LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

An Interpreter for Instruction Cells

Computation Structures Group Memo 208
July 1981

Kenneth W. Todd

This research was supported by the Department of Energy under contract no. DE-AC02-79ER10473.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

An Interpreter for Instruction Cells

Kenneth Todd

30 June 1981

In my thesis [5], I explore the problem of translating programs written in the functional language VAL [1] into instruction cells for the Dennis-Misunas Form 1 data flow computer [2, 3]. To aid in the testing and debugging of the translation schemes that have been developed, an interpreter for these instruction cells has been written. This note describes both the format of the instruction cells being interpreted and the highlights of this interpreter.

The Instruction Cell

It has been preordained that the size of an instruction cell for the prototype data flow machine be a fixed 32 bytes in length. Adopting this convention, I have proposed a way of dividing up these 32 bytes, which has been modified by other members of the Computation Structures Group of the MIT Laboratory for Computer Science. The current version appears in Figure 1. Each cell contains an opcode, its operands, the numbers of those cells that are to receive the result of execution, and some accounting information.

Byte #0 has no functional purpose except to mark that the cell is not being used, in which case this byte is all zeroes.

Byte #3 holds the *acknowledgments needed value* and the *received bits* for the three operands. The acknowledgments needed value contains the number of *acknowledge signals* that the cell needs to receive from other cells before it can execute. With each reception of an acknowledge signal, the number held in this field is decremented by 1. When it reaches zero, all expected acknowledgments have been received by the cell and it is then ready to execute, pending the arrival of its operands. The received bits are used to mark whether or not the values for each of the operands have arrived. If an operand is either a constant or unused or has received its value from a cell that has previously executed, then the corresponding received bit is clear;

otherwise the bit is set. The RV bit is of significance only if the R3 bit is clear, being set if the third operand is true and clear if **false.**

Byte #2 contains the *true acknowledgment reset value* and the *constant bits,* followed by the *false acknowledgment reset value* and the constant bits again in byte #1. One of these two bytes is chosen to reset the contents of byte #3 after the cell executes. The process of determining which byte to choose will be explained later on in this memo. The reset values are used to reset the acknowledgments needed value after the cell executes. The constant bits mark whether or not the operands are constant. If a constant bit is clear then the corresponding operand is either a constant value or unused by the instruction specified by the opcode. The CV bit is of significance only if C3 is clear, in which case it is used to reset the RV field.

Bytes #4 and #5 together hold the *link to the next ready cell,* with byte #5 housing the most significant byte. This field is used to implement a queue of instruction cells that are ready to be executed.

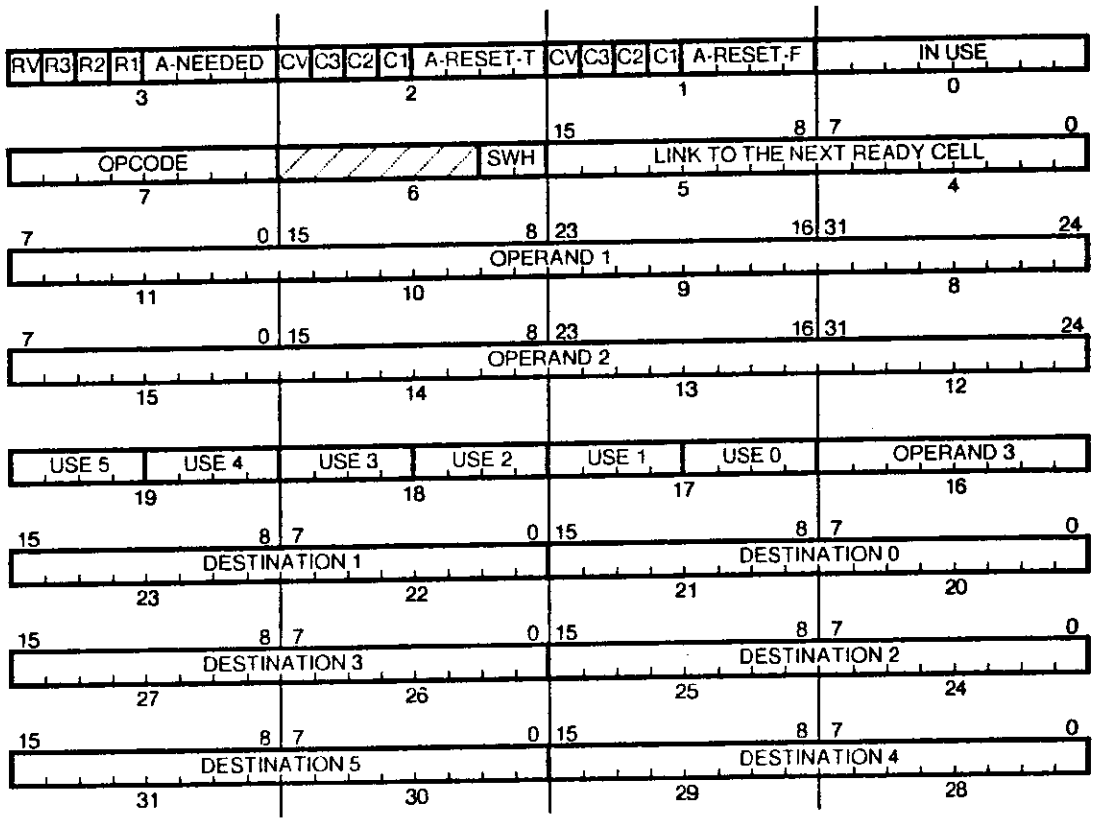

Figure 1. Format for an instruction cell for the Dennis-Misunas data flow machine.

One register in the processing element is designated to point to the first cell on this queue. The link field in this first cell gives the number of the second cell in this queue, whose link field points to the third cell, and so on. The end of the queue is marked by a second register, which holds the number of the last cell on the queue. When a program is being executed, the first cell on the queue executes, sending its result and acknowledgments to other cells. Each cell on the receiving end tests to see whether or not it has become ready to execute. If it has, then the link field of the cell pointed to by the end-of-queue register is set to this newly enabled cell, and the register in turn is also updated to point to this cell. When a cell has completed its execution, the number of this cell is compared with that held in the end-of-queue register. If they are the same then the queue is empty and the program terminates; otherwise the next cell in the queue (*i.e.*, the cell pointed to by the link field of the cell that just executed) is executed.

Byte #7 is reserved for the *opcode.* The use of eight bits makes it possible to accommodate an instruction set of $2^8 = 256$ distinct instructions, which is more than enough for our present purposes. Currently, the size of the instruction set accepted by the interpreter is only 58, which allows plenty of room for expansion. This set is summarized in Appendix 1.

Bytes #8 through #16 are used to contain the values of the *operands.* Operands #1 and #2 can be of any scalar type, either Boolean, integer, real, character, or null, whereas the third operand is of special use and restricted to be of type Boolean only. As suggested in Tucker's thesis [6], the VAL error values are encoded in the most significant byte of the value fields of the operands. However, I suggest a slight modification to his representation of error values, to allow for a simpler test for arithmetic error. My version is shown in Table 1. Since error values are such a basic and necessary part of VAL, it is assumed that the processing elements executing the instruction cells are specially designed so that arithmetic and logic involving error values can be handled and manipulated in the manner described in the VAL manual [1] efficiently.

Note that the cell retains no information concerning the types of operands #1 and #2. There is no need for this since VAL is a strongly-typed language and can perform all the type checking necessary at compile time to insure that the instruction cell program generated is type safe. The instruction cell itself

Table 1. Revised representations of error values.

76543210 (bits of most significant byte of operand)	
0xxxxxxx	good value, of type **real**, **integer**, **boolean**, **character**, or **null**
10001000	**unknown**
10101000	**pos_over**
11101000	**neg_over**
10011000	**pos_under**
11011000	**neg_under**
10001100	**zero_divide**
10000110	**miss_elt**
10000100	**undef**

bit 7 set if an error value

bit 6 set if negative

bit 5 set if overflow (when bit 7 is on)

bit 4 set if underflow (when bit 7 is on)

bit 3 set if arithmetic error (when bit 7 is on)

bit 2 set if **zero_divide**, **miss_elt**, or **undef**

---

determines the types of its operands through the opcode, if such a distinction is necessary.

The rest of the instruction cell is used for specifying which cells receive the result of execution and which ones receive an acknowledgment. Each *destination field* can specify the number of a cell that gets something, and its corresponding *use field* determines what that something is, if anything. Table 2 gives the different interpretations for the bit patterns of the use field. As can be observed, the third operand gives every instruction (except the merge and serializer, which use this operand for their own special purpose) the built-in capability of both T- and F-gates [3].

A problem of what to do emerges in the case where the third operand is neither **true** nor **false** but an error value instead. For this reason, the two-bit *switch handler* field in byte #6 has been added that acts as a three-way switch. One switch position informs the cell that an error value for the third operand is to be

Table 2. Interpretation of the use field in the instruction cell.

3210 (bits)	
xx00	the destination is unused
0001	if operand #3 = **false** then send an acknowledgment to the destination
0010	if operand #3 = **true** then send an acknowledgment to the destination
0011	send an acknowledgment to the destination unconditionally
0101	if operand #3 = **false** then send the result to operand #1 of the destination
0110	if operand #3 = **true** then send the result to operand #1 of the destination
0111	send the result to operand #1 of the destination unconditionally
1001	if operand #3 = **false** then send the result to operand #2 of the destination
1010	if operand #3 = **true** then send the result to operand #2 of the destination
1011	send the result to operand #2 of the destination unconditionally
1101	if operand #3 = **false** then send the result to operand #3 of the destination
1110	if operand #3 = **true** then send the result to operand #3 of the destination
1111	send the result to operand #3 of the destination unconditionally

bits 3 and 2 clear if acknowledgment

bit 3 clear and bit 2 set if result sent to operand #1

bit 3 set and bit 2 clear if result sent to operand #2

bits 3 and 2 set if result sent to operand #3

bit 1 set if used when operand #3 is **true**

bit 0 set if used when operand #3 is **false**

---

treated exactly as if it were the value **true** instead. By doing so, a more efficient translation for VAL's iterator that handles error cases correctly can be constructed. In a similar fashion, a second position is used to mark that the error value is to be treated as if it were the value **false**. The other switch position indicates that special action is to take place if the third operand is an error value. When in this position, only those results and acknowledgments specified to be sent out unconditionally (use field = "xx11") are actually sent out; all others are ignored.

Since each destination field is 16 bits wide, a maximum of 2^{16} = 65,536 instruction cells can be

addressed. Adding in the fact that each cell is 32 bytes in length, this places the largest size of memory that can be used at 32×2^{16} = two megabytes.

An instruction cell cannot execute until it is ready. For all instructions except the serializer and merge, having received all three operands and all acknowledgments (*i.e.*, R1, R2, and R3 must be clear and the acknowledgments needed value must be zero) constitutes the ready state for an instruction cell. This can be checked by simply testing the seven low bits of byte #3 for zero. The serializer needs all acknowledgments, a value for its the third operand, and a value for either its first or second operand to be ready. Determining if the merge instruction is ready to execute is not as simple a task as it is for the other cells. A necessary requirement is that the merge have received all acknowledge signals and its third operand. If the value of the third operand is **true** or if it is an error value and the switch handler indicates that it is to be handled as if it were **true**, then the value of the first operand is needed in addition to the above. If the third operand is **false** or if it is an error value and the switch handler says to treat such values as **false**, then it is the second operand that is needed along with the above. If the third operand is an error value and the switch handler does not specify how it is to be taken, then the above requirement proves to be sufficient as well as necessary.

The post-execution resetting process of an instruction cell consists of resetting the acknowledgments needed value and the received bits of those operands whose values were consumed during execution. If the value of operand #3 is **true** or if it is an error value treated as **true**, then the acknowledgments needed field is reset to the true acknowledgment reset value. For a merge cell, the received bits of the first and third operands are reset. For the serializer, the received bits of the third operand and the selected operand are reset. For all other cells, all operand received bits are reset and the entire resetting process is accomplished by copying the contents of byte #2 into byte #3. In a similar manner, if the value of operand #3 is **false** or if it is an error value treated as **false**, then the acknowledgments needed field is reset to the false acknowledgment reset value. For a merge cell, the received bits of the second and third operands are reset. For the serializer, the received bits of the third operand and the selected operand are reset. For all other cells, all operand received bits are reset and the entire resetting process is accomplished by copying the contents of byte #1

into byte #3.

If operand #3 is an error value that is neither treated as **true** or **false**, then a more complicated resetting scheme is used, the purpose of which is to produce translations that transparently and efficiently handle error values as the predicates of **if** constructs in VAL. Since the serializer does not permit error values for the third operand, it need not be covered. For the merge instruction, the true acknowledgment reset value is used to reset the acknowledgments needed value. The only received bit that is reset is that of the third operand. In addition, the value produced as a result of the execution of the merge is **undef[*]**. All the other instructions reset the received bits of all three operands but keep the acknowledgments needed value unchanged at zero.

Since operand #3 plays a significant part in both determining if the merge instruction is ready to execute and in resetting the instruction cell, it is highly advantageous to have a copy of its value located somewhere near bytes #1, #2, and #3 when performing either of these two tasks. This is the reason behind having the RV and CV bits present in these bytes.

As a final note, it is not my claim that this is the best format possible for an instruction cell. Considering byte #0 and part of byte #6, the four bytes reserved for the value of the second operand in single-operand instructions, and four and a half bytes for each unused destination, the amount of wasted storage can be staggering, exceeding 50% at times. Cells requiring a fanout greater than the six currently supplied must achieve this be using one of the six destinations to send the result to a distribute cell, thus introducing a time delay in addition to wasting space. It is expected that future versions of the instruction cell format will address and attempt to solve these problems by using a variable-sized format.

The Instruction Cell Interpreter

The interpreter of the type of instruction cells as outlined in the previous section was coded in the programming language CLU [4] on the MIT-XX DEC-20 computer system. It consists of a static relocating loader with an external function linker and an executor that runs programs in three different modes, which aids in the analysis and debugging of programs. Appendix 2 lists the current set of commands that this interpreter is capable of performing.

Table 3 shows the internal representation of an instruction cell by the interpreter. All of the fields can be manually set by the user with the exception of the link field, which the interpreter manages automatically.

The interpreter accepts as input files of one of two forms. The first is in a CLU encoded form, and a file whose suffix is ".valics" is recognized as being of this form (*e.g.*, "foo.valics"). The file itself contains five parts. The first is the highest cell number appearing in the file, which is one less than the number of cells in the file since cells are numbered starting with zero. Next comes the cell numbers of those cells that are to receive the input parameters to the instruction cell program held in the file, which is followed by the corresponding receiving operand numbers of those cells. After these come the cell numbers of those cells that produce the return values. Finally, the cells themselves appear. To load an encoded file, the "decode foo" command is given to the interpreter, where "foo" is the name of the ".valics" file.

The second form accepts files that contain an "assembly language" representation of the cells. Such a file is suffixed by ".ic" (*e.g.*, "foo.ic"). The first things looked for in the file are the links. They consist of a series of lines from among the following three:

```
$in <cell number>.<operand number>          ! for an input
$out <cell number>[condition]               ! for an output
$ext <first name of ".ic" file> <cell number>   ! for an external function call
```

The first one is an input link. When the file is being used as an external function, these links specify the order of the parameters to the function through the order in which they are listed, and also the cell and operands numbers of where to send the parameter values. When one or more occur in the main program (*i.e.*,

Table 3. Internal representation of an instruction cell by the interpreter.

```
cell = record[
    operation_tag: oneof[. . .],          ! corresponds to OPCODE
    ack_needed: int,                      ! corresponds to A_NEEDED
    ack_reset_t: int,                     ! corresponds to A_RESET_T
    ack_reset_f: int,                     ! corresponds to A_RESET_F
    switch_handler: oneof[                ! corresponds to SWH
        standard,
        error_as_t,
        error_as_f: null],
    link: int,                            ! corresponds to LINK TO THE NEXT READY CELL
    operand: array[record[
            constant: bool,               ! corresponds to C1, C2, C3
            received: bool,               ! corresponds to R1, R2, R3
            value: oneof[                 ! corresponds to OPERAND #1, OPERAND #2,
                val_null: val_null,       ! and OPERAND #3
                val_boolean: val_boolean,
                val_integer: val_integer,
                val_real: val_real,
                val_character: val_character]]],
    destination: array[oneof[             ! corresponds to USE 0 and DESTINATION 0, etc.
            unused: null,
            acknowledge_true,
            acknowledge_false,
            acknowledge_both,
            result_to_1_true,
            result_to_1_false,
            result_to_1_both,
            result_to_2_true,
            result_to_2_false,
            result_to_2_both,
            result_to_3_true,
            result_to_3_false,
            result_to_3_both,
            send_to_host_both,
            send_to_host_true,
            send_to_host_false: int]]]
```

---

the file name given in the load command), for each one listed, they serve mainly as documentation so that the user can know where to send the inputs to the function using the "send" command.

The second one is an output link. For each one found in the main program, the loader adds a "send_to_host_both" destination to the cell specified in the link if no condition is specified. If the "t"

condition is given then the added destination is "send_to_host_true"; likewise, "send_to_host_false" is added as a destination is the condition is "f". When present in an external function, these links inform the loader of the order and location of the values returned from the function.

The last one is an external function link, which specifies where in the program a function is to be called. The value of the i^{th} parameter is the result obtained by executing instruction cell *cell number* $- i$, whose acknowledgments needed value and both the true and false acknowledgment reset values are initially 0. At load time, these three values are set by the loader and a destination is added to connect the actual to the formal. Instruction cell *cell number* is an identity instruction cell with its single operand and one of its destinations set by the loader and its acknowledgments needed value and both the true and false acknowledgment reset values set to 1. When the function is to be called, an acknowledgment to cell *cell number* is sent, which eventually leads to the firing of the cells containing the values of the parameters of the function. The function then executes and when it finally terminates, the i^{th} returned value will have been stored in the first operand of instruction cell *cell number* $+ i$, ready for use by the calling program.

After the links come the specifications for each of the instruction cells. Each specification is headed by:

@ <cell number> <opcode>[switch] <ack_needed> <ack_reset_t> <ack_reset_f>

The set of opcodes is given in Appendix 1. To set the switch handler, the value of the switch is given as either "-T" or "-F", to indicate that the cell is to handle error values as if they were either **true** or **false** respectively. As for the acknowledgments needed and reset values, they are restricted as follows:

$$0 < \text{ack\_needed} < \text{a\_reset\_t} < 15$$
$$0 < \text{ack\_needed} < \text{a\_reset\_f} < 15$$

This is followed by the operand specifications, one per operand:

```
# 1 <value>        ! for a constant operand
# 0 1 <value>      ! for an operand that has received its value
# 0 0 [value]      ! for an operand that is waiting to receive its value
```

For single-operand instructions, only one specification is given. For instructions requiring two operands, two specifications are given with the first specification listed for the first operand and the second one for the

second operand. If the third operand is to be used, then the same convention is used as with the first two operands, but with the "#" replaced by a "&"; otherwise the third operand is set at the constant value **true**. The value part can be either a real, integer, Boolean, character, or null value, or it can be the word "base", which causes the loader to set the value of the operand to the cell frame (*i.e.*, the address) where the first cell of the file is loaded. If an operand is in the waiting to receive state, then the value part, if given, is ignored.

Finally, the destinations are specified by the following:

%<cell number>.<operand number>[condition] ! send the result or acknowledgment

If the operand number is zero then an acknowledgment is to be sent to the specified cell number; otherwise it is the number of the operand of the cell that is to receive the result of execution. If no condition is given, then the result or acknowledgment is to be sent unconditionally; otherwise, if the condition is "t" or "f" then the result or acknowledgment is sent out if the value of the third operand is **true** or **false** respectively. Although the current format of a cell limits the number of destinations that can be specified to six, the interpreter places no such restriction, thus allowing an arbitrary number of destinations to be specified per cell. There is no direct way to specify a "send to host" destination; instead it is done indirectly through the output links as was explained above.

Each cell's specifications may appear on a single line or be spread out over many. However, no more than one cell may be specified per line.

Comments can occur within the scope of the instruction cell file. Each one begins with an exclamation mark ("!") and runs the length of the line. They are permitted before or after any link or complete specification of a cell, but not in the midst of any one.

As an example, consider the problem of deriving the value of the following expression when given values for real a and b:

$$(a + b) * (a - b)$$

An instruction cell program that could perform this task might look like this:

```
! this ic program calculates (a+b) * (a-b)
$in 0.1          ! a
$in 1.1          ! b
$out 4           ! (a+b) * (a-b)
@0 dist 0 2 2    #0 0          %2.1 %3.1
@1 dist 0 2 2    #0 0          %2.2 %3.2
@2 add 0 1 1     #0 0 #0 0     %0.0 %1.0 %4.1
@3 sub 0 1 1     #0 0 #0 0     %0.0 %1.0 %4.2
@4 mult 0 0 0    #0 0 #0 0     %2.0 %3.0
```

The distribute instruction cells are needed since the input links can only specify one destination for the input value.

Assuming the above program is in a file called "foo.ic", a session with the interpreter to execute it might look as follows (the items underscored are typed in by the user):

```
* load foo

Input ports: 0.1 1.1

Output cells: 4

5 cells loaded from foo.ic

* send 0.1 3.0
* send 1.1 6.0
* srun
PASS 1: 0 1
PASS 2: 2 3
PASS 3: 4
OUTPUT 1: -27.0

TOTAL NUMBER OF PASSES IS 3
TOTAL NUMBER OF CELLS EXECUTED IS 5

*
```

The loading phase places the five cells of the program in the instruction cell memory and adds a "send_to_host_both" destination to cell number 4. The "send" command sends values to the input ports of the program, 3.0 to the port for the value of *a* and 6.0 to the port for the value of *b*. In addition, it discovers that cells 0 and 1 become enabled as a result of receiving values and thus appends these two cell numbers to the rear of the ready queue. The "srun" command repeats the process of first printing out the numbers of

those cells currently on the ready queue and then executing them (constituting a "pass") until there is no cell ready to execute. In this run, a is 3.0 and b is 6.0, which should (and does) produce a result of -27.0.

As a second example, if given the sine function, then the tangent function can be derived from it by using the following identity:

$$\tan(x) = \sin(x)/\sin(x+\pi/2)$$

Given a file "sin.ic" that performs as expected, the following instruction cell program calculates the tangent of its input value:

```
! function tan(x: real returns real)
! this function calculates the tangent of x
$in 0.1          ! x
$out 7           ! tan(x)
$ext sin 2
$ext sin 5
@0 id 0 2 2      #0 0                  %1.1 %2.0 %4.1 %5.0
@1 id 0 0 0      #0 0                  %0.0
@2 id 1 1 1      #1 0
@3 id 0 1 1      #0 0                  %7.1
@4 add 0 0 0     #0 0 #1 1.57079633    %0.0
@5 id 1 1 1      #1 0
@6 id 0 1 1      #0 0                  %7.2
@7 div 0 0 0     #0 0 #0 0             %3.0 %6.0
```

Of the two input file formats, encoded files are loaded in considerably less time. The interpreter can create ".ic" files from ".valics" files through the use of the "dump" command, and the "encode" command achieves translations in the other direction.

Appendix I - Instruction Cell Opcodes

Boolean Instruction Cell Opcodes

Opcode	*Operation Name*	*Type of Operands*	*Type of Result*	*Result Produced*
AND	Logical and	[1], [2]: **boolean**	**boolean**	[1] $\wedge$ [2]
OR	Logical or	[1], [2]: **boolean**	**boolean**	[1] $\vee$ [2]
NOT	Logical negation	[1]: **boolean** [2]: not used	**boolean**	$\neg$[1]
EQV BEQ	Logical equivalence/ Test for equality	[1], [2]: **boolean**	**boolean**	[1] $\equiv$ [2]
XOR BUEQ	Logical exclusive-or/ Test for inequality	[1], [2]: **boolean**	**boolean**	[1] $\oplus$ [2]
BFAN3	Fan to operand #3	[1]: **boolean** [2]: **integer**	**boolean**	[1] sent to operand #3 of cell whose address is in [2]

Character Instruction Cell Opcodes

Opcode	*Operation Name*	*Type of Operands*	*Type of Result*	*Result Produced*
CEQ	Test for equality	[1], [2]: **character**	**boolean**	[1] = [2]
CUEQ	Test for inequality	[1], [2]: **character**	**boolean**	[1] $\neq$ [2]
CTI	Conversion from character to integer	[1]: **character** [2]: not used	**integer**	**integer**([1])

Integer Instruction Cell Opcodes

Opcode	*Operation Name*	*Type of Operands*	*Type of Result*	*Result Produced*
IADD	Addition	[1], [2]: **integer**	**integer**	[1] + [2]
ISUB	Subtraction	[1], [2]: **integer**	**integer**	[1] − [2]
IMULT	Multiplication	[1], [2]: **integer**	**integer**	[1] * [2]
IDIV	Division	[1], [2]: **integer**	**integer**	[1] ÷ [2]
IMOD	Modulus	[1], [2]: **integer**	**integer**	[1] **mod** [2]
IABS	Absolute value	[1]: **integer** [2]: not used	**integer**	| [1] |
IMIN	Minimum	[1], [2]: **integer**	**integer**	**min**([1],[2])
IMAX	Maximum	[1], [2]: **integer**	**integer**	**max**([1],[2])
IEQ	Test for equality	[1], [2]: **integer**	**boolean**	[1] = [2]
IUEQ	Test for inequality	[1], [2]: **integer**	**boolean**	[1] ≠ [2]
ILT	Test for less than	[1], [2]: **integer**	**boolean**	[1] < [2]
ILE	Test for less than or equal to	[1], [2]: **integer** [1], [2]: **integer**	**boolean**	[1] ≤ [2]
IGT	Test for greater than	[1], [2]: **integer**	**boolean**	[1] > [2]
IGE	Test for greater than or equal to	[1], [2]: **integer**	**boolean**	[1] ≥ [2]
ITR	Conversion from integer to real	[1]: **integer** [2]: not used	**real**	**real**([1])
ITC	Conversion from integer to character	[1]: **integer** [2]: not used	**character**	**character**([1])

Real Instruction Cell Opcodes

Opcode	*Operation Name*	*Type of Operands*	*Type of Result*	*Result Produced*
ADD	Addition	[1], [2]: **real**	**real**	[1] + [2]
SUB	Subtraction	[1], [2]: **real**	**real**	[1] − [2]
MULT	Multiplication	[1], [2]: **real**	**real**	[1] * [2]
DIV	Division	[1], [2]: **real**	**real**	[1] ÷ [2]
ABS	Absolute value	[1]: **real** [2]: not used	**real**	| [1] |
MIN	Minimum	[1], [2]: **real**	**real**	**min**([1],[2])
MAX	Maximum	[1], [2]: **real**	**real**	**max**([1],[2])
EQ	Test for equality	[1], [2]: **real**	**boolean**	[1] = [2]
UEQ	Test for inequality	[1], [2]: **real**	**boolean**	[1] ≠ [2]
LT	Test for less than	[1], [2]: **real**	**boolean**	[1] < [2]
LE	Test for less than or equal to	[1], [2]: **real** [1], [2]: **real**	**boolean**	[1] ≤ [2]
GT	Test for greater than	[1], [2]: **real**	**boolean**	[1] > [2]
GE	Test for greater than or equal to	[1], [2]: **real**	**boolean**	[1] ≥ [2]
RTI	Conversion from real to integer	[1]: **real** [2]: not used	**integer**	**integer**([1])

Untyped Instruction Cell Opcodes

Opcode	*Operation Name*	*Type of Operands*	*Type of Result*	*Result Produced*
DIST ID	Distribute Identity	[1]: any [2]: not used	type of [1]	[1]
MERGE	Merge	[1]: any [2]: any	type of operand used	if [3] = **true** then [1] elseif [3] = **false** then [2] else **undef**[*]
SER	Serializer	[1]: any [2]: any	type of operand used	if [2] is not received or ([1] is received and [3] = **false**) then [1] else [2]; [3] : = if [1] is used then **true** else **false**
FAN1	Fan to operand #1	[1]: any [2]: **integer**	type of [1]	[1] sent to operand #1 of cell whose address is in [2]
FAN2	Fan to operand #2	[1]: any [2]: **integer**	type of [1]	[1] sent to operand #2 of cell whose address is in [2]
ACKFAN	Acknowledgment fan	[1]: **integer** [2]: **integer**	none	acknowledgment sent to cell whose address is in [1]
PO	Test for positive overflow	[1]: any [2]: not used	**boolean**	[1] = **pos_over**[*]
NO	Test for negative overflow	[1]: any [2]: not used	**boolean**	[1] = **neg_over**[*]
OVER	Test for overflow	[1]: any [2]: not used	**boolean**	[1] = **pos_over**[*] ∨ [1] = **neg_over**[*]
PU	Test for positive underflow	[1]: any [2]: not used	**boolean**	[1] = **pos_under**[*]
NU	Test for negative underflow	[1]: any [2]: not used	**boolean**	[1] = **neg_under**[*]
UNDER	Test for underflow	[1]: any [2]: not used	**boolean**	[1] = **pos_under**[*] ∨ [1] = **neg_under**[*]
UNKN	Test for unknown	[1]: any [2]: not used	**boolean**	[1] = **unknown**[*]

Opcode	*Operation Name*	*Type of Operands*	*Type of Result*	*Result Produced*
ZD	Test for division by zero	[1]: any [2]: not used	**boolean**	[1] = **zero_divide[*]**
AE	Test for arithmetic error	[1]: any [2]: not used	**boolean**	[1] = **pos_over[*]** ∨ [1] = **neg_over[*]** ∨ [1] = **pos_under[*]** ∨ [1] = **neg_under[*]** ∨ [1] = **unknown[*]** ∨ [1] = **zero_divide[*]**
ME	Test for missing element	[1]: any [2]: not used	**boolean**	[1] = **miss_elt[*]**
UNDEF	Test for undefined	[1]: any [2]: not used	**boolean**	[1] = **undef[*]**
ERR	Test for error	[1]: any [2]: not used	**boolean**	[1] = **pos_over[*]** ∨ [1] = **neg_over[*]** ∨ [1] = **pos_under[*]** ∨ [1] = **neg_under[*]** ∨ [1] = **unknown[*]** ∨ [1] = **zero_divide[*]** ∨ [1] = **miss_elt[*]** ∨ [1] = **undef[*]**
FTEST	Forall test	[1], [2]: **integer**	**integer**	if is error([1]) ∨ is error([2]) then **undef[*]** elseif [1] > [2] + 1 then **undef[*]** elseif [1] = [2] + 1 then **true** else **false**

Appendix II - Top-Level Commands to the Instruction Cell Interpreter

Note: Only enough of the command to distinguish it from all the others needs to be typed in, the portion of which is shown here in capital letters.

* ?

This command prints out a brief summary of all the possible commands to the instruction cell interpreter.

* BPC [list of cell numbers]

This command clears those break points given in the list of cell numbers. If the list is omitted, then all breakpoints are cleared.

* BPS [list of cell numbers]

This command sets those cells given in the list as breakpoints. If the list is omitted, then all cells are set as breakpoints.

* Clear

This command clears the instruction cell memory, all breakpoints, and the queue of ready instruction cells.

* DEcode <first name of '.valics' file>

This command takes the given ".valics" file and loads it into the first block of the instruction cell memory, appending all cells that are ready to execute to the rear of the queue of ready cells. The first name of the file is placed in the first entry of the memory map. Should some other file occupy the first block, the command is inhibited.

* DUmp <first name of loaded file>

The memory map is searched for the first occurrence of the specified file. If and when it is found, its cells are dumped into a ".ic" file.

* Encode

This command takes the instruction cell program loaded at base 0 (if there is one) and encodes it in a ".valics" file.

* EXit

This command exits the interpreter, permitting reentrance.

* Help

This is the same as the "?" command.

* Load <first name of '.ic' file>

This command takes the given ".ic" file and loads it into an available slot in the instruction cell memory, performing all necessary linking in the process and appending all cells that are ready to execute to the rear of the queue of ready cells. The first name of the file is placed in the corresponding memory map entry. The command is atomic, *i.e.*, should the load routine detect one or more errors, the state of the interpreter is left unchanged.

* Map

This command prints out the memory map.

* Queue

This command prints out the cell numbers of those cells in the ready queue.

* QUIT

This command exits the instruction cell interpreter.

* RElease <first name of loaded file>

The memory map is searched for the first occurrence of the specified file. If and when it is found, all cells that are on the ready queue cells and contained in the associated partition of the instruction cell memory are removed from the queue and all cells in that partition of memory are reinitialized and cleared as breakpoints. That partition is now ready to be reloaded with another instruction cell program.

* RUn

This is the command to execute in normal mode all instruction cells on the queue of ready instruction cells. As other instruction cells become ready, they are inserted at the end of the queue. Control is passed back to the top-level when the queue becomes empty. Should a runtime error occur, the pointer to the front of the queue is moved to the next ready instruction cell on the queue and control is passed back to the top-level.

* SEnd <cell number>,<operand number> [value]

This command is used to send values of type Boolean, integer, real, character, or null to operands and to send acknowledgments to cells. If the operand number is given as "0", then the value field is not used and an acknowledge signal is sent to the cell.

* SPy <cell number>

This is a command to pretty-print the contents of a cell.

* SRun

This command executes the instruction cells in the print-statistics mode. Its operation is like that of the run command, except that with each pass through the queue of ready cells, the queue is printed out.

* SS

This command executes the cells in the single-step mode. Only those cells present in the ready queue at the time the command is given are executed. As other cells become ready, they are inserted at the end of the queue but control is returned to the top level before commencement of their execution. Runtime errors are handled in the same fashion as in the run command.

References

[1] Ackerman, William B. and Dennis, Jack B. "VAL—A Value-Oriented Algorithmic Language Preliminary Manual." Technical Report 218, MIT Laboratory for Computer Science, Cambridge, MA, 13 June 1979.

[2] Dennis, Jack B., Leung, Clement K., and Misunas, David P. "A Highly Parallel Processor Using a Data Flow Machine Language." Computation Structures Group Memo 134-2, MIT Laboratory for Computer Science, Cambridge, MA, January 1977 (revised June 1980).

[3] Dennis, Jack B. and Misunas, David P. "A Preliminary Architecture for a Basic Data-Flow Processor." The Second Annual Symposium on Computer Architecture Conference Proceedings, January 1975, pp. 126-132. Also Computation Structures Group Memo 102, MIT Laboratory for Computer Science, Cambridge, MA, August 1974.

[4] Liskov, Barbara, *et.al.* "CLU Reference Manual." Technical Report 225, MIT Laboratory for Computer Science, MIT, Cambridge, MA, October 1979.

[5] Todd, Kenneth W. "High Level VAL Constructs in a Static Data Flow Machine." S.M. Thesis, MIT Laboratory for Computer Science, MIT, Cambridge, MA, 19 February 1981.

[6] Tucker, Richard. "Implementation of Arithmetic for the Data Flow Machine Processing Unit." S.B. Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1980. Also Computation Structures Group Memo 195, MIT Laboratory for Computer Science, Cambridge, MA, June 1980.