```
                ;Global and local register allocation and use
                ;Registers marked "*" are defined in USYS

000020           g0.temp==g0                    ;  scratch
000021           g1.temp==g1                    ;  scratch
000022           g2.temp==g2                    ;  scratch
000023           g3.rone==g3                    ;  "1"
000024           g4.rsign==g4                   ;  "100000"
000025           g5.temp==g5                    ;  scratch
                ;g6.temp                        ;* Scratch
000027           g7.temp==g7                    ;  scratch
                ;g10.state                      ;* Machine state
000031           g11.temp==g11                  ;  scratch
000032           g12.temp==g12                  ;  scratch
                ;g13.mbr                        ;* Saved MBR for main
                ;g14.misc                       ;* Copy of MISC
000035           g15.temp==g15                  ;  scratch

                ;Register Block Allocations for the H316:

                ;RB.MAIN, Main Instruction Emulation

000200          rb.main=200
000000           L0.pc==L0                      ; Program counter
000003           L3.a==L3                       ; Accumulator
000004           L4.b==L4                       ; B-register
000005           L5.x==L5                       ; Index register
000006           L6.curr==L6                    ; Current running process pointer
000007           L7.cbit==L7                    ; CBIT register
000010           L10.flags==L10                 ; sign=inhibited, lsb=measurement mod
000011           L11.sp==L11                    ; User stack pointer

                ;RB.MSHIFT, Macroshift Emulation (also some clock stuff)

000220          rb.mshift == 220
000000           L0.shfcbit==L0                 ; For macro CARRY bit
                ; L1.ret1==L1   ; Local return, level 1
000002           L2.shftmp==L2                  ; Save SIGN bit within arithmetic shi
                                                ; (may get interrupted)
000003           L3.shfmar==L3                  ; Save MAR across interrupts
000004           L4.sc==L4                      ; Shift count

                ;RB.SOFTINT, Software microinterrupt registers

000240          rb.softint == 240
000000           10.base == 10                  ;save old base for INT.SCAN, etc.
000003           13.wmask ==L3                  ;mask for matches
000004           L4.wmar==L4                    ; non-zero = watch this mar
000005           15.inspc == 15                 ;instruction left-off
000006           L6.wmbr==L6                    ; if L4<>0, this is old value
000007           17.pmscan == 17                ;process manager scan needed before n
                                                ;-1 = just scan, +1 = goad, +2 = timi
000243           regs rb.softint+3              ;over to L3.WMASK
000243           exp 177777                     ;assume all bits
```

```
                ;Random System Symbols and Definitions

                ;Dispatch definitions:

000002           firstd==2                      ; First dispatch for every instructio
000003           op==3                          ; dispatch on op code for mem ref ins

                ;Carry-bit definitions:

000020                  cbitset == 20           ;means "Carry Set" for "L6.CBIT"
000000                  cbitreset == 0          ;means "Carry not Set" for "L6.CBIT"
000020                  cbitcoll==20            ; Bit where relevant saved ALU status
000020                  alu.cbit==20            ; ALU status "carry" bit

                ;Various MASKs

000377                  msk377==377
037777                  msk37777==37777
077777                  msk77777==77777
177777                  msk177777==177777
```

```
        ;Patches to "USYS.MIC" system code

020157          uram startdum           ;overlay dummy start routine
020157          start->upc
```

```
        ;Interrupt entrance vectors - all in uram

        ;uram location zero

020000          uram 20000              ;reserved to trap dispatch faults

        uram0::
020000   crash->upc
020001   4->temp

        ;Microprogrammable interrupt service (and Macro-TRAP)

020040          uram 20040

        softint::
020040   soft -> upc                    ; service routine is at soft
020041   rb.softint -> base             ;set up base for us

        ;Macroprogram and Microprogram traps
        ;
        ;       TRAP (in usys-prom) is called whenever the microprogram
        ;detects an error or exceptional condition in the macroprogram.
        ;A code describing the condition is in TEMP when TRAP is called.
        ;The micro-system error registers are setup and USYS is notified
        ;of the trap.
```

```
        ;Main (no microinterrupt => emulate awhile . . .)

        ;       The microcode comes here to emulate the next macroinstruction.
        ;From here, the code dispatches based on the macroinstruction.
        ;Memory reference instructions all go to a common routine to
        ;calculate the effective address; subsequently, in a second
        ;dispatch, they will go to individual routines for the specific
        ;memory operation involved. Other macroinstructions
        ;generally dispatch immediately to a specialized routine for that
        ;particular instruction.
        ;
        ;       This "dispatching", of course, is made possible by the
        ;special MIR-asscciated hardware and the associated dispatch memory.
        ;An illegal macroinstruction will select a dispatch address of
        ;uram 0 (start of microcode ram, currently 20000) from dispatch
        ;memory.  This jump is trapped with code 4.  (Other problems might
        ;also cause a jump to uram 0.  for example, we have seen
        ;INTS supply a value which led to such a jump; in that case, the
        ;value of INTS, saved after the trap, revealed the cause.)

        ;Assumes:       Instruction in MBR,
        ;               BASE set to RB.MAIN,
        ;               MISC & g14.misc "correct"
        ;
        ;exits to:      memory reference operand fetch or other class
        ;
        ;With:          TEMP:   Current page
        ;               MIR:    Copy of instruction
        ;               No memory activity (or cycle 3)

        ;System convention:     Go to "INTS" with MBR restored
        ;                       BASE set to RB.MAIN
        ;                       MISC "correct", and copied to g14.misc

020050          uram 20050

020050  main:   mbr->mir                  ; put macroinstruction in mir
020051          disp(firstd)->upc         ; dispatch
020052          LC.pc&177000->temp        ; get current macro page
        ; [eah] used to be " & 77000 " ...
```

```
        ;Initialization (start the macroprogram emulation)


        ;       This is called as a subroutine from MDDT (using "G").
        ;If the status is ZERO, no argument was given to the "G" command,
        ;and a "CONTINUE" is performed.  If the status is NONZERO, an
        ;argument was supplied with the "G" command, meaning "MASTER
        ;CLEAR and start at the PC given by the argument (in the MBR)".

021360          uram unext

        start:
021360   g7.base->base                    ;g7.base always holds "rb.main"
021361   base->g13.mbr                    ;save the old base (g13.mbr is safe)

021362   mbr->L0.pc                       ;load the pc for "start"

021363   1->g3.rone                       ;setup the global registers
021364   100000 -> g4.rsign

021365   0->L4.b                          ;clear B, SP, X, C
021366   0->L11.sp
021367   0->mbr,L5.x
021370   0->mar(w)
021371   0->L7.cbit                       ;[1] clear the H316 carry bit

021372   rb.softint -> base               ;[2] set up the rest of the init code
021373   0 -> 17.pmscan                   ;say we need no scans, just in case
021374   0 -> 15.inspc                    ;no instruction debreak
021375   g14.misc & ~(m.modef0!m.modef1!m.modef2) -> g14.misc ;in hardware an
021376   g14.misc ! (m.progint!m.modef0!m.modef2) -> g14.misc, misc ;also set
021377   rb.ioos -> base                  ;[1] switch bases
021400   pm.clear -> upc                  ;[2] clear the process manager
021401   (.+1) -> 12.ret2                 ;[3][4]
021402   io.clear -> upc                  ;clear the IO world, too
021403   (.+1) -> 12.ret2
021404   rb.softint -> base               ;a watch pending?
021405   14.wmar ls
021406   ifnot zero watch.go -> upc       ;yes, start it
021407   ifnot zero (.+1) -> 11.ret1
021410   g13.mbr -> base                  ;restore USYS's BASE
021411   11.ret1 -> upc                   ;return to it
021412   nop
```

```
                ;Memory reference instruction operand fetch

                ;Enter here from MAIN (through first dispatch) when the opcode is
                ;a memory reference type. There are eight basic types of memory refer
                ;address calculations possible: normal, index, indirect, indexed indi
                ;for both page zero (sector bit = 0) and the current page (sector bit
                ;The sector, index and indirect bits of the instruction are part of
                ;the dispatch address computation so we can branch to one of eight
                ;labels (below) to handle a given instruction. MAIN sets TEMP equal
                ;to the page number bits of the PC (L0.PC .and. 177000) for use
                ;in the current page flavor of the operand fetch.

                ex0:                            ;Page zero
021413           mirfld->mar(r)
021414           disp(op)->upc                  ; [1]
021415           nop                            ; [2]

                ex0i:                           ;Page zero, indirect
021416           mirfld->mar(r)
021417           nop                            ; [1]
021420           nop                            ; [2]

                ex0ia:                          ;(enter here from current page, indir
021421           mbr->mar(r)
021422           disp(op) -> upc                ;[1] dispatch
021423           nop                            ;[2]

                ex0x:                           ;Page zero, indexed
021424           mirfld+L5.x->mar(r)
021425           disp(op)->upc                  ; [1]
021426           nop                            ; [2]

                ex0ix:                          ;Page zero, indexed, indirect
021427           mirfld->mar(r)
021430           nop                            ;[1]
021431           nop                            ;[2]

                ex0ib:                          ;(enter here from current page, index
021432           mbr+L5.x->mbr
021433           mbr->mar(r)
021434           disp(op)->upc                  ;[1]
021435           nop                            ;[2]

                ex1:                            ;current page
021436           temp -> g0.temp
021437           mirfld ! g0.temp -> mar(r)
021440           disp(op) -> upc                ;[1]
021441           nop                            ;[2]

                ex1i:                           ;current page, indirect
021442           temp -> g0.temp
021443           mirfld ! g0.temp -> mar(r)
021444           ex0ia -> upc                   ;[1]
021445           nop                            ;[2]

                ex1x:                           ;curent page, indexed
021446           temp + l5.x -> temp
```

```
021447    temp -> g0.temp
02145C    mirfld + g0.temp -> mar(r)
021451    disp(op) -> upc                    ;[1]
021452    nop                                ;[2]

          ex1ix:                             ;current page, indexed, indirect
021453    temp -> g0.temp
021454    mirfld ! g0.temp -> mar(r)
021455    ex0ib -> upc                       ;[1]
021456    nop                                ;[2]
```

```
        ;Memory reference operators

        ;Assume:        Operand in MBR [third memory cycle]
        ;               Effective address in MAR
        ;
        ;Exit to:       Interrupt
        ;
        ;With:          MBR:    Next instruction [3rd cycle]
        ;               PC:     Pointing to current (new) instruction
        ;               CBIT:   Status updated if appropriate


021457  lda:    L0.pc+1->L0.pc,mar(r)
021460          ints->upc                       ; [1]
021461          mbr->L3.a                       ; [2] mbr stable until [3]

021462  ana:    L0.pc+1->L0.pc,mar(r)           ; [3]
021463          ints->upc                       ; [1]
021464          L3.a&mbr->L3.a                  ; [2]

021465  era:    L0.pc+1->L0.pc,mar(r)           ; [3]
021466          ints->upc                       ; [1]
021467          L3.a ? mbr->L3.a                ; [2]

021470  add:    L0.pc+1->L0.pc,mar(r)           ; [3]
021471          L3.a+mbr->L3.a  LS              ; [1]
021472          ints->upc                       ; [2]
021473          alust->L7.cbit                  ; [3]

021474  sub:    L0.pc+1->L0.pc,mar(r)           ; [3]
021475          L3.a-mbr->L3.a  LS              ; [1]
021476          alust->L7.cbit                  ; [2]
021477          ints->upc                       ; [3]
021500          L7.cbit?alu.cbit->L7.cbit       ; alu latches carry backwards
                                                ; on subtracts

021501  jmp:    mar ls                          ;look at PC before using it
021502          ifnot zero ints->upc            ; next instruction is already
021503          ifnot zero mar -> l0.pc
                                                ; in mbr (since it is this
                                                ; instruction's "operand"), so
                                                ; don't fetch

021504          crash -> upc                    ;else trap jump to zero
021505           6 -> temp                      ;JST/JMP to zero

021506  sta:     L3.a->mbr
021507          mar->mar(w) ls
021510          if zero mbr->L5.x               ; [1]
021511          nop                             ; [2]
021512          L0.pc+1->L0.pc,mar(r)           ; [3]
021513          ints->upc                       ; [1]
021514          0                               ; [2]
```

```
        ;       memory reference operators


021515  jst:    LC.pc+1->L0.pc, mbr
021516          mar->mar(w) ls
021517          if zero crash -> upc    ;[1] JST 0 is illegal
021520          if zero 6 -> temp       ;[2]
021521          g3.rone + mar -> mar(r) ; [3]
021522          ints->upc               ; [1]
021523          mar->L0.pc

021524  cas:    L3.a?mbr          LS    ; [3]
021525          if neg cas1->upc
021526          L3.a-mbr          LS    ; same sign, do subtract
021527          if neg L0.pc+2->L0.pc
021530          if zero L0.pc+1->L0.pc
021531          L0.pc+1->L0.pc,mar(r)
021532          ints->upc               ; [1]
021533  cas1:   L3.a     LS             ; [2] different sign
021534          if neg L0.pc+2->L0.pc
021535          L0.pc+1->L0.pc,mar(r)
021536          ints->upc               ; [1]
021537          0                       ; [2]

021540  ima:    mbr->temp
021541          L3.a->mbr
021542          mar->mar(w) ls
021543          if zero mbr->L5.x
021544          fetch->upc              ; [1]
021545          temp->L3.a              ; [2]
```

```
        ;       memory reference operators

021546  irs:    mbr+g3.rone->mbr ls
021547          mar->mar(w)
021550          if zero L0.pc+1->L0.pc  ; [1]
021551          mar  LS                 ; [2] is it to the index reg?
021552          L0.pc+1->L0.pc,mar(r)   ; [3]
021553          ints->upc               ; [1]
021554          if zero mbr->L5.x       ; [2]

021555  ldx:    mbr->mbr,L5.x           ; setup the edac syndrome
021556          0->mar(w)               ; connector fixes 40000 bit ;[eah]?
021557          fetch->upc              ; [1]
021560           nop                    ; [2]

021561  stx:    L5.x->mbr
021562          mar->mar(w)
021563          fetch->upc              ; [1]
021564           nop                    ; [2]
```

```
        ;     Control instructions and utility routines
        ;                Note: also see interrupt-related instructions
        ;       Note:    come here to do a macro "NOP", or to finish
        ;                current macroinstruction emulation by fetching
        ;                the next macroinstruction
        ;       Also:    (fetch->upc costs nothing on memory cycle 1)

        ;       Various NOPs, unimplemented (optional M316) instructions,
        ; and added/augmented instructions come here.


        skp:
        skip:                                   ;The generic "SKIP"
 021565  10.pc + 2 -> 10.pc, mar(r)             ;bump the PC and fetch
 021566  ints -> upc                            ;[1]
 021567  nop                                    ;[2]

        nopp:
        fetch:
 021570  10.pc + 1 -> 10.pc, mar(r)             ;fetch the next Macro-Instruction
 021571  ints -> upc                            ;[1] dismiss to the next micro-task
 021572  nop                                    ;[2]

        fetch.this:                             ;enter here to re-fetch the current
 021573  rb.main -> base                        ;fix base
 021574  10.pc -> mar(r)                        ;prefetch it
 021575  ints -> upc                            ;[1]
 021576  nop                                    ;[2]

        fetch.base:
 021577  rb.main -> base
 021600  10.pc + 1 -> 10.pc, mar(r)
 021601  ints -> upc                            ;[1]
 021602  nop                                    ;[2]

        notyet:
 021603  crash -> upc                           ;not yet implemented
 021604  2 -> temp

        hlt:
 021605  crash -> upc                           ;the basic "HALT the Processor"
 021606  1 -> temp

        iab:
 021607  L3.A -> temp                           ; "Exchange A and B"
 021610  L4.B -> mbr
 021611  mbr -> L3.A
 021612  L0.PC + 1 -> L0.PC, mar(r)             ;start the next instruction fetch
 021613  ints -> upc                            ; [1]
 021614  temp -> L4.B                           ; [2]
```

```
;     Notes on the carry bit:


; The H316 has one "carry" bit.  These instructions affect it
; (taken from Honeywell manual, page 2-14):
;
;                Reset if                  Set if
;
;        ADD     No overflow               Overflow
;        SUB     No overflow               Overflow
;        ACA     A was not 077777          A was 077777
;        ACA     A was not 077777 or       A was 077777 and
;                carry bit was 0           carry bit was 1
;        RCB     (Always)                  (Never)
;        SCB     (Never)                   (Always)
;        CSA     A was positive            A was negative
;
;     Shifts     Tricky:  See Honeywell manual, page 3-6
;                (page 2-14 is wrong)
;
; Addition X+Y=Z has overflow iff x and y have the same
; sign, and z has a different sign.  Subtraction X-Y=Z
; has overflow iff y and z have the same sign, and x
; has a different sign.  Emulating the H316 carry
; bit is hardest for macro addition and subtraction,
; so the discussion below will focus on these.
;
;
; The MBB's ALU status word is latched with these bits:
;
;        X XXX XXX XPQ RSX XOZ
;
;        P = High order bit of register input to alu
;        Q = High order bit of source bus input to ALU
;        R = High order bit of alu result
;        S = ALU carry
;        O = ALU result, low bit (not relevant here)
;        Z = (1 IFF ALU result=0) (not relevant here)
;        X = Unused bit
;
; After an addition or a subtraction, the alu carry bit is
; what you would expect for straight binary addition and
; subtraction, reflecting an old-fashioned "carry-out"
; on addition and "borrow" on subtraction.
;
;
; A macro ADD of accumulator to memory is emulated by
; putting the accumulator's contents in the alu's register
; input, putting the memory's contents in the alu's source
; bus input, and performing an ADD.  A macro SUBTRACT of
; memory from accumulator is emulated by putting the
; accumulator's contents in the alu's register input,
; the memory's contents in the alu's source bus input,
; and doing a SUBTRACT.  (Just as a macro SUBTRACT always
; SUBTRACTS memory from accumulator, so does a micro
; SUBTRACT always SUBTRACT source bus from register.)
; If the MBB's ALU had an "overflow" status bit similar to
; that of the H316, we could use that bit to compute the
```

```
; H316 carry bit after adds and subtracts.  Although the ALU
; has no such status bit, the "overflow" condition can be
; computed from the bits that are kept.  Consider the ALU as
; doing a straight binary addition or subtraction.  Let
; C0 be the carry-out from (or borrow into) the high order bit.
; Let C1 be the carry into (or borrow from) the high order bit.
; It is well known, and straightforward to verify by cases,
; that the "overflow" bit, for both addition and subtraction,
; is just C0?C1.  Now C0 is just the alu's carry status bit,
; and C1 is the exor of all three high-order bits
; (of the alu's two inputs and one output).  thus, the "overflow"
; bit is just the exor of all four alu status bits listed
; above.
;
;
; It is easy to save the ALU status word but hard
; to EXOR the relevant bits.  Therefore the strategy
; for emulating the H316 carry bit is as follows.
; after each macro addition or subtraction, the
; alu status is saved in a dedicated register, "CBIT".
; when the carry bit is needed, register "CBIT" is
; decoded by exoring the relevant bits.  (only "SSC",
; "SRC", "ACA", and "INK" use the carry bit.)
;
; Of course, not only addition and subtraction, but
; also the other macro instructions listed above,
; must properly set "CBIT".  this is not hard.
; the program uses two constants, "CBITSET" and
; "CBITRESET", which when loaded into "CBIT"
; will be decoded as "carry set" and "carry reset"
; respectively.  To emulate a macro instruction
; which must compute "CBIT", the program can decide whether
; the carry bit is set or reset and then load "CBITSET"
; or "CBITRESET" into "CBIT".
;
; Actually, for the macro instructions "AOA" and "ACA",
; the microcode can take a short cut.  The H316 carry
; bit is determined just as for a macro addition
; (accumulator+1 or accumulator+carry bit).  The
; microcode can therefore simply perform a micro addition
; and save the ALU status in CBIT, just as it does
; for the "ADD" instruction.



; * * The above is wrong in one detail.  For microsubtracts,
; the MBB's alu really latches the carry bit as the
; complement of the "borrow" bit.  For example,
; 2-1=1 has no borrow and therefore sets the ALU's
; carry, while 1-2=-1 has a borrow and therefore
; resets the ALU's carry.  The microcode fixes this
; oddity by toggling the carry bit as it saves
; away the ALU status after a relevant microsubtract.
; In fact, the only relevant microsubtract is that
; used in emulating a macrosubtract (at label "SUB").
```

```
        ;       Arithmetic and halfword instructions


021615  icl:    L3.a->mbr
021616          smbr->L3.a
021617  cal:    L0.pc+1->L0.pc,mar(r)
021620          ints->upc                       ; [1]
021621          0377&L3.a->L3.a                 ; [2]

021622  icr:    L3.a->mbr
021623          smbr->L3.a
021624  car:    L0.pc+1->L0.pc,mar(r)
021625          ints->upc                       ; [1]
021626          177400&L3.a->L3.a               ; [2]

021627  aoa:    L3.a+1->L3.a    LS
021630          L0.pc+1->L0.pc,mar(r)
021631          ints->upc                       ; [1]
021632          alust->L7.cbit                  ; [2] ok since aoa sets macro carry b
                                                ; same as macro add of 1 to accumulat

021633  aca:    carry->upc                      ; decode the macro carry bit
021634           (.+1)->L1.ret1
021635          ifnot zero 1->temp              ; temp has carry bit (0 or 1)
021636          L3.a+temp->L3.a LS
021637          L0.pc+1->L0.pc,mar(r)
021640          ints->upc                       ; [1]
021641          alust->L7.cbit                  ; [2]  ok because aca sets macro carr
                                                ; bit same as macro add of carry
                                                ; to accumulator


021642  ica:    L3.a->mbr
021643          L0.pc+1->L0.pc,mar(r)
021644          ints->upc                       ; [1]
021645          smbr->L3.a                      ; [2] mbr changes at [3]


        ; Routine CARRY, level 1, arg<L7.cbit>, uses<mar,g0,temp>
        ; Decodes L7.cbit
        ;   If M316 carry bit is reset, returns:
        ;     0 in g0
        ;     0 in temp
        ;     ALU status "zero"
        ;   If M31o carry bit is set, returns:
        ;     CBITCOLL in g0 (CBITCOLL is nonzero)
        ;     CBITCOLL in temp
        ;     ALU status "nonzero"

021646  carry:  L7.cbit->mar                   ; collect the four bits of L7.cbit
021647          mar(sr)->g0                     ; into one bit
021650          mar(sr)
021651          mar?g0->mar,g0
021652          mar(sr)
021653          mar?g0->g0
021654          L1.ret1->upc
021655          cbitcoll&g0->temp,g0    LS ; test and save that bit
```

```
        ;     Arithmetic and halfword instructions


021656  chs:   L0.pc+1->L0.pc,mar(r)
021657         ints->upc                  ; [1]
021660         L3.a ? 100000->L3.a        ; [2]

021661  cra:   L0.pc+1->L0.pc,mar(r)
021662         ints->upc                  ; [1]
021663         0->L3.a                    ; [2]

021664  ssp:   L0.pc+1->L0.pc,mar(r)
021665         ints->upc                  ; [1]
021666         L3.a&msk77777->L3.a        ; [2]

021667  rcb:   L0.pc+1->L0.pc,mar(r)
021670         ints->upc                  ; [1]
021671         cbitreset->L7.cbit         ; [2]

021672  csa:   cbitreset->L7.cbit
021673         L3.a    LS
021674         if neg cbitset->L7.cbit
021675         L0.pc+1->L0.pc,mar(r)
021676         ints->upc                  ; [1]
021677         L3.a&msk77777->L3.a        ; [2]

021700  cma:   L0.pc+1->L0.pc,mar(r)
021701         ints->upc                  ; [1]
021702         177777 ? L3.a->L3.a        ; [2]

021703  tca:   177777 ? L3.a->L3.a
021704         L0.pc+1->L0.pc,mar(r)
021705         ints->upc                  ; [1]
021706         L3.a+1->L3.a               ; [2]

021707  ssm:   L0.pc+1->L0.pc,mar(r)
021710         ints->upc                  ; [1]
021711         100000!L3.a->L3.a          ; [2]

021712  scb:   L0.pc+1->L0.pc,mar(r)
021713         ints->upc                  ; [1]
021714         cbitset->L7.cbit           ; [2]
```

```
        ;       skip instructions


        ; Skp:  (done by srx 0)

021715  src:    carry->upc              ; decode carry bit
021716           (.+1)->L1.ret1
021717          if zero L0.pc+1->L0.pc  ; if carry bit reset, skip
021720          L0.pc+1->L0.pc,mar(r)
021721          ints->upc               ; [1]
021722           nop                    ; [2]

021723  sze:    L3.a            LS
021724          if zero L0.pc+1->L0.pc
021725          L0.pc+1->L0.pc,mar(r)
021726          ints->upc               ; [1]
021727           nop                    ; [2]

021730  slz:    L3.a            LS
021731          ifnot odd L0.pc+1->L0.pc
021732          L0.pc+1->L0.pc,mar(r)
021733          ints->upc               ; [1]
021734           nop                    ; [2]

021735  spl:    L3.a            LS
021736          ifnot neg L0.pc+1->L0.pc
021737          L0.pc+1->L0.pc,mar(r)
021740          ints->upc               ; [1]
021741           nop                    ; [2]
```

```
        ;       Skip instructions


        ; nopp:  (done by ssx 0)

021742  ssc:    carry->upc                  ; decode carry bit
021743           (.+1)->L1.ret1
021744           ifnot zero L0.pc+1->L0.pc ; if carry bit set, skip
021745           L0.pc+1->L0.pc,mar(r)
021746           ints->upc                  ; [1]
021747            nop                       ; [2]

021750  snz:    L3.a              LS
021751           ifnot zero L0.pc+1->L0.pc
021752           L0.pc+1->L0.pc,mar(r)
021753           ints->upc                  ; [1]
021754            nop                       ; [2]

021755  sln:    L3.a              LS
021756           if odd L0.pc+1->L0.pc
021757           L0.pc+1->L0.pc,mar(r)
021760           ints->upc                  ; [1]
021761            nop                       ; [2]

021762  smi:    L3.a              LS
021763           if neg  L0.pc+1->L0.pc
021764           L0.pc+1->L0.pc,mar(r)
021765           ints->upc                  ; [1]
021766            nop                       ; [2]
```

```
;     Notes on shift instructions:


; Shift instructions are octal numbers XXXXNN,
; where XXXX is the op code and nn the shift count.
;
; NN is 77 to shift one bit, 76 to shift two bits,
; and so on, with 01 specifying a shift of 63. bits.
; 00 is funny:  no shift, but always reset the C-bit.
;
; There are twelve kinds of shifts:  left or right;
; long (uses a and b registers) or short (uses
; only a register); and logical (end-off, zero-fill),
; arithmetic (special attention to sign bit), or
; rotate.  the op code is 04yy, where yy is the sum of:
;       10 if left
;        4 if not long
;        1 if arithmetic
;        2 if rotate
;
;
; Here are the op codes, with the mnemonics and
; names from the Honeywell manual, pages 3-5 to 3-6.
; For further details, including the behavior of the
; "carry bit", see the manual.
;
; LGR 0404 Logical right shift
; LRL 0400 Long logical right shift
; LGL 0414 Logical left shift
; LLL 0410 Long logical left shift
;
; ARS 0405 Arithmetic right shift
; LRS 0401 Long arithmetic right shift
; ALS 0415 Arithmetic left shift
; LLS 0411 Long arithmetic left shift
;
; ARR 0406 Right rotate
; LRR 0402 Long right rotate
; ALR 0416 Left rotate
; LLR 0412 Long left rotate
;
; These mnemonics and names are awkward.  We keep
; the mnemonics, since they appear in every macro-program.
; below, we simplify the names to the format
;
;       (A) B C
;
;Where
;
;       A = "long"
;       B = "right" or "left"
;       C = "logical" or "arithmetic" or "rotate"
;
; The M8B will also do a rotate if the op code's 1 bit
; is on.  Thus, for example, 0407 generates a right rotate.
; The H316 seems to have this undocumented feature.
; The current H316 software actually uses that such instructions
; will actually produce a shift of zero bit places
```

```
        ; in the case of a shift count of zero.  For example, the
        ; software would rely on 040700 producing a shift of zero places.
        ;
        ;
        ; Dispatch goes to a routine below such as LRL, LGR,
        ; etc.  This routine loads the mar and mbr as
        ; appropriate, calls shflup to do shifts, and
        ; then reads mar and mbr to get the result.
        ; Shflup branches out on logical/arithmetic/rotate,
        ; and then on right/left.  The twelve dispatch
        ; addresses for the twelve types of instructions
        ; can be collapsed insofar as loading mbr, mar and
        ; getting the result is the same.  This happens
        ; as follows:
        ;       3 longs: left and right are the same
        ;       1 longs: rotate same as logical
        ;       1 non-long rotates: left and right are the same
        ;       1 non-long lefts: rotate same as arithmetic
        ;                         (doesn't work for right because
        ;                         sign bit must be extended)
```

```
        ;       Shift instructions


        ; LRL - Long right logical

021767  lrl:    L3.a->mar
021770           L4.b->mbr
                 ; call shflup
021771            shflup->upc
021772            (.+1)->L1.ret1
021773           mar->L3.a
021774           fetch->upc
021775           mbr->L4.b

        ; LRS - Long right arithmetic = Long left arithmetic

022021  lrs=lls

        ; LRR - Long right rotate = Long right logical

021767  lrr=lrl

        ; LGR - Right logical

021776  lgr:    0->mar
021777           L3.a->mbr
                 ; call shflup
022000            shflup->upc
022001            (.+1)->L1.ret1
022002           fetch->upc
022003           mbr->L3.a

        ; ARS - Right arithmetic

022004  ars:    0->mar
022005           L3.a->mbr                LS
022006           if neg 177777->mar
                 ; call shflup
022007            shflup->upc
022010            (.+1)->L1.ret1
022011           fetch->upc
022012           mbr->L3.a

        ; ARR - Right rotate

022013  arr:    L3.a->mar
022014           L3.a->mbr
                 ; call shflup
022015            shflup->upc
022016            (.+1)->L1.ret1
022017           fetch->upc
022020           mar->L3.a
```

```
        ;       Shift instructions


        ; LLL - Long left logical = Long right logical

021767  lll=lrl

        ; LLS - Long left arithmetic

022021  lls:    L4.b->mbr
022022          mar(sl)
022023          L3.a->mar
                ; call shflup
022024           shflup->upc
022025           (.+1)->L1.ret1
022026          mar->L3.a
022027          0->mar
022030          mbr     LS
022031          mar(sr)
022032          cbitreset->mar          ; not needed if we assume 0 in
                                        ; cbit is decoded as "carry reset"
022033          if odd cbitset->mar
022034          mir->g0
022035          g0&1000         LS
022036          if zero mar->L7.cbit    ; fix c-bit for long right
022037          L4.b    LS
022040          if neg g4.rsign!mbr->mbr
022041          fetch->upc
022042          mbr->L4.b

        ; LLR - Long left rotate = Long right rotate

021767  llr=lrr

        ; LGL - Left logical

022043  lgl:    0->mbr
022044          L3.a->mar
                ; call shflup
022045           shflup->upc
022046           (.+1)->L1.ret1
022047          fetch->upc
022050          mar->L3.a

        ; ALS - Left arithmetic = Left logical

022043  als=lgl

        ; ALR - Left rotate = Right rotate

022013  alr=arr
```

```
        ;       Shift loops


        ; On entry:      MIR =>  shift instruction
        ;                MAR<->MBR => A and B as needed
        ;                L1.ret1 => The return address
        ;
        ;
        ; These routines shift the combined mar and mbr the number of
        ; places and direction specified in the instruction.  At the end
        ; of the shift, the H316 carry bit is set as specified
        ; in the instruction.
        ;
        ; Shflup uses its own register block.  It restores the block
        ; before returning to the calling routines (lgl, etc.).
        ; On return, it passes its local carry bit register, shfcbit,
        ; to the caller's carry bit register, cbit.


 022051 shflup: rb.mshift->base
 022052         mir->L4.sc
 022053         L4.sc&77->L4.sc LS      ; get shift count
 022054         if zero g7.base->base
 022055         if zero fetch->upc      ; the rest of these routines can
                                        ; know that at least 1 shift is neede
 022056         if zero cbitreset->L7.cbit ; clear carry bit to simulate
                                        ; H316's funny shift of zero places
 022057         cbitreset->L0.shfcbit   ; clear carry bit so that routines
                                        ; below need only set it as needed
 022060         L4.sc+(-100)->L4.sc
 022061         mir->g0
 022062         g0&200          LS      ; test: rotate if 1
                                        ; (do this test first
                                        ; to support undocumented rotates.
                                        ; for example, 403xx is a rotate,
                                        ; not an arithmetic.  for more info,
                                        ; see notes above.)
 022063         ifnot zero shfrot->upc
 022064         g0&100          LS      ; test: arith shift if 1
 022065         ifnot zero shfarith->upc
 022066         if zero shfsimple->upc  ; otherwise, simple ("logical") shift
 022067         0


                                        ; everyone returns here
 022070 shflup1: msk177777->g0          ; **convenience clear bits 16-19
                                        ; (unnecessary; just makes ddt printo
                                        ;  easier to read, with high order ze
 022071         mar&g0->mar
 022072         mbr&g0->mbr

 022073         L0.shfcbit->temp
 022074         g7.base->base
 022075         L1.ret1->upc
 022076         temp->L7.cbit
```

```
        ;     shift loops - simple shifts

022077  shfsimple: g0&1000 LS          ; test: left if 1 (g0 has macroinstru
022100          ifnot zero shfl->upc


                ; simple right shift

022101  shfr:    shfr1->L1.ret1
022102  shfr1:   if intp shfint->upc
022103          0
022104          L4.sc+1->L4.sc  LS
022105          if neg shfr1->upc
022106          if neg mar(sr)
022107          mbr             LS
022110          mar(sr)                 ; do the final shift
022111          shflup1->upc
022112          if odd cbitset->L0.shfcbit

                ; simple left shift

022113  shfl:    shfl1->L1.ret1
022114  shfl1:   if intp shfint->upc
022115          0
022116          L4.sc+1->L4.sc  LS
022117          if neg shfl1->upc
022120          mar(sl) LS
022121          shflup1->upc
022122          if neg cbitset->L0.shfcbit
```

```
                ;       shift loops - rotates

022123  shfrot: g0&1000  LS                ; test: left if 1 (g0 has macroinstru
022124          ifnot zero shflr->upc

                ; right rotate

022125  shfrr:  shfrr1->L1.ret1
022126  shfrr1: if intp shfint->upc
022127          mbr                 LS
022130          mar(sr)
022131          if odd g4.rsign!mar->mar
022132          L4.sc+1->L4.sc  LS
022133          if neg shfrr1->upc
022134          mar                 LS
022135          shflup1->upc
022136          if neg cbitset->L0.shfcbit

                ; left rotate

022137  shflr:  shflr1->L1.ret1
022140  shflr1: if intp shfint->upc
022141          mar                 LS      ; these two cannot be combined
022142          mar(sl)                     ; because of possible interrupt
022143          if neg g3.rone!mbr->mbr
022144          L4.sc+1->L4.sc  LS
022145          if neg shflr1->upc
022146          mbr                 LS
022147          shflup1->upc
022150          if odd  cbitset->L0.shfcbit
```

```
        ;       shift loops - arithmetic shifts


022151  shfarith: g0&1000 LS              ; test: left if 1 (g0 has macroinstru
022152          ifnot zero shfla->upc

                ; Right arithmetic shift

022153  shfra:   shfra1->L1.ret1
022154          mar->L2.shftmp
022155          L2.shftmp&100000->L2.shftmp ;sign bit to propagate
022156  shfra1: if intp shfint->upc
022157          mar!L2.shftmp->mar
022160          L4.sc+1->L4.sc  LS
022161          if neg shfra1->upc
022162          if neg mar(sr)
022163          mbr             LS
022164          mar(sr)
022165          mar!L2.shftmp->mar
022166          shflup1->upc
022167          if odd cbitset->L0.shfcbit
                                        ;for long shift

                ; Arithmetic left shift

022170  shfla:   shfla1->L1.ret1
022171          mar->L2.shftmp
022172          L2.shftmp&100000->L2.shftmp ;sign bit: will watch for a chang
022173  shfla1: if intp shfint->upc
022174          mar?L2.shftmp   LS
022175          if neg cbitset->L0.shfcbit
022176          L4.sc+1->L4.sc  LS
022177          if neg shfla1->upc
022200          mar(sl)
022201          mar?L2.shftmp   LS
022202          shflup1->upc
022203          if neg cbitset->L0.shfcbit
```

```
        ;Shift/Rotate interrupt debreak

        shfint:
022204   mar -> 13.shfmar              ;save current MAR
022205   shfret -> temp                ;set up call to INT.INSTR
022206   int.instr -> upc              ;transfer there
022207   rb.softint -> base            ;set up base

        shfret:
02221C   rb.mshift -> base            ;restore base
022211   11.ret1 -> upc                ;return
022212   13.shfmar ->mar               ;restore MAR
```

```
        ;Special combination instructions


        ; SZESRC  100041  SZE!SRC  Skip if zero and carry reset
        ; CARICL  141144  CAR!ICL  Copy left half of A to right
        ; CALICR  141250  CAL!ICR  Copy right half of A to left

 022213 szesrc: L3.a    LS
 022214         if zero src->upc
 022215         ifnot zero L0.pc+1->L0.pc,mar(r)
 022216         ints->upc                ; [1]
 022217          nop                     ; [2]

 022220 caricl: L3.a&177400->mbr,L3.a
 022221         L0.pc+1->L0.pc,mar(r)
 022222         ints->upc                ; [1]
 022223         smbr?L3.a->L3.a          ; [2]

 022224 calicr: L3.a&377->mbr,L3.a
 022225         L0.pc+1->L0.pc,mar(r)
 022226         ints->upc                ; [1]
 022227         smbr?L3.a->L3.a          ; [2]
```

```
        ;Definitions for JFFO. These definitions cleave a sixteen bit
        ;word into logarithmically-decreasing sub-words. By successive
        ;application of these masks to the input value, JFFO can compute
        ;the number of leading zeroes in a word in LOG2(16) iterations,
        ;where a linear search requires 15 (worst case).

100000  first1 ==       100000
040000  second1 ==      first1 / 2
020000  third1 ==       second1 / 2
010000  fourth1 ==      third1 / 2
004000  fifth1 ==       fourth1 / 2
002000  sixth1 ==       fifth1 / 2
001000  seventh1 ==     sixth1 / 2
000400  eighth1 ==      seventh1 / 2

000200  nineth1 ==      eighth1 / 2
000100  tenth1 ==       nineth1 / 2
000040  eleventh1 ==    tenth1 / 2
000020  twelfth1 ==     eleventh1 / 2
000010  thirteenth1 ==  twelfth1 / 2
000004  fourteenth1 == thirteenth1 / 2
000002  fifteenth1 ==  fourteenth1 / 2
000001  sixteenth1 ==  fifteenth1 / 2

140000  first2 ==       first1 ! second1
030000  second2 ==      first2 / 4
006000  third2 ==       second2 / 4
001400  fourth2 ==      third2 / 4

000300  fifth2 ==       fourth2 / 4
000060  sixth2 ==       fifth2 / 4
000014  seventh2 ==     sixth2 / 4
000003  eighth2 ==      seventh2 / 4

170000  first4 ==       first2 ! second2
007400  second4 ==      first4 / 20
000360  third4 ==       second4 / 20
000017  forth4 ==       third4 / 20

177400  first8 ==       first4 ! second4
000377  second8 ==      first8 / 400
```

```
        ;Machine crash values returned by this module
000100  random.0 ==     100             ;base of errors
000100  random.jffz = random.0 + 0      ;JFFO called with arg=0
```

```
        ;FFO instruction

        ffo:
 02223C   13.a -> temp ls                ;called with zero?
 022231   if zero fetch -> upc           ;yes, non-skip
 022232   ifnot zero temp -> g0.temp     ;else call JFFO
 022233   jffo -> upc
 022234   (.+1) -> 11.ret1
 022235   g0.temp -> temp
 022236   10.pc + 2 -> 10.pc, mar(r)     ;and skip
 022237   ints -> upc
 022240   temp -> 13.a                   ;return with number in A
```

```
        ;Subroutine JFFO, Level 1

        ;Enter with quantity in G0.TEMP, returns number of leading zeroes in
        ;same. Calling with G0 = 0 gets to crash with RANDOM.JFFZ.


        jffo:
022241   g0.temp ls                     ;check for illegal case
022242   if zero crash -> upc           ;if bogus, trap
022243   if zero random.jffz -> temp

022244   second8 -> mbr                 ;need to get at first8 but it's exten
                                        ;so we swap the low8 into the high8 u
022245   g0.temp & s16mbr ls            ;cleave into two halve-words
022246   if zero jffo.second8 -> upc    ;if first is zero, second must contai

        jffo.first8:
022247   g0.temp & first4 ls            ;try for upper four
022250   if zero jffo.second4 -> upc    ;if it's zero, must be second four

        jffo.first4:
022251   g0.temp & first2 ls            ;try four upper 2
022252   if zero jffo.second2 -> upc

        jffo.first2:
022253   g0.temp & first1 ls            ;(sign)
022254   if zero 1 -> g0.temp           ;if zero, it's second 1
022255   l1.ret1 -> upc
022256   ifnot zero 0 -> g0.temp        ;if non-zero, it's first 1 (sign)

        jffo.second2:
022257   g0.temp & third1 ls
022260   if zero 3 -> g0.temp
022261   l1.ret1 -> upc
022262   ifnot zero 2 -> g0.temp

        jffo.second4:
022263   g0.temp & third2 ls
022264   if zero jffo.fourth2 -> upc

        jffo.third2:
022265   g0.temp & fifth1 ls
022266   if zero 5 -> g0.temp
022267   l1.ret1 -> upc
022270   ifnot zero 4 -> g0.temp

        jffo.fourth2:
022271   g0.temp & seventh1 ls
022272   if zero 7 -> g0.temp
022273   l1.ret1 -> upc
022274   ifnot zero 6 -> g0.temp
```

```
        ;Somewhere in second8...

        jffo.second8:
022275   g0.temp & third4 ls
022276   if zero jffo.fourth4 -> upc

        jffo.third4:
022277   g0.temp & fifth2 ls
022300   if zero jffo.sixth2 -> upc

        jffo.fifth2:
022301   g0.temp & nineth1 ls
022302   if zero 11 -> g0.temp
022303   11.ret1 -> upc
022304   ifnot zero 10 -> g0.temp

        jffo.sixth2:
022305   g0.temp & eleventh1 ls
022306   if zero 13 -> g0.temp
022307   11.ret1 -> upc
022310   ifnot zero 12 -> g0.temp

        jffo.fourth4:
022311   g0.temp & seventh2 ls
022312   if zero jffo.eighth2 -> upc

        jffo.seventh2:
022313   g0.temp & thirteenth1 ls
022314   if zero 15 -> g0.temp
022315   11.ret1 -> upc
022316   ifnot zero 14 -> g0.temp

        jffo.eighth2:
022317   g0.temp & fifteenth1 ls
022320   if zero 17 -> g0.temp
022321   11.ret1 -> upc
022322   ifnot zero 16 -> g0.temp
```

```
        ;Miscellaneous instructions ( rdclok, memhi )

        rdclok:
022323   rb.clock -> base
022324   13.clk1 -> temp                ;read USYS 100 usec clock
022325   14.clk2 -> mir
022326   rb.main -> base                ;into A
022327   mir -> 14.b                    ;hi order into B
022330   10.pc + 1 -> 10.pc, mar(r)     ;and continue along
022331   ints -> upc                    ;[1]
022332   temp -> 13.a                   ;[2]

        memhi:
022333   rb.clock -> base               ;memsize is in rb.clock
022334   113.memsize - 4000 -> temp     ;subtract off for M8B use
022335   rb.main -> base
022336   10.pc + 1 -> 10.pc, mar(r)
022337   ints -> upc
022340   temp -> 13.a
```

```
        ;Queuing Primitives

        ; ENQ - add a new item before some existing item
        ; DEQ - remove the first (oldest) item
        ; RMQ - remove some arbitrary item
        ; MVQ - move one queue/heap to another (not implemented!)

        ;Queues are made up of one header and zero or more
        ;items. Items have 3 overhead words. Headers have the
        ;save 3, plus a length word

000000  q.forw == 0                     ;forward pointer
000001  q.back == 1                     ;backward pointer
000002  q.hedr == 2                     ;pointer to queue header
000003  q.leng == 3                     ;length (headers only)

        ;These instructions are capable of generating traps
        ;if consistency checks fail. The error codes returned
        ;are:

000020  qerr.0 == 20                    ;base for error codes
000020  qerr.illf == qerr.0 + 0         ;illegal forward pointer
000021  qerr.illb == qerr.0 + 1         ;illegal back pointer
000022  qerr.illh == qerr.0 + 2         ;illegal header pointer
000023  qerr.illl == qerr.0 + 3         ;illegal length

        ;Since these operations are used by both the macrocode
        ;and microcode, there are two parts to this implementation.
        ;The first part consists of the actual microcode to
        ;do the operations. This part takes its arguments
        ;in G0.TEMP and G1.TEMP, as needed. The second part
        ;makes up the interface to the 316 macrocode (taking its
        ;args in A and X and passing them to G0 and G1 as needed).

        ;Further documentation can be found in QUEUE.MSS (QUEUE.LPT).
```

```
        ;ENQ(item,next_item)

        ;Accepts in G0 a pointer to the new item to be added and in G1
        ;a pointer to the item which should succeed it when the operation
        ;finishes. ENQ has no fail return but can cause traps. Returns
        ;new length in MIR.

        q.enq:
022341    g1.temp -> mbr ls                 ;write next_item -> forw(item)
022342    ifnot zero g0.temp + q.forw -> mar(w) ls ;write
022343    if zero crash -> upc              ;[1] relies on q.forw==0, check for
022344     qerr.illf -> temp                ;[2]  ... illegal item arg

022345    g1.temp + q.back -> mar(r)        ;read back(next_item) to get current
022346    nop                               ;[1]  ... previous item
022347    nop                               ;[2]
022350    mbr -> mbr, g6.temp ls            ;save back(next_item) and check it
022351    g0.temp + q.back -> mar(w)        ;write it into our back pointer
022352    if zero crash -> upc              ;[1] back(next_item) cannot be zero
022353    if zero qerr.illb -> temp         ;[2]

022354    ifnot zero g1.temp + q.hedr -> mar(r) ;copy next_item's hedr into i
022355    nop                               ;[1]
022356    nop                               ;[2]
022357    mbr -> mbr ls                     ;check it, too
022360    g0.temp + q.hedr -> mar(w)
022361    if zero crash -> upc              ;[1] hedr cannot be at 0
022362    if zero qerr.illh -> temp         ;[2]

022363    mbr -> g2.temp
022364    ifnot zero g2.temp + q.leng -> mar(r) ;update the queue's length in
022365    nop                               ;[1]
022366    nop                               ;[2]
022367    g3.rone + mbr -> mbr
022370    mar -> mar(w)                     ;write it
022371    mbr -> mir ls                     ;[1]
022372    nop                               ;[2]

        ;At this point, the new item has all it's fields filled in
        ;and the length has been adjusted. We must now change the
        ;back pointer of NEXT_ITEM and the forward pointer of the
        ;old item before NEXT_ITEM to point to ITEM. A copy of
        ;NEXT_ITEM's Q.BACK word was cached in G6.TEMP above.

022373    g0.temp -> mbr
022374    g1.temp + q.back -> mar(w)        ;write NEXT_ITEM's back pointer
022375    nop                               ;[1]
022376    nop                               ;[2]
022377    g6.temp + q.forw -> mar(w)        ;write old previous item's forward po
022400    l1.ret1 -> upc                    ;return to caller
022401     1 -> temp ls                     ;set non-zero flag
```

```
        ;DEQ(header)

        ;Accepts a header pointer in G1.TEMP and returns the
        ;first item on that queue in G0.TEMP. If the queue is
        ;empty, the ZERO flag is set in ALUST, else non-zero
        ;is true. Returns new length in MIR.

        q.deq:
022402    g1.temp + q.leng -> mar(r)     ;adjust length (maybe)
022403    nop                            ;[1]
022404    -1 -> g2.temp                  ;[2] get a nice constant
022405    mbr + g2.temp -> mbr           ;update length, save away in MIR and
022406    mbr -> mir ls
022407    ifnot neg g1.temp + q.leng -> mar(w) ;write the new length back
022410    if neg l1.ret1 -> upc          ;[1] return with ZERO=true if so (fai
022411    if neg 0 ls                    ;[2]

        q.deq.rmq:                     ;(enter here to remove item following
022412    g1.temp + q.forw -> mar(r) ls ;read header's forward ptr
022413    if zero crash -> upc           ;[1] header cannot be at zero
022414     qerr.illh -> temp             ;[2]

022415    mbr -> mar(r), g0.temp ls      ;make sure forw(header) isn't zero
022416    if zero crash -> upc           ;[1]  ... and save forw(header) in G0
022417     qerr.illf -> temp             ;[2]
022420    mbr -> g2.temp ls
022421    g1.temp -> mbr                 ;write header -> back(forw(forw(heade
022422    g2.temp + q.back -> mar(w)
022423    if zero crash -> upc           ;[1] make sure forw(forw(header)) isn
022424     qerr.illf -> temp             ;[2]

022425    g2.temp -> mbr                 ;write (forw(header)) -> forw(header)
022426    g1.temp + q.forw -> mar(w)
022427    nop                            ;[1]
022430    nop                            ;[2]

        ;Now zero the first three (q.forw,q.back,q.hedr) words of
        ;the newly-removed item (address in G0.TEMP)

022431    0 -> mbr
022432    g0.temp + q.forw -> mar(w)
022433    nop                            ;[1]
022434    nop                            ;[2]
022435    g0.temp + q.back -> mar(w)
022436    nop                            ;[1]
022437    nop                            ;[2]
022440    g0.temp + q.hedr -> mar(w)
022441    l1.ret1 -> upc                 ;[1] return
022442     1 -> temp ls                  ;[2] set non-zero indicating good ret
```

```
        ;RMQ(item)

        ;Removes ITEM from whatever heap (queue) it's on. Accepts ITEM in
        ;G0.TEMP. Has no fail return but can go to crash. Returns new
        ;length in MIR. We do it by managing the Q.LENG counter here, then
        ;calling Q.DEQ.RMQ with G1.TEMP pointing to the item (or header)
        ;immediately preceding the thing we want to remove, thus allowing
        ;the common code to treat it as a DEQ.

        q.rmq:
 022443   g0.temp + q.hedr -> mar(r)     ;find the header
 022444   g0.temp ls                     ;[1] but check to be sure ITEM not at
 022445   if zero crash -> upc           ;[2]
 022446    qerr.illf -> temp

 022447   mbr -> g2.temp ls              ;fetch old length
 022450   g2.temp + q.leng -> mar(r)     ;and check header's address
 022451   if zero crash -> upc           ;[1]
 022452    qerr.illh -> temp             ;[2]

 022453   mbr -> g2.temp ls              ;read and test old length
 022454   if zero crash -> upc           ;if it's zero, RMQ impossible
 022455    qerr.illl -> temp
 022456   g2.temp - 1 -> mbr ls          ;decrement it now
 022457   mar -> mar(w)                  ;write it back
 022460   mbr -> mir                     ;[1] return new length
 022461   nop                            ;[2]

 022462   g0.temp + q.back -> mar(r)     ;find item before one to be removed
 022463   nop                            ;[1]
 022464   nop                            ;[2]
 022465   mbr -> g1.temp ls              ;set up G1.TEMP for DEQ.RMQ and test
 022466   ifnot zero q.deq.rmq -> upc
 022467   if zero crash -> upc           ;if it's zero, illegal!
 022470    qerr.illb -> temp
```

```
        ;ENQ 316 interface:
        ;       LDX ITEM
        ;       LDA TO_BE_NEXT_ITEM
        ;       ENQ
        ;        <went from 0 -> 1 element>
        ;        <normal return>

        q316.enq:
022471    15.x -> temp
022472    temp -> g0.temp
022473    13.a -> temp
022474    temp -> g1.temp
022475    q.enq -> upc
022476    (.+1) -> l1.ret1

022477    mir -> g2.temp
022500    g2.temp - 1 ls                      ;look at new length
022501    ifnot zero l0.pc + 1 -> l0.pc ;ifnot one now, skip
022502    l0.pc + 1 -> l0.pc, mar(r)    ;fetch next
022503    ints -> upc                         ;[1]
022504     nop                                ;[2]
```

```
        ;DEQ 316 Interface:
        ;       LDA QUEUE_HEADER
        ;       DEQ
        ;         <nothing to DEQ>
        ;         <here with item in X>

        q316.deq:
022505    13.a -> temp
022506    temp -> g1.temp
022507    q.deq -> upc
022510     (.+1) -> l1.ret1

022511    ifnot zero l0.pc + 1 -> l0.pc ;ZERO set by Q.DEQ
022512    g0.temp -> mbr
022513    0 -> mar(w)
022514    fetch -> upc                  ;[1]
022515     mbr -> 15.x                  ;[2]
```

```
        ;RMQ 316 interface:
        ;       LDX ITEM
        ;       RMQ
        ;        <queue went from 1 -> 0, now empty>
        ;        <normal return>

        q316.rmq:
022516    15.x -> temp
022517    temp -> g0.temp
022520    q.rmq -> upc
022521     (.+1) -> l1.ret1

022522    mir ls                                ;check length
022523    ifnot zero 10.pc + 1 -> 10.pc ;skip unless empty now
022524    10.pc +1 -> 10.pc, mar(r)
022525    ints -> upc                           ;[1]
022526     nop                                  ;[2]
```

```
        ;LITES instruction.  Takes A=low 16 bits, X=upper 4 bits, displays th
        ;in the MII board in slot 0's lights.

        lites:
022527    0 -> mbr
022530    15.x -> mar                   ;put low 4 bits of X into high 4 of M
022531    mar(sr)                       ;this works even in 16 bit mode
022532    mar(sr)                       ;see page 38 of MBB Blue Book
022533    mar(sr)
022534    mar(sr)
022535    mbr -> g0.temp                ;turn off the high 4 bits of the 16 b
022536    g0.temp & ~170000 -> mbr
022537    13.a & 177777 -> 13.a         ;make sure it's masked down
022540    13.a ! mbr -> mbr             ;fill in low 16 bits

        lites.patch:                    ;every board has lites, but not at th
                                        ;the same place.  the following two i
                                        ;are overlaid with something like...

022541    fetch -> upc                  ; board-address -> g0 (use some Gx.te
022542    nop                           ; g0 + lights-displacement -> mar(wio

022543    m2.ios -> misc2
022544    fetch -> upc
022545    nop
```

```
        ;CHK instruction

        chk:
022546   15.x -> mar(r)                    ;start the read anyway
022547   14.b ls                           ;[1] are we done yet?

        chk2:
022550   if zero fetch -> upc              ;[2] yes, all done
022551   ifnot zero 13.a + mbr -> 13.a     ;accumulate this word
022552   15.x + 1 -> 15.x, mar(r)          ;fetch next
022553   ifnot intp chk2 -> upc            ;[1] and close the loop
022554   14.b - 1 -> 14.b ls               ;[2]  ... counting as we go

        chk3:
022555   ints -> upc
022556   mir -> mbr

        ;BLT instruction

        blt:
022557   13.a -> mar(r)                    ;read the source
022560   14.b ls                           ;[1] done yet?

        blt2:
022561   if zero fetch -> upc              ;[2] yes, exit
022562   mbr -> mbr
022563   15.x -> mar(w)                    ;start the write
022564   15.x + 1 -> 15.x                  ;[1] update
022565   14.b - 1 -> 14.b ls               ;[2]
022566   13.a + 1 -> 13.a, mar(r)
022567   ifnot intp blt2 -> upc
022570   nop

        blt3:
022571   ints -> upc
022572   mir -> mbr
```

```
        ;PUSH instruction

        push:
022573   mbr -> mbr                          ;copy data
        push1:
022574   111.sp -> mar(w) ls                 ;write it
022575   ifnot zero fetch -> upc             ;[1] make sure SP wasn't at zero
022576   ifnot zero 111.sp - 1 -> 111.sp ;[2] update stack pointer

        stack.err:
022577   crash -> upc                        ;else crash
022600    5 -> temp

        ;PUSHA instruction

        pusha:
022601   push1 -> upc
022602   13.a -> mbr

        ;CALL instruction

        call:
022603   10.pc + 1 -> mbr                    ;set up to PUSH 10.pc
022604   mar -> 10.pc ls                     ;and branch there
022605   ifnot zero push1 -> upc
022606   if zero crash -> upc
022607   6 -> temp

        ;POP instruction

        pop:
02261C   mar -> g0.temp                      ;save instruction's effective address
022611   111.sp + 1 -> 111.sp, mar(r)        ;read out the stack entry
022612   111.sp - 1 ls                       ;[1] is SP at zero?
022613   if zero stack.err -> upc            ;[2]
022614   ifnot zero mbr -> mbr               ;update mbr
022615   g0.temp -> mar(w)
022616   fetch -> upc                        ;[1]
022617   nop                                 ;[2]

        ;POPA

        popa:
02262C   111.sp + 1 -> 111.sp, mar(r)        ;read out the stack entry
022621   111.sp - 1 ls                       ;[1] is SP at zero?
022622   if zero stack.err -> upc            ;[2]
022623   ifnot zero mbr -> 13.a
022624   10.pc + 1 -> 10.pc, mar(r)
022625   ints -> upc                         ;[1]
022626   nop                                 ;[2]

        ;RET instruction

        ret:
022627   111.sp + 1 -> 111.sp, mar(r)        ;read out the stack entry
02263C   111.sp - 1 ls                       ;[1] is SP at zero?
022631   if zero stack.err -> upc            ;[2]
```

```
022632   ifnot zero mbr -> 10.pc, mar(r)
022633   ints -> upc                          ;[1]
022634   nop                                  ;[2]
```

```
        ;RETSKP instruction

        retskp:
022635   l11.sp + 1 -> l11.sp, mar(r)   ;read out the stack entry
022636   l11.sp - 1 ls                  ;[1] is SP at zero?
022637   if zero stack.err -> upc       ;[2]
02264C   ifnot zero 1 -> 10.pc          ;set up for increment
022641   10.pc + mbr -> 10.pc, mar(r)
022642   ints -> upc                    ;[1]
022643   nop                            ;[2]

        ;SETSKP instruction

        setskp:
022644   l11.sp + 1 -> mar(r) ls        ;increment top of stack
022645   if zero stack.err -> upc       ;[1]
022646   nop                            ;[2]
022647   g3.rone + mbr -> mbr
02265C   l11.sp + 1 -> mar(w)
022651   fetch -> upc                   ;[1]
022652   nop                            ;[2]

        ;SP2A instruction

        sp2a:
022653   10.pc + 1 -> 10.pc, mar(r)
022654   13.a -> temp                   ;[1]
022655   ints -> upc                    ;[2]
022656   temp -> l11.sp

        ;A2SP instruction

        a2sp:
022657   10.pc + 1 -> 10.pc, mar(r)
02266C   l11.sp -> temp                 ;[1]
022661   ints -> upc                    ;[2]
022662   temp -> 13.a

        ;SP2X instruction

        sp2x:
022663   l11.sp -> mbr
022664   0 -> mar(w)
022665   fetch -> upc                   ;[1]
022666   mbr -> 15.x                    ;[2]
```

```
022667     unext == .
```

```
        ;Software interrupt logic

        ;Come here when all hardware interrupts have been processed. Check
        ;for any pended macroinstruction emulation (L5.INSPC has left-off
        ;address), then check for required call to PM.SCAN because some
        ;interrupt has snuck in and required a scheduling scan. If there
        ;are no other interrupts to processes, clear the programmable interru
        ;and return to FETCH as usual.

        soft:
 022667   15.inspc -> temp ls
 022670   ifnot zero soft.instr -> upc
 022671   if zero 17.pmscan -> temp ls
 022672   ifnot zero soft.scan -> upc
 022673   if zero 14.wmar -> mar(r) ls
 022674   ifnot zero soft.watch -> upc   ;[1]

 022675   if zero g14.misc & ~m.progint -> g14.misc, misc ;[2] nothing to do -
 022676   rb.main -> base                ;[1]
 022677   10.pc -> mar(r)                ;[2] prefetch next instr
 022700   nop                            ;[3] [1]
 022701   nop                            ;[4] [2]
 022702   ints -> upc                    ; (must wait full four cycles)
 022703   nop

        soft.instr:
 022704   0 -> 15.inspc                  ;clear it
 022705   temp -> upc                    ;run it
 022706   rb.main -> base                ;in RB.MAIN's base

        soft.scan:
 022707   0 -> 17.pmscan                 ;clear it
 022710   pm.scan -> upc                 ;run it
 022711   rb.ioos -> base                ;in NMFS base

        soft.watch:
 022712   16.wmbr ? mbr -> mbr           ;[3] compute which bits changed
 022713   13.wmask & mbr ls              ;any we care about?
 022714   ifnot zero crash -> upc
 022715   ifnot zero 3 -> temp           ;yes, give code 3 trap

 022716   rb.main -> base                ;all's well, go to main directly
 022717   10.pc -> mar(r)
 022720   main -> upc                    ;[1]
 022721   nop                            ;[2]
```

```
        ;Software microinterrupt, continued

        ;This routine is entered with a UPC in TEMP. It copies TEMP
        ;into L5.INSPC and then sets a software interrupt and goes to
        ;ints. In this way, it's like an instruction emulation level
        ;debreak. Enter with RB.SOFTINT.

        int.instr:
022722   g14.misc ! m.progint -> g14.misc, misc ;set up the interrupt
022723   temp -> l5.inspc                ;[1]stash our address
022724   rb.main -> base                 ;[2] change base for MAIN
022725   nop                             ;[3]
022726   nop                             ;[4]
022727   ints -> upc                     ; (must allow full four cycles)
022730   nop                             ; (because MISC affects INTS)

        ;Come to INT.SCAN when an interrupt wishes to debreak but cause
        ;a macro progress manager scan at the next opportunity. Preserves
        ;caller's BASE and returns through L1.

        int.scan:
022731   rb.softint -> base              ;change to our base
022732   base -> l0.base                 ;capture caller's base first
022733   g14.misc ! m.progint -> g14.misc, misc
022734   -1 -> l7.pmscan                 ;[1] set the flag
022735   l0.base -> base                 ;[2] restore caller's base
022736   l1.ret1 -> upc                  ;[3] return
022737   nop                             ;[4] [1]
```

```
022740          unext == .
        # 17 "m.mic"
```

```
# 1 "console.mic"
```

```
        ;  This file contains additional commands added to the uDDT.
        ;  The following commands are implemented:
        ;
        ;       P       display the PC register
        ;       A       display the A register
        ;       B       display the B register
        ;       X       display the X register
        ;       N       display the current PCB register
        ;       S       display the current stack pointer
        ;       W       halt when specified macromemory word changes

000200  preg == rb.main+10
000203  areg == rb.main+13
000214  breg == rb.main+14
000205  xreg == rb.main+15
000206  nreg == rb.main+16              ;current PCB
000210  sreg == rb.main+110
```

```
020116          uram pdt.uram.e         ;patch into the command chain

020116          ifnot zero pdt.more -> upc
020117           g0.temp - ascii.P LS

        ;decoding chain for the 316 specific commands follows:

022740          uram unext

        pdt.more:
022740          preg -> temp
022741          if zero pdt.hreg -> upc

022742          ifnot zero g0.temp - ascii.A LS
022743          areg -> temp
022744          if zero pdt.hreg -> upc

022745          ifnot zero g0.temp - ascii.B LS
022746          breg -> temp
022747          if zero pdt.hreg -> upc

022750          ifnot zero g0.temp - ascii.X LS
022751          if zero pdt.hreg -> upc
022752          xreg -> temp

022753          ifnot zero g0.temp - ascii.N ls
022754          nreg -> temp
022755          if zero pdt.hreg -> upc

022756          ifnot zero g0.temp - ascii.S ls
022757          sreg -> temp
022760          if zero pdt.hreg -> upc

022761          ifnot zero g0.temp - ascii.W ls
022762          if zero pdt.watch -> upc

022763          ifnot zero pdt.quest->upc ;type "?" and re-enter uDDT
022764           nop

        ;Print one of the emulation registers.

        pdt.hreg:
022765          l5.flags & f.nnull LS
022766          ifnot zero L11.num1 - temp LS ;If any argument,
022767          ifnot zero pdt.quest -> upc ; it must be the register address
022770          l5.flags & ~f.mode -> l5.flags ;  clear out the command
022771          l5.flags ! ascii.R -> l5.flags ;   and put in R.
022772          if zero pdt.R -> upc
022773           temp -> l6.point

        pdt.watch:
022774          l5.flags & f.nnull ls
022775          if zero watch.off -> upc ;if no arg, clear watching
022776          ifnot zero l11.num1 -> temp
022777          rb.softint -> base      ;set up the watch registers
023000          base -> l0.base
```

```
023001          temp -> 14.wmar
023002          watch.go -> upc         ;enable watching, set upL6.WMBR
023003          (.+1) -> 11.ret1
023004          pdt.crprmt -> upc
023005          10.base -> base

        ;Subroutine WATCH.GO, call with BASE=SOFTINT, will set up watch soft
        ;if machine is running.

        watch.go:
023006          14.wmar -> mar(r)
023007          g10.state & sst.run ls  ;[1] is this running?
023010          if zero 11.ret1 -> upc  ;[2] no, skip it
023011          mbr -> 16.wmbr          ;yes, record current value
023012          11.ret1 -> upc          ;start returning
023013          ifnot zero g14.misc ! m.progint -> g14.misc, misc ;set interr

        watch.off:
023014          rb.softint -> base      ;switch bases
023015          base -> 10.base
023016          0 -> 14.wmar            ;disable watching
023017          pdt.crprmt -> upc
023020          10.base -> base
023021  unext==.

        # 18 "m.mic"
```

```
# 1 "crash.mic"
```

```
        ;Define where the crash areas are in macro memory:

 003400 crash.system = 3400
 003420 crash.state  = 3420
 003440 crash.main   = 3440
 003460 crash.clock  = 3460
 003500 crash.ioos   = 3500
 003520 crash.xxx    = 3520
 003540 crash.soft   = 3540

        ;Here on crashes, save machine state

 023021         uram unext

        crash:
 023021  save.state -> upc              ;first, set up RB.STATE
 023022  (.+1) -> g0.temp

 023023  crash.state -> mar             ;set up the MAR
 023024  rb.state -> base               ;and BASE
 023025  write.rb -> upc                ;write it
 023026  (.+1) -> g0.temp               ;return through G0

 023027  crash.main -> mar             ;set up the MAR
 023030  rb.main -> base               ;and BASE
 023031  write.rb -> upc                ;write it
 023032  (.+1) -> g0.temp               ;return through G0

 023033  crash.clock -> mar             ;set up the MAR
 023034  rb.clock -> base               ;and BASE
 023035  write.rb -> upc                ;write it
 023036  (.+1) -> g0.temp               ;return through G0

 023037  crash.ioos -> mar             ;set up the MAR
 023040  rb.ioos -> base               ;and BASE
 023041  write.rb -> upc                ;write it
 023042  (.+1) -> g0.temp               ;return through G0

 023043  crash.soft -> mar
 023044  rb.softint -> base
 023045  write.rb -> upc
 023046  (.+1) -> g0.temp

 023047  crash.xxx -> mar               ;write to XXX area
 023050  rb.state -> base               ;get to saved state
 023051  10.base -> base                ;then to that base
 023052  write.rb -> upc
 023053  (.+1) -> g0.temp

 023054  rb.state -> base
 023055  15.temp - 1 ls

 023056  if zero ascii.h. -> temp       ;if machine halted, print an "h"
 023057  pdt.load -> upc                 ;always try for a load
 023060   ifnot zero ascii.t. ->  temp   ;if trapped print a "t" on the conso
```

```
        ;Subrcutine WRITE.RB, level G0.TEMP

        ;Writes the 16-word register block located at BASE into the memory lo
        ;at MAR.

        write.rb:
 023061   20 -> g1.temp                    ;get a counter
 023062   1 -> g3.rone                     ;just in case it got smashed
        write.rb1:
 023063   10 -> mbr                        ;fetch a word
 023064   mar -> mar(w)
 023065   g1.temp - 1 -> g1.temp ls        ;[1] count this one
 023066   g3.rone + base -> base           ;[2] step to next
 023067   ifnot zero write.rb1 -> upc
 023070   g3.rone + mar -> mar             ;next word, please
 023071   g0.temp -> upc                   ;else return
 023072   nop

 023073  unext == .

        # 19 "m.mic"
```

```
# 1 "pm.mic"
```

```
;Native-Mode C/30 Process Manager

;This file contains code to run the new Process Mechanism of the
;Native-Mode C/30. This mechanism is closely coupled to the multi-
;buffered IOCB logic ("New IO") contained in "IO.MIC".

;Edit History
;
; Who      When                  What
;
; EAH     9-Feb-82        Made IO.GOAD tell ENQ.TIMING when it goaded T
;                         Also made ENQ.PENDING clear REPOKED and RETIM
;
; EAH    16-Apr-82        Re-designed process rescheduling algorithm an
;                         about context switching".  Also made TPR (TDV
;                         only one form (no TDV(0), no TDV other proces
;                         Added goad queue.  Made DPR(self) illegal.
;
; EAH    23-May-82        Removed KREG (R3) and made B (L4.B in R8.MAIN
;                         saved instead.  Shuffled R1 & R2 down by one
;                         PCB to make room for B.  Added FFO, BLT and C
;                         Removed extra code from SCAN.SCAN
;
; ADO    4 June 82        changed for new config scheme, removed NMFS (
;                         entry point from drivers, and definition of t
;                         of the entry point from this module.  Changed
;                         dispatch (at io.dd.call) and all tables in de
;                         drivers to use one word entries.
;
; ADO    11 Aug 82        Fixed bug at clock.again: whereby an entry no
;                         removed from the timer queue by the next cloc
;                         interrupt caused no processes to recieve thei
;                         goads for a long time.  Now the code skips ov
;                         these entries.
```

```
;All about context switching...

;For each of the 32 pending heaps, there is a bit in an attention (AT
;word. L10.ATTN1 holds the most significant 16 bits, L11.ATTN2, the l
;The position of the most significant "1" in these words identifies t
;process which should be run asap; we always run highest priority fir
;Finding this hi-order process and context switching to it is fairly
;costly - far too costly to do on every instruction.

;There a few cases where we must check for a possible context switch:
; a. Whenever we call ENQ.PENDING (it might be higher priority than c
; b. Whenever we do an SPR
; c. Whenever we do an ENB (things might be waiting to run at higher

;For instruction-level operations, cases A,B,C work just fine.  When
;I/O completes (IODD.IPUT) or processes timeout (IO.CLOCK), things ge
;more complex because the corresponding GOAD occurs at interrupt leve
;could interfere with the operation of an instruction such as SPR, et
;To solve this problem, we set L7.PMSCAN to +1 tp force a GOAD and +2
;force an RMQ TIMING / ENQ PENDING.
```

```
000000  dcb.q.forw == 0                 ;chain pointer
000001  dcb.q.back == 1                 ;backward chain pointer
000002  dcb.q.hedr == 2                 ;header pointer

000003  dcb.priority == 3               ;priority of this DCB
000003  dcb.state == 3                  ;shares space with DCB state

000004  dcb.goad == 4                   ;goad pointer
000005  dcb.type == 5                   ;device type (if any) associated with
000006  dcb.pc == 6                     ;macro context PC
000007  dcb.a == 7                      ;macro context A
00001C  dcb.x == 10                     ;macro context X
000011  dcb.f == 11                     ;macro context flags (carry bit)
000012  dcb.b == 12                     ;macro context B
000013  dcb.sp == 13                    ;macro context SP
000014  dcb.r1 == 14                    ;macro r1,r2
000015  dcb.r2 == 15
000016  dcb.time == 16                  ;wakeup time if on TIMING heap
000017  dcb.runh == 17                  ;hi order runtime for process
00002C  dcb.runl == 20                  ;lo order runtime
000021  dcb.rb == 21                    ;RB for the device
000022  dcb.iocb == 22                  ;current IOCB being used by DD
000023  dcb.get == 23                   ;ptr to GET queue header
000024  dcb.put == 24                   ;ptr to PUT queue header
```

```
                ;Definitions in DCB.TYPE

000077  type.all == 77                    ;device type
000000  type.software == 0                ;software device (must be zero)
                                          ;see include file of IO.DEFS for more

                ;definitions in DCB.RB

001777  rb.all == 1777

                ;Bits in DCB.PRIORITY/DCB.STATE

100000  state.idle == 100000              ;this DCB is in IDLE state
040000  state.ready == 040000             ;etc
020000  state.timing == 020000
010000  state.pending == 010000
004000  state.retimed == 004000
002000  state.repcked == 002000
170000  state.all = state.idle!state.ready!state.timing!state.pending

001000  state.skip == 001000              ;cause this to skip eventually
000400  state.regcad == 000400            ;copied into repoked eventually

000037  priority.all == 37
000000  priority.0 == 0
```

```
        ;Call offsets into the Micro device drivers. Each is offset by 2 word
        ;to allow for UPC settling time at the top of the driver itself.

000000  dd.adv.offset == 0              ;here on ADV's
000001  dd.ddv.offset == 1              ;here on DDV's
000002  dd.act.offset == 2              ;here on ACT's
000003  dd.hpk.offset == 3              ;here on HPK's
```

```
        ;Register block defintiions

000300  rb.ioos == 300                  ;IO Operating System base
000000   10.penh == 10                  ;macrocode pending header table addre
000003   13.ret3 == 13                  ;level 3 return
000004   14.rdyh == 14                  ;ready header address
000005   15.timh == 15                  ;timing header address
000006   16.dcb  == 16                  ;always contains a DCB pointer
000007   17.ddupc == 17                 ;used to hold DD's UPC
000010   110.attn1 == 110               ;most significant 16 bits of system a
000011   111.attn2 == 111               ;least significant 16 bits thereof
000012   112.sgq == 112                 ;start of goad queue
000013   113.egq == 113                 ;end of goad queue
000014   114.dirty == 114               ;someone has altered the state of the
000015   115.time == 115                ;time of day in 1.6ms ticks
000016   116.ddbase == 116              ;used to hold DD's BASE
000017   117.idlh == 117                ;idle header address

000250  rb.tid == 250                  ;timing insertion/deletion (share w/
000000   10.tinext == 10                ;next one to be examined in heap
000004   14.titime == 14                ;time of one being inserted
000005   15.tithis == 15                ;address of one being inserted

000260  rb.bits == 260                 ;table of bits
000260   regs rb.bits
000260   exp first1
000261   exp second1
000262   exp third1
000263   exp fourth1
000264   exp fifth1
000265   exp sixth1
000266   exp seventh1
000267   exp eighth1
000270   exp nineth1
000271   exp tenth1
000272   exp eleventh1
000273   exp twelfth1
000274   exp thirteenth1
000275   exp fourteenth1
000276   exp fifteenth1
000277   exp sixteenth1
```

```
                ;RB.DEV, Device Driver Dispatch (temporary &&)


000010  iowhat == 10                    ; dispatch to find device driver entr
001600   dispatch 1600                   ; looks in dispatch memory starting h

                dsp.dev:                ; (this magic courtesy of the mirdb)
001600   exp nodev                       ;(0) null

001602   exp cm1i.0,cm1o.0      ;(1,2) cheap MII 2651, input, output
001604   exp i18.0,o18.0        ;(3,4) MII 1822, input and output

001605   exp icac.0                     ;(5) MBM Console/Cassette input
001606   exp ocac.0                     ;(6) MBM Console/Cassette output

001610   exp m52i.0,m52o.0        ;(7,8) MTI 2652, input and output
001612   exp i18.0,o18.0        ;(9,10) MTI 1822, input and output

001614   exp m52i.0,m52o.0        ;(11,12) MSYNC 2652, input and output
001616   exp nodev,nodev    ;(13,14) Simp input and output

000016  dev.max == (.-dsp.dev-1)          ;highest allowed device type

023073          uram unext
                nodev:
023073          baddev -> upc
023074          baddev -> upc
023075          baddev -> upc
023076          baddev -> upc

                baddev:
023077          crash -> upc
023100          ioerr.idt -> temp

023101          unext == .
```

```
        ;Error codes returned to crash. Comments ending in "?" signify
        ;internal errors which should never happen. Other can be induced by
        ;broken macrocode.

000030  ioerr.0 == 30                    ;base of error numbers

000030  ioerr.dcb0   == ioerr.0 + 0      ;DCB at zero
000031  ioerr.advni  == ioerr.0 + 1      ;ADV DCB not on IDLE heap
000032  ioerr.acti   == ioerr.0 + 2      ;ACT DCB on IDLE heap

000033  ioerr.rmqnis == ioerr.0 + 3      ;RMQ.IDLE not in IDLE state
000034  ioerr.rmqniq == ioerr.0 + 4      ;RMQ.IDLE not in IDLE queue
000035  ioerr.rmqnrs == ioerr.0 + 5      ;RMQ.READY not in IDLE state
000036  ioerr.rmqnrq == ioerr.0 + 6      ;RMQ.READY not in IDLE queue
000037  ioerr.rmqnts == ioerr.0 + 7      ;RMQ.TIMING not in TIMING state
000040  ioerr.rmqntq == ioerr.0 + 10     ;RMQ.TIMING not in TIMING queue
000041  ioerr.rmqnps == ioerr.0 + 11     ;RMQ.PENDING not in PENDING state
000042  ioerr.rmqnpq == ioerr.0 + 12     ;RMQ.PENDING not in PENDING queue

000043  ioerr.an1    == ioerr.0 + 13     ;RMQ.PENDING didn't see attn bit=1
000044  ioerr.an0    == ioerr.0 + 14     ;ENQ.PENDING didn't see attn bit=0
000045  ioerr.pqe    == ioerr.0 + 15     ;PENDING QUEUE empty but attn bit=1

000046  ioerr.idt    == ioerr.0 + 16     ;illegal value in DCB.TYPE
000047  ioerr.ntr    == ioerr.0 + 17     ;nothing to run
000050  ioerr.agi    == ioerr.0 + 20     ;attempt to goad idle dcb
000051  ioerr.ddvi   == ioerr.0 + 21     ;attempt to DDV yourself while INHibi
000052  ioerr.eiq    == ioerr.0 + 22     ;empty idle queue at start-up
000053  ioerr.rbmu   == ioerr.0 + 23     ;DCB.RB pointing to wrong place
000054  ioerr.bxa1   == ioerr.0 + 24     ;Bad AC arg to XDV
000055  ioerr.bxa2   == ioerr.0 + 25     ;Bad B-reg arg to XDV
000056  ioerr.ilts   == ioerr.0 + 26     ;illegal IOCB.SIZE (transfer size)
000057  ioerr.tcwn   == ioerr.0 + 27     ;transfer count went negative ?
000060  ioerr.ddiia  == ioerr.0 + 30     ;DD saw illegal IOCB address
000061  ioerr.get2   == ioerr.0 + 31     ;DD attempted to get 2 current IOCBs
000062  ioerr.put0   == ioerr.0 + 32     ;DD attempted to put with no current
000063  ioerr.csii   == ioerr.0 + 33     ;DD couldn't shake last IOCB during D
000064  ioerr.imbdf  == ioerr.0 + 34     ;DD wants input side APR'ed to APR ou
000065  ioerr.rfii   == ioerr.0 + 35     ;attempt to RFI while INHibited
000066  ioerr.uint   == ioerr.0 + 36     ;DD saw unexpected interrupt (see BAS
000067  ioerr.noff   == ioerr.0 + 37     ;can't do this with NMFS off
000070  ioerr.ddvs   == ioerr.0 + 40     ;can't DDV yourself
000071  ioerr.aprwh  == ioerr.0 + 41     ;apr of wrong half first


000130  ioerr.sat    == ioerr.0 + 100    ;first sat ucode error code
000167  ioerr.sat.max== ioerr.0 + 137    ;last sat ucode error code
023101          uram unext
```

```
        ;PM.CLEAR, Subroutine, level 2

        ;Called from START when we want to master clear the process manager.

        pm.clear:
023101   0 -> 15.timh
023102   12.ret2 -> upc                 ;return
023103   0 -> 16.curr                   ;say no current process
```

```
        ;NMFS instruction.

        ;Enter with L3.A pointing to start of queue header area for NMFS. We
        ;an IO.CLEAR, etc. If A=0, we turn NMFS off, else, we start it schedu

        io316.nmfs:
 023104  rb.iocs -> base                    ;set up base
 023105  io.clear -> upc                    ;clear the IO world
 023106  (.+1) -> l2.ret2

        ;now set up current state to be nil

 023107  0 -> l10.attn1                     ;no pending processes, please
 023110  0 -> l11.attn2
 023111  0 -> l12.sgq                       ;nothing on goad queue
 023112  0 -> l13.egq
 023113  0 -> l15.time                      ;start at time zero

 023114  rb.main -> base
 023115  0 -> l6.curr
 023116  l10.flags ! 100000 -> l10.flags ;become inhibited

 023117  l3.a -> mbr ls                     ;make sure we like it, too
 023120  if zero l0.pc + 1 -> l0.pc, mar(r) ;if it's zero, emulate the next
 023121  if zero ints -> upc                ;[1] instruction in-line
 023122  ifnot zero l10.flags & ~100000 -> l10.flags ;[2] if this is a turn-o

 023123  4 -> g0.temp                       ;a useful constant for below
 023124  rb.iocs -> base                    ;fix base
 023125  mbr -> l17.idlh                    ;set up idle pointer (before mbr chan
 023126  g0.temp + mbr -> mbr
 023127  mbr -> l4.rdyh
 023130  g0.temp + mbr -> mbr
 023131  mbr -> l5.timh
 023132  g0.temp + mbr -> mbr

        ;Now find the top thing on the IDLE heap and move it to it's own PEND
        ;heap. We could call GOAD, but it bumps the PC, etc.

 023133  l17.idlh + q.forw -> mar(r)    ;read the addr of the top of the IDLE
 023134  mbr -> l0.penh                  ;[1] (base of pending headers, actual
 023135  nop                             ;[2]
 023136  l17.idlh - mbr ls               ;is it empty?
 023137  if zero crash -> upc            ;if so, illegal initialization
 023140  if zero ioerr.eiq -> temp       ; ++ empty idle queue

 023141  mbr -> l6.dcb
 023142  l6.dcb + dcb.state -> mar(r)    ;clean up the PCB
 023143  priority.all -> g2.temp         ;[1]
 023144  g2.temp ! state.idle -> g2.temp ;[2]
 023145  g2.temp & mbr -> mbr
 023146  l6.dcb + dcb.state -> mar(w)
 023147  nop                             ;[1]
 023150  nop                             ;[2]
 023151  0 -> mbr
 023152  l6.dcb + dcb.goad -> mar(w)     ;make sure it's off the gaod queue
 023153  io.rmq.idle -> upc              ;[1] remove the DCB from the IDLE que
```

```
023154    (.+1) -> 12.ret2              ;[2]
023155    io.enq.ready -> upc           ;put it on the ready queue
023156    (.+1) -> 12.ret2
023157    io.goad -> upc                ;and goad it (non-skip)
023160    dummy -> 11.ret1              ;eventually ends up in SOFT

          dummy:                        ;here from all over with softint pend
023161    ints -> upc                   ;do all I/O, then do softint
023162    rb.main -> base               ;dummy instruction cause of SOFTINT:
```

```
        ;INH and ENB instructions

        ;These instructions merely set and clear the sign of L10.flags in RB.
        ;prevent the scheduler from changing contexts.

        io316.inh:
 023163  rb.ioos -> base                  ;clear dirty flag
 023164  0 -> l14.dirty
 023165  rb.main -> base
 023166  l0.pc + 1 -> l0.pc, mar(r)       ;next instruction, please
 023167  ints -> upc                      ;[1]
 023170  l10.flags ! 100000 -> l10.flags ;[2]

        io316.enb:
 023171  l6.curr ls                       ;can't ENB unless there's a process
 023172  if zero crash -> upc
 023173  if zero ioerr.noff -> temp       ; ++ attempt to ENB with NMFS off

 023174  l10.flags ls                     ;going from INH to ENB?
 023175  l0.pc + 1 -> l0.pc, mar(r)
 023176  ifnot neg ints -> upc            ;[1] no, from ENB to ENB, NOP
 023177  ifnot zero l10.flags & ~100000 -> l10.flags ;[2] do the ENB
 023200  rb.ioos -> base                  ;check to see if DIRTY set
 023201  l14.dirty ls
 023202  if zero ints -> upc              ;hasn't been dirtied since INH
 023203  rb.main -> base
 023204  int.scan  -> upc                 ;force a scan soon
 023205  dummy -> l1.ret1                 ;and eventually end up at SOFT
```

```
        ;ADV instruction

        ;Enter with L5.X pointing to DCB to be moved from IDLE to READY
        ;heap. Verify that the DCB is in fact on the IDLE queue and that
        ;its state is IDLE. Also call the device-driver (DD) at its ADV
        ;entry point to perform device-specific functions.

        io316.adv:
023206   l6.curr ls                         ;is NMFS on?
023207   if zero crash -> upc
023210   if zero ioerr.noff -> temp
023211   l5.x -> temp                       ;get DCB address into our locals
023212   rb.ioos -> base                    ;which live here
023213   temp -> l6.dcb

023214   l6.dcb + dcb.state -> mar(r)       ;clean up the PCB
023215   priority.all -> g2.temp            ;[1]
023216   g2.temp ! state.idle -> g2.temp    ;[2]
023217   g2.temp & mbr -> mbr
023220   l6.dcb + dcb.state -> mar(w)
023221   nop                                ;[1]
023222   nop                                ;[2]
023223   0 -> mbr
023224   l6.dcb + dcb.goad -> mar(w)        ;make sure it's off the gaod queue
023225   io.rmq.idle -> upc                 ;[1] remove the DCB from the IDLE que
023226   (.+1) -> l2.ret2                   ;[2]
023227   io.enq.ready -> upc                ;and put it on the ready queue
023230   (.+1) -> l2.ret2

023231   dd.adv.offset -> temp              ;call the device driver at this entry
023232   io.dd.call -> upc
023233   fetch.base -> l2.ret2
```

```
        ;DDV Instruction

        ;Enter with L5.X pointing to DCB to be de-activated. This is one
        ;of the cases where we don't know what heap the DCB is on. For
        ;this reason, we must interrogate its STATE bits to deduce how to
        ;remove it. If the DCB is already de-activated, hardly any
        ;processing takes place (see below).

        ;One particularly troublesome point is DDV'ing your own process, whic
        ;we disallow.

        ;In any case, we move the DCB to the IDLE queue and call the
        ;device driver to clean up any hardware (or register block)
        ;related operations.

        io316.ddv:
023234   l6.curr ls                     ;is NMFS on?
023235   if zero crash -> upc
023236   if zero ioerr.noff -> temp
023237   l5.x -> temp                   ;put DCB address in more
023240   l6.curr - temp ls              ;outselves?
023241   if zero crash -> upc
023242   if zero ioerr.ddvs -> temp

023243   l5.x + dcb.state -> mar(r)     ;read the DCB's state
023244   rb.ioos -> base                ;[1] over to new base
023245   temp -> l6.dcb                 ;[2] copy L5.X (RB.MAIN) here

023246   mbr -> g0.temp                 ;need to do math
023247   g0.temp & state.timing ls      ;on timing heap (most likely)
023250   ifnot zero ddv.timing -> upc

023251   if zero g0.temp & state.pending ls ;[1] on a pending heap (next most
023252   ifnot zero ddv.pending -> upc

023253   if zero g0.temp & state.ready ls ;[1] on ready heap?
023254   ifnot zero ddv.ready -> upc

        ;The only heap it can possibly be on is the IDLE heap.

        ddv.idle:
023255   if zero io.rmq.idle -> upc     ;[1] remove it from idle
023256   (.+1) -> l2.ret2
023257   io.enq.idle -> upc             ;then put it back on (at end)
023260   fetch.base -> l2.ret2
```

```
        ;Here when the DCB was on the timing heap. Remove it, call the
        ;device driver to clean up and exit.

        ddv.timing:
023261   io.rmq.timing -> upc           ;remove it from the TIMING queue
023262   (.+1) -> 12.ret2
023263   io.enq.idle -> upc             ;place it on the IDLE queue
023264   (.+1) -> 12.ret2

        ddv.dd.call:
023265   dd.ddv.offset -> temp          ;call at DDV offset
023266   io.dd.call -> upc
023267   fetch.base -> 12.ret2

        ;Here when the DCB was on the READY heap. Performs similar
        ;actions as DDV.TIMING, except for the READY heap.

        ddv.ready:
023270   io.rmq.ready -> upc
023271   (.+1) -> 12.ret2
023272   io.enq.idle -> upc
023273   ddv.dd.call -> 12.ret2
```

```
        ;Here when the DCB was on a PENDING queue. Remove it from the
        ;pending queue, insert it on the IDLE queue, call the device
        ;driver to clean up and return.

        ddv.pending:
023274   io.rmq.pending -> upc          ;remove from pending queue
023275   (.+1) -> 12.ret2
023276   io.enq.idle -> upc             ;put on idle
023277   ddv.dd.call -> 12.ret2
```

```
        ;ACT Instruction

        ;Enter with L5.X pointing to DCB to ACT upon, L5.A and B having
        ;device-dependant values. After verifying that the DCB is not
        ;IDLE, we call the DD at its ACT entry. We copy L5.A into G0.TEMP
        ;for the device driver to read. This means that DD.CALL cannot
        ;trash G0.TEMP.

        io316.act:
023300   l3.a -> temp
023301   temp -> g0.temp                ;set up G0 to have user's accumulator
023302   dd.act.offset -> g6.temp       ;enter IO.ACT.HPK with ACT.OFFSET

        io.act.hpk:                     ;(enter here from HPK instruction, to
023303   l6.curr ls                     ;is NMFS on?
023304   if zero crash -> upc
023305   if zero ioerr.noff -> temp
023306   l5.x -> temp                   ;copy DCB address
023307   rb.icos -> base
023310   temp -> l6.dcb

023311   l6.dcb + dcb.state -> mar(r)   ;read the current state
023312   state.idle -> g2.temp          ;[1] get a nice mask
023313   g6.temp -> temp                ;[2] and copy caller's arg for IO.DD.
023314   g2.temp & mbr ls               ;is this DCB idle?
023315   ifnot zero crash -> upc        ;yes, trap
023316   ifnot zero ioerr.acti -> temp  ; ++ attempt to ACT on an idle device

023317   io.dd.call -> upc              ;(IO.DD.CALL checks for type 0)
023320   fetch.base -> l2.ret2

        ;HPK Instruction

        ;Similar to ACT in that the DCB specified in X must not be idle.
        ;Simply call the DD at its HPK entry to check for new conditions
        ;affecting the device IO.

        io316.hpk:
023321   io.act.hpk -> upc              ;enter common part
023322   dd.hpk.offset -> g6.temp       ;[1] with HPK.OFFSET as arg
```

```
        ;SPK Instruction

        ;SPK simply calls IO.GOADSKIP to cause the specified process to run.
        ;Enter with L5.X having DCB to goad.

        io316.spk:
 023323   l6.curr ls                     ;is NMFS on?
 023324   if zero crash -> upc
 023325   if zero ioerr.noff -> temp
 023326   l5.x -> temp                   ;copy X to usual place
 023327   l0.pc + 1 -> l0.pc
 023330   rb.ioos -> base
 023331   temp -> l6.dcb

 023332   io.goadskip -> upc             ;call GOAD, does INT.SCAN
 023333   dummy -> l1.ret1               ;eventually end up at SOFT
```

```
        ;TDV Instruction

        ;Here to insert the current DCB on the timing
        ;heap. A contains the maximum time the DCB should remain on the timin
        ;heap (the NEWIO document describes this in greater detail).

        ;the wake-up time is computed and written into DCB.TIME for processin
        ;later by RFI when the process debreaks and the DCB.TDV.PENDING
        ;bit is set.

        io316.tdv:
023334   16.curr -> temp ls              ;is NMFS on?
023335   if zero crash -> upc
023336   if zero ioerr.noff -> temp

023337   16.curr + dcb.state -> mar(r)   ;read DCB.STATE
023340   state.retimed -> g2.temp        ;[1] get retimed bit into useful plac
023341   13.a -> temp                    ;[2] set up for later
023342   mbr ! g2.temp -> mbr            ;set the bit
023343   16.curr + dcb.state -> mar(w)   ;write it back
023344   rb.icos -> base                 ;[1] get at time
023345   115.time + temp -> temp         ;[2] compute wakeup time from RB.MAIN
023346   rb.main -> base                 ;fix BASE
023347   temp -> g2.temp, mbr            ;copy to MBR for writing, below

023350   16.curr + dcb.time -> mar(w)    ;write it into DCB
023351   fetch -> upc                    ;[1]
023352   nop                             ;[2]
```

```
        ;RFI Instruction

        ;RFI does the following checks (in order):
        ; 1. If REPOKED it set, it clears it, "skips" and moves this DCB to e
        ; 2. If RETIMED is set, it clears it and moves to somewhere in TIMING
        ; 3. If neither is set, it moves to end of READY

        io316.rfi:
023353   110.flags ls                    ;better not be inhibited!
023354   16.curr + dcb.state -> mar(r)   ;look at the flags
023355   if neg crash -> upc             ;[1]
023356   if neg ioerr.rfii -> temp       ;[2]  ++ attempt to RFI while inhibit
023357   mbr -> g2.temp                  ;move flags over

023360   g2.temp & state.repoked ls      ;repoked?
023361   ifnot zero 10.pc + 1 -> 10.pc
023362   10.pc + 1 -> 10.pc, mbr         ;always update PC
023363   16.curr + dcb.pc -> mar(w)
023364   16.curr -> temp                 ;[1] copy over to NMFS
023365   rb.icos -> base                 ;[2]
023366   temp -> 16.dcb

023367   g2.temp & state.repoked ls      ;repoked?
023370   ifnot zero rfi.to.pending -> upc
023371   g2.temp & state.retimed ls
023372   ifnot zero rfi.to.timing -> upc ;was this one RETIMED?

        rfi.to.ready:                   ;(no one enters here)
023373   if zero io.rmq.pending -> upc  ;[2] move the process to the ready qu
023374   (.+1) -> 12.ret2
023375   io.enq.ready -> upc
023376   (.+1) -> 12.ret2
023377   int.scan -> upc                 ;do a scan soon
023400   dummy -> 11.ret1
```

```
        ;Here to queue this DCB back to the end of the pending queue.

        rfi.to.pending:
023401   io.rmq.pending -> upc          ;move it to the end
023402   (.+1) -> i2.ret2
023403   io.enq.pending -> upc
023404   (.+1) -> i2.ret2
023405   int.scan -> upc                ;[1] then force the scheduler to run
023406   dummy -> i1.ret1               ;[2] if it finds same, no problem, el

        ;Here when we are RFI'ing a process which has a pending TDV(n).
        ;call ENQ.TIMING to insert it on the right spot in the TIMING
        ;heap. ENQ.TIMING knows how to check the time window and possibly
        ;put it back on the PENDING queue again. See ENQ.TIMING for more
        ;info.

        rfi.to.timing:
023407   g2.temp & ~state.retimed -> mbr
023410   i6.dcb + dcb.state -> mar(w)   ;write it back
023411   io.rmq.pending -> upc          ;[1] and remove it
023412   (.+1) -> i2.ret2               ;[2]
023413   io.enq.timing -> upc
023414   (.+1) -> i2.ret2
023415   int.scan -> upc
023416   dummy -> i1.ret1               ;eventually end up at SOFT
```

```
        ;PCB instruction

        ;Returns pointer to current PCB (L6.CURR) or 0 if there isn't one (NM

        io316.pcb:
023417   16.curr -> mbr                 ;read current DCB address ...
023420   0 -> mar(w)
023421   fetch -> upc                   ;[1] done
023422   mbr -> 15.x                    ;[2] ... into X
```

```
        ;Subroutines IO.GOAD and IO.GOADSKIP, Level 1

        ;Adds L6.DCB to GOADQ (GOADSKIP sets "s" bit, too).  Also sets L14.DI

        io.goadskip:
023423   l6.dcb + dcb.state -> mar(r)    ;get current goad bits
023424   nop                             ;[1]
023425   state.skip -> g0.temp           ;[2] setup for OR'ing
023426   g0.temp ! mbr -> mbr
023427   l6.dcb + dcb.state -> mar(w)    ;turn on STATE.SKIP
023430   nop                             ;[1]
023431   nop                             ;[2]

        io.goad:
023432   l6.dcb + dcb.goad -> mar(r)     ;is this already on the goad queue?
023433   l6.dcb -> temp                  ;[1] for later
023434   -1 -> l14.dirty                 ;[2] set the dirty flag for INH/ENB t
023435   mbr ls
023436   ifnot zero l1.ret1 -> upc       ;if so, no need to do anything
023437   if zero -1 -> mbr               ;if not on queue, put at end
023440   l6.dcb + dcb.goad -> mar(w)
023441   l12.sgq ls                      ;[1] put on front, too?
023442   if zero temp -> l12.sgq         ;[2] yes
023443   if zero int.scan -> upc         ;and we're done
023444   if zero temp -> l13.egq         ;return through level 1
023445   l6.dcb -> mbr                   ;no, set link to this one in previous
023446   l13.egq + dcb.goad -> mar(w)
023447   int.scan -> upc                 ;[1] return through level 1
023450   temp -> l13.egq                 ;[2] and then fix end
```