VCODE Reference Manual (Version 2.0)

Guy E. Blelloch   Siddhartha Chaterjee   Jay Sipelstein   Marco Zagha

May 15, 1994

1 Primitive types

There are six primitive data types in VCODE:

- Integer (*int*): signed integers.
- Boolean (*bool*): booleans (elements have value `T` or `F`).
- Floating point (*float*): double-precision floating point numbers.
- Character (*char*): characters.
- Segment descriptor (*segdes*): specifies a partitioning of one or more vectors into segments.

All VCODE objects are homogeneous vectors with elements of one of the above types, or are an object of type (*pair*) which is an untyped pair of VCODE objects. We have attempted to keep the number of VCODE primitives small. There are no scalar instructions; scalars are simply single-element vectors. Unsegmented vectors are represented as segmented vectors with but a single segment.

2 Instructions

VCODE instructions are classified into five types: elementwise operations, vector instructions, segment descriptor instructions, control instructions, and I/O instructions. For each instruction, we indicate the input and output types, any constraints that must be satisfied by its arguments, and the effect of the instruction. The appropriate response to an error condition resulting from not meeting any constraints is unspecified and left to the implementation.

VCODE is a stack based language. All VCODE instructions take and remove their operands from the stack, and return their results to the stack.

In VCODE, vector indices start at 0.

In the following, vectors of type *segdes* will be denoted by a vector of segment lengths delimited by square brackets. The actual machine representation of segment descriptors is implementation-dependent.

Instruction summaries are in the following form:

{*result types*} `op {type specifiers}` {*input types*}

If there are multiple possible result or input types, the | symbol will used the separate them. The op and type specifiers are part of the syntax of VCODE and appear in the program text. We then show before and after stack images in a sample execution of the operation.

2.1 Elementwise operations

Elementwise operations perform a (simple) operation on each element of a vector and return a vector of the results, in the same order as the argument. All operands of an elementwise operation must be of the same length.

1. {*int* | *float*} `+` {`INT`,`FLOAT`} {*int int* | *float float*}

   Returns the sum of the top two vectors on the stack.

(1 2 3 0 4) (3 4 0 2 3) ⋮	`+ INT`	(4 6 3 2 7) ⋮

2. {*int* | *float*} `-` {`INT`,`FLOAT`} {*int int* | *float float*}

   Returns the difference of the top two vectors on the stack.

(4 5 1 3 2) (7 6 8 9 5) ⋮	`- INT`	(3 1 7 6 3) ⋮

3. {*int* | *float*} `*` {`INT`,`FLOAT`} {*int int* | *float float*}

   Returns the product of the top two vectors on the stack.

(3 1 5 2 0) (0 1 4 3 2) ⋮	`* INT`	(0 1 20 6 0) ⋮

4. {*int* | *float*} `/` {`INT`,`FLOAT`} {*int int* | *float float*}

   Returns the quotient of the top two vectors on the stack. The divisor vector must not have any zero elements.

(2 4 3 8 1) (5 7 9 7 3) ⋮	`/ INT`	(2 1 3 0 3) ⋮

5. {*int*} `%` {*int int*}

   Returns the remainder of the top two vectors on the stack. The divisor vector must not have any zero elements.

(2 4 3 8 4) (5 7 9 7 -13) ⋮	`%`	(1 3 0 7 -1) ⋮

6. {*bool*} `<` {`INT`,`FLOAT`} {*int int* | *float float*}

   Returns the result of the "less than" test applied to the top two vectors on the stack.

(7 4 2 1 0) (3 4 5 2 1) ⋮	`< INT`	(T F F F F) ⋮

7. {*bool*} `<=` {`INT`,`FLOAT`} {*int int* | *float float*}

   Returns the result of the "less than or equal to" test applied to the top two vectors on the stack.

(7 4 2 1 0) (3 4 5 2 1) ⋮	`<= INT`	(T T F F F) ⋮

8. {*bool*} `>` {`INT`,`FLOAT`} {*int int* | *float float*}

Returns the result of the “greater than” test applied to the top two vectors on the stack.

(7 4 2 1 0) (3 4 5 2 1) ⋮	`> INT`	(F F T T T) ⋮

9. $\{bool\}$ `>=` $\{$`INT,FLOAT`$\}$ $\{int\ int \mid float\ float\}$

Returns the result of the “greater than or equal to” test applied to the top two vectors on the stack.

(7 4 2 1 0) (3 4 5 2 1) ⋮	`>= INT`	(F T T T T) ⋮

10. $\{bool\}$ `=` $\{$`INT,FLOAT`$\}$ $\{int\ int \mid float\ float\}$

Returns the result of the equality test applied to the top two vectors on the stack.

(3 4 5 2 1) (7 4 2 1 0) ⋮	`= INT`	(F T F F F) ⋮

11. $\{bool\}$ `!=` $\{$`INT,FLOAT`$\}$ $\{int\ int \mid float\ float\}$

Returns the result of the inequality test applied to the top two vectors on the stack.

(3 4 5 2 1) (7 4 2 1 0) ⋮	`!= INT`	(T F T T T) ⋮

12. $\{int\}$ `LSHIFT` $\{int\ int\}$

Returns the result of shifting each element of the data vector left by the amounts in the shift vector. The elements of the data vector must be non-negative. The elements of the shift vector must be non-negative and not too large (typically < 32 or 64, depending on the implementation). The vacated bits are zero-filled.

(1 1 2 2 0) (6 8 3 7 8) ⋮	`LSHIFT`	(12 16 12 28 8) ⋮

13. $\{int\}$ `RSHIFT` $\{int\ int\}$

Returns the result of shifting each element of the data vector right by the amounts in the shift vector. The elements of the data vector should non-negative. The elements of the shift vector must be non-negative and not too large (typically < 32, or 64, depending on the implementation). The vacated bits are filled either with the sign bit or with zero, depending on the implementation (shifting negative values is strongly discouraged).

(1 1 2 2 0) (6 8 3 7 8) ⋮	`RSHIFT`	(3 4 0 1 8) ⋮

14. $\{bool \mid int\}$ `NOT` $\{$`BOOL, INT`$\}$ $\{bool \mid int\}$

Returns the negation (boolean for `bool`s, bitwise for `int`s) of the vector.

(T F F T T) ⋮	`NOT BOOL`	(F T T F F) ⋮

15. {*bool* | *int*} `AND` {`BOOL,INT`} {*bool bool* | *int int*}

Returns the result of applying a bitwise or boolean AND to the top two vectors on the stack.

(T T F F) (T F T F) ⋮	`AND BOOL`	(T F F F) ⋮

16. {*bool* | *int*} `OR` {`BOOL,INT`} {*bool bool* | *int int*}

Returns the result of applying a bitwise or boolean OR to the top two vectors on the stack.

(1 2 3 4) (2 3 4 5) ⋮	`OR INT`	(2 3 7 5) ⋮

17. {*bool* | *int*} `XOR` {`BOOL,INT`} {*bool bool* | *int int*}

Returns the result of applying a bitwise or boolean exclusive OR to the top two vectors on the stack.

(T T F F) (T F T F) ⋮	`XOR BOOL`	(F T T F) ⋮

18. {*int* | *bool* | *float*} `SELECT` {`INT,BOOL,FLOAT`} {*bool int int* | *bool bool bool* | *bool float float*}

Returns the result of selecting from the two data vectors based on the value of the selection vector: a true value means pick from the first argument; a false value means select from the second argument.

(4 1 9 7) (2 0 1 6) (T F F T) ⋮	`SELECT INT`	(2 1 9 6) ⋮

19. {*int*} `RAND` {*int*}

Returns an integer vector of random numbers based on the ranges specified in the data vector. If the range specification is n, the number will be in the range 0–$(n-1)$. The elements of the data vector must be positive.

(2 5 8 8) ⋮	`RAND`	(1 4 3 7) ⋮

20. {*int*} `FLOOR` {*float*}

Returns the vector values truncated towards negative infinity. Note that if *float* is represented with more bits than *int*, this conversion (and those following) may lose significant digits.

(2.6 2.5 0.3 -0.7) ⋮	`FLOOR`	(2 2 0 -1) ⋮

21. {*int*} `CEIL` {*float*}

Returns the vector values truncated towards positive infinity.

(2.6 2.5 0.3 -0.7) ⋮	`CEIL`	(3 3 1 0) ⋮

22. {*int*} `TRUNC` {*float*}

Returns the vector values truncated towards zero.

(2.6 2.5 0.3 -0.7) ⋮	`TRUNC`	(2 2 0 0) ⋮

23. {*int*} `ROUND` {*float*}

Returns the vector values rounded to the nearest integer. If the fractional part is 0.5, the number is rounded to the nearest even integer.

(2.6 2.5 0.3 -0.7) ⋮	`ROUND`	(3 2 0 -1) ⋮

24. {*float*} `I_TO_F` {*int*}

Converts the *int* vector into a *float* vector.

(1 2 3 -1) ⋮	`I_TO_F`	(1.0 2.0 3.0 -1.0) ⋮

25. {*bool*} `I_TO_B` {*int*}

Converts the *int* vector into a *bool* vector. Zero is mapped to `F`; any other value is mapped to `T`.

(1 0 3 -1) ⋮	`I_TO_B`	(T F T T) ⋮

26. {*int*} `B_TO_I` {*bool*}

Converts the *bool* vector into a *int* vector. `T` is mapped to 1, and `F` to 0.

(F T) ⋮	`B_TO_I`	(0 1) ⋮

27. {*float*} `LOG` {*float*}

Returns the natural logarithm of the vector. All elements must be positive.

(1.0 3.2 4) ⋮	`LOG`	(0.0 1.163 1.386) ⋮

28. {*float*} `SQRT` {*float*}

Returns the square root of the vector. All elements must be non-negative.

(2.0 4.0 6.0 9.0) ⋮	`SQRT`	(1.414 2.0 2.449 3.0) ⋮

29. {*float*} `EXP` {*float*}

Returns the exponential function of the vector.

(1.0 2.0 -1.0) ⋮	`EXP`	(2.718 7.389 0.367) ⋮

30. {*float*} `SIN` {*float*}

Returns the sine of the vector.

(0.0 3.1 -1.7 ⋮	`SIN`	(0.000 0.002 -0.992) ⋮

31. {*float*} `COS` {*float*}

Returns the cosine of the vector.

(0.0 3.1 -1.7 ⋮	`COS`	(1.000 -0.999 -0.129) ⋮

32. {*float*} `TAN` {*float*}

Returns the tangent of the vector.

(0.0 3.1 -1.7 ⋮	`TAN`	(0.000 -0.002 7.697) ⋮

33. {*float*} `ASIN` {*float*}

Returns the inverse sine of the vector.

(0.0 1.0 -1.0 ⋮	`ASIN`	(0.000 1.571 -1.571) ⋮

34. {*float*} `ACOS` {*float*}

Returns the inverse cosine of the vector.

(0.0 1.0 -1.0 ⋮	`ACOS`	(1.571 0.000 3.142) ⋮

35. {*float*} `ATAN` {*float*}

Returns the inverse tangent of the vector.

(0.0 1.0 -1.0 ⋮	`ATAN`	(0.000 1.000 -1.000) ⋮

36. {*float*} `SINH` {*float*}

Returns the hyperbolic sine of the vector.

(0.0 1.0 -1.0 ⋮	`SINH`	(0.000 1.172 -1.172) ⋮

37. {*float*} `COSH` {*float*}

Returns the hyperbolic cosine of the vector.

(0.0 1.0 -1.0 ⋮	`COSH`	(1.000 1.543 1.543) ⋮

38. {*float*} `TANH` {*float*}

Returns the hyperbolic tangent of the vector.

(0.0 1.0 -1.0 ⋮	`TANH`	(0.000 1.000 -1.000) ⋮

2.2 Vector instructions

A vector V of type *int*, *bool* or *float* and a vector S of type *segdes* are said to be **compatible** iff the length of V is equal to the sum of the lengths of the segments described by S. We will use this notion of compatibility to define constraints for this class of instructions.

1. {*int* | *float*} `+_SCAN {INT,FLOAT}` {*int segdes* | *float segdes*}

   Returns the segmented plus-scan of the vector. The vector and the segment descriptor must be compatible. The identity for the operation is 0.

[3 3] (1 3 2 3 5 1) ⋮	`+_SCAN INT`	(0 1 4 0 3 8) ⋮

2. {*int* | *float*} `*_SCAN {INT,FLOAT}` {*int segdes* | *float segdes*}

   Returns the segmented multiply-scan of the vector. The vector and the segment descriptor must be compatible. The identity for the operation is 1.

[3 3] (1 3 2 3 5 1) ⋮	`*_SCAN INT`	(1 1 3 1 3 15) ⋮

3. {*int* | *float*} `MAX_SCAN {INT,FLOAT}` {*int segdes* | *float segdes*}

   Returns the segmented max-scan of the vector. The vector and the segment descriptor must be compatible. The identity for the operation is $-\infty$ (the minimum value of the data type for a given implementation).

[3 3] (1 3 2 3 5 1) ⋮	`MAX_SCAN INT`	($-\infty$ 1 3 $-\infty$ 3 5) ⋮

4. {*int* | *float*} `MIN_SCAN {INT,FLOAT}` {*int segdes* | *float segdes*}

   Returns the segmented min-scan of the vector. The vector and the segment descriptor must be compatible. The identity for the operation is ∞ (the maximum value of the data type for a given implementation).

[3 3] (1 3 2 3 5 1) ⋮	`MIN_SCAN INT`	(∞ 1 1 ∞ 3 3) ⋮

5. {*bool* | *int*} `AND_SCAN {INT, BOOL}` {*bool segdes* | *int segdes*}

   Returns the segmented AND-scan of the vector. The vector and the segment descriptor must be compatible. The identity element for the operation is T.

[3 3] (T F T F T T) ⋮	`AND_SCAN BOOL`	(T T F T F F) ⋮

6. $\{$*bool* | *int*$\}$ `OR_SCAN` $\{$`INT, BOOL`$\}$ $\{$*bool segdes* | *int segdes*$\}$

Returns the segmented OR-scan of the vector. The vector and the segment descriptor must be compatible. The identity for the operation is F.

Stack before	Operation	Stack after
[3 3] (T F T F T T) ⋮	`OR_SCAN BOOL`	(F T T F F T) ⋮

7. $\{$*bool* | *int*$\}$ `XOR_SCAN` $\{$`INT, BOOL`$\}$ $\{$*bool segdes* | *int segdes*$\}$

Returns the segmented XOR-scan of the vector. The vector and the segment descriptor must be compatible. The identity for the operation is F.

Stack before	Operation	Stack after
[3 3] (T F T F T T) ⋮	`XOR_SCAN BOOL`	(F T T F F T) ⋮

8. $\{$*int* | *float*$\}$ `+_REDUCE` $\{$`INT,FLOAT`$\}$ $\{$*int segdes* | *float segdes*$\}$

Returns the segmented plus-reduction of the vector. The vector and the segment descriptor must be compatible. The identity for the operation is 0.

Stack before	Operation	Stack after
[3 3] (1 3 2 3 5 1) ⋮	`+_REDUCE INT`	(6 9) ⋮

9. $\{$*int* | *float*$\}$ `*_REDUCE` $\{$`INT,FLOAT`$\}$ $\{$*int segdes* | *float segdes*$\}$

Returns the segmented multiply-reduction of the vector. The vector and the segment descriptor must be compatible. The identity for the operation is 1.

Stack before	Operation	Stack after
[3 3] (1 3 2 3 5 1) ⋮	`*_REDUCE INT`	(6 15) ⋮

10. $\{$*int* | *float*$\}$ `MAX_REDUCE` $\{$`INT,FLOAT`$\}$ $\{$*int segdes* | *float segdes*$\}$

Returns the segmented max-reduction of the vector. The vector and the segment descriptor must be compatible. The identity for the operation is $-\infty$ (the minimum value of the data type for a given implementation).

Stack before	Operation	Stack after
[3 3] (1 3 2 3 5 1) ⋮	`MAX_REDUCE INT`	(3 5) ⋮

11. $\{$*int* | *float*$\}$ `MIN_REDUCE` $\{$`INT,FLOAT`$\}$ $\{$*int segdes* | *float segdes*$\}$

Returns the segmented min-reduction of the vector. The vector and the segment descriptor must be compatible. The identity for the operation is ∞ (the maximum value of the data type for a given implementation).

Stack before	Operation	Stack after
[3 3] (1 3 2 3 5 1) ⋮	`MIN_REDUCE INT`	(1 1) ⋮

12. $\{$*bool* | *int*$\}$ `AND_REDUCE` $\{$`INT, BOOL`$\}$ $\{$*bool segdes* | *int segdes*$\}$

Returns the segmented AND-reduction of the vector. The vector and the segment descriptor must be compatible. The identity element for the operation is T.

[3 3] (T F T F T T) ⋮	`AND_REDUCE BOOL`	(F F) ⋮

13. {*bool* | *int*} `OR_REDUCE` {`INT, BOOL`} {*bool segdes* | *int segdes*}

Returns the segmented OR-reduction of the vector. The vector and the segment descriptor must be compatible. The identity for the operation is F.

[3 3] (T F T F T T) ⋮	`OR_REDUCE BOOL`	(T T) ⋮

14. {*bool* | *int*} `XOR_REDUCE` {`INT, BOOL`} {*bool segdes* | *int segdes*}

Returns the segmented XOR-reduction of the vector. The vector and the segment descriptor must be compatible. The identity for the operation is F.

[3 3] (T F T F T T) ⋮	`XOR_REDUCE BOOL`	(F F) ⋮

15. {*int* | *bool* | *float*} `PERMUTE` {`INT,BOOL,FLOAT`} {*int int segdes*
| *bool int segdes*
| *float int segdes*}

Segmented simple permute operation. Calling the arguments **(data index segdes)**, the following constraints must be satisfied:

- **data** and **segdes** must be compatible.
- **index** and **segdes** must be compatible.
- The elements of **index** must form a permutation within each segment of **segdes**.

The elements of **data** are permuted to the locations specified by **index**, within each segment. The result vector is compatible with **segdes**.

[4 2] (3 1 0 2 1 0) (0 1 2 3 4 5) ⋮	`PERMUTE INT`	(2 1 3 0 5 4) ⋮

16. {*int* | *bool* | *float*} `DPERMUTE` {`INT,BOOL,FLOAT`} {*int int int segdes segdes*
| *bool int bool segdes segdes*
| *float int float segdes segdes*}

Segmented permute operation with default. Calling the arguments **(data index default src-seg dst-seg)**, the following constraints must be satisfied:

- **data** and **src-seg** must be compatible.
- **index** and **src-seg** must be compatible.
- **default** and **dst-seg** must be compatible.
- The elements of **index** must be distinct and properly bounded within each segment of **dst**.

- **src-seg** and **dst-seg** must have the same number of segments, and each element of **dst-seg** must be greater than or equal to the corresponding element of **src-seg**.

The elements of **data** are permuted to the positions indicated by the elements of **index**. Any position that does not receive a value from **data** is filled with the value at the corresponding position of **default**. The result has the same length as **default** and is compatible with **dst-seg**.

[5 3] [4 2] (5 5 5 5 5 6 6 6) (3 4 0 2 2 1) (0 1 2 3 4 5) ⋮	`DPERMUTE INT`	(2 5 3 0 1 6 5 4) ⋮

17. {*int* | *bool* | *float*} `FPERMUTE` {`INT,BOOL,FLOAT`} {*int int bool segdes segdes* | *bool int bool segdes segdes* | *float int bool segdes segdes*}

Segmented permute operation with flags. Calling the arguments **(data index flags src-seg dst-seg)**, the following constraints must be satisfied:

- **data** and **src-seg** must be compatible.
- **index** and **src-seg** must be compatible.
- **flags** and **src-seg** must be compatible.
- The elements of **index** not masked out by **flags** must be distinct and properly bounded within the segments defined in **src-seg**.
- **src-seg** and **dst-seg** must have the same number of segments.

The elements of **data** not masked out by **flags** are permuted to the positions indicated by the corresponding elements of **index**. The contents of any position in **src** not so indicated are undefined. The result is compatible with **dst-seg**.

[3 1] [4 2] (T F T T T F) (2 1 0 1 0 0) (0 1 2 3 4 5) ⋮	`FPERMUTE INT`	(2 3 0 4) ⋮

18. {*int* | *bool* | *float*} `BPERMUTE` {`INT,BOOL,FLOAT`} {*int int segdes segdes* | *bool int segdes segdes* | *float int segdes segdes*}

Segmented back-permute instruction. Calling the arguments **(data index src-seg dst-seg)**, the following constraints must be satisfied:

- **data** and **src-seg** must be compatible.
- **index** and **dst-seg** must be compatible.
- The elements of **index** must be properly bounded within the segments defined in **src-seg**.
- **src-seg** and **dst-seg** must have the same number of segments.

The elements of **data** are copied to the position in the result according to the corresponding element of **index**. This is the inverse operation of `PERMUTE` when the elements of **index** in a segment form a proper permutation of the allowed indices of the corresponding segment of **dst-seg**.

Stack before	Instruction	Stack after
[4 2] [3 3] (2 0 1 1 0 2) (1 2 3 4 5 6) ⋮	`BPERMUTE INT`	(3 1 2 2 4 6) ⋮

19. $\{$*int* $|$ *bool* $|$ *float*$\}$ `BFPERMUTE` $\{$`INT,BOOL,FLOAT`$\}$ $\{$*int int bool segdes segdes* $|$ *bool int bool segdes segdes* $|$ *float int bool segdes segdes*$\}$

Segmented back-permute instruction with flags. The unfilled positions in the vector are set to 0 for `int`s and `float`s, and F for `bool`s. Calling the arguments **(data index flag src-seg dst-seg)**, the following constraints must be satisfied:

- **data** and **src-seg** must be compatible.
- **index** and **flag** must have the same length.
- **index** and **dst-seg** must be compatible.
- The elements of **index** not masked out by **flag** must be properly bounded within the segments defined in **src-seg**.
- **src-seg** and **dst-seg** must have the same number of segments.

Stack before	Instruction	Stack after
[4 3] [3 3] (F T T F T T T) (1 0 1 9 1 0 2) (1 2 3 4 5 6) ⋮	`BFPERMUTE INT`	(0 1 2 0 5 4 6) ⋮

20. $\{$*int* $|$ *bool* $|$ *float*$\}$ `DFPERMUTE` $\{$`INT,BOOL,FLOAT`$\}$ $\{$*int int bool int segdes segdes* $|$ *bool int bool bool segdes segdes* $|$ *float int bool float segdes segdes*$\}$

Segmented default permute operation with flags. Calling the arguments **(data index flag default src-seg dst-seg)**, the following constraints must be satisfied:

- **data** and **src-seg** must be compatible.
- **index** and **src-seg** must be compatible.
- **flags** and **src-seg** must be compatible.
- **default** and **dst-seg** must be compatible.
- The elements of **index** not masked out by **flag** must be properly bounded within the segments defined in **src-seg**.
- **src-seg** and **dst-seg** must have the same number of segments.

Stack before	Instruction	Stack after
[5 3] [4 2] (5 5 5 5 5 6 6 6) (T F T F T F) (3 4 0 2 2 1) (0 1 2 3 4 5) ⋮	`DFPERMUTE INT`	(2 5 5 0 5 6 6 4) ⋮

21. $\{int \mid bool \mid float\}$ `EXTRACT {INT,BOOL,FLOAT}` $\{int\ int\ segdes \mid bool\ int\ segdes \mid float\ int\ segdes\}$

Segmented extract operation. Calling the arguments **src index segdes**, the following constraints must be satisfied:

- **src** and **segdes** must be compatible.
- **index** must have as many elements as **segdes** has segments.
- The elements of **index** must be non-negative and bounded above by the segment lengths (this means that no segment of **src** may be of length 0).

The elements specified by **index** are extracted from **src**, one per segment, and returned. The result has the same size as **index**.

Stack before	Operation	Stack after
[4 2] (2 1) (0 1 2 3 4 5) ⋮	`EXTRACT INT`	(2 5) ⋮

22. $\{int \mid bool \mid float\}$ `REPLACE {INT,BOOL,FLOAT}` $\{int\ int\ int\ segdes \mid bool\ int\ bool\ segdes \mid float\ int\ float\ segdes\}$

Segmented replace operation. Calling the arguments **src index values segdes**, the following constraints must be satisfied:

- **src** and **segdes** must be compatible.
- **index** and **values** must have the same length.
- **index** must have as many elements as **segdes** has segments.
- The elements of **index** must be non-negative and bounded above by the segment lengths (this means that no segment of **src** may be of length 0).

The result is a vector that has the values of **src** in all locations except those specified in **index** (one per segment), where it has the values of **values**. The result is compatible with **segdes**.

Stack before	Operation	Stack after
[4 2] (3 -1) (2 1) (0 1 2 3 4 5) ⋮	`REPLACE INT`	(0 1 3 3 4 -1) ⋮

23. $\{int\ segdes \mid bool\ segdes \mid float\ segdes\}$ `PACK {INT,BOOL,FLOAT}` $\{int\ bool\ segdes \mid bool\ bool\ segdes \mid float\ bool\ segdes\}$

Segmented pack operation. Calling the arguments **data flags segdes**, the following constraints must be satisfied:

- **data** and **flags** must have the same length.
- **data** and **segdes** must be compatible.

The result is a vector and a segment descriptor for that vector. The values in the vector are those from **data** corresponding to true elements in **flags**.

Stack before	Operation	Stack after
[4 2] (T F T F T F) (0 1 2 3 4 5) ⋮	`PACK INT`	[2 1] (0 2 4) ⋮

24. {*int*} `RANK_UP` {`INT, FLOAT`} {*int segdes* | *float segdes*}

Segemented upwards rank. The second argument must be compatible with the first. The result is a segmented vector of indices giving the increasing order of the data. The rank is stable.

[4 2] (3 2 3 8 5 3) ⋮	`RANK_UP INT`	(1 0 2 3 1 0) ⋮

25. {*int*} `RANK_DOWN` {`INT, FLOAT`} {*int segdes* | *float segdes*}

Segemented downwards rank. The second argument must be compatible with the first. The result is a segmented vector of indices giving the decreasing order of the data. The rank is stable.

[4 2] (3 2 3 8 5 3) ⋮	`RANK_DOWN INT`	(1 3 2 0 0 1) ⋮

26. {*int* | *bool* | *float*} `DIST` {`INT,BOOL,FLOAT`} {*int segdes* | *bool segdes* | *float segdes*}

Segmented distribute. The second argument is a segment descriptor describing the destination vector. The elements of the first vector are replicated across segments to create a result compatable with the segment descriptor.

[3 2] (2 4) ⋮	`DIST INT`	(2 2 2 4 4) ⋮

27. {*int*} `INDEX` {*int int segdes*}

Segmented index. Arguments are unsegmented vectors specifying initial values, strides, and a segment descriptor for the result. Both the initial value and the stride can be any integer. Index can be thought of as an extention of the distribute operation.

[3 2] (1 4) (0 0) ⋮	`INDEX`	(0 1 2 0 4) ⋮

28. {*int*} `LENGTH` {`INT,BOOL,FLOAT`} {*int* | *bool* | *float*}

Returns the length of an unsegmented vector.

(4 3 2 1 0 7 6) ⋮	`LENGTH INT`	(7) ⋮

2.3 Segment descriptor instructions

1. {*segdes*} `MAKE_SEGDES` {*int*}

Returns a segment descriptor given the lengths of the segments. All the elements of the input must be non-negative.

(3 2) ⋮	`MAKE_SEGDES`	[3 2] ⋮

2. {*int*} `LENGTHS` {*segdes*}

   Returns the lengths of the segments given the segment descriptor.

[3 3]	`LENGTHS`	(3 3)
⋮		⋮

2.4 Control instructions

The control operatations manipulate the VCODE stack or control program flow of execution. Vectors on the VCODE stack are numbered, with the 0th vector being on top. The stack instructions are the only ones that may operate on elements of type *pair*.

1. {*vector+*} `COPY I J` {*null*}

   Copies `I` elements starting from position `J` within the stack to the top of the stack. The stack must be at least `I+J` deep. `I` and `J` must be non-negative.

		(2 3 4)
		(4 5 6)
(1 2 3)	`COPY 2 1`	(1 2 3)
(2 3 4)		(2 3 4)
(4 5 6)		(4 5 6)
⋮		⋮

2. {*null*} `POP I J` {*null*}

   Deletes `I` elements starting from position `J` within the stack and compresses the stack. The familiar stack pop operation is the special case `POP 1 0`. The stack must be at least `I+J` deep. `I` and `J` must be non-negative.

(1 2 3)		
(2 3 4)		
(3 4 5)	`POP 2 1`	(1 2 3)
(4 5 6)		(4 5 6)
⋮		⋮

3. {*null*} `CPOP I J` {*null*}

   This is a combined COPY and POP instruction. `I` elements, starting at position `J` on the stack are moved to the top of the stack. The stack must be at least `I+J` deep. `I` and `J` must be non-negative.

(1 2 3)		(2 3 4)
(2 3 4)		(3 4 5)
(3 4 5)	`CPOP 2 1`	(1 2 3)
(4 5 6)		(4 5 6)
⋮		⋮

4. {*pair*} `PAIR` {*value value*}

   Creates a *pair* out of the two values on top of the stack.

(1 2 3)		
(2 3 4)	`PAIR`	(pair)
(3 4 5)		(3 4 5)
⋮		⋮

5. {*value value*} `UNPAIR` {*pair*}

   Restores the contents of a pair to the stack. PAIR and UNPAIR are inverse operations.

		(1 2 3)
(pair)	`UNPAIR`	(2 3 4)
(3 4 5)		(3 4 5)
⋮		⋮

6. {*null*} `FUNC LABEL` {*null*}

   Defines a function with the name `LABEL`.

7. {*null*} `CALL LABEL` {*null*}

   Calls the function named `LABEL`.

8. {*null*} `RET` {*null*}

   Returns from function call.

9. {*null*} `IF` {*bool*}

   Beginning of a conditional statement. A boolean vector `test` of length 1 must be on top of the stack. If `test` has value `T`, the statements following the `IF` are executed until the matching `ELSE` statement is reached, and then control goes to the statement following the matching `ENDIF`. If `test` has value `F`, control branches to the statement following the matching `ELSE`. Both branches must leave the stack in the same state with respect to depth and data types. Nested conditionals are matched to the closest enclosing scope.

10. {*null*} `ELSE` {*null*}

    See description of `IF`.

11. {*null*} `ENDIF` {*null*}

    See description of `IF`.

12. {*int* | *bool* | *float* | *char* | *segdes*} `CONST` {`INT,BOOL,FLOAT,CHAR,SEGDES`} `VAL` {*null*}

    Puts the constant vector `VAL` on top of the stack.

		(3 4 5)
(2 3 4)	`CONST INT (3 4 5)`	(2 3 4)
(T F)		(T F)
⋮		⋮

    The special syntax `CONST CHAR "string"` puts a string constant on the stack.

2.5 I/O instructions

For I/O, vectors are syntactically of the form `(val1 val2 ... valn)`. Segment descriptors are of the form `[val1 val2 ... valn]` where the values are the lengths representation of the descriptor.

1. {*bool char int* | *bool char bool* | *bool char float* | *bool char segdes*} `READ` {`INT,BOOL,FLOAT,CHAR,SEGDES`} {*null*}

   Reads a vector of the appropriate type from the standard input stream. A `READ` of a segment descriptor reads in the lengths of the segments and internally converts it to the appropriate internal representation for segment descriptors. There are three return values: a boolean status flag (`T` if the `READ` was successful, `F` otherwise), an character vector holding an error message (empty if no error), and finally the result of the read.

(2 3 4) ⋮	`READ INT`	(T) () (4 6 1 -2 3) (2 3 4) ⋮

2. {*bool char*} `WRITE` {`INT,BOOL,FLOAT,CHAR,SEGDES`} {*int* \| *bool* \| *float* \| *char* \| *segdes*}

   Writes a vector of the appropriate type from the stack to the standard output stream. A `WRITE` of a segment descriptor prints the lengths of the segments enclosed in square brackets. A `WRITE CHAR` of an integer string assumes the elements of the string are `ASCII` values for characters. The return values are a status flag and an error message (see `READ`).

(4 6 1 -2 3) (2 3 4) ⋮	`WRITE INT`	(T) () (2 3 4) ⋮

3. {*bool char int*} `FOPEN` {*char int*}

   Opens a file. The first argument is interpreted as a character string representing the pathname of a file to open. The second arguement is an integer vector of length 1 that specifies the mode of the opened file:

   1 open read only

   2 create for writing (creates new file, truncates existing file)

   3 append on write (creates new file, appends to old)

   Three vectors are returned to the stack: a status flag, an error message, and an integer. This integer should be thought of as a file descriptor and will be used to refer to the file being opened by any other file I/O functions. Many files can be open at the same time; the exact number is system dependent.

(1) "path/to/file" ⋮	`FOPEN`	(T) () (17) ⋮

4. {*bool char*} `FCLOSE` {*int*}

   Closes the file referred to by the argument (see `FOPEN`). Returns a status flag and an error message.

(17) ⋮	`FCLOSE`	(T) () ⋮

5. {*bool char*} `FWRITE` {`INT,BOOL,FLOAT,CHAR,SEGDES`} {*int int* \| *int bool* \| *int float* \| *int char* \| *int segdes*}

   Same as *WRITE*, except that the vector is written to the file referred to by the integer argument.

(17) (4 6 1 -2 3) (2 3 4) ⋮	`FWRITE INT`	(T) () (2 3 4) ⋮

6. {*bool char int* | *bool char bool* | *bool char float* | *bool char segdes*} `FREAD` {`INT,BOOL,FLOAT,SEGDES`} {*int*}

   Same as *READ*, except that the vector is read from the file referred to by the integer argument.

(17) (2 3 4) ⋮	`FREAD INT`	(T) () (4 6 1 -2 3) (2 3 4) ⋮

7. {*bool char int char*} `FREAD_CHAR` {*int int char*}

   This function is used for reading character input. The first argument refers to the file from which to read. The second argument is an integer giving the maximum number of characters to read, with a negative number indicating no maximum. The third argument is a vector of characters that should act as delimiters and halt the read. The return values are a status flag and error message, followed by a stopping condition and the result. The stopping condition indicates the reason that a successful read terminated and has the value -1 if an end of file was encountered, -2 if the specified maximum number of characters were read, and otherwise holds the delimeter that was read.

(17) (20) ";.:" ⋮	`FREAD_CHAR`	(T) () (-1) "testing" ⋮

2.6 Miscellaneous features

Comments are delimited by `{` and `}`. Multiline comments are permitted. The language is case-sensitive.