VCOP Kernel C

Reference Guide

Literature Number: SPRUHB9B
January 2012–Revised November 2012

Contents

Reference Guide
SPRUHB9B–January 2012–Revised November 2012

VCOP Kernel C Reference Guide

This document describes the Embedded Vector Engine (EVE) Vector Core (VCOP) kernel C compiler, a tool packaged with the ARP32 compiler used for programming the VCOP. The vector core is written in a C-like high-level language called VCOP Kernel C, which is specified here. The kernel compiler translates a kernel written in VCOP Kernel C into a plain C function containing the low-level assembly statements of the kernel.

1 VCOP Kernel C Language

1.1 Notational Conventions

This document uses the following conventions:

- Program listings and examples are shown in a special typeface.

  Here is a sample of C code:

```
#include <stdio.h>
main()
{
                  printf("Hello, cruel world!\n");
}
```

- In syntax descriptions, the instruction, command, or directive is in a bold typeface and parameters are in an italic typeface. Portions of a syntax that are in bold should be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered.

1.2 Summary

Example 1. VCOP Kernel

```
void SYMBOL ( <type> SYMBOL, <type> SYMBOL, ... )
{
   <declarations>
   <agen_statements>
   <init_statements>
   <vload_statements>
   for (LCV = 0; LCV < <pexpr>; LCV++)
   {
      <declarations>
      <agen_statements>
      <init_statements>
      <vload_statements>
      for (LCV = 0; LCV < <pexpr>; LCV++)
      {
         <declarations>
         <agen_statements>
         <init_statements>
         <vload_statements>
         for (LCV = 0; LCV < <pexpr>; LCV++)
         {
            <declarations>
            <agen_statements>
```

Example 1. VCOP Kernel (continued)

```
            <init_statements>
            <vload_statements>
            for (LCV = 0; LCV < <pexpr>; LCV++)
            {
               <declarations>
               <init_statements>
               <agen_statements>
               <vload_statements>

               <vop_statements>
               <vstore_statements>
               <vexit_statement>
            }
         }
         <vstore_statements>
      }
      <vstore_statements>
   }
   <vmove_statement>
   <vexit_label>
}
```

Example 2 illustrates a basic kernel function. A kernel file can contain multiple functions like this as long as the function names are different. In addition, each function can contain more than one loop, allowing for the ability to perform multiple operations on a set of data per function.

Example 2. Basic Kernel Function

```
<declaration> :
      __vector Vreg ;
      __agen Agen ;

<init_statement> :
      Vreg.clear() ;       // Vreg = 0;
   |  Vreg = <pexpr> ;     // VINIT
   |  Vreg = Vreg ;        // VCMOV

<agen_statement> :
      Agen = LCV*(<pexpr>) + LCV*(<pexpr>) + LCV*(<pexpr>) + LCV*(<pexpr>) ;
      Agen = 0 ;

<vload_statement> :
      Vreg = <pexpr> [ Agen ] ;

<vop_statement> :
     Vreg = Vreg op Vreg ;
   | Vreg = SYMBOL ( Vreg, Vreg, ... ) ;
   | Vreg = Vreg           // VOR
   | if ( (LCV == 0) (&& (LCV == 0))* ) Vreg = Vreg;               //VCMOV I<X>_ZERO
   | if ( last(LCV,<pexpr>) (&& last(LCV,<pexpr>))* ) Vreg = Vreg; //VCMOV LAST_I<X>
   | if (Vreg[0]) goto <name>;

<vstore_statement>
      <pexpr> [ Agen ] = Vreg ;
      Vreg = Vreg ;         // VCMOV

<vmove_statement>
      Vreg = Vreg ;         // VCMOV

<vexit_label>
      <identifier>:;
```

1.3 Language Elements

The following sections contain an informal grammatical specification for the VCOP Kernel C language. The grammar syntax is specified in the boxed text within the sections. The grammar uses a handful of placeholder symbols. (Note: The SYMBOL and TYPE parameters are described in Section 1.4.)

LCVs (Loop Control Variables) are the loop counter variables. They can only appear in the loop header and at other explicitly specified points. Any C identifier can be used.

LCV	**:** *SYMBOL*

Agens are symbols corresponding to the address generators.

Agen	**:** *SYMBOL*

Vregs are symbols corresponding to vector registers.

Vector registers can be specified using symbolic names by declaring them in the kernel as type __vector. Any C identifier can be used as the name of these registers. Within this specification any identifier beginning with uppercase 'V' represents a Vreg (for example Vsrc, Vdest, Vpred).

V*reg*	**:** *SYMBOL*

Integer constants are required in various contexts. They can be expressed using normal C syntax, for example 0xFF00 or 1234. In this specification any identifier beginning with uppercase 'K' represents an integer constant.

K	**:** *integer constant*

A TYPE is a single-token expression of a C type. See Section 1.4.

TYPE	**:** *SYMBOL* or *keyword*

A <pexpr> is a C expression that appears in the kernel that is used to initialize a parameter register in the parameter block. These are parsed and evaluated when possible by the kernel compiler.

<pexpr>	**:** *C expression*

In general any C expression that is valid where the <pexpr> is encountered can be used. However, the following caveats apply:

- A <pexpr> cannot have side effects.
- A <pexpr> cannot depend on the evaluation of any other expression in the kernel. In other words, a <pexpr>'s value **must** be invariant throughout the kernel. A <pexpr> cannot refer to an LCV, an Agen, a Vreg, or another <pexpr>.

These rules are not checked by the kernel compiler.

Within this document any identifier enclosed in angle brackets is taken to be a <pexpr>.

Within <pexprs> the following idioms are available:

- For a kernel parameter declared with one of the __vptr data types, the idiom sizeof(*p) evaluates to the size in bytes of each data element. For example, given the declaration:

  ```
  void kernel(... __vptr_int16 buffer)
  ```

  The expression sizeof(*buffer) evaluates to 2.

- The constant VCOP_SIMD_WIDTH evaluates to the width (number of elements) of the SIMD vector registers. For the current VCOP variant this value is 8.
- Constants may be denoted as signed/unsigned integers via C-style casts to 'int' or 'unsigned'. For example, (int)0xFFFFFFFF would be equivalent to -1.

These idioms are useful for doing address computations in a type-independent way.

1.4 Kernel Declaration

The syntax for a kernel declaration is:

void *SYMBOL* **(** *TYPE SYMBOL*, *TYPE SYMBOL*, ... **)**
{
}

The return type of a kernel **must** be void.

The parameter names, represented by SYMBOL, can be any valid C identifiers.

For parameters that are used as the base address of load and store operations, the TYPE identifies the data type of the load and store, and **must** be one of the following:

__vptr_int8	Signed 8 bit data
__vptr_uint8	Unsigned 8 bit data
__vptr_int16	Signed 16 bit data
__vptr_uint16	Unsigned 16 bit data
__vptr_int32	Signed 32 bit data
__vptr_uint32	Unsigned 32 bit data

When the kernel function is called, the argument expressions corresponding to these parameters can have the corresponding built-in types. For example:

```
unsigned int my_data[SIZE];              // data buffer declaration in host code
void kernel(... __vptr_uint32 input ...);   // kernel declaration

// call the kernel
kernel(... my_data ...);    // my_data[] implicitly converted to __vptr_uint32
```

Parameters that are not used as a base address in the kernel can be declared using other C types. Such parameters are not significant to the kernel compiler and can only appear in a <pexpr>. However, the following restrictions apply:

- The parameter types must be single tokens. For example ‘int’ is OK but ‘unsigned int’ is not. Type qualifiers (const, volatile, restrict) are not allowed. Use a typedef to create one-word types.
- Derived types are restricted to arrays and functions.

Examples:

```
void kernel(    unsigned     a,       // OK, built-in one-word type
                MYTYPE*      b,       // OK, pointer to one-word type
                MYTYPE*      c[],     // OK, array of pointers
                MTTYPE       (*d)(),  // error: pointer to function
                volatile int *e)      // error: no qualifiers allowed
```

1.5 For Loops

The body of a kernel consists of a ‘for’ loop, with up to three additional loops nested within it, for a total of up to four nesting levels.

The syntax for a for loop is:

```
for (int LCV = 0; LCV < <pexpr>; ++LCV)
{
...
}
```

The LCV (Loop Control Variable) can be any identifier.

The three parts of the loop header **must** be exactly as shown and adhere to the following rules:

- The LCV must be declared in the initialization statement as int, and initialized to 0
- The comparison part must be specified as LCV less-than <*pexpr*>, where the <pexpr> represents the trip count of the loop (see the rules for <pexpr> in Section 1.3).
- The trip count must be greater than zero; that is, the loop must execute at least once.
- The increment part must increment the LCV by exactly one, using any of the forms LCV++, ++LCV, or LCV += 1. No other statements or expressions may appear in the for loop header
- The loop body must be enclosed in braces even if it consists of only one statement

1.6 Vector Register Declarations

Vector registers specified with symbolic names **must** be declared as type __vector. The declaration **must** occur prior to the first reference to the symbol. Any valid C identifier can be used.

The syntax for a vector register declaration is:

```
__vector Vreg;
```

In addition, if the identifier used is of the form V0, V1… V15, then these registers will map directly to their corresponding hardware register name. This allows for manual register allocation in kernels with heavy register pressure.

1.7 Initialization

There are three forms of the initialization statement. The beginning of the declaration specifies the form:

- The first explicitly initializes all elements of a vector register from the given <pexpr>. This statement can appear at any loop nesting level, but **must** appear before any other statement that references the given Vreg. The syntax is:

```
Vreg = <pexpr> ;
Vreg.clear();
```

  The first line is equivalent to using VINT. The second line is equivalent to Vreg = 0.
- The second form initializes all elements of a vector register to the current value of a loop control variable, which is equivalent to VINIT. This form **must** appear within the loop controlled by the named variable (including nested within an inner loop). The syntax is:

```
Vreg = LCV;
```

- The third form copies one vector register to another, which is equivalent to VCMOV. It can appear at any loop nesting level. The syntax is:

```
Vreg = Vreg ;
```

The remainder of the statements have the same syntax:

```
if ( (LCV == 0) (&& (LCV == 0))* ) Vreg = Vreg;
if ( last(LCV,<pexpr>) (&& last(LCV,<pexpr>))* ) Vreg = Vreg;
```

The first if statement is equivalent to VCMOV I<X>_ZERO. The second if statement is equivalent to VCMOV LAST_I<X>.

1.7.1 Conditional Move Supplementary Syntax

There exist two more forms of the conditional move which take user input instead of choosing the condition intelligently by instruction position. Due to the way expressions are parsed, the syntax for these forms are highly constrained.

A user-defined conditional move must declare two things to the translator:

- LCVs to check:
  - These are the full names of the LCVs used in the loop statement.
  - The LCV for a foreach loop is illegal in conditional moves.
  - Due to hardware constraints, if an LCV is considered, all LCVs in a deeper ‘for’ loop must also be considered.
- End condition:
  - Includes checking when the loop is in its first iteration or its last iteration.
  - The first iteration form simply compares LCVs to 0.
  - The last iteration form uses a special ‘last’ macro that returns true if the LCV given matches the last iteration of a loop using <pexpr> as its tripcount. This expression must be the same as the one used in the ‘for’ loop statement.

Note that mixing first iteration and last iteration conditions is illegal, as is comparing an LCV to its tripcount with ‘==’. If there is more than one condition, then there must be a set of parentheses around each one.

1.8 Address Generation

The syntax for address generation and initialization is:

```
__agen Agen;
Agen = LCV*(<pexpr>) + LCV*(<pexpr>) + LCV*(<pexpr>) + LCV*(<pexpr>);
Agen = 0 ;
```

VCOP uses inductive form addressing. That is, addresses are formed by adding base addresses to address generators, which are successively incremented after each iteration. Note that an agen term is required to be an LCV multiplied by a single paramter expression;

```
__agen Agen; Agen = LCV*<pexpr1>/<pexpr2>   // Illegal, equivalent to (LCV*<pexpr1>)/<pexpr2>
Agen = LCV*(<pexpr1>/<pexpr2>) // Legal
```

The general form of an address expression in a load is:

```
Vreg = <pexpr>[Agen];
```

The general form of an address expression in a store is:

```
<pexpr>[Agen] = Vreg;
```

Here the address is formed by the offset value in the address generator Agen to the base address represented by the <pexpr>.

Calculation of the increment amounts is somewhat tedious and error-prone. Therefore the tool supports an abstraction that allows per-iteration offsets to be expressed using a linear combination of coefficients and LCVs. The general form of the expression is a sum-of products, where each addend is a product of an LCV and a coefficient <pexpr>.

The agen expression must be a sum-of products, where each addend is a product of an LCV and a coefficient <pexpr>, or 0.

For example, the following illustrates the calculation of the address offset within a 2D matrix:

```
for (int row = 0; row < nrows; ++row)
{
   for (int col = 0; col < ncols/__VCOP_SIMD_WIDTH; ++col)
   {
      A0 = row*ncols*sizeof(*input) + col*sizeof(*input);
      Vn = input[A0];
      ...
   }
}
```

In this example the LCVs are row and col. The coefficient <pexpr> for row is 'ncols*sizeof(*input)' and the coefficient for col is 'sizeof(*input)'. The sizeof expressions are necessary because the offset in the Agen is in bytes.

1.9 Load Statements

This is the basic form of a vector load statement. It can be at any level of the function **except** outside of a loop containing a foreach() repeat loop structure.

V*reg* **=** <*pexpr*> **[** *Agen* **] ;**

In addition, the load **must** be placed at the highest nested loop possible given the address generator's values. For example, in a loop with indices I1, I2, I3, and I4, respective to nesting order, a load using an agen defined by I1 and I2 must be placed in the I2 loop and not inside the I3 or I4 loops. This is because loads only occur in the hardware when an address changes.

Note: This constraint does not apply to Histogram or Table Lookup loops

Because loads only occur when an agen's value has changed, inner loop loads that use address generators whose increments are not constant or a function of macros and data pointer type sizes will be pessimistically defaulted to conditional to avoid erroneous assumptions. Therefore, it is beneficial to use statically defined increments when possible so that more accurate determinations can be made. The option --vcop_unconditional_loads exists to change this default behavior should the user know that increments are nonzero, or that assuming otherwise will not affect behavior.

The <pexpr> represents the base address for the load. It **must** be a symbol from the parameter list declared with one of the __vptr types from section Error: Reference source not found. The kernel compiler uses this type to determine the data type of the load.

VCOP supports various distribution options that control how data elements read from memory are mapped into the element of the vector register destination. These are specified though a modifier syntax which is taken from member function invocations in C++.

The various modifiers are:

```
Vreg = <pexpr>[Agen].npt();
Vreg = <pexpr>[Agen].onept();
Vreg = <pexpr>[Agen].circ2();
Vreg = <pexpr>[Agen].ds2();
Vreg = <pexpr>[Agen].us2();
Vreg = <pexpr>[Agen].dist(K1,K2,K3,K4,K5,K6,K7,K8);
Vreg = <pexpr>[Agen].dist( <pexpr> );
Vreg = <pexpr>[Agen].nbits();
```

For more information on these distributions, see the *Embedded Vector Engine (EVE) Programmer's Guide* (SPRUHC1).

The 'dist' modifier corresponds to the 'custom' distribution option (Described in the Programmer's Guide), which is specified as an array of offsets, each of which specifies the offset of the memory datum to be loaded into one lane of the SIMD vector. These offsets are in units of the memory data type.

In the first form of the dist modifier, the array of offsets is given as a list of constant arguments to the modifier. In the second form, the vector is given as a pointer to (or array of) integers, where each element is given by <pexpr>[0], <pexpr>[1], and so on.

1.9.1 Deinterleaved Loads

There is an additional distribution option for deinterleaved loads, which reads data from memory into a pair of alternating registers. The destination pair is specified using a special pair syntax, as follows:

(V*reg1***, V***reg2***) =** <*pexpr*>**[***Agen***].deinterleave();**

1.10 Vector Operations

Move

```
Vdst = Vsrc;                                  // VCMOV/VOR
if (LCV == 0 && ... LCV == 0) Vdst = Vsrc;    // VCMOV
```

Add, Subtract

```
Vdst = Vsrc1 + Vsrc2;                         // VADD Vsrc1, Vsrc2, Vdst
Vdst += Vsrc;                                 // VADD Vsrc, Vdst, Vdst
Vdst += Vsrc1 + Vsrc2;                        // VADD3 Vsrc1, Vsrc2, Vdst, Vdst
Vdst = Vsrc1 + Vsrc2 + Vdst;                  // VADD3 Vsrc1, Vsrc2, Vdst, Vdst
Vdst = Vsrc1 + hi(Vsrc2);                     // VADDH Vsrc1, Vsrc2, Vdst

Vdst = Vsrc1 - Vsrc2;                         // VSUB Vsrc1, Vsrc2, Vdst
Vdst -= Vsrc;                                 // VSUB Vdst, Vsrc, Vdst

(Vsrc1, Vsrc2).addsub();                      // VADDSUB Vsrc1,Vsrc2

Vdst += Vsrc1 - Vsrc2;                        // VADIF Vsrc1, Vsrc2, Vdst, Vdst
Vdst = Vsrc1 - Vsrc2 + Vdst;                  // VADIF Vsrc1, Vsrc2, Vdst, Vdst
```

Absolute Value

```
Vdst = abs(Vsrc);                             // VABS Vsrc, Vdst
Vdst = abs(Vsrc1 - Vsrc2);                    // VABSDIF Vsrc1, Vsrc2, Vdst
Vdst += abs(Vsrc1 - Vsrc2);                   // VSADD Vsrc1, Vsrc2, Vdst, Vdst
```

Multiply, Multiply/Accumulate

```
Vdst = Vsrc1 * Vsrc2;                         // VMPY Vsrc1, Vsrc2, Vdst
Vdst += Vsrc1 * Vsrc2;                        // VMADD Vsrc1, Vsrc2, Vdst, Vdst
Vdst = Vsrc1 * Vsrc2;                         // VMSUB Vsrc1, Vsrc2, Vdst, Vdst
```

And, Or, Xor, Not

```
Vdst = ~Vsrc;                                 // VNOT Vsrc, Vdst

Vdst = Vsrc1 & Vsrc2;                         // VAND Vsrc1, Vsrc2, Vdst
Vdst &= Vsrc;                                 // VAND Vsrc, Vdst, Vdst
Vdst &= Vsrc1 & Vsrc2;                        // VAND3 Vsrc1, Vsrc2, Vdst, Vdst
Vdst = Vsrc1 & Vsrc2 & Vdst;                  // VAND3 Vsrc1, Vsrc2, Vdst, Vdst

Vdst = Vsrc1 & ~Vsrc2;                        // VANDN Vsrc1, Vsrc2, Vdst
Vdst &= ~Vsrc;                                // VANDN Vdst, Vsrc, Vdst

Vdst = Vsrc1 | Vsrc2;                         // VOR Vsrc1, Vsrc2, Vdst
Vdst |= Vsrc;                                 // VOR Vsrc, Vdst, Vdst
Vdst |= Vsrc1 | Vsrc2;                        // VOR3 Vsrc1, Vsrc2, Vdst, Vdst
```

```
Vdst = Vsrc1 | Vsrc2 | Vdst;              // VOR3 Vsrc1, Vsrc2, Vdst, Vdst

Vdst = Vsrc1 ^ Vsrc2;                     // VXOR Vsrc1, Vsrc2, Vdst
Vdst ^= Vsrc;                             // VXOR Vsrc, Vdst, Vdst
```

Shift

```
Vdst = Vsrc1 << Vsrc2;                    // VSHF Vsrc1, Vsrc2, Vdst
Vdst |= Vsrc1 << Vsrc2;                   // VSHFOR Vsrc1, Vsrc2, Vdst, Vdst
```

Min, Max

```
Vdst = min(Vsrc1, Vsrc2);                 // VMIN Vsrc1, Vsrc2, Vdst
Vdst = max(Vsrc1, Vsrc2);                 // VMAX Vsrc1, Vsrc2, Vdst
(Vsrc1, Vsrc2).minmax();                  // VSORT2 Vsrc1, Vsrc2
(Vsrc2, Vdst2) = minf(Vsrc1,Vsrc2)        // VMINSETF Vsrc1, Vsrc2, Vsrc2, Vdst2
(Vsrc2, Vdst2) = maxf(Vsrc1,Vsrc2)        // VMAXSETF Vsrc1, Vsrc2, Vsrc2, Vdst2
```

Comparisons

```
Vdst = Vsrc1 == Vsrc2;                    // VCMPEQ Vsrc1, Vsrc2, Vdst
Vdst = Vsrc1 > Vsrc2;                     // VCMPGT Vsrc1, Vsrc2, Vdst
Vdst = Vsrc1 < Vsrc2;                     // VCMPGT Vsrc2, Vsrc1, Vdst
Vdst = Vsrc1 >= Vsrc2;                    // VCMPGE Vsrc1, Vsrc2, Vdst
Vdst = Vsrc1 <= Vsrc2;                    // VCMPGE Vsrc2, Vsrc1, Vdst
Vdst = pack(Vsrc1 >= Vsrc2);              // VBITPK Vsrc1, Vsrc2, Vdst
```

Conditional Assignment

```
Vdst = select(Vsrc1, Vsrc2, Vdst);        // VSEL Vsrc1,Vsrc2,Vdst,Vdst
Vdst = unpack(Vsrc1, Vsrc2);              // VBITUNPK Vsrc1, Vsrc2, Vdst
(Vsrc1,Vsrc2).swap(Vcond);                // VSWAP Vcond, Vsrc1, Vsrc2
```

Bit Manipulation

```
Vdst = round(Vsrc1, Vsrc2);               // VRND Vsrc1, Vsrc2, Vdst
Vdst = apply_sign(src1, src2);            // VSIGN Vsrc1, Vsrc2, Vdst

Vdst = count_bits(Vsrc);                  // VBITC Vsrc, Vdst
Vdst = reverse_bits(Vsrc);                // VBITR Vsrc, Vdst
Vdst = transpose_bits(Vsrc);              // VBITTR Vsrc, Vdst
Vdst = leading_bit(Vsrc1, Vsrc2);         // VLMBD Vsrc1, Vsrc2, Vdst
Vdst = binlog(Vsrc);                      // VBINLOG Vsrc, Vdst
Vdst = interleave_bits(Vsrc1, Vsrc2);     // VBITI Vsrc1, Vsrc2, Vsrc3
(Vdst1, Vdst2) = deinterleave_bits(Vsrc); // VBITDI Vsrc, Vdst1, Vdst2
(Vdst1, Vdst2) = jus16(Vsrc);             // VSHF16 Vsrc, Vdst1, Vdst2
```

Interleave, Deinterleave

```
(Vsrc1, Vsrc2).interleave();              // VINTRLV Vsrc1,Vsrc2
(Vsrc1, Vsrc2).deinterleave();            // VDINTRLV Vsrc1, Vsrc2
(Vsrc1, Vsrc2).interleave2();             // VINTRLV2 Vsrc1,Vsrc2
(Vsrc1, Vsrc2).interleave4();             // VINTRLV4 Vsrc1,Vsrc2
(Vsrc1, Vsrc2).deinterleave2();           // VDINTRLV2 Vsrc1, Vsrc2
```

The operands in these instructions **must** be of the same type as listed here. Balanced parentheses are allowed in operations not involving function calls or left shifts to aid in readability.

1.10.1 Destructive Instruction Register Aliasing

Vector registers can be aliased with another name in the following instructions:

```
(Vsrc1, Vsrc2).addsub();                  // VADDSUB Vsrc1,Vsrc2
(Vsrc1, Vsrc2).minmax();                  // VSORT2 Vsrc1, Vsrc2
(Vsrc1, Vsrc2).interleave();              // VINTRLV Vsrc1,Vsrc2
(Vsrc1, Vsrc2).deinterleave();            // VDINTRLV Vsrc1, Vsrc2
(Vsrc1, Vsrc2).interleave2();             // VINTRLV2 Vsrc1,Vsrc2
(Vsrc1, Vsrc2).interleave4();             // VINTRLV4 Vsrc1,Vsrc2
(Vsrc1, Vsrc2).deinterleave2();           // VDINTRLV2 Vsrc1, Vsrc2
(Vsrc1, Vsrc2).swap(Vcond);               // VSWAP Vcond, Vsrc1, Vsrc2
```

To take advantage of this syntax, the user must provide a destination pair to the line, for example:

```
(Vsum, Vdiff) = (Vsrc1, Vsrc2).addsub();  // VADDSUB Vsrc1,Vsrc2
```

After that point in the loop, the source registers will be considered dead, and all access to the results of the instruction must reference the destination registers. Failure to do so will result in extra move cycles being present in the final loop. The kernel compiler will recognize these cases and throw a warning.

If the source registers in these instructions come from a conditional load, then the compiler will interpret this alias as a vector-to-vector move followed by the instruction. This could cause a performance loss over not aliasing.

1.11 Store Statements

This is the basic form of a vector store statement. It **must** within the outermost for loop, or a nested inner loop, following any vector operation statements.

<pexpr> **[** *Agen* **] = V***reg* **;**

The <pexpr> represents the base address for the store. It **must** be a symbol from the parameter list declared with one of the __vptr types from Section 1.4. The kernel compiler uses this type to determine the data type of the store.

As with loads, VCOP supports various distribution options that control how data elements are mapped from elements of the vector register into memory. These are specified though a modifier syntax applied to the address expression:

The various store modifiers are:

```
<pexpr>[Agen].npt() = Vreg;
<pexpr>[Agen].onept() = Vreg;
<pexpr>[Agen].ds2() = Vreg;
<pexpr>[Agen].offset_np1() = Vreg;
<pexpr>[Agen].s_scatter(Vindex) = Vreg;
<pexpr>[Agen].p_scatter(Vindex) = Vreg;
<pexpr>[Agen].skip() = Vreg;
```

The scatter form uses the elements of the Vindex vector as offsets to store the corresponding elements of the source vector. The index vector register **must** use byte offsets and **must** be unique. No more than one vector register can be used as scatter's offset register. The difference between s_scatter() and p_scatter() is that s_scatter takes 8 cycles to store 8 values, while p_scatter() stores in parallel and takes 1 cycle. However, the p_scatter() distribution causes errors when duplicate memory locations are detected, whereas s_scatter() forces the latest duplicate to take precedence.

1.11.1 Predication

Stores can be predicated by the elements of a vector register. A zero element in the predicate register blocks the store of the corresponding element of the source register.

<pexpr> **[** *Agen* **] = V***reg***.predicate(V***pred***) ;**

A predicate can be used in conjunction with the distribution options shown in Section 1.11. For example:

```
data[A0].ds2() = Vsrc.predicate(Vindex);
```

The hardware requires registers V1, V2, and V3 to be the predicate register. In addition, loads only go into even registers (V0, V2, .., V14). This means that to load more than one register for predication, the user **must** load the values into an intermediate register and then move it into another register. The tool will take care and make sure the move places the values into one of the three predicate registers.

1.11.2 Interleaved Stores

Interleaved stores store elements from an alternating pair of vector registers into memory. This is specified using the same pair syntax as interleaved loads:

<*pexpr*> **[** *Agen* **].interleave() = (V***reg1***, V***reg2***);**

Interleaved stores cannot be predicated, but predication can still be done by first interleaving the two vector registers, and then performing two n-point predicated stores.

1.11.3 Collating Stores

Collating stores are similar to predicated stores in that the predicate register controls which elements of the source vector are written. However, in the collating store each element for which the predicate is true is written sequentially, whereas in the predicated store each element is written at its normal location according to the address generator and the distribution option. In this way the collating store is the inverse of the expanding load.

<*pexpr*> **[collate(V***pred***)] = V***src***;**

To indicate that a loop is to be used for a collate store, you must designate a single base address that will be used as the target of the store. To do this, the following function must be called just before the first **for** loop. This should not be called before a **foreach** repeat loop but instead the first **for** loop inside the repeat loop.

_COLLATE(<*pexpr***>);**

Collating stores do not have explicit address generators associated with them. Each such store has a built-in implicit address generator that is initialized from the base address <pexpr>. Each time the store executes the address generator is incremented by the number of items stored, corresponding to the number of non-zero elements in the predicate register.

The <pexpr> used for collating stores can only be used for one store within the kernel. For example:

```
output[collate(V1)] = Vsrc1;
output[collate(V2)] = Vsrc2;      // error, 'output' used for two collating stores
```

Collating stores cannot be used with other distribution options.

1.12 Rounding and Saturation

Saturation occurs in two contexts: stores, and selected vector operations. Saturation is expressed via the saturate( ) modifier applied to the vector register on the right-hand-side of the assignment. There are several forms, corresponding to the saturation modes of the hardware:

<*pexpr*>**[***Agen***] = V***reg***;**	No saturation
<*pexpr*>**[***Agen***] = V***reg***.saturate();**	Saturate to type of <pexpr>
<*pexpr*>**[***Agen***] = V***reg***.saturate(-***K***,***K***);**	SYMM, signed data
<*pexpr*>**[***Agen***] = V***reg***.saturate(0,***K***);**	SYMM, unsigned data
<*pexpr*>**[***Agen***] = V***reg***.saturate(***K1***,***K2***);**	ASYMM, K1 == -K2-1
<*pexpr*>**[***Agen***] = V***reg***.saturate(**<*bits*>**);**	ASYMM, half-open interval Signed: -[-2^(bits--1), 2^(bits-1)-1] Unsigned: [0, 2^(bits)-1]
<*pexpr*>**[***Agen***] = V***reg***.saturate(**<*min*>**,** <*minset*>**,** <*max*>**,** <*maxset*>**);**	4PARAM
<*pexpr*>**[***Agen***] = V***reg***.saturate(**<*min*>**,** <*max*>**);**	4PARAM, with maxset=max, minset=min

- Constants are interpreted as integers whose signedness is dependent on the type of the address used, but are stored as shorts if they can be represented in 16 bits. This avoids issues where users might want to represent a negative bound through an unsigned value. For example, using a signed short 65533 (0xfffd) is equivalent to -3. However, it will be treated as 65533 in the VCOP-C Kernel Compiler. Instead, use 4294967293 (0xfffffffd).
- SYM and ASYMM modes require constant, nonnegative bounds. Non-constant expressions and negative constants (K or K1 < 0) used as saturation bounds always generate the 4PARAM form.
- The ASYMM forms generate errors for unsigned data.
- The ASYMM32 form requires that the expressions be in exactly one of the two forms shown.
- The expressions used as arguments to the 4PARAM forms are all <pexprs> that **must** fit in 16 bits. The assembler does not check this.

Rounding can be applied to store instructions separately or in conjunction with saturation. Rounding is expressed with the round() modifier:

<*pexpr*>**[***Agen***] = V***reg***;**	No rounding
<*pexpr*>**[***Agen***] = V***reg***.round(**<*bits*>**);**	value = (value + 1<<(bits-1)) >> bits
<*pexpr*>**[***Agen***] = V***reg***.truncate(**<*bits*>**);**	value = value >> bits

The parameter <bits> is a <pexpr>, so it can be a constant or an expression. If it is an expression, its value **must** be 0-39. The assembler does not check this.

Rounding, saturation, and predication can be combined. For example:

```
data[Agen] = Vreg.round( ... ).saturate( ... );  // combine rounding and sat.
data[Agen] = Vreg.saturate( ... ).predicate(Vn);  // combine sat. and predication
```

1.12.1 Rounding in Operations

Rounding, in addition to being available for stores, is also available for some operations. There are three modes: rounding, truncation, and left shift. This is specified as a modifier on the result expression:

V*dst***= (***vector expression***).round(**<*bits*>**);**	rnd_mode = round
V*dst***= (***vector expression***).truncate(**<*bits*>**);**	rnd_mode = truncate
V*dst***<<**<*bits*>**);**	rnd_mode = left-shift

Examples:

```
Vdst = (Vsrc1 * Vsrc2).round(15);        // VMPY with rounding
Vdst += (Vsrc1 * Vsrc).truncate(15); // VMADD with truncation
Vdst += (Vsrc1 * Vsrc2) << 1;          // VMADD with left shift
```

1.13 Repeat Loop

You can specify that a specific VLOOP be repeated a number of times, each time using a different set of parameter registers. To signify that a loop is to be repeated, a special ‘foreach’ syntax is used:

foreach (*LCV***,**<*pexpr*>**, *K***)**
...**VLOOP**
}

The <pexpr> is the trip count; the number of times that you want the vloop to run. The constant is a user-defined ‘maximum’ number of times that you desire to run the loop.

All load/store array parameters that use the LCV **must** be declared as one of these new types:

__vptr_int32_arr	__vptr_uint32_arr
__vptr_int16_arr	__vptr_uint16_arr
__vptr_int8_arr	__vptr_uint8_arr

While simple parameter arrays can be declared as signed or unsigned '*short**'

The repeat loops are unlike 'for' loops and must adhere to the following rules:

- While there can be agen and register declarations, there can be no definitions inside the 'foreach' loop and before the first nested 'for' loop
- There can be no operations outside the first nested 'for' loop:

```
void kernel(__vptr_uint16_arr in_ptr, unsigned short* parms)
{
        foreach (I0,cnt,5)
        {
                V2 = parms[I0];
                for(I1 = 0;I1 < 2; I1++)
                {
                        ...
                        V1 = in_ptr[I0][A1];
                        ...
                }
        }
}
```

In this loop example, the kernel compiler assumes that in_ptr is an array of size at least 'cnt', which does not exceed 5, and that contains address locations for input arrays. It uses the array at location in_ptr[0] for the first iteration, in_ptr[1] for the second, etc. as the base address of the load into V1.

The vector register V2 takes the value of parms[0..cnt] as the VLOOP is repeated.

1.14 Early Exit

VCOP Kernel C supports the early exiting of a VLOOP given the truth values of a specified vector register's 0th lane. Only two levels of exit are possible. One level exits only the VLOOP, allowing a repeat loop structure to continue with the next iteration. The second level exits outside of the repeat structure, moving on to a different VLOOP or continuing program execution.

The syntax for this early exit takes the form of a *goto* statement. The user specifies a conditional and a label and the kernel compiler will generate the proper exit level and instruction.

For example, to end a VLOOP's execution when two vector registers are equal:

```
for(int i3=0; i3 < 1; i3++)
{
         for(int i4=0; i4<1; i4++)
         {
                  __vector V1 = 4;
                  __vector V2 = 4;
                  __vector V3 = V1 == V2;
                  if (V3[0]) goto end_loop1;
                  ...
         }
}
end_loop1:;
```

For a repeat loop, the exit instruction can either be a regular loop exit as above, or it can be placed as the last instruction of the foreach block and the label placed just outside to indicate that when true, execution will jump out of the foreach loop and continue through the function.

```
foreach(I0, rpt_count, 5)
{
        __vector Vmax = 20
        for(int i4=0; i4<10; i4++)
```

```
        {
                __vector V1 = 5;
                __vector V2 += V1;
                __vector V3 = V2 > Vmax;
                ...
         }
         if (V3[0]) goto end_loop1;
}
end_loop1:;
```

There can be only one early exit instruction per vloop. This will be checked by the tool.

1.15 Circular Buffers

The user can use circular addressing on loads and stores. There is support for buffers of size 1K, 2K, 4K, 8K, 16K, and 32K. To trigger the compiler to generate code for circular buffer addressing, add this syntax at the end of the pexpr-agen pair:

<*pexpr*>**[<***agen***>] % CIRC**_<*SZ*>**;**

1.16 Histogram

The histogram loop is a different type of loop that allows for the updating of a histogram through normal increments, or through weighted data that is initialized or loaded from memory. Due to the constrained nature of the histogram loop on the VCOP itself, the language syntax presented here must be followed strictly to prevent errors.

1.16.1 Loop Structure

To indicate that an upcoming outer **for** loop is to be considered a histogram loop, the user must call the following function just before the **for** loop. Any initializations or moves must come after this identifier or they will be ignored.

_HISTOGRAM(*K***);**

The constant K is the number of parallel tables to use. Valid values are 1, 2, 4, and 8.

The _HISTOGRAM structure must come before the outer **for** loop in the presence of a **foreach** loop.

The loop itself may contain only a small set of instructions:

- A minimum of 2 and a maximum of 3 agen declarations
  - All agens must be unique to the instruction that uses it. That means that even if two agens are equal, they must still both be declared and defined as such.
- An index load to determine which data points to update in the histogram
- A weight load/init that will be used as increment value for the histogram data
- Histogram update which will load, increment, and store the histogram data

1.16.2 Index Load

The index load loads a vector register with a number of indices that will be used to access various bins in the histogram. This instruction is as a normal load with the constraint that distribution may only be 1PT, NPT, US2, DS2, or NBITS. The NBITS load mode is only supported for 1 or 8 parallel histograms.

1.16.3 Weight Load/Initialization

Increments for the various histogram bins being loaded can either be initialized through a regular initialization or a load.

If a load is being used, there are further constraints on the base address being used:

- The base address must be either 8 or 16 bits
- The base address must have the same sign as the histogram base address
- The base address must be smaller or equal in size to the histogram base address

A special pseudo-distribution can be used to differentiate between the index load and the weight load, but weight loads support the same distributions that the index load supports. (1PT, NPT, US2, DS2, and NBITS):

```
Vreg = <pexpr>[agen].weight_load(); //Equivalent to .npt()
Vreg = <pexpr>[agen].npt();
Vreg = <pexpr>[agen].onept();
Vreg = <pexpr>[agen].us2();
Vreg = <pexpr>[agen].ds2();
Vreg = <pexpr>[agen].nbits(); // 1 or 8 parallel histograms only
```

1.16.4 Histogram Update

The final instruction in a histogram loop is the update. This instruction performs 3 specific tasks:

- Histogram bin load
- Bin increment
- Histogram bin store

```
Vreg = <pexpr>[agen].hist_update(Vindex, Vweight);
```

Before the load, the index register values can be rounded or saturated by attaching any of the modifiers found in Section 1.12 to the index register's identifier.

After being loaded and incremented, the data values are saturated to the minimum/maximum values of the histogram's type and stored back into the histogram base address.

The results of the bins after incrementing are stored within 'Vreg'.

1.17 Table Lookup

The table lookup loop is a loop unlike the compute loop in that it allows the user to access an address in memory set up as a special table structure, allowing for the copy of data to another memory location. Due to the constrained nature of the table lookup loop on the VCOP itself, the language syntax presented here must be followed strictly to prevent errors.

1.17.1 Loop Structure

To indicate that an upcoming outer **for** loop is to be considered a table lookup loop, you must call one of the following functions just before the **for** loop. Any initializations or moves must come after this identifier or they will be ignored. LOOKUP specifies a table lookup. EXPAND specifies an expanding load.

```
_LOOKUP(K1, K2);
_EXPAND(<pexpr>);
```

The constant K1 is the number of parallel tables to use, and K2 is the number of points per table to load. Valid values for both constants are 1, 2, 4, and 8. In addition, the product of both values cannot be greater than 8. These two functions are mutually exclusive, a single loop cannot have both a table lookup and an expanding load.

The pexpr must be a parameter base address that is to be the source of an expanding load.

The loop structure must come before the outer **for** loop in the presence of a **foreach** loop.

The loop itself may contain only a small set of instructions:

- A minimum of 2 and a maximum of 3 agen declarations
  - All agens must be unique to the instruction that uses it. That means that even if two agens are equal, they must still both be declared and defined as such.
- An index load to determine which data points in the table to load
- A table load to take data from the tables

OR

- An expanding load to conditionally load data from an address
- A store of the data to another base address location

1.17.2 Index Load

The index load loads a vector register with a number of indices that will be used to access various indices of each table. This instruction is as a normal load with the constraint that distribution may only be 1PT, NPT, US2, DS2, or NBITS. The NBITS load mode is only supported for 1 or 8 parallel table lookups.

1.17.3 Table Load/Expand

The load of data can occur in one of two ways. A regular table load uses:

- Index register
- Data type of the table base address
- Number of parallel tables
- Number of data points per table

The syntax is:

V*reg* **=** <*pexpr*>**[agen].lookup(V***index***);**

To determine which values in the base address array to load. As with the histogram load, the index register may be rounded/truncated/saturated by attaching modifiers from Section 1.12 to the index register identifier.

As an alternate to a regular table load, the table lookup loop is the only loop type that can use the expanding load distribution. To leverage this instruction in place of a table load, a load with syntax similar to a collated store is used:

V*reg* **=** <*pexpr*>**[expand(V***index***)];**

This performs a special load into ‘Vreg’. Refer to the VCOP Functional Spec: *Vector load* for more information on the expanding load. Note that this load does not require an address generator.

1.17.4 Data Store

The final step of the table lookup stores the loaded data into an address for later use. This takes the form of a normal store with a choice of two special pseudo-distributions, which specify NPT or SKIP distribution, respectively:

<*pexpr*>**[***agen***].table_npt() = V***reg***;**
<*pexpr*>**[***agen***].table_skip() = V***reg***;**

These are differentiated from regular NPT and SKIP distributions because only a number of values equal to the product of parallel tables and data points are stored out.

Like other stores, the vector register can be rounded/truncated/saturated.

In addition to the rounding and saturation, an expanding load loop can predicate the store with the index register to leave unloaded data points unaltered.

1.18 User Responsibilities

Due to the separation between the kernel compiler and the rest of the toolchain, there are a few limitations on its use. These problems can only be handled by the user through kernel design or by careful oversight of the user.

1.18.1 Syntax Limitations

The syntax presented within this document is often the sole representation of operations in the VCOP Kernel C language. The tool can only create at most a single instruction for every line in the kernel. For example, the following lines are invalid:

- Vdst = base1[Addr] + base2[Addr]

  Must be represented by two loads and then an add, such as:

```
Vsrc1 = base1[Addr];
Vsrc2 = base2[Addr];
Vdst = Vsrc1 + Vsrc2;
```

- Vdst = (V2 * V3) + (V4 * V5)

  Must be represented by two multiplies and an add, or a multiply and a multiply-add

```
Vtmp1 = V2 * V3;
Vtmp2 = V4 * V5;
Vdst = Vtmp1 + Vtmp2;
```

  OR

```
Vdst = V2 * V3;
Vdst += V4 * V5;
```

In addition, the language does not support most operations in the non-innermost loops. The exceptions are address generator expressions, vector initialization, loads, stores, and register-to-register moves.

1.18.2 Circular Buffer Alignment

When declaring an array intended to be used with a circular buffer, it is the user's responsibility to align the variable on the proper boundary. However, the base address used in the kernel function does not have to be on this boundary. For more information, see the VCOP Functional Spec: *Circular buffer addressing support*.

1.19 Common Errors

1.19.1 Kernel Structure

1.19.1.1 For and Foreach Loops

- "loop depth is too high, maximum nest depth for 'for' loops is 4"

  The VCOP processor only supports up to 4 nested for loops.

- "cannot have more than one nested loop at the same depth"

  The tool only supports consecutive nested for loops. If this functionality is required, split the loops into two.

- "loop tripcount cannot exceed 65535”

  A single loop’s tripcount is greater than the 16-bit limit in the hardware

- "vcop instructions cannot directly precede a foreach loop"

  There are one or more instructions that reside inside a ‘foreach’ loop that precede the first nested ‘for’ loop.
- "foreach maximum iterations <K> is not constant"

  To allow for a correctly sized parameter block, the tool needs to know exactly how many times the loop might be run.
- "foreach loop at illegal depth"

  A ‘foreach’ loop exists inside of a ‘for’ loop.

1.19.1.2 Hardware Limits

- "too many arithemetic operations, maximum is 40"

  Split the loop into two, if more operations are required.
- "too many loads, maximum is 8"

  Look into the deinterleave distribution, if the values are of 8 or 16 bits and the kernel requires more than 8 loads.
- "too many stores, maximum is 8"
- "too many init statements, maximum is 16"
- "too many address generators (Max 8), cannot allocate <Agen>"
- "trying to initialize/load <K> vregs, maximum is 16"

  The VCOP does not have support for register spilling. This allows for early detection of failed register allocation before the compiler runs.
- "loop uses too many parameter registers (<K>); rewrite to use fewer"

  The maximum number of parameter registers per loop is 64. Each unqiue load and store pointer uses 2, each unique initialization uses 1 or 2, each agen can use up to 4, and rounding/saturation may use up to 5.

1.19.1.3 VCOP Execution

- "operations must be inside the innermost loop”

  Only loads, stores, agen statements, and conditional moves can be found outside of the innermost for loop.
- "illegal store outside of loop structure"

  There is no support for a store that is outside of 4 nested for loops.
- "vector register <Vreg> cannot be defined after a store on line <K>"

  Due to the way the hardware schedules its instructions, all operations will be run before stores. Therefore, this operation should be moved before any stores, if possible.
- "vector register <Vreg> cannot be defined after operation on line <K>"

  Due to the way the hardware schedules its instructions, register initializations always run before loads and operations. Therefore, this operation should be moved before all stores and operations.
- "vector register <Vreg> is dead here, did you mean <Vreg>?"

  After a register aliasing instruction, (see Section 1.10.1) the source register ids are considered dead until the end of the outermost loop of the current nested structure.
- “allocated register <Vreg> must be left unallocated or allocated to <Vreg>”

  Due to the constraints of certain instructions, the manual register allocation is considered illegal and must be changed.

1.19.1.4 Histogram/Table Lookup

- "address generator <id> must be unique to this instruction"

  The address generators requires for table lookup and histogram loops are constrained and must be different for each instruction, even if their values are different.

- "data register <Vreg> must be loaded before being used"

  The index/data register used in table loads, expanding loads, and histogram updates must be loaded before it may be used.
- "loads inside histogram/table lookup loops must be one of: .npt(), .onept(), .us2(), .ds2()"

  The listed distributions are the only accepted distributions for an index load inside of a table lookup or histogram loop.
- "histogram instructions are not allowed in the this loop"

  An instruction only allowed in a histogram loop is being used inside of a compute or table lookup loop.
- "histogram loops do not support this instruction"

  The only instructions a histogram loop supports are loads, address generator calculations, and the histogram update instruction.
- "number of histograms must be a constant value of 1, 2, 4, or 8"

  Only 1, 2, 4, or 8 parallel histograms are currently supported. These values must be presented as constants inside the kernel.
- "histogram weight register ${1} must be loaded or directly initialized"

  The weight register used in histogram updates must be loaded or initialized before it is used.
- "weight pointer type must be 8 or 16 bits and be equal to or smaller than the histogram pointer type"

  The base address of the weight values cannot be 32 bits, nor can it be larger than the size of the histogram base address.
- "weight pointer sign must match the histogram pointer sign"

  The base address of the weight values must have the same sign as the histogram base address.
- "table lookup instructions are not allowed in the this loop"

  An instruction only allowed in a table lookup loop is being used inside of a compute or histogram loop.
- "table lookup loops do not support this instruction"

  The only instructions a table lookup loop supports are loads, a single store, address generator calculation, and the table load instruction.
- "number of tables and points must be constant values of 1, 2, 4, or 8"

  Only 1, 2, 4, or 8 parallel tables or loaded data points per table are currently supported. These values must be presented as constants inside the kernel.
- "the product of table count and points must not exceed SIMD width (8)"

  The product of parallel table count and data points is equivalent to how many values will be loaded into the vector register, so therefore the number cannot exceed the number of lanes in the register.
- "table store predicate register ${1} must be the loaded index register"

  The only valid predicate register for a store inside of a table lookup is the index register.
- "vector register ${1} must be loaded through a table or expanding load"

  The source register for a store inside of a table lookup loop must have been the destination register of a table load instruction or an expanding load.
- "table lookup store distribution must be .table_npt() or .table_skip()"

  These are the only two distributions allowed in a lookup table or expanding load loop. They store a number of values equal to the number of tables multiplied by the number of points per table to the address.

1.19.2 Operations

- "method doesn't exist"

  Recheck the method name used or refer to the documentation.
- "identical vectors used in two destination operation"

  It is illegal to use a single register as both destination registers.
- "<opcode>: destination register must be the third source register"

Operations with three source registers use the third source register as the destination register.

- "minf/maxf require first in output pair be the same as second source"

  In the minf/maxf functions, first destination register must be the same register as the second source register in the pair.

1.19.2.1 Operation Round Parameters

- "cannot set rounding options on <opcode>"

  Only VMPY, VMADD, and VMSUB support operation rounding.
- "operation rounding parameter <pexpr> is non-constant"

  All operation rounding parameters must be constant.
- "<pexpr> is not a valid round/trunc value, must be one of {8, 15, 16}"

  The hardware only supports 8, 15, and 16 bit rounding/truncating on operations
- "<pexpr> is not a valid left shift value, must be 1”

  The hardware only supports a left shit of 1 bit on operations

1.19.2.2 Early Exit

- "label does not match <id> seen at line <K>"

  The identifiers used for the ‘goto’ and the label must match.
- "label must be outside either a foreach or outermost for loop"

  Outside of a ‘foreach’ and outside of the outermost ‘for’ loop are the only supported places to exit.
- "early exit must be inside either the innermost or foreach loops"

  The only two supported locations to check for an early exit condition is at the end of the inner loop operations or after the inner VLOOP of a repeat loop.
- "early exit only supports checking of lane 0 of vector registers"

  Use <Vreg>[0] inside the if statement.
- "loop can only have one exit instruction, first one is on line <K>"

  This is a hardware constraint.
- "value on right hand side of conditional must be constant 0"

  Early exit can only check for nonzero values in the 0^{th} lane of the register.
- "no instruction may occur after an early exit"

  Early exit jumps to the label after final instruction in the inner loop, so this constraint allows for behavior to be equivalent to c++.
- "exit label has no matching goto"

  A label is required to define where the early exit will jump to upon activation.

1.19.2.3 Conditional Move

- "<string> is not a valid loop variable"

  The supplied identifier is not a loop variable
- "moving vector register <VREG> into itself"

  Moving a register into itself is meaningless code and will be removed before link.
- "cannot mix comparisons to tripcounts and zero in conditional moves"

  The hardware only supports conditions involving all first-iteration checks, or all last-iteration checks; mixed conditions are illegal.
- "expression does not match tripcount of loop using lcv <LCV>"

  The tripcount found in the last(<lcv>,<tripcount>) can differ from the original value given in the for loop only by the amount of parentheses surrounding it.
- "loop variable can only be directly compared to 0, did you mean last(<LCV>,<PEXPR>) or (<LCV> == 0)?"

The only direct comparison allowed in conditional moves is to constant 0

- "conditional move constraints are not supported by the hardware"

  When using the ‘if’ syntax for conditional moves, the loop control variables to be checked must follow the order of innermost to outermost without skipping.

1.19.3 Load/Store

- "illegal load/store address expression"

  Either a base address for a store/load doesn’t contain a base address, or there are more than one instances of a base address.
- "address generator <id> is uninitialized"

  There is no sum-of-products or 0 value assigned to an address generator.
- “<id> and <id> are targets of loads, cannot be register pair”

  The registers that a load can use differ from those that can be part of a pair in an interleave store. Transfer the second source register into another using a register-to-register move to rectify this issue.
- "distribution option doesn't exist"

  Recheck and make sure that the distribution option specified is supported.
- "distribution mode does not support word (32-bit) type"

  The DS2, Skip, Interleave, and Deinterleave load and store distributions do not support 32-bit values.
- "unrecognized circular buffer size"

  Circular buffer size must be designated by CIRC_<X>K where <X> is 1, 2, 4, 8, 16, or 32.
- "interleaved store cannot be predicated"

  The interleave distribution does not have support for predication. Manually interleave the values in the registers and then perform two predicated stores.
- "only one register per kernel can predicate the scatter store"

  There is only one vector register that can b
- "collated store is already predicated by <id>"

  The collate distribution acts as a lane-to-lane predicated store and cannot be predicated normally.
- "collated stores cannot have a second distribution option"

  A collated store is a distribution and cannot override or combine with another distribution.
- "too many unique custom distributions"

  The hardware only supports 3 distinct custom distribution arguments.
- ".dist() arguments must be either 8 constants or one pointer to such"

  The custom distribution requires user arguments.
- "multi-argument custom requires all values to be constants"

  The indices for the custom distribution must be constant values.
- "single-argument custom requires a pointer to an array of integers"

  The argument for a custom distribution of this form must be an array or pointer to an array.
- “ instructions can only be saturated once"

  Only one saturation parameter can be specified per store
- “ instructions can only be rounded/truncated once"

  Only one round/truncation can be specified per store
- "manually allocated register <Vreg> must be <Vreg> due to load/store"

  Because loads work on adjacent registers, the manual allocation as defined is invalid and must be redefined as given by this error.
- "<Vreg> will be allocated to <Vreg>, make sure <Vreg> is not used elsewhere"

  Because of a manually allocated register in part of the pair, this vector register can only be allocated to a single register. This is a warning because using the required machine register anywhere but in place

of this identifier causes undefined behavior.

- “register <Vreg> is invalid, must be <odd/even>"

  The base register for a load must be an even MR (V0, 2.., 14) whereas the secondary register in an interleave/deinterleave must be odd (V1, 3.., 15)
- “weight load distribution is only allowed in histogram loop”

  The weight load distribution is a pseudo-distribution in the histogram loop structure and is not defined in any other loop type.
- “load must be placed outside the for loop structure since <AGEN> is 0”

  Because the agen associated with this load is 0, it will only be loaded once. Therefore, it needs to be placed outside of the ‘for’ loop to better represent the execution.
- “load must be placed at the loop depth defined by the variable <LCV>”

  The load is too deeply nested in the ‘for’ loop structure and must be moved to the loop level associated with the deepest loop control variable used by the address generator.
- “register of conditional load aliased; performance may suffer unless the --vcop_unconditional_loads option is supplied"

  A conditional load, or a load whose address generator is not known to have non-zero increments, loads to a vector register that cannot be reused within the loop because the value must be preserved. This precludes the register from being aliased to another and instead causes an extra VCOP statement to be generated, which may impact performance. The --vcop_unconditional_loads option instead assumes that inner loop loads with unknown increments are unconditional, and thus can alias its destination register with another.
- “nbits() load requires histogram/table count to be 1 or 8"

  The NBITS distribution is only supported when _HISTOGRAM or _LOOKUP defines 1 or 8 parallel objects
- “Negative SYMM/ASYMM bound is undefined, defaulting to 4PARAM"

  The saturation matches the SYMM/ASYMM requirements, but uses a negative bound value. This is undefined behavior under SYMM/ASYMM, but valid under 4PARAM.

1.19.4 Miscellaneous

- "illegal agen expression"

  A product in the sum does not contain a valid loop control variable
- "vreg pair expected"

  The deinterleave and interleave distributions require a pair of vector registers.
- "unexpected vreg pair"

  All load/store distributions require only a single vector register as the destination or source.
- "vector register <id> cannot be the main in a pair with <id>"

  Either the register is being paired with itself, or the main register was the second source register in a previous load.
- "<id> is used before it is initialized; may cause unexpected behavior"

  A vector register is being used in an operation or store before an initialization or load, or before the register is used as the destination of an operation.
- "redeclaration of '<type> <id>' declared on line <K>."

  An address generator or vector register is declared in the same scope as another vector register or address generator of the same name, or an address generator is declared with the same name as a vector register, or vice versa.
- "declaration of internal vector register 'V<0-15>'"

  The internal vector registers which map directly to their corresponding machine registers do not need to be declared.
- "command line only supports 1 source and 1 destination file"

  Currently the tool only supports input including the source kernel file and the destination c file. If you

see this during compilation, report and document this.

- "Illegal expression: value expected to be <K>, but is <pexpr>"
  In 'for' loops, the LCV must start at 0 and must be incremented by 1 per iteration.
- "internal error in parser interface"
  This is a fatal internal error; report and document this.

2 VCOP Kernel C Compiler Usage

2.1 Environment

After installing the tools, make sure that the environment variable ARP32_C_DIR is assigned to the path leading to the ARP32's library files. There is a special file in the directory 'vcop' in the library path called 'vcop.h' that defines many of the required types and functions for proper compilation. This file must be included in any non-kernel file that uses VCOP specific code.

2.2 Output of Kernel File

After the kernel file is translated, three functions will be available for use in any driver c file. These functions must be declared as prototypes in the driver c file to be used properly.

*kernel_func_name***(***full_argument_list***);**

This function syntax provides an easy way to just run a kernel. It takes care of ARP32 buffer switching, kernel initialization, and running:

*kernel_func_name*_**init(***full_argument_list***, unsigned short* pblock);**
*kernel_func_name*_**vloops(pblock);**
*kernel_func_name*_**param_count();**

These two functions, used together, allow for full customization of a kernel's runtime. The init function sets up a user-defined location in memory indicated by 'pblock' so that it can be used as the parameter register set for the kernel function. In addition, the function returns the exact number of parameter registers taken up by the kernel function. The vloops function takes 'pblock' and simply runs the kernel instructions. The param_count function simply returns the number of parameter registers that are used by the designated kernel function. This is useful when the user wants the register count but does not want to spend time reinitializing.

For example, say there exists a kernel file 'kernel.k' with the following kernel functions:

Example 3. File kernel.k

```
void test_kernel_1(__vptr_uint32 in, __vptr_uint32 out, short num_rows)
{ ... }

void test_kernel_2(__vptr_uint32 in, __vptr_uint32 out, short num_rows)
{ ... }
```

Then in the C file, we may simply call functions like one would call it in C:

Example 4. C File Call Functions

```
#include "vcop.h"

short num_rows=2;

#DATA_SECTION (input, "Adata")
```

Example 4. C File Call Functions (continued)

```
unsigned int input[2*num_rows] = {0, 1, 2, 3, 4, 5, 6, 7, 7, 6, 5, 4, 3, 2, 1, 0 };
#DATA_SECTION (output1, "data")
unsigned int output1[2*num_rows] = {0};
#DATA_SECTION (output2, "Cdata")
unsigned int output2[2*num_rows] = {0};

#ifdef VCOP_HOST_EMULATION
   #include "kernel.k"
#else
   void test_kernel_1(__vptr_uint32 in, __vptr_uint32 out, short num_rows);
   void test_kernel_2(__vptr_uint32 in, __vptr_uint32 out, short num_rows);
#endif


int main()
{
   test_kernel_1(input, output1, num_rows);
   test_kernel_2(output1, output2, num_rows);
}
```

However, for more control, you can perform all the initializations at once, and then call the actual VCOP code sequentially. Host emulation is not supported with this execution method.

Example 5. Performing All Initializations

```
#include "vcop.h"

#define MAX_PARAMS 200
short num_rows=2;

#DATA_SECTION (input, "Adata")
unsigned int input[2*num_rows] = {0, 1, 2, 3, 4, 5, 6, 7, 7, 6, 5, 4, 3, 2, 1, 0 };
#DATA_SECTION (output1, "Cdata")
unsigned int output1[2*num_rows] = {0};
#DATA_SECTION (output2, "Cdata")
unsigned int output2[2*num_rows] = {0};

unsigned short parameter_block[MAX_PARAMS];

//Host emulation does not work with custom execution
void test_kernel_1_init(__vptr_uint32 in, __vptr_uint32 out, short num_rows,
                        unsigned short* pblock);
void test_kernel_2_init(__vptr_uint32 in, __vptr_uint32 out, short num_rows,
                        unsigned short* pblock);
void test_kernel_1_vloops(unsigned short* pblock);
void test_kernel_2_vloops(unsigned short* pblock);

int main()
{
   int num_params_acc = 0;
   int first_kernel_offset = 0;
   int second_kernel_offset = 0;

   ... //Switch buffers to VCOP
   num_params_acc += test_kernel_1_init(input, output1, num_rows,
                                &parameter_block[first_kernel_offset]);
   second_kernel_offset = num_params_acc;
   num_params_acc += test_kernel_2_init(output1, output2, num_rows,
                                &parameter_block[second_kernel_offset]);
   test_kernel_1_vloops(&parameter_block[first_kernel_offset]);
   test_kernel_2_vloops(&parameter_block[second_kernel_offset]);
```

Example 5. Performing All Initializations (continued)

```
    ... //Switch buffers back to system and check/use output
}
```

2.3 Compilation

2.3.1 Host Emulation

The vcop.h file contains a model by which a kernel may be compiled for a host machine without the presence of a VCOP. To activate this mode, the host compiler must be called with the following options in addition to the regular options:

-D VCOP_HOST_EMULATION
-I *arp32_library_directory*/**vcop**

The -D option activates the host model in vcop.h. The -I (uppercase i) option includes the vcop.h file.

In addition, the source files cannot have any code that when compiled, contains ARP32-specific code.

2.3.2 Target

To compile a program for VCOP, the ARP32 compiler must be called with silicon version 210. That is to say, an example shell call for the source files testbench.c and kernel.k would look like:

```
cl-arp32 -v210 --abi=eabi testbench.c kernel.k -z -l rtsarp32_v200.lib -l link.vcop.cmd
```

The ARP32 libraries, as well as 'vcop.h', must be found by the compiler either through ARP32_C_DIR or via the command line. Here is the syntax:

-I *arp32_library_directory*
-I *arp32_library_directory*/**vcop**

The first -I option specifies the location of the ARP32 libraries. The second -I option specifies the location of the vcop.h file.