VRL-1 COMPUTER

Victor Richard Lesser
Computer Science 231
Term Project
Due December 13, 1966
This report was prepared as a term project for Computer Science 231, December 1966. It is issued herein for completeness because other similar reports will appear in this series.
Design Philosophy

The basic design philosophy of this computer was to merge the stack type machine with a general register machine. In attempting to pursue this idea, a lot of thought was given to capturing the essence of how a stack type machine works. In this attempt to understand this type of machine, it was realized that the basic idea was extremely simple; namely a stack operation was just an implicit command to the hardware to fetch the operands in a specific way. The thought immediately arose that maybe the programmer should have some type of facility to describe how operands are to be fetched and where the result is to be stored independent of the basic instruction. Through this scheme, the density of instructions obtained through a stack machine could be achieved while still preserving the flexibility of having many general purpose registers. This machine has been designed to give the programmer this flexibility through a very different type of general register scheme. This general register scheme is set in a machine having 60-bit word length and $2^{18}$ addressable cells.

There has been given thought to how such concepts as program reference table, associative memory for paging, and bounds registers can be incorporated into the general design of this computer, but due to limited time and since these subjects are peripheral to the main purpose of this project, they have not been implemented in the design of this computer.
**TABLE OF CONTENTS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description of General Registers</td>
<td>1</td>
</tr>
<tr>
<td>Description of Instruction Format</td>
<td>6</td>
</tr>
<tr>
<td>Arithmetic Capabilities</td>
<td>10</td>
</tr>
<tr>
<td>Instruction Repertoire</td>
<td>11</td>
</tr>
<tr>
<td>Arithmetic Instructions</td>
<td>12</td>
</tr>
<tr>
<td>Logical Instructions</td>
<td>15</td>
</tr>
<tr>
<td>Testing Instructions</td>
<td>16</td>
</tr>
<tr>
<td>Shifting and Scale Instructions</td>
<td>20</td>
</tr>
<tr>
<td>Jump Instructions</td>
<td>23</td>
</tr>
<tr>
<td>Subroutine Linkage Instructions</td>
<td>24</td>
</tr>
<tr>
<td>Transfer and Exchange Instructions</td>
<td>26</td>
</tr>
<tr>
<td>Partial Word Instructions</td>
<td>27</td>
</tr>
<tr>
<td>Miscellaneous Register Instructions</td>
<td>30</td>
</tr>
<tr>
<td>Interrupt Structure</td>
<td>32</td>
</tr>
<tr>
<td>Input/Output Structure</td>
<td>35</td>
</tr>
<tr>
<td>Appendix - Sample Programs</td>
<td></td>
</tr>
</tbody>
</table>
General Registers

The general register and its ability to be more than just a place to store temporary results is the key element in the design of this computer. There are 32 general registers, each register is a 60-bit word with an associated word of 17 bits which designates the function and usage of the particular general register. A register can represent the address of an operand, a data item, an indexed address of an operand, and a data item with an increment. Additionally, the associated word defines the interaction of this general register with the other general registers. Registers 0, 28, 29, 30, 31 serve the following special purposes: register 0 is always a constant 0; register 28 contains the program counter and the bit position of the last instruction executed plus three 5-bit registers - BPC, BSC, MPR; register 29 is the interrupt product register; register 30 is the interrupt mask register; register 31 is used in input-output operations and interrupt operations.

The following is a description of the fields contained in the associated word of each register, and its usage.

Format of Register K

\[
\begin{array}{c|c|c|c|c|c}
K & A_k & I_k & S_k & D_k & P_k \\
0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]

- \( A_k = 0 \), register \( k \) is a data word.
- \( A_k = 1 \), register \( k \) is an address.
- \( I_k = 0 \), register \( k \) does not self-index.
- \( I_k = 1 \), register \( k \) is self indexing.
- \( S_k \) = the address of the register which is the second operand.
\[ D_k = \text{the address of the register which is the destination of the operation.} \]

\[ B_k = \text{the address of the register which is an index register for register } k. \]

This type of register concept has been designed to be used with a syllable like instruction where the instruction is an operation code and a 5-bit register designator. The associated data word is used to define the operands involved in the particular instruction. The following terminology and functions will be useful in describing how the associated data word is used to fetch the operands for an instruction:

\[ C(K) = \text{the contents of register } k \text{ (the 60-bit data word contained in register } k). \]

\[ M(A) = \text{the 60-bit word contained at memory location } A. \]

\[ V(K) = \text{a function which defines the effective data represented by register } k. \]

\[ T_V(K) = \text{a function which defines the location of an operand; e.g. a general register } K, \text{ or an address.} \]

The value of \( V(K) \) can be defined recursively:

\[
V(0) = 0 \\
V(K) = \begin{cases} 
\text{if } A_k = 0, & C(K) + V(B_k) \\
\text{if } A_k = 1, & M(C(K) + V(B_k)) 
\end{cases}
\]

The value of \( T_V(K) \) can be defined as follows:

\[
T_V(K) = \begin{cases} 
\text{if } A_k = 0, \text{ register } K \\
\text{if } A_k = 1, \text{ the address specified by the lower 18 bits of } [C(K) + V(B_k)] 
\end{cases}
\]
This type of indexing scheme allows the computation of the following type of symbol in one instruction:

\[
A_1 + A_2 + \cdots + A_m + B_1 + B_2 + \cdots + B_n + C_{k_1 + k_2 + \cdots + k_s}.
\]

Let us consider a general arithmetic operation (Op. 1) which operates on two operands and transfers the results to some location. In terms of symbols previously defined, the results of executing \([\text{Op. 1-}K]\) where \(K\) is the specified register can be described by the following:

\[
TV(D_K) \leftarrow V(K) \text{ op. 1 } V(S_K) \quad \text{and} \quad \text{if } I_K = 1 \quad C(K) \leftarrow C(K) + 1.
\]

The following examples show the power of this type of operand fetching scheme. The first example indicates how the statement \(A[D[J]] \text{ op. 1 } B[J] \rightarrow C[I+J]\) could be performed in one syllable operation.

Let us consider the following register configuration:

<table>
<thead>
<tr>
<th>reg 1</th>
<th>reg 2</th>
<th>reg 3</th>
<th>reg 4</th>
<th>reg 5</th>
<th>reg 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>J</td>
<td>I</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

then the instruction op 1-1 would do the following:

\[
TV(D_I) \leftarrow V(1) \text{ op. 1 } V(S_1).
\]
\[ V(1) = M[A + V(B1)] \]
\[ V(B1) = V(4) = M[D + V(B4)] \]
\[ V(B4) = V(5) = J + V(B5) \]
\[ V(B5) = V(0) = 0 \]
\[ V(B4) = J \]
\[ V(B1) = M[D + J] = D[J] \]
\[ V(1) = M[A + D[j^r]] = A[D[J]] \]
\[ V(S1) = V(2) \]
\[ V(2) = M[B + V(S)] = M[B + J] = B[J] \]
\[ TV(D_1) = TV(4) \]
\[ TV(4) = \text{address} \ C(4) + V(B_6) \text{ since } A_4 = 1 \]
\[ V(B_6) = I + V(B_5) = I + J \]
so \[ TV(4) = \text{address} \ C + I + J = C[I + J] \]
\[ C[I + J] \leftarrow A[D[J]] \text{ op } 1 \ B[J] \]

The second example shows how 2 registers can be used so as to act like a stack.

Consider the following register configuration:

<table>
<thead>
<tr>
<th>reg. 1</th>
<th>address of 2nd element of stock</th>
<th>A</th>
<th>I</th>
<th>S</th>
<th>D</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg. 2</td>
<td>contents of top of stock</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

then \( op \ 1-1 \) would do the following: perform \( op \ 1 \) on the top two elements of the stack, pop up the stack, and place the results of \( op \ 1 \) in the top of the stack. The actions can be described as follows:

\[ TV(D_1) \leftarrow V(1) \text{ op } 1 V(S_1) \]
\[ V(i) = M[c(i) + V(p_i)] = M[c(i) + V(0)] = M[c(i)] \]

= contents of the second element of the stack.

\[ V(S_i) = V(z) = c(z) \]

= top elements of the stack

\[ TV(S_i) = TV(z) = \]

= register \& since \( A_2 = 0 \).

So we have register 2 \( \leftarrow \) (the 2nd element of the stack) op 1 (top stack element), and \( C(1) \leftarrow C(1) + 1 \), so the stack is popped up.
Description of Instructions

Word Format

A computer word consists of 60 bits, and may be interpreted as one 60 bit data word, 5-12 bit instructions, 2-30 bit instructions, or 1-30 bit instructions, and 2-12 bit instructions. In that last case, a 6-bit NOP is used to fill out the 60 bit word. There are three general classes of instructions:

Implicit mode:

This is a 12-bit instruction, with a 7-bit operation code, and a 5-bit register designator. There are usually three operands associated with a specific instruction in this class. These operands are obtained through an analysis of 17 bits associated with the specified register. This instruction is intended to be used in loops where the same type of operation is performed repeatedly. For this type of use, it is felt that this instruction type will be extremely efficient.

Arbitrary mode:

This is a 30-bit instruction with a 7-bit operation code, a self index mode bit, and 3 operand registers and mode bits for each register. An instruction in this class performs similar actions as its implicit mode instruction counterparts, except the first level of operand fetching is determined by the information in the instruction rather than from the
associated bits of the source register. The G bits indicate whether a register is to be considered as a data or an address, the F bits indicate whether the B register field of the specified register will be used as an index, and I specifies whether the source register R_l will be auto-indexed at the end of the operation. If we designate VA(K) as the effective contents of a register then

\[
VA(K) = \begin{cases} 
  if \ G_F = 0 and F_K = 0; & C(K) \\
  if \ G_K = 1 and F_K = 0; & M(C(K)) \\
  if \ G_K = 0 and F_K = 1; & C(K) + V(B_K) \\
  if \ G_K = 1 and F_K = 1; & M(C(K) + V(B_K)) 
\end{cases}
\]

notice G,F only apply to first level of addressing; and TVA(K) as the location of destination:

\[
TVA(K) = \begin{cases} 
  if \ G_K = 0; \text{ register } K \\
  if \ G_K = 1; \text{ memory location } C(K) \\
  if \ G_K = 1; \text{ memory location } C(K) + V(B_K) 
\end{cases}
\]

This mode allows an instruction to use a general register in any of the four modes:

1) contents only;
2) address only;
3) contents with increment;
4) indexed address.
This gives much more flexibility to an arbitrary mode instruction over an implicit mode but the former is $2\frac{1}{2}$ times as long in length.
Arithmetics

There are two modes of arithmetics: fixed and floating point. In fixed point arithmetics a data word represents a 60-bit number in 2's complement notation. In floating point arithmetic, the 60-bit floating point word has the following format:

\[
\begin{array}{c|c|c}
\text{S} & \text{B} & \text{K} \\
0 & 1 & \text{11-12} & \text{59}
\end{array}
\]

S = sign of coefficient K.
B = a biased exponent of 20008.
K = an integer coefficient.

In order to facilitate multiprecision operations, there exists a 5-bit MPR register which designates a second register in arithmetic operations. This second register will contain the residue of rounding operations in floating point operations, the top 60 bits of an integer multiply, an indicator for arithmetic overflow etc. Arithmetic operations resulting in underflow, overflow, etc., will be interruptable conditions. It is hoped that this brief summary will be enough to indicate the arithmetic capabilities of this machine, and its ability to handle programmed multiprecision arithmetic.
Instruction Repertoire

Instruction will be grouped according to their general use. All instructions of the immediate mode will be prefixed by I; likewise arbitrary mode instructions will be prefixed by J, and special mode by S. The exact effect of instruction will be given in terms of symbols already defined, e.g. V(K), TV(K). It will be assumed that implicit modes instruction have the format described previously: OP-K, and likewise arbitrary mode instructions: OP-F₁F₂F₃G₁G₂G₃IR₁R₂R₃.
Arithmetic Instructions

Fixed Point Arithmetics

IADD \( TV(D_K) \leftarrow V(K) + V(S_K) \)

JADD \( TVA(R_2) \leftarrow VA(R_1) + VA(R_2) \)

Integer add - forms the sum of two operands in integer format. Arithmetic overflow causes bit 59 of MPR register to be complemented, and is an interruptable condition.

ISUB \( TV(D_K) \leftarrow V(K) - V(S_K) \)

JSUB \( TVA(R_2) \leftarrow VA(R_1) - VA(R_2) \)

Integer subtract - forms the difference of two operands in integer format.

IMUL \( TV(D_K) \leftarrow [V(K) \times V(S_K)]_{60-119} \)

JMUL \( TVA(R_2) \leftarrow [VA(R_1) \times VA(R_2)]_{60-119} \)

Integer multiply - forms the product of two operands in integer format. The top 60 bits of 120 bit product is stored in the registers designated by the MPR.

IDIV \( TV(D_K) \leftarrow \text{Quotient of } [V(K)/V(S_K)] \)

JDIV \( TVA(R_2) \leftarrow \text{Quotient of } [VA(R_1)/VA(R_2)] \)

Integer divide - forms the quotient and remainder of two operands in integer format. The remainder is stored in the register designated by the MPR.
Floating Point Operations

The residue of all floating point operations caused by rounding and normalization will be stored in the register designated by the MPR.

\[
\begin{align*}
\text{IADDF} & \quad TV(D_k) \leftarrow V(K) + V(S_k) \\
\text{JADDF} & \quad TVA(R_3) \leftarrow VA(R_1) + VA(R_2)
\end{align*}
\]

Floating Add - Forms the sum of two operands packed in floating point format.

\[
\begin{align*}
\text{ISUBF} & \quad TV(D_k) \leftarrow V(K) - V(S_k) \\
\text{JSUBF} & \quad TVA(R_3) \leftarrow VA(R_1) - VA(R_2)
\end{align*}
\]

Floating Subtract - Forms the difference of two operands packed in floating point format

\[
\begin{align*}
\text{IMULF} & \quad TV(D_k) \leftarrow V(K) \times V(S_k) \\
\text{JMULF} & \quad TVA(R_3) \leftarrow VA(R_1) \times VA(R_2)
\end{align*}
\]

Floating Multiply - Forms the product of two operands packed in floating point format.

\[
\begin{align*}
\text{IDIVF} & \quad TV(D_k) \leftarrow V(K) / V(S_k) \\
\text{JDIVF} & \quad TVA(R_3) \leftarrow VA(R_1) / VA(R_2)
\end{align*}
\]

Floating Divide - Forms the quotient of two operands packed in floating point format.

\[
\begin{array}{cccccccc}
\text{SADX} & OPCODE & F & G & R & S & Y \\
0 & 6 & 7 & 9 & 10 & 11 & 14 & 15 & 16 & 24
\end{array}
\]
Add to Exponent - This instruction adds a signed value field "y" to the signed exponent of the quantity represented by VA(R₁) and the result is stored in TVA(R₁). If VA(R₁) = 0, the instruction is a NOP. If VA(R₁) ≠ 0, and is negative, VA(R₁) will be completed before the add operation and recompleted after the operation.

SMPR-K

Set MPR Register - the MPR register is set to K.
Logical Instructions

**IAND**

$$[TV(D_k)]_c \leftarrow [\overline{V(k)}]_c \land [\overline{V(S_k)}]_c$$

**JAND**

$$[TVA(R)]_c \leftarrow [\overline{VA}(R)]_c \land [\overline{VA}(R_2)]_c$$

**Logical And**

Forms a bit by bit logical and operation of the two operands.

**IOR**

$$[TV(D_k)]_c \leftarrow [\overline{V(k)}]_c \lor [\overline{V(S_k)}]_c$$

**JOR**

$$[TVA(R)]_c \leftarrow [\overline{VA}(R)]_c \lor [\overline{VA}(R_2)]_c$$

**Logical Or**

Forms a bit by bit logical or operation of the two operands.

**IEXOR**

$$[TV(D_k)]_c \leftarrow [\overline{V(k)}]_c \lor [\overline{V(S_k)}]_c$$

**JEXOR**

$$[TVA(R)]_c \leftarrow [\overline{VA}(R)]_c \lor [\overline{VA}(R_2)]_c$$

**Exclusive Or**

Forms a bit by bit logical exclusive or of the two operands.

**IIMP**

$$[TV(D_k)]_c \leftarrow [\overline{V(k)}]_c \Rightarrow [\overline{V(S_k)}]_c$$

**JIMP**

$$[TVA(R)]_c \leftarrow [\overline{VA}(R)]_c \Rightarrow [\overline{VA}(R_2)]_c$$

**Implication**

Forms a bit by bit logical implication of the two operands.

**IEQV**

$$[TV(D_k)]_c \leftarrow [\overline{V(k)}]_c \Leftrightarrow [\overline{V(S_k)}]_c$$

**JEQV**

$$[TVA(R)]_c \leftarrow [\overline{VA}(R)]_c \Leftrightarrow [\overline{VA}(R_2)]_c$$

**Equivalence**

Forms a bit by bit logical equivalence of the two operands.
Testing Instructions

If the condition is satisfied after the designated operation a
skip exit is performed (skip the next syllable) otherwise the next normal
instruction is executed.

The following set of instructions all perform the same function
except they branch on different conditions: the immediate instruction
performs $TV(K) \leftarrow V(K) + V(S_K)$, and the arbitrary instruction performs
$TVA(R_1) \leftarrow VA(R_1) + VA(R_2)$. Notice if $R_2 = 0$ or $S_K = 0$, this point of
instruction has no effect.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITEQ</td>
<td>$V(K) = V(D_K)$</td>
<td>Increment and branch on equal</td>
</tr>
<tr>
<td>JTEQ</td>
<td>$VA(P_i) = VA(P_3)$</td>
<td>Increment and branch on equal</td>
</tr>
<tr>
<td>JTNE</td>
<td>$V(K) \neq V(D_K)$</td>
<td>Increment and branch on not equal</td>
</tr>
<tr>
<td>JTNE</td>
<td>$VA(P_i) \neq VA(P_3)$</td>
<td>Increment and branch on not equal</td>
</tr>
<tr>
<td>ITGE</td>
<td>$V(K) \geq V(D_K)$</td>
<td>Increment and branch on less than or equal</td>
</tr>
<tr>
<td>JTGE</td>
<td>$VA(P_i) \geq VA(P_3)$</td>
<td>Increment and branch on greater than or equal</td>
</tr>
</tbody>
</table>
**Logical Comparison Test** - tests the logical product of two operands for equality with a third operand.
The following additional testing instructions have already been
duplicated by other instructions but these specific instructions give
much more capability to the syllable mode instruction, without requiring
much in hardware addition.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IKEQ</td>
<td>SKIP EXIT IF C(K) = 0</td>
</tr>
<tr>
<td>IKNE</td>
<td>SKIP EXIT IF C(K) ≠ 0</td>
</tr>
<tr>
<td>IKLE</td>
<td>SKIP EXIT IF C(K) ≤ 0</td>
</tr>
<tr>
<td>IKGE</td>
<td>SKIP EXIT IF C(K) ≥ 0</td>
</tr>
<tr>
<td>IKLT</td>
<td>SKIP EXIT IF C(K) &lt; 0</td>
</tr>
<tr>
<td>IKGT</td>
<td>SKIP EXIT IF C(K) &gt; 0</td>
</tr>
<tr>
<td>IVEQ</td>
<td>SKIP EXIT IF V(K) = 0</td>
</tr>
<tr>
<td>IVNE</td>
<td>SKIP EXIT IF V(K) ≠ 0</td>
</tr>
<tr>
<td>IVLE</td>
<td>SKIP EXIT IF V(K) ≤ 0</td>
</tr>
<tr>
<td>IVGE</td>
<td>SKIP EXIT IF V(K) ≥ 0</td>
</tr>
<tr>
<td>IVLT</td>
<td>SKIP EXIT IF V(K) &lt; 0</td>
</tr>
<tr>
<td>IVGT</td>
<td>SKIP EXIT IF V(K) &gt; 0</td>
</tr>
<tr>
<td>IZEQ</td>
<td>SKIP EXIT IF M[C(K)] = 0</td>
</tr>
<tr>
<td>IZNE</td>
<td>SKIP EXIT IF M[C(K)] ≠ 0</td>
</tr>
<tr>
<td>IZLE</td>
<td>SKIP EXIT IF M[C(K)] ≤ 0</td>
</tr>
<tr>
<td>IZGE</td>
<td>SKIP EXIT IF M[C(K)] ≥ 0</td>
</tr>
<tr>
<td>IZLT</td>
<td>SKIP EXIT IF M[C(K)] &lt; 0</td>
</tr>
<tr>
<td>IZGT</td>
<td>SKIP EXIT IF M[C(K)] &gt; 0</td>
</tr>
</tbody>
</table>
BIT 7 =
- 0, BP represents a (7-bit) bit position
- 1, BP represents $F_2G_2R_2$ format, where the bit position is the value of VA(R2).

BIT 8 =
- 0, SKIP EXIT IF VA(R1) BP = 0
- 1, SKIP EXIT IF VA(R1) BP = 1

= 0, NO ACTION

BIT 9-10 =
- 1, SET BIT VA(R1) BP = 1
- 2, CLEAR BIT VA(R1) BP = 0
- 3, COMPLEMENT BIT VA(R1) BP = VA(R1) BP + 1

BIT 12-13 =
- 0, NO ACTION
- 1, SHIFT RIGHT 1 BIT
- 2, SHIFT LEFT 1 BIT

Bit sensing - this instruction is multipurpose and can accomplish the following functions: bit sensing, bit setting; storage shift; storage skip. The instruction tests the BPth bit of the operand specified by VA(R1), and performs an action according to the operate bits.
Shifting and Sealing Instruction

\[
\text{SHFT}
\]\n
- \( B7 = 0 \), \( SC \) represent a 7-bit shift count
- \( B7 = 1 \), \( SC \) represents \( F \_G \_R \_R \_G \) where \( VA(R_2) \) is the shift count
- \( B8 = 0 \), direction of shift depending upon \( BIT \_9 \)
- \( B8 = 1 \), direction of shift depends upon sign of shift count, positive right, negative left
- \( B9 = 0 \), right shift
- \( B9 = 1 \), left shift
- \( B10 = 0 \), end off shifting
- \( B10 = 1 \), end around shifting
- \( B11 = 0 \), no sign extension
- \( B11 = 1 \), sign extension

Single word shift - this is a shift operation on the operand specified by \( VA(R_1) \) where the particular type of shift is dependent upon the operate bits.

\[
\text{SHFTD}
\]\n
- \( BIT \_7 = 0 \), end off shifting
- \( BIT \_7 = 1 \), end around shifting
BIT 8 =
  = 0, end off shifting
  = 1, end around shifting

Double word shift - forms the 120-bit operand represented by VA(R₁) - VA(R₂). The shift count is VA(R₃) where its sign determines the direction of shift.

Double word shift right (left) - forms the 120-bit operand represented by VA(R₁) - VA(R₂). The shift count is SC where the direction is right (left) depends upon whether the instruction is SHFDR, SHFDL. The operate bits are interpreted the same as the bits in SHFTD.

Scale operation - Shift the operand VA(R₁) left circularly until the most significant digit is to the right of bit 0. Shift count "SC" is reduced by one for each shift. The operation terminates if SC = 0, or the most significant digit is to the right of bit 0. At the end of the operation,
TV(R₃) ← the number of shifts executed. If R₃ = 0, this operation is ignored.
Jump Instructions

IJSF  Jump forward K-syllables, this instruction can be used only when it starts at either bits 0, 12, 24, 36, 48.

IJSB  Jump backward K-syllable, the same restriction of usage as above.

IJTF  Jump forward K-syllable, starting from beginning of next full word.

IJTB  Jump backward K-syllables, starting from end of last full word.

IJWF  Jump forward K-words, start executing next instruction at \( P+K \), where \( P \) is current program counter.

IJWB  Jump backward K-words, starting executing next instruction at \( P-K \), where \( P \) is current program counter.

IJPV  Jump to \( V(K) \)

IJPC  Jump to \( C(K) \)

IJPI  Jump to \( C(K) + V(B_K) \)

SJPV  Jump to \( y + V(K) \)

SJPC  Jump to \( y + C(K) \)

SCPI  Jump to \( y + C(K) + V(B_K) \)
Subroutine Linkage

There are two different types of subroutine calls, one oriented towards the non-recursive procedure, the other a recursive procedure.

SNRC

Non-recursive subroutine call; this type of instruction is normally called a return jump. The instruction jumps to the address represented by \( y + V(K) + 1 \), and stores the instruction SJPV 0 \([\text{program counter} + 1]\) in the word \( y + V(K) \). Therefore in order to exit from the called routine, just an unconditional jump to \( y + V(K) \) will accomplish the task.

INRC

A non-recursive subroutine call in syllable mode. In this instruction \( V(K) \) replace \( y + V(K) \) in SNRC.

SPRC

Recursive subroutine call; the following actions are taken. A jump to address specified by \( y \); \( TV(K) \leftarrow V(D_K) \); \( C(K) \leftarrow C(K) - 1 \); \( TV(D_K) \leftarrow [\text{program counter} + 1] \). This assumes the two registers \((K, D_K)\) configured as a stack as described previously. So essentially, this is a push down stack operation, with top element of stack becoming the program counter + 1.
IPRE

Recursive subroutine exit; this instruction does an unconditional jump to the address specified by the top of the stock, and then pops up the stack:

\[
\begin{align*}
\text{jump to } TV(D_K) \\
TV(D_K) &\leftarrow V(K) \\
C(K) &\leftarrow C(K) + 1
\end{align*}
\]
Transfer and Exchange Instructions

ITRV \( TV(D_K) \leftarrow V(K) \)

ITRC \( TV(D_K) \leftarrow C(K) \)

IRCV \( C(K) \leftarrow V(D_K) \)

ITRD \( TV(D_K) \leftarrow C(K) + V(B_K) \)

IRCC \( C(D_K) \leftarrow C(K) \)

IEXV \( TV(D_K) \leftarrow V(K), TV(K) \leftarrow V(D_K) \)

IEXC \( TV(D_K) \leftarrow C(K); TV(K) \leftarrow C(D_K) \)

IEXCC \( C(D_K) \leftarrow C(K); C(K) \leftarrow C(D_K) \)

SEXC

\[
\begin{array}{cccccc}
\text{OP} & \text{CO} & \text{PC} & F_2G_2R_2 & F_6G_6P_6 \\
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 & 3 \\
\end{array}
\]

OPERATE BITS

Bits 7-8

0 \( TVA(R_1) \leftarrow VA(R_2) \)

\( TVA(R_2) \leftarrow VA(R_1) \)

1 \( TVA(R_1) \leftarrow VA(R_2) \)

\( TVA(R_2) \leftarrow 0 \)

2 \( TVA(R_1) \leftarrow 0 \)

\( TVA(R_2) \leftarrow VA(R_1) \)

3 \( TVA(R_1) \leftarrow 0 \)

\( TVA(R_2) \leftarrow 0 \)
Byte Operations

In using these byte instructions, two pieces of data will be implicit: the size of the byte; and its position in the source word. This information is obtained by the BSC, and BPC registers each 5 bits. They point to registers whose effective contents: V(K) determine the value for each of the parameters.

IBSC  BSC ← K

IBPC  BPC ← K

STBY

```
<table>
<thead>
<tr>
<th>V(BPC)</th>
<th>V(BSC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

OPERATE BITS

BIT 7 =

= 0; insert store

= 1; clear destination operand VA(R2) ← 0;

BIT 8 =

= 0; BP2 ← V(BPC) - align store with data store

= 1; BP2 ← 59 - align store with right-most bit of register

BIT 9-10 =

= 0, no action

= 1, V(BPC) ← V(BPC) + V(BCS)

= 2, V(BPC) ← V(BPC) - V(BCS)

Transfer Byte - This is a comprehensive byte manipulation instruction.

The V(BPC) represents the position of right-most bit of the byte.

Depending upon the mode bits the following thing will happen:

\[
\begin{align*}
&[VA(F_1)]_{BP_1 - V(BCS)} \leftarrow [VA(F_1)]_{BP_1 - V(BCS)} \\
&\text{Through bits 9-10 you can do an indexed transfer through the data word.}
\end{align*}
\]

\[\text{Note: BIT 14 =}\]

\[
\begin{cases}
0, & BP_1 ← V(BPC) \\
-59 & \text{aligns with register bit of negative}
\end{cases}
\]
STCB

Compare Byte - It works exactly the same as the transfer byte instruction except a comparison operation instead of a transfer is done. Bits 11-13 indicate the skip conditions for this test instruction.

<table>
<thead>
<tr>
<th>BITS 11-13</th>
<th>SKIP IF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>=</td>
</tr>
<tr>
<td>1</td>
<td>≠</td>
</tr>
<tr>
<td>2</td>
<td>&lt;</td>
</tr>
<tr>
<td>3</td>
<td>&gt;</td>
</tr>
<tr>
<td>4</td>
<td>≤</td>
</tr>
<tr>
<td>5</td>
<td>≥</td>
</tr>
</tbody>
</table>
Stock Operation Codes

The necessary instructions for popping up the stack have already been developed, but there must be some capability to push down the stack.

**IPSH**

\[
\begin{align*}
TV(K) & \leftarrow V(D_K) \\
TV(D_K) & \leftarrow V(S_K) \\
C(K) & \leftarrow C(K) - 1
\end{align*}
\]

Push down stack syllable

**JPSH**

\[
\begin{align*}
TV(R_1) & \leftarrow V(R_2) \\
TV(R_2) & \leftarrow V(R_3) \\
C(R_1) & \leftarrow C(R_1) - 1
\end{align*}
\]

Arbitrary mode push down stack instruction.

**SPSH**

\[
\begin{array}{c|c|c}
\hline
D & L & \delta \\
\hline
k & \eta & \gamma \\
\hline
\end{array}
\]

\[
\begin{align*}
TV(K) & \leftarrow V(D_K) \\
TV(D_K) & \leftarrow y \\
C(K) & \leftarrow C(K - 1)
\end{align*}
\]

Push down stack with immediate constant.
Miscellaneous Register Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICLV</td>
<td>V(K) ← 0</td>
</tr>
<tr>
<td>ICLC</td>
<td>C(K) ← 0</td>
</tr>
<tr>
<td>ICLR</td>
<td>A_k ← 0; I_k ← 0; S_k ← 0; D_k ← 0; B_k ← 0</td>
</tr>
<tr>
<td>ICLRA</td>
<td>A_k ← 0;</td>
</tr>
<tr>
<td>ICLRI</td>
<td>I_k ← 0;</td>
</tr>
<tr>
<td>ISLRA</td>
<td>A_k ← 1;</td>
</tr>
<tr>
<td>ISLRI</td>
<td>I_k ← 1;</td>
</tr>
<tr>
<td>SRMB</td>
<td>![SRMB Diagram]</td>
</tr>
</tbody>
</table>

Set associated bits of register K;

\[
A_k ← A; I_k ← I; S_k ← S; D_k ← D; B_k ← B. 
\]

The following instructions have this format:

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
0 & 6 & 7 & 8 & 11 & 12 & 13 & 14 & 15 \\
\hline
\end{array}
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
</table>
| STRML       | \[
\left[M(y)\right]_{30-59} ← \begin{cases} 0 & \text{if instruction} \\ \text{SRMB } & k \ A_k \ I_k \ S_k \ D_k \ B_k \end{cases} 
\] |
| STRMU       | \[
\left[M(y)\right]_{0-29} ← \begin{cases} 0 & \text{if instruction} \\ \text{SRMB } & k \ A_k \ I_k \ S_k \ D_k \ B_k \end{cases} 
\] |

This instruction is used by the interrupt routine to save the register setting. A 30-bit instruction is created which will reset the associated bits of the particular register when executed.

S L D A

\[
\begin{aligned}
&c(K) ← y; \ A_k ← 1; I_k ← 0; S_k ← 0; D_k ← 0; \\
&B_k ← 0;
\end{aligned}
\]
This instruction is used to load a simple address

\[
\text{SLDD} \quad C(k) \leftarrow 0; A_k \leftarrow 0; T_k \leftarrow 0; S_k \leftarrow 0; D_k \leftarrow 0; P_k \leftarrow 0
\]

This instruction is useful in loading an immediate constant

\[
\text{SCDD} \quad C(k) \leftarrow C(k') + \frac{y}{d}
\]

This instruction is used as an immediate add

\[
\text{ICAD} \quad C(D_k) \leftarrow C(k) + C(S_k)
\]

\[
\text{ICSB} \quad C(D_k) \leftarrow C(k) - C(S_k)
\]

Contents add $y$ subtract syllable mode — this gives you capability of adding to registers which are address increments.
**Interrupt Structure**

There are 60 conditions which can cause an interrupt of instruction execution in a main program. The class of interrupts is divided into two categories; internal interrupts and external interrupts. An internal interrupt condition is caused by the main program, e.g. arithmetic overflow, underflow, etc. The external interrupt originates from a particular I/O channel, e.g. completion of operations or a channel; end of tape (for magnetic tape), character transfer from teletype. The internal interrupt conditions are always sensed by the computer, while the particular I/O condition whose interrupt is desired must be selected by the programmer. In order to give the programmer the option of selectively processing the interrupt conditions previously selected, an interrupt mask register and active/deactive interrupt system instruction is provided. An interrupt condition is processed in the following way, each of the 60 conditions is wired to a particular bit in a 60-bit data word called the interrupt register; if a condition is sensed the corresponding bit in the interrupt register will be set to one; an interrupt of main program instruction execution will occur if, and only if, the interrupt system is active and the logical product of the interrupt register and interrupt mask register is non-zero; this product is contained in the interrupt product register and is computed even if the interrupt system is inactive; this allows identification of other interrupts while processing an interrupt. The interrupt conditions are numbered 0-59 corresponding to their bit position in the interrupt register. If interrupt conditions occur simultaneously, the one with the lowest numbered position will be used in the discussion following. In order to facilitate the quick
execution of interrupt subroutines a duplicate set of 32 general registers is provided. The following actions occur when an interrupt is processed:

1) finish processing of last main program instruction;
2) deactivate the interrupt system;
3) switch to other set of general registers;
4) insert address and bit position of next instruction to be processed in main program in bits 0-23 of register 31;
5) perform unconditional jump to address \( P + y \), where \( P \) is program counter of new set of registers, and \( y \) is interrupt condition number.

When the interrupt processing subroutine is finished, all that is necessary to do is switch the general register and active the interrupt system, and the main program execution will continue where left off.

There is the possibility that upon processing the interrupt it is not desired to return to the same place in the main program. This is accomplished by an instruction which exchanges the set of general registers but places a new address for program counter in set of main program general registers.

The interrupt instructions are

\[
\text{SIRT} \quad \begin{array}{cccc}
\text{CP} & \text{OP} & \text{PE} & \text{M} \\
0 & 6 & 7 & 8 \\
\end{array}
\]

\[
\text{BIT 7} =
\begin{align*}
0 & , \text{active interrupt system} \\
1 & , \text{do not active interrupt system}
\end{align*}
\]
Return from interrupt - Exchange general registers

SEIR

BIT 7 = 0, do not activate interrupt system
= 1, activate interrupt system

Exchange general registers - Exchange register sets, and replace program counter in new register by y.
Input/Output

This computer will have the normal complement of I/O instructions. The only unused features are that when selecting a device on a channel (these can be up to 60-x channels, where x indicates the number of internal interrupts recognizable) a unique number can be assigned to this device. In addition, a channel can be set to single character input mode. In this mode when a device on a channel is ready to transfer a character, the particular channel will be set for an interrupt condition. There is then a special instruction which will input the characters and device number into specific general registers.

SIPC

Input character; inputs a character from a device of channel CN and place the character in TVA(R₁), and the number of the device inputing the character is stored in TVA(R₂). The instruction will perform a skip exit if the channel has a character to input; otherwise it will transfer the status bits of the channel to TVA(R₁), and perform a normal exit. Therefore, we can differentiate interrupts conditions deriving from input character transfer versus other channel interrupt conditions. The interrupt condition will be cleared upon performing this instruction.
APPENDIX - SAMPLE PROGRAMS
Matrix Multiply

\[ A_{m \times n} \cdot B_{n \times l} \rightarrow C_{m \times l} \]

\[
\begin{align*}
A[i, j] &= A + (i-1)N + (j-1) \\
B[i, j] &= B + (j-1)L + (k-1) \\
C[i, j] &= C + (i-1)L + (k-1)
\end{align*}
\]

\[
C[i, j] = \sum_{j=1}^{n-1} A[i, j] \times B[j, k]
\]
\[
= \sum_{j=0}^{n-1} \left[ A + (i-1)N + j \right] \times \left[ B + j \cdot L + (k-1) \right]
\]

ASSUME GENERAL REGISTER HAVE THIS INITIAL FORM

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>(+I,11)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>(+J,11)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>(+H,11)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>(+I,11)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>N</td>
<td>(+J,11)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>L</td>
<td>(+K,11)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>O</td>
<td>(+I)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>O</td>
<td>(+J)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>O</td>
<td>(+K)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>O</td>
<td>(TEMP)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>O</td>
<td>(SUM)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>B</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**Matrix Multiply Program**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMULF-1</td>
<td>( V(10) \leftarrow V(1) + V(8) )</td>
</tr>
<tr>
<td></td>
<td>( \text{TEMP} \leftarrow A[I, J] \cdot B[J, K] )</td>
</tr>
<tr>
<td>ITLT-8</td>
<td>( V(8) \leftarrow V(8) + V(12) ); \text{SKIP} ( V(8) \leq V(5) )</td>
</tr>
<tr>
<td></td>
<td>( J \leftarrow J + 1 ); \text{SKIP IF} ( J \geq N )</td>
</tr>
<tr>
<td>JSF-4</td>
<td>\text{JUMP FORWARD 4 SYLABLES}</td>
</tr>
<tr>
<td>ICAD-6</td>
<td>( C(2) \leftarrow C(6) + C(22) )</td>
</tr>
<tr>
<td></td>
<td>( C(22) \leftarrow B \cdot \text{base address} + (J-1) \cdot L )</td>
</tr>
<tr>
<td>ADDF-10</td>
<td>( V(11) \leftarrow V(10) + V(11) )</td>
</tr>
<tr>
<td></td>
<td>( \text{SUM} \leftarrow \text{SUM} + \text{TEMP} )</td>
</tr>
<tr>
<td>JSB-5</td>
<td>\text{JUMP BACKWARDS 5 SYLABLES}</td>
</tr>
<tr>
<td></td>
<td>\text{GO TO INSTRUCTION 4}</td>
</tr>
<tr>
<td>ITRU-11</td>
<td>( V(3) \leftarrow V(11) )</td>
</tr>
<tr>
<td></td>
<td>( C[I, K] \leftarrow \text{SUM} )</td>
</tr>
<tr>
<td>ICLV-8</td>
<td>( V(8) \leftarrow 0 )</td>
</tr>
<tr>
<td></td>
<td>( J \leftarrow 0 )</td>
</tr>
<tr>
<td>ICLV-11</td>
<td>( V(11) \leftarrow 0 )</td>
</tr>
<tr>
<td></td>
<td>( \text{SUM} \leftarrow 0 )</td>
</tr>
<tr>
<td>IEC-1</td>
<td>( C(2) \leftarrow C(13) )</td>
</tr>
<tr>
<td></td>
<td>( C(2) \leftarrow B \cdot \text{base address} )</td>
</tr>
<tr>
<td>ITGE-9</td>
<td>( V(9) \leftarrow V(9) + V(12) ); \text{SKIP IF} ( V(9) \leq V(1) )</td>
</tr>
<tr>
<td></td>
<td>( K \leftarrow K + 1 ); \text{SKIP IF} ( K \leq k )</td>
</tr>
<tr>
<td>JSB-11</td>
<td>\text{JUMP BACKWARDS 11 SYLABLES}</td>
</tr>
<tr>
<td></td>
<td>\text{GO INSTRUCTION 4}</td>
</tr>
<tr>
<td>ICAD-5</td>
<td>( C(2) \leftarrow C(7) + 5 )</td>
</tr>
<tr>
<td></td>
<td>( C(2) \leftarrow A \cdot \text{base address} + (I-1) \cdot L )</td>
</tr>
<tr>
<td>Line</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>14</td>
<td>ICHD-3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>ITGD-7</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>JSB-15</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

This program loop for matrix multiply was done in 16 syllables, which is 3/5 computer words.
Linear Search

Search the array L containing N items, each
encoded under a spot in the element B.

Consider the following general register configuration

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>I</th>
<th>S</th>
<th>D</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>N.K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Linear Search programs

1 ITNE-1
   SKIP IF V(1) ≠ V(2)
   SKIP IF M[L+J.K] = B

2 FINISHED
   V(3) specifies the element in the
   array which = B

3 ITGE-3
   V(3) ← V(3) + V(4)
   SKIP IF V(3) ≤ V(5)
   C(3) ← J.K

4 IJSB-3
   JUMP BACKWARD 5 SYLLABLES
   GO TO INSTRUCTION 1

5 NO ANSWER
Binary Search

We want to locate the address of a word between addresses A and B so the words equal to the element D where M[A+I] = M[A+I+1] for all I ≤ B-A.

Algorithm
1) Compute \( \frac{A+B}{2} \), if \( \frac{A+B}{2} = B \) go to step 5
2) if \( M(\frac{A+B}{2}) \leq D \), no go to step 4
3) \( A \leq \frac{A+B}{2} \), go to step 1
4) \( B \leq \frac{A+B}{2} \), go to step 1
5) if \( M(A) = D \), then A is correct address, step
6) if \( M(B) = D \), then B is correct address, step
7) neither 5 or a true solution
Binary Search Programs

Consider following register configuration:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
</tr>
</tbody>
</table>

1. IRCV-1
   \[ c(1) \leftarrow v(3) \]
   After first step, means \((A+B)/2 \rightarrow A\)

2. IADD-1
   \[ v(3) \leftarrow v(1) + v(2) \]
   form \(c(3) \leftarrow (A+B)\)

3. SHIFT
   (Shift right \(1\) bit, end off, register)
   \(c(3) \leftarrow (A+B)/2\)

5\(1/2\) NOOP
   6 bit vector

6. ITMV-3
   \(T(3) \leftarrow v(3) + v(0); \text{Skip if } v(3) > v(0)\)
   \(\text{Skip if } (A+B)/2 \neq B\)

7. JSF-5
   Skip forward 5 symbols

8. ITLE-4
   \(T(4) \leftarrow v(4) + v(0); \text{Skip if } v(4) > v(4)\)
   \(\text{Skip if } M[A+B/2] \leq D\)

9. JSB-8
   Skip backward 8 symbols
   \(A \leftarrow (A+B)/2\)
10 \text{ IPCL-3} \\ C(2) \leftarrow C(5) \\ B \leftarrow (A+B)/2 \\ \{ \text{End of main search loop} \}

11 \text{ JSB-9} \\ \{ \text{Skip backward 9-pycabli} \}

12 \text{ JMV(205)} \quad \text{Skip if } M[c(2)] = C(5) \quad \text{Skip if } M[EB] = D

14 \frac{1}{2} \quad \text{Finished } MEAJ = D

15 \frac{1}{2} \quad \text{failure}

16 \text{ JMV(105)} \quad \text{Skip if } M[c(1)] = C(5) \quad \text{Skip if } M[EA] = D

17 \frac{1}{2} \quad \text{Finished } MEAJ = D

18 \frac{1}{2} \quad \text{No answer}
Linked List Search

Assume the following, said word represents an address and data. The address points to the next word in list. The list ends if the word is all zeroes. The 18-bit address will be an bit 42-59 of the word, bits 0-41 of final one data.

Given a pointer to the beginning of the list, A, and an element, B which is 42 bits long and right justified in the 60-bit word, it is desired to find the address of the last element whose data part is equal to B.

Assume initial register configuration:

a) the register contains the byte code is 3
b) the register containing the pointer in word is 4

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>I</th>
<th>S</th>
<th>D</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Notice that the address of all must:

double the loop.
1.rrCV-1 c(5) ← V(1)
c(5) ← MEN
2.  SHTM 01000 00(5) 00(2) (align data on negative 2 free bits)
   [c(5)]_0-41 with [c(2)]_18-59
   SKIP IF ≠
4½  FINISHED - reg 1 contains address of cell
5½  rrp

6.  IKNU-5 c(5) ≠ 0
7.  End of last non-answer
8. rrCV-5 c(1) ← V(5)
c(1) ← MEN or element of list after A
9.  IKNU-5 c(1) ≠ 0
10. End of last non-answer
11. SHTM 01000 00(1) 00(2)
13½. Finishing 5th column, address of cell
14½. JTH-10 go back to parent routine, jump, check starting at last word.
LISP search in terms of car and cdr.

It is assumed that the 60-bit data word contains two address pointers in bits 12-29, and 42-59. A car instruction updates the address specified by bits 12-29, and a cdr instruction updates the address specified by bits 42-59. Given a string of cars and cdrs starting from address A, how would you implement this?

Implementation for any string of cars and cdr's forms an array, where a car is represented by an array element containing 29, and a cdr is represented by an array element containing 59. Let us suppose there are no cars and cdr's.

Assume the following register configuration:

- Byte page register is 3.
- Byte position register is 2.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Cen and Cen Branches program

1 STBY, 1100 60(1) 10(1) This instruction takes byte from address specified by register 6 to the contents of register 10. The byte is at 18, and byte position is obtained through the indirect field of 13 + register 4 (or depending on offset whether it's 16 or 18). Store that element in the string and can cost 6 or 8 bits position 0, 5 7.

3/5 ITGE-4 V(4) ← V(4) + V(6), skip if V(4) ≥ V(5)

4 1/2 I JWB-0 K ← K + 1, skip if K ≥ N

5 1/2 Finished - register 1 contains address of last element for Cen 8 clans chain.
\[ \text{Table 2: Correlation Coefficients} \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Note: The correlation coefficients are provided for each pair of variables:
- A and B: 0.9
- B and C: 0.8
- C and D: 0.7
- D and E: 0.6
- E and F: 0.5
- F and A: 0.4

The correlation coefficients indicate a strong positive relationship between the variables, with A and B showing the highest correlation of 0.9.
<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ISUB - 6</td>
<td>[v(5) \leftarrow v(6) - v(7)] [c(5) \leftarrow x - 1]</td>
</tr>
<tr>
<td>1</td>
<td>ITRV - 1</td>
<td>[t(v(3)) \leftarrow v(1)] [c(3) \leftarrow a[k+1]]</td>
</tr>
<tr>
<td>2</td>
<td>STCB != 0/1/2</td>
<td>00(3) 00(8) Compare byte in C(3) with U(8), and increment byte position counter</td>
</tr>
<tr>
<td>4 1/2</td>
<td>FINISHED COMPARE =</td>
<td></td>
</tr>
<tr>
<td>5 1/2</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>IKLIT - 5</td>
<td>Skip unit if byte position less than zero, need to fetch new chip word</td>
</tr>
<tr>
<td>7</td>
<td>IJSB - 5</td>
<td>Go back to instruction - 2, and check other fields in words</td>
</tr>
<tr>
<td>8</td>
<td>ITGE - 2</td>
<td>[v(5) \leftarrow v(2) + v(7)] [skip exit if v(0) &lt; 0] [k &lt; k + 1, check if k &lt; 0]</td>
</tr>
<tr>
<td>9</td>
<td>IJSB - 9</td>
<td>Fetch new data, and reset byte counter by jumping to instruction 0</td>
</tr>
<tr>
<td>10</td>
<td>FINISHED NO COMPARISON</td>
<td></td>
</tr>
</tbody>
</table>

Instructions 0 to 3 are fast executable in zero; need 2 1/2 words.
Simple subroutine jump and returns

Can just use a simple return jump instruction
ISB RC 0 (address of function), and parameters over general registers.

Simple Evaluation

(look at example in manual)

Multi-precision Arithmetic

(discussed facility of MPR registers which permits residues of arithmetic operations)

R Count

(not implemented)

Recursive Subroutine

(discussed Subroutine calls which push down addresses on stack and also have operations for pushing down the parameters of subroutine)

? (Am getting tired)
Interrupt and Store Character

The program is to store a character from a teletype, check whether the buffer is full, and if it is set a flag. As has been noted in the manual, an interrupt will cause a jump to the location pxy where y is the channel number and p is the program counter as the new set of registers.

Implementation

Let us consider 4 devices among

where

\[ A[K] \] = current word of buffer for the \( K \)th device,
\[ B[K] \] = current but position in the current word of
buffer for \( K \) device,
\[ C[K] \] = the address of the last word in buffer
for the \( K \)th device,
\[ D[K] \] = the overflow flag for device.

Consider that the new set of registers exchanged
in the interrupt processing here the following
format, where the \( BPC = 4 \), \( BSC = 2 \).
At location pty, we have the following code:

1 571PC  Y-00(T)-00(8)  So this instruction places the device number in register 7, and the input character in 8.

3½  (Interrupt caused by other than character transfer)

4½  I-JP.C-9  jump to character store routine

Character store routine

(Continued)
1. STBY 00:00.0 00(3); 00(7) transfer character to word
   A[K], where bit position B[K], also

3½. ITGT 10

4½. SRT 0

5½. NOP

6. ISUB 4

7. ITLE 4

8. ITRC 1

9. SRT 0

Jump back to main program

BL[K]=B[K]-51, reset byte position
   counter to beginning of word


Set area byte counter, D[K]=1