6.49 Using vector instructions through built-in functions

On some targets, the instruction set contains SIMD vector instructions that operate on multiple values contained in one large register at the same time. For example, on the i386 the MMX, 3DNow! and SSE extensions can be used this way.

The first step in using these extensions is to provide the necessary data types. This should be done using an appropriate `typedef`:

```
typedef int v4si __attribute__ ((vector_size (16)));
```

The `int` type specifies the base type, while the attribute specifies the vector size for the variable, measured in bytes. For example, the declaration above causes the compiler to set the mode for the `v4si` type to be 16 bytes wide and divided into `int` sized units. For a 32-bit `int` this means a vector of 4 units of 4 bytes, and the corresponding mode of `foo` will be V4SI.

The `vector_size` attribute is only applicable to integral and float scalars, although arrays, pointers, and function return values are allowed in conjunction with this construct.

All the basic integer types can be used as base types, both as signed and as unsigned: `char`, `short`, `int`, `long`, `long long`. In addition, `float` and `double` can be used to build floating-point vector types.

Specifying a combination that is not valid for the current architecture will cause GCC to synthesize the instructions using a narrower mode. For example, if you specify a variable of type V4SI and your architecture does not allow for this specific SIMD type, GCC will produce code that uses 4 SIs.

The types defined in this manner can be used with a subset of normal C operations. Currently, GCC will allow using the following operators on these types: +, -, *, /, unary minus, ^, |, &, ~, %.

The operations behave like C++ valarrays. Addition is defined as the addition of the corresponding elements of the operands. For example, in the code below, each of the 4 elements in *a* will be added to the corresponding 4 elements in *b* and the resulting vector will be stored in *c*.

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a, b, c;

c = a + b;
```

Subtraction, multiplication, division, and the logical operations operate in a similar manner. Likewise, the result of using the unary minus or complement operators on a vector type is a vector whose elements are the negative or complemented values of the corresponding elements in the operand.

In C it is possible to use shifting operators <<, >> on integer-type vectors. The operation is defined as following: {a0, a1, ..., an} >> {b0, b1, ..., bn} == {a0 >> b0, a1 >> b1, ..., an >> bn}. Vector operands must have the same number of elements. Additionally second operands can be a scalar integer in which case the scalar is converted to the type used by the vector operand (with possible truncation) and each element of this new vector is the scalar's value. Consider the following code.

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a, b;

b = a >> 1;     /* b = a >> {1,1,1,1}; */
```

In C vectors can be subscripted as if the vector were an array with the same number of elements and base type. Out of bound accesses invoke undefined behavior at runtime. Warnings for out of bound accesses for vector subscription can be enabled with `-Warray-bounds`.

You can declare variables and use them in function calls and returns, as well as in assignments and some casts. You can specify a vector type as a return type for a function. Vector types can also be used as function arguments. It is possible to cast from one vector type to another, provided they are of the same size (in fact, you can also cast vectors to and from other datatypes of the same size).

You cannot operate between vectors of different lengths or different signedness without a cast.

A port that supports hardware vector operations, usually provides a set of built-in functions that can be used to operate on vectors. For example, a function to add two vectors and multiply the result by a third could look like this:

```
v4si f (v4si a, v4si b, v4si c)
{
  v4si tmp = __builtin_addv4si (a, b);
  return __builtin_mulv4si (tmp, c);
}
```