Appendix C
The M-code Interpreter (in Modula-2)

The document in this section is to be regarded as the definition of the Lilith virtual machine as agreed upon by the compiler writers and the hardware team. Casual readers may find this document too concise and uncommented. There are other papers which may help one in digesting the contents of this document. They are:

'The Personal Computer Lilith' by N. Wirth, ETH report number 40

Chapter 3 of this manual

'The Lilith Architecture, its Design in view of Code Generation',
an unpublished paper by Christian Jakobi.

The M-code interpreter

N.Wirth, Ch.Jacobi, V.Viniger

last modification: facts: Feb.81; details: 4.1.82

The following Modula-2 Program is an extension of the Appendix 1 in "The Personal Computer Lilith".

Its purpose is to do docomentation of how we made the actual implementation on the Lilith. It is not used for introduction. It's use is more a reference what is microprogrammed exactly.

The undefined opcodes in the instruction set:

```
  21B, 334B:  reserved for use by the compiler
  214B, 215B: reserved for use for arithmetics

  237B second byte >3: reserved for floating arithmetics
  246B second byte  0: reserved for debugging new instructions
                  <>0: reserved for supporting special hardware, extensions
  247B second byte >5: reserved for operative system needs
```

Table of instructions

		0	20	40	60	80	A0	C0	E0
		0	40	100	140	200	240	300	340
0	0	LI0	LLV	LGV	LSV0	LSV	READ	FOR1	MOV
1	1	LI1	LLD	LGD	LSV1	LSD	WRITE	FOR2	CMP
2	2	LI2	LEV	LGV2	LSV2	LSD0	DSKR	ENTC	DDT
3	3	LI3	LED	LGV3	LSV3	LXFV	DSKW	EXC	REPL
4	4	LI4	LLV4	LGV4	LSV4	LSTA	SETRK	TRAP	BBLT
5	5	LI5	LLV5	LGV5	LSV5	LXB	UCHK	CHK	DCH
6	6	LI6	LLV6	LGV6	LSV6	LXV	ESC	CHKZ	UNPK
7	7	LI7	LLV7	LGV7	LSV7	LXD	SYS	CHKS	PACK
8	10	LI8	LLV8	LGV8	LSV8	DADD	ENTP	EQL	GB
9	11	LI9	LLV9	LGV9	LSV9	DSUB	EXP	NEQ	GB1
A	12	LI10	LLV10	LGV10	LSV10	DMUL	ULSS	LSS	ALOC
B	13	LI11	LLV11	LGV11	LSV11	DDIV	ULEQ	LEQ	ENTR
C	14	LI12	LLV12	LGV12	LSV12		UGTR	GTR	RTN
D	15	LI13	LLV13	LGV13	LSV13		UGEQ	GEQ	CX
E	16	LI14	LLV14	LGV14	LSV14	DSHL	TRA	ABS	CI
F	17	LI15	LLV15	LGV15	LSV15	DSHR	RDS	NEG	CF
10	20	LIB	SLV	SGV	SSV0	SSV	LODFV	OR	CL
11	21		SLD	SGD	SSV1	SSD	LODFD	XOR	CL1
12	22	LIV	SEV	SGV2	SSV2	SSD0	STORE	AND	CL2
13	23	LID	SED	SGV3	SSV3	SXFV	STOFV	COM	CL3
14	24	LLA	SLV4	SGV4	SSV4	TS	STOT	IN	CL4
15	25	LGA	SLV5	SGV5	SSV5	SXB	COPT	LIN	CL5
16	26	LSA	SLV6	SGV6	SSV6	SXV	DECS	MSK	CL6
17	27	LEA	SLV7	SGV7	SSV7	SXD	PCOP	NOT	CL7
18	30	JPC	SLV8	SGV8	SSV8	FADD	UADD	ADD	CL8
19	31	JP	SLV9	SGV9	SSV9	FSUB	USUB	SUB	CL9
1A	32	JPFC	SLV10	SGV10	SSV10	FMUL	UMUL	MUL	CL11
1B	33	JPF	SLV11	SGV11	SSV11	FDIV	UDIV	DIV	CL10
1C	34	JPBC	SLV12	SGV12	SSV12	FCMP	UMOD		CL12
1D	35	JPB	SLV13	SGV13	SSV13	FABS	ROR	BIT	CL13
1E	36	ORJP	SLV14	SGV14	SSV14	FNEG	SHL	NOP	CL14
1F	37	ANDJP	SLV15	SGV15	SSV15	FFCT	SHR	MOVF	CL15

Fixed addresses

dec	oct	hex	
0	0	0	(F-register of module 0 [software])
1	1	1	(initialization flag of module 0 [software])
2	2	2	(string pointer of module 0 [software])
3	3	3	device mask
4	4	4	P-register
5	5	5	saved P register
6	6	6	boot flag

```
...               7 free locations
 14   16    E     trap vector
 16   20   10     interrupt vector for line 8 (clock)
 18   22   12     interrupt vector for line 9 (disk)
...
 30   36   1E     interrupt vector for line 15
 32   40   20     data frame table
                  (96? entries; firmware allows 256; compiler allows 224)
                  free area
*)


MODULE Interpreter;

  FROM SYSTEM IMPORT WORD;

  TYPE
    Request = [0..15];
    Mask    = BITSET; (* 8..15 correspond to interrupt lines, 7 to the trap *)

  CONST
      NILL       = 177777B;        (*NIL*)
      devstatadr =  3;             (*devicestatus address*)
      tlc        = 16B;            (*trap location adr*)
      dft        = 40B;            (*data frame table adr*)

      (*Trap Error Numbers*)
      (*------------------*)

      end        =  0;  (*end*)
      instrChk   =  1;  (*illegal instruction*)
      prioChk    =  2;  (*priority error*)
      storageChk =  3;  (*storage overflow*)
      rangeChk   =  4;  (*range violation*)
      addrChk    =  5;  (*NIL access or invalid computed address*)
      realOvfl   =  6;  (*floating point overflow*)
      cardOvfl   =  7;  (*cardinal overflow (maskable)*)
      intOvfl    =  8;  (*integer overflow  (maskable)*)
   (* software assignments:
      funcErr    =  9;  (*function return error [software]*)
      halt       = 10;  (*halt called [software]*)
      assertErr  = 11;  (*assertion error [software]*)
    *)


  VAR
  (* the following global variables represent registers in hardware *)
      PC:  CARDINAL;                (*program counter*)
      IR:  CARDINAL;                (*instruction register*)
      F:   CARDINAL;                (*code frame base address*)
      G:   CARDINAL;                (*data frame base address*)
      H:   CARDINAL;                (*stack limit address*)
      L:   CARDINAL;                (*local segment base address*)
      S:   CARDINAL;                (*stack pointer*)
      P:   CARDINAL;                (*process base address*)
      M:   Mask;                    (*software priority mask*)
      MSK: Mask;                    (*hardware interrupt mask*)
      REQ:  BOOLEAN;                (*interrupt request (bit in condition code)*)
      ReqNo: Request;               (*request number, 8..15*)
      overflow: BOOLEAN;            (*overflow (bit in condition code)*)

    (* auxiliary variables used over single instructions only *)
      i, j, k, lsz, la[illegible]r, lup, llow: CARDINAL;
      fromea, toea: CARDINAL; (*framepointers*)
      b:       BOOLEAN;
      lm:      BITSET;
      sb, db, sbmd, dbmd, fo, x, y: CARDINAL;  (*used in the display handling instrucions*)
      f1, f2: REAL;

    (* data store*)
      stk[0]:  ARRAY [0..177777B] OF CARDINAL;


  MODULE InstructionFetch;
    IMPORT F, PC;
    EXPORT next, next2;
```

```
  VAR code[0]: ARRAY [0..777777B] OF [0..255]; (*2**18 bytes*)
             (*code[0]..code[377777B] shares the memory with stk*)

  PROCEDURE next(): CARDINAL;
  BEGIN
    INC(PC);
    RETURN code[4*F+PC-1]
  END next;

  PROCEDURE next2(): CARDINAL; (*get next two code bytes*)
  BEGIN
     INC(PC, 2);
     RETURN code[4*F+PC-2]*400B + code[4*F+PC-1]
  END next2;

END InstructionFetch;


MODULE ExpressionStack;
  EXPORT push, pop, Dpush, Dpop, empty,
         expStackSize;

  CONST expStackSize = 16;
  VAR sp: CARDINAL;                              (*expression stack pointer*)
      a: ARRAY [0..expStackSize-1] OF CARDINAL;  (*expression stack*)
         (* the array x is represented in the hardware as
            a fast LIFO memory *)

  PROCEDURE push(w: CARDINAL);
  BEGIN a[sp] := w; INC(sp)
  END push;

  PROCEDURE pop(): CARDINAL;
  BEGIN DEC(sp); RETURN(a[sp])
  END pop;

  PROCEDURE empty():BOOLEAN;
  BEGIN RETURN sp=0
  END empty;

  PROCEDURE Dpush(d: REAL);
  BEGIN a[sp] := high(d); INC(sp); a[sp] := low(d); INC(sp)
  END Dpush;

  PROCEDURE Dpop(): REAL;
  BEGIN DEC(sp,2); RETURN pair(a[sp], a[sp+1])
  END Dpop;

BEGIN sp := 0;
END ExpressionStack;


PROCEDURE mark(x: CARDINAL; external: BOOLEAN);       (*sets a stack mark*)
  VAR i: CARDINAL;
BEGIN i := S;
  stk[S] := x;   INC(S);                  (*static link*)
  stk[S] := L;   INC(S);                  (*dynamic link*)
  IF external THEN stk[S] := PC+100000B   (*return address and external flag*)
  ELSE stk[S] := PC
  END;
    INC(S,2);                             (*reseved field for interrupt mask*)
  L := i
END mark;

PROCEDURE saveRegs;
BEGIN
  saveExpStack;
  stk[P  ] := G;  stk[P+1] := L;
  stk[P+2] := PC; stk[P+3] := CARDINAL(M);
  stk[P+4] := S;  stk[P+5] := H+24;
  (* stk[P+6] is reserved for error code *)
  (* stk[P+7] is reserved for error trap mask *)
END saveRegs;

PROCEDURE restoreRegs(changeMask: BOOLEAN);
BEGIN
```

```
    G :- stk[P];    F  :- stk[G];
    L :- stk[P+1];  PC :- stk[P+2];
    IF changeMask THEN
      M :- Mask(stk[P+3]);
      MSK :- M+Mask(stk[devstatadr])
    END;
    S :- stk[P+4];  H :- stk[P+5]-24;
    restoreExpStack;
  END restoreRegs;

  PROCEDURE Transfer(changeMask: BOOLEAN; to, from: CARDINAL);
    VAR j: CARDINAL;
  BEGIN
    j :- stk[to]; saveRegs; stk[from] :- P;
    P :- j; restoreRegs(changeMask);
  END Transfer;

  PROCEDURE Trap(n: CARDINAL);
  BEGIN
    (* INTEGER, CARDINAL overflow is maskable, but the masking
       need not be programed realy in the procedure trap *)
    IF NOT (n IN ( Mask(stk[P+7])-  (Mask({0..15})-Mask({cardOvfl,intOvfl}))  )) THEN
      stk[P+6] :- n;
      IF 7 IN Mask(stk[devstatadr]) THEN
        LOOP (* ! *) END
      END;
      INCL(stk[devstatadr], 7);
      Transfer(TRUE, tlc, tlc+1);
    ELSE (*the trap has been masked*)
           (*the value for the expressionstackpointer must be correct*)
  END;
  END Trap;

  PROCEDURE saveExpStack;
    VAR c: CARDINAL;
  BEGIN
    c :- 0; (*expression stack counter*)
    WHILE NOT empty() DO
      stk[S] :- pop(); INC(S); INC(c);
    END;
    stk[S] :- c; INC(S);
  END saveExpStack;

  PROCEDURE restoreExpStack;
    VAR c: CARDINAL;
  BEGIN
    DEC(S); c :- stk[S];
    WHILE c>0 DO
      DEC(c); DEC(S); push(stk[S])
    END
  END restoreExpStack;

  PROCEDURE get(N: CARDINAL): CARDINAL;
  BEGIN
    (* i :- input from channel N *)
    RETURN i
  END get;

  PROCEDURE put(x, N: CARDINAL);
  BEGIN
    (* output x to channel N *)
  END put;

  PROCEDURE readBootFile;
    VAR bootflag: CARDINAL;
  BEGIN
    (* read the boot file according to key and set bootflag *)
    stk[6] :- bootflag
  END readBootFile;

BEGIN (*main*)
  stk[5] :- P;          (*allows debugging*)
  readBootFile;
  P :- stk[4]; restoreRegs(TRUE);
  LOOP
    IF REQ THEN
      (*preferable for software: INCL(stk[devstatadr], ReqNo);*)
```

```
      Transfer(TRUE,2*ReqNo, 2*ReqNo+1);
    END;
   IR := next();
   CASE IR OF
     0B .. 17B: (*LI0 - LI15 load immediate*)  push(IR MOD 16) |

     20B: (*LIB  load immediate byte*)  push(next()) |

     21B: (*reserved for future instruction as needed by the compiler*)
          Trap(instrChk) |

     22B: (*LIW  load immediate word*)  push(next2()) |

     23B: (*LID  load immediate double word*)
          push(next2()); push(next2()) |

     24B: (*LLA  load local address*)   push(L+next()) |

     25B: (*LGA  load global address*)  push(G+next()) |

     26B: (*LSA  load stack address*)
          push(pop()+next());
          IF overflow THEN Trap(addrChk) END |

     27B: (*LEA  load external address*) push(stk[dft+next()]+next()) |

     30B: (*JPC  jump conditional*)
          IF pop() = 0 THEN PC := PC + next2() ELSE INC(PC,2) END  |

     31B: (*JP   jump*)  PC := PC + next2() |

     32B: (*JPFC  jump forward conditional*)
          IF pop() = 0 THEN PC := PC + next() ELSE INC(PC) END  |

     33B: (*JPF  jump forward*)   PC := PC + next()  |

     34B: (*JPBC  jump backward conditional*)
          IF pop() = 0 THEN PC := PC - next() ELSE INC(PC) END  |

     35B: (*JPB  jump backward*)  PC := PC - next()  |

     36B: (*ORJP  short circuit OR *)
          IF pop()=0 THEN INC(PC)
          ELSE push(1); PC := PC+next()
          END |

     37B: (*ANDJP  short circuit AND *)
          IF pop()=0 THEN push(0); PC := PC+next()
          ELSE INC(PC)
          END |

     40B: (*LLW  load local word*)     push(stk[L+next()])  |

     41B: (*LLD  load local double word*)
          i := L+next(); push(stk[i]); push(stk[i+1])  |

     42B: (*LEW  load external word*)
          push(stk[stk[dft+next()]+next()]) |

     43B: (*LED  load external double word *)
          i := stk[dft+next()]+next();
          push(stk[i]); push(stk[i+1]) |

     44B .. 57B: (*LLW4-LLW15*)        push(stk[L + (IR MOD 16)])  |

     60B: (*SLW  store local word*)    stk[L+next()] := pop() |

     61B: (*SLD  store local double word*)
          i := L+next(); stk[i+1] := pop(); stk[i] := pop()  |

     62B: (*SEW  store external word*)
          stk[stk[dft+next()]+next()] := pop() |

     63B: (*SED  store external double word *)
          i := stk[dft+next()]+next();
          stk[i+1] := pop(); stk[i] := pop() |
```

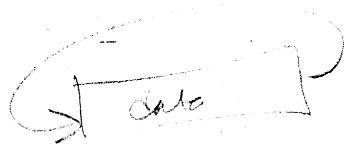
```
 64B .. 77B: (*SLW4-SLW15  store local word*)
      stk[L+(IR MOD 16)] := pop()  |

100B: (*LGW  load global word*)  push(stk[G+next()]) |

101B: (*LGD  load global double word*)
      i := next()+G; push(stk[i]); push(stk[i+1]) |

102B .. 117B: (*LGW2 - LGW15  load global word*)
      push(stk[G + (IR MOD 16)]) |

120B: (*SGW  store global word*)   stk[G+next()] := pop() |

121B: (*SGD  store global double word*)
      i := G+next(); stk[i+1] := pop(); stk[i] := pop()  |

122B .. 137B: (*SGW2 - SGW15  store global word*)
      stk[G + (IR MOD 16)] := pop() |

140B: (*LSW0  load stack addressed word*)
      k := pop();
      IF k=NILL THEN Trap(addrChk)
      ELSE
        push(stk[k])
      END |

141B .. 157B: (*LSW1 - LSW15  load stack addressed word*)
      push(stk[pop()+(IR MOD 16)]);
      IF overflow THEN Trap(addrChk) END  |

160B: (*SSW0  store stack-addressed word*)
      k := pop(); j := pop();
      IF j=NILL THEN Trap(addrChk)
      ELSE
        stk[j] := k
      END |

161B .. 177B:  (*SSW1 - SSW15  store stack-addressed word*)
      k := pop(); i := pop()+(IR MOD 16);
      IF overflow THEN Trap(addrChk)
      ELSE
        stk[i] := k
      END |

200B: (*LSW  load stack word*) i := pop()+next(); push(stk[i]);
      IF overflow THEN Trap(addrChk) END |

201B: (*LSD  load stack double word*)
      i := pop()+next(); push(stk[i]); push(stk[i+1]);
      IF overflow (*either addition*) THEN Trap(addrChk) END  |

203B: (*LXFW  load indexed frame word*)
      k := pop() + pop()*4 (*18 bits*);
      push(stk[k]);
      IF overflow(*18 bits*) THEN Trap(addrChk) END |

202B: (*LSD0  load stack double word*)
      i := pop(); push(stk[i]); push(stk[i+1]);
      IF (i=NILL) THEN Trap(storageChk) END |

204B: (*LSTA  load string address *)
      push(stk[G+2]+next()) |

205B: (*LXB  load indexed byte*)
      i := pop(); j := pop(); k := stk[j + (i DIV 2)];
      IF overflow THEN Trap(addrChk)
      ELSIF i MOD 2 = 0 THEN push(k DIV 400B)
                        ELSE push(k MOD 400B)
      END |

206B: (*LXW load indexed word*)
      i := pop()+pop(); push(stk[i]);
      IF overflow THEN Trap(addrChk) END |

207B: (*LXD  load indexed double word *)
      i := 2*pop()+pop();
      IF overflow(*addition or multiplication*) OR (i=NILL) THEN Trap(addrChk)
```

```
      ELSE
        push(stk[i]); push(stk[i+1])
      END |

   210B: (*DADD  double precision addition*)
         (* f1 := Pop2(); f2 := Pop2(); Push2(f1+f2) *)  |

   211B: (*DSUB  double precision subtraction*)
         (* f1 := Pop2(); f2 := Pop2(); Push2(f1-f2) *)  |

   212B: (*DMUL  double precision multiplication*)
         (* f1 := FLOAT(Pop()); f2 := FLOAT(Pop()); Push2(f1*f2) *)  |

   213B: (*DDIV  double precision division*)
         (* f1 := FLOAT(Pop()); f2 := Pop2();
            f1 := f2/f1+remainder*..; Push2(f1) *)  |

   214B, 215B: (*reserved for future multiprecision instructions*)
               Trap(instrChk) |

   216B: (*DSHL  double precision multiplication by 2*)
         (* f1 := Pop2(); Push2(2.0*f1) *)    |

   216B: (*DSHR  double precision division by 2*)
         (* f1 := Pop2(); Push2(f1 / 2.0) *)  |


   220B: (*SSW  store stack word*)
         k := pop(); i := pop()+next();
         IF overflow THEN Trap(addrChk)
         ELSE
           stk[i] := k
         END |

   221B: (*SSD  store stack double word*)
         k := pop(); j := pop(); i := pop()+next();
         IF overflow OR (i=NILL) THEN Trap(addrChk)
         ELSE
           stk[i] := j; stk[i+1] := k
         END |

   222B: (*SSD0  store stack double word*)
         k := pop(); j := pop(); i := pop();
         IF i=NILL THEN Trap(addrChk)
         ELSE
           stk[i] := j; stk[i+1] := k
         END |

   223B: (*SXFW  store indexed frame word*)
         i := pop();
         k := pop() + pop()*4; (*18 bits*)
         IF overflow(*18 bits*) THEN Trap(addrChk)
         ELSE
           stk[k] := i
         END |

   224B: (*TS  test and set*)
         i := pop(); push(stk[i]); stk[i] := 1 |

   225B: (*SXB  store indxed byte*)
         k := pop(); i := pop(); j := pop() + (i DIV 2);
         IF overflow THEN Trap(addrChk)
         ELSIF i MOD 2 = 0 THEN stk[j] := k*400B + (stk[j] MOD 400B)
                           ELSE stk[j] := (stk[j] DIV 400B) * 400B + k
         END |

   226B: (*SXW  store indexed word*)
         k := pop(); i := pop()+pop();
         IF overflow THEN Trap(addrChk)
         ELSE
           stk[i] := k
         END |

   227B: (*SXD  store indexed double word*)
         k := pop(); j := pop(); i := 2*pop()+pop();
         IF overflow(*multiplication or addition*) OR (i=NILL) THEN Trap(addrChk)
         ELSE
```

```
      stk[i] := j; stk[i+1] := k
    END |

  230B .. 233B: (*FADD - FDIV*)
    (* floating point operations *)
    IF overflow (*OR zerodivide*) THEN Trap(realOvfl) END |

  234B: (*FCMP  floating compare *)
    f1 := Dpop(); f2 := Dpop();
    IF f1>f2 THEN push(0); push(1)
    ELSIF f1<f2 THEN push(1); push(0)
    ELSE push(0); push(0)
    END |

  235B: (*FABS  floating absolute value *) |

  236B: (*FNEG  floating negative *) |

  237B: (*FFCT  floating functions *)
    i := next();
    IF i=0 THEN (*FLOAT float*)
    ELSIF i=1 THEN (*FLOATD float double*)
    ELSIF i=2 THEN (*FIX fix*)
      (*may cause a floating overflow trap*)
    ELSIF i=3 THEN (*FIXD fix double*)
    (*ELSIF .......*)          (*lateron additions... *)
    ELSE Trap(instrChk)
    END |

  240B: (*READ*)  stk[pop()] := get(pop()) |

  241B: (*WRITE*) put(pop(), pop()) |

  242B: (*DSKR  disk read*)
    put(5, 9);  (* clr addr *)
    i := pop();
    FOR i := i TO i+127 DO
      stk[i] := get(8);
    END |

  243B: (*DSKW  disk write*)
    put(5, 9);  (* clr addr *)
    FOR i := 1 TO 3 DO put(0, 8); END;
    put(0A6A6H, 8);
    i := pop();
    FOR i := i TO i+127 DO
      put(stk[i], 8);
    END |

  244B: (*SETRK  set track*)
    REPEAT k := get(9) UNTIL 13 IN BITSET(k);
    i := pop();
    put(i DIV 100H, 10);
    put(i MOD 100H, 11) |

  246B: (*ESC*)  (*escape*)
     i := next();
     IF i=0 THEN
       (*micro jump to high micro ram; used for debugging new instructions*)
     ELSIF i IN {1..3} THEN (*printer instructions*)
     ELSE (* free for extensions; now boots, is considered as error *)
     END |

  247B: (*SYS  rarely used system functions: dump, boot,... *)
    i := next();
    IF i=0 THEN i := pop()       (* bootstrap the machine from boot #i *)
    ELSIF i=1 THEN               (* dump *)
    ELSIF i=2 THEN push(P)       (* read P register *)
    ELSIF i=3 THEN               (* set H register *)
      i := pop(); H := i-24; stk[P+5] := i;
    ELSIF i=4 THEN push(H+24)    (* read H register *)
    ELSIF i=5 THEN push(1)       (* version number for 13.2: 1*)
  (*ELSIF i=... THEN ...*)       (* lateron additions... *)
    ELSE Trap(instrChk)
    END |

  250B: (*ENTP  entry priority*)
```

```
      i := next();
      lm := (*Mask of i bits*) Mask{0..i-1}; (*where {0..-1} means {}*)
      IF NOT (lm >= M) THEN Trap(prioChk)
      ELSE
        stk[L+3] := CARDINAL(M);
        M := lm;
        MSK := Mask(stk[devstatadr])+M
      END |

251B: (*EXP  exit priority*)
      M := Mask(stk[L+3]);
      MSK := Mask(stk[devstatadr])+M |

256B: (*TRA  coroutine transfer*)
      Transfer(BOOLEAN(next()), pop(), pop()) |

257B: (*RDS  read string*)   k := pop(); i := next();
      REPEAT
        stk[k] := next2(); INC(k); DEC(i)
      UNTIL i < 0  |

260B: (*LODFV  reload expression stack after function return*)
      i := pop(); restoreExpStack; push(i) |

261B: (*LODFD  reload expression stack after function return*)
      i := pop(); j := pop(); restoreExpStack; push(j); push(i) |

262B: (*STORE  save expression stack*)
      IF S>H-(expStackSize+1) THEN
        PC := PC-1;
        Trap(storageChk);
      ELSE  saveExpStack  END |

263B: (*STOFV  store expression stack with formal procedure variable on top*)
      i := pop();
      IF S>H-(expStackSize+1) THEN Trap(storageChk)
      ELSE
        saveExpStack; stk[S] := i; INC(S)
      END |

264B: (*STOT  copy from expression stack to procedure stack*)
      IF S>=H THEN Trap(storageChk)
      ELSE
        stk[S] := pop();
        INC(S)
      END |

265B: (*COPT  copy element on top of expression stack*)
      i := pop(); push(i); push(i) |

266B: (*DECS  decrement stackpointer*)
      DEC(S) |

267B: (*PCOP  storage allocation and copy for value parameter *)
      stk[L+next()] := S;
      lsz := pop(); k := S+lsz;
      IF overflow OR (k > H) THEN Trap(storageChk)
      ELSE
        ladr := pop();
        WHILE lsz>0 DO
          stk[S] := stk[ladr]; INC(S); INC(ladr); DEC(lsz)
        END
      END |

270B: (*UADD*)  j := pop(); i := pop(); push(i+j);
                IF overflow THEN Trap(cardOvfl) END |

271B: (*USUB*)  j := pop(); i := pop(); push(i-j);
                IF overflow THEN Trap(cardOvfl) END |

272B: (*UMUL*)  j := pop(); i := pop(); push(i*j);
                IF overflow THEN Trap(cardOvfl) END |

273B: (*UDIV*)  j := pop(); i := pop(); push(i DIV j);
                IF j=0 THEN Trap(cardOvfl) END |

274B: (*UMOD*) j := pop(); i := pop(); push(i MOD j);
```

```
        IF j=0 THEN Trap(cardOvfl) END |

   275B: (*ROR*)  j := pop(); (*rotate top right by j MOD 16 positions*) |

   276B: (*SHL*)  j := pop(); (*shift top left by j MOD 16 positions*)  |

   277B: (*SHR*)  j := pop(); (*shift top right by j MOD 16 positions*)  |

   300B: (*FOR1  entry forloop *)
         i := next(); (* =0: up; <>0: down *)
         lup := pop(); llow := pop(); ladr := pop();
         k := PC+next2();
         IF ((i=0)  AND (llow<=lup)) OR
            ((i<>0)  AND (llow>=lup)) THEN (* enter the for loop *)
           stk[ladr] := llow;
           stk[S] := ladr; INC(S);
           IF S>=H THEN Trap(storageChk)
           ELSE
             stk[S] := lup; INC(S)
           END
         ELSE (* dont execute the for loop *)
           PC := k
         END |

   301B: (*FOR2  end forloop *)
         lup := stk[S-1]; ladr := stk[S-2];
         lsz := INTEGER(next()); (* range -128..+127 *)
         k := PC+next2();
         i := stk[ladr]+lsz;
         IF overflow OR ((lsz>=0) AND (i>lup))
                     OR ((lsz<=0) AND (i<lup))
         THEN (* termination *)
           DEC(S,2)
         ELSE (* continuation of the loop *)
           stk[ladr] := i;
           PC := k
         END |

   302B: (*ENTC  entry to a case statement*)
         PC := PC+next2();
         k := pop();
         llow := next2(); lup := next2();
         stk[S] := PC+2*(lup-llow)+4;
         IF S>=H THEN Trap(storageChk)
         ELSE
           INC(S);
           IF (k >= llow) AND (k <= lup) THEN
             PC := PC+2*(k-llow+1)
           END;
           PC := PC+next2()
         END |

   303B: (*EXC  exits a case statement*)
         DEC(S); PC := stk[S] |

   304B: (*TRAP*)
         i := pop(); Trap(i MOD 16) |

   245B, 305B: (*UCHK, CHK   check j <= i <= k *)
         k := pop(); j := pop(); i := pop(); push(i);
         IF (i<j) OR (i>k) THEN Trap(rangeChk) END  |

   306B: (*CHKZ  check i <= k *)  (*CARDINAL*)
         k := pop(); i := pop(); push(i);
         IF i>k THEN Trap(rangeChk) END  |

   307B: (*CHKS  check sign bit*)
         k := pop(); push(k);
         IF INTEGER(k)<0  THEN Trap(rangeChk) END |

   310B: (*EQL*)  j := pop(); i := pop();
                  IF i = j THEN push(1) ELSE push(0) END  |

   311B: (*NEQ*)  j := pop(); i := pop();
                  IF i # j THEN push(1) ELSE push(0) END  |

   312B, 252B: (*LSS, ULSS*) j := pop(); i := pop();
```



```
           IF i < j THEN push(1) ELSE push(0) END  |

313B, 253B: (*LEQ, ULEQ*) j := pop(); i := pop();
           IF i <= j THEN push(1) ELSE push(0) END |

314B, 254B: (*GTR, UGTR*) j := pop(); i := pop();
           IF i > j THEN push(1) ELSE push(0) END  |

315B, 255B: (*GEQ, UGEQ*) j := pop(); i := pop();
           IF i >= j THEN push(1) ELSE push(0) END |

316B: (*ABS*) i := pop(); push(ABS(INTEGER(i)));
             IF i=100000B THEN Trap(intOvfl) END |

317B: (*NEG*) i := pop(); push(-INTEGER(i));
             IF i=100000B THEN Trap(intOvfl) END |

320B: (*OR*)   j := pop(); i := pop(); (* k := i OR j *) push(k) |

321B: (*XOR*)  j := pop(); i := pop(); (* k := i XOR j *) push(k)|

322B: (*AND*)  j := pop(); i := pop(); (* k := i AND j *) push(k)|

323B: (*COM*)  push(-pop()-1) |

324B: (*IN*)   j := pop(); i := pop(); (* k := i IN j; gives FALSE if k>15 *) push(k) |

325B: (*LIN  load immediate NIL *) push(NILL) |

326B: (*MSK*)  j := pop(); (* k := mask with j MOD 16 ones *) push(k) |

327B: (*NOT*)  i := pop(); (* i := NOT i *) (*boolean: 0<->1*) push(i) |
               (* current implementation:
                  b := BITSET(pop());
                  IF 15 IN b THEN EXCL(b, 15) ELSE INCL(b, 15) END;
                  push(CARDINAL(b)) *)

330B: (*ADD*)  j := pop(); i := pop(); push(i+j);
               IF overflow THEN Trap(intOvfl) END |

331B: (*SUB*)  j := pop(); i := pop(); push(i-j);
               IF overflow THEN Trap(intOvfl) END |

332B: (*MUL*)  j := pop(); i := pop(); push(i*j);
               IF overflow THEN Trap(intOvfl) END |

333B: (*DIV*)  j := pop();
               i := pop(); push(i DIV j);
               IF (j=0) OR ((j=-1) AND (i=-32768)) THEN Trap(intOvfl) END |

334B: (*reserved for future instruction as needed by the compiler*)
               Trap(instrChk) |

335B: (*BIT*)  j := pop(); (* k := {j MOD 16} *) push(k) |

336B: (*NOP*) |

337B: (*MOVF  move frame *)
      i := pop();
      fromea := pop()+pop()*4; (*18 bits*)
      toea   := pop()+pop()*4; (*18 bits*)
      IF overflow(*18 bits*) THEN Trap(addrChk)
      ELSE
        WHILE i>0 DO
          stk[toea] := stk[fromea]; INC(toea); INC(fromea); DEC(i)
        END
      END |

340B: (*MOV  move block*)
      k := pop(); j := pop(); i := pop();
      IF (j=NILL) OR overflow(*on i+k*) THEN Trap(addrChk) (*source may wrap around*)
      ELSE
        WHILE k>0 DO stk[i] := stk[j]; INC(i); INC(j); DEC(k) END
      END |

341B: (*CMP  compare blocks*)
      k := pop(); j := pop(); i := pop();
```

```
        IF overflow (* on either: j+k or i+k*) THEN Trap(addrChk)
        ELSIF k=0 THEN push(0); push(0)
        ELSE
          WHILE(stk[i] # stk[j]) AND (k > 0) DO
            INC(i); INC(j); DEC(k)
          END;
          push(stk[i]); push(stk[j])
        END |

  342B: (*DDT  display dot*)
        y := pop(); x := pop(); dbmd := pop(); i := pop()
        (* display point at x,y
           according to the mode i
           inside the bitmap described at dbmd
           may cause a Trap(addrChk) *) |

  343B: (*REPL  replicate pattern *)
        db := pop(); sb := pop(); dbmd := pop(); i := pop()
        (* replicate the pattern at sb
           over the block described at db
           inside the bitmap described at dbmd
           according to the mode i
           may cause a Trap(addrChk) *) |

  344B: (*BBLT  bit block transfer*)
        sbmd := pop(); db := pop(); sb := pop(); dbmd := pop(); i := pop()
        (* transfer the block described at sb
           inside the bitmap described at sbmd
           to the block described at db
           inside the bitmap described at dbmd
           according to the mode i
           may cause a Trap(addrChk) *) |

  345B: (*DCH  display character*)
        j := pop(); db := pop(); fo := pop(); dbmd := pop()
        (* convert the character j
           from the font stored at fo
           to the block described at db
           inside the bitmap described at dbmd
           may cause a Trap(addrChk) *) |

  346B: (*UNPK  unpack*) |

  347B: (*PACK  pack*) |

  350B: (*GB  get base adr n levels down*)
        i := L; j := next();
        REPEAT
          i := stk[i]; DEC(j)
        UNTIL j=0;
        push(i) |

  351B: (*GB1  get base adr 1 level down*)  push(stk[L]) |

  352B: (*ALLOC  allocate block*)
        i := pop(); push(S); S := S + i;
        IF overflow OR (S > H) THEN S := pop(); Trap(storageChk) END |

  353B: (*ENTR  enter procedure*)
        i := next();  S := S + i;
        IF overflow OR (S > H) THEN S := S - i; Trap(storageChk) END |

  354B: (*RTN  return from procedure*)
        S := L; L := stk[S+1]; i := stk[S+2];
        IF i < 0
        THEN (* external *)
          G := stk[S];
          F := stk[G];
          PC := i - 100000B;
        ELSE (* local *)
          PC := i;
        END |

  355B: (*CX  call external procedure*)
        j := next(); i := next();
        mark(G, TRUE);
        G := stk[dft+j];
```

```
        F := stk[G];
        PC := 2*i; PC := next2()  |

   356B: (*CI  call procedure at intermediate level*)
        i := next(); mark(pop(), FALSE);
        PC := 2*i; PC := next2()  |

   357B: (*CF  call formal procedure*)
        k := stk[S-1];
        mark(G, TRUE);
        j := k DIV 400B;
        G := stk[dft+j];
        F := stk[G];
        PC := 2*(k MOD 400B); PC := next2()  |

   360B: (*CL  call local procedure*)
        i := next(); mark(L, FALSE);
        PC := 2*i; PC := next2()  |

   361B .. 377B:  (*CL1 - CL15  call local procedure*)
        mark(L, FALSE);
        PC := 2*(IR MOD 16); PC := next2()

   ELSE Trap(instrChk)
   END
  END (*LOOP*)
END Interpreter.
```

Appendix D
The M-code Interpreter (microprogram)

The listing for the M-code interpreter in microcode is found on the standard software release disk under the name: intl3dot2.*. The full listing of this program will not be printed in preliminary versions of this manual.