1) - T, RBase ← RBase[Temp2];          *Compare to varlen class
   T ← (Temp2) and (IscMsk.c), DblBranch[.VarOop, .NotVar, Carry];

.VarOop:
   Lu ← T - (11c);
   Temp2 ← T + (ClFree.c), DblBranch[OctL, Exact, Alu > =0];

.NotVar:
   RBase ← RBase[Temp2];
   Lu ← T - (Octv.c);                              *Extract inst size
   Temp2 ← ClFree.c, DblBranch[OctL, Exact, Alu > =0];

Exact:
   Call[Hash];                                     *Hash class of object being freed--dirty
   Call[Dirty];
   RBase ← RBase[Temp2];
   MemBase ← AemMemBase;
   T ← (T) + (Temp2);                              *Might cause a purge
   Temp2 ← (Fetch ← T);
   T ← Temp3;
   Arg1 ← T;
   MyTemp ← Md;
   T ← Temp2;
   Store ← T, Md ← Temp3;                          *Class[Free] ← RefOop
   T ← MyTemp;
   Temp2 ← T, Call[Hash];                          *Temp2 ← Class[FreeListHead]
   Call[Dirty];                                    *//hash object being freed
   RBase ← RBase[Temp2];
   MemBase ← AemMemBase;
   Store ← T, Md ← Temp2, Branch[FiRet];           *RefOop[0]←TempX1
```

***Have to call real Finst in Nova otherwise**

```
OctL:
    T ← Temp3;                          *Call Finst with RefOop in Ac0
    Arg1 ← T;
    T ← (35c), Branch[NovaCall];        *35 traps to 76400

FiRet:
    MemBase ← AemMemBase;
    RBase ← RBase[Father];
    Lu ← (Father) + 1;                  *Finst returns here via trap
    T ← Father, DblBranch[RTop, .Here, Alu=0];  *Father not nil=>more Recuf

RTop:
    Subroutine;
    RBase ← RBase[SavR1];
    Link ← SavR1;                       *Father is nil!  Restore Link, and then...
    Return;                             *Return to original caller!

    Top Level;
```