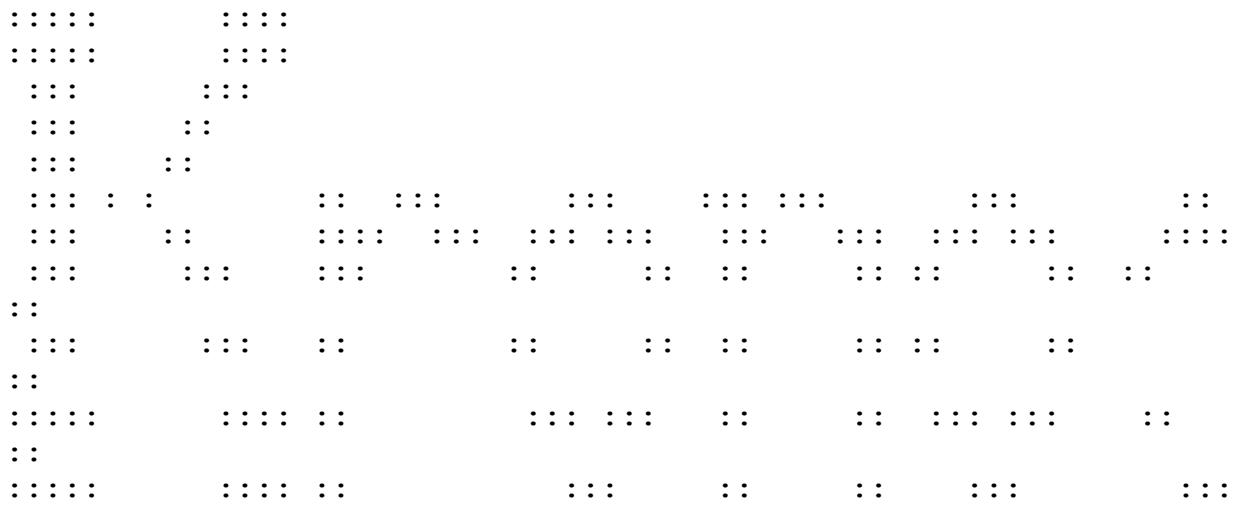
- USSR Academy of Sciences Siberian Division -

- Computing Center -



The Architecture of KRONOS Family Processors

Novosibirsk - 1988

We hope information offered in this guide is valid. Nevertheless we shall be very glad if user reads this guide critically and responds us about all questions, misses, disadvantages and wishes. Authors acknowledge with thanks and pleasure everybody who participated in guide producing.

Our address is:

Computing Center, prospect Lavrentieva, 6, Novosibirsk, 630090, USSR.
Phone: 35-50-67.

Version from 16.10.88

Contents

INTRODUCTION

The architecture of KRONOS family processors is oriented to the support of the high-level languages (C, Modula-2, Pascal, Occam) and thereby gives possibility to design modern conceptions for computer application. 32-bit machine word allows one to use processors of the family for scientific research. The wide address space (2 billion words) gives possibility to use virtual memory for designing object-oriented models and artificial intelligence systems. The hardware support of the interruption handler (for the events and processes synchronization) and compact code give us surance that the processors of KRONOS family may be successfully used in the real-time systems.

Any processor of the family may be used in a single computer or in a multiprocessing system as well.

KRONOS processors

1. KRONOS-2 is the first implementation of preceding concepts. It is embedded in "Electronica-60" computer and is compatible with its peripheral devices and memory. The processor is designed on chips of 1802, 1804, 155 and 531 series. In contrast to "Electronica-60", the processor has 32-bit word, twice performance, and address space which reaches 4Mbyte.

2. KRONOS-2.5 is the development of KRONOS-2 processor distinguished by higher performance: 1 million instructions over stack per second. Interface with the external devices is performed via MULTIBUS 1.

3. In KRONOS-2.6 the possibility of using direct communication channels with high capacity for integration several processors is expected, thus providing usage of processors 2.6 extended by emulators of arithmetic operations as the basis of the system MARS-T. High capacity is provided by transputer organization of the system and servers and functional units included. Processors 2.6 may also be used in workstations of MARS system.

Now the single-chip variant of KRONOS processor is designed. The perspective of this work are provided by simplicity of instruction set hardware design and the necessity in personal computers with high power, supporting high-level language programming. Modern element base provides the creation of systems with performance about 5 million instructions per second for each transputer element.

KRONOS-2.X processors family

KRONOS-2.X processors family consists of processors having been designed over different element bases with the usage of different interface buses. All of them have M-code as the instruction set and differ only by performance and peripheral control methods.

Engineering characteristics of KRONOS processors

Processor	Kronos 2.2	Kronos 2.5	Kronos 2.6
Standard	Q-22 (c) DEC	(c) Intel	EuroCard E-2
Number of cards	1	2	2-8
Bus	Q-bus 22	Multibus-1	local
RAM size	4 Mbyte	2,5 Mbyte	8 Gbyte
Clock rate, mHz	4	3	3
Number of op. over the stack, m. per sec.	0,6	1	1,5

This guide gives the entire overview of processors of the KRONOS family architecture. The first chapter introduces the main notions of architecture. Chapter "M-code interpreter" gives declaration in Modula-2 of the processor instruction set followed by comments. The volume is ended by examples illustrating Modula-2 compiler and processing.

VIRTUAL MODULA-2 MACHINE

The Virtual Modula-2 Machine and its interpreters support the process of program execution. This Chapter characterizes M-code, defines semantics of its instructions illustrating some of them.

The main differencies of VM2M from traditional machines:

1) evaluation of expressions on quick stack with small fixed depth. Blasting of this stack (copying in memory) during a procedure call;

2) separation of code and data areas for each process which provides reenterability for all programs and even for their constituents (modules);

3) refusal to use absolute addressing even in a code segment. Usage of displacement tables of subprogram entries simplifies their invocations;

4) advanced kinds of addressing reflects notions of modern programming languages. There is addressing for local, global, external and intermediate objects;

5) special instructions simplify implementation of loops, calls, case statements and some other ones;

6) a table of separately loaded modules provides organizing dynamic loading-linking-execution of programs;

7) there are multivalue operating instructions.

Futher it is supposed that reader knows followed Modula-2 notions:

PROGRAM
MODULE
IMPORT-EXPORT of OBJECTS
GLOBAL VARIABLES
PROCEDURE
LOCAL PROCEDURE
LOCAL VARIABLES

Input/output essentially depend on implementation and are not considered here.

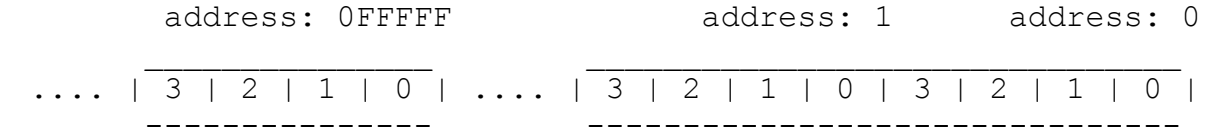
VM2M consists of **processor**, **stack** for the storage of values and expressions evaluation (further it is designated as **A-stack**), register-pointer (**P-register**), **code segment** and data.

VM2M Processor serves for data elaboration and interpretation of M-code control instructions.

A-stack is a quick stack of a small depth (whose elements are 32-bit machine words) over which the traditional instructions putting on (Push) and taking (with deleting) from (Pop) the stack top are defined. Hardware supports control for stack overflow/underflow with corresponding interruption raising.

VM2M **Memory** is considered as a linear sequence of 32-bit words each related with 32-bit number - its address. Only the whole word may be addressed (i.e. a machine word is the

addressing unit).



The program which is currently being executed is said to be a **process**.

Such structures exist in memory during VM2M processing: separately-compiled modules, **procedure stacks** (further **P-stack**), **process descriptors** and table of pointers on global domains of loaded modules (the so-called global Data Frame Table - **DFT**).

Process descriptor consists of 7 machine words which contain pointers to the informational data structures related with the process. These pointers determinate the process context, i. e. the whole information needed for a current process execution by virtual machine. The base address of the current process descriptor lies in P-register of the **processor**.

In terms of Modula-2 a process descriptor is the record:

```
TYPE Process_Descriptor = RECORD
                            F, PC, G, H, S, L, M: WORD;
                          END;
```

Every descriptor field has a special purpose, which is described below. Further descriptor fields will be called for conveniency as processor registers (e.g. "G-register" instead of "Process_Descriptor.G").

Note. The existing implementation of VM2M (processors of KRONOS family) in fact have specialized registers which contain copies of corresponding fields of the current process descriptor during the execution time. When process switching takes place, register contents is copied in memory starting from the address contained in P-register.

Module which is loaded in memory and ready for execution consists of **code segment**, **global data area**, constant area of composite types, i.e. strings, arrays, records (**string pool**), external module link area - local **DFT** (if module contains object import).

Code segment occupies contiguous area and has a complex structure. **Procedure table** (up to 256 words) lies at the top. It contains displacements in bytes between a segment base and the origin of the corresponding procedure. Thus, the inner procedure is precisely identified by its number in the module procedure table. During the execution of the inner procedure, F-register contains a pointer to the code segment base. PC (Program Counter) contains offset in bytes between the code segment base and byte which contains the next instruction.

Global data area of module occupies a contiguous area of words. The single-word data are stored there immediately but composite variables are represented with the help of pointers in special areas of memory. G-register points to the global data area base of a current module. Thus, access to module global variables is organized as G-register indexation, i.e. the global word with number 3 is situated in the cell with the address [G]+3 ([REG] denotes contents of register "REG"). The first two words of the global data area contain special information: the lower word (its address equals 0) of the area contains the pointer to the code segment base of the module (copy of F-register) and the next word contains pointer to the string pool base.

String pool occupies a contiguous area and contains literals of composite types (strings, arrays, records). The first word (it offset equals 1) of the module global data area points to the string pool base.

External module **link area** (local DFT) occupies contiguous area and may be represented as an array of words, whose elements contain pointer to elements of the global DFT referring to the global data area base of the external module. Thus, external modules may be identified by the index in the module local DFT, i. e. DFT[i] contains the address of a pointer to the global data area of i-th external module.

Local DFT is situated immediately before the module global data area, i.e. the element DFT[i] address equals [G]-i-1. Remark: such organization of information gives possibility to access to statically existing entities (procedures, variables, i.e. external module objects which may be imported), only knowing the address of the global data area base.

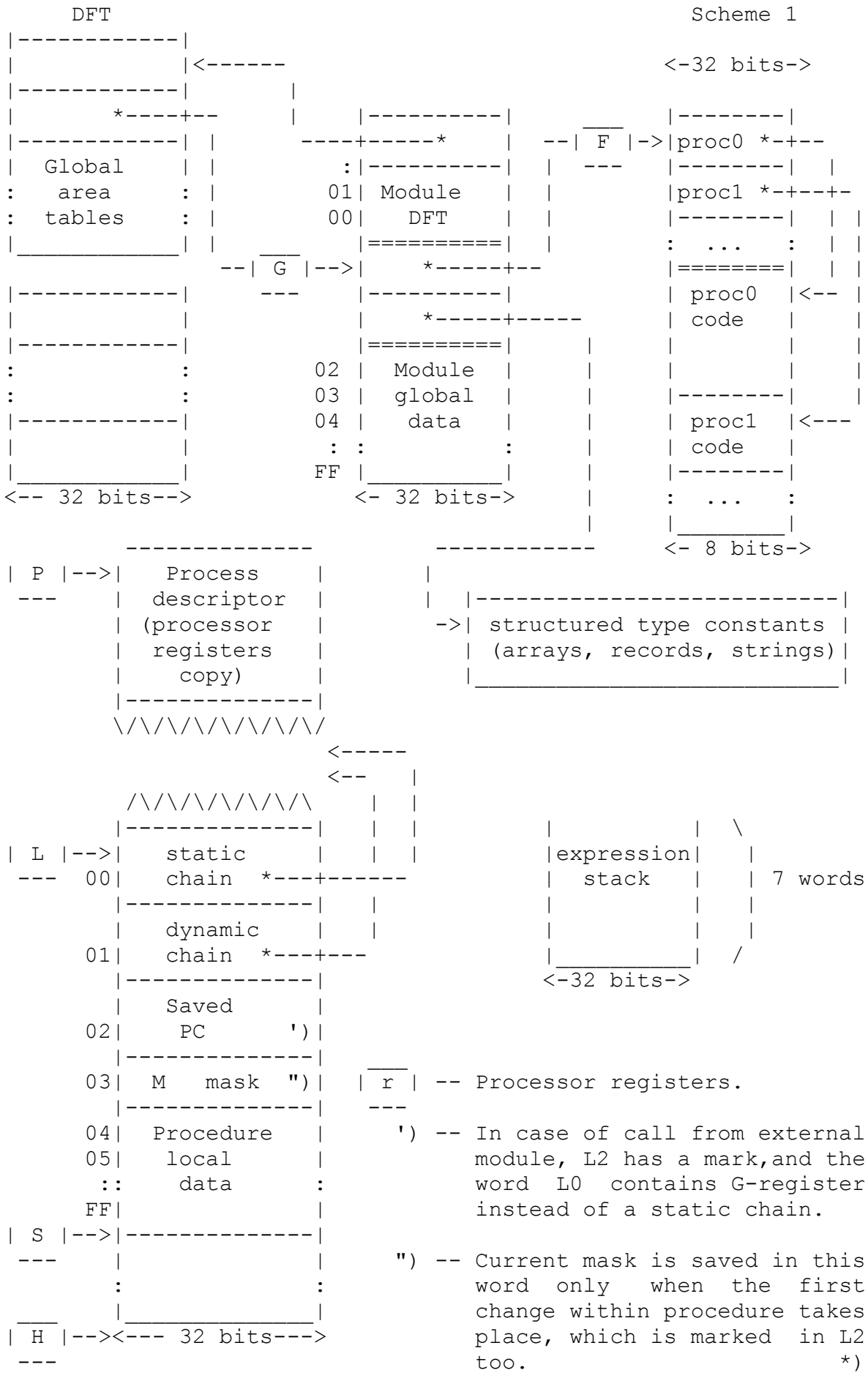
P-stack is used for **procedure local data** location and organization of procedure calls and return from them. Three processor registers point to stack: S-register points to the top of the P-stack, i.e. the first free P-stack word. H-register points to the upper word in the storage area reserved for stack (limit for S increasing). L-register points to the local data area base of the procedure which is being executed (current procedure). The overlap of the registers S and H (i.e. situation when [S]>=[H]) is determined at the hardware level with raising corresponding interruption.

Current procedure **local data** area is stored in the P-stack and represents a sequence of words which contain either single-word local variables or pointers to a storage area base allocated for variables of composite types. Access to the local data is performed as indexation of L-register, i.e. i-th word has [L]+i address. Special instructions simplify access to words with numbers from 4 to 255.

The first four local words (with numbers 0..3) are reserved for special aims: the lower word points either to the local data area base of the enveloping procedure (i.e.the procedure which immediately contains a current procedure) or on a calling module global data area if the procedure is

called from the external module (**static chain**). The next word points to the data area base of the procedure which calls the current one (**dynamic chain**). The next word contains the value for PC returning. In the last word, the processor interrupt mask is saved (see remark for scheme (1) and M-code interpreter).

Scheme (1) illustrates code and statical and dynamical data organization of process.


DFT
Scheme 1
<-32 bits->
Global area tables
Module DFT
Module global data
proc0
proc1
proc0 code
proc1 code
<-- 32 bits-->
<- 32 bits->
<- 8 bits->
Process descriptor (processor registers copy)
structured type constants (arrays, records, strings)
static chain
dynamic chain
Saved PC
M mask
Procedure local data
expression stack
7 words
r
-- Processor registers.
-- In case of call from external module, L2 has a mark,and the word L0 contains G-register instead of a static chain.
-- Current mask is saved in this word only when the first change within procedure takes place, which is marked in L2 too. *)

VM2M code (**M-code**) represents the byte stream, i.e. opcode always occupies a single byte. VM2M instructions have no address fields, but opcode determines where operands are situated. The following addressing modes are used in VM2M:

```
local            : address = [L] + N
global           : address = [G] + N
external         : address = DFT[M]^ + N
intermediate     : address = {[L]^} + N OR [[L]^]^ + N OR ...
indirect         : address = [POP()]^ + N,
```

where M is the external module number, N - object index, "^" symbol denotes indirection operator.

Intermediate addressing mode represents the access to non-local variables in procedure and realizes the pass through the procedure static chain.

Some instructions contain from 4 bits to 4 bytes of immediate operands, and opcode always determines the total length of immediate addresses following it. Program counter (PC) increments from 1 to 4 bytes in dependence on the instruction mode.

VM2M instructions may be divided into the groups as follows: arithmetical-logical instructions, control organization instructions, instructions over A-stack, auxiliary instructions.

Stack instructions contain constant loading instructions, instructions of stack exchang with local, global, external and other data.

All arithmetical-logical instructions operate over one or two stack elements and store the result again on the stack.

Control organization instructions are represented by conventional conditional and unconditional jump instructions, special instructions for implementing FOR and CASE statements, procedure calling and returning instructions. They also include instructions TRAP and TRANSFER (for process switch).

There are different modes of procedure parameter passing. In this implementation parameter passing is realized through the arithmetical stack, which reduces consumptions on parameter processing during procedure call.

All interruptions are handled as switching of those processes whose pointers are contained in the corresponding interrupt vector elements (i.e. the element index equals to the interruption number).

VM2M has a collection of auxiliary instructions simplifying processing of multivalues, procedure parameters, I/O instructions and so on.

More comprehensive information about VM2M may be obtained after the search of M-code interpreter program text which plays the role of microprogram specification of KRONOS family processors.

```
(*            THE  KRONOS  2.X   PROCESSOR  FAMILY

                        INSTRUCTION SET

     00      20      40      60      80      A0      C0      E0

00   LI0     LLW     LXB     LSW0            LSS     MOVE    INCL
01   LI1     LGW     LXW     LSW1    QUIT    LEQ     RDS     EXCL
02   LI2     LEW     LGW2    LSW2    GETM    GTR     LSTA    SLEQ
03   LI3     LSW     LGW3    LSW3    SETM    GEQ     COMP    SGEQ
04   LI4     LLW4    LGW4    LSW4    TRAP    EQU     GB      INC1
05   LI5     LLW5    LGW5    LSW5    TRA     NEQ     GB1     DEC1
06   LI6     LLW6    LGW6    LSW6    TR      ABS     CHK     INC
07   LI7     LLW7    LGW7    LSW7    IDLE    NEG     CHKZ    DEC

08   LI8     LLW8    LGW8    LSW8    ADD     OR      ALLOC   STOT
09   LI9     LLW9    LGW9    LSW9    SUB     AND     ENTR    LODT
0A   LI0A    LLW0A   LGW0A   LSW0A   MUL     XOR     RTN     LXA
0B   LI0B    LLW0B   LGW0B   LSW0B   DIV     BIC     NOP     LPC
0C   LI0C    LLW0C   LGW0C   LSW0C   SHL     IN      CX     *BBU
0D   LI0D    LLW0D   LGW0D   LSW0D   SHR     BIT     CI     *BBP
0E   LI0E    LLW0E   LGW0E   LSW0E   ROL     NOT     CF    **BBLT
0F   LI0F    LLW0F   LGW0F   LSW0F   ROR     MOD     CL

10   LIB     SLW     SXB     SSW0 ***IO0     DECS    CL0     SWAP
11   LID     SGW     SXW     SSW1    IO1     DROP    CL1     LPA
12   LIW     SEW     SGW2    SSW2    IO2     LODFV   CL2     LPW
13   LIN     SSW     SGW3    SSW3    IO3     STORE   CL3     SPW
14   LLA     SLW4    SGW4    SSW4    IO4     STOFV   CL4     SSWU
15   LGA     SLW5    SGW5    SSW5    IO5     COPT    CL5
16   LSA     SLW6    SGW6    SSW6    IO6     CPCOP   CL6
17   LEA     SLW7    SGW7    SSW7    IO7     PCOP    CL7

18   JFLC    SLW8    SGW8    SSW8    FADD    FOR1    CL8
19   JFL     SLW9    SGW9    SSW9    FSUB    FOR2    CL9
1A   JFSC    SLW0A   SGW0A   SSW0A   FMUL    ENTC    CL0A
ACTIV
1B   JFS     SLW0B   SGW0B   SSW0B   FDIV    XIT     CL0B    USR
1C   JBLC    SLW0C   SGW0C   SSW0C   FCMP    ENTS    CL0C    SYS
1D   JBL     SLW0D   SGW0D   SSW0D   FABS            CL0D   *NII
1E   JBSC    SLW0E   SGW0E   SSW0E   FNEG    ORJP    CL0E
1F   JBS     SLW0F   SGW0F   SSW0F   FFCT    ANDJP   CL0F
INVLD


  * -- Not implemented in KRONOS 2.2.
 ** -- Not implemented.
*** -- Instructions IO0..IO7 define the bus-processor interface.

-------------------------------------------------------------
(c) COPYRIGHT KRONOS Research Group      1985,1986,1987

    Last modification                    30-Aug-1987     *)
```

```
MODULE Kronos_Interpreter;  (* Leo 27-Nov-85. (c) KRONOS *)
                            (* Ned 30-Aug-87. (c) KRONOS *)

(* This interpreter is hardware specification. *)

FROM SYSTEM    IMPORT  ADDRESS, WORD, ADR;
FROM KRONOS    IMPORT  ROR, ROL, SHR, SHL;

TYPE CPUs = (KRONOS2_2, KRONOS2_5, KRONOS2_6);

VAR cpu: CPUs;

CONST ESdepth = 7; (* Expression stack depth *)

TYPE
  BYTE     = [0..255];
  WORD16   = [0..0FFFFh];
  PC_Range = WORD16;
  CodePtr  = POINTER TO ARRAY PC_Range OF BYTE;

VAR
  PC:    PC_Range;    (* program counter *)
  IR:    [0..0FFh];   (* instruction register *)
  F :    CodePtr;     (* code segment address *)
  G :    ADDRESS;     (* global data segment address *)
  L :    ADDRESS;     (* local data segment address *)
  S :    ADDRESS;     (* P-stack top address *)
  H :    ADDRESS;     (* P-stack bound *)
  P :    ADDRESS;     (* process descriptor address *)
  M :    BITSET;      (* interrupt mask *)
  Ipt:   BOOLEAN;     (* interrupt request *)
  IptNo: WORD16;      (* interrupt number *)

(* Hardware fixed interrupt numbers:

   01h    timer
   02h    processor halt
   03h    memory violation
   04h    power crash
   05h    processor error
   06h    interrupt vector input error
   07h    unimplemented instruction
   08h    on procedure call        (KRONOS 2.2)
   09h    on procedure return      (KRONOS 2.2)

   0Bh    trace (interrupt on each instruction) (KRONOS 2.5,2.6)

   40h    P-stack overflow (S>H)
   41h    integer overflow
   42h    floating overflow
   43h    floating underflow
   44h    address overflow
```

```
   49h    instruction INVLD
   4Ah    check bounds error
   4Bh    wrong instruction parameter (hardware ASSERT)
   4Ch    expression stack overflow or underflow

   Note 1. See the procedure NotMasked for interrupts masking
   method.

   Note 2. If there are processor models in brackets, then
   interrupt raises only for these models.
*)

CONST  (* bit numbers in the word L2 (see fig.) *)
  ExternalBit   = 1Fh;
  ChangeMaskBit = 1Eh;

CONST
  NonVectBit = 1Fh; (* bit masking program interrupts *)

(* Note. H-register contains the stack bound decremented on
   ESdepth+1 words to save the expression stack on process
   transfer.
*)

VAR Core: ARRAY ADDRESS OF WORD;
    ByteCore: ARRAY OF BYTE; (* combined with Core *)

MODULE InstructionFetch;

  IMPORT F, PC, WORD, WORD16, BYTE, Core, ADDRESS;
  EXPORT Next, Next2, Next4, GetPc;

  PROCEDURE Next(): BYTE;
  BEGIN INC(PC); RETURN INTEGER(F^[PC-1])
  END Next;

  PROCEDURE Next2(): WORD16;
  BEGIN RETURN Next()+Next()*100h
  END Next2;

  PROCEDURE Next4(): WORD;
  BEGIN RETURN Next2()+Next2()*10000h
  END Next4;

  PROCEDURE GetPc(procno: INTEGER): INTEGER;
  (* Gives PC at a procedure entry *)
  BEGIN RETURN Core[ADDRESS(F)+procno]
  END GetPc;

END InstructionFetch;

MODULE Mask;
```

```
  IMPORT M, NonVectBit, BYTE;
  EXPORT NotMasked;

  PROCEDURE NotMasked(N: BYTE): BOOLEAN;
  BEGIN
    IF    (N>=0Fh) & (N<3Fh)  THEN RETURN (0 IN M)
    ELSIF (N< 0Fh) & (N>0)    THEN RETURN (0 IN M) & (N IN M)
    ELSIF (N =3Fh)            THEN RETURN (NonVectBit IN M)
    ELSE ASSERT(FALSE)
    END
  END NotMasked;

END Mask;

MODULE ExpressionStack;
  IMPORT WORD, ESdepth, Ipt, IptNo;
  EXPORT Push, Pop, Empty;

  VAR A: ARRAY [0..ESdepth-1] OF WORD;  sp: [0..ESdepth];

  PROCEDURE Push(X: WORD);
  BEGIN A[sp]:=X;
    IF sp<ESdepth THEN INC(sp) ELSE Ipt:=TRUE; IptNo:=4Ch END;
  END Push;

  PROCEDURE Pop(): INTEGER;
  BEGIN
    IF sp=0 THEN Ipt:=TRUE; IptNo:=4Ch ELSE DEC(sp) END;
    RETURN A[sp];
  END Pop;

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

BEGIN sp:=0 END ExpressionStack;

MODULE ProcessSupport;
  IMPORT PC,G,F,H,L,S,P,M, Core, NotMasked, ESdepth
       , CodePtr, WORD16, ADDRESS;
  FROM ExpressionStack IMPORT Pop, Push, Empty;

  EXPORT SaveExpStack, RestoreExpStack, Transfer, TRAP;

  PROCEDURE SaveExpStack;
    VAR c: CARDINAL; (* stack depth counter *)
  BEGIN c:=0;
    WHILE NOT Empty() DO Core[S]:=Pop(); INC(S); INC(c) END;
    Core[S]:=c; INC(S);
  END SaveExpStack;

  PROCEDURE RestoreExpStack;
    VAR c: CARDINAL; (* stack depth counter *)
  BEGIN DEC(S); c:=Core[S];
```

```
    WHILE c>0 DO DEC(c); DEC(S); Push(Core[S]) END;
  END RestoreExpStack;

  PROCEDURE SaveRegs;
  BEGIN SaveExpStack;
    Core[P+0]:=G;  Core[P+1]:=L;
    Core[P+2]:=PC; Core[P+3]:=CARDINAL(M);
    Core[P+4]:=S;  Core[P+5]:=H+ESdepth+1;
  END SaveRegs;

  PROCEDURE RestoreRegs;
  BEGIN
    G:=Core[P+0];    F :=CodePtr(Core[G]);
    L:=Core[P+1];    PC:=Core[P+2];  M:=BITSET(Core[P+3]);
    S:=Core[P+4];    H:=Core[P+5]-ESdepth-1;
    RestoreExpStack;
  END RestoreRegs;

  PROCEDURE Transfer(pFrom,pTo: ADDRESS);
    VAR j: CARDINAL;
  BEGIN (* Note: pFrom may be equal to pTo *)
    j:=Core[pTo]; SaveRegs; Core[pFrom]:=P; Core[1]:=P;
    P:=j;         RestoreRegs;             Core[0]:=P;
  END Transfer;

  PROCEDURE TRAP(N: WORD16);
  BEGIN Core[P+6]:=N;
    IF N>3Fh THEN N:=3Fh END;
    IF NotMasked(N) THEN Transfer(N*2,Core[N*2+1]) END;
  END TRAP;

END ProcessSupport;

(* P-stack marking before a procedure call *)

PROCEDURE  Mark(X: ADDRESS; External: BOOLEAN);
  VAR i: ADDRESS;
BEGIN i:=S;
  Core[S]:=X; INC(S); (* static chain *)
  Core[S]:=L; INC(S); (* dynamic chain *)
  IF External THEN Core[S]:=WORD(BITSET(PC)+{ExternalBit})
  ELSE             Core[S]:=PC
  END; INC(S,2); L:=i;
END Mark;

PROCEDURE ioP2_2;
BEGIN ASSERT(IR-90h IN {0..7});
(* See KRONOS 2.2 specification *)
END ioP2_2;

PROCEDURE ioP2_5;
BEGIN ASSERT(IR-90h IN {0..7});
(* Input/output requests transmitted to other processors via the
```

```
   common memory.
*)
END ioP2_5;

PROCEDURE ioP2_6;
BEGIN ASSERT(IR-90h IN {0..7});
(* Depends on a bus *)
END ioP2_6;

(* Working variables of interpreter: *)

VAR i,j,k: CARDINAL;   X,Y  : REAL;
    v,w  : BITSET;     a,b  : CHAR;
    adr,adr1,sz,hi,low: CARDINAL;

PROCEDURE ConsolMicroProgram;
BEGIN
(* Consol  microprogram  provides program bootstrap and executes
   Transfer(0,1) by the operator's command "Go".
*)
END ConsolMicroProgram;

PROCEDURE Interpret;
BEGIN
  CASE IR OF
   00h..0Fh:(* LI0..LI0F  Load Immediate *) Push(IR MOD 10h);

  |10h: (* LIB  Load Immediate Byte *) Push(Next())
  |11h: (* LID  Load Immediate Double byte *) Push(Next2())
  |12h: (* LIW  Load Immediate Word *) Push(Next4())
  |13h: (* LIN  Load Immediate NIL  *) Push(NIL)
  |14h: (* LLA  Load Local  Address *) Push(L+Next())
  |15h: (* LGA  Load Global Address *) Push(G+Next())
  |16h: (* LSA  Load Stack  Address *) Push(Pop()+Next())
  |17h: (* LEA  Load External Addres *)
        i:=G-Next()-1; (* Module DFT index *)
        adr:=Core[i]; (* Pointer to a global DFT element *)
        Push(Core[adr]+Next())
  |18h: (* JLFC Jump Long  Forward Condition *)
        IF Pop()=0 THEN PC:=Next2()+PC
        ELSE INC(PC,2) END
  |19h: (* JLF  Jump Long  Forward *) PC:=Next2()+PC;
  |1Ah: (* JSFC Jump Short Forward Condition *)
        IF Pop()=0 THEN PC:=Next()+PC
        ELSE INC(PC) END
  |1Bh: (* JSF  Jump Short Forward *) PC:=Next()+PC;
  |1Ch: (* JLBC Jump Long  Back    Condition *)
        IF Pop()=0 THEN PC:=-Next2()+PC
        ELSE INC(PC,2) END
  |1Dh: (* JLB  Jump Long  Back    *) PC:=-Next2()+PC;
  |1Eh: (* JSBC Jump Short Back    Condition *)
        IF Pop()=0 THEN PC:=-Next()+PC
        ELSE INC(PC) END
```

```
|1Fh: (* JSB  Jump Short Back    *) PC:=-Next()+PC;
|20h: (* LLW  Load Local  Word *)  Push(Core[L+Next()])
|21h: (* LGW  Load Global Word *)  Push(Core[G+Next()])
|22h: (* LEW  Load External Word *)
      i:=G-Next()-1; adr:=Core[Core[i]]; (* external G *)
      Push(Core[adr+Next()])
|23h: (* LSW  Load Stack addressed Word *)
       Push(Core[Pop()+Next()])
|24h..2Fh: (* LLW0..LLW0F Load Local Word *)
      Push(Core[L+IR MOD 10h])
|30h: (* SLW  Store Local  Word *)  Core[L+Next()]:=Pop()
|31h: (* SLW  Store Global Word *)  Core[G+Next()]:=Pop()
|32h: (* SEW  Store External Word *)
      i:=G-Next()-1; adr:=Core[Core[i]]; (* external G *)
      Core[adr+Next()]:=Pop()
|33h: (* SSW  Store Stack addressed Word *)
      i:=Pop(); Core[Pop()+Next()]:=i
|34h..3Fh: (* SLW0..SLW0F  Store Local Word *)
      Core[L+IR MOD 10h]:=Pop()

|40h: (* LXB  Load Indexed Byte *)
      i:=Pop(); Push(ByteCore[Pop()*4+i]);
|41h: (* LXW  Load Indexed Word *)
      i:=Pop(); Push(Core[Pop()+i])
|42h..4Fh: (* LGW02..LGW0F  Load Global Word *)
      Push(Core[G+IR MOD 10h])
|50h: (* SXB  Store Indexed Byte *)
      j:=Pop(); i:=Pop(); ByteCore[Pop()*4+i]:=j;
|51h: (* SXW  Store Indexed Word *)
      j:=Pop(); i:=Pop(); Core[Pop()+i]:=j
|52h..5Fh: (* SGW02..SGW0F  Store Global Word *)
      Core[G+IR MOD 10h]:=Pop()

|60h..6Fh: (* LSW00..LSW0F  Load  Stack addressed Word *)
      Push(Core[Pop()+IR MOD 10h])
|70h..7Fh: (* SSW00..SSW0F  Store Stack addressed Word *)
      i:=Pop(); Core[Pop()+IR MOD 10h]:=i

|80h: TRAP(7h);
|81h: (* QUIT   Stop processor *) ConsolMicroProgram
|82h: (* GETM   Get Mask *) Push(M)
|83h: (* SETM   Set Mask *)
      IF NOT (ChangeMaskBit IN BITSET(Core[L+2])) THEN
        (* mask is changed the first time *)
        Core[L+2]:=WORD(BITSET(Core[L+2])+{ChangeMaskBit});
        Core[L+3]:=WORD(M)
      END; M:=BITSET(Pop);
|84h: (* TRAP  interrupt simulation *)  TRAP(Pop())
|85h: (* TRA   Transfer control between process *)
      i:=Pop(); Transfer(Pop(),i)
|86h: (* TR    Test & Reset *)
      i:=Pop(); Push(Core[i]); Core[i]:=0
|87h: (* IDLE  IDLE process *)
```

```
        DEC(PC); REPEAT (* not occupying the bus *) UNTIL Ipt

(* In  the  following  six instructions and also in instructions
   MOD,  NEG,  ABS,  FOR2,  INC,  DEC,  INC1, DEC1 the interrupt
   IptNo=41h is raised in case of overflow.
*)
  |88h: (* ADD  integer ADD *) Push(Pop()+Pop())
  |89h: (* SUB  integer SUB *) i:=Pop(); Push(Pop()-i);
  |8Ah: (* MUL  integer MUL *) Push(Pop()*Pop())
  |8Bh: (* DIV  integer DIV *) i:=Pop(); Push(Pop() DIV i)
  |8Ch: (* SHL  integer SHift Left *)
        i:=Pop() MOD 20h; Push(SHL(Pop(),i))
  |8Dh: (* SHR  integer SHift Right *)
        i:=Pop() MOD 20h; Push(SHR(Pop(),i))

  |8Eh: (* ROL  word ROtate Left  *)
        i:=Pop() MOD 20h; Push(ROL(Pop(),i))
  |8Fh: (* ROR  word ROtate Right *)
        i:=Pop() MOD 20h; Push(ROR(Pop(),i))
  |90h..97h: (* io section *)
        CASE cpu OF
          |KRONOS2_2: ioP2_2
          |KRONOS2_5: ioP2_5
          |KRONOS2_6: ioP2_6
        ELSE ASSERT(FALSE);
        END

(* In the following eight instructions the interrupts 42h or 43h
   are  raised respectively in case of overflow or order
   underflow.
*)
  |98h: (* FADD  Float ADD *) Push(REAL(Pop())+REAL(Pop()))
  |99h: (* FSUB  Float SUB *) X:=REAL(Pop()); Push(REAL(Pop())-X)
  |9Ah: (* FMUL  Float MUL *) Push(REAL(Pop())*REAL(Pop()))
  |9Bh: (* FDIV  Float DIV *) X:=REAL(Pop()); Push(REAL(Pop())/X)
  |9Ch: (* FCMP  Float CoMPare *) X:=REAL(Pop()); Y:=REAL(Pop());
        IF    X<Y THEN Push(1); Push(0)
        ELSIF X>Y THEN Push(0); Push(1)
        ELSE Push(0); Push(0) END
  |9Dh: (* FABS  Float ABS *) X:=REAL(Pop());
        IF X<0.0 THEN Push(-X) ELSE Push(X) END
  |9Eh: (* FNEG  Float NEG *) Push(-REAL(Pop()))
  |9Fh: (* FFCT  Float FunCTions *) i:=Next();
        IF    i=0 THEN Push(FLOAT(INTEGER(Pop())))
        ELSIF i=1 THEN Push(TRUNC(   REAL(Pop())))
        ELSE DEC(PC); TRAP(7h) END;

 |0A0h: (* LSS  int LeSS             *) i:=Pop(); Push(Pop()<i)
 |0A1h: (* LEQ  int Less    or EQual *) i:=Pop(); Push(Pop()<=i)
 |0A2h: (* GTR  int GreaTeR          *) i:=Pop(); Push(Pop()>i)
 |0A3h: (* GEQ  int Greater or EQual *) i:=Pop(); Push(Pop()>=i)
 |0A4h: (* EQU  int EQUal     *) Push(Pop()=Pop())
 |0A5h: (* NEQ  int Not EQual *) Push(Pop()#Pop())
```

```
|0A6h: (* ABS  int ABSolute value *) Push(ABS(Pop()))
|0A7h: (* NEG  int NEGate         *) Push(-Pop())

|0A8h: (* OR   logical bit per bit OR *)
       v:=BITSET(Pop()); w:=BITSET(Pop()); Push(w+v)
|0A9h: (* AND  logical bit per bit AND *)
       v:=BITSET(Pop()); w:=BITSET(Pop()); Push(w*v)
|0AAh: (* XOR  logical bit per bit XOR *)
       v:=BITSET(Pop()); w:=BITSET(Pop()); Push(w/v)
|0ABh: (* BIC  logical bit per bit BIt Clear *)
       v:=BITSET(Pop()); w:=BITSET(Pop()); Push(w-v)
|0ACh: (* IN   membership to bitset *)
       v:=BITSET(Pop()); Push(Pop() IN v)
|0ADh: (* BIT  setBIT *) i:=Pop();
       IF (i<0) OR (i>=20h) THEN TRAP(4Ah)
       ELSE w:={}; INCL(w,i); Push(w) END

|0AEh: (* NOT  boolean NOT (not bit per bit!) *) Push(Pop()=0)
|0AFh: (* MOD  integer MODulo *) i:=Pop(); Push(Pop() MOD i)

|0B0h: (* DECS  DECriment S register (reverse to ALLOC) *)
       DEC(S,Pop())
|0B1h: (* DROP *) i:=Pop();
|0B2h: (* LODF  reLOaD expr. stack after Function return *)
       i:=Pop(); RestoreExpStack; Push(i)
|0B3h: (* STORE STORE expr. stack before function call *)
       IF S+ESdepth+1>H THEN DEC(PC); TRAP(40h)
       ELSE SaveExpStack
       END
|0B4h: (* STOFV STOre expr. stack with Formal function Value
          on top before function call (see instruction CF)
        *)
       IF S+ESdepth+2>H THEN DEC(PC); TRAP(40h)
       ELSE i:=Pop(); SaveExpStack; Core[S]:=i; INC(S) END
|0B5h: (* COPT  COPy Top of expr. stack *)
       i:=Pop(); Push(i); Push(i)
|0B6h: (* CPCOP Character array Parameter COPy *)
       i:=Pop(); (* High *) sz:=(i+4) DIV 4;
       IF S+sz>H THEN Push(i); DEC(PC); TRAP(40h)
       ELSE Core[L+Next()]:=S; adr:=Pop();
         WHILE sz>0 DO Core[S]:=Core[adr]; INC(S); INC(adr) END
       END
|0B7h: (* PCOP  structure Parameter allocate and COPy *)
       i:=Pop(); (* High *) sz:=i+1;
       IF S+sz>H THEN Push(i); DEC(PC); TRAP(40h)
       ELSE Core[L+Next()]:=S; adr:=Pop();
         WHILE sz>0 DO Core[S]:=Core[adr]; INC(S); INC(adr) END
       END
|0B8h: (* FOR1  enter  FOR statement *)
       IF S+2>H THEN DEC(PC); TRAP(40h)
       ELSE sz:=Next(); (* =0 up; #0 down *)
         hi:=Pop(); low:=Pop(); adr:=Pop(); k:=Next2()+PC;
         IF ((sz=0) & (low<=hi)) OR ((sz#0) & (low>=hi)) THEN
```

```
          Core[adr]:=low;
          Core[S]:=adr; INC(S); Core[S]:=hi; INC(S);
        ELSE (* loop isn't executed *) PC:=k
        END
      END
|0B9h: (* FOR2  end of FOR statment *)
      hi:=Core[S-1]; adr:=Core[S-2]; sz:=Next();
      IF sz>7Fh THEN sz:=7Fh-sz END; (* step [-128..127] *)
      k:=-Next2()+PC;  i:=Core[adr]+sz;
      IF ((sz>=0) & (i>hi)) OR ((sz<0) & (i<hi)) THEN
        DEC(S,2); (* terminate *)
      ELSE Core[adr]:=i; PC:=k (* continue *)
      END
|0BAh: (* ENTC  ENTer Case statment *)
      IF S+1>H THEN DEC(PC); TRAP(40h)
      ELSE PC:=Next2()+PC; (* jump to case table *)
        k:=Pop(); low:=Next2(); hi:=Next2();
        Core[S]:=PC + 2*(hi-low) + 4; INC(S); (* PC for exit *)
        IF (k>=low) & (k<=hi) THEN
          PC:=PC+2*(k-low+1) (* jump into case table *)
        END;
        PC:=-Next2()+PC (* jump back to variant's code *)
      END
|0BBh: (* XIT  eXIT from case or control structure *)
      DEC(S); PC:=Core[S]
|0BCh: (* ENTS ENTer control Structure *)
      IF S+1>H THEN DEC(PC); TRAP(40h)
      ELSE Core[S]:=Next2()+PC; INC(S) END
|0BEh: (* ORJP   short circuit OR  JumP *)
      IF Pop()#0 THEN Push(1); PC:=Next()+PC
      ELSE INC(PC) END
|0BFh: (* ANDJP  short circuit AND JumP *)
      IF Pop()=0 THEN Push(0); PC:=Next()+PC
      ELSE INC(PC) END

|0C0h: (* MOVE   MOVE block *) sz:=Pop();
      i:=Pop(); j:=Pop();
      WHILE sz>0 DO
        Core[j]:=Core[i]; INC(i); INC(j); DEC(sz)
      END
|0C1h: (* RDS  ReaD String *) sz:=Next();
      IF sz>20h THEN DEC(PC,2); TRAP(4Bh)
      ELSE adr:=Pop();
        WHILE sz>0 DO Core[adr]:=Next4(); INC(adr); DEC(sz) END
      END
|0C2h: (* LSTA  Load STring Address *) Push(Core[G+1]+Next2());
|0C3h: (* COMP  COMPare strings *) i:=Pop()*4; j:=Pop()*4;
      REPEAT a:=CHAR(ByteCore[i]); b:=CHAR(ByteCore[j]);
      INC(i); INC(j)
      UNTIL (a=0c) OR (b=0c) OR (a#b); Push(a); Push(b)
|0C4h: (* GB  Get procedure Base n levels down *)
      i:=L; k:=Next();
      WHILE k>0 DO i:=Core[i]; DEC(k) END; Push(i)
```

```
 |0C5h: (* GB1 Get procedure Base 1 level down *) Push(Core[L])
 |0C6h: (* CHK   range bounds CHecK *)
        hi:=Pop(); low:=Pop(); i:=Pop(); Push(i);
        IF (i<low) OR (i>hi) THEN
          Push(low); Push(hi); TRAP(4Ah)
        END
 |0C7h: (* CHKZ  array bounds CHecK (low=Zero) *)
        hi:=Pop(); i:=Pop(); Push(i);
        IF (i<0) OR (i>hi) THEN Push(hi); TRAP(4Ah) END
 |0C8h: (* ALLOC ALLOCate block *) sz:=Pop();
        IF S+sz>H THEN  Push(sz); DEC(PC); TRAP(40h)
        ELSE Push(S); INC(S,sz) END
 |0C9h: (* ENTR  ENTeR procedure *) sz:=Next();
        IF S+sz>H THEN DEC(PC,2); TRAP(40h)
        ELSE INC(S,sz) END
 |0CAh: (* RTN   ReTurN from procedure *)
        S:=L; L:=Core[S+1]; PC:=WORD(BITSET(Core[S+2])*{0..0Fh});
        IF ExternalBit IN BITSET(Core[S+2]) THEN
          (* external called *)
          G:=Core[S]; F:=CodePtr(Core[G])
        END;
        IF ChangeMaskBit IN BITSET(Core[S+2]) THEN
          (* mask was changed *)
          M:=BITSET(Core[S+3])*{0..10h}
        END;
 |0CBh: (* NOP   No OPeration  *)
 |0CCh: (* CX    Call eXternal *)
        IF S+4<=H THEN j:=Core[G-Next()-1]; (* big DFT *)
          i:=Next(); Mark(G,TRUE);
          G:=Core[j]; F:=CodePtr(Core[G]); PC:=GetPc(i);
        ELSE DEC(PC); TRAP(40h) END
 |0CDh: (* CI    Call procedure at Intermediate level *)
        IF S+4<=H THEN
          i:=Next(); Mark(Pop(),FALSE); PC:=GetPc(i);
        ELSE DEC(PC); TRAP(40h) END
 |0CEh: (* CF    Call Formal procedure *)
        IF S+3<=H THEN i:=Core[S-1]; DEC(S); Mark(G,TRUE);
          k:=i DIV 1000000h; i:=i MOD 1000000h;
          G:=Core[i]; F:=CodePtr(Core[G]); PC:=GetPc(k);
        ELSE DEC(PC); TRAP(40h) END
 |0CFh: (* CLN
0CПo LCal procedure *)
        IF S+4<=H THEN i:=Next(); Mark(L,FALSE); PC:=GetPc(i);
        ELSE DEC(PC); TRAP(40h) END
 |0D0h..0DFh: (* CL0..CL0F  Call Local procedure *)
        IF S+4<=H THEN Mark(L,FALSE); PC:=GetPc(IR MOD 10h);
        ELSE DEC(PC); TRAP(40h) END
 |0E0h: (* INCL  INCLude in set *)  i:=Pop();
        IF (i<0) OR (i>1Fh) THEN Push(i); DEC(PC); TRAP(4Ah)
        ELSE j:=Pop(); w:=BITSET(Core[j]); INCL(w,i);
          Core[j]:=CARDINAL(w)
        END
 |0E1h: (* EXCL  EXCLude from set *)  i:=Pop();
```

```
        IF (i<0) OR (i>1Fh) THEN Push(i); DEC(PC); TRAP(4Ah)
        ELSE j:=Pop(); w:=BITSET(Core[j]); EXCL(w,i);
          Core[j]:=CARDINAL(w)
        END
 |0E2h: (* SLEQ  bitSet Less    or EQual *)
        w:=BITSET(Pop()); v:=BITSET(Pop()); Push(v<=w)
 |0E3h: (* SGEQ  bitSet Greater or EQual *)
        w:=BITSET(Pop()); v:=BITSET(Pop()); Push(v>=w)
 |0E4h: (* INC1  INCrement by 1 *) INC(Core[Pop()])
 |0E5h: (* DEC1  DECrement by 1 *) DEC(Core[Pop()])
 |0E6h: (* INC   INCrement *) i:=Pop(); INC(Core[Pop()],i)
 |0E7h: ( *)
        IF S+1>H THEN DEC(PC); TRAP(40h)
        ELSE Core[S]:=Pop(); INC(S)
        END
 |0E9h: (* LODT  LOaD   Top of proc stack *)
        DEC(S); Push(Core[S])
 |0EAh: (* LXA   Load indeXed Address *)
        sz:=Pop(); i:=Pop(); adr:=Pop(); Push(adr+i*sz)
 |0EBh: (* LPC   Load Procedure Constant *)
        i:=Next(); j:=Next(); Push(j*1000000h+Core[G-i-1])

(* The  following  3  instructions  deal  with  bit  slices. Bit
   address  is the pair (address, bit offset). Bit offset can be
   greater than 32. The slice may be out of word bounds.
*)
 |0ECh: (* BBU  Bit Block Unpack *)
        sz:=Pop();
        IF (sz<1) OR (sz>32) THEN
          Push(sz); DEC(PC); TRAP(4Ah)
        END;
        i:=Pop(); adr:=Pop();
        (* j  is  a  bit  slice  of  size -sz- beginning from bit
           address (adr,i).
        *)
        Push(j);
 |0EDh: (* BBP  Bit Block Pack *)
        j:=Pop(); sz:=Pop();
        IF (sz<1) OR (sz>32) THEN
          Push(sz); DEC(PC); TRAP(4Ah)
        END;
        i:=Pop(); adr:=Pop();
        (* Packing  up -sz- of least significant bits from j and
           put them to address (adr,i).
        *)
 |0EEh: (* BBLT Bit BLock Transfer *)
        sz:=Pop(); (* size can be greater than 32 *)
        i:=Pop(); adr:=Pop();
        j:=Pop(); adr1:=Pop();
        (* Copies -sz- bits from (adr,i) to (adr1,j) *)
 |0F0h: (* SWAP *)
        i:=Pop(); j:=Pop(); Push(i); Push(j)
 |0F1h: (* LPA Load Parameter Address *)
```

```
        Push(L-Next()-1);
 |0F2h: (* LPW Load Parameter WORD *)
        Push(Core[L-Next()-1]);
 |0F3h: (* SPW Store Parameter WORD *)
        Core[L-Next()-1]:=Pop();
 |0F4h: (* SSWU Store Stack Word Undestructive *)
        i:=Pop(); Core[Pop()]:=i; Push(i)
 |0FAh: (* ACTIVe process *) Push(P)
 |0FBh: (* USR User defined functions *) i:=Next(); (*  *)
 |0FCh: (* SYS rare SYStem functions *)
        CASE Next() OF
         |00h: (* PID Processor IDent *)
               (* Push(PID) *)
         |01h..0FFh: (* depends on a processor model *)
        END;
 |0FDh: (* NII Never Implemented Instruction *) TRAP(7h);
 |0FFh: (* INVLD INVaLiD operation *)          TRAP(49h)
  ELSE  (* unimplemented instruction *) TRAP(7h)
  END (*CASE*)
END Interpret;

BEGIN (* "Power On" *)
 (* cpu:=KRONOS2_2 | KRONOS2_5 | KRONOS2_6 *)
 ORIGIN(ByteCore,ADR(Core),SIZE(Core));
 ConsolMicroProgram;
 LOOP
   IF Ipt THEN TRAP(IptNo) END;
   IR:=Next();
   Interpret;
 END (*LOOP*)
END Kronos_Interpreter.
```

GENERAL DESCRIPTION OF KRONOS INSTRUCTIONS

This description is not an independent document. It must be perceived as the detailed, vast comments for "Kronos Interpreter". Definitions of the main concepts of Kronos acrhitecture are omitted here. They may be found in the section "VIRTUAL MODULA-2 MACHINE".

The execution of any instruction is starting from fetching its code. Instruction code is 8 bit width and treated as unique code of instruction. It restrictly determines the algorithm of instruction execution. There are not any fields or addressation modes packed in the instruction code. Instruction code is a byte and nothing more. In rare cases the group of instructions may have the same code (the so-named escape instructions). In such cases (as, for example FFCT) the next byte determines which instruction of the group will be executed.

After instruction code fetching, the algorithm of its execution is detemined. Some operands of the instruction layed on Expression Stack (ES), the other (if necessary) may follow the instruction code, as a sequence of bytes. The last ones are named immediately (constant) operands, but they are never absolute addresses. Some notions are necessary for the instruction description (see, also, "Kronos Interpreter").

Code Fetching

Code fetching is performed by the procedures "Next", "Next2" and "Next4". They are fetching a byte, two bytes, or a word (four bytes), respectively, and increase the program counter (PC) by 1,2 or 4. In terms of Modula-2 these procedures may be written:

```
PROCEDURE Next(): BYTE;
BEGIN INC(PC); RETURN INTEGER(F^[PC-1]) END Next;

PROCEDURE Next2(): WORD16;
BEGIN RETURN Next()+Next()*100h END Next2;

PROCEDURE Next4(): WORD;
BEGIN RETURN Next2()+Next2()*10000h END Next4;
```

Expression Stack

Expression stack is used for evaluation expressions and manipulations with instruction arguments. It executes two operations "Pop" (take from the stack) and "Push" (put into

the stack):

```
PROCEDURE Push(X: WORD);
BEGIN
  A[sp]:=X;
  IF sp<ESdepth THEN INC(sp) ELSE Ipt:=TRUE; IptNo:=4Ch END;
END Push;

PROCEDURE Pop(): INTEGER;
BEGIN
  IF sp=0 THEN Ipt:=TRUE; IptNo:=4Ch ELSE DEC(sp) END;
  RETURN A[sp];
END Pop;
```

If the stack is empty when "Pop" is executed or full before "Push" trap 4Ch is raised (this useful feature is added for compiler debugging).

Words "pop from the stack" denote following: "take the value of the top element of the stack and then decrement the stack counter by 1".

Words "push into the stack" denotes following: "put the value above the top element of the stack and then increment the stack counter by 1".

By default, the word "stack" without any prefix denotes "expression stack".

4-bits Immediate Operands

Since Kronos instruction set is very small (less than 128 instructions) it was possible to use 4 least significant bits in some frequently executed instructions for short (4 bits) form of offset. These instructions may be described as a set of 16 different instructions (in "Kronos Interpreter") or as a single 4 bit instruction with 4 bit the immediately followed operand (in this DESCRIPTION).

Note.
All numbers used in the text are hexidecimal, and so are postfixed by character 'h' - 'hex'.

LI0..LI0F
Load Immediate

```
Operation Code:                 4 bits          0h
Immediate Operands:             4 bits          00h..0Fh
Instruction Length:             1 byte
Action:
     Loads the value of the least significant 4 bits of
     instruction code (in the range 00h..0Fh), and pushes it
     into the stack as 32 bit words with leadings zeros. PC
     increments by 1.

Interpreter:
     Push(IR MOD 10h);
```

LIB
Load Immediate Byte

```
Operation Code:                 1 byte          10h
Immediate Operands:             1 byte          00h..0FFh
Instruction Length:             2 байта
Action:
     Loads the value of the immediate operand (byte following
     the instruction code) in the range 0..0FFh, and pushes it
     into the stack as 32 bit words with leadings zeros. PC
     increments by 2.

Interpreter:
     Push(Next())
```

LID
Load Immediate Double Byte

```
Operation Code:                 1 byte          11h
Immediate Operands:             2 байта         00h..0FFFFh
Instruction Length:             3 bytes
Action:
     Loads the value of the immediate operand (2 bytes
     following the instruction code) in the range 0..0FFFFh,
     and pushes it into the stack as 32 bit words with
     leadings zeros. PC increments by 3.

Interpreter:
     Push(Next2())
```

LIW
Load Immediate Word

```
Operation Code:                 1 byte          12h
Immediate Operands:             4 байта         00h..0FFFFFFFFh
```

Instruction Length: 5 байт
Action:
Loads the value of the immediate operand (4 bytes following the instruction code), and pushes it into the stack as 32 bit words. PC increments by 5.

Interpreter:
Push(Next4())

Note:
Kronos used complement to 2**32 representaion for negative integer numbers. The integer numbers lies in the range:

```
MIN(INTEGER)=-2**31 .. MAX(INTEGER)=2**31-1
     80000000h      ..      7FFFFFFFh
```

"-1" reprsents as 0FFFFFFFFh.

Only the instruction LIW allows one to push negative numbers into the stack. But there exists the other method to generate small negative numbers by loading of its absolute value using LI,LIB,LID and then change a sign by the instruction NEG. For current models of processors it is more effective.

LIN
Load Immediate Nil

Operation Code: 1 byte 13h
Immediate Operands: none
Instruction Length: 1 byte
Action:
Loads the non-existent address (NIL) and pushes it into the stack. PC increments by 1. The NIL value is specific and has been changed from one processor model to another, without the influence on program portability.

Interpreter:
Push(NIL)

LLA
Load Local Address

Operation Code: 1 byte 14h
Immediate Operands: 1 byte 00h..0FFh
Instruction Length: 2 bytes
Action:
Adds contents of L_register and the immediate operands (byte following the instruction code) and pushes the sum into the stack. PC increments by 2. This instruction

allows one to accept the address of a local variable at the top of the stack. When offset of a variable is greater than 0FFh, the problem is solved by combining LLA, LI,LIB,LID or LIW and ADD instructions.

Interpreter:

Push(L+Next())

LGA

Load Global Address

Operation Code:	1 byte	15h
Immediate Operands:	1 byte	00h..0FFh
Instruction Length:	2 байта	

Action:

Adds contents of the G_register and the immediate operands (byte following the instruction code) and pushes the sum into the stack. PC increments by 2. This instruction allows one to accept the address of a global variable at the top of the stack. When offset of a variable is greater than 0FFh, the problem is solved by combining LLA, LI,LIB,LID or LIW and ADD instructions.

Interpreter:

Push(G+Next())

LSA 16h

Load Stack Address

Operation Code:	1 byte	16h
Immediate Operands:	1 byte	00h..0FFh
Instruction Length:	2 байта	

Action:

Adds contents of the top of the stack and the immediate operands (byte following the instruction code), pops the top of the stack and pushes the sum into the stack. PC increments by 2. This instruction allows one to overhead the address layed at the top of the stack by address incremented by offset. It is used for accepting addresses of fields when record address has been already loaded into the stack. When offset of the field is greater than 0FFh, the problem is solved by combining LI,LIB,LID or LIW and ADD instructions.

Note:

LSA XX is semantically equivalent to the sequence LIB XX ADD. In the RISC model of Kronos this instruction will be discarded.

Interpreter:

Push(Pop()+Next())

LEA
Load External Address

```
Operation Code:                 1 byte          17h
Immediate Operands:             2 байта         00h..0FFh
Instruction Length:             3 bytes
```
Action:

The first the immediate operand (byte following the instruction code) is the module number (M) and the next one is the word offset (N). Loads the memory word at the address G_register-M-1. It is the pointer to module global area table (the so-called DFT), and after dereferencing (see Note), it (EA) points to the global area of the module M. Pushes the sum of N and EA into the stack. PC increments by 3.

Note:

In the RISC model of Kronos processor external dereferensing will be discarded.

Interpreter:

```
i:=G-Next()-1; (* index in local DFT of current module *)
adr:=Core[i];  (* pointer to DFT                       *)
Push(Core[adr]+Next())
```

For RISC model:

```
i:=G-Next()-1; (* index in local DFT of current module *)
adr:=Core[i];  (* pointer to G-Area of External Module *)
Push(adr+Next())
```

JFLC
Jump Forward Long Condition

```
Operation Code:                 1 byte          18h
Immediate Operands:             2 bytes         00h..0FFFFh
Instruction Length:             3 bytes
```
Action:

Takes 2 byte immediate operand - the potential offset to the next instruction (OFFSET). PC increments by 3. Pops the top of the stack and if it is zero the PC increments by OFFSET, otherwise (any nonzero value) goes to the next instruction.

Interpreter:

```
IF Pop()=0 THEN PC:=Next2()+PC
ELSE INC(PC,2) END
```

JFL
Jump Forward Long

```
Operation Code:                 1 byte          19h
Immediate Operands:             2 bytes         00h..0FFFFh
Instruction Length:             3 bytes
Action:
     Takes  2  byte immediate operand - the offset to the next
     instruction (OFFSET). PC increments by 3+OFFSET.

Interpreter:
     PC:=Next2()+PC;
```

JFSC

Jump Forward Short Condition

```
Operation Code:                 1 byte          1Ah
Immediate Operands:             1 byte          00h..0FFh
Instruction Length:             2 bytes
Action:
     Takes  1 byte immediate operand - the potential offset to
     the  next  instruction (OFFSET). PC increments by 2. Pops
     the  top  of the stack and if it is zero PC increments by
     OFFSET,  otherwise  (any  nonzero value) goes to the next
     instruction.

Interpreter:
     IF Pop()=0 THEN PC:=Next()+PC
     ELSE INC(PC) END
```

JFS

Jump Forward Short

```
Operation Code:                 1 byte          1Bh
Immediate Operands:             1 byte          00h..0FFh
Instruction Length:             2 bytes
Action:
     Takes  1  byte immediate operand - the offset to the next
     instruction (OFFSET). PC increments by 2+OFFSET.

Interpreter:
     PC:=Next()+PC;
```

JBLC

Jump Back Long Condition

```
Operation Code:                 1 byte          1Ch
Immediate Operands:             2 bytes         00h..0FFFFh
Instruction Length:             3 bytes
Action:
     Takes  2 byte immediate operand - the potential offset to
     the  next  instruction (OFFSET). PC increments by 3. Pops
     the  top of the stack and if it is zero the PC decrements
```

by OFFSET, otherwise (any nonzero value) goes to the next instruction.

```
Interpreter:
     IF Pop()=0 THEN PC:=-Next2()+PC
     ELSE INC(PC,2) END
```

JBL

Jump Back Long

```
Operation Code:                 1 byte          1Dh
Immediate Operands:             2 bytes         00h..0FFFFh
Instruction Length:             3 bytes
```

Action:

Takes 2 byte immediate operand - the offset to the next instruction (OFFSET). PC decrements by OFFSET-3.

```
Interpreter:
     PC:=-Next2()+PC;
```

JBSC

Jump Back Short Condition

```
Operation Code:                 1 byte          1Eh
Immediate Operands:             1 byte          00h..0FFh
Instruction Length:             2 bytes
```

Action:

Takes 1 byte immediate operand - the potential offset to the next instruction (OFFSET). PC increments by 2. Pops the top of the stack and if it is zero the PC decrements by OFFSET, otherwise (any nonzero value) goes to the next instruction.

```
Interpreter:
     IF Pop()=0 THEN PC:=-Next()+PC
     ELSE INC(PC) END
```

JBS

Jump Back Short

```
Operation Code:                 1 byte          1Fh
Immediate Operands:             1 byte          00h..0FFh
Instruction Length:             2 bytes
```

Action:

Takes 1 byte immediate operand - the offset to the next instruction (OFFSET). PC decrements by OFFSET-2.

```
Interpreter:
     PC:=-Next()+PC;
```

LLW
Load Local Word

```
Operation Code:                 1 byte          20h
Immediate Operands:             1 byte          00h..0FFh
Instruction Length:             2 bytes
```
Action:

Evaluates EA as a sum of the L_register and the immediate operand. Loads a word from EA memory location and pushes it into the stack. PC increments by 2. When offset is greater than 0FFh, the problem may be solved by the combination of instructions LLA 00, LID|LIW offs, ADD, LSW0.

Interpreter:

```
Push(Core[L+Next()])
```

LGW
Load Global Word

```
Operation Code:                 1 byte          21h
Immediate Operands:             1 byte          00h..0FFh
Instruction Length:             2 bytes
```
Action:

Evaluates EA as a sum of the G_register and the immediate operand. Loads a word from EA memory location and pushes it into the stack. PC increments by 2. When offset is greater than 0FFh, the problem may be solved by combination of instructions LGA 00, LID|LIW offs, ADD, LSW0.

Interpreter:

```
Push(Core[G+Next()])
```

LEW
Load External Word

```
Operation Code:                 1 byte          22h
Immediate Operands:             2 bytes         00h..0FFFFh
Instruction Length:             3 bytes
```
Action:

The first immediate operand (byte following the instruction code) is the module number (M) and next one is the word offset (N). Loads the memory word at the address G_register-M-1. It is the pointer to module global area table (the so-called DFT), and after dereferencing (see Note for LEA) it (EA) points to the global area of the module M. Adds N to EA, loads the word from memory addressed by EA and pushes it into the stack. PC increments by 3.

```
Interpreter:
     i:=G-Next()-1; adr:=Core[Core[i]]; (* external G *)
     Push(Core[adr+Next()])
```

LSW 23h

Load Stack addressed Word

Operation Code:	1 byte	23h
Immediate Operands:	1 byte	00h..0FFh
Instruction Length:	2 bytes	

Action:

Evaluates EA as a sum of contents of top of the stack and the immediate operand (byte following the instruction code). Pops the top of the stack and pushes a word loaded from EA memory location into the stack. PC increments by 2.

```
Interpreter:
     Push(Core[Pop()+Next()])
```

LLW4..LLW0F

Load Local Word

Operation Code:	4 bits	2Xh
Immediate Operands:	4 bits	04h..0Fh
Instruction Length:	1 byte	

Action:

Evaluates EA as a sum of the L_register and the immediate 4 bit operand (the value of the least significant 4 bits of instruction code). Loads a word from EA memory location and pushes it into the stack. PC increments by 1.

Note:

The first 4 words of the local area are used for call/return information, and can not be used for local variables.

```
Interpreter:
     Push(Core[L+IR MOD 10h])
```

SLW

Store Local Word

Operation Code:	1 byte	30h
Immediate Operands:	1 byte	00h..0FFh
Instruction Length:	2 bytes	

Action:

Evaluates EA as a sum of the L_register and the immediate byte operand. Pops a word from the stack and stores it at EA memory location. PC increments by 2.

```
Interpreter:
     Core[L+Next()]:=Pop()
```

SGW 31h

Store Global Word

```
Operation Code:                 1 byte          31h
Immediate Operands:             1 byte          00h..0FFh
Instruction Length:             2 bytes
Action:
```

Evaluates EA as a sum of G_register and the immediate byte operand. Pops a word from the stack and stores it at EA memory location. PC increments by 2.

```
Interpreter:
     Core[G+Next()]:=Pop()
```

SEW

Store External Word

```
Operation Code:                 1 byte          32h
Immediate Operands:             2 bytes         00h..0FFh
Instruction Length:             3 bytes
Action:
```

Similar to LEW, but store the popped word in memory.

```
Interpreter:
     i:=G-Next()-1; adr:=Core[Core[i]]; (* external G *)
     Core[adr+Next()]:=Pop()
```

SSW

Store Stack addressed Word

```
Operation Code:                 1 byte
Immediate Operands:             1 byte
Instruction Length:             2 bytes
Action:
```

Pops the value from the stack. Evaluates EA as a sum of the address popped from the stack and the immediate operand (byte following the instruction code). Stores the value at EA memory location. PC increments by 2.

```
Interpreter:
     i:=Pop(); Core[Pop()+Next()]:=i
```

SLW4..SLW0F

Store Local Word

```
Operation Code:                 4 bits          3Xh
```

```
Immediate Operands:             4 bits          04h..0Fh
Instruction Length:             1 byte
Action:
     Evaluates EA as a sum of the L_register and the immediate
     4  bit operand (the value of the least significant 4 bits
     of the instruction code). Stores the word popped from the
     stack at EA memory location. PC increments by 1.

Interpreter:
     Core[L+IR MOD 10h]:=Pop()
```

LXB

Load indeXed Byte

```
Operation Code:                 1 byte          40h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Pops  the  index  (the upper one) and the base address of
     the  byte  array  from  the  stack.  Loads  addressed  by
     base+index byte from memory, and pushes it into the stack
     as  32 bit word extending by leading zeros. PC increments
     by 1.

Interpreter:
     i:=Pop(); Push(ByteCore[Pop()*4+i]);
```

LXW

Load indeXed Word

```
Operation Code:                 1 byte          41h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Pops  the  index  (the upper one) and the base address of
     the  word  array  from  the  stack.  Loads  addressed  by
     base+index  word  from  memory,  and  pushes  it into the
     stack.  PC increments by 1.

Interpreter:
     i:=Pop(); Push(Core[Pop()+i])
```

LGW2..LGW0F

Load Global Word

```
Operation Code:                 4 bits          4*h
Immediate Operands:             4 bits          02h..0Fh
Instruction Length:             1 byte
Action:
     Evaluates EA as a sum of the G_register and the immediate
```

4 bit operand (the value of the least significant 4 bits of the instruction code). Loads a word from EA memory location and pushes it into the stack. PC increments by 1.

Note:
The first 2 words of the global area of module are used for pointers to code and constant segments, and can not be used for global variables.

Interpreter:
```
Push(Core[G+IR MOD 10h])
```

SXB
Store indeXed Byte

Operation Code:	1 byte	50h
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:
Pops the value (the upper one) from the stack and truncate it to the least significant byte. Then pops the index and the base address of the byte array from the stack. Store the value at memory location addressed by base+index byte. PC increments by 1.

Interpreter:
```
j:=Pop(); i:=Pop(); ByteCore[Pop()*4+i]:=j;
```

SXW
Store indeXed Word

Operation Code:	1 byte	51h
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:
Pops the value (the upper one) from the stack. Then pops the index and the base address of the word array from the stack. Store the value at memory location addressed by base+index word. PC increments by 1.

Interpreter:
```
j:=Pop(); i:=Pop(); Core[Pop()+i]:=j
```

SGW2..SGW0F
Store Global Word

Operation Code:	4 bits	5*h
Immediate Operands:	4 bits	02h..0Fh
Instruction Length:	1 byte	

Action:
Evaluates EA as a sum of the G_register and the immediate 4 bit operand (the value of the least significant 4 bits of the instruction code). Stores word popped from the stack at EA memory location. PC increments by 1.

Interpreter:
```
Core[G+IR MOD 10h]:=Pop()
```

LSW0..LSW0F
Load Stack addressed Word

Operation Code:	4 bits	6*h
Immediate Operands:	4 bits	00h..0Fh
Instruction Length:	1 byte	

Action:
Evaluates EA as a sum of popped from the stack address and the immediate 4 bit operand (the value of the least significant 4 bits of the instruction code). Loads the word from EA memory location and pushes it into the stack. PC increments by 1.

Interpreter:
```
Push(Core[Pop()+IR MOD 10h])
```

SSW0..SSW0F
Store Stack addressed Word

Operation Code:	4 bits	7*h
Immediate Operands:	4 bits	00h..0Fh
Instruction Length:	1 byte	

Action:
Pops the value from the stack. Evaluates EA as a sum of popped from the stack address and the immediate 4 bit operand (the value of the least significant 4 bits of the instruction code). Stores the value at EA memory location. PC increments by 1.

Interpreter:
```
i:=Pop(); Core[Pop()+IR MOD 10h]:=i
```

QUIT
stop processor

Operation Code:	1 byte	81h
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:
PC increments by 1. If bit 2 in the M_register (Mask) ON then TRAP(2), otherwise stops processor. If console

microprogram is implemented then control transfers to it.

Note:
In the RISC model of Kronos the other trap subsystem will be implemented and QUIT will be a privileged instruction.

Interpreter:
ConsolMicroProgram

GETM
GET Mask

Operation Code:	1 byte	82h
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:
Loads M_register and pushes it into the stack. PC increments by 1.

Note:
In the RISC model of Kronos the other trap subsystem will be implemented. Actions of GETM & SETM will be changed and they will be privileged instructions.

Interpreter:
Push(M)

SETM
SET Mask

Operation Code:	1 byte	83h
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:
PC increments by 1. If this is the first change of M_register in this procedure (30-th bit is OFF in the local word 02h) then the current contents of M_register is saved in the local word 03h and 30-th bit is set in the local word 02h. (Instruction RTN will restore the old mask after return.) A new mask value is popped from the stack and put into M_register.

Interpreter:

```
IF NOT (ChangeMaskBit IN BITSET(Core[L+2])) THEN
  (* Mask is changed first time *)
  Core[L+2]:=WORD(BITSET(Core[L+2])+{ChangeMaskBit});
  Core[L+3]:=WORD(M)
END; M:=BITSET(Pop);
```

TRAP
interrupt simulation

Operation Code: 1 byte 84h
Immediate Operands: none
Instruction Length: 1 byte
Action:
PC increments by 1. Pops the 32 bit trap number from the stack and raises trap with this number if it is enabled.

Interpreter:
TRAP(Pop())

TRA
TRAnsfer control between processes

Operation Code: 1 byte 85h
Immediate Operands: none
Instruction Length: 1 byte
Action:
PC increments by 1.
The whole contents of the process is determined by the whole contents of the processor: registers, expression the stack, and the state of the process memory (P-stack is the stack for procedure calls). The whole contents of the processor is saved at P-stack and into the process descriptor (see "Kronos Interpreter"). The address of a new pointer of the process descriptor pops from the expression the stack, then old pointer to the process descriptor (P_register of the processor) is saved at the next popped address, and then P_register is changed by the value loaded from memory at new process descriptor pointer location. After that all the registers are restored from the new process descriptor, and the expression stack is restored from new P-stack, and control transfers to the current instruction of a new process.

Interpreter:
i:=Pop(); Transfer(Pop(),i)

TR
Test & Reset

Operation Code: 1 byte 86h
Immediate Operands: none
Instruction Length: 1 byte
Action:
PC increments by 1. Pops the EA from the stack. LOCKs memory to the processor bus. Loads the value from the memory location EA and writes 0 to memory. Pushes the value into the stack. UNLOCKs memory to the processor

bus. May be used for an exclusive access to the data from several processors.

Interpreter:

```
i:=Pop(); Push(Core[i]); Core[i]:=0
```

IDLE

	IDLE process	
Operation Code:	1 byte	87h
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:

PC is not changed. Processor waits for the interrupt not Busying the bus. The process wich executes this Instruction will be never continued. After the interrupt service and continued it it will execute IDLE again.

Interpreter:

```
DEC(PC); REPEAT (* not busy the bus *) UNTIL Ipt
```

ADD

integer ADDition

Operation Code:	1 byte	88h
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:

PC increments by 1. Pops two words from the stack, adds them and pushes the result into the stack. If integer overflow happens and 31-t bit in the M_register is ON, raises trap 41h, elsif it is OFF, puts 41h in 6-th word of the process descriptor. The result after integer overflow is undefined. It may not be the module 2**32 of addition!

Note:

All instructions of integer arithmetics have such conventions about interrupt raising and result after overflow.

Interpreter:

```
Push(Pop()+Pop())
```

SUB

integer SUBtraction

Operation Code:	1 byte	89h
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:
PC increments by 1. Pops two words from the stack, substracts the upper one from the lower one and pushes the result into the stack (see, also, ADD).
Interpreter:
```
i:=Pop(); Push(Pop()-i);
```

MUL
integer MULtiplication

Operation Code:	1 byte	8Ah
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:
PC increments by 1. Pops two words from the stack, multiplies them and pushes the result into the stack (see, also, ADD).

Interpreter:
```
Push(Pop()*Pop())
```

DIV
integer DIVision

Operation Code:	1 byte	8Bh
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:
PC increments by 1. Pops two words from the stack, divides the lower one by the upper one and pushes the result into the stack (see also ADD).

Interpreter:
```
i:=Pop(); Push(Pop() DIV i)
```

SHL
integer SHift Left

Operation Code:	1 byte	8Ch
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:
PC increments by 1. Pops two words from the stack and shifts the lower one arithmetically left by the upper one and pushes the result into the stack (see also ADD). SHL(x,n) is semantically equivalent to MULT(x,2**n) and so the rules for interrupt raising are the same.

Interpreter:
```
i:=Pop() MOD 20h; Push(SHL(Pop(),i))
```

SHR
integer SHift Right

```
Operation Code:                 1 byte          8Dh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     PC  increments  by  1.  Pops two words from the stack and
     shifts the lower one arithmeticaly right by the upper one
     and  pushes  the  result  into  the stack (see also ADD).
     SHR(x,n) is semantically equivalent to DIV(x,2**n) and so
     the rules for interrupt raising are the same.

Note:
     SHR(-1,1) = -1 DIV 2 = 0 !!!
Interpreter:
     i:=Pop() MOD 20h; Push(SHR(Pop(),i))
```

ROL
word ROate Left

```
Operation Code:                 1 byte          8Eh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     PC increments by 1. Pops the shift count and the shifting
     value  from  the  stack.  Rotates  left  the  value until
     count=0  and  then  pushes  the  result  into  the stack.
     Overflow never happens. 31 bit goes directly to 0 bit.
Interpreter:
     i:=Pop() MOD 20h; Push(Pop()<<i)
```

ROR
word ROate Right

```
Operation Code:                 1 byte          8Fh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     PC increments by 1. Pops the shift count and the shifting
     value  from  the  stack.  Rotates  right  the value until
     count=0  and  then  pushes  the  result  into  the stack.
     Overflow never happens. 0 bit goes directly to 31 bit.
Interpreter:
     i:=Pop() MOD 20h; Push(Pop()>>i)
```

IO0..IO7
Input-Output

```
Operation Code:                 1 byte          90h..97h
```

```
Immediate Operands:             ?
Instruction Length:             ?
Action:
     Instruction  for  I/O  bus communications. Individual for
     each model of processor and I/O bus.

Interpreter:
     CASE cpu OF
       |Kronos2_2: ioP2_2
       |Kronos2_5: ioP2_5
       |Kronos2_6: ioP2_6
     ELSE ASSERT(FALSE);
     END
```

FADD

Float ADDition

```
Operation Code:                 1 byte          98h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     PC  increments  by  1.  Pops  two  32 bits words from the
     stack,  adds them as 32-bits float numbers and pushes the
     result   into   the   stack.  If  real  overflow/underflow
     happened  and  31 bit in the M_register is ON, raise trap
     42h/43h,  elsif  it  is  OFF  puts  42h/43h  in word 6 of
     process descriptor. The result after the real overflow is
     undefined, after underflow 0.0e+00.
Note:
     Overflow/Underflow  discipline  is  common  for all FLOAT
     instructions. Current models of processors use 32-bit DEC
     (tm) real numbers representation.

Interpreter:
     Push(REAL(Pop())+REAL(Pop()))
```

FSUB

Float SUBtraction

```
Operation Code:                 1 byte          99h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     PC  increments  by  1.  Pops  two  32 bits words from the
     stack,  substracts  the  upper  one from the lower one as
     32-bits  float  numbers  and  pushes  the result into the
     stack.

Interpreter:
     X:=REAL(Pop()); Push(REAL(Pop())-X)
```

FMUL
Float MULtiplication

```
Operation Code:                 1 byte          9Ah
Immediate Operands:               none
Instruction Length:             1 byte
```
Action:

PC increments by 1. Pops two 32 bits words from the stack, multiplies them as 32-bits float numbers and pushes the result into the stack.

Interpreter:

```
Push(REAL(Pop())*REAL(Pop()))
```

FDIV
Float DIVision

```
Operation Code:                 1 byte          9Bh
Immediate Operands:               none
Instruction Length:             1 byte
```
Action:

PC increments by 1. Pops two 32 bits words from the stack, divides the lower one by the upper one as 32-bits float numbers and pushes the result into the stack.

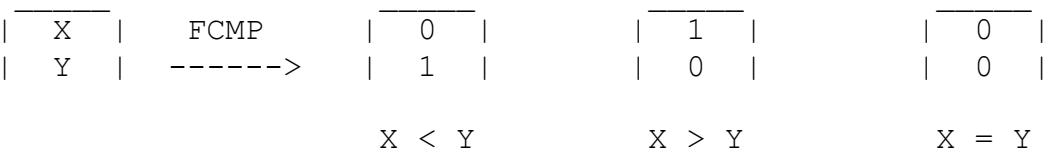
Interpreter:

```
X:=REAL(Pop()); Push(REAL(Pop())/X)
```

FCMP
Float CoMPare

```
Operation Code:                 1 byte          9Ch
Immediate Operands:               none
Instruction Length:             1 byte
```
Action:

PC increments by 1.
Pops Rigth and Left real values from the stack. Compares them as a real numbers and pushes the results. At the place of the greatest value is 1, at other is 0. If they equals pushes 0,0 pair.



Pair of instructions:

```
          FCMP EQU        FCMP NEQ
          FCMP LSS        FCMP GTR
          FCMP LEQ        FCMP GEQ
```

forms the full system of real comparisons.

Interpreter:

```
X:=REAL(Pop()); Y:=REAL(Pop());
IF    X<Y THEN Push(1); Push(0)
ELSIF X>Y THEN Push(0); Push(1)
ELSE Push(0); Push(0) END
```

FABS

Float ABSolute

Operation Code: 1 byte 9Dh
Immediate Operands: none
Instruction Length: 1 byte

Action:

Pops a real number from the stack and pushes its absolute value into the stack. PC increments by 1.

Interpreter:

```
X:=REAL(Pop());
IF X<0.0 THEN Push(-X) ELSE Push(X) END
```

FNEG

Float NEGative

Operation Code: 1 byte 9Eh
Immediate Operands: none
Instruction Length: 1 byte

Action:

Pops a real number from the stack, negates it and pushes it back into the stack. PC increments by 1.

Interpreter:

```
Push(-REAL(Pop()))
```

FFCT

Float FunCTion

Operation Code: 1 byte 9Fh
Immediate Operands: 1 byte
Instruction Length: 2 bytes

Action:

Case of the next byte following the command:

next = 0

pops an integer number from the stack, transfers it into the appropriate real number and pushes it back into the the stack. PC increments by 1.

next = 1

pops a real number from the stack, truncates it into the

appropriate integer number and pushes it back into the the stack. PC increments by 1. (Integer overflow may happen).

```
Interpreter:
     i:=Next();
     IF    i=0 THEN Push(FLOAT(INTEGER(Pop())))
     ELSIF i=1 THEN Push.(TRUNC(   REAL(Pop())))
     ELSE DEC(PC); TRAP(7h) END;
```

LSS

integer LeSS

```
Operation Code:                1 byte          0A0h
Immediate Operands:              none
Instruction Length:            1 byte
```

Action:

Pops the Upper word and then the Lower word. Compares them and if the Lower integer less than the Upper one pushes 1 otherwise 0. PC increments by 1. Overflow never happens.

```
Interpreter:
     i:=Pop(); Push(Pop()<i)
```

LEQ

integer Less or EQual

```
Operation Code:                1 byte          0A1h
Immediate Operands:              none
Instruction Length:            1 byte
```

Action:

Pops the Upper word and then the Lower word. Compares them and if the Lower integer <= the Upper one pushes 1 otherwise 0. PC increments by 1. Overflow never happens.

```
Interpreter:
     i:=Pop(); Push(Pop()<=i)
```

GTR

integer GreaTeR

```
Operation Code:                1 byte          0A2h
Immediate Operands:              none
Instruction Length:            1 byte
```

Action:

Pops the Upper word and then the Lower word. Compares them and if the Lower integer > the Upper one pushes 1 otherwise 0. PC increments by 1. Overflow never happens.

```
Interpreter:
     i:=Pop(); Push(Pop()>i)
```

GEQ
integer Greater or EQual

```
Operation Code:                 1 byte          0A3h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Pops  the  Upper  word  and then the Lower word. Compares
     them  and  if the Lower integer >= the Upper one pushes 1
     otherwise 0. PC increments by 1. Overflow never happens.

Interpreter:
     i:=Pop(); Push(Pop()>=i)
```

EQU
integer EQUal

```
Operation Code:                 1 byte          0A4h
Immediate Operands:               none
Instruction Length:             1 byte
Action:

     Pops  the  Upper  word  and then the Lower word. Compares
     them  and  if  the  Lower  word = the Upper word pushes 1
     otherwise 0. PC increments by 1. Overflow never happens.

Interpreter:
     Push(Pop()=Pop())
```

NEQ
integer NOt eQual

```
Operation Code:                 1 byte          0A5h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Pops  the  Upper  word  and then the Lower word. Compares
     them  and  if  the  Lower word <> the Upper word pushes 1
     otherwise 0. PC increments by 1. Overflow never happens.

Interpreter:
     Push(Pop()#Pop())
```

ABS
integer ABSolute

```
Operation Code:                 1 byte          0A6h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
```

Pops the integer value and pushes its absolute value into the stack. PC increments by 1. Overflow happens for -MIN(INTEGER).

Interpreter:

```
Push(ABS(Pop()))
```

NEG

integer NEGative

```
Operation Code:                 1 byte          0A7h
Immediate Operands:               none
Instruction Length:             1 byte
```

Action:

Pops the integer value, negates it and pushes its absolute value into the stack. PC increments by 1. Overflow happens for -MIN(INTEGER).

Interpreter:

```
Push(-Pop())
```

OR

logical bit per bit OR

```
Operation Code:                 1 byte          0A8h
Immediate Operands:               none
Instruction Length:             1 byte
```

Action:

Pops 2 words from the stack, executes OR for all 32 bits of these words and pushes the result into the stack. PC increments by 1.

Interpreter:

```
v:=BITSET(Pop()); w:=BITSET(Pop()); Push(w+v)
```

AND

logical bit per bit AND

```
Operation Code:                 1 byte          0A9h
Immediate Operands:               none
Instruction Length:             1 byte
```

Action:

Pops 2 words from the stack, executes AND for all 32 bits of these words and pushes the result into the stack. PC increments by 1.

Interpreter:

```
v:=BITSET(Pop()); w:=BITSET(Pop()); Push(w*v)
```

XOR
logical bit per bit XOR

```
Operation Code:                 1 byte          0AAh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
```

Pops 2 words form the stack, executes XOR for all 32 bits of these words and pushes the result into the stack. PC increments by 1.

```
Interpreter:
     v:=BITSET(Pop()); w:=BITSET(Pop()); Push(w/v)
```

BIC
logical bit per bit BIt Clear

```
Operation Code:                 1 byte          0ABh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
```

Pops the Upper word and then the Lower word from the stack. Clears in Lower one all bits which are ON in the Upper word and pushes the result into the stack. PC increments by 1.

```
Interpreter:
     v:=BITSET(Pop()); w:=BITSET(Pop()); Push(w-v)
```

IN
IN bitset

```
Operation Code:                 1 byte          0ACh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
```

Pops the Scale (S) word and then the element (N) from the stack. If N>=0 and N<=31 and bit number N set ON in S, then pushes 1, otherwise 0 into the stack. PC increments by 1.

```
Interpreter:
     v:=BITSET(Pop()); Push(Pop() IN v)
```

BIT
set BIT

```
Operation Code:                 1 byte          0ADh
Immediate Operands:               none
```

```
Instruction Length:            1 byte
Action:
     Pops  the bit number (N) and pushes the word, in wich all
     bits  are zeros except bit with number N, into the stack.
     If  N<0 or N>31 then trap 4Ah is raised. PC increments by
     1.

Interpreter:
     i:=Pop();
     IF (i<0) OR (i>=20h) THEN TRAP(4Ah)
     ELSE w:={}; INCL(w,i); Push(w) END
```

NOT

```
boolean NOT (not bit per bit!)

Operation Code:                1 byte          0AEh
Immediate Operands:              none
Instruction Length:            1 byte
Action:

     If  the popped value equals 0, pushes 1, otherwise 0 into
     the the stack. PC increments by 1.
Interpreter:
     Push(Pop()=0)
```

MOD

```
integer MODule

Operation Code:                1 byte          0AFh
Immediate Operands:              none
Instruction Length:            1 byte
Action:
     Pops  the  Upper  and then the Lower operands. Pushes the
     integer  remainder  from the Lower by the Upper division.
     Integer overflow may happen.

Interpreter:
     i:=Pop(); Push(Pop() MOD i)
```

DECS

```
DECrement S_register

Operation Code:                1 byte          0B0h
Immediate Operands:              none
Instruction Length:            1 byte
Action:
     Decrements  the  S_register  by the value popped from the
     stack. PC increments by 1.

Interpreter:
     DEC(S,Pop())
```

DROP
DROP

```
Operation Code:                 1 byte          0B1h
Immediate Operands:               none
Длина команды:                  1 byte
Action:
     Ignores the popped value. PC increments by 1.

Interpreter:
     i:=Pop();
```

LODF
reLOaD the expression stack after Function return

```
Operation Code:                 1 byte          0B2h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Pops the value from the stack. Reload the stack by the
     contents saved before at the P-stack. Pushes the value
     over the reloaded stack. PC increments by 1. This
     instruction is used for reloading the saved stack after
     function calls.

Interpreter:
     i:=Pop(); RestoreExpStack; Push(i)
```

STORE
STORE the expression stack

```
Operation Code:                 1 byte          0B3h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     If it is possible (the free space between S and H
     registers presents), saves the expression stack,
     incrementing S_register. Puts the depth of the saved
     expression stack at the top of the P-stack (pointed by
     S_register-1). PC increments by 1. If the free space for
     saving is absent, trap 40h is raised.

Interpreter:
     IF S+ESdepth+1>H THEN DEC(PC); TRAP(40h)
     ELSE SaveExpStack
     END
```

STOFV
STORE the expression stack with Formal function value at the top

```
Operation Code:                 1 byte          0B4h
```

Immediate Operands: none
Instruction Length: 1 byte
Action:

Pops the value from the stack after the stack saving in the same manner as in the instruction STORE. After that the popped at the very begining value stores (see STOT) at the top of the P-stack. PC increments by 1. Traps 40h may be raised. This instruction is useful for the stack saving when the top word is the formal function which is called by the CF instruction (see).

Interpreter:

```
IF S+ESdepth+2>H THEN DEC(PC); TRAP(40h)
ELSE i:=Pop(); SaveExpStack; Core[S]:=i; INC(S) END
```

COPT

COPy Top of expression the stack

Operation Code: 1 byte 0B5h
Immediate Operands: none
Instruction Length: 1 byte
Action:

Pushes twice the popped value. PC increments by 1.

Interpreter:

```
i:=Pop(); Push(i); Push(i)
```

CPCOP

Character array Parameter COPy

Operation Code: 1 byte 0B6h
Immediate Operands: 1 byte
Instruction Length: 2 bytes
Action:

Loads the immediate operand N (the next byte followed by the instruction code). A value of the S_register is stored in the local word N (at memory location L_register+N). Popped High of array is transformed to a word size. Then array is copied from address popped from the stack into the memory pointed by the S_register, with incrementing the S_register. PC increments by 2. Trap 40h may happen.

Note:

In the RISC Kronos processor this instruction will be discarded.

Interpreter:

```
i:=Pop(); (* High *) sz:=(i+4) DIV 4;
IF S+sz>H THEN Push(i); DEC(PC); TRAP(40h)
```

```
     ELSE Core[L+Next()]:=S; adr:=Pop();
       WHILE sz>0 DO Core[S]:=Core[adr]; INC(S); INC(adr) END
     END
```

PCOP

structure Parameter allocate and COPy

```
Operation Code:                 1 byte          0B7h
Immediate Operands:             1 byte
Instruction Length:             2 bytes
```

Action:

Loads the immediate operand N (the next byte followed by the instruction code). A value of the S_register is stored in the local word N (at memory location L_register+N). Loads High of array from the stack and increments it by 1 to transform into size. Then array is copied from popped from the stack address into the memory pointed by the S_register, with incrementing the S_register. PC increments by 2. Trap 40h may happen.

Note:

In RISC Kronos processor this instruction will be discarded.

Interpreter:

```
     i:=Pop(); (* High *) sz:=i+1;
     IF S+sz>H THEN Push(i); DEC(PC); TRAP(40h)
     ELSE Core[L+Next()]:=S; adr:=Pop();
       WHILE sz>0 DO Core[S]:=Core[adr]; INC(S); INC(adr) END
     END
```

FOR1

enter FOR statement

```
Operation Code:                 1 byte          0B8h
Immediate Operands:             3 bytes
Instruction Length:             4 байта
```

Action:

Pops High and Low bounds of FOR loop from the stack, then pops the cycle parameter address. Loads the sign of step of cycle from the first immediate operand (byte following the instruction code). Loads Offset of the first instruction out of loop from the second immediate operand (2 bytes). PC increments by 4. Stores Low bound of the cycle in parameter. If conditions of the loop are not satisfied exits out of loop with incrementing the PC by Offset, else enters the loop (goes to the next instruction). Address of cycle parameter and High bounds of the loop stores at the P-stack. Trap 40h may happen.

Note:

In the RISC Kronos processor this instruction will be discarded.

```
Interpreter:
     IF S+2>H THEN DEC(PC); TRAP(40h)
     ELSE sz:=Next(); (* =0 up; #0 down *)
       hi:=Pop(); low:=Pop(); adr:=Pop(); k:=Next2()+PC;
       IF ((sz=0) & (low<=hi)) OR ((sz#0) & (low>=hi)) THEN
         Core[adr]:=low;
         Core[S]:=adr; INC(S); Core[S]:=hi; INC(S);
       ELSE (* цикл не исполняется не разу *) PC:=k
       END
     END
```

FOR2

end of FOR statement

```
Operation Code:                 1 byte          0B9h
Immediate Operands:             3 bytes
Instruction Length:             4 байта
```

Action:

Loads the step of cycle from the first immediate operand (byte following the instruction code) and decrements the step by 128 to restrict it in the range -128..+127. Loads the Offset to the first instruction of the cycle (2 bytes) from the second immediate operand. PC increments by 4. Loads the High bound and cycle parameter address from the P-stack. Adds the step to the cycle parameter. If the cycle condition are satisfied, jumps to the first instruction of the loop with decrementing the PC by Offset, else drops High and address from the P-stack with decrementing the S_register by 2 and exits out of loop (goes to the next instruction). Note: FOR1 executes only at the first iteration of loop.

Note:

In the RISC Kronos processor this instruction will be discarded.

```
Interpreter:
     hi:=Core[S-1];
     adr:=Core[S-2];
     sz:=Next();
     IF sz>7Fh THEN
       sz:=7Fh-sz    (* шаг [-128..127] *)
     END;
     k:=-Next2()+PC;
     i:=Core[adr]+sz;
     IF ((sz>=0) & (i>hi)) OR ((sz<0) & (i<hi)) THEN
       DEC(S,2); (* terminate *)
     ELSE Core[adr]:=i; PC:=k (* continue *)
     END
```

ENTC

ENTer Case statement

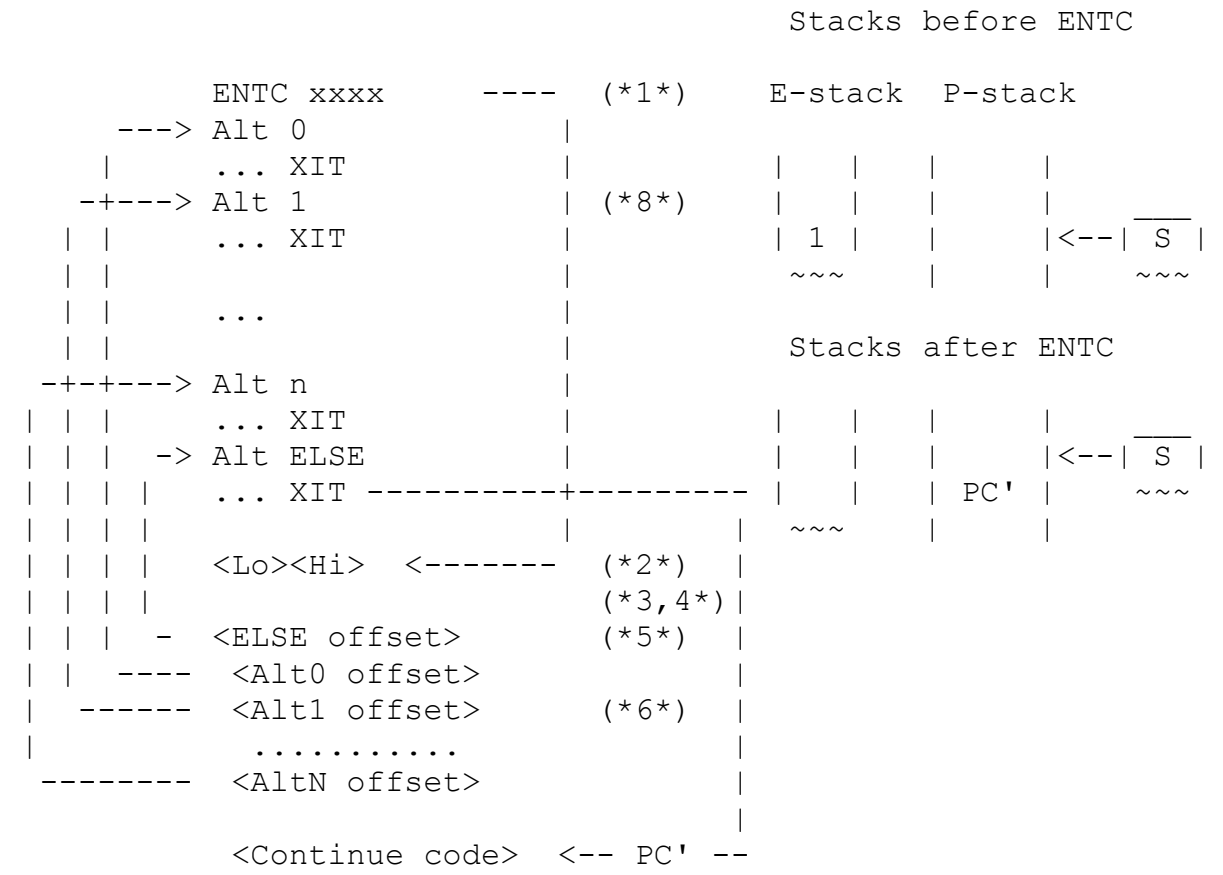
```
Operation Code:                 1 byte          0BAh
Immediate Operands:             2 bytes         00h..0FFFFh
Instruction Length:             3 bytes +  case table size
```

Action:

Selects the needful alternative jump from the case table. Trap 40h may happen.

Note:

In the RISC Kronos processor this instruction will be discarded.



Interpreter:

```
IF S+1>H THEN DEC(PC); TRAP(40h)
ELSE PC:=Next2()+PC; (* jump to case table *)
  k:=Pop(); low:=Next2(); hi:=Next2();
  Core[S]:=PC + 2*(hi-low) + 4; INC(S);(*PC for exit*)
  IF (k>=low) & (k<=hi) THEN
    PC:=PC+2*(k-low+1) (* jump into case table *)
  END;
  PC:=-Next2()+PC (* jump back to variant's code *)
END
```

XIT
eXIT from case

```
Operation Code:                 1 byte          0BBh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Loads  the  PC  value  from  the  P-stack  and decrements
     S_register. Jumps to the specified PC.

Note:
     In  the  RISC  Kronos  processor this instruction will be
     discarded.

Interpreter:
     DEC(S); PC:=Core[S]
```

ADDPC
ENTer control Structure

```
Operation Code:                 1 byte          0BCh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Adds  value  from  the PC_register (after the instruction
     code fetching) to the top of the stack.

Interpreter:
     Push(PC+Pop());
```

JUMP
ENTer control Structure

```
Operation Code:                 1 byte          0BDh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Jumps to the instruction wich PC has been popped from the
     top of the stack.

Interpreter:
     PC:=Pop();
```

ORJP
short circuit OR JumP

```
Operation Code:                 1 byte          0BEh
Immediate Operands:             1 byte
Instruction Length:             2 bytes
Action:
```

Implements the MacCarthy disjunction (conditional OR). If the value popped from the stack is not equal to zero (TRUE) pushes 1 into the stack and jumps to the end of condition (the offset taken from the next byte), otherwise goes to the next instruction.

```
Interpreter:
     IF Pop()#0 THEN
       Push(1); PC:=Next()+PC
     ELSE
       INC(PC)
     END
```

ANDJP

short circuit AND JumP

```
Operation Code:                1 byte          0BFh
Immediate Operands:            1 byte
Instruction Length:            2 bytes
```

Action:

Implements the MacCarthy conjunction (conditional AND). If the value popped from the stack is equal to zero (FALSE) pushes 0 into the stack and jumps to the end of condition (offset taken from the next byte), otherwise goes to the next instruction.

```
Interpreter:
     IF Pop()=0 THEN
       Push(0); PC:=Next()+PC
     ELSE
       INC(PC)
     END
```

MOVE

MOVE block

```
Operation Code:                1 byte          0C0h
Immediate Operands:              none
Instruction Length:            1 byte
```

Action:

Pops size (in words), source and destinator addresses from the stack. Moves the sequence of words (in address increasing direction) from source to destinator address.

Note:

May be used to filling area of memory by pattern.

```
Interpreter:
     sz:=Pop();
     i:=Pop(); j:=Pop();
     WHILE sz>0 DO
```

```
      Core[j]:=Core[i]; INC(i); INC(j); DEC(sz)
    END
```

CHKNIL
ReaD String

```
Operation Code:                 1 byte          0C1h
Immediate Operands:             none
Instruction Length:             1 byte
Action:
    Checks  the  address  on  the  top of the stack. If it is
    equal to NIL raises the interrupt.
Interpreter:
    adr:=Pop(); Push(adr);
    IF adr=NIL THEN TRAP(3) END;
```

LSTA
Load STring Address

```
Operation Code:                 1 byte          0C2h
Immediate Operands:             2 bytes
Instruction Length:             3 bytes
Action:
    Loads  the immediate 2 byte operand and adds its value to
    the  value  of  Global Word 1 (string pointer) and pushes
    sum into the stack.

Note:
    Will be discarded in the RISC Kronos architecture.

Interpreter:
    Push(Core[G+1]+Next2());
```

COMP
COMPare strings

```
Operation Code:                 1 byte          0C3h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
    Pops from the stack the base addresses of two strings and
    compares them byte after byte until they are not equal or
    one  of  them  is  zero byte (string terminator). In both
    cases  pushes the values of two last bytes into the stack
    in  the  same order as the source strings was. After this
    instruction  any compare instruction (EQU, NEQ, LSS, GTR,
    LEQ, GEQ) may be applied for the string comparison.

Interpreter:
    i:=Pop()*4; j:=Pop()*4;
```

```
     REPEAT a:=CHAR(ByteCore[i]); b:=CHAR(ByteCore[j]);
     INC(i); INC(j)
     UNTIL (a=0c) OR (b=0c) OR (a#b); Push(a); Push(b)
```

GB

Get procedure Base n level down

```
Operation Code:                 1 byte          0C4h
Immediate Operands:             1 byte
Instruction Length:             2 bytes
Action:
     Takes  the  number  of  levels  N from the byte immediate
     operand.  Goes  through  the  procedure  chain  of  saved
     L_registers  until  the depth N will be extracted. Pushes
     the value on the N'th L_register into the stack.

Note:
     Will be discarded in the RISC Kronos architecture.

Interpreter:
     i:=L; n:=Next();
     WHILE n>0 DO i:=Core[i]; DEC(n) END;
     Push(i)
```

GB1

Get procedure Base 1 level down

```
Operation Code:                 1 byte          0C5h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     The short form of the instruction GB 01.

Note:
     Will be discarded in the RISC Kronos architecture.

Interpreter:
     Push(Core[L])
```

CHK

range bounds CHecK

```
Operation Code:                 1 byte          0C6h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Pops  high  and  low  bounds and index from the stack and
     checks  whether the index lies in the range low..high. If
     it  is  out of the range raises the interrupt 4Ah else no
     operation.  In  both cases pushes the index backward into
```

the stack.

```
Interpreter:
     hi:=Pop(); low:=Pop(); i:=Pop(); Push(i);
     IF (i<low) OR (i>hi) THEN
       Push(low); Push(hi); TRAP(4Ah)
     END
```

CHKZ

array bounds CHecK (low=Zero)

```
Operation Code:                 1 byte          0C7h
Immediate Operands:               none
Instruction Length:             1 byte
```

Action:

Pops the high bound and the index from the stack and checks wheter index lies in range 0..high. If it is out of range raises the interrupt 4Ah else no operation. In both cases pushes the index backward into the stack.

```
Interpreter:
     hi:=Pop(); i:=Pop(); Push(i);
     IF (i<0) OR (i>hi) THEN Push(hi); TRAP(4Ah) END
```

ALLOC

ALLOCate block

```
Operation Code:                 1 byte          0C8h
Immediate Operands:               none
Instruction Length:             1 byte
```

Action:

Pops the size of structure from the stack, takes a value of the S_register as the start address of the structure, increments the S_register by the size. Pushes the base address of allocated area into the stack. If incremented S_register greater than the H_register raises the interrupt 40h without memory allocation.

```
Interpreter:
     sz:=Pop();
     IF S+sz>H THEN
       Push(sz); DEC(PC); TRAP(40h)
     ELSE
       Push(S); INC(S,sz)
     END
```

ENTR

ENTeR procedure

```
Operation Code:                 1 byte          0C9h
```

```
Immediate Operands:             1 byte
Instruction Length:             2 bytes
Action:
```

Takes the next byte immediate operand as a number of necessary local variables for procedure execution and allocates a local area for them. It is equivalent to the sequence LIB <n> ALLOC DROP. The 40h interrupt may be raised.

```
Interpreter:
     sz:=Next();
     IF S+sz>H THEN
       DEC(PC,2); TRAP(40h)
     ELSE
       INC(S,sz)
     END
```

RTN

ReTurN from procedure

```
Operation Code:                 1 byte          0CAh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
```

Restores the previous value of the S_register (to release memory used for procedure execution) from the procedure links area (see the instruction CALL), the value of the L_register of procedure, from wich this one was called, the value of the PC_register and, if the external call was performed (ExternalBit is in saved PC_word of the links area), restores the G_register of caller procedure's module.

If processor mask (M_register) was changed during the procedure execution then the old mask value is restored from the apropriate links area word. So no one procedure may change M_register longer than its own execution.

```
Note:
```

Restoring after mask change will be discarded in the RISC Kronos architecture.

```
Interpreter:
     S:=L;
     PC:=WORD(BITSET(Core[S+2])*{0..0Fh});
     L:=Core[S+1];
     IF ExternalBit IN BITSET(Core[S+2]) THEN
       (* external called *)
       G:=Core[S]; F:=CodePtr(Core[G])
     END;
     IF ChangeMaskBit IN BITSET(Core[S+2]) THEN (* mask was
changed *)
       M:=BITSET(Core[S+3])*{0..10h}
     END;
```

NOP
No OPeration

```
Operation Code:                 1 byte          0CBh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     It is not too hard to understand this instruction.
```

CX
Call eXternal

```
Operation Code:                 1 byte          0CCh
Immediate Operands:             2 bytes
Instruction Length:             3 bytes
Action:
     External Call.

     If  there are less than 4 words between S and H registers
     then  raises  the interrupt 40h. Takes the next immediate
     byte  operand  as  a module number and the next immediate
     byte  operand  as  a procedure number. After that a new 4
     words  procedure  links area is constructed. In zero word
     puts  the  current  value of the G_register, in the first
     word  puts  the  current  value of the L_register, in the
     second word puts the current value of the PC_register (16
     bits)  combined  with  the  ExternalBit mark. After that,
     using  module  number,  takes a new G_register from local
     (by indirection through global) DFT. Dereferences the new
     G_register  and  obtains  the  F_register, adds procedure
     number  to  it  and  obtains  a  new  PC  value  from the
     appropriate  word  of  procedure table. And then jumps to
     select the instruction (the begin of called procedure).

Note:
     Extra  dereferencing  through  the  global  DFT  will  be
     discarded in the RISC Kronos architecture.

Interpreter:
     IF S+4<=H THEN j:=Core[G-Next()-1]; (* big DFT *)
       i:=Next(); Mark(G,TRUE);
       G:=Core[j]; F:=CodePtr(Core[G]); PC:=GetPc(i);
     ELSE DEC(PC); TRAP(40h) END
```

CI
Call procedure at Intermediate level

```
Operation Code:                 1 byte          0CDh
Immediate Operands:             1 byte
Instruction Length:             2 bytes
```

Action:

Similar to CX but obtains PC using own F_register and own module procedure table. Stores in zero word of procedure words a value popped from the stack (we assume that it may be pushed by GB <n> instruction). This value may be used by GB <n> LSW <x> or GB <n> ... SSW <x> to organize the access to the intermediate level variables.

Interpreter:

```
IF S+4<=H THEN
  i:=Next(); Mark(Pop(),FALSE); PC:=GetPc(i);
ELSE
  DEC(PC); TRAP(40h)
END
```

CF

Call Formal procedure

Operation Code:	1 byte	0CEh
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:

Takes a procedure value from the top of the P-stack and decrements the S_register by 1. Takes the most significant byte of the procedure value as a procedure number and three bytes as an address of a global DFT entry, where the G_register of called procedure host module is lied. Then executes similary CX instruction.

Note:

Extra dereferencing through the global DFT will be discarded in the RISC Kronos architecture.

Interpreter:

```
IF S+3<=H THEN i:=Core[S-1]; DEC(S); Mark(G,TRUE);
  k:=i DIV 1000000h; i:=i MOD 1000000h;
  G:=Core[i]; F:=CodePtr(Core[G]); PC:=GetPc(k);
ELSE DEC(PC); TRAP(40h) END
```

CL

Call Local procedure

Operation Code:	1 byte	0CFh
Immediate Operands:	1 byte	
Instruction Length:	2 bytes	

Action:

Similar CI instruction but stores the value of the L_register in zero word of procedure links area. May be used to call the procedures at the same lexicographic level.

```
Interpreter:
     IF S+4<=H THEN
       i:=Next(); Mark(L,FALSE); PC:=GetPc(i);
     ELSE
       DEC(PC); TRAP(40h)
     END
```

CL0..CL0E

Call Local procedure

```
Operation Code:                 4 bits          0D0h
Immediate Operands:             4 bits          00h..0Fh
Instruction Length:             1 byte
```

Action:

Equivalent to CL instruction but extracts the procedure number from four the least significant bits of the instruction code.

```
Interpreter:
     IF S+4<=H THEN Mark(L,FALSE); PC:=GetPc(IR MOD 10h);
     ELSE DEC(PC); TRAP(40h) END
```

INCL 0E0h

INCLude in set

```
Operation Code:                 1 byte
Immediate Operands:               none
Instruction Length:             1 byte
```

Action:

Pops the bit number and the destinator address from the stack. If the bit number outs of the range 0..31 then the interrupt 4Ah is raised. Sets bit with appropriate number in the destinator word.

```
Interpreter:
     i:=Pop();
     IF (i<0) OR (i>1Fh) THEN Push(i); DEC(PC); TRAP(4Ah)
     ELSE j:=Pop(); w:=BITSET(Core[j]); INCL(w,i);
       Core[j]:=CARDINAL(w)
     END
```

EXCL

EXCLude from set

```
Operation Code:                 1 byte          0E1h
Immediate Operands:               none
Instruction Length:             1 byte
```

Action:

Pops the bit number and the destinator address from the stack. If the bit number outs of the range 0..31 then

interrupt 4Ah is raised. Sets the bit with appropriate number in the destinator word.

```
Interpreter:
     i:=Pop();
     IF (i<0) OR (i>1Fh) THEN Push(i); DEC(PC); TRAP(4Ah)
     ELSE j:=Pop(); w:=BITSET(Core[j]); EXCL(w,i);
       Core[j]:=CARDINAL(w)
     END
```

SLEQ 0E2h

bitSet Less or EQual

```
Operation Code:                 1 byte
Immediate Operands:               none
Instruction Length:             1 byte
```

Action:

Pops the bitsets Top and Under. If bitset Under is the subset of Top then pushes 1 (TRUE) else 0 (FALSE).

Note:

Will be discarded in the RISC Kronos architecture.

```
Interpreter:
     w:=BITSET(Pop()); v:=BITSET(Pop()); Push(v<=w)
```

SGEQ

bitSet Greater or EQual

```
Operation Code:                 1 byte          0E3h
Immediate Operands:               none
Instruction Length:             1 byte
```

Action:

Pops the bitsets Top and Under. If bitset Top is the subset of Under then pushes 1 (TRUE) else 0 (FALSE).

```
Interpreter:
     w:=BITSET(Pop()); v:=BITSET(Pop()); Push(v>=w)
```

INC1

INCrement by 1

```
Operation Code:                 1 byte          0E4h
Immediate Operands:               none
Instruction Length:             1 byte
```

Action:

Increments a value at the memory location at popped address by 1.

```
Interpreter:
     INC(Core[Pop()])
```

DEC1
DECrement by 1

```
Operation Code:                 1 byte          0E5h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Decrements  a  value  at  the  memory  location at popped
     address by 1.

Interpreter:
     DEC(Core[Pop()])
```

INC
INCrement

```
Operation Code:                 1 byte          0E6h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Pops  the  step from the stack. Increments a value at the
     memory location at popped address by the step.

Interpreter:
     i:=Pop(); INC(Core[Pop()],i)
```

DEC
DECrement

```
Operation Code:                 1 byte          0E7h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Pops  a  step  from  the stack. Decrements a value at the
     memory location at popped address by the step.

Interpreter:
     i:=Pop(); DEC(Core[Pop()],i)
```

STOT
STOre the Top at the procedure stack

```
Operation Code:                 1 byte          0E8h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Stores  popped  from the stack value at memory located by
     the  L_register. Increments the L_register by 1. If there
     is  less  than  1  word  between S and H registers raises
     interrupt 40h.
```

```
Interpreter:
     IF S+1>H THEN DEC(PC); TRAP(40h)
     ELSE Core[S]:=Pop(); INC(S)
     END
```

LODT

LOaD the Top of the procedure stack

```
Operation Code:                1 byte          0E9h
Immediate Operands:              none
Instruction Length:            1 byte
Action:
     Decrements  the S-register by 1. Pushes value loaded from
     located by S_register.

Interpreter:

     DEC(S); Push(Core[S])
```

LXA

Load indeXed Address

```
Operation Code:                1 byte          0EAh
Immediate Operands:              none
Instruction Length:            1 byte
Action:
     Pops  a  size  and  an  index of an element and a base of
     structure  array.  Multiplies  the size and the index and
     adds to the base and pushes the result.

Interpreter:
     sz:=Pop(); i:=Pop(); adr:=Pop(); Push(adr+i*sz)
```

LPC

Load Procedure Constant

```
Operation Code:                1 byte          0EBh
Immediate Operands:            2 bytes
Instruction Length:            3 bytes
Action:
     Takes  a  module  number  from  the  next  immediate byte
     operand  and the procedure number from the next immediate
     byte  operand.  Takes  the word with offset equals to the
     module number from the local DFT (the address of entry in
     global  DFT)  and packs it in three the least significant
     bytes  of the resulting procedure value. Procedure number
     is  packed  in  the  most significant byte of the result.
     Pushes the result into the stack.
```

```
Interpreter:
     i:=Next(); j:=Next(); Push(j*1000000h+Core[G-i-1])
```

The following 3 instructions deal with bit slices. Bit address is the pair (address, bit offset). Bit offset can be greater than 32. The slice may out of word bounds.

BBU

Bit Block Unpack

```
Operation Code:                 1 byte          0ECh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
```

Pops a size of a bit slice (1..32), a bit offset and a base address from the stack. Forms bit slice of bits from memory located by base address and bit offset, extends it by leading zeros and pushes it into the stack.

```
Interpreter:
     sz:=Pop();
     IF (sz<1) OR (sz>32) THEN
       Push(sz); DEC(PC); TRAP(4Ah)
     END;
     i:=Pop(); adr:=Pop();
     (* j:="sz bits from bit address adr*32+i" *)
     Push(j);
```

BBP

Bit Block Pack

```
Operation Code:                 1 byte          0EDh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
```

Pops value, size of bit slice (sz), offset and base from the the stack, truncate the least significant sz bits from value, and packs it at memory located by base address and bit offset.

```
Interpreter:
     j:=Pop(); sz:=Pop();
     IF (sz<1) OR (sz>32) THEN
       Push(sz); DEC(PC); TRAP(4Ah)
     END;
     i:=Pop(); adr:=Pop();
     (* "pack sz bits from j at bit address adr*32+i *)
```

BBLT

Bit BLock Transfer

```
Operation Code:                 1 byte          0EEh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Pops  the  size  of  bit slice, the source bit offset and
     base address and destinator offset and base. Transfer bit
     sequence  from  the  soucre bit address to the destinator
     one.  NOTE!  If source and destinator areas are overlaied
     then the instruction result may be unexpected.

Interpreter:
     sz:=Pop();
     i:=Pop(); adr:=Pop();
     j:=Pop(); adr1:=Pop();
     (* "transfer sz bits from bit address
         adr*32+i to bit address adr*32+i"
     *)
```

SWAP

SWAP

```
Operation Code:                 1 byte          0F0h
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Swaps two top words on the stack.

Interpreter:
     i:=Pop(); j:=Pop(); Push(i); Push(j)
```

LPA

Load Parameter Address

```
Operation Code:                 1 byte          0F1h
Immediate Operands:             1 byte
Instruction Length:             2 bytes
Action:
     Substracts the next byte immediate operand and 1 from the
     L_register and pushes the result into the stack.

Interpreter:
     Push(L-Next()-1);
```

LPW

Load Parameter Word

```
Operation Code:                 1 byte          0F2h
Immediate Operands:             1 byte
Instruction Length:             2 bytes
Action:
```

Substracts the next byte immediate operand and 1 from the L_register, loads a word from the result memory location and pushes it into the stack.

Interpreter:

```
Push(Core[L-Next()-1]);
```

SPW

Store Parameter Word

Operation Code:	1 byte	0F3h
Immediate Operands:	1 byte	
Instruction Length:	2 bytes	

Action:

Pops a value from the stack. Substracts the next byte immediate operand and 1 from the L_register, stores a value at the result memory location and pushes it into the stack.

Interpreter:

```
Core[L-Next()-1]:=Pop();
```

SSWU

Store Stack Word Undistractive

Operation Code:	1 byte	0F4h
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:

The same as the instruction SSW but pushes popped value backward into the stack.

Interpreter:

```
i:=Pop(); Core[Pop()]:=i; Push(i)
```

ACTIV

ACTIVe process

Operation Code:	1 byte	0FAh
Immediate Operands:	none	
Instruction Length:	1 byte	

Action:

Pushes address of the active process descriptor into the stack.

Interpreter:

```
Push(P)
```

USR

USeR defined functions

```
Operation Code:                 1 byte          0FBh
Immediate Operands:             1 byte
Instruction Length:             2 bytes
Action:
     Instruction is reserved for the future extensions.

Interpreter:
     i:=Next(); (*  *)
```

SYS

SYStem rarely functions

```
Operation Code:                 1 byte          0FCh
Immediate Operands:             1 byte
Instruction Length:             2 bytes
Action:
     Instruction is reserved for the future extensions.
```

NII

Never Implemented Instruction

```
Operation Code:                 1 byte          0FDh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Raises the interrupt 07h.

Interpreter:
     TRAP(7h);
```

INVLD

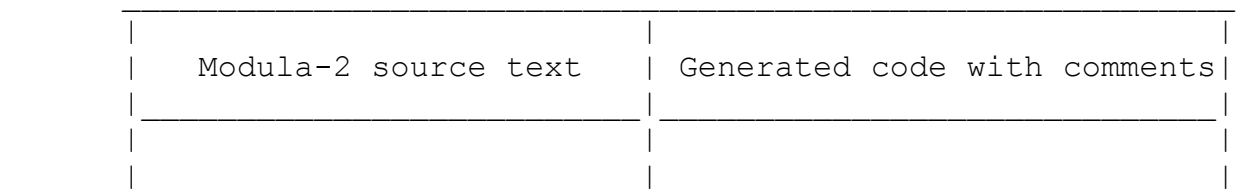
INVaLiD command

```
Operation Code:                 1 byte          0FFh
Immediate Operands:               none
Instruction Length:             1 byte
Action:
     Raises the interrupt 49h.

Interpreter:
     TRAP(49h)
```

ILLUSTRATIONS FOR THE PROCESSORS ARCHITECTURE

Given partition may be used as a manual for the KRONOS family processors architecture, their instruction set and Modula-2 compiler code generation process.

The information will be present in the next form:



Note that code will be given only for illustrative needs and thereby may differ from the code really generated by the current version of Modula-2 compiler. The reasons for it are follows:

1) optimizations which increase the code efficiency but harm the recognizing are not reflected in examples;

2) some examples contain the several variants of code generation but in the compiler current version only the one version is of course implemented;

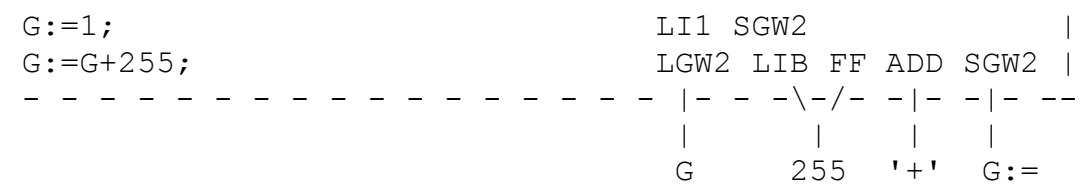
3) the dynamic control instructions (for example range check instructions) are omitted.

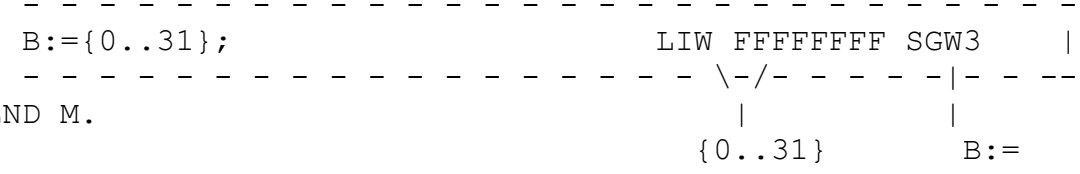
The M-code instructions mnemonics are used in examples. The insructions formal description is provided by M-code interpreter.

1. STATEMENTS

1.1. Assignment

```
MODULE M;    (* Global variables assigment *)

VAR  G: INTEGER;
     B: BITSET;

BEGIN
```

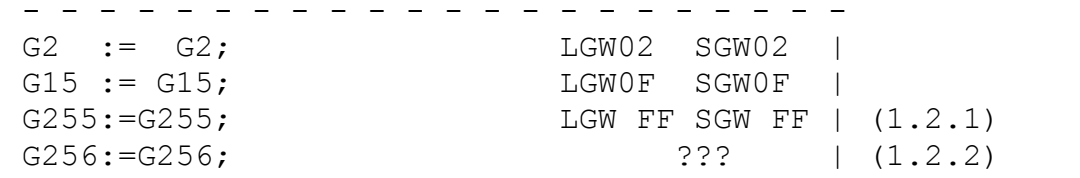





```
END M.
```

1.2. Access to global variables

```
MODULE M;      (* Greate number of global varialbes *)

VAR  G2,G3,G4, ... ,G255:INTEGER;
     G256: INTEGER;

BEGIN
```




```
END M.
```

Note 1.2.1. Access to the first 14 global variables (with numbers from 2 till 15) has a half-byte (hex) offset and for variables with numbers from 16 till 255 has byte offset.

Note 1.2.2. This is wonder, that a man can generate such much number of global variables for single module, but if it is so, we can generate:

```
 LGA FF LSW1      LGA FF SSW1
```

or something else ...

1.3. Access to external variables

The global variables of another modules are called "external variables".

```
DEFINITION MODULE M;
  (* external module exported variable "i" *)
  VAR i: INTEGER;
END M.
```

```
                          number of module "M" in
                          local DFT of module "N"
MODULE N;                              |
                                       | the variable "i"
FROM M IMPORT i;                       | number in module "M"
                                       |  |
BEGIN  (* read & write  external variable *)
- - - - - - - - - - - - - - - - - - - - |- |- - - - - - --
| i:=i;                          LEW 01 02  SEW 01 02 |
- - - - - - - - - - - - - - - - - - - - - - - - - - - - --
END N.
```

More detail information about local DFT may be obtained from procedure call's examples in Chapter 2.

1.4. IF-statement

```
MODULE M;  (* conditional statement *)
```

```
VAR bool : BOOLEAN;                    +--->      LGW2
    G3,G4: INTEGER;                    | (1.4.1) JSFC 04  --+
                                       |                    |
BEGIN                                  |                    |
 - - - - - - - - - - - - - - - - - -          LI3 SGW3   |
|IF bool THEN G3:=3 ELSE G4:=4 END;                        |
 - - - - - - - - - - - - - - - - - -  (1.4.2) JSF 02   --|--+
END M.                                 |      _____________|  |
                                       |      |-->LI4 SGW4     |
                                       |      _________________|
                                       +---->  +-->
```

Note 1.4.1. If E-stack top contains 0 then PC increments 4 to omit THEN clause (0 is interpreted as FALSE any other value not equal to zero as TRUE).

Note 1.4.2. Unconditional jump to skip ELSE clause.

1.5. LOOP-statement

Code generation for LOOP-statement implemented with the help of jump instruction:

```
MODULE M;

BEGIN                              -->          - - - - - - -
  - - - - - - - - - - - - - - - --/          |--> JSF 02  --|--|
  | LOOP   EXIT END;                                         |  |
  - - - - - - - - - - - - - - - --\              JSB 04  --   |
END  M.                            -->         - - - - - - - -
                                               |-->
```

If LOOP-statement body is too large, then jump instruction operand may consists of two bytes. We hope that LOOP body less then 65K bytes code.

1.6. REPEAT-statement

```
 MODULE M;

   VAR G2:INTEGER; bool: BOOLEAN;

 BEGIN
  - - - - - - - - - - - - - - - - - ->   _____________________
  | REPEAT G2:=1 UNTIL bool;           |--> LI1  SGW2       |
  - - - - - - - - - - - - - - - - - ->       LGW3 JSBC 05 --|
 END M.
```

1.7. FOR-statement

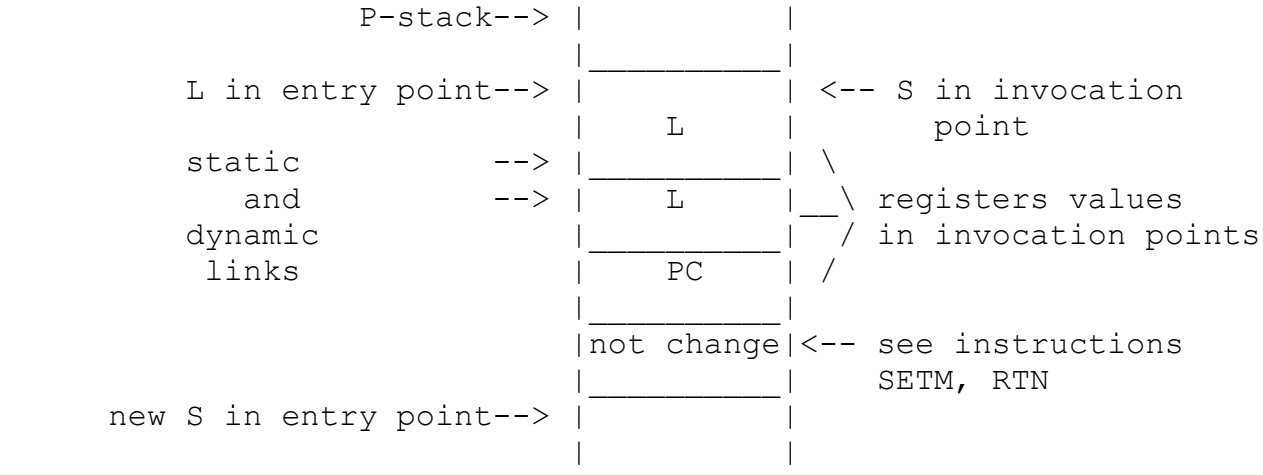
```
MODULE M;

  VAR i,G3:INTEGER;

BEGIN
                                  ____
                                /    LI0              low
                               /     SGW2             =:i
                              /      LIB     7F       high
                             /       SGW4             temporary word
                            /                        for high
 - - - - - - - - - - - -           LGW2    <----     i
  FOR i:=0 TO 127 BY 2 DO          SGW3          |    =:G3
                                   LGW2          |    i
    G3:=i                          LI2           |    step
                                   ADD           |    +
  END;                             COPT          |
 - - - - - - - - - - - - -         SGW2          |    =:i
                             \     LGW4          |    high
                              \    GTR           |    > i ?
                               \   JBSC    -----
                                \___

END M.
```

2. PROCEDURES

2.1. Procedure declaration and it's call

```
MODULE M;

PROCEDURE P; (* procedure number will be 1 *)

      - - - - - - - - - - - - - - - - - -
BEGIN | RETURN END P;              RTN  |
      - - - - - - - - - - - - - - - - - -

BEGIN        (* procedure number will be 0 *)
 - - - - - - - - - - - - - - - - - - - - -
 | P;                                CL1 | (2.1.1)
 - - - - - - - - - - - - - - - - - - -|- -
END M.                                |
                       local procedure 1 call
```

Note 2.1.1. CL1 marks P-stack as follows:



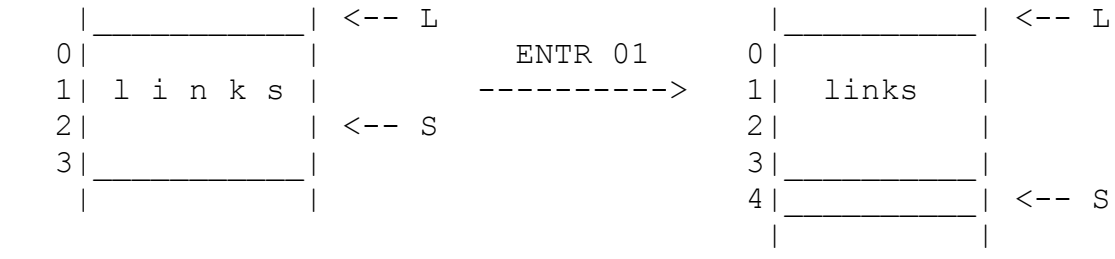
After marking new L = old S and new S = old S + 4. The offset of a corresponding procedure PC is extracted by indexing procedures table (see Note 2.1.2) with procedure number. RTN takes from the P-stack procedure links area values for PC and L in the invocation point and restore them in PC and L registers. The L content is assigned to S register during the return.

Note 2.1.2. Procedure table is the table which gives procedure start PC offset relatively module code frame base F by the procedure number. The module initial part is considered as zero procedure (i.e. procedure whith number 0).

2.2. Access to procedure's local variables

```
MODULE M;

PROCEDURE P; (* procedure 1 *)

                        procedure local variables quantity
                                                |
      - - - - - - - - - - - - - - - - - - - - - |- - - --
VAR   |L4: INTEGER;                 ENTR 01      | (2.2.1)
BEGIN |L4:=0; RETURN                 LI0 SLW4 RTN |
      - - - - - - - - - - - - - - - - - - - - - - |- - --
END P;                                          |
                              first 4 words (with numbers 0..3)
                                  are occupied by links area

BEGIN        (* procedure 0 *)
- - - - - - - - - - - - - - - - - - - - - - - - - -
| P                                 CL1    |
- - - - - - - - - - - - - - - - - - - - - - - - - -
END M.
```

Note 2.2.1. ENTR 01 increments S-register by 1:



After all it is possible to operate with local variables by LLW, SLW, LGW and SGW instructions, based on the L-register.

2.3. Nested procedures

```
MODULE M;

PROCEDURE p1;  (* procedure 1*)
  VAR p1L4:INTEGER;

  PROCEDURE p2; (* procedure  2*)

  VAR p2L4:INTEGER;            put embedding procedure p1
                               L-register on A-stack
  BEGIN                             |
     - - - - - - - - - - - - - - - -|- - - - - - - ---
     |p1L4:=12;                     GB1  LI12  SSW4   |
     |p2L4:=11;                     LI11 SLW4  RTN    |
     - - - - - - - - - - - - - - - - - - - - - - - - - -
  END p1;
                         procedure number in procedure table
BEGIN                                          |
  - - - - - - - - - - -- - - - - - - - - - - - -|- - -
  |p1L4:=2;                        LI2  SLW4   |     |
  |p2;                             LLA 00  CI 02 RTN | (2.3.1)
  - - - - - - - - - - -- - - - - - - -|- -|- - - - -
END p1;                                |   |
                       L-register value   |
               intermediate level procedure call instruction

BEGIN
- - - - - - - - - - - - - - - - - - - - - - -
|p1;                                 CL1  |
- - - - - - - - - - - - - - - - - - - - - - -
END M.
```

Note 2.3.1. CI takes from A-stack base register value, calling procedure and memorizing this value in static chain (instead of L-register which is memorized by CL instruction). GB1 passes procedure static chain on 1 level, accepts base address of low (0) level procedure static chain from which procedure p1 is called, and puts it on A-stack. By this base address the access to corresponding procedure local variables is provided with the help of LSW and SSW instructions. In this example compiler generated CI instead of CL because static chain identical the dynamic one.

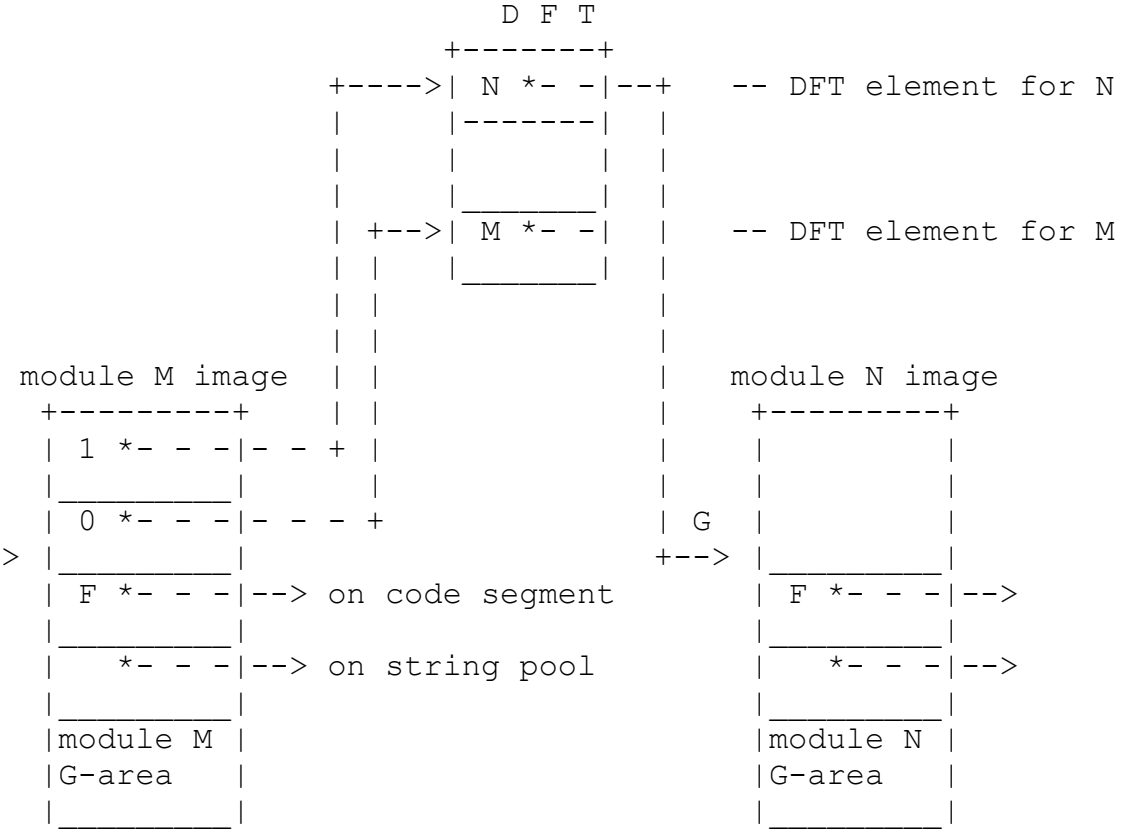
2.4. External procedure call

```
DEFINITION MODULE N;
```

```
PROCEDURE proc1;

END N.


MODULE M;

FROM N IMPORT proc1;
```

```
                                        module number
                                          |
PROCEDURE p1;                             |
                                          | module's proc number
BEGIN                                     |  |
- - - - - - - - - - - - - - - - - - - - -| -|- - -
|  proc1;                          CX 01 01 RTN |  (2.4.1)
- - - - - - - - - - - - - - - - - - - - - - - - -
END p1;

END M.
```

Note 2.4.1. In local DFT element with number 1 corresponds module N and after loading reffers on DFT element which points on module N G-area. The first area word contains module N F-register, thus providing to reach its code segment:



CX puts G-register in static chain links area and marks (putting flag in the second local word) that call was external. RTN instruction analyses this bit and restores G-register value if it's necessary.

2.5. Multivalues allocation

```
 MODULE M;

 PROCEDURE p;
     - - - - - - - - - - - - - - - - - - - - - - - - - - -
 VAR |i: INTEGER;                    ENTR 02              |
     |A: ARRAY [0..15] OF INTEGER;  LIB 10 ALLOC SLW5    |(2.5.1)
     - - - - - - - - - - - - - - - - - - - - - - - - - - -
 BEGIN
   - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   |i:=0;                         LI0  SLW4   |          |
   |                                                     |
   |                               A    i                |
   |                               |    |                |
   |A[i]:=1;                      LLW5 LLW4 LI1 SXW      | (2.5.2)
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - (2.5.3)
END p;

END M.
```

Note 2.5.1. ALLOC takes from A-stack multivalue size, puts back S-register value and moves S on the given size thus reserving the needful number of words on P-stack for multivalue. After all array address is memorized in some local word.

Note 2.5.2. Indexation without range checking. If it's not explicitly updated the range check is switched on.

Note 2.5.3. During procedure return RTN instruction transposes S in L thus frees all memory (in particular allocated by ALLOC instruction on P-stack) for local objects.

2.6. Return from module initial part

In this Modula-2 programming system implementation an initial procedure (module body) during returning puts on A-stack memory size reserving on P-stack after module initialization, i.e. <multivalues size in words> + 4 words of initial procedure links. This information is used by tasking initializer:

```
 MODULE M;
```

```
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 |VAR A: ARRAY [0..0Fh] OF INTEGER;    LIB 010 ALLOC SGW2    |
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

 BEGIN (* procedure 0 *)
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 |END M.                                  LIB 14 RTN          |
 - - - - - - - - - - - - - - - - - - - - - - - - - -|- - - - - - - - -
                                                   14h = 10h + 4h
```

If module hasn't multivalues it must return 0:

```
 MODULE M;
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 |BEGIN END M.                                LI0 RTN |
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

2.7. Operation over pocedure values

```
 MODULE M;

 TYPE proc1=PROCEDURE (INTEGER);

 PROCEDURE P(p1 : proc1); (* procedure # 1 *)


                        to save A-stack on P-stack
                           | load procedure value P1
                           | from procedure P 4-th word
 BEGIN                     |     |  put value on P-stack
                           |     |     | from A-stack
   - - - - - - - - - - - - -|- - -|- - -|- - - - - - - - - - -
   |p1(1);                 SLW4   LLW4  STOT  LI1  CF  RTN  |
   - - - - - - - - - - - - - - - - - - - - - - - -|- -|- - - - -
 END P;                                           |   |
                                                parameter|
                                                       formal procedure
                                                              call


 PROCEDURE p(w: INTEGER); (* procedure # 2 *)
 BEGIN
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   |                                  SLW4     RTN |
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 END p;


 VAR v: proc1;
```

```
                                 module number equals 0,
                                   as procedure is own
                                                |
                                                | procedure number
BEGIN                                           |  |
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  |v:=p;                                LPC 00 02  SGW2       |
  |v(5);                                LGW2  STOT  LI5  CF   |
  |P(v);                                LGW2  CL1             |
  |P(p);                                LPC 00 02  CL1        |
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
END M.
```

```
                          |<---     single word    --->|
                          |_____________________________|
Procedure value:          | 31..24 | 23 . . . . . .0 |
                          -----|-------------|----------
                               |             |
                           procedure      module's DFT
                             number       entry point
```

Procedure links during procedure call by CF instruction (similarly as CX):

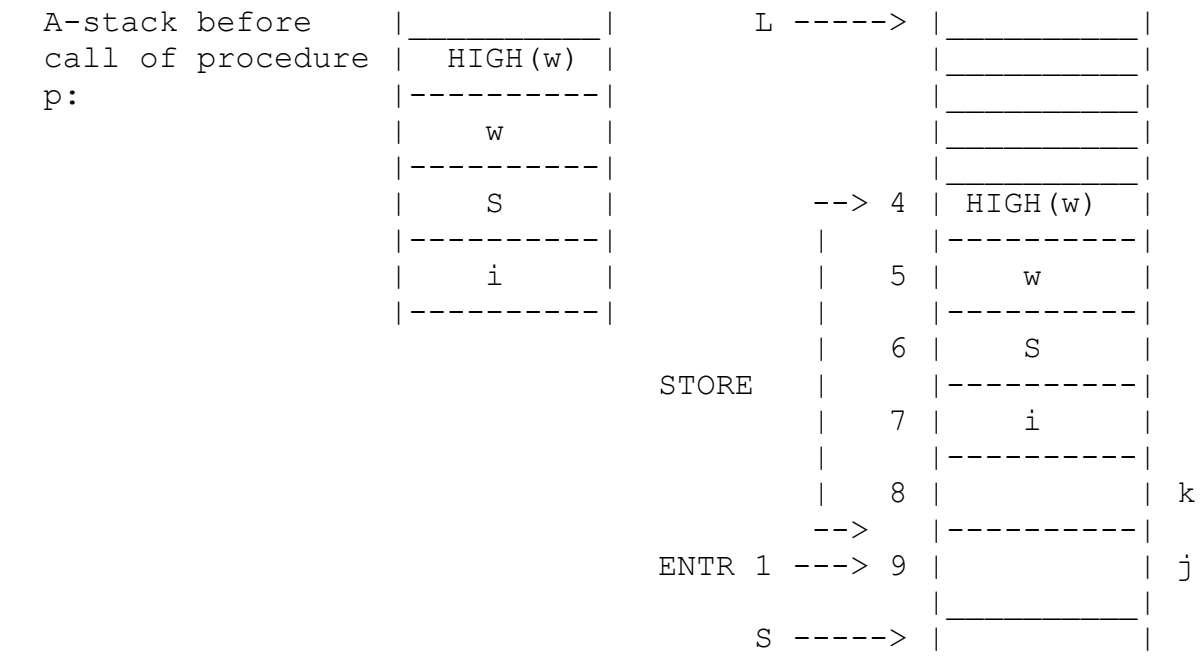
```
                    L'-->   |           |
                            .           .
                            |-----------|
                    S', L-->|G-register|
                            |___________|
                            |    L'     |
                            .-----------.
                    S -->   .           .
                            |           |
```

2.8. Parameter passing

```
MODULE M;

TYPE String=ARRAY [0..255] OF CHAR;

PROCEDURE P(i: INTEGER; S: String; VAR w: ARRAY OF CHAR);
  VAR k,j:INTEGER;
BEGIN

(* parameter saving and variable allocation *)
```

```
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 |                                 STORE ENTR 01    |  (2.8.1)
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

 (* copying of array S called by value *)

 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 |k:=HIGH(S);                        LIB FF  SLW8   |
 |j:=HIGH(w);                        LLW4 SLW9 RTN  |  (2.8.2)
 |                                    |    |        |
 |                                   HIGH  j        |
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 END P;

 VAR
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     |str8:ARRAY[0..7] OF CHAR;  LI2 ALLOC SGW2    |
     |str :String;               LIB 40 ALLOC SGW3 |
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

BEGIN
                     loading 'abc' constant address on A-stack
                                    |
 - - - - - - - - -  - - - - - - -|- - - - - - - - - - - - -
 |p(1,'abc',str8);        LI1 LSTA 0001 LGW2 LI7 CL1 |
 |                                                   |
 |str:='def';             LGW3 LSTA *ind* LI1 MOVE   | (2.8.3)
 |                                                   |
 |p(2,str,str);           LI2 LGW3 LGW3 LIB FF CL1   |
 - - - - - - - - -  - - - - - - - - - - - - - - - - - - - -

 END M.
```

Note 2.8.1. STORE writes A-stack in P-stack, reserving storage for local objects in P-stack: 4-th word contains HIGH(w), 5-th word contains w address, 6-th word contains S address, 7-th word contains i, 8-th word contains the number of parameters. ENTR 01 instruction terminates storage allocation reserving 9-th word for j:



Note 2.8.2. The instructions for array S copy which is called by value are represented here.

```
I)  LIB 40        II)  LLW6          III)  LLW6
    ALLOC              LIB FF              LIB 3F
    COPT               CPCOP 06            PCOP 06
    LLW6
    LIB 40
    MOVE
    SLW6
```

The necessity of introducing instructions CPCOP and PCOP is quite clear. These instructions serve for allocation and copy of multiparameters.

Note 2.8.3. LSTA *ind* by relative address in string pool loads on A-stack the corresponding string constant address. Designated instruction sequence may be replaced by: LGW3 RDS 01 XYZ0, where X,Y,Z are codes of symbols 'x', 'y', 'z'.

2.9. Function call over nonempty stack

```
MODULE M;

PROCEDURE f(i,j: INTEGER): INTEGER;
BEGIN
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
| RETURN  i+j              STORE LLW5 LLW4 ADD RTN |
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
END f;
```

```
PROCEDURE p(i,j: INTEGER);
  VAR k:INTEGER;
BEGIN
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
| k:=i+j;                       STORE LLW5 LLW4 ADD SLW6 RTN |
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
END p;
VAR v: PROCEDURE(INTEGER,INTEGER): INTEGER;
BEGIN
- - - - - - - - - - -- - - - - - - - - - - - - - - - - - - -
| p(1,f(2,3));   LI1     STORE LI2 LI3 CL1 LODFV CL2 |
| v:=f;          LPC 00 01 SGW2                      |
| p(1,v(2,3));   LI1 LGW2 STOFV LI2 LI3 CF LODFV CL2 |(2.9.1)
|                                                    |
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
END M.
```

Note 2.9.1. Instructions sequence LI1 LI2 CL1 CL2 would be wrong because CL1 takes all values from the stack in entry point and 1 is among them but it's not destinated for it.

3. EXPRESSIONS

3.1. Word arrays indexation

```
MODULE M;

VAR x: ARRAY [0..3] OF INTEGER;
    i: INTEGER;

BEGIN
- - - - - - - - - - - - - - - - - - - - - -
|  x[i]:=1;    LGW2 LGW3 LI3 CHKZ LI1 SXW |-- with range
|  i:=x[1];    LGW2 LI1  LXW SGW2         |   check
- - - - - - - - - - - - - - - - - - - - - -

- - - - - - - - - - - - - - - - - - -
|  x[i]:=1;      LGW2 LGW3 LI1 SXW  | -- without range check
|  i:=x[1];      LGW2 LI1  LXW SGW2 |    (3.1.1)
- - - - - - - - - - - - - - - - - - -
END M.
```

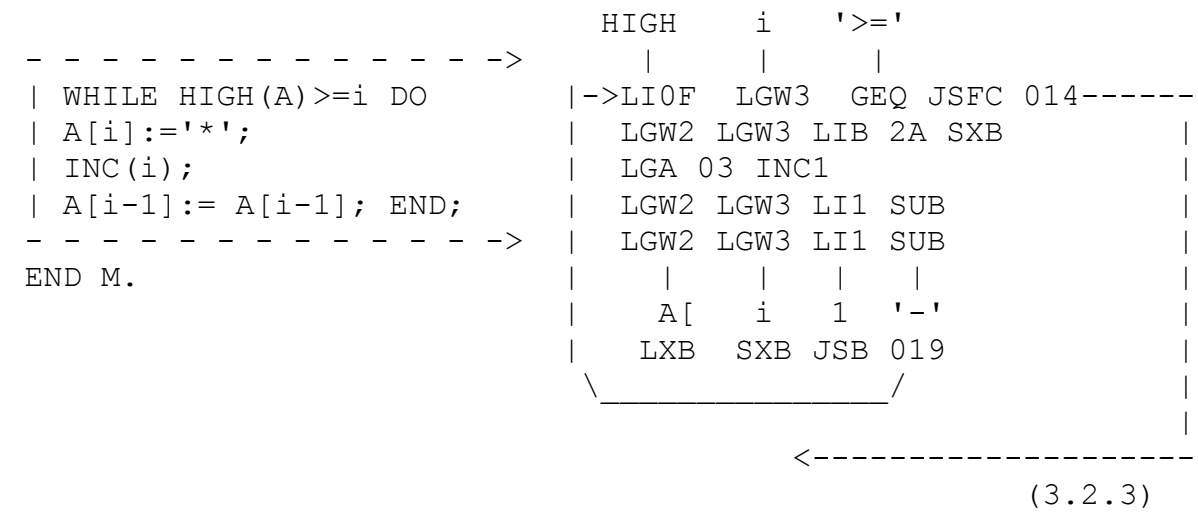
Note 3.1.1. Compiler uses here constant addressing generating next code:

```
- - - - - - - - - - - - - - - - - - - - -
|    i:=x[1]        LGW2 LSW1     SGW2  |
- - - - - - - - - - - - - - - - - - - - -
```

and thereby range checking becomes unnecessary.

3.2. Byte arrays indexation

```
MODULE M;
    - - - - - - - - - - - - - - - - - - - - - - -
VAR | A:ARRAY [0..0Fh] OF CHAR;  LI4 ALLOC SGW2-|--(3.2.1)
    - - - - - - - - - - - - - - - - - - - - - - -
      i:INTEGER;                                    (3.2.2)

BEGIN
- - - - - - - - - - - - - - - - - - - - - - - -
|i:=0;                              LI0 SGW3 |
- - - - - - - - - - - - - - - - - - - - - - - -
```



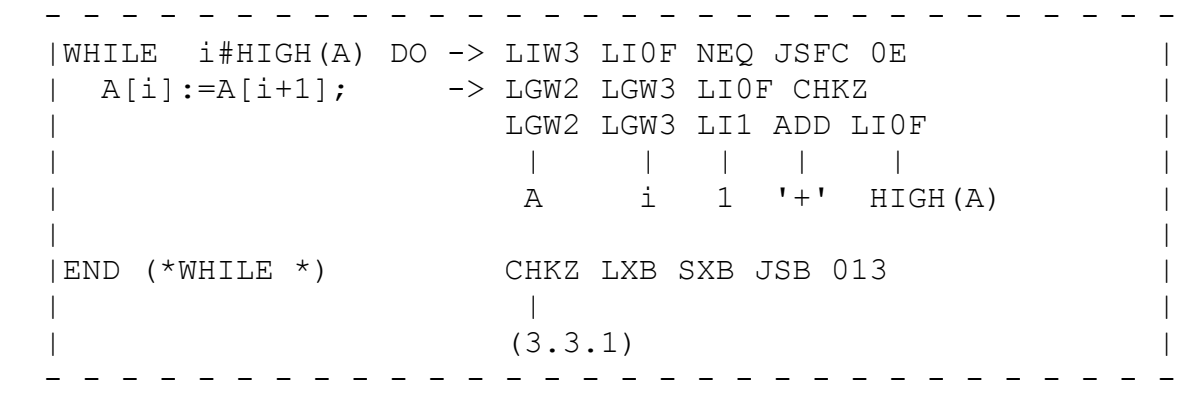
Note 3.2.1. Second global word contains array address.

Note 3.2.2. Char arrays are always packed.

Note 3.2.3. LXB and SXB are similar LXW and SXW but operate over byte. 0 <byte address> LXB and 0 <byte address> SXB realize absolute byte addressing.

3.3. Byte arrays indexation with range check

```
MODULE M;

VAR A:ARRAY [0..0Fh] OF CHAR;
    i:INTEGER;



BEGIN i:=0;
```



Note 3.3.1. CHKZ checks whether the A-stack second element lies between 0 and stack top (which defines a bound). If it's so instruction makes bound deletion or raises TRAP(4Ah) otherwise.

3.4. Range check

```
MODULE M;

VAR x:[10h..20h];
BEGIN
   - - - - - -  - - - - - - - - - - - - - - - - - - - - - - - -
  |x:=13h;                    LIB 13 LIB 10 LIB 20 CHK SGW2 |
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - | - - - -
END M.                                                        |
                                                          (3.4.1)
```

Note 3.4.1. CHK makes range check. This check is made during compilation time by compiler which generates code LIB13 SGW2.

3.5. Operatin over BITSET type object.

```
MODULE M;
VAR b1, b2:BITSET;
BEGIN
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  |b1:={}; b2:={1};                LI0  SGW2  LI2  SGW3 |
  |b1:=b1+b2;                      LGW2  LGW3  OR  SGW3 |
  (*     *                                    AND
         /                                    XOR
         -                                    BIC      *)

  |b1:={1}; INCL(b1,2);            LGA 2  LI2  INCL     |
  |         EXCL(b1.2);            LGA 2  LI2  EXCL     |
 (* 2 IN b1                       LI2  LGW2   IN       *)
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
END M.
```

3.6. ANDJP and ORJP instructions

```
MODULE M;
.
.
BEGIN
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 |IF FALSE AND TRUE THEN END;   LI0  ANDJP 1  LI1  JSFC  |
 |IF TRUE OR FALSE  THEN END;   LI1  ORJP 1   LI0  JSFC  |
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
END M.                                                  (3.6.1)
```

Note 3.6.1. After optimization compiler will generate nothing instead of such funny code.