DSP-C Specification

July 2001 Revision

Lance Hammond
lance@stanford.edu

Smart Memories Group
Stanford University

1. DSP-C: An Introduction

DSP-C is an extension to C that tries to solve some of C's most annoying problems, at least when one is attempting to use it to write highly optimal multimedia code. It proposes a relatively small group of low-level enhancements to the C language that should simplify the problems encountered by both the programmer *and* the compiler when attempting to deal with this code.

One of the benefits of C is that it has always been a sort of "high level assembly language." It *is* a high level language, but it is very simple and close enough "to the metal" that experienced assembly language programmers can guess fairly well how code written using C will be converted into machine instructions, much more so than with many other high level languages. It also offers direct access through operators to most of the more exotic machine instructions, such as bitwise logic and a variety of integer lengths, that are present on virtually all general purpose CPUs today but are unsupported by many other languages. There are a couple of drawbacks to this approach now, however. The first is that while making a simple compiler for C is fairly painless, making an *optimizing* compiler for C is a major headache. In particular, its pointers are so powerful and versatile that there is almost no way for a compiler to statically analyze how they might be used. Since DSP algorithms rely on vectorization and parallelization to provide good speed, a poorly optimizing compiler can be fatal. The second problem is that its selection of operators dates from the days of the DEC PDP series in the early 1970's. Unfortunately, ISAs have enlarged with specialized multimedia instructions and dedicated DSP processors since then, but C has not kept up. Since these instructions are very important in multimedia operations, we need to update it with some of the most important new operations.

There have been a multitude of libraries (mostly) and extensions to C (a few) proposed and sometimes implemented over the years in an effort to solve these problems, but the majority of them either be written by language designers — who tend to overlook the incredible performance differences that a few "nice" high-level constructs can incur compared with tight but hacked code — or by DSP vendors — who target their extensions to the quirks in their own architecture. The result has been a maddening babble of incompatible and often even awkward solutions to some fairly simple problems.

DSP-C cuts beneath the clutter of libraries and extensions by proposing a pair of low-level core extensions — DSP integers and CAAs — that solve many of C's multimedia problems by themselves in a way that no library could do so cleanly. The basic philosophy of the DSP-C extensions is to make the common case simple and easy to program. While some of the extensions provided can result in some pretty ugly-looking code when they are used, the most

common (and important) elements should result in fairly clean and uncluttered code. Much of this document describes a family of new operators to add to C and a variety of enhancements or options that can be used with DSP integers and CAAs, but most of these details follow the 90/10 rule: 90% of the document is devoted to the 10% of time spent programming oddball corner cases that seem to crop up occasionally. Frankly, even if the only thing one does with DSP-C is to use the two new data types, it will have gone a long way towards making multimedia software easier to program and compile. Continuing this philosophy, LCSL (chapter 5) is devoted to moving program complexity off into library routines — and out of programs — by making the libraries "smarter."

1.1. Some DSP-C "Standards"

In order to make DSP-C easier to learn and read, its additions try to follow a few standard guidelines that may or may not be obvious. Some of the most important ones are listed here, to make code in the later chapters that more intelligible. As we attempt to write parsers for these language semantics, we may have to adjust the language slightly to make it easily machine-readable as well as human-readable.

- **(()) and [[]]:** DSP integers and CAAs, the most important parts of DSP-C, try to use double sets of parentheses and brackets as their "standard" syntactical structure. They are similar to each other, which helps make it obvious that both are a part of DSP-C. Also, the brackets are similar to the existing [] C operator, which they largely replace.
- **dsp_ and lcsl_:** Terms which will probably have to become keywords have generally had this addition stuck onto the front. Hopefully, this will prevent them from colliding with too many names in existing code, but we may have to come up with another prefix. On the other hand, we may be able to cut prefixes if name conflicts prove to be less of a problem than I expect they might be. This is especially the case with the lcsl_ prefix.
- **Structured Stuff:** More complex aspects of DSP-C's data types have generally been confined to braces or brackets so that they won't clutter up the namespace with a lot of extra keywords. Most things that fit into these categories either won't need to be keywords at all or can be "modal" keywords that are only enabled where they might legally be encountered.
- **: and ,:** In general, commas are used to separate dimensions and colons to separate sub-ports in CAA ranges. There are lots of other ways these two bits of punctuation are used, but these two uses are generally been punctuation-coded because they occur together in the midst of most cluttered declarations that may occur on a fairly regular basis.
- **Optimization Options:** Many places in the more complicated areas of DSP-C offer programmers locations to give the compiler "hints" on how to compile code. While the language may need to be modified to make it more compiler-friendly in some ways, many small compiler problems can probably be solved with a hint in the right place.

Along with these "standard" features, any DSP-C source code should include the following comment at the top of the file, before any non-comment code is written:

```
/*!* DSP-C Source V.2000.07.00 DEBUG=ON *!*/
```

The version field in this comment will allow programmers to mark their code with an appropriate version identifier, making it easier to compile in the future as DSP-C changes. The debugging indicator signals to DSP-C whether or not to enable range checking and other runtime debug capabilities for CAAs. It may be overriden by an existing compiler debug flag, of course.

1.2. DSP-C Development Effort

DSP-C will follow a staged introduction over the next year or two, following the approximate pattern described in Figure 1. A compiler to "downgrade" DSP-C to ANSI C will allow our applications programmers to start using DSP-C in a reasonably short timeframe, although without any performance benefit. Successive stages of development replace portions of the source-to-source code translator with progressively more aggressive code generators. As time goes by, more compiler technology students will be needed to accommodate this transition.

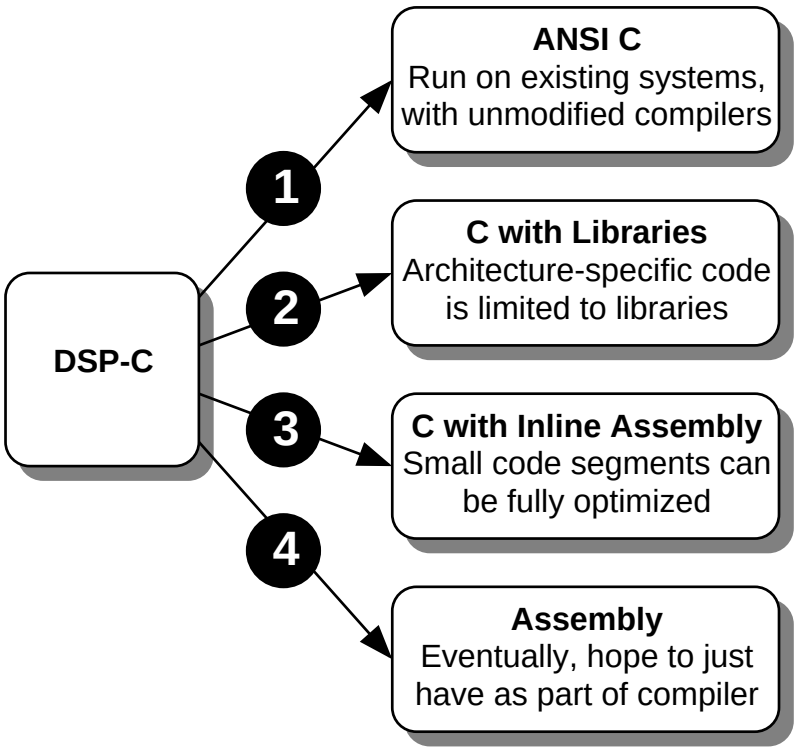

Figure 1: A diagram illustrating the increasing complexity of code generators for DSP-C over time, from 1-4.

1.3. Feedback

While eventually this document will probably grow into more of a standard or specification for what we are doing, at this point it is primarily a concrete proposal that will hopefully get everyone onto the same wavelength and give us a springboard for further advancement. As a result, feedback on the document is of utmost importance. Anything from the small ("that keyword is illogical") to the large ("throw out chapter 4 and replace it with stream C") are viable and potentially valuable suggestions. Comments on the document itself are also welcome. Feel free to send an email when you encounter a part that is unclear, under-specified, or under-explained. With a document this long and complex, it is not always possible to determine what different readers will find most complex or straightforward. Also, there are some particular bits of information that I would like to hear from various sub-groups.

- **Hardware:** Read this document with a particular eye towards translation of language structures to hardware structures. This will probably be most important in chapters 2 and 3, where more low-level features are described.
- **Compiler:** Look more for places where the language could be improved to help facilitate the development of a flexible yet highly optimizing compiler. Most of the key portions of the document applicable to this are in chapters 4 and 5.
- **Applications:** Try to code up a few algorithms with the code constructs described in this document and see if they help you write your code more easily and efficiently. If not, think about how you would like the structures given tweaked or changed completely. Opinions on how you like the syntax provided is also helpful.

It is impossible to be all things to all people, but hopefully the portions of the proposal that cause the most difficulty for one group of people or another can be ironed out and DSP-C can become a language extension that is at least reasonable for many to use.

1.4. Document Layout

The remainder of this document is divided into four main chapters, each addressing a major aspect of DSP-C's additions to the C language.

Chapter 2 discusses the data types that DSP-C adds. The primary addition is the DSP integer, which has fully specified bit width and precision information included. This is very important for DSP and multimedia applications because they often use a wide variety of specific integer data types. With normal C, the programmer is at the mercy of the rather vague integer data types that C provides, such as the all-popular `int` type — that is *usually* 32 bits, but can vary. While libraries can help nail down C's data types more accurately, it is difficult or impossible to specify

odd-size word lengths or levels of precision with libraries. DSP integers handle all of these situations, and give the compiler the information it needs to do automatic conversion and alignment of the different types. The chapter also discusses a proposed complex number scheme that can be added to C and complements the DSP integer well, since many frequency domain signal processing applications use complex numbers at some stages.

The third chapter discusses a wide variety of low — and not-so-low — operators that should be added to C to allow the specification of common low-level multimedia operations. These operations can often be translated into single instructions or small sets of instructions with powerful DSP or multimedia architectures, but they cannot be specified in any simple manner with C. Libraries can help, by providing inline assembly versions of the special instructions, but these are often very architecture specific. The presence of inline assembly may also impose restrictions on the abilities of some compilers to produce optimized code. DSP-C's selection of operators should make some of the most common operators fundamental within the C language, and open to full optimization. Also, a possible architecture for making DSP-C's operator set extensible is provided.

The heart of the document is in chapter 4, which discusses the additions to facilitate vectorization and parallelization of code. After a discussion of the large-scale compiler techniques we will need to combine together in order to get a reasonably fast compiler, the chapter dives into a discussion of DSP-C's most significant addition to C: the constrained access array, or CAA. This is a new type of array added to C that prevents all use of normal C pointers within it. This alone is a great help to optimizing compilers, but DSP-C also adds special syntax for operating on large ranges of CAAs using explicitly parallel operations that should help make the compiler's job more straightforward while reducing some of the extraneous code in C incurred by massive loop nests.

Finally, the last chapter discusses Library Control Scripting Language (LCSL), an addition to the interface C libraries that can make them much more powerful and flexible. With the multitude of data types that DSP-C can provide (thanks to fully flexible bit widths in DSP integers) and also due to the fact that DSP programmers insist on choosing library routines that have been optimized to use the compiler controlled memory found on DSP chips to their best advantage, DSP libraries can easily become beasts with a multitude of architecture-specific variants on every library call. LCSL attempts to define an interface language for libraries that lets the library automatically choose the best routine or group of routines from within itself. A single call to a simple routine name like `FFT` can work for a variety of different inputs and on a variety of different architectures, because LCSL will select the proper library call for the data and memory situation given by each call.

2. DSP-C Data Types

DSP-C introduces two key new data types. One is completely new, while the other is based on an existing semi-standard extension to C. In addition, a third possible extension is proposed. Together, these extensions allow the variety of fundamental data types used in DSP operations to also be fundamental data types in C. Having the fundamental data types at this level allows the building of code generators that can properly generate instructions necessary for highly optimized DSP kernel code.

2.1. The DSP Integer

C contains only a limited supply of fundamental data types — integers of 8, 16, and 32/64 bits and two sizes of floating point numbers. The integer types have an additional problem in that the code generated for a given description (`short int`, `int`, or `long`) is actually implementation-dependent. The same code running on different processors may therefore be compiled with integers of different lengths. In the past, libraries have been developed to address the variable-length problem, translating fixed-length data types to the C type descriptions on particular architectures. For traditional computing applications, fairly simple libraries solved these problems very well.

Unfortunately, signal processing applications often require a breathtaking variety of very specific integer (and occasionally floating point) types. Bit widths range from 4–8 for some data, such as video color values or telephone-quality audio, up to 50–80 bits for extended-precision accumulators used as temporaries in some stages of algorithms to reduce rounding errors. Because various stages of DSP algorithms must carefully trade off storage space and communication bandwidth (favoring small sizes) and mathematical precision (favoring larger sizes), any particular algorithm will often include several different sizes of integers and then intermix them extensively. It would be extremely helpful to be able to express these varying lengths simply and directly in code, and let a compiler worry about the varying layout and alignment of these types in registers (especially) and memory (to a lesser extent).

Adding more complexity to this mess is the fact that “integers” in DSP applications are often not treated like “integers” in the traditional sense. Instead, most DSP code uses integer variables to represent fractional numbers, trading off range to increase the precision of the values represented. Table 1 illustrates the tradeoffs that can be made with a 16-bit integer, for example. Like varying data sizes, varying precision levels are often used in the same code as the programmer trades off minimal data size versus mathematical precision. In real code, additional

bits below the binary point are often maintained within temporary values to minimize rounding error, while additional bits above the binary point are sometimes added to temporarily increase range. The latter adjustment can be critical with many DSP algorithms, because saturation or overflow may occur at different times if the range is not properly maintained throughout the course of the algorithm, distorting the results. When varying data size and precision are combined, keeping data in proper alignment over the course of multiple calculations can be a nightmare for programmers. For example, Figure 2 shows how the product of a multiplication may need to be aligned in different ways when the otherwise identical inputs have different levels of precision. Again, it would be extremely helpful to simply have a compiler handle low-level issues like alignment automatically for the programmer.

Type	Range	Precision
Normal 16-bit integer	-32768 to +32767	1.00000
8 bits to right of binary point	-128 to + 127.99609	0.00391
15 bits to right of binary point	-1 to +0.99997	0.00003

Table 1: A few possible configurations of a 16-bit integer with varying binary point locations.

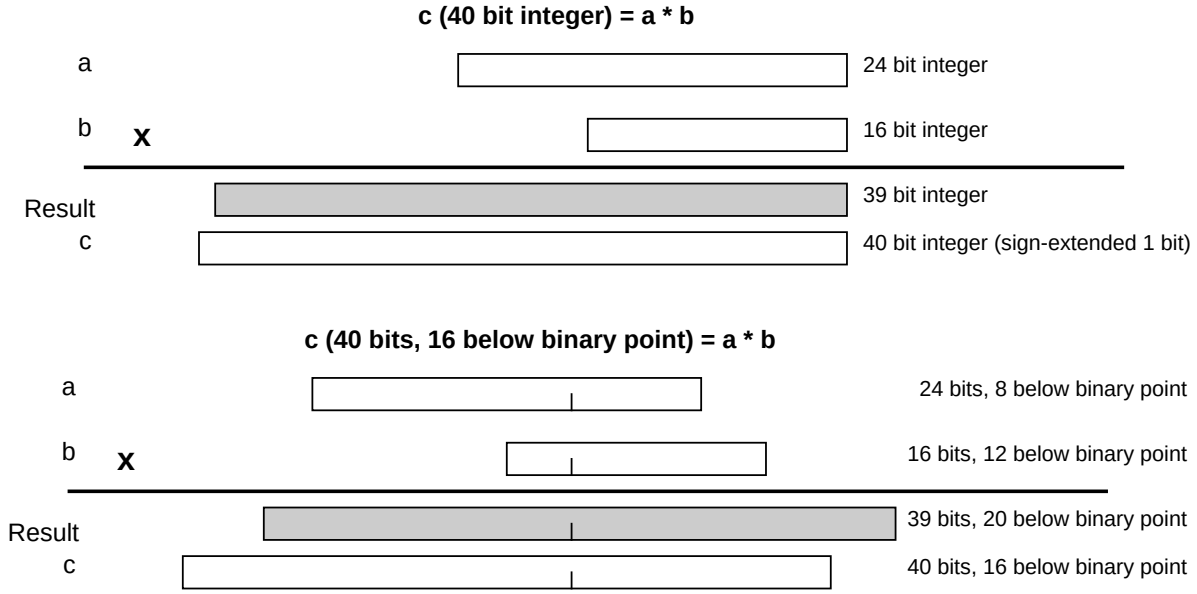

Figure 2: Bit alignment of varying input types to a simple multiply operation.

While libraries have been developed in an effort to address these needs (such as VSIPL or various manufacturer-generated ones), it is hard for a simple yet portable C library to offer a full spectrum of the necessary types without becoming very large and unwieldy. Even worse, handling the alignment issues of variable-precision integers is virtually impossible for a library, because at best a low-level C library can really only translate data types without adding

unwanted complexity. The fundamental C data types have no concept of varying precision, so there is no way to truly express this concept in C without making complex and unwieldy data types (such as an integer with another integer paired with it to specify the precision, in a pseudo-FP manner). As a result, the most common technique used commercially is for programmers to target architecture-specific libraries that define the bit widths natively available on a particular DSP chip or family of processors. Precision factors are still handled by hand, for the most part, with manual shifting of operands where necessary. While this keeps the libraries reasonable, it also tends to make the code generated confusing and non-portable. With the DSP Integer, we believe that we can solve all of these problems in a straightforward, portable way.

2.1.1. Syntax & Usage

Unlike complicated libraries, we propose simply adding a single construct to the C language to handle any possible DSP integer. It is simply an integer with an additional bit width and precision specification. Here is a proposed syntax:

```
<<unsigned>> int identifier((<<exactly>> bit width <<, binary point
      location>>));
```

In practice, definitions using this format are very simple. The following specifies a 24-bit 2's complement signed integer with 8 bits below the binary point (1 "sign" bit, 15 integer bits, 8 fractional bits). Only positive values of *bit width* and *binary point location* are legal.

```
int value((24, 8));
```

If the `unsigned` keyword is added, it works just like it does in ANSI C. The same integer is defined, but in an unsigned form. Hence, this definition is the same as the previous one, but without the sign bit (16 integer bits, 8 fractional bits).

```
unsigned int value((24, 8));
```

If fractional bits aren't needed, the binary point location can be omitted to just make a precision integer, as in the following example, which defines a 24-bit signed integer.

```
int value((24));
```

The `exactly` form requests that the processor round off values assigned to this variable between every calculation, so that it never is temporarily assigned extra precision. This is discussed more in the next section. The following is therefore identical to the first definition,

above, except that it is guaranteed to always be *only* 24 bits long. In general, this form should only be used for the most mathematically demanding algorithms, because code generated it will usually run more slowly due to the additional rounding required.

```
int value((exactly 24, 8));
```

Multiple definitions on one line are also possible, just as in ANSI C. The precision of each variable must be specified with its identifier, much like the * associated with the definition of each pointer.

```
int first((24, 8)), second((16, 4)), third((8));
```

Further common C modifications to definitions, such as storage-class-specifiers, type-qualifiers, pointers, and initializers may be made just as if the DSP integer was a normal integer. Even though it is very simple, this data type goes a long way towards making C much more usable for a wide range of DSP programming.

2.1.2. Translation to Architectural Data Types

While it is possible to specify virtually any type of desired integer with a DSP integer, that doesn't necessarily mean that it will be possible to find native support for all but a small subset of them on any particular architecture. As a result, a large part of the compiler's job when using these integers is to translate from the DSP integer types to the processor's natively supported types.

In general, this may be done through a simple transformation: additional bits of precision are padded onto the low end of the number. This variably extra precision may cause results to vary slightly from one architecture to another, but in general the variation should be extremely low. For critical variables, the `exactly` keyword can be used to prevent any extra precision from being added. Since in practice this is done by adding extra rounding, where necessary, to keep any extra bits zeroed out at all times, it should be done only for the most mathematically critical variables. The upside of this transformation is that the range of the variable is always guaranteed to be constant on any architecture. Thus, overflow and saturation will always occur at essentially the same time on any architecture (except in the rare cases when extra precision happens to affect overflow). As these events can cause a much more dramatic effect on the output than slight variations in the least significant bits, it is essential that they occur in the same way on any implementation of an algorithm.

The compiler may also automatically generate DSP integers by combining multiple architectural types into a single DSP integer type. This will occur most often when code is ported down to smaller chips like 16-bit DSPs, but the system is able to handle integers of any size on any architecture — and 1024-bit integers are not unheard of in cryptography! Long integers are simulated in software so that they act just as if they were a native data type. Having these longer data types automatically synthesized by the compiler when necessary relieves the programmer from worrying about their presence or absence on different architectures.

In addition, the compiler now has enough information to automatically make decisions about issues such as alignment before each calculation and rounding afterwards. Here are a few rules of thumb that about what may be expected from a compiler automatically handling the mixing of several different types of DSP integers.

- The results of calculations are always aligned so that the most significant bit of the DSP integer corresponds with the most significant bit of the architectural type, to maintain the proper overflow and saturation behavior.
- "Extra" precision bits may be eliminated at any time, in part or altogether, when inputs to a mathematical operation are right-shifted in order to maintain proper output alignment.
- On the other hand, "extra" precision bits may also be added at any time when variables are promoted up to larger architectural integers, when these integer formats are necessary to maintain the proper output precision. In general, however, this should have no effect because the added input bits will always be zeroed out.
- Temporary values in mid-expression are always maintained with enough precision to avoid mid-expression rounding, whenever this is possible without requiring additional precision-extension instructions that may slow down the code (an architectural specific option, but a necessary one for achieving the fastest possible code). For example, the additional bits of precision generated by a multiply instruction are propagated through later calculations in an expression, if this is possible on a particular architecture. This rule is necessary because extended-precision accumulation is a common DSP technique, yet fast code is usually even more critical.
- When DSP integers are mixed with normal integers, the normal integers are just treated as special DSP integers of type ((*architecturally-defined bit width*, 0)). When mixed with non-integer standard C types, they follow the standard casting and promotion rules as integers would. Of course, when converting to and from floating point values, fractional values will be maintained intact during transitions as much as possible, an impossible trick with standard integers.

As we develop compilation technology for DSP integers, the rules and techniques for handling the translation to and from architectural data types may require adjustment. The rules and concepts described here, however, should give us a good starting position.

2.1.3. Rounding and Saturation Operators

Sometimes extra bits of precision may not be desirable, as they may cause slight variations in the results on different architectures. However, the solution of making all calculations involving variables force rounding with the exactly keyword may be too extreme. Hence, DSP-C includes some rounding operators for explicitly adding rounding in key locations. Alternately, these may be used when a specific rounding mode is required at a specific location for mathematical precision reasons.

```
dsp_round(expression, binary_point_bits)
dsp_round_to_nearest(expression, binary_point_bits)
dsp_round_to_zero(expression, binary_point_bits)
dsp_round_to_infinity(expression, binary_point_bits)
dsp_round_to_plusinfinity(expression, binary_point_bits)
dsp_round_to_minusinfinity(expression, binary_point_bits)
```

All of these operators take the value of the input expression and round it off so that it maintains *binary_point_bits* of precision (or, at the $2^{-binary\_point\_bits\ place}$ position). Negative values of *binary_point_bits* are acceptable, if rounding off at some positive power of two is desired. The different forms just affect the rounding technique used. Both the `_round` and `_round_to_nearest` forms perform a round-to-nearest operation (rounding to 0 if the next bit after the rounding position is 0, or rounding away from 0 if it is 1). The `_to_zero` and `_to_infinity` forms are opposites, rounding towards zero (truncation) or away from zero in a fixed manner. Similarly, the `_to_plusinfinity` and `_to_minusinfinity` are opposites, rounding up or down in a fixed manner. They are all used in a fairly straightforward manner. The following all round off variable x to the nearest 1/4 (0.25, or 2 points of binary precision), but using the different rounding rules.

```
dsp_round(x, 2)
dsp_round_to_nearest(x, 2)
dsp_round_to_zero(x, 2)
dsp_round_to_infinity(x, 2)
dsp_round_to_plusinfinity(x, 2)
dsp_round_to_minusinfinity(x, 2)
```

On the other hand, it may sometimes be necessary to limit large values arbitrarily — a “high-end rounding,” or saturation. The following operation truncates a value when this is necessary.

`dsp_saturate(`*`expression`*`, `*`range_bits`*`)`

This operation takes the input expression and cuts it off if it has more than *range_bits* of precision (or, at the 2^{+range_bits} position). Negative values of *range_bits* are acceptable, if saturation at some negative power of two is desired. The value is truncated unless the input expression is too large to be expressed in the output precision. In these cases, it is limited at the maximum/minimum value expressible in the output precision. For example, the following code limits the value of the variable x to within ±4 (but not equal to 4).

```
dsp_saturate(x, 2)
```

These operators cover all of the basic cases, but it may prove desirable to add more exotic ones in the future.

2.1.4. An Example

To illustrate the function of the DSP integers, the following simple FIR filter uses a variety of sizes and precisions in a way that is typical of DSP kernels.

```
int input((24,23))[128];
int coeff((24,23))[128];
int current_position, i;

// Initialize ports & input
current_position = 0;
for (i=0; i<128; i++) input[i] = 0;
// Later, initialize coefficients, themselves

// Routine called once for each input
int((24,23)) fir_filter(int new_data((24,23)))
{
      int a((56,47));         // Extended-precision accumulator
      int *coptr((24,23));
      int j;

      a = 0;                              // Initialize values
      coptr = &(coeff[0]);
      input[current_position++] = new_data;      // Record input
      for (j=current_position; j < 128; j++)     // Filter loop
            a += input[j] * *(coptr++);          // Reduces
      for (j=0; j < current_position; j++)       // Filter loop
            a += input[j] * *(coptr++);          // Reduces
      fir_filter = dsp_round_to_nearest(a / 128);
}
```

There are several critical features of this code that should be pointed out:

- The inputs are a sequence of 24-bit, fully fractional numbers. These are a common size in high-quality audio algorithms. It is also common to find slight variations on these algorithms that use 16-bit inputs and 24-bit filter coefficients. DSP-C is capable of handling both variations (or more) with slight adjustments to the datatypes.
- The 56-bit accumulator used in the filter has both extended precision, so rounding is not necessary until the end of the filter, and extended range, so that overflow or saturation cannot normally occur (both are undesirable in these applications).
- The actual core calculation, including the necessary alignment shifts and showing temporary values, is illustrated in Figure 3a. On most architectures, these variables will be simulated with 32 and/or 64 bit integers, as in Figure 3b. Extra precision bits are added to most of the values as shown, while one of the inputs must have some of its extra precision eliminated prior to multiplication in order to maintain alignment.
- When the calculation is complete, the final result is divided (a right shift, in this case) and rounded off using an explicit rounding operator. Rounding could also be done implicitly by assigning the value in the accumulator to a lower-precision datatype, but this explicit operator ensures that no additional precision is carried after the rounding.

Even simple-looking lines of code like this can hide a considerable amount of alignment and rounding complexity that would normally have to be handled manually with most existing DSP programming environments. The time savings of handling this complexity handled automatically for larger codes can be easily extrapolated from this.

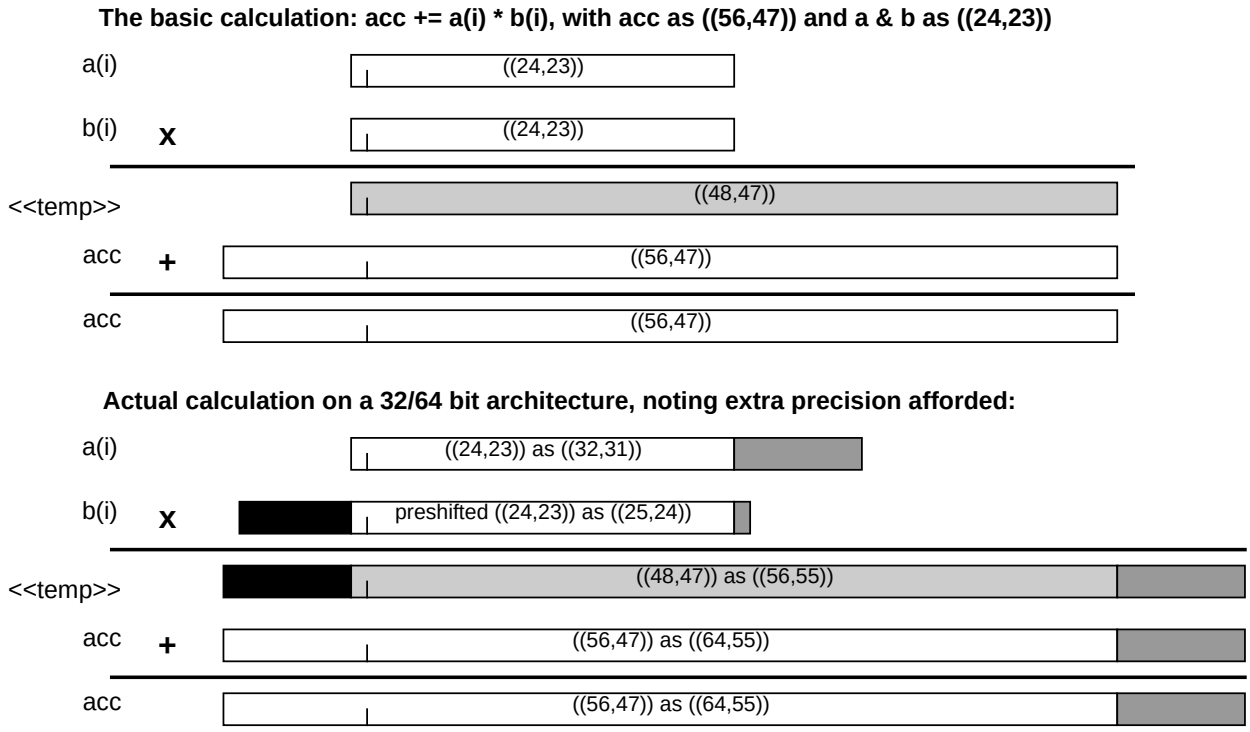

Figure 3: An illustration of the alignment and precision issues in the central line of the FIR filter example. A) View of the calculation if the architecture supports all data types natively. B) View of the calculation on an architecture that supports only 32 and 64 bit integers, with the necessary shifting and extra precision.

2.2. Complex Numbers

Many DSP algorithms use complex numbers as well as real ones, thanks in large part to the fact that the common FFT algorithm generates complex results. An ANSI semi-standard for complex numbers in C has already been developed, but having it (or something very similar) as an integral part of DSP-C would be mutually beneficial, since it is currently available only on a few compilers and would be very helpful with DSP operations.

2.2.1. Syntax & Usage

The keyword `complex` may be added to the declaration of any existing integer, floating point, or DSP integer data type. Only a few types may not be made complex: `void`, `char`, `enum`s, and bit fields. However, there would be little or no reason to make any of these complex, so this is not much of a limitation. Thus, the following two declarations create a complex integer, complex DSP integer, and complex floating point number, respectively.

```
complex int ci;
int complex dspci((24,23));
float complex cf;
```

All standard ANSI C declaration modifiers (storage class specifiers, type qualifiers, pointer indicators, and initializers) may be added to these declarations, just as in standard ANSI C. In memory, each complex number is represented as a pair of numbers of the original base type, one for the real part of the number and one for the complex part. From the programmer's point of view, however, they appear to be a single value.

The imaginary portion of complex constants may be specified by simply taking an ordinary, real constant of the desired type and adding an `i` or `j` at the end of it. This `i` or `j` must not be separated from the base constant by a space, multiplication symbol, or anything else — instead, it must be an integral part of the constant. Similarly, an `i` or `j` by itself will not be interpreted by the compiler to be "1x*i*," but instead will be interpreted as the variable `i`. A few examples of proper and improper use follow.

```
w = 5 + 2i;       /* Correct! */
x = 3i;           /* Correct! */
y = 3 + i;        /* Wrong: i is a variable here */
z = 5 + bi;       /* Wrong: bi is a variable name, not b*i */
```

Implicit type conversions from standard types to the real part of complex types, or from one complex type to another, follow the same promotion rules as the base types in standard C. Implicit type conversions from complex types to real types, however, are not allowed. Instead, the following portion-extraction operators must be used first to pick out either the real or imaginary part of the number.

```
real part =       creal(complex_expression)
imaginary part =  cimag(complex_expression)
```

`creal` and `cimag` may be used for more than just a quick way to get the two subsections of a complex variable out separately, however. They may also be used in any expression just like any other value of the base C type, and may even have their addresses assigned to real pointers, which may then access them independently of the fact that they are part of a complex number.

Most standard C operators also work with complex numbers, but a few exceptions exist. Table 2 summarizes the effects of various C operators with complex numbers. Examples for the binary operators use `(a+bi) OP (c+di)` as their basis.

Operator	Rule/Effect
`&`	Allowed to take addresses of complex numbers or their real components via the `&(cimag())` and `&(creal())` keywords
`*` (pointer)	Allowed to point to complex numbers or their real components independently using `*(&(cimag()))` and `*(&(creal()))`
`+, -`	Unary versions allowed, work on real & imaginary parts separately
`~, !`	Not allowed
`++, --`	Not allowed, but works normally for `complex *` pointers
`(), [], ., ->`	Use in functions, arrays, structures, and unions is normal
`sizeof`, casting	Work normally. `sizeof` always returns two times the size of the base type, obviously.
*, *=	Works normally, result = (ac – bd) + (bc + ad)*i*
/, /=	Works normally, result = ((ac+bd)/(c*c+d*d)) + ((bc–ad)/(c*c+d*d))*i*
%, %=	Not allowed
+, +=	Works normally, result = (a + c) + (b + d)*i*
-, -=	Works normally, result = (a – c) + (b – d)*i*
<<, >>, &, ^, |, &&, ||, <<=, >>=, &=, ^=, |=	Not allowed
<. >, <=, >=	Not allowed
==, !=	Works normally. Both real and imaginary parts must match to be equal.
Condition tests (`if`, `switch`, `? :`)	Unlike a normal `int`, a complex cannot be used to control any of these operations
=	Works normally

Table 2: A summary of the effect of existing C operators on complex number types.

A pair of complex-only operators have also been defined to handle conjugates and absolute values, important operations that are not necessary with real numbers. It should be noted that the absolute value operator is not an actual part of the proposed ANSI standard due to the fact that it requires a square root operation, but since it is very critical for high-performance DSP algorithms that need to convert complex frequency domain information into a real form for further processing, we have included it as an operator instead of a library routine. The examples assume an input of `a+bi`.

```
conjugate =       conj(complex_expression)      = a - bi
absolute value =  cabs(complex_expression)      = sqrt(a*a+b*b)
```

For more low-level details, readers should consult a document on the proposed standard.

2.3. Short Floating Point

For some applications, primarily in graphics, floating point may be very desirable but full-fledged 32-bit IEEE 754 single floating point values are not necessary — and the data bandwidths encountered tend to force us to make do with a smaller data type. In these cases, it may be desirable to introduce a 16-bit floating point type.

If we would choose to include such a type, it should be defined after taking a survey of available hardware and determining what sizes are actually available and supported by industry. A format with 1 sign bit, 5 exponent bits, and 10 mantissa bits might be a logical starting point, but hardware availability could adjust this.

2.3.1. Syntax & Usage

Since most of C's existing floating-point support is just fine, we would really only need to add a way to declare these variables. The simplest way would be to make the existing ANSI C `short` keyword work along with the `float` keyword as well as the `int` one:

```
short float identifier;
```

This would really be the only addition necessary, unless other sizes of floating point numbers were also desirable, such as 24-bit ones.

3. DSP-C Operators

Along with some new types, new operators are also essential to allow us to use the bewildering array of new instructions that DSP processors and conventional processors with multimedia extensions are capable of executing. One of the great features of C is that it allows a programmer to use virtually every normal mathematical operation commonly supported at the machine level in conventional microprocessors, including all of the various bit-level operations that other high level languages have often overlooked. Since C was developed in the 1970's, however, the variety of instructions natively supported by processors has increased dramatically, and we need to be able to generate code for at least some of these new instructions in order to get maximum performance from our software.

Today, most compilers either cannot generate code with these instructions at all, or can only generate code with them through the use of library entries that unfortunately may limit the ability of the compiler to fully optimize the code near the new instruction. Even if function call overhead is made unnecessary by inlining the instruction, opportunities for optimization are still often missed. It is frequently difficult to move non-dependent instructions past these "library instructions," in either direction, as the compiler cannot be completely sure as to how any side effects of the new instruction may interact with its normally generated instructions. Also, it is often hard for library-defined instructions to take advantage of new features such as MMX-style SIMD instruction execution without manual alignment of the input and output, even though this can be critical for maximum performance on numerical algorithms.

Unless otherwise noted, each of these operators should compile down into a single instruction or a small number of instructions on most architectures. For the more complex instructions, a large inlined chunk of code or a library call may be used instead. Essentially, these operators have been chosen because they are fairly close to actual machine instructions, but not so close that they are too architecture specific.

The syntax for these operators is still somewhat flexible. All operators have a function-like syntax provided, but a few also have an operator syntax starting with a `^` symbol. A different symbol, such as `@` or `%`, might also be a good choice as a leading indicator, if this type of symbolic notation is used at all.

3.1. Saturating Add/Subtract

One of the most fundamental mathematical differences between most traditional computer mathematics and DSP mathematics is the presence of saturating addition and subtraction. While most computer addition and subtraction overflows and wraps around to the opposite extreme if the maximum or minimum representable sizes are exceeded (also called *modulo* mathematics), the results of many DSP operations are better represented by a system that simply stops when the maximum or minimum representable value is reached. Oftentimes, this requirement comes from the fact that DSP applications are typically time-sensitive, so it is not possible to halt the application and deal with special overflow conditions using exception handling code. Instead of having special code to handle the occasional overflow, these applications must make do with the answer proved by the add or subtract in the first place. Because most DSP applications are working with real-world signals, it is better to be close but not quite correct than to be flipped to the opposite end of the scale. In video terms, for example, if a result is "whiter than white" it is better to just return "white" than to return a nearly-black pixel value.

To give a concrete example, suppose that we were adding 94 and 41 using 8-bit signed numbers (where the maximum value representable is 127 and the minimum is -128). The answer is 135, so with most addition the answer obtained would be –121 with an overflow indication. However, with DSP signals it is better to be close to the value at 127 than be stuck way down at –121. Hence, the add is saturated out at 127. Similar saturation occurs when we try to go below –128.

3.1.1. Syntax & Usage

For the most part, saturating adds and subtracts can freely replace conventional adds and subtracts in code. They also have the same precedence. We just have to have an operator symbol to distinguish them. Here are proposals for both an operator-like notation and a function-like notation.

```
result = operand1 ^+ operand2
result = operand1 ^- operand2
result = dsp_satadd(operand1, operand2)
result = dsp_satsub(operand1, operand2)
```

These are then compiled down to saturating add instructions instead of normal add instructions.

3.2. Fused Multiply-Add / Multiply-Subtract

Most architectures that support DSP or multimedia operations have some sort of fused multiply-add and/or multiply-subtract instruction(s). Hence, any DSP-C compiler must be able to find opportunities to take multiplies that feed into addition instructions and transform them into single instructions. There is no need to have any sort of special syntax for this, but just rules and guidelines to follow.

3.2.1. Implementation Notes

After creating graphs of the dataflow through C expressions, any multiply that leads directly into an add or subtract should be fair game for transformation into a fused operation, unless significant alignment or other transformations must be added between them. If multiplies feed into both operands of an addition, the right-hand multiply should be the one fused to the addition.

3.3. Minimum/Maximum

Many DSP algorithms look for minima or maxima in their inputs at some stage of the algorithm. Compression algorithms are a common location, but even simple scaling routines may examine their input first to find maximum values, in order to prevent overflow later during the actual operation. As many DSP architectures supply instructions specifically to speed up these operations, it makes sense to include them in DSP-C.

3.3.1. Syntax & Usage

The simplest and most common form of minimum and maximum instructions simply examine two or more inputs and return the maximum or minimum value from the set of inputs. These can be expressed most succinctly using a function-like notation. The first form examines several scalar operands, while the second finds the minimum or maximum of an entire CAA (see the next chapter) with a single, very complex operation. The array given may have multiple dimensions.

```
minimum = dsp_min(operand1, operand2 <<, operand3 . . .>>)
maximum = dsp_max(operand1, operand2 <<, operand3 . . .>>)
array_minimum = dsp_min(array)
array_maximum = dsp_max(array)
```

However, it is also easy to express the two-input versions of these operations using the existing ternary operator in standard ANSI C. As a result, it would be a good idea to pattern-match for these conditions. It should be noted that <= or >= can be freely substituted for < or > in the

following, since we will simply be selecting from identical operands when they happen to be equal.

```
minimum = ( operand1 < operand2 ? operand1 : operand2 )
minimum = ( operand1 > operand2 ? operand2 : operand1 )
maximum = ( operand1 > operand2 ? operand1 : operand2 )
maximum = ( operand1 < operand2 ? operand2 : operand1 )
```

Finally, one may want to simply know where the minimum or maximum value occurs among a group of numbers. Variations of the `dsp_min` and `dsp_max` operations work to provide this information, instead, as an `int` value. The basic array forms only work with single-dimensional CAAs, since C functions cannot return multiple integers very easily and defining "first" or "last" in a multi-dimensional array is much trickier.

```
position = dsp_first_minpos(operand0, operand1 <<, operand2 . . .>>)
position = dsp_first_maxpos(operand0, operand1 <<, operand2 . . .>>)
position = dsp_last_minpos(operand0, operand1 <<, operand2 . . .>>)
position = dsp_last_maxpos(operand0, operand1 <<, operand2 . . .>>)
position = dsp_first_minpos(one-dimensional_array)
position = dsp_last_minpos(one-dimensional_array)
position = dsp_first_maxpos(one-dimensional_array)
position = dsp_last_maxpos(one-dimensional_array)
```

If positions in multi-dimensional arrays are necessary, however, the longer form of these operations allows the specification of the dimension sequencing using the *order_array* operand and returns the position of the maximum or minimum in the *position_array* operand. Both one-dimensional arrays should be composed of some sort of integer type and have a number of elements equal to or larger than the number of dimensions in the *array* (any extra elements are ignored). The *order_array* specifies how "first" and "last" are to be defined in the event of multiple equal minima or maxima by numbering the dimensions as 0, 1, 2, etc. The elements of all the dimensions are then "flattened" out into a single-dimensional array according to this ordering. For example, an order array of 0, 2, 1 would, for purposes of ordering, cause a 3-D [[X,Y,Z]] CAA to flatten out to a 1-D array of: [[0,0,0]], . . . , [[0,sizeY,0]], [[0,0,1]], . . . , [[0,sizeY,1]], [[0, 0, 2]], . . . , [[0, sizeY, sizeZ]], [[1, 0, 0]], . . . [[sizeX,sizeY,sizeZ]]. The coordinates of the first or last minimum or maximum in the array are then returned in the *position_array* in the order of the dimensions in the original array (and **not** in the order specified by the *order_array*). To continue the previous example, the *position_array* would return with the X,Y,Z coordinates of the chosen array, and **not** the X,Z,Y coordinates (even though that is how they were searched).

```
array_minimum = dsp_first_minpos(array, order_array, position_array)
array_minimum = dsp_last_minpos(array, order_array, position_array)
```

```
array_maximum = dsp_first_maxpos(array, order_array, position_array)
array_maximum = dsp_last_maxpos(array, order_array, position_array)
```

While these operations are not strictly a form of the minimum / maximum function, they can be optimized during vector reduction in a similar way, because they are associative (and can be made commutative, with a bit of adjustment). The scalar forms return a number indicating which element from among the ones they were given was the minimum or maximum, starting with 0 with the first entry given and counting up (i.e. returning the operand number). If more than one element is equal to the minimum or maximum value, the `_first_` versions of the operators return the first one, while the `_last_` versions of the operators return the last one.

3.3.2. Implementation Notes

Overall, these operations can just be seen as a specialized and associative form of multiplexing, described below. However, they should be given special treatment for two reasons. First, many architectures include special instructions just to handle this special case. Furthermore, since we know that all of these functions are associative, we can perform reduction optimization when these functions are used as reduction operators. See the next chapter for more on this.

The various array forms of the instructions will typically be implemented as a short inline loop inserted right into the code. Having these little loops as a single, fundamental operation can greatly help during compiler optimization, because they are often used as reduction steps that require specialized optimization.

3.4. Multiplexing

To replace control flow in data-intensive algorithms, many multimedia architectures include specialized multiplexing or conditional move instructions, that perform a conditional data move or a selection between two inputs, under the control of an independent input operand. Furthermore, in most of these architectures multiple multiplexing operations may occur in parallel through the use of SIMD instructions. The result of these instructions is similar to that obtained through the use of C's ternary operator (*`condition ? select1 : select2`*), so a DSP-C compiler should be able to convert ternary operations into native multiplexing instructions, instead of control flow instructions, whenever possible.

3.5. Predication (`if` statements)

Another way to replace control flow at the heart of an otherwise data-intensive algorithm is to use predicated result code. This is when code within an `if` statement is executed whether or not

it is actually needed. Afterwards, the result(s) is/are either used or discarded, based on whether or not the original `if` condition evaluated to a non-zero value. In hardware, these functions can be performed by having predicated instructions, that actually cause writeback from instructions to be cancelled if they are unneeded, or by having a multiplexing operation at the end of the "predicated" region conditionally make results permanent. The Intel IA-64 architecture, for example, has probably the most sophisticated version of predication found on any architecture, included to reduce or eliminate the need for large numbers of control flow instructions.

We may choose to allow small `if` control structures in optimized DSP-C vectorized loops if they can be successfully predicated, to prevent the introduction of excessive control flow into highly optimized code. On some architectures, primarily uniprocessors or MIMD multiprocessors, it may even be possible to support longer `if` statements through traditional control flow techniques, even mid-loop. Over the course of compiler development, the exact policies for handling `if` statements should be determined.

3.6. Count leading / trailing 0s / 1s

It is sometimes necessary to find the highest-order set or cleared bits in an integer word, or, conversely, the lowest-order set or cleared bits. Certain pseudo-FP and cryptography algorithms are prime candidates for these operations, for example. Several architectures available today implement some form of these instructions, although they are not very standardized (some find ones, some zeroes, results may vary by 1, etc.). The following operators use the hardware instructions to find the desired information with the minimum number of instructions on the target architecture.

3.6.1. Syntax & Usage

All of these operators examine the operands given and return the appropriate result. Each pair of operations actually returns the same result of type `int`, in the range of 0 to the word length of the operand. The `leading` and `trailing` versions of the operator return the number of contiguous 0's or 1's, starting from the most significant bit (`leading`) or the least significant bit (`trailing`), until the block is interrupted. The `first` and `last` versions of the operator return the bit position of the first (most significant) or last (least significant) bit of the type requested. Since the bits are numbered from 0 on up to the bit width – 1 (the bit width itself is returned if none are found), from the end of the word where the search starts, the result is always equal to the paired `leading` or `trailing` form of the function.

```
result = dsp_leading_0s(operand)
result = dsp_first_1(operand)
```

```
result = dsp_leading_1s(operand)
result = dsp_first_0(operand)

result = dsp_trailing_0s(operand)
result = dsp_last_1(operand)

result = dsp_trailing_1s(operand)
result = dsp_last_0(operand)
```

Both operators in each pair can be used interchangeably. The two forms are provided simply for program readability purposes. Generally, a programmer will choose a version depending upon whether he or she is interested in the 0's or the 1's in the word.

3.7. Rotation

Rotation instructions have been around in processors for ages, but ANSI C happened to forget them amidst all of the other operations it natively implemented. They are just a variation on the basic shift instructions that shift-in bits at one end of the input word as they are shifted out of the other, instead of shifting in 0's or sign-extension bits. They have some uses in utility byte-rearrangement, cryptography, etc.

3.7.1. Syntax & Usage

Rotations work just like left and right shifts in existing C code. They also have the same precedence. We just have to have an operator symbol to distinguish them. Here are proposals for both an operator-like notation and a function-like notation.

```
result = operand1 ^<< shift_amount
result = operand1 ^>> shift_amount
result = dsp_rotleft(operand1, shift_amount)
result = dsp_rotright(operand1, shift_amount)
```

These are then compiled down to rotation instructions instead of left or right shift instructions.

3.7.2. Implementation Notes

It should be noted that these won't be implementable as single-instruction operations when there are additional precision bits at the end of a DSP integer type, since we must execute the rotation as if those bits didn't exist. In these cases, a short sequence of instructions should be generated to rotate the bits around *only* in the declared length of the DSP integer. On the other hand, this effect should not normally be a problem with conventional shift instructions.

3.8. Appending

Some bit manipulation operations may require that two or more integer variables be appended together into a single, larger variable before other computation. The most common situation in DSP-C will be if these variables are appended prior to one of the permutation operations, defined next.

3.8.1. Syntax & Usage

Append operators attach two or more operands to each other bitwise to form a single, larger value. The most significant bit of *operand1* becomes the most significant bit of the *result*, while the least significant bit of the last *operand* becomes the least significant bit of the *result.* Other operands are included in the middle of the *result* in the order they appear.

```
result = operand1 ^| operand2 << ^| operand3 . . . >>
result = dsp_append(operand1, operand2 << , operand3 . . . >>)
```

For the two-input case, if the two inputs are DSP integers of type ((m,n)) and ((x,y)), respectively, then the output consists of a single, longer DSP integer of type ((m+x, undefined)). Versions with more inputs simply extrapolate on this model. Precision information is lost in the transition, because it makes little or no sense afterwards. Instead, any assignment of the appended result to a variable of type ((i, j)) simply takes the lowest-order i bits and sticks them into the variable. It is the programmer's responsibility to make sure that this actually makes sense for his or her application.

3.8.2. Implementation Notes

Internally, this operation may not even be turned into an instruction. It just tells the compiler how to "view" a group of inputs prior to some further operation that manipulates them. A common case would be that two appended variables would be fed into an architectural byte-packing instruction as its two operands. Appending is mostly used to permit the general permutation instructions to deal with a wide variety of inputs of different lengths without requiring many different forms of that operator. It may also be used to manually pack data before it is stored in memory. In cases like these, one or more instructions will probably need to be generated to handle the necessary data manipulation.

3.9. Splitting

The opposite of appending is splitting up large types into multiple subparts. The appending operations may be used in reverse to facilitate this process.

3.9.1. Syntax & Usage

Split operators cut an operand bitwise to form two or more smaller values. The least significant bit of the *operand* becomes the least significant bit of the last *result*. Other results are cut out from the *operand* in the order they appear, with *result1* receiving the most significant bits. Once all results have been filled, any extra bits at the most significant end of *operand* are discarded. If too few bits are available, additional 0 bits will be appended to the most significant end of the *operand* as necessary.

```
result1 ^| result2 << ^| result3 . . . >> = operand
dsp_split(result1, result2 << , result3 . . . >>) = operand
```

For the two input case, if the two outputs are DSP integers of type ((m,n)) and ((x,y)), respectively, then the input should consists of a single, longer DSP integer of type ((m+x, any)). The low-order x bits of the input will be assigned to the ((x,y)) result, while the high-order m bits of the input will be assigned to the ((m,n)) result. If the input has more than m+x bits, the high order bits of the input beyond this point will be discarded. On the other hand, if the input has fewer than m+x bits, then parts of the outputs will be zeroed, starting with the high-order bits of the ((m,n)) input. Versions with more inputs simply extrapolate on this model. Precision information in the input operand is ignored for the purposes of this operation. It is the programmer's responsibility to make sure that the results of the split actually makes sense for his or her application.

3.9.2. Implementation Notes

Internally, this operation is exactly like appending, but in reverse. It may not require any instructions at all, instead just requiring a modification as to how the subsequent instructions "view" their inputs. On the other hand, it may compile down to a sequence of instructions, especially if it's responsible for unpacking data coming in from memory.

From a compiler point of view, this operation is very unusual because it is "performed" on the left-hand-side of an expression (it's just appending, but on the "wrong" side!). The grammatical adjustments necessary to handle this situation may turn out to be too complex, so an alternate syntax may be necessary.

3.10. Byte-by-Byte Permutation

Many data structures used for DSP applications contain data packed in unusual ways. Much of the time, a programmer may simply tell DSP-C what type of data is in the input or output and then use the append and split operators to pack and unpack the data, respectively. However, on occasion a finer level of control may be necessary, or dynamic rearrangement of input operands may be required. For these cases, a full byte permutation operation is provided.

3.10.1. Syntax & Usage

This operator will produce a *result* of the same size as the input *operand*, but with the bytes rearranged according to the pattern encoded in the *control_vector* input, a CAA of any kind of `int` variable. If possible, try to specify a constant vector with the `const` keyword, since this may result in faster code. In general, `unsigned char` integers in the control vector are preferred, although other sizes and subtypes will work, too.

```
result = dsp_permute(operand, control_vector)
result = dsp_bytepermute(operand, control_vector)
```

If the size of the *operand* is not a multiple of 8, 0 bits are appended at the most significant end to make its length a multiple of 8. The bytes in both *result* and *operand* are numbered from `0` (least significant) on up to *end_byte* = `(sizeof_in_bytes(`*operand*`) – 1)`. Bytes are then rearranged so that `result_bytes[n] = operand_bytes[control_vector[[n]]]`. Input bytes may be replicated multiple times in the output by simply repeating the same values in the control vector. If values in *control_vector* are larger than *end_byte*, then the corresponding output bytes are undefined. If the control vector has fewer than (*end_byte* + 1) elements, then the most significant bytes of the *result* (the ones with no control information) will just be equal to the most significant bytes of the *operand*, passed straight through. Figure 4 depicts a typical use of a permutation operand.

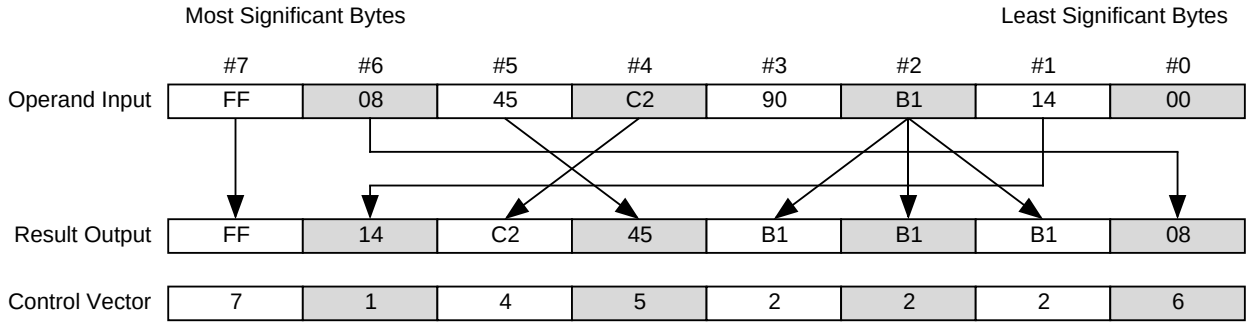

Figure 4: A typical byte-by-byte permutation. Note how some bytes are moved around, some stay in place, some are not copied at all, and others are replicated multiple times.

3.10.2. Implementation Notes

Except on architectures with a full permute instruction (like PowerPC's Altivec), code generated from byte permutation operations will often be considerably different depending upon whether the control vector is a constant or not. If it is, then the compiler should figure out the smallest number of instructions that can produce the necessary permutation of input bytes, and generate that sequence of instructions. On most architectures, this will be a combination of shifts, rotates, packs, unpacks, "swizzles," etc. On the other hand, a more generic piece of code can be generated when the control vector is a variable, since the code will have to be able to handle any combination of inputs. The only code variation in this case may be caused by the length of the input.

3.11. Bit-by-Bit Permutation

This is essentially the same as byte-by-byte permutation, defined previously, but down another level. It can also be used for packing and unpacking operands, but is more useful in handling cryptography kernels or the generation and decoding of variable-length bitstreams of data.

3.11.1. Syntax & Usage

This operator will produce a *result* of the same size as the input *operand*, but with the bytes rearranged according to the pattern encoded in the *control_vector* input, a CAA of any kind of `int` variable. If possible, try to specify a constant vector with the `const` keyword, since this may result in faster code. In general, `unsigned char` integers in the control vector are preferred, although other sizes and subtypes will work, too. Unlike with byte permutation, larger integers may even be required to provide enough range (1024 bit inputs are common in some forms of cryptography, for example).

result = dsp_bitpermute(*operand*, *control_vector*)

The bits in both *result* and *operand* are numbered from `0` (least significant) on up to *end_bit* = `(sizeof_in_bits(`*operand*`) – 1)`. Bits are then rearranged so that `result_bits[n] = operand_bits[control_vector[[n]]]`. Input bits may be replicated multiple times in the output by simply repeating the same values in the control vector. If values in *control_vector* are larger than *end_bit*, then the corresponding output bits are undefined. If the control vector has fewer than (*end_bit* + 1) elements, then the most significant bits of the *result* (the ones with no control information) will just be equal to the most significant bits of the *operand*, passed straight through. Figure 4 also applies to a bit-by-bit permutation, if the values of the input and output are bits instead of bytes. The process is virtually identical.

3.11.2. Implementation Notes

Code should be generated for these permutations in a manner analogous to the byte-by-byte permutation code generation. As in that case, an optimized or generic code block may be produced depending upon whether the control vector is a constant. However, the speed difference between the two cases will probably be much greater in this case, so the use of constant permutations is much more important.

3.12. Transcendental Math

Several multimedia architectures nowadays are offering high-speed support of a limited selection of transcendental mathematical functions, primarily roots and powers. Hence, it would be helpful to have these functions be actual operators, and not library calls (although on some architectures they may still be emulated using library calls). On the other hand, full implementations of trigonometric functions are generally still relegated to software, because they are much more complex routines and typically do not occur as often in key functions. Most DSP algorithms which do rely on trigonometry work fine using lookup tables or can handle the delay of a library routine.

3.12.1. Syntax & Usage

Using the operator form of these operations should be almost exactly like using the ANSI C math library. In fact, we may just want to pattern-match for calls to those library routines and turn the existing library calls into operators on a case-by-case basis.

```
result = dsp_sqrt(operand)
result = dsp_root(operand, desired_root)

result = dsp_power2(operand)
result = dsp_exp(operand)
result = dsp_power(operand, desired_power)

result = dsp_log2(operand)
result = dsp_ln(operand)
result = dsp_log(operand, base)
```

Special versions of these functions are supplied for base-2 and base-e forms, because there are often hardware instructions optimized for these particular bases (in fact, they are often the only forms implemented natively).

3.12.2. Implementation Notes

These operators really only make sense for use with floating point inputs and outputs, since those are the only versions commonly available in hardware — and often the only versions that can be mathematically accurate enough to be worthwhile. We may want to allow integer inputs, just for consistency, but these will virtually always result in calls to library functions. Defining the possible error that could result from any call to integer versions of these routines is another potential source of difficulty.

Versions of these functions for floating point complex numbers should also be supported, although the code generated will obviously be considerably different.

3.13. Galois-Field Arithmetic

Galois field arithmetic is an unusual closed-form boolean mathematics used in some error-correcting code algorithms such as Reed-Solomon coding (used in CD players, among other things). These are commonly implemented using lookup tables on most architectures, instead of actual instructions. Hence, it may not be worth it to include them as fundamental operations. On the other hand, if they are available then instructions may be generated on the rare architectures that support them.

3.13.1. Syntax & Usage

The four common Galois-field operations (addition, multiplication, logarithms, and exponentials) would probably best be described using function-like notation.

```
result = dsp_galois_add(operand1, operand2)
result = dsp_galois_mult(operand1, operand2)
result = dsp_galois_log(operand)
result = dsp_galois_exp(operand)
```

These are then compiled down to rotation instructions instead of left or right shift instructions.

3.13.2. Implementation Notes

If these are implemented at all, it would probably only make sense to support 8-bit integers with the operators (the size used for most ECC algorithms). If we address ECC applications wholeheartedly, it might also make sense to include operations such as boolean division (used in CRC algorithms), linear feedback shift register operations, and others. Mathematically, this area is probably the most unusual DSP application because of its boolean algebra.

3.14. More?

Many more instructions may need to be added to this list, since these instructions only cover the features provided by a few of the leading multimedia architectures that we believe are generally useful to many applications. Low-level operator support for other multimedia instructions, or groups of instructions, that might find valid usage in a wide variety of programs and algorithms could also be added.

On the other hand, some multimedia instructions specified in architectures are very unusual or application-specific. In these cases, it would generally be better to omit them from the baseline DSP-C specification to avoid adding too many hardware-specific hooks to the language. One example that is reasonably common is the SAD (sum of absolute differences) instruction, found on several architectures to speed up MPEG encoding. This single-application instruction probably does not belong in the baseline DSP-C specification for two reasons. First, it is extremely application specific. In general, it has been designed for a single application. Second, it is a very complex instruction that can easily be implemented in slightly different ways on different architectures. As a result, a default "SAD" operator might not be easily compilable to the actual SAD instructions on all different architectures.

Instead, a notation for adding multimedia instructions to the compiler on an architecture-by-architecture basis should be a part of the DSP-C specification. While using these instructions will result in somewhat architecture-specific code (going against the DSP-C philosophy), it is also often necessary in order to get the highest performance possible (which *is* a part of the DSP-C philosophy). As a result, it would be a good idea to have an "instruction extension notation" designed so that the other features of DSP-C (sophisticated integers, compiler optimization, vectorization, etc.) will still be able to work. Having such a mechanism will add some complexity to DSP-C, but it will allow programmers to extend the language somewhat in the future without resorting to chunks of assembly language every time a new instruction is encountered. The following two examples attempt to show how a 32-bit SIMD / 64-bit `ADD` instruction and a `SAD` instruction could be encoded. The key points to notice in the code are described in the following list.

- **`formats`:** This section establishes a series of prototypes for the instruction, which the programmer can then use in normal DSP-C code. Looping and other, similar constructs are allowed to allow for the specification of a wide variety of precision levels with a minimum of code here. SIMD parallel inputs are noted by having multiple appended sub-variables in each input and output. The compiler may then automatically try to schedule parallel operations to each of these slots separately, since this is a `dsp_parallel_instruction` (as noted at the top).

- **`operands`:** This section notes legal inputs and output registers, in the assembler's register specifier notation. The `operand` function is just a `sprintf`-like library function that returns a valid operand specifier if given the textual name of the operand. The fact that the accumulator input to the SAD instruction is limited to a single, specialized register is specified here.
- **`assembly`:** This section specifies the actual text that the compiler must produce to feed into a later assembly step.
- **`side_effects`:** Flags or architecture-specific registers that may be modified while the instruction executes, or that are used implicitly as inputs, should be noted here. Constraints on the implicit inputs should be noted, such as the fact that the carry bit must be cleared before the `ADD` instruction is allowed to execute.
- **`emulation`:** This is the code that the compiler can use to replace the function prototype in order to make legal and non-architecture specific DSP-C code, for emulation purposes on architectures without the instruction. In the case of the `SAD` instruction, this is done by inserting a library call into the code. Except for the expansion of operand inputs with the `operand` library call, the code replacement done by this section is the same as the insertion of code performed by DSP-C intelligent libraries, as is described in the "Intelligently Polymorphic Libraries" chapter.
- **Reduction controls:** Since the `SAD` instruction is actually an optimized reduction step, it is necessary to tell DSP-C this in order for it to optimize vector reductions correctly in automatically vectorized code, as is described in the "Vectorization of Loops" chapter. After the instruction is declared as being a `dsp_reduction_instruction`, the instruction setup tool looks for the reduction-control fields, which tell the compiler what coding structure to look for in a loop when a reduction is occurring, and how results produced on parallel processing elements can be combined into a single result, if this is even possible. The `interleaved` combining form is used when elements of the array are passed out to processors alternately in a round-robin fashion. This requires the reduction operation to be both commutative *and* associative. On the other hand, the `divided` form is used when the array is split into a number of contiguous blocks equal to the number of parallel processing elements and then the blocks are passed to each processor. This form only requires that the reduction operation be associative. In this case, both operations are the same.

Here are the actual instruction specifications. It should be noted that these will never be part of any C-language file, but instead would be fed into the compiler beforehand.

```
dsp_parallel_instruction ADD
{
      int n;

      formats:
            for(n=0; n<64; n++)
                  int((64,n)) ADD(int((64,n)) in1, int((64,n)) in2);
            for(n=0; n<32; n++)
                  (int((32,n)) ^| int((32,n))) ADD(
                  (int((32,n)) ^| int((32,n))) in1,
                  (int((32,n)) ^| int((32,n))) in2);

      operands:
            in1:  for(n=1; n<32; n++) operand(“r%d”,n);
            in2: for(n=1; n<32; n++) operand(“r%d”,n);
            ADD: for(n=0; n<32; n++) operand(“r%d”,n);
                  operand(“acc”);
                  operand(“mfhi”);
                  operand(“mflo”);

      assembly:
            “ADD “, in1, “,”, in2

      side_effects:
            flag_change(“overflow”);
            flag_change(“carry”);
            implicit_input(“carry”, must_clear);

      emulation:
            ADD = in1 + in2;
}

dsp_reduction_instruction SAD
{
      int n;

      formats:
            int((64,0)) SAD(
                  int((8,0)) ^| int((8,0)) ^|
                  int((8,0)) ^| int((8,0)) ^|
                  int((8,0)) ^| int((8,0)) ^|
                  int((8,0)) ^| int((8,0))) in1,
                  int((8,0)) ^| int((8,0)) ^|
                  int((8,0)) ^| int((8,0)) ^|
                  int((8,0)) ^| int((8,0)) ^|
                  int((8,0)) ^| int((8,0))) in2,
                  int((64,0)) accumulator);

      operands:
            in1:  for(n=1; n<32; n++) operand(“r%d”,n);
```

```
            in2: for(n=1; n<32; n++) operand(“r%d”,n);
            accumulator: operand(“acc”);
            SAD: operand(“acc”);

      assembly:
            “SAD “, in1, “,”, in2

      side_effects:

      reduction_initialize:
            dsp_reduce = 0;
      serial_reduction_step:
            dsp_reduce = SAD(in1, in2, dsp_reduce);
      parallel_interleaved_reduction_step:
            dsp_reduce = dsp_partial1 ^+ dsp_partial2;
      parallel_divided_reduction_step:
            dsp_reduce = dsp_partial1 ^+ dsp_partial2;

      emulation:
            graphics_sad(in1, in2, operand(“acc”));
}
```

These examples only tries to provide a potential model for an extension language. We might also include additional features from existing extensible compilers such as GCC. If we choose to pursue this technique, a much more formal specification will have to be devised. Also, flexibility will have to be included in the specification, since the definition of new instructions is inherently architecture-specific.

4. Vectorization and Parallelization of Loops

Arguably the most important aspect of DSP-C is the collection of adjustments made in order to facilitate the vectorization and parallelization of loops. These adjustments will undoubtedly be the focus of most of our future work to refine and improve DSP-C, because the ability of compilers and programmers to use these constructs to communicate information from easy-to-understand code into powerful optimizing routines must be tested out with a variety of applications and programming styles before the design of the language is finalized. We will undoubtedly be able to borrow the techniques necessary to perform the many steps in the compilation process from existing work. However, DSP architectures offer a wide variety of interesting and unusual complications to the basic compiler algorithms as a result of their unusual, highly optimized memory architectures. Streaming data, compiler controlled local memory, and other unusual obstacles must be overcome to produce high-quality DSP code.

This chapter describes the basic goals of future compiler work and some of the language constructs that we hope to add to C as a way to facilitate the development of compilers that support these techniques. The first part of the chapter describes the goals of our compilation process, covering the things we well both need to borrow and invent. The majority of the chapter discusses a new construct added to C to replace its native array structure: constrained access arrays (CAAs). This new data type, combined with standard C operations on them or the explicitly parallel operations that they allow, are a relatively simple and systematic way to approach the process of DSP and stream programming.

4.1. DSP-C Vectorization Philosophy

There is a fundamental conflict at the heart of every effort to make an optimized language for an application like DSP, for which there is a multitude of architectures that differ in some fairly fundamental ways but need to share some of the same code base.

First and foremost, we want the fastest code possible. If faster code can be written, then higher-quality results can be achieved (important for something like graphics) or cheaper chips can perform the fixed amount of work required (important for many high-volume embedded applications). Usually, to get this we must write custom code for our target architecture which takes advantage of all its unusual features. However, in order to achieve faster time-to-market we would like to be able to leverage existing code as much as possible. Usually the only existing *and* portable code for many DSP algorithms is written in C. Unfortunately, it is apt to be very unoptimized and slow. Its loops are usually almost impossible to vectorize or parallelize due to

aliasing problems. Therefore, we would like to add some "stuff" to C to patch up these problems without screwing up the language too much. Hopefully, we will be able to keep code as close as possible to C — and therefore, make the job of code writers who already know C or are porting code from C fairly easy. After all, if it is hard to learn or use DSP-C, then no one will choose to use it. At the same time, we need to be able to get a significant percentage of the speedup that can be obtained from hand-optimization of algorithms. If we add some "stuff" and do not get reasonably close to what is possible by hand, then we will have no choice but to add more special hooks for programmers to use.

This section describes the compiler optimization features that we think will be necessary to achieve good results during the vectorization and parallelization stages of compilation towards DSP architectures. We feel that constrained access arrays are the single most critical new language structure necessary for allowing these optimizations to be performed by a reasonable compiler. We have also provided for a limited selection of optimization flags, that should help to smooth over some potentially difficult spots that these structures alone may not allow us to find. Hopefully, additional major additions to C will be unnecessary, but only some significant compiler work will allow us to determine whether or not this is the case.

4.1.1. Pick DSP Loops

The first aspect of making a good vectorizing compiler is just being able to pick good loops in the program to spend time performing vector analysis on. To do this, we need to define what a "good" loop actually is. The simplest definition, that works for most key DSP loops, is that the loop is a parallel kernel, as depicted in Figure 5. In other words, it takes one or more vectors as its primary inputs, operates on them in some sort of element-wise manner, and then produces either a vector output or a scalar output calculated by performing some sort of mathematical reduction operation across the vector elements. Code which spends a considerable amount of time walking non-array data structures (like trees), uses large amounts of recursion, or contains critical, non-reduction-type data dependencies between the processing of vector elements should be largely bypassed for purposes of this analysis.

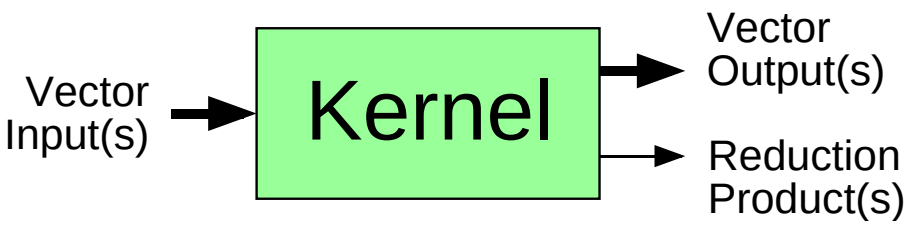

Figure 5: A fundamental DSP-C kernel loop consists of a loop with one or more vector inputs and vector and/or reduction-type scalar outputs. Unusual data access patterns are minimized.

Once high-level vector loops have been identified, we must penetrate down through the levels of nested loops to work on the innermost levels of the loop first.

4.1.2. Code Fragment Generation

Starting at the bottom of the identified loops, our compiler must start building dataflow graphs through the individual operations that make up the low-level code. These operations can be as fine-grained as simple instructions, or they can represent larger groups of instructions. When the dataflow among low level instructions is established, we should move quickly up to the code fragment level — groups of instructions dependent upon one or more loads — as the memory behavior of the kernel is the primary point of optimization that we will need to explore. These fragments are illustrated in Figure 6. Any potential for communication or synchronization between code fragments through registers or by using stores leading to loads in other fragments should be noted at this point, as should potentially parallel code fragments. The pattern of these loads and stores is the primary information that needs to be communicated up to higher level optimization routines, which should be initiated at this point.

After performing higher level optimization, we will eventually try to return and schedule these code fragments and the instructions inside them. The primary task at that point is to rearrange code fragments to hide the latency between load address generation stages and data returns. Generally, these gaps will only be a few cycles, since most of these loads and stores should be to on-chip memory, but scheduling small code fragments into these available cycles can dramatically affect performance in the long run, as the number of loads in DSP algorithms is generally very high. Inner loop unrolling may also be performed at this point in order to provide a larger selection of fragments for optimization. Final code generation and peephole optimization may then commence.

4.1.3. Bottom-Up Memory Analysis

The core of the analysis that must be performed on vectorizable loops is memory analysis. This is the most important part of the compilation process, and the one that we have attempted to ease the most with the addition of CAAs. Once basic code fragments and their memory access behavior have been established, memory analysis becomes the core of most of the rest of the steps.

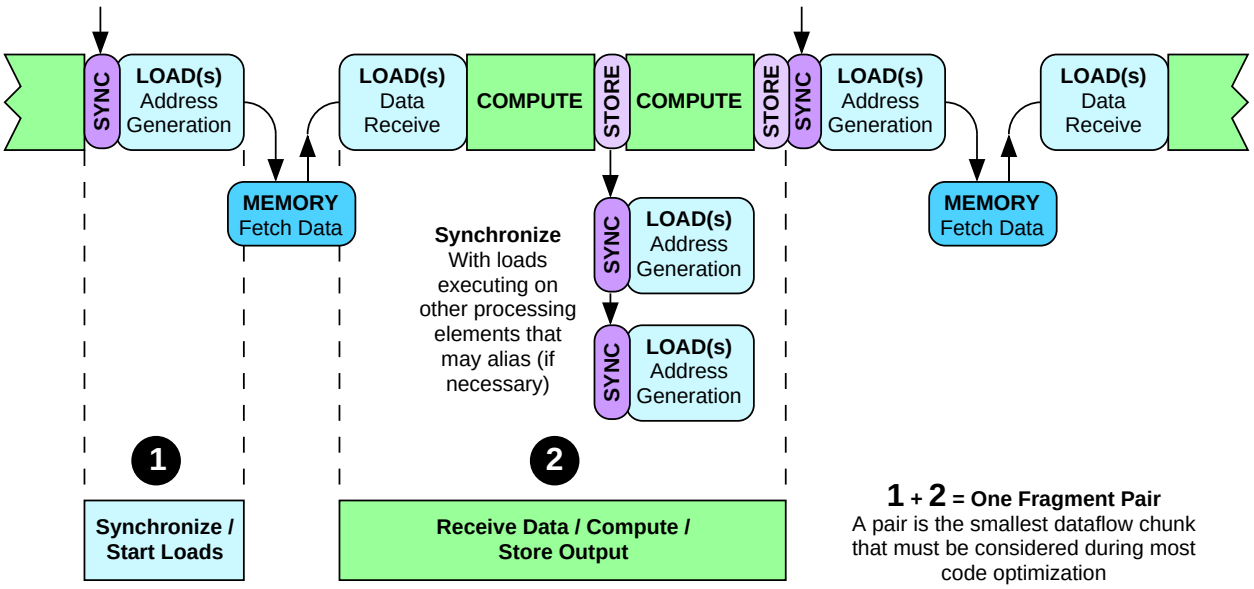

Figure 6: A series of code fragments. Note the relationship between loads, stores, and computation blocks in the fragments.

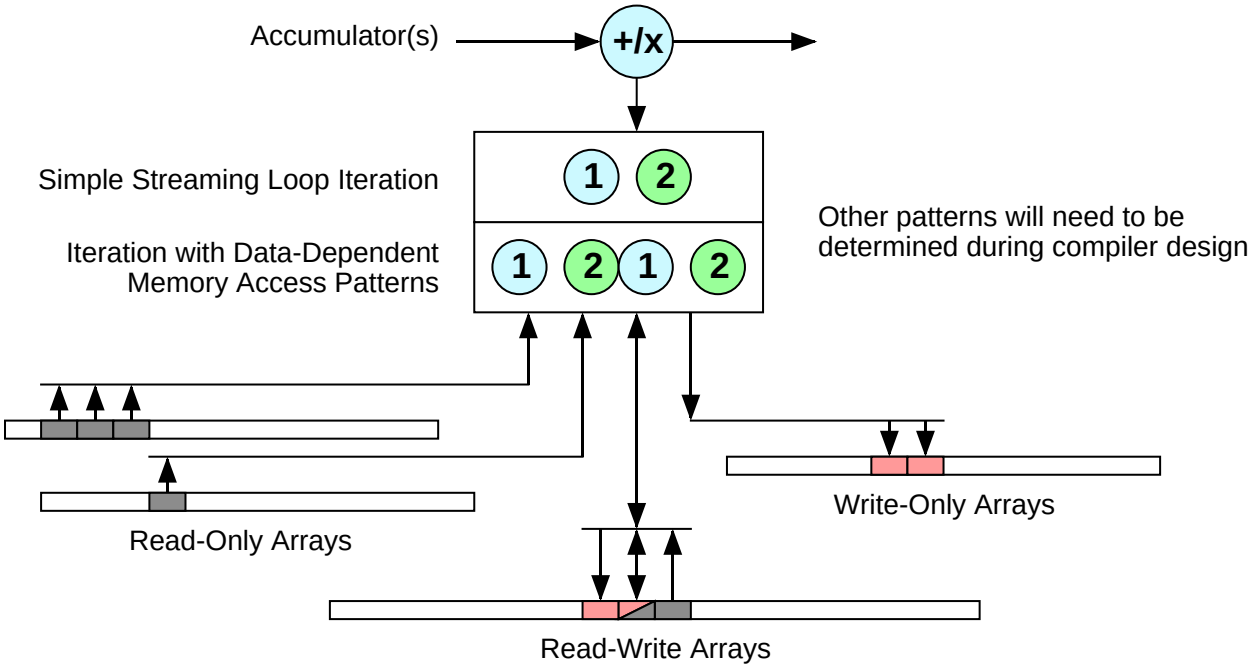

Figure 7: The dataflow in and out of a single loop iteration — or, the smallest "kernel" we can consider. We need to be able to calculate the "footprint" of this dataflow to allow further alias analysis and memory optimization.

First, the patterns of address access should be determined and maintained in some sort of symbolic format — probably as a form of functions of the loop induction variables. The collection of loads and stores made by each iteration should then be transformed into a symbolic main memory "footprint," as is illustrated in Figure 7, along with information on any loads and stores that may require synchronization if they are executed on multiple parallel functional units. A major compiler writing difficulty will be simplifying the symbolic results at each step to keep them manageable. We should also determine the total size of the memory "footprint" for later analysis in memory optimization routines.

Once we have the memory footprint of a single inner loop nest, we may move up the nesting levels one at a time to create the memory footprints at each level of the hierarchy. What was a single access in the base iteration will generally become an entire vector of accesses, a vector will become a 2-D matrix, and so on. This is illustrated in Figure 8. The reason we must maintain "footprints" in a symbolic format is inherent in this process, as each level of the hierarchy will apply a new function to the footprint in order to create the larger composite footprint addressed by the higher level.

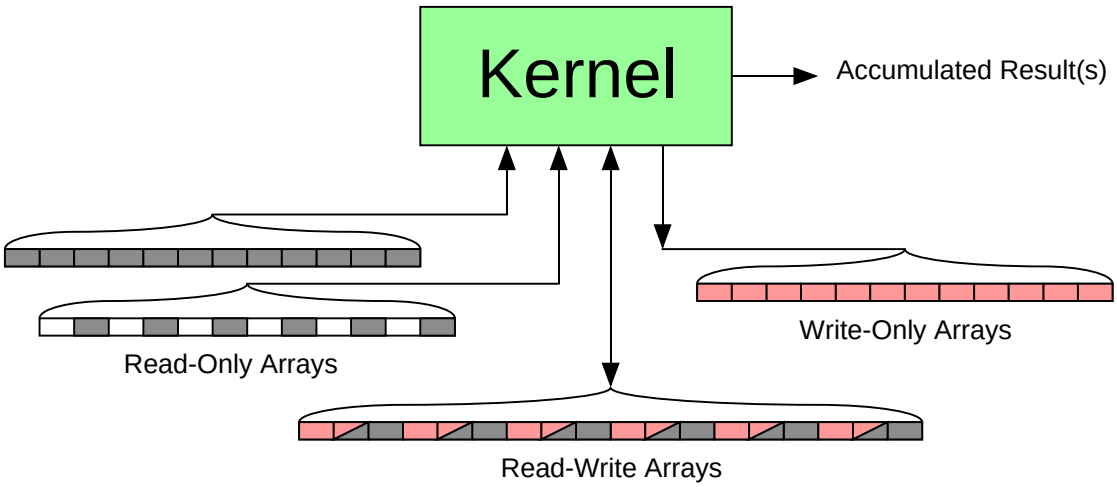

Figure 8: The dataflow in and out of an entire kernel made up of a loop, or several nested loops. This dataflow "footprint" must be determined by building up the footprint from nested kernels.

4.1.4. Parallel Sub-Kernel Optimizations

Once we have memory footprints for the various loops, we need to classify the different levels of the loop hierarchy into serial and parallel regions, as depicted in Figure 9. This mainly consists of examining the memory footprints to see where they overlap or have dependencies from one

loop or loop iteration to another. In these cases, we must mark those portions of the kernel as being serialized. This memory behavior is common in tasks like FFTs or sorts, for example, where multiple "passes" over the same data must be performed. In other cases, the nested components (iterations or inner loops) may be fully parallel. This is common if the outer loop scans over one dimension of input while the inner loop scans over another dimension, for example. From here on out, serialized levels of the nesting tree will be ignored for most optimization purposes. Instead, we can focus on the parallel kernels, which are much more amenable to optimization.

The first potential optimization we can try is loop reordering. If two or more nested levels of loops are all parallel, we can generally interchange them without problems. Before continuing with later optimization, we should actually "generate" all of these reordered versions of the code and use all of them during later analysis. Eventually, some may be discarded as being obviously poor choices, in general because they have memory access patterns that do not lay out well in memory, but it is a reasonable idea to keep multiple versions of the loop until the end of compilation and allow the programmer or profiler to choose from among them.

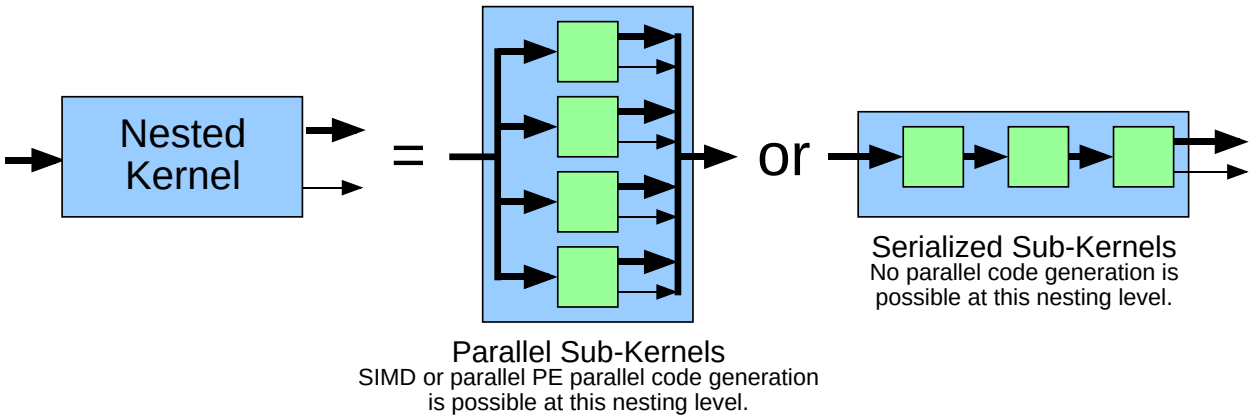

Figure 9: When examining sets of nested kernels, we now need to determine whether or not the sub-kernels are parallel (nested loop iterations work on different elements, such as a different dimension) or serial (nested loop iterations work on same elements, doing serialized steps). We can perform further optimization on parallel kernel stages.

The other major optimization that needs to be applied to parallel kernels is the choice of locations to use parallelism. Most architectures that we will be targeting have parallel execution engines, MMX-style long-word SIMD instructions, or both. As a result, we will generally need to identify the level of the kernel nest that is most applicable to these architectures — or at least be able to generate code for all of the possibilities and let a profiler try them out to see which is the best choice. Once we have a list of possible parallel kernel levels that we can use, it is a simple matter to try and assign one or both forms of parallelism to each one before attempting

the final stages of code generation and optimization. One major decision that must be made, however, is the choice of memory access patterns. Figure 10 illustrates the two major possibilities — interleaved access or division of the input vector into equal-size chunks. Issues of layout in memory, like locality of memory references, and memory word layout, which affects the number of instructions required to pack data into SIMD vector registers, are two key parameters here.

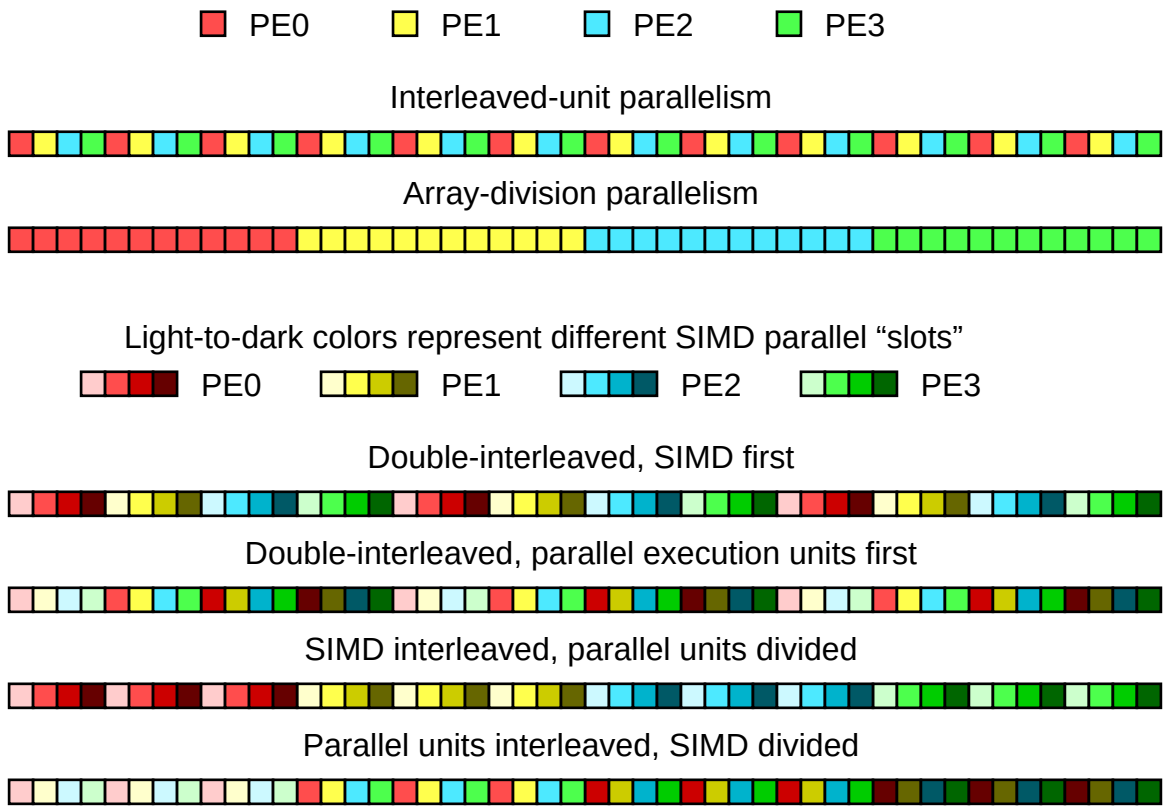

Figure 10: Different ways of dividing up a vector among several parallel functional units or SIMD instruction (MMX-style) processing engines.

4.1.5. Local-Main Memory Allocation

Once we have a broad selection of nested loops to choose from, we need to make a final, critical decision using the memory access footprint of each routine. We need to pick the point at which to break the kernel to allow data to be moved in from off-chip to the on-chip memories. Figure 11 shows the three levels of memory that DSP-C uses conceptually to optimize code. Registers are managed by the code generators, and the main memory by the OS, but the loop optimizing section of the compiler is generally responsible for allocation of local memory. The basic algorithm to choose a break point in the nesting chain is to compare the sizes of the “footprints” at the various levels of the nesting hierarchy to the actual amount of local memory present. We

can climb the hierarchy until the memory required for a loop iteration is too large for the available local memory, and then back down a level. If the footprint of a loop iteration at this level only uses a small fraction of the local memory, then we can merge several of these loop iterations together in an attempt to use more. We may also need to juggle multiple blocks of allocated memory at once. The Imagine project already has a fairly well-tested allocator designed to solve this problem, however.

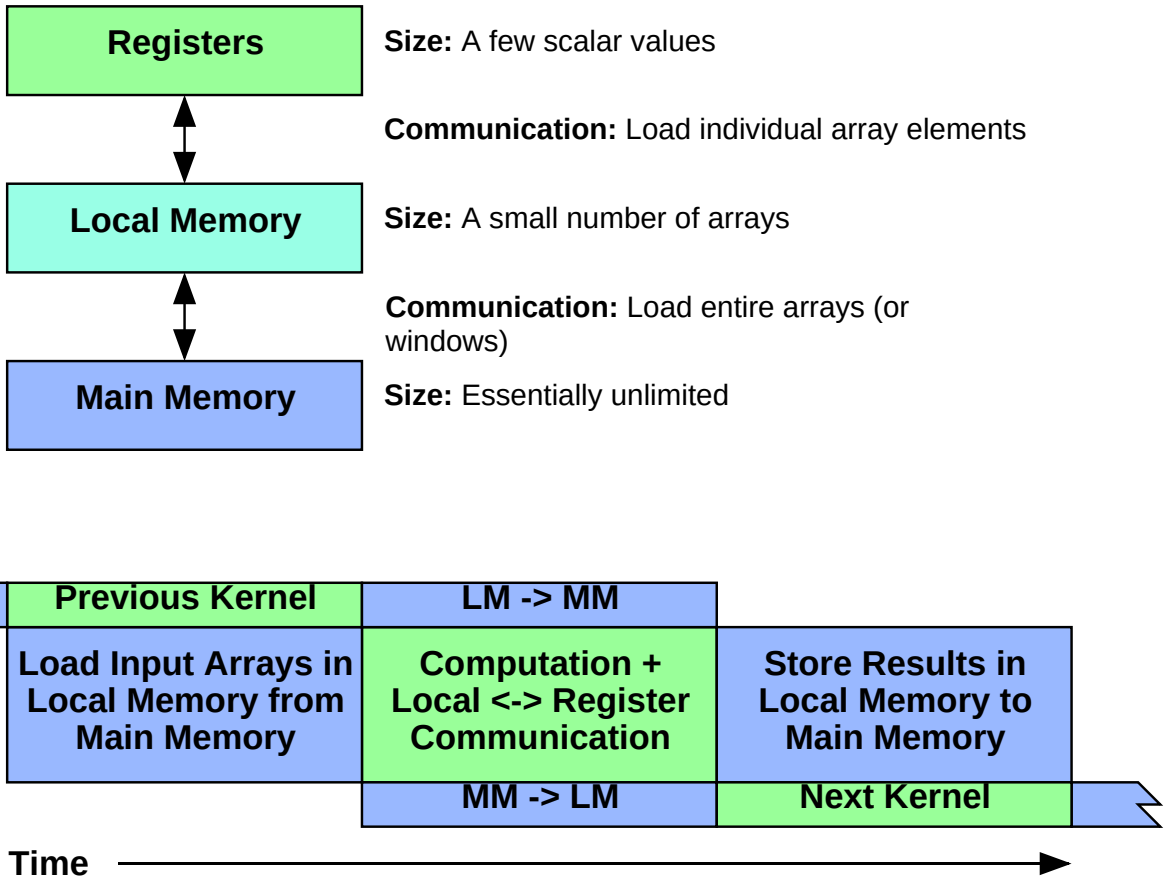

Figure 11: The three levels of memory that we divide DSP architecture memories into for purposes of high-level, architecture-neutral memory optimization. This also shows how one would hope to pipeline memory accesses and computation kernels over the course of time.

Once our loop has been divided into sections that use the local memory efficiently, we can attempt to insert code blocks that control the movement of data to and from main memory. If possible, we would like to have execution of our kernel work under the high-level pipeline defined in the lower portion of Figure 11. We would like to have data streaming into local memory before our computation kernel needs it, and at the same time we want completed results

streaming out to main memory so we can re-use the local memory again. Hopefully, controlling this behavior based on the usage of local memory will work well in most cases, but we may need to perform some extra optimization steps to ensure that our pipeline stages are sized correctly in time as well as in memory usage.

	Generic CPU	DSP Processor	Imagine	VIRAM	Smart Memories
Main to Local	Prefetching or cache misses	Block DMA or "slow loads"	Stream Loads	Vector Loads	Block DMA or "slow loads"
Flexible?	Yes	Maybe	Yes	Yes	Maybe
Local to Registers	Loads	Loads	SRF → Core	(Implicit with inst.)	Loads
Flexible?	Yes	Yes			Yes
Operations	Scalar instructions	Scalar instructions	Scalar kernel ops + comm	Vector instructions	Scalar instructions
Flexible?			Yes, core to core wires		
Registers to Local	Stores	Stores	Core → SRF	(Implicit with inst.)	Stores
Flexible?	Yes	Yes			Yes
Local to Main	(Implicit, as cache lines evicted)	Block DMA or "slow stores"	Stream Stores	Vector Stores	Block DMA or "slow stores"
Flexible?		Maybe	Yes	Yes	Maybe

Table 3: A comparison of the different levels of memory and their intercommunication capabilities present in several DSP architectures of interest. The flexibility lines indicate where odd memory addressing or inter-processing element communication may occur.

One other potential headache to deal with during memory allocation is when our algorithm requires unusual memory access patterns. On some architectures, we have the flexibility to do unusual addressing at virtually any level of the memory hierarchy. On others, however, we may be forced to do this addressing only at either the main or the local memory hierarchy interfaces, but not both. Table 3 examines the features of several architectures that we might consider targeting, to provide a viewpoint about how much of an issue this might be. Most offer the ability to use flexible access patterns at multiple interfaces, but neither Imagine nor VIRAM do. Because the speed of the main memory interface is generally *much* slower than the speed of the

local memory interface, the need to schedule odd memory access patterns so that they may be performed by the main memory interface without slowing the kernel may come to dominate all other parallel loop scheduling issues completely on these architectures. VIRAM sidesteps the problem primarily by keeping its main memory on-chip, so the main memory interface is not such a bottleneck. Imagine uses some core-to-core communication that is probably not particularly compiler-friendly. This is an issue that we will need to explore in the future.

4.2. Constrained Access Arrays (CAAs)

The main tool used by DSP-C to enable vectorization of loops in C is the addition of the more sophisticated *constrained access array* (or CAA) type. In essence, a CAA is just a standard array that does not allow any C pointers to point at its internal contents, so artificial aliasing problems just disappear, leaving only the true potential dependencies between accesses to the same array as possible locations for aliasing. This just patches the basic C problem in that there is no way for a programmer to *express* non-aliasing arrays, and thereby tell the compiler when optimizations are safe. This is important because compilers must generate correct code first, and fast code second. Once we simplify the aliasing problem with CAAs, verifying correctness is a much easier task for the compiler, and so building a fast, vectorizing compiler is also much easier. In addition to much simpler alias analysis, the hidden internal structures of the CAAs allow compilers to vary the memory allocation of the arrays themselves without running the risk of breaking code. For example, CAAs may be allocated as column-major, row-major, or actually alternate over the course of a program's execution.

It is possible to use CAAs simply as unaliasable and flexible arrays, but they also have other features. Direct multidimensional array support, internal pointers, range checking, scatter/gather operations, subranges within arrays, and explicitly parallel operations are all straightforward extensions of the basic CAA definition. The use of these features is the topic of the remainder of this chapter.

4.2.1. Syntax & Usage

Because there are so many aspects to CAA use, they have been listed in several subsections. These are arranged in an approximate fashion from the mundane to the exotic.

```
int j [[1024,2]]{
      port input[[circular, circular]];
                        // Port with addressing options
      port output;
      range critical[[,circular]];   // Defined range
      optimize_speed;          // Compiler hint (not requirement!)
```

```
};
. . .
// Basic CAA access and use
a = j[i,j] + 1;               // Direct use
b = j.input;                  // Access via port
c = j.output[[+16, ++]];            // Port use + move
j.input[[++, –]];             // Port move
. . .
// Explicitly ranged CAA access and use
j[0:1023,1] += 1;                   // Direct use
j.critical += 1;                    // Predefined range
j.critical[[++:–,]] += 1;     // Range use + move
j.critical[[+16:+16,++]];     // Range move
```

4.2.1.1. Basic definition

Simple CAAs are defined in a manner almost like traditional C arrays. The only difference is that CAAs always use doubled `[[ ]]` symbols instead of `[ ]` ones. For example, the following two arrays are essentially the same, except the second is a CAA:

```
int w[10];
int x[[10]];
```

Multidimensional CAAs are defined in a manner slightly differently from C. While C's technique of tacking on additional array dimensions as extra, independent arrays is workable hack, it makes more sense to have multidimensional arrays be *explicitly* multidimensional, from the definition forward. Hence, the following two definitions may be used to declare 3-D arrays in DSP-C that are functionally equivalent, but the structure actually produced by the CAA may be somewhat different.

```
int y[10][10][10];
int z[[10,10,10]];
```

Most of the more complex usages of CAAs involve adding features or options to them. In order to allow this, one must be able to add lines of code that will be included by the compiler into the basic definition of the CAA. This is accomplished using an optional `struct`-like set of braces at the end of the CAA definition, between the `]]` and the trailing semicolon. If no additional options are needed, these braces may be omitted.

```
int y[[10]]
{
      option1;
      option2;
```

```
      // Additional options can go here
};
```

Much like the elements of a `struct`, any options that can be used by a programmer directly in the course of using the CAA are accessed using the `.` operator. The specific use of each option will be described along with the option itself.

4.2.1.2. Simple Usage

Most usage of CAAs will be in a fairly traditional, array-like pattern. The following are legal uses of the `x` and `z` CAAs defined previously. When on the right-hand-side of an expression, the `[[]]` operator returns the value of the array named to its left, while on the left-hand-side of the expression it serves to store a value into the array. Multiple dimensions are specified by placing commas between the coordinates of each dimension.

```
value = x[[0]];
x[[4]] = 5;
value = z[[1,2,4]];
z[[9,3,0]] = 23;
```

CAAs are also "self-aware" about how large they are and how many dimensions they contain, unlike standard C arrays. This allows the more Java-like feature of array range checking to be easily implemented. Compile-time range and dimension checking is always enabled, and would easily catch all of the following mistakes made with the previous arrays. As our compiler develops, it should be able to catch many possible out-of-range cases that may occur when a loop would go one element off of the end of the array, for example.

```
value = x[[-1]];
value = z[[1,10,4]];
value = x[[0,2]];
value = z[[1,2]];
```

Range checking may also occur at runtime, to catch all possible range errors, but this is an option that will generally only be enabled during the debugging process. It is turned on and off using the basic DSP-C header comment, as was described in the first chapter.

4.2.1.3. Pointers and CAAs

The main feature of CAAs is that they prevent pointer aliasing to their contents by simply forbidding C pointers that may try to point inside them. Practically speaking, this affects the function of the unary `&` (address) operator in C. While it's perfectly legal to determine the

address of a normal C array element, this is absolutely forbidden with CAAs. Thus, one of the following lines is legal, but the other will cause a compile-time error.

```
int* ptr;

ptr = &(w[3]);          // Legal!
ptr = &(x[[3]]);        // Error!
```

While it is not possible to aim a pointer at an element of a CAA, it is legal to point a CAA pointer at a CAA itself. These are defined just like a pointer that could be aimed inside the CAA, but with an additional `[[]]` at the end. When CAAs are assigned to a pointer, the pointer just becomes another way to refer to the entire CAA. It may **not** be dereferenced in any way.

```
int *ptr;                           // int pointer
int *caa_ptr[[]];                   // Pointer to CAA of int

ptr = w;                // Legal!
ptr = x;                // Error: Can't assign CAA to non-CAA pointer
caa_ptr = x;            // Legal!
value = *ptr;           // Legal!
value = *caa_ptr;       // Error: Can't dereference CAA pointer
value = caa_ptr[[0]];   // Legal: caa_ptr is now equivalent to x
```

The size and dimensionality of a CAA do not need to be included in the definition, since they are associated permanently with each particular array. Options may be included with the pointer, but they are not essential. However, if they are not included, then they cannot be used. CAAs may be freely assigned to any CAA pointer that has a subset of their capabilities without casting, including to `void *[[]]`; pointers, but not to pointers with different or extra capabilities. When a CAA is assigned to a pointer with a greater set of capabilities, it is actively checked to make sure that the assignment is correct. This checking may even be done at runtime, when CAA range checking is enabled, if fully static analysis of the upwards assignment is impossible. Any non-CAA pointers, including fully generic `void` pointers, may **not** be used to hold CAAs, since the proper type checking is not supplied by the compiler with any of those types.

```
void *any_ptr;                      // Generic pointer
void *any_caa[[]];                  // Generic CAA pointer
int *caa_ptr_long1[[]]{ option1; }; // Pointer to CAA of int w/"option1"
int *x1[[10]]{ option1; };          // Version of x w/ option1
int *x2[[10]]{ option2; };          // Version of x w/ option2

any_ptr = x;            // Error: Must use CAA pointer
caa_ptr = x1;           // Legal: caa_ptr is subset of x1
caa_ptr_long1 = x;      // Error: Pointer has extra capabilities
caa_ptr_long1 = x1;     // Legal: Full match
```

```
caa_ptr_long1 = x2;     // Error: Wrong capabilities
any_caa = x1;           // Legal: Void CAA is always a subset
caa_ptr_long1 = any_caa;// Legal: Statically provable to be correct
any_caa = x2;           // Legal!
caa_ptr_long1 = any_caa;// Error: Statically has wrong capabilities
```

The different varieties of CAA pointers, and their potential uses, are summarized in Table 4 to provide a handy reference.

Type of CAA Pointer	Example	Pass reference to the CAA?	Perform [[]] access?	Use options?
Void	`void *ptr[[]];`	YES	—	—
No options	`int *ptr[[]];`	YES	YES	—
Full definition	`int *ptr[[]] { option };`	YES	YES	YES

Table 4: Possible pointers that may be used with a full CAA.

CAA pointers are used to pass CAAs to functions, and follow the same rules there. The input to a function will match a prototype that corresponds to the entire CAA definition, or just a subset of it. The following examples illustrate the various legal and illegal options.

```
void function1(void *input[[]]);
void function2(int *input[[]]);
void function3(int *input[[]]{ option1; });

function1(x);     // Legal: Void CAA is always a subset
function1(x1);    // Legal!
function2(x);     // Legal!
function2(x1);    // Legal!
function3(x);     // Error: Prototype pointer has extra capabilities
function3(x1);    // Legal!
function3(x2);    // Error: Wrong capabilities in the prototype
```

If the function chooses to address the received CAA as a simple array type, that is acceptable usage. However, it will not have access to any of the CAA's additional options in this case. If access to extra options is necessary, then the CAA must be passed using its full type specification, options and all. There is no "in between" option — additional options are an all-or-none proposition. In general, however, this limitation should prove to be of little difficulty. CAAs are generally passed as simple arrays to library routines, while functions that may need to access special features will probably be able to share a `typedef` of the full CAA with the caller.

4.2.1.4. Size Utility

Because CAAs maintain their own size and dimensionality internally, it is possible for an operator to return these figures at runtime. These simple operators perform the obvious functions. The basic `length` operator returns the length of a 1-D array or the first dimension of a multidimensional array. For multidimensional arrays, the `dimensions` operator will usually have to be used first to determine the number of dimensions in a multi-dimensional array before the `dim_length` operator can be used on each dimension in turn.

```
result_as_int = dsp_length(CAA)
result_as_int = dsp_dimensions(CAA)
result_as_int = dsp_dim_length(CAA, dimension)
```

The use of these operators can allow library code to work with varying array sizes without problems and without the need to pass lots of extraneous "size" integers around.

4.2.1.5. Ports

Sometimes it is just not possible to completely eliminate pointers to array elements from code. Occasionally, it is just a much clearer way of writing programs. More frequently, it is a result of programmers who have learned to prefer pointer address arithmetic over the course of years of programming. To aid these programmers (or programmers who have to port code they may have written), CAAs support pointers to their elements as an option that may be attached to each CAA. Because each port is associated with a particular CAA, references cannot cause aliasing any more than standard array references can (in fact, they may be translated into standard array references trivially).

A port is defined in the options section associated with a CAA according to the syntax below. A pair of examples follow, for reference.

```
port name <<[[dim0_options << , dim1_options . . . >>]]>>;
port hot;                       // Simple port
port cold[[circular]];          // With increment specifiers
port warm[[circular, , bit_reversed circular]];
```

Each port is a simple pointer in to the array, but it may be moved in different ways using the specifiers in the `[[]]` fields. As is shown, these specifiers may be applied to each dimension of a CAA independently using comma separators. The default is for the pointer to increment and decrement like any address pointer, up or down by a number of array elements requested. If the pointer is moved off of an end of the array, then a range check error is declared. As an alternative, the `circular` mode modifies this behavior so that the port "wraps around" to the other end of the array when it runs off of the end. This is a helpful mode in applications where

circular buffers must be implemented, for example. The `bit_reversed` mode makes any increments or decrements occur with bit reversed arithmetic — the carry chain in the adder performing the increment or decrement is effectively connected backwards, so carries propagate down towards the least significant bits. This is useful when doing FFT address arithmetic, for example, but it should be used with caution on code targeted for architectures without native support for bit reversal. In general, whenever `bit_reversed` mode is used, it will be paired with `circular` mode by putting them together, as is shown above.

Once ports are defined, they are used by using their name attached to the name of the base CAA using the standard `struct .` operator. Each use returns or writes a value from or to the CAA depending upon whether the port reference is on the left hand or right hand side of an expression. Either the element pointed at by the port or one at an offset from the port's current location is read or written. As a side effect, the port may be moved by the operation. The following syntax functions to read elements from the array and/or adjust the location of the port.

```
element = caa.port;
caa.port = value;
element = caa.port[[<<expression>><<, expression, . . . >>]];
caa.port[[<<expression>><<, expression, . . . >>]] = value;
```

Expression	Reference Is	Port Adjustment	Interpretation
expr	port + *expr*	—	Basic read/write offset
= *expr*	*expr*	*expr*	Sets port to value given (initialize)
^^ *expr*	port + *expr*	port + *expr*	Pre-increment of port location
expr ^^	port	port + *expr*	Post-increment of port location
^^	port	port + 1	Port++

Table 5: Summary of possible expression types in a port adjustment expression.

The basic form of the port access just returns or writes the current element of the CAA pointed at by the port. When the `[[]]` extension is added, one or more expressions may be used to modify this default behavior and/or move the port. Each `[[]]` field must contain N-1 commas, where N is the number of dimensions in the CAA. The expressions contained between the commas then affect each dimension, in turn. The different varieties of expressions affect the reference and the port in ways described in Table 5. A few sample references are given below.

```
value = x.hot;                  // Simple reference
x.hot = value;                  // Simple write
x.hot[[=0, =0]];                // 2-D initialization
value = x.hot[[2,-2]];          // Offset read
```

```
value = x.hot[[2^^,]];         // Post-increment dim #0 by 2
value = x.hot[[^^, ^^]];       // Double post-increment
```

The address operator (&) may also be applied to ports, but how the results may be used is very constrained by the compiler to keep a semantic "firewall" maintained between different CAAs. The primary reason to use the address operator is to assign one port (or an offset form of it) to another. This only works if both ports are assigned to the same CAA. Attempts at assigning ports to non-CAA pointers or ports to other CAAs will result in compile-time errors. Also, if code must recover the port location in numeric form for further calculation, then the address operator permits this, as it overrides the normal port behavior of dereferencing itself. Function-style operators can then be used to extract each dimension of the port address in integer form separately.

```
port_address = &caa.port                              // Assignments
caa.port = port_address
position_as_int = dsp_port(port_address, dimension)   // Operator

x.cold = &x.hot;               // Assign port x.hot to x.cold, not values
value_0 = dsp_port(&x.hot,0);  // Operators to find where port is
value_1 = dsp_port(&x.hot,1);
```

One final bit of syntax that may be used to define ports is the temporary port declaration. While most ports are defined to be a permanent part of any particular CAA, it is also possible to define temporary ports, usually for CAAs that have been passed in to a library function. These are defined along with any other temporary variables at the head of a function, with the name of the CAA added to the normal definition form between the keyword `dsp_port` (the additional `dsp_` tag is added because this line is out in the middle of a normal routine) and the port's identifier. For example, the following example makes a temporary port for its input `x`.

```
void foo(int *x[[]])
{
      dsp_port x my_port;

      x.my_port[[=0]] = 1;
      . . .
}
```

It is possible to declare a temporary port in any routine after a CAA has been declared, but the most frequent use of this form will be for new ports added to input arrays since otherwise it is usually easier to just add another permanent port to the CAA.

4.2.1.6. Array Ranges

Another common way to access a CAA is through an array range, or subarray. This is an option that may be added to a CAA, much like a port. Each range specifies a block of elements within the array by specifying two opposite "corners" of the block, a special port in the middle of the block that is used to set an origin for calculations using the range, and a stride for indicating how size of steps that are taken in any particular direction within the range.

The definition and use of ranges are *identical* to ports, except that each dimension of the range is associated with a set of four numbers, instead of just one. This quartet of numbers, divided by : operators, replaces each of the single expressions in the definition and use of a port. The first number specifies a port at the corner of the range closest to 0 in each dimension, or the "bottom" of the range. The second specifies a port at the corner of the range farthest from 0, the "top" of the range. The third represents an actual port representing the "origin" position within the range, which can be used in some calculations. Finally, the fourth number in each set represents the stride used with ranges for calculations and adjustments of ports or sub-ranges within the range. It should be noted that the stride has no effect on the adjustment of the other three basic ports making up the range, instead affecting only operations on parallel operations with the range and options of the range itself, such as range ports or sub-ranges. Also, the stride can be anything between ± range size in a particular dimension (it is the only range control value that is signed).

The syntax and a few definitions and uses are given below. When fewer than three colons are present in any particular dimensional definition or expression field, the rightmost colons are assumed. Because of the larger amount of information maintained by a range, its range address operators are somewhat more complex. They are simplified somewhat by returning ports as intermediate values, so the port operators may then be used to extract particular bits of information. Like usual, dimensions are numbered starting at zero.

```
// Basic definitions, within CAA option blocks
range name <<[[dim0_options << , dim1_options . . . >>]]>><<{options}>>;
w h e r e                  d i m X _ o p t i o n s                  =
      bottom_options:top_options<<:start_options<<:stride_options>> >>
range hot;                          // Simple range
range cold[[circular::circular:]];  // With increment specifiers
range warm[[circular:, , ::bit_reversed circular]]{ port baby; };
                                    // With increment specifiers & a port

// Temporary definitions
dsp_range x temp;                   // Temporary simple range

// Basic range adjustment
caa.range<<[[expression:expression<<:expression<<:expression>>>>
      <<, expression, . . . >>]]>>
```

```
x.hot[[=0:=127:=0:=1, =0:=127:=0:=1]]      // 2-D initialization
x.hot[[2^^:-2^^:2^^, 2^^:-2^^:2^^]]       // Shrink in by 2 on each side
x.hot[[::10^^, ::10^^]]                    // Move "current" port

// Port range adjustment (cannot affect stride, as ports are unsigned)
caa.range[[&port_bottom:&port_top<<:&port_origin>>>>]]
x.hot[[&x.lower_edge:&x.upper_edge:&x.middle]];

// Range address operators

range_address = &caa.range                          // Assignments
caa.range = range_address
bottom_as_port = dsp_range_bottom(range_address)    // Operators
top_as_port = dsp_range_top(range_address)
origin_as_port = dsp_range_origin(range_address)
stride_as_int = dsp_range_stride(range_address, dimension)

x.hot = &x.cold;                           // Range-to-range assignment
x.port = dsp_range_bottom(&x.cold);        // Portion extraction
value = dsp_range_stride(&x.cold, 0);      // Get first dimension
```

Since each range represents more than one value, no scalar value can be returned from a range adjustment operation, unlike port adjustments. However, range-to-range operations are possible, and are described in the section on explicitly parallel options. It should also be noted that like ports, ranges may be declared as temporaries, using an equivalent syntax.

There are a few other features of ranges. Since ranges are effectively small CAAs themselves, they may also have full sets of options, just like a standard CAA, including ports and sub-ranges. For example, a port to the range `warm` was declared above. All ports or sub-ranges work within the current coordinate space of the range. A port in a 2-D range at [[0,0]] is actually pointing into the original CAA at the "origin" of the range, while a port at [[1,1]] is pointing at location (origin_x + stride_x, origin_y + stride_y). Points in the range that are "below" the origin can be accessed by using negative offsets. It is possible to use this technique to provide fully remapped coordinate systems within a CAA. For example, the following code remaps a CAA to be used backwards (with all dimensions running in reverse) and with a –64 to +63 coordinate system instead of the base 0 to 127 system.

```
// Definition
int buffer[[128,128]]
{
      range rev
      {
            port rev;
      };
      range mid
      {
```

```
            port origin;
      };
};

// Initialization
buffer.rev[[=0:=127:=127:=-1, =0:=127:=127:=-1]];
buffer.rev.rev[[=0, =0]];
buffer.mid[[=0:=127:=64:=1, =0:=127:=64:=1]];
buffer.mid.origin[[=0, =0]];

// Now buffer.rev.rev = reversed array
if (buffer[[0,0]] == buffer.rev.rev[[127,127]])
      printf("I'll always equal TRUE\n");
// Now buffer.mid.origin = re-started array
if (buffer[[65,65]] == buffer.mid.origin[[1,1]])
      printf("I'll always equal TRUE\n");
```

While array ranges can be used for remapping of array coordinates like this, it is not a primary consideration. Instead, their primary use is in providing input to explicitly parallel functions, which are described in more detail below.

4.2.1.7. Port Arrays

The "evil twin" of ranges are port arrays. These are CAA options which superficially resemble ranges, since they act identically in many ways. The declarations and uses are quite similar, but the internal workings of port arrays are much different. Instead of mapping contiguous or strided blocks of CAAs, they are blocks of ports, each of which can point anywhere in the base CAA. These are mostly used to gather together elements in a sparse CAA for processing in dense fashion. They can also be used to provide a completely arbitrary remapping of a CAA to a different namespace. For compilers, the use of a port array is a serious red flag that the code they are about to optimize probably uses unusual memory access patterns, which may be difficult on some architectures.

A port array is defined in the options section associated with a CAA according to the syntax below. A pair of examples follow, for reference. The number of dimensions in the port array definition must match the number of dimensions in the base CAA.

```
port_array name[[dim0_size << , dim1_size . . . >>]];
port_array hot[[10,20]];                  // Simple port array
port_array cold[[10,20]]{ port mama; };   // Port array with sub-port

dsp_port_array x temp[[5,10]];// Temporary port, outside options block
```

Once it is defined, the port array is a CAA itself, attached to the base CAA, containing an array of limited-function ports. These ports are "limited-function" in that they can only be set, and not incremented (although we could loosen up this limitation in the future, if incrementing of port array pointers becomes necessary, by adapting the pre-increment operation). Elements are set using the port "&" operation in an assignment, as elements of a port array are legal ports to the CAA themselves. Hence, the following are legal ways to load array references into a port array.

```
x.hot[[3,2]] = &x[[34,120]];              // Constant assign
x.hot[[x,y]] = &x[[x*x+y,y*y+x]];         // Variable mapping
x.hot[[3,2]] = &x.port;                   // Port-to-constant mapping
x.cold.mama = &x[[34,120]];               // Constant-to-port mapping
x.cold.mama = &x.port;                    // Port-to-port mapping
```

Values may also be read out of a port array into normal ports for a CAA, allowing the port array to be examined at a later time.

```
x.port = x.hot[[3,2]];              // Constant readout
x.port = x.cold.mama;               // Port-to-port readout
```

Once a mapping has been loaded into a port array, it may be used just like a range for purposes of sub-ports, sub-ranges, and parallel operations. It should be noted, however, that all of these operations will probably be slower with a port array than they would be with a range, so the use of port arrays should be strictly limited to instances when data actually needs to be scattered to or gathered from a CAA in an arbitrary way that a range can't handle.

4.2.1.8. Architecture Specific Optimizations

In their option blocks, CAAs may also specify flags that try to indicate to the compiler good potential tricks for optimization. These are specified by "options" that are simply a single identifier or an identifier and a value, separated by an equals sign. These are all possible legal optimization flags.

```
optimize_speed;
mips_allocate_local;
imagine_register = 34;
```

Architecture-specific options should probably start with some sort of architectural specifier, as in the examples. The compiler will try to parse any optimizations presented and use any that it understands to help compile the code. If an optimization is unrecognized or cannot be handled safely, however, then it will be ignored, as these are simply "hints" to the compiler.

4.2.2. Parallelism Limitations

There are many types of code that will make code difficult or impossible to vectorize and/or parallelize, even with the replacement of C arrays by CAAs. Code which uses too many C pointers for non-array tasks may still be too difficult to analyze. Code which uses a significant amount of control flow in `if` statements also might prove intractable, as predication can only "parallelize" control flow within certain reasonable limits. Code which is written in an inherently serial manner, with a continuous stream of dependent data, will naturally prove difficult to automatically attack.

Most of these are insurmountable problems that we will only be able to handle by setting rules on how loops may be coded in DSP-C. Aside from a potential ban on `if` statements, the exact limitations that we will need to impose will have to be determined during the course of compiler and application development. There are several problems that should be surmountable, however, by a properly equipped compiler. This section attempts to define these critical areas for future compiler development.

4.2.2.1. Aliasing Analysis

CAAs completely solve memory aliasing problems as long as one happens to only use arrays exclusively either for reading or writing within a loop that has been targeted for vectorization or parallelization. However, when some arrays are targeted for both tasks full alias analysis as described at the beginning of this chapter will need to be performed. However, the task is made tractable by the fact that only the set of memory references to each single array needs to be considered, and not cross-array references.

Among the different read and write references in an array, we will need to be able to determine the memory "footprint" of each iteration, using symbolic calculations, as was already shown in FIGURE. This will then need to be hierarchically built up into a full-loop memory footprint for each nesting level of the loop, as shown in FIGURE, to allow alias analysis and checking at each level. Hopefully, we should be able to leverage technology in existing vectorizing and parallelizing compilers to handle a fair amount of this work.

4.2.2.2. Runtime Alias Checking

After a function call returns and at the beginning of functions, we may have to insert code to handle some of the alias analysis at runtime. When our static aliasing analysis indicates that there is a potential for aliasing — or not — depending upon the values of variables at the beginning of a function or after potential changes during a function call, the compiler should insert code to check and see if the potential aliasing actually will occur at runtime. In general,

the collection of variables which may have an impact upon aliasing behavior is fairly small (on a few integers used later to select elements from CAAs and possibly a port or two), so these checks shouldn't be time-consuming. If our runtime checks determine that aliasing will not occur, then we can use optimized loop code. On the other hand, if the aliasing will occur, then slower, more serial code can be executed instead. The following simple example shows how anti-aliasing runtime code could be helpful.

```
void my_routine(int *in1[[]], int *in2[[]], int offset)
{
      // RUNTIME ALIASING TEST HERE
      if ((offset <= 0) || (offset >= dsp_length(in1))
            // Then use optimized routine . . . .

      // Original user code
      for(i=0; i < dsp_length(in1); i++)
            in1[[i+offset]] = in1[i] + in2[i];
}
```

In a real system, it would be helpful if the compiler could determine when this sort of code is needed and insert it automatically when necessary. Also, it would be helpful to have the compiler actually compare the speeds of the different routines that it may be inserting. If the real speedup from the "faster" routine is trivial, then it may be better to simply cut the tests and forget the whole thing. Sometimes, however, it will be dramatically faster to use an optimized loop, so this concept should not be completely discarded.

4.2.2.3. Reduction Operations

One important family of data-dependent operations that we will need to support well are the various reduction operations. Many DSP algorithms reduce their input to a scalar or an array with fewer dimensions (which is actually several parallel reductions). Reductions are often coded in a serial manner with some form of loop-carried variable acting as the reduction product accumulator. If we do not optimize these data dependencies well, we will probably severely limit our potential for finding parallelism. Provided that the *reduction operation* used to combine each array element's result with the accumulator is associative, we can simply split the reduction into several parts, execute them in parallel, and combine them at the end to get the same result as the original, serial algorithm. If the reduction operation is commutative as well, we may even completely rearrange the reduction, possibly by interleaving elements. For example, the following formulas demonstrate a sum reduction rearranged for parallel operation by a divided array (requires associative only) or an interleaved array (requires both associativity and commutativity). The bold additions indicate where we need to communicate between the parallel elements to combine sub-sums from the different processing elements into a single sum.

Serial: 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15
Divided: (0 + 1 + 2 + 3) **+** (4 + 5 + 6 + 7) **+** (8 + 9 + 10 + 11) **+** (12 + 13 + 14 + 15)
Interleaved: (0 + 4 + 8 + 12) **+** (1 + 5 + 9 + 13) **+** (2 + 6 + 10 + 14) **+** (3 + 7 + 11 + 15)

Several existing operators in DSP-C can be used as reduction operators. The simple add and multiply operators are probably the most important ones. Saturating adds and the minimum/maximum operators added in the previous chapter are also targets for reduction optimization. Table 6 details how they work to break down the reduction process using three different kinds of steps. Serial reduction steps perform the same job that the original serial algorithm did — sweeping down an array serially, element by element. Divided parallel reduction step combine the results of parallel operations on divided arrays (the bold additions in the "divided" example above), while interleaved parallel reduction steps combine the results of parallel operations on interleaved arrays (the bold additions in the "interleaved" example above). Because interleaved parallelism requires commutativity, the interleaved step is rather complex for the normally non-commutative operators that find the location of maxima or minima.

Operation	Comm? / Assoc?	Serial Reduction	Divided Parallel Reduction Step	Interleaved Parallel Reduction Step
+	Yes/Yes	A + B	A + B	A + B
^+	Yes/Yes	A ^+ B	A ^+ B	A ^+ B
*	Yes/Yes	A * B	A * B	A * B
Minimum	Yes/Yes	min(A,B)	min(A,B)	min(A,B)
Maximum	Yes/Yes	max(A,B)	max(A,B)	max(A,B)
Where first minimum?	No/Yes	(B < A ? B, &B : A, &A)	(B < A ? B, &B : A, &A)	(A = B ? (&B < &A ? B, &B : A, &A) : (B < A ? B, &B : A, &A))
Where first maximum?	No/Yes	(B > A ? B, &B : A, &A)	(B > A ? B, &B : A, &A)	(A = B ? (&B < &A ? B, &B : A, &A) : (B > A ? B, &B : A, &A))

Table 6: Summary of reduction operations built into DSP-C. Note the complex interleaved parallel reduction step required to overcome the lack of commutativity in the first minimum and maximum functions. The accumulated product of earlier reductions is always contained in A, while B represents the next input.

Because reduction optimizations have been done already in many parallelizing compilers such as SUIF, we hope to be able to leverage existing work for most of the analysis needed to find and optimize reduction variables in our codes. However, it may prove necessary to establish a

protocol of explicit accumulators to help identify reductions — a pre-declaration of variables as accumulators so that we know optimizations must be performed on them.

Of course, another policy is to just avoid parallel reductions as much as possible. Whenever a reduction is encountered, it's not a bad idea to simply avoid parallelizing at that particular loop nesting level if there are any other nearby nesting levels that are somewhat reasonable targets. If the reduction operation is not from the family in Table 6, this may be the only possible policy to follow.

4.2.3. Examples

The following example takes the FIR filter example from the chapter on DSP integers and turns its arrays into CAAs, instead. Most of the code is the same, but the use of the circularly-addressed port pointer `input.i` allows the pair of loops in the original code to be transformed into a single loop. On architectures without hardware support for circular addressing, this will be transformed back into a pair of loops, but in any case it is easier to read this way without the artificial loop breaks caused by addressing requirements.

```
int input((24,23))[[128]]{ port i[[circular]]; };
int coeff((24,23))[[128]], i;

// Initialize ports & input
input.i[[=0]];
for (i=0; i<128; i++) input[[i]] = 0;
// Later, initialize coefficients, themselves

// Routine called once for each input
int((24,23)) fir_filter(int new_data((24,23)))
{
      int a((56,47));         // Extended-precision accumulator
      int j;

      a = 0;
      input.i[[^^]] = new_data;                  // Record input
      for (j=0; j < 128; j++)                    // Filter loop
            a += input.i[[^^]] * coeff[[j]];     // Reduces
      fir_filter = dsp_round_to_nearest(a / 128);
}
```

The following 2-dimensional filter shows how multidimensional CAAs work — pretty much just like standard ones, except that now we can guarantee to the compiler that no aliasing may occur between the various array references.

```
int input((8,0))[[640,480]];
```

```
int output((8,0))[[640,480]];
int coeff((16,15))[[9,9]];

// Assume that the main program sets up inputs & coefficients

// Routine called once for each frame
void video_filter(void)
{
      int a((32,15));         // Extended-precision accumulator
      int x, y, f_x, f_y;

      for (x=4; x <= 635; x++)            // Pixel loops
      for (y=4; y <= 475; y++)            // Pixel loops
      {
            a = 0;
            for (f_x=-4; f_x <= 4; f_x++)       // Filter loops
            for (f_y=-4; f_y <= 4; f_y++)
                  a += input[[x+f_x, y+f_y]] * coeff[[f_x+4, f_y+4]];
            output[[i,j]] = dsp_round_to_nearest(a / (9*9));
      }

      // More code will be needed to handle the edges, but
      //   this has been omitted for brevity's sake.
}
```

These examples only scratch the surface of the ways in which CAAs can be used. We will no doubt need to refine the CAA interface and capabilities after coding many more examples up in different programmers' coding styles and targeting different hardware.

4.3. Explicitly Parallel Operations

If standard C loops performed using CAAs as input and output arrays are not enough for a compiler to find and extract the necessary parallelism, then DSP-C also offers much more explicit parallel operations. These are performed simply by specifying computations with entire ranges of input in place of the normal scalar inputs. Parallel operations may then be performed across the entire range of inputs to the parallel "kernel" of code simultaneously.

In order to facilitate the writing of code in kernels using ranges as inputs, the `dsp_for` loop structure is introduced here. It acts much like a normal `for` loop, except that it takes a series of ranges as input and feeds them in to the body of the loop on an element-by-element basis. This allows the body of loop to be written much as a standard scalar loop might be written, even though it is performing a considerably different operation.

4.3.1. Syntax & Usage

The syntax of an explicitly parallel operation can vary somewhat, depending upon how long the parallel kernel is. A "single-line" operation may be encoded just like a normal scalar operation, but with one or more ranges as input and/or output to the line. On the other hand, a "multi-line" operation, with the `dsp_for` keyword, acts much like an entire normal loop in C. It is up to the programmer to choose which form to use, based primarily on the complexity of the operation that he or she may be attempting to encode.

This section also details immediate use ranges, which are a simple way to specify a range just as it is needed for use by an explicitly parallel operation. In practice, programmers will probably specify ranges in this way as much or more often than by using the fully named ranges described in the previous section.

4.3.1.1. Immediate Use Ranges

Oftentimes, one will only need to specify a range just as an input to a parallel operation. In these situations, creating and initializing a named range just to use it as a temporary value is just too much trouble. As a result, DSP-C allows the definition of "immediate use" ranges using a modified version of the usual CAA access syntax. By including one to three colons in the "access," it is converted into an immediate range description. The following syntax and examples show how these short-term ranges may be specified.

```
range = caa[[d0_bot:d0_top<<:d0_origin<<:d0_stride>>>> <<, dimension1, . . .
      >>]]
range = x[[1:10]];                  // 1-D range
range = y[[0:5, 3:10]];             // 2-D range
range = z[[0:10:5:2, 0:10:5:3]];    // Adjusted origin & stride
```

The number of dimensions in the range must match the number of dimensions in the base CAA. If the origin is omitted, then it is assumed to be at the lower corner of the range (equal to the "bottom" value in all dimensions). If the stride is omitted, then it is assumed to be 1 in all dimensions.

4.3.1.2. Single-Line Operations

Single-line explicitly parallel operations look just like standard lines of C code performing expressions with scalar variables — except that at least one variable is a range instead of a scalar. When these are encountered, DSP-C automatically turns them into small, explicitly parallel loops, potentially with multiple dimensions.

```
output[[0:127]] = input[[0:127]];              // Stream copy
```

```
output[[0:127]] = input[[0:127:127:-1]];        // Reverses input
output[[0:127]] = x[[0:127]] + y[[0:127]];      // Element-wise add
output.hot = x.hot + y.hot;                     // With named ranges
x[[0:127]] *= y[[0:127]];                       // x is read-write
x[[0:127]] += value;                            // Add a scalar to all
acc += x[[0:127]];                              // Reduction
```

When this code is executed, each range is streamed through the scalar kernel defined by the line of code, with the ranges in the text replaced by scalar values from within the range on an element-by-element basis. Along each dimension of each range, elements from one end of the range to the other will be taken and used for calculation, in this order: origin, origin + stride, origin + 2*stride, . . . , maximum usable, minimum usable, minimum + stride, . . . , origin – stride. The maximum usable element is the last element of the form "origin + N*stride" less than or equal to the top of the range, while the minimum usable is the last element with the same form greater than or equal to the bottom of the range. Thus, the following ranges produce the resulting computation orders.

```
Range[0:10:0:1] yields 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Range[0:10:5:1] yields 0, 1, 2, 3, 4, -5, -4, -3, -2, -1     (range #s)
                       5, 6, 7, 8, 9, 0, 1, 2, 3, 4          (CAA #s)
Range[0:10:0:2] yields 0, 1, 2, 3, 4                         (range #s)
                       0, 2, 4, 6, 8                         (CAA #s)
Range[0:10:5:2] yields 0, 1, 2, -2, -1                       (range #s)
                       5, 7, 9, 1, 3                         (CAA #s)
Range[0:10:5:-2] yields 0, 1, 2, -2, -1                      (range #s)
                        5, 3, 1, 9, 7                        (CAA #s)
```

The same dimensions of all ranges must normally be iterated together, so array transposition is impossible. However, the `transpose` operator allows the dimensions in any range to be arbitrarily transposed before elements are extracted. Here is the syntax, valid only within an explicitly parallel operation, and an important example: a matrix multiply. One should note that the numbers taken as input by the operator are the values of the input dimensions in the order of the output dimensions.

```
output_range = dsp_transpose(input_range, output_dim_0, output_dim_1 <<,
      output_dim_2, . . . >>)
acc += x[[0:15,0:4]] * dsp_transpose(y[[0:4,0:15]], 1, 0);
```

This syntax suffers from a pair of occasionally bothersome limitations. For any dimension, all iterating ranges must have an equal number of active elements or the compiler (or runtime error detection, if necessary) will declare an error. Also, the compiler is given free rein to select the order of dimensional operations, as no ordering is implied by the syntax. Therefore, no calculations that depend upon the loop ordering should be performed by this technique. The

compiler will try to catch these, if possible, but some may not be declared as errors. If any of these problems are encountered, the loop will need to use the multi-line syntax, instead.

4.3.1.3. Multi-Line Operations

If one of the limitations of single-line operations becomes a limiting factor, or if a kernel is just too complicated to fit into a single C expression, then multi-line explicit kernels can be used. Multi-line operations use the `dsp_for` keyword to start a loop that is similar to the standard C `for` loop except that it specifies ranges from which to stream elements instead of a scalar loop induction. The following syntax specifies a `dsp_for` loop.

```
dsp_for(range_spec <<, range_spec, . . . >> <<; dim_spec0 <<, dim_spec1 . .
      .>>>>)
{
      loop_body_code
}

where: dim_spec => << dimensional_run_length >> <<: dimension_order >>
       range_spec => iteration_port = range
```

With the dimensional specifiers, one can specify how long the kernel is supposed to execute down a particular dimension, if the different ranges may happen to disagree about the length. This can be supplied for all dimensions, or just for selected dimensions (the others will revert to the default mode of finding common ground among all input ranges or declaring an error). In addition, the programmer may specify a dimensional looping order by adding a *dimension_order* specifier of 0 (outermost loop), 1 (next loop), etc. If desired, all or only a subset of these may be specified. In this way, the programmer can specify all loop ordering, which dimension should be just the outer loop, or nothing at all. The `dsp_for` loop at the bottom of the examples, without any specifiers, works much like a single-line operation, except that it may consist of multiple lines.

```
// Loop with fully specified run lengths and dimension order
dsp_for(x.i = x.my_range, y.i = y[[0:9,0:20]]; 10:0, 10:1)

// Loop with just dimension order
dsp_for(x.i = x.my_range, y.i = y[[0:9,0:20]]; :0, :1)

// Loop with just a single run length
dsp_for(x.i = x.my_range, y.i = y[[0:9,0:20]]; 10:, )

// Loop with no dimension specifiers
dsp_for(x.i = x.my_range, y.i = y[[0:9,0:20]])
```

Within the body of the `dsp_for` loop, code is fairly normal, but with certain limitations, as described below. The "induction variables" used within the loop are the *iteration_port*s specified by the user. In the previous examples, `x.i` would be the induction variable from CAA `x` and `y.i` the induction variable from CAA `y`. These may be used within the body of the loop to represent the current "view" of the streamed-in ranges visible to the loop iteration. Most other code within the loop body is fairly normal.

4.3.2. Limitations

The primary limitation on the code in any explicitly parallel kernel is that the iterated ranges may **only** be accessed through their iteration ports. This does not prohibit the use of offsets from the iteration ports, but it does prohibit random accesses anywhere else in the arrays. In this manner, the "footprint" of each kernel iteration in the CAA should be fairly simple to analyze and always essentially the same, relative to the position of the iteration port. This should allow a reasonably simple scheme to determine which elements of the CAA need to be read by each processing element on each loop iteration — the pattern of the "stream" of data being fed into the processing elements.

Most scalars within each loop iteration should be used locally only. In other words, they should be initialized to a value by every loop iteration *before* they are read. However, the use of external variables will sometimes be necessary. Scalars and small arrays from outside the loop may be used in a read-only within the loop without problems, as long as they all fit within local memory (or potentially even just the registers on Imagine), but if they are also written then analysis becomes much more difficult. When read-write scalar variables are found to carry from one range element "loop iteration" to another, then the compiler marks the variable as a potential reduction variable. At that point, reduction variable optimizations analogous to those performed in the case of standard CAAs may be done. If basic reduction optimizations cannot be performed, then it may be necessary to declare the loop non-vectorizable. If finding and optimizing these variables proves too difficult, then it may be necessary to mark them with special keywords, much as was discussed previously.

One final limitation is on control flow. As was previously discussed for basic CAA analysis, it may be necessary to limit the use of `if` statements within any explicit kernel in order to allow vectorization. This limitation could be as simple as a complete forbidding of `if` statements altogether, or may even be architecturally-dependent. The exact shape of the control flow limitation required will need to be determined after we have coded many more applications to DSP-C and tried to compile them.

4.3.3. Examples

The examples used earlier with basic CAAs and standard C loops have been redone using explicitly parallel kernels in this section. The code is very similar, but the loops are obviously done in an entirely different way. For the 1-D filter, only the initialization and inner loop have been converted, using single-line explicit kernels.

```
int input((24,23))[[128]];
int coeff((24,23))[[128]], input_location;

// Initialize ports & input
input_location = 0;
input[[0:127]] = 0;
// Later, initialize coefficients, themselves

// Routine called once for each input
int((24,23)) fir_filter(int new_data((24,23)))
{
      int a((56,47));         // Extended-precision accumulator

      a = 0;
      input[[input_location++]] = new_data;      // Record input
      a += input[[0:127:input_location]] * coeff[[0:127]];   // Reduces
      fir_filter = dsp_round_to_nearest(a / 128);
}
```

The 2-D filter example uses an explicit, multi-line kernel for the outer loops and a single-line kernel to replace the inner loops. The `dsp_for` operation has requested that the "x" direction of the loop be done first, but has not requested explicit run lengths, so the loop will run for the full length of the ranges — and the compiler will check to make sure that all of the range lengths match.

Because the filter needs to process several elements from around the iteration port, we use the port to access a slightly expanded range of locations in the `input` CAA. Because all of this adjustment is relative to the `input.i` port, however, it is legal. We just have a "footprint" of more than one element in that particular array for each iteration of the kernel. Since the "footprint" of adjacent kernel iterations overlaps, the accesses to the memory should probably be optimized somewhat by the compiler, which may involve some register allocation and (on Imagine) some inter-processing element communication.

```
int input((8,0))[[640,480]];
int output((8,0))[[640,480]];
int coeff((16,15))[[9,9]];
```

```
// Assume that the main program sets up inputs & coefficients

// Routine called once for each frame
void video_filter(void)
{
      int a((32,15));         // Extended-precision accumulator
      int f_x, f_y;
      dsp_port input i;       // Temporary loop iteration ports
      dsp_port output i;

      // Main outer loop, x direction first
      dsp_for( :0, :1 ; input.i = input[[4:635, 4:475]],
            output.i = output[[4:635, 4:475]])
      {
            a = 0;
            a += input[[&input.i[[-4, -4]],&input.i[[4, 4]]]] *
                  coeff[[0:8, 0:8]];
            output.i = dsp_round_to_nearest(a / (9*9));
      }

      // More code will be needed to handle the edges, but
      //   this has been omitted for brevity's sake.
}
```

More sophisticated algorithms will have more complex multi-line kernel loop bodies, more complicated layers of nested loops, and more complicated data reference patterns to be analyzed within the loops. Hopefully, however, we will be able to develop compiler technology that can deal with a large percentage of the possible combinations to find useful speedup.

4.3.4. Other Possible Imagine Influence

As a guiding design philosophy in the first revision, we have tried to keep the various CAA enhancements in DSP-C fairly systematic and coherent, without too many special cases and oddball add-ons that the compiler will have to somehow deal with. However, we may want to enhance this basic definition with more elements from existing vector or streaming languages such as stream C and/or kernel C, if we find that in practice they simply work better than the definitions here — either in terms of ease of programming or for code generation. Enhancements that are *too* architecture specific should be avoided if at all possible, though, since code portability is another primary goal for DSP-C. The "portability" of any proposed enhancements will have to be decided on a case by case basis, of course, since guidelines are hard to set up.

4.4. Architecture-Specific Optimizations for Loops

No matter how well we are able to build compilers, there will be times when the compiler blows a decision about how or where to parallelize a loop. Therefore, DSP-C offers the programmer the option of associating optimization flags, or "suggestions," with each loop. In this way, the programmer can offer strong hints to the compiler as to the best way to optimize any particular bit of code — at least to the extent that the compiler is able to understand the hints. If it can't understand a hint, however, it is free to ignore it. These work just like the previously mentioned architecture-specific optimizations for CAAs, except that they apply to loops, instead. Common uses for these optimizations will be hints like "parallelize here!" or "use SIMD instructions here!"

4.4.1. Syntax & Usage

After any C looping keyword (usually `for` or `dsp_for`, but `while` or `do` will work, too), the programmer may insert a `[[]]` field. Within this field are comma-delineated lists of identifiers or identifiers plus integer values that are fed directly into the compiler to help its optimization. The following is a legal loop with optimization flags.

```
for[[mmx_here, mips_no_parallel, imagine_slicing = 8]](x=0; x<1024; x++)
```

As with other potential locations for optimization flags, it is helpful to have the name of the architecture at the beginning of any architecture-specific flags, for clarity and to prevent collisions between the flags used by different compilers.

5. Intelligently Polymorphic Libraries

Because many common DSP operations such as filters and FFTs have been programmed before, there should be little or no reason for programmers to "reinvent the wheel" and recreate these routines each time they may happen to use them. Unfortunately, the way C libraries work makes it difficult to write easy-to-read and portable code for DSP applications using a single library because DSP programmers want to use *highly optimized* library code. In fact, that is often the most compelling reason to use DSP libraries, because hand optimized kernels can often achieve speedups greater than the most advanced compilers are capable of producing. While most portable C libraries can simply be encoded right in C, as a "one size fits all" option, DSP programmers will want their library functions to be extremely well-optimized for the base architectures that the library may target and the multitude of different inputs that may be fed into the library. If we were to provide a simple C library, then few real programmers would choose to use it, since they would tend to just make their own custom library routines optimized for their specific environment. Therefore, DSP-C libraries must be able to take a simple library call, such as `fft`, and *intelligently* map it to a large family of routines highly optimized to handle a wide variety of different conditions. Before discussing how DSP-C handles the problem of making intelligent libraries, we must explore the problems with existing C libraries that must be surmounted.

The first problem is dealing with the large variety of DSP-C types that may be input into routines. C++-style function polymorphism (static, not virtual) is a reasonable start, but its fixed rules for translating combinations of varying numbers of inputs and the types of inputs into functions is insufficient for DSP-C. Because bit lengths and precisions may be specified for each variable individually, a ridiculous number of possible combinations for input and output quickly become possible. It would be more reasonable to do a C++-style translation after DSP-C data types are assigned to integer types native to the target architecture, but this only handles the problem if precision is not an issue in the choice of library routines — and it may be. Instead, we really want a mechanism to have our library routines be able to declare *ranges* of values for DSP integer sizes and precisions for all of their inputs that they are able to handle.

An even more complex problem is dealing with the varying, compiler-controlled memory allocation that the library routines will have to handle. Sometimes inputs will be preloaded in local memory . . . but sometimes they won't. Sometimes a few of the inputs will fit into local memory, but the others won't. Some very large operand arrays may not even fit into local memory at all, and will therefore have to be moved in and out of the local memory in blocks of some kind. If other routines nearby are reserving chunks of compiler-controlled memory, the

actual amount of memory may vary from one part of a routine to another. Sometimes a programmer will want to use a very memory-conservative but slow or memory-hungry but fast routine. Today, these problems are usually dealt with by having a wide variety of different library routines optimized for different local memory environments. Obviously, calls to these libraries tend to be very architecture-specific. No single call in portable code can possibly deal with the multitude of architecture-specific optimizations that may be necessary. Hence, we really want some way to have the *library* select routines within *itself* based on the memory environment at the time of the call and architecture-inspecific optimization concepts like "fastest" or "minimal memory."

The final problem is dealing with varying combinations in the numbers and arrangements of inputs to routines. This problem is also one that is addressed by C++ polymorphism, but the memory limitations that DSP-C must deal with complicate matters greatly. If an array is used as an input to the library more than once, or as both an input *and* an output, then the optimization to the library routine's core may be very different, since the data may or may not need to be in memory twice. These situations are more common than one might expect, since "in-place" FFTs, for example, are used widely in DSP applications — and tend to be optimized in a different manner than FFTs that use different input and output arrays.

When all of these varying problems are combined, the task of creating a set of general rules to control library routine polymorphism, in the C++ style, is nearly impossible. Instead, it makes more sense for code to be added to the libraries themselves that can be run at *compile time* to select the routine to pick out the best routine for the given situation. If the decision is too difficult, or dependent upon information only available at runtime like variable array lengths, it may even return a family of reasonable routines with information that can be used to choose from among them during code profiling or at runtime. This compile-time language really needs to be a sort of scripting language, similar to Perl — or perhaps even Perl itself with some specialized libraries. It would also bear a resemblance to some macro preprocessors, and therefore some code from one or more of those packages might also be utilized in the actual building of a language.

This chapter attempts to describe the necessary features that any version of DSP-C LCSL (Library Control Scripting Language) must have. Any specific implementation of LCSL will by definition be architecture-dependent, and so some will have all of these features, some only a subset, and some will no doubt need more. It should also be noted that the enumerated types described below are generally only simple examples, and may vary dramatically from architecture to architecture.

5.1. An LCSL Routine

Each base function (the generic function actually called by the library user) is mapped to a single LCSL control routine in the library. Most of the routine is typically a big `switch`-like statement that takes the combination of inputs, outputs, and environmental conditions present in a particular invocation of the function and returns a bit of code that can be put in place of the original call.

Generally, this returned code will be one of three possible things. The first possibility is a call to an actual library routine that performs the function in the best way. For example, an `fft` might be transformed into an architecture and input-specific `fft_i32o32ip` function that is actually present in the library. The second possibility is to return a bit of code that can be inserted inline to replace the library call. Finally, it may be necessary to combine both techniques. A bit of inline code may select between several library routines depending upon conditions that are only known at runtime (most often variable array lengths). If there are a number of possible choices that may prove viable, depending upon conditions outside the control of the library such as the local memory requirements of other routines, then it is even possible for an LCSL control routine to return a family of substitutions that may be selected by the compiler at compile time or after profiling.

5.1.1. Syntax & Usage

Because the potential input and output from an LCSL control routine are fairly complex, these are given their own sections. Instead, this section just gives the basic outline of an LCSL control routine — and it's pretty simple. It should be noted that the proposed simplified syntax here is only a C-like guideline, and may expand somewhat if we choose to implement LCSL in true C, Perl, or some other language that has pre-existing requirements.

```
lcsl_routine base_function_name
{
      << script, mostly a switch-like structure >>
}
```

And that's it (for now, at least)! The real "magic" of LCSL is contained in the inner structure of every routine, and the inputs to it (which it can use to select an appropriate routine) and outputs from it (which give the code substitutions and any rules for using them). The actual script code should be maintained in a ".lcsl" file (or something similar) associated with the library in question.

It should be noted that the LCSL control routine actually replaces any prototypes that may normally occur in the header file, since its internal switch structure effectively "custom builds" prototypes for any invocation of the base function during the script call — or returns an error message if the programmer used an illegal combination of inputs. Instead of prototypes, library header files must simply declare that they are defining LCSL routines in the header files.

```
lcsl_routine base_function_name();
```

When the compiler actually compiles a routine with one of these header lines, it invokes the appropriate LCSL script. This script then determines whether or not the needed "prototype" of the function actually exists, and the appropriate library routine to select if it does. For programmers, however, it may be helpful to include comments in the header files recommending techniques for using the library functions correctly.

When an LCSL-controlled function is actually called, additional "optimization" arguments may be included using a CAA-like notation. None are required (otherwise they would be inputs), but they can be effective at helping compilation. Here is a sample call:

```
FFT[[minimum_memory, mips_length=1024]](input_array, output_array);
```

These optional optimizations listed in the `[[]]` region are parsed and fed into both the LCSL script and the compiler itself, where they can be used to help tune the choice of routine. It should be noted that many of these optimizations will be architecture-specific, so for clarity purposes it helps to have optimization names that include the architecture names in these cases (like `mips_length`, above). Optimizations may also include optional integer arguments after an = sign, as in the example. On architectures or with compilers that don't recognize particular optimizations, they are simply ignored.

5.2. Input to Each Routine

The input to each LCSL routine attempts to give the routine as complete a picture as possible of the way the routine has been called — information on inputs and outputs — and a picture of the environment that the routine must work within. Both of these have some architectural-specific features, so the exact definition of the inputs to an LCSL routine can only be completely defined on an architecture-by-architecture (and perhaps even compiler-by-compiler) basis. This is acceptable, because an LCSL routine is associated with each architecture-specific copy of a library.

5.2.1. Operands

Most information that must be supplied to the LCSL routine is about the nature of the various operands coming into or going out of the library routine. This information is supplied using an array of structures that define all of the necessary information about each operand, plus a few other general bits of information. In order to allow the internal structure of these variables to vary without breaking LCSL code, access routines to look at various parts of the data are supplied.

The first routine simply allows one to determine the number of input operands present.

```
int lcsl_operand_count();
```

Once the operand count has been acquired, an LCSL routine may examine its inputs. Pointers to trees containing type information on them are contained in an array with entries starting at 0 (for the first input given after the function name) on up. Type analysis has been done for ages, so most versions of LCSL would probably just incorporate an existing analysis package of some sort, extended so that it would work with the new DSP-C types. In this model for how a version of LCSL might work, a family of routines are available to examine different known aspects of each operand's type. The following illustrates a simple example of how type trees might be laid out in DSP-C.

```
enum lcsl_node_type { lcsl_compound, lcsl_named, lcsl_element };
enum lcsl_type
{
      /* Compound nodes */
      lcsl_type_struct,
      lcsl_type_union,
      lcsl_type_caa,
      lcsl_type_range,
      lcsl_type_port_array,

      /* Element modifiers */
      lcsl_type_const,
      lcsl_type_complex,
      lcsl_type_pointer,
      lcsl_type_function,
      lcsl_type_array,

      /* Terminal element nodes */
      lcsl_type_void,
      lcsl_type_bitfield,
      lcsl_type_unsigned_char,
      lcsl_type_signed_char,
```

```
      lcsl_type_unsigned_short_int,
      lcsl_type_signed_short_int,
      lcsl_type_unsigned_int,
      lcsl_type_signed_int,
      lcsl_type_unsigned_long_int,
      lcsl_type_signed_long_int,
      lcsl_type_float,
      lcsl_type_double,
      lcsl_type_long_double,
      lcsl_type_unsigned_dsp_integer,
      lcsl_type_signed_dsp_integer,
      lcsl_type_short_float,
      lcsl_type_port
};

/* This is an approximate node definition, which    */
/* should be accessed with helper functions to help */
/* make LCSL code somewhat more portable.           */

typedef struct
{
      /* Type tree maintenance variables */
      lcsl_node_type node_type;
      lcsl_type type;
      lcsl_type_node* next;
      lcsl_type_node* down;

      /* Type information variables */
      char *identifier;
      int int_size;
      int int_precision;
      int dimensions;
      lcsl_lengthtype lengthtypes[];
      int lengths[];
      /* . . plus other architecture-specific fields . . */

} lcsl_type_node;
```

This particular definition uses a tree of `lcsl_type_node` structures to contain type information on a variable. Most types are encoded using straight-line chains of *element*-type nodes connected using the `next` pointers in each node. *Compound* nodes (structures and unions) indicate that the tree branches, with the `down` pointer used to lead to a list of *named* nodes. Each of these nodes has a string identifier and uses both pointers. The `down` pointer points to the next element in the containing structure or union, while the `next` pointer points to the first element in the chain containing the type of this node's variable. A couple of sample trees are illustrated in Figure 12.

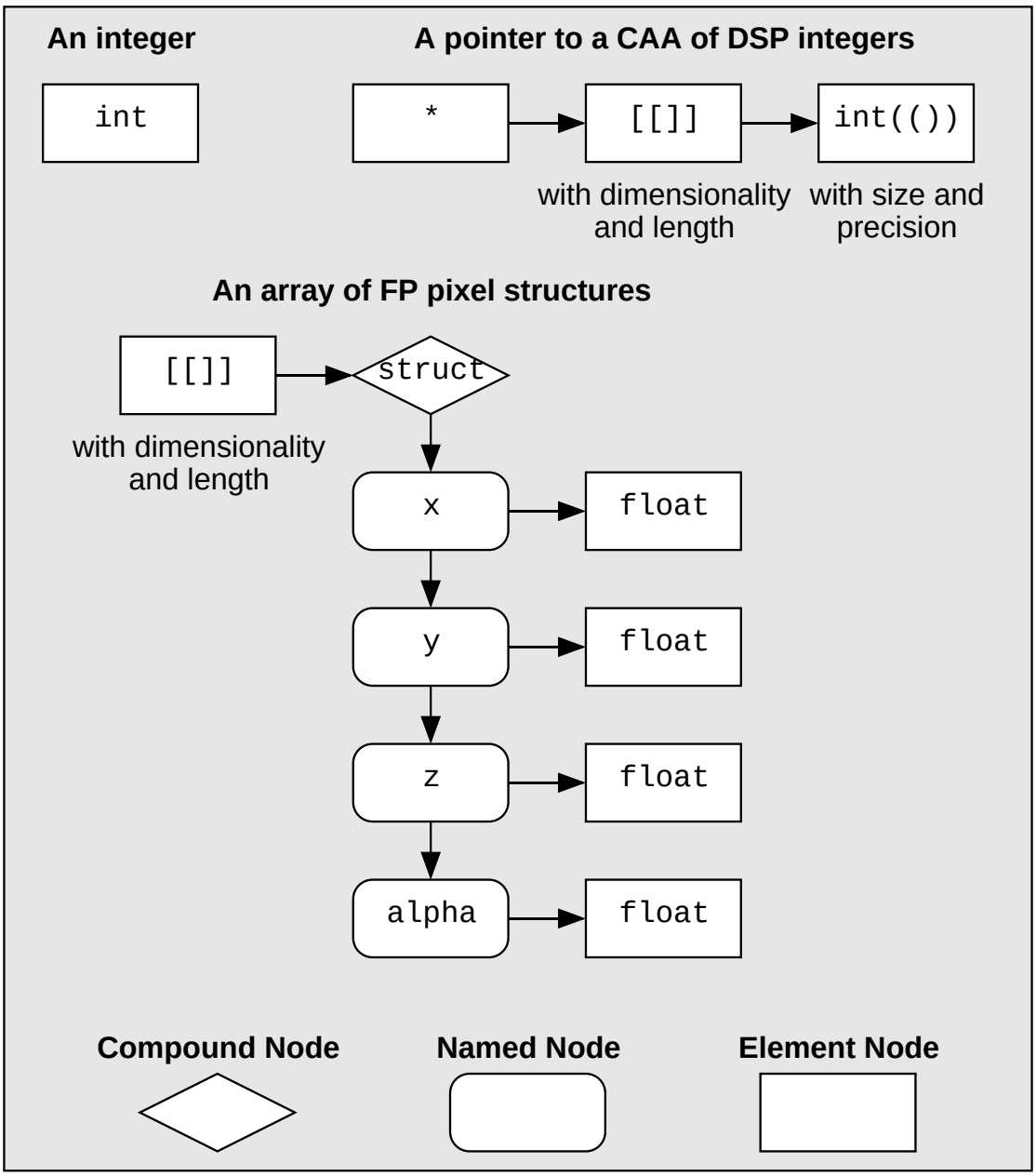

Figure 12: Three sample type trees using the system described in the text.

```
lcsl_type_node* lcsl_operand_basenode(int position);
bool lcsl_type_iscompound(lcsl_type_node* check);
bool lcsl_type_isnamed(lcsl_type_node* check);
bool lcsl_type_iselement(lcsl_type_node* check);
lcsl_type lcsl_type_nodetype(lcsl_type_node* check);
lcsl_type_node* lcsl_type_next(lcsl_type_node* check);
lcsl_type_node* lcsl_type_down(lcsl_type_node* check);
```

These helper functions perform fairly obvious jobs with the architecturally-defined `lcsl_type_node` structure. The first function returns the base type node of any particular input variable. The second group of three just test to see whether or not a node is of one of the three main types, while the latter three return the appropriate fields within the type definition. The `bool` type is just a 0/1 true/false variable, as defined in C++.

```
bool lcsl_type_equaltext(lcsl_type_node* check, char* test_text);
```

This function offers one example of what could eventually become a small family of helper utility routines that perform common functions needed by LCSL scripts to test type trees. In this case, it takes a type tree (or portion of one) starting at the given check node and compares the type contained in the tree with the one given in string form as *test_text*. If they are the same, it returns `true`. For example, a tree consisting solely of an `int` node would match the string `"int"` while a tree describing a 1) pointer to a 2) CAA containing 3) DSP integers would match a string of `"(int(()) [[]])*"`. The latter type is a very common input to DSP-C library calls, so it will be tested for on a regular basis. For reference, trees for both types appear back in Figure 12. Note that exact DSP integer size and precision values and CAA dimensionalities or sizes are not an issue with these functions. Instead, they are tested using their own families of helper functions. The `equaltext` function also allows loose matching of CAAs to the input type given, using the rules given in the previous chapter (so a "`(void [[]])*`" would match any CAA, but means that the library routine should limit its use of the CAA's abilities).

```
enum lcsl_archtype { lcsl_u16, lcsl_s16, lcsl_u32, lcsl_s32, etc. };

char* lcsl_operand_identifier(lcsl_type_node* subtype_location);
int lcsl_operand_intsize(lcsl_type_node* subtype_location);
int lcsl_operand_intprecision(lcsl_type_node* subtype_location);
lcsl_archtype lcsl_operand_archtype(lcsl_type_node* subtype_location);
```

These routines return special information from tree entries that have them. The first returns the identifier name from any base node (compound or simple), as a string. The others return the size, precision, and actual architectural integer type information from a DSP integer element node. Size also works for traditional C bitfields, for codes that use them. To save a lot of use of `lcsl_type_next`, they will automatically skim through a non-branching tree until they find a valid DSP integer node. The `lcsl_archtype` return from the `archtype` routine is obviously very architecture-dependent. Also, this routine may not be entirely necessary because the writers of LCSL scripts will probably already know their machine's integer size-to-type matchups already, and therefore be able to infer this information themselves. Similar routines allow information to be recovered from arrays.

```
enum lcsl_lengthtype { lcsl_fixed, lcsl_variable, lcsl_variable_min,
      lcsl_variable_max, lcsl_variable_minmax };

int lcsl_operand_dimensions(lcsl_type_node* subtype_location);
lcsl_lengthtype lcsl_operand_lengthtype(lcsl_type_node* subtype_location, int
      dimension);
int  lcsl_operand_fixedlength(lcsl_type_node*  subtype_location,  int
      dimension);
int lcsl_operand_minlength(lcsl_type_node* subtype_location, int dimension);
int lcsl_operand_maxlength(lcsl_type_node* subtype_location, int dimension);
```

When an array (normal, CAA, or port array) element token is encountered in the type, it is helpful to be able to determine its length (and dimensionality, for a CAA or port array) when making memory optimization decisions. The `lengthtype` function tells the script how much information is known about the array at compile time — from fixed to completely variable with no known bounds — on a dimension-by-dimension basis. The other functions just return lengths for the different dimensions under different conditions. While it should always be possible for a compiler to feed the length of a fixed-length array to an LCSL script, it may be difficult or impossible to nail down the length of variable-length arrays at all. If this is the case, the `minlength` and `maxlength` functions may be omitted.

Like the DSP integer routines, these functions can automatically skim down a non-branching tree and return information on the first array found. It should be noted, however, that traditional C multidimensional arrays are actually encoded as multiple array nodes, so it will still take some tree manipulation to get all of the information from them. CAAs do not offer this problem, because they are a completely self-contained type at any dimensionality.

```
enum lcsl_inctype { lcsl_normal, lcsl_circular, lcsl_bit_reversed,
      lcsl_circular_reversed };

lcsl_inctype lcsl_operand_inctype(lcsl_type_node* subtype_location, int
      dimension);
int lcsl_operand_rangestart(lcsl_type_node* subtype_location, int dimension);
int lcsl_operand_rangeend(lcsl_type_node* subtype_location, int dimension);
int  lcsl_operand_rangeorigin(lcsl_type_node*  subtype_location,  int
      dimension);
int  lcsl_operand_rangestride(lcsl_type_node*  subtype_location,  int
      dimension);
```

These routines perform similar functions, but return information pertinent to the ports and ranges within the options list for a CAA.

```
enum lcsl_memlocation { lcsl_register, lcsl_local, lcsl_main, lcsl_reg_local,
      lcsl_local_main, lcsl_reg_local_main };
```

```
lcsl_memlocation lcsl_operand_location(int operand_position);
lcsl_memlocation  lcsl_operand_compoundlocation(int  operand_position,
      lcsl_type_node* subtype_location);
```

An LCSL script may need to know where the compiler wants to have each operand before the routine. `lcsl_operand_location` performs this request. For simple variables, it can simply work using the operand position, but for compound variables it may need the actual type node for part of the input to determine its location in memory. The script can then use the returned information to pick an appropriately optimized version of the library call, with the proper sets of memory references included. However, these locations should only be taken as guidelines. If the script really wants or needs to have the memory in a different configuration, it can provide memory reconfiguration requests to the compiler in its output (see below), and force the compiler to comply with a varying memory setup.

Some architectures may also have more complex `lcsl_memlocation` types that specify more detailed options such as separate X and Y local memories or other possible data locations. The exact definition of what the `lcsl_memlocation` entries mean to the library routine is also implementation-specific. Usually, “register” variables are just scalars passed directly according to the architecture’s procedure call variable passing rules, while the others are passed via pointers to the real variables in memory. For these variables, “local” variables are ones that have been pre-loaded by the compiler into the local memory bank (even if there is another copy out in main memory, too), and “main” variables are ones that the routine will have to recover from main memory itself before they can be used. However, there are a multitude of possible variations on these basic guidelines, depending upon the memory interfaces present in a particular architecture.

Using these — or similar — library calls, an LCSL routine should be able to make most necessary library selection decisions, although some architectures may require additional routines or variations on these routines to make all necessary information available.

5.2.2. General Environment

In addition to information about the input and output operands, a routine may need to know information about the system or memory environment that it is expected to operate in, if it must choose between routines with different types of optimizations. Most of these environmental requests will be very architecture-specific.

```
int lcsl_mem_minimum();
int lcsl_mem_maximum();
```

If these functions are supported, they would give the amount of local memory that the compiler has completely free at this point or can give the routine if it performs the maximum amount of data flushing to main memory. Many others are possible, also.

In addition, the user may have requested that the scripts make particular decisions during optimization. While a few common ones are suggested below, most will be very specific to the architecture. For each library call, the DSP-C compiler will parse all requested optimizations and present a `bool` array to the LCSL script, containing flags (and possibly values) only for optimizations that are pertinent to this architecture. This array can then be accessed using the following access function to test any pertinent optimizations.

```
enum lcsl_opts { lcsl_minimum_memory, lcsl_fast_code, etc. };

bool lcsl_optimizations(lcsl_opts which_opt);
int lcsl_opts_values(lcsl_opts which_opt);
```

Both functions are accessed using the enumerated type `lcsl_opts`, which lists the potential optimizations that are legal on this architecture. The first function returns a boolean value indicating whether or not the user chose that particular optimization during this library routine invocation, while the second returns any values that may have been given along with the optimization. These values, if they are used at all, are optimization-specific, and may be interpreted in many different ways.

5.3. Output from Each Routine

Along with some information that allows the compiler to simulate having a real prototype, each LCSL script returns one or more text substitutions that may be used to replace the original library call with C code that will compile normally (using the standard library mechanisms). The inserted code need not just be a simple library call, however. It can just as easily be inlined code or a set of calls with an `if` statement or two to select from among them. If multiple substitutions are returned, then the script must also return information allowing the compiler to select an appropriate choice from among them.

Of course, the programmer may have tried to call a function in an illegal manner. In these cases, it is the responsibility of the script to return some sort of error message indicating what the problem may be. Because of this, the script performs the job of enforcing proper prototype usage on the programmer, instead of the compiler.

5.3.1. Substitution Output

The primary returned information from any LCSL script is the code that will be substituted into the original program to replace the original, generic library call with the actual architecture-specific library call or calls, potentially with some support code around it. This is performed by a single, very important function in LCSL.

```
int lcsl_return(char* substituted_code_text, char** operands);
```

The integer returned by this function is the substitution number, or "*sub_number*" field. When an LCSL script returns more than one potential substitution, each will be given a unique substitution number to differentiate it from the others. These numbers can then be used with the other output routines to associate particular variable or compiler options with particular substitutions.

The code returned in the *substituted_code_text* string is effectively inserted into the original program in place of the base library call invoked by the programmer. Within it, any identifier strings that match those listed in the *operands* string array are converted into the actual inputs to the base library call (with `operands[0]` = first base library input, `operands[1]` = second input, etc.). Other text in the *substituted_code_text* string is left intact. For example, the following is a fairly straightforward substitution:

Base call:

```
FFT(x,y)
```

Substitution returned:

```
"fft_i32o32nip(input_array,output_array,1024);"
operands[0] = "input_array", operands[1] = "output_array"
```

Final effective call:

```
fft_i32o32nip(x,y,1024);
```

With this return, the LCSL script turns a straightforward FFT call into an ugly, highly-optimized library FFT routine for a particular architecture. While most of this should be fairly straightforward, the semicolon at the end should bother most users of macros. However, this is actually correct LCSL usage. Unlike traditional C macro invocations, LCSL library text substitution is not done in such a way that may require extra parenthesis or cause problems with multiple evaluation. Instead, the actual substituted code is much more like the following:

```
_dsp_temp_0 = x;
```

```
_dsp_temp_1 = y;
fft_i32o32nip(_dsp_temp_0,_dsp_temp_1,1024);
```

This may be complicated a bit if the function has a return value and is used in the middle of another expression. For example, if the FFT returns an `int` error code, and is used in the test parameter of an `if` statement, the results will look like this:

Base call:

```
if((error = FFT(x,y)) != 0)
      printf("There was an error #%d!\n",error);
```

Substitution returned:

```
"dsp_return fft_i32o32nip(input_array,output_array,1024);"
operands[0] = "input_array", operands[1] = "output_array"
```

Final effective call:

```
if((error = fft_i32o32nip(x,y,1024)) != 0)
      printf("There was an error #%d!\n",error);
```

Actual code is like:

```
_dsp_temp_0 = x;
_dsp_temp_1 = y;
_dsp_temp_R = fft_i32o32nip(_dsp_temp_0,_dsp_temp_1,1024);
if((error = dsp_return) != 0)
      printf("There was an error #%d!\n",error);
```

The `dsp_return` keyword is used within a substitution just like a traditional C `return` to indicate which value should be treated as the original library call's "return value."

The substitution code can contain an entire function — or portion of one — itself. It may declare variables, call one or more architectural library routines, or do pretty much anything else that a full function can. If new variables are declared, the compiler ensures that no namespace conflicts occur when the code is substituted into the original program. For example, the following substitution sums the elements of a one-dimensional, variable-length CAA. At runtime, the substitution determines if the CAA is short enough to fit into local memory it just performs the sum with a small inline loop. Otherwise, a library routine that has support for controlling dataflow between main memory and local memory is invoked. In this case, `operand[0]` is defined to be the identifier "`array`."

```
int i, len, acc((32,24)) = 0;

len = dsp_length[[array]];
if (len > 128) acc = sigma_mainmem(array, len);
```

```
      else for(i=0; i<len; i++) acc = acc + array[[i]];
dsp_return acc;
```

Even more complex substitutions are also possible. With the flexibility provided by LCSL's substitution engine, just about any macro or library call substitution may be handled. About the only thing that substitutions cannot do is declare globals, type definitions, or other things that appear in C outside of functions themselves. However, any of these which may be necessary may easily be declared in advance in the libraries that the LCSL's substitutions reference, so it should never be a limitation.

5.3.2. Compiler-Driving Output

A script may return more than one possible text substitution, so we need to be able to inform the compiler what the differences between the different returned code segments are. Some of these details simply help the compiler select from among the returned routines, but some will force adjustments to surrounding code. Most of these adjustments involve moving variables to different locations in memory before or after the routine is called. The following helper routines attach requirements or "hints" to the returned text substitutions for the compiler.

```
void  lcsl_compile_newarchtype(int  sub_number,  int  operand_position,
      lcsl_archtype new_type);
void  lcsl_compile_newarchtypecompound(int  sub_number,  lcsl_type_node*
      subtype_location, lcsl_archtype new_type);
void  lcsl_compile_preshift(int  sub_number,  int  operand_position,  int
      preshift_amount);
void  lcsl_compile_preshiftcompound(int  sub_number,  lcsl_type_node*
      subtype_location, int preshift_amount);
void  lcsl_compile_location(int  sub_number,  int  operand_position,
      lcsl_memlocation location);
void  lcsl_compile_locationcompound(int  sub_number,  lcsl_type_node*
      subtype_location, lcsl_memlocation location);
```

These routines specify when a substitution requires one or more input variables with different requirements from the input guidelines given by the compiler. An LCSL script is free to "suggest" a substitution with variables moved into a new architectural type, in a different memory location, or in a preshifted form. For example, it might be possible to make faster versions of a routine if the inputs are reassigned to a different, longer architectural data type or if they are pre-loaded into local memory. If possible, however, the LCSL script should attempt to return at least one substitution that follows the given input guidelines, since the compiler probably made them for a reason, but it is not a hard requirement if no library routines match the compiler's request.

Preshifting of inputs means that they are right-shifted within their architectural registers (and 0 or sign-extended in the process, depending upon whether or not they are signed), eliminating some of the extra bits of precision that would otherwise be associated with the variable. While instruction outputs should not be preshifted, to preserve data type range information, it is perfectly acceptable for inputs. This technique was illustrated back in Figure 3b, with the second input to the multiply. If a single array will be used by multiple function calls (or the same call in a loop) as an input preshifted by the same amount, it makes sense for the compiler to just preshift it once and feed the same array into all of the different invocations. For example, a coefficient array to a filter called within an inner loop may be better maintained in a preshifted form, so the filter doesn't have to shift the coefficients every time its called.

```
void lcsl_compile_localspace(int sub_number, int space_needed);
```

This routine is a very important one that tells the compiler just how much local memory the substituted library code will need to hold any temporary values or buffers, if it can be precalculated in advance. This allows the compiler to make decisions about memory allocation across the library call without actually performing lots of low-level analysis on the call each time. On architectures with a simple local memory structure, this call may be used more or less as-is, but on ones with more complex local memories the inputs to this call may need to be much more complex.

```
void lcsl_compile_runtime(int sub_number, int approx_time);
void lcsl_compile_runorder(int sub_number, int order_number);
```

These calls give the compiler hints about how fast different substitutions might be. The first should be used if the LCSL script actually wants to try and figure out an approximate runtime for the routine (in architecturally-dependent units), while the second can be used if the script just wants to order the routines from 0 (fastest) on up to the slowest, sequentially. A compiler interpreting these can then examine substitutions from the fastest to the slowest, judging the requirements of each (unusual input and increased memory usage) against the potential performance gain. Some selection decisions may be then made automatically, if one choice is obviously better, while others may be put off until profiling information from execution of the different forms is obtained. As we perform compiler development, heuristics will have to be developed to streamline this process for each architecture that we target.

Finally, one might ask why we would want to have the compiler make decisions from among several choices like this, anyway? The primary reason is that these library routines may be deep inside a series of nested loops, and called in sequence with several other routines. Because the compiler has considerable control over memory in many DSP architectures, and because many of

the input and output operands will be large arrays, it is helpful to allow the compiler to perform global optimization of the data movement among memory levels, even through library routines. For example, it may decide to anchor one particular array in local memory that several library routines use in succession, while letting other arrays flow in and out from main memory. This will allow greater minimization of unnecessary and time-consuming data movement. It is necessary to do this optimization by selecting among multiple choices returned from LCSL scripts because memory movement optimization can really only be done *after* all the calls to the libraries have been resolved. Conversely, on architectures without a bank of local memory that can be controlled by the compiler, most of this returned information would be of little or no importance, if it is even sent back at all.

On another note, it is also helpful to provide opportunities for the compiler to provide multiple options to the compiler for preshifting of inputs. When a preshifted array may be used repeatedly, by several routines, it makes sense to keep it that way at all times. On the other hand, if the preshifted array is only used once, then it makes more sense to do the shifting on the array as each element is loaded in for computation. Giving the compiler the choice between routines of both types can allow more optimal code to be generated. On architectures without compiler-controlled memory, this will probably be the only notification returned most of the time.

5.3.3. Prototype Output

Finally, it is necessary for a compiler to simulate having a real prototype in its .h file, instead of just the LCSL shell header information. Much information — input types, for example — can be inferred right from the call, assumed to be correct when the LCSL script doesn't return an error condition. However, a couple parts of the prototype cannot be implied so directly. These routines cover the gaps.

```
void lcsl_proto_const_all(int variable_position);
void lcsl_proto_const(int sub_number, int variable_position);
void lcsl_proto_outtype_all(char* type_string);
void lcsl_proto_outtype(int sub_number, char* type_string);
```

The `const` function prototype not only helps fill out the "official" C prototype, but can also help the compiler optimize code around the library routine. By noting which inputs to the library routine are used *only* as inputs, the declaration can minimize unnecessary duplication of arrays that may be generated by some compilers when an array is used for both input and output simultaneously. The *variable_position* field refers to the position of variables in the original call, while the *substitution_number* field allows different library routines returned to have different constant behavior. The `_all` form applies the declaration to all possible returned substitutions all at once, if they treat their input identically.

Finally, the `out_type` functions just notify the compiler of the type of variable that each particular version of the function returns (if they may vary). This should just be a character string corresponding to the type text that would normally precede the function call in a prototype.

5.3.4. Error Output

Output in the event of an error is normally fairly simple. The script may just return nothing at all, in which case the compiler must assume that the programmer made an error of some kind and halt accordingly. Alternately, DSP-C provides a mechanism for returning an error message that the compiler can use to inform the user what kind of error they made. When this function is encountered, the script should always terminate.

```
lcsl_error(int code, char* message);
```

The exact structure of the information returned to the compiler will vary on a compiler-to-compiler basis, but some sort of numerical code and human-readable string are suggested.

5.4. A Simple Example

This small example library contains only a single routine, which adds all of the elements of a single-dimensional `float` CAA together and returns the sum as a `float`. Within the library itself, there are two different optimized versions. `sigma_main` streams its input in from main memory, keeping local memory free. The faster `sigma_local` streams its input from a buffer in local memory. It is up to the LCSL script to choose properly between these two options. The following list points out some of the highlights.

- **Input Prototype Enforcement:** The beginning portion of the script checks to make sure that the caller has actually given us a legal input, the job normally done by a compiler automatically by comparing the input with a set of prototypes. For a simple routine like this, prototype enforcement would have actually worked fine. However, all it takes is one or two DSP integers and prototype enforcement gets much more tricky. The `lcsl_error` function is used to return error messages, when necessary.
- **Prototype Declaration:** Once we've actually determined that this is a legal call, we tell the compiler what we're going to output and about any constant inputs. However, if these could vary depending upon the exact library routine selected, then we might choose to hold off on this until later.
- **Output based on Memory Usage:** The first output option checks to see if the variable is already locally-allocated, and chooses to use the fast, local library call in that case. The next

option sees if the array is too large for local memory and switches to the main memory version, instead. Many of the choices and optimizations made by LCSL scripts will be based on memory management decisions like this, because the memory behavior of DSP algorithms will often be a key optimization point or limiting factor.

- **Output based on Optimizations:** When memory limits do not force decisions, the script tries to take user-requested optimizations into account. However, physical resource limits should always be handled before these "suggestions."
- **Multiple Returns:** When multiple routines seem reasonable, more than one return may be made to the compiler. It can then choose from among the different returns using global optimization information or simply by trying all of the options during profiling runs.
- **Complex Output:** For the variable-length array case, the script spits out some code that makes decisions on-the-fly. This code is inserted into the original program like a macro expansion, using the rules given earlier in the chapter. It even calls a few utility routines (which we assume always exist here). When no single library routine fits the situation, but you still want to support a particular input, this is the way to go.

This short example should give you a basic idea as to the goal of LCSL scripting. It may be desirable to add even more power or simplified versions of common structures in the futures, as we try to build an actual working version and write library control scripts with it.

```
/* ROUTINE: float sigma(float input[[]]); */
/*  Sums across an array */

lcsl_routine sigma
{
      lcsl_type_node* in_ptr;
      int in_len, sub;
      char* out, ops = "in";

      /* INPUT PROTOTYPE ENFORCEMENT (error checking) */

      /* Check number of operands */
      if (lcsl_operand_count() != 1)
      {
            lcsl_error(1, "sigma: Wrong # of operands!\n");
            break;
      }

      /* Check type of operands */
      in_ptr = lcsl_operand_basenode(0);
      if (!lcsl_type_equaltext(in_ptr, "(float[[]])*"))
      {
            lcsl_error(2, "sigma: Requires CAA of float\n");
            break;
```

```
}

/* Check dimensionality of operands */
if (lcsl_operand_dimensions(in_ptr) != 1)
{
      lcsl_error(3, “sigma: Only for 1-D arrays\n”);
      break;
}

/* PROTOTYPE DECLARATION */
lcsl_proto_const_all(0);
lcsl_proto_outtype_all(“float”);

/* OUTPUT SELECTION */

/* See if it’s already in local memory, use local form if so */
if (lcsl_operand_location(0) == lcsl_local)
{
      lcsl_return(“dsp_return sigma_local(in);”, &ops);
      break;
}

/* If optimized for low memory use, use main memory form */
if (lcsl_optimizations(lcsl_minimum_memory))
{
      lcsl_return(“dsp_return sigma_main(in);”, &ops);
      break;
}

/* See if the array is variable-length */
if (lcsl_operand_lengthtype(in_ptr) == lcsl_fixed)
{
      in_len = lcsl_operand_fixedlength(in_ptr);

      /* If the array is too large for local memory, use main */
      if (in_len*sizeof(float) > lcsl_mem_maximum())
      {
            lcsl_return(“dsp_return sigma_main(in);”, &ops);
            break;
      }

      /* If optimized for fast code, force use of local form */
      if (lcsl_optimizations(lcsl_fast_code))
      {
            sub = lcsl_return(“dsp_return sigma_local(in);”,&ops);
            lcsl_compile_location(sub, 0, lcsl_local);
            lcsl_compile_localspace(sub, in_len*sizeof(float));
            break;
      }
```

```
            /* Otherwise, let compiler make the choice! */
            if (lcsl_optimizations(lcsl_fast_code))
            {
                  /* Fast, but memory hog */
                  sub = lcsl_return("dsp_return sigma_local(in);",&ops);
                  lcsl_compile_location(sub, 0, lcsl_local);
                  lcsl_compile_localspace(sub, in_len*sizeof(float));
                  lcsl_compile_runorder(sub, 0);

                  /* Or slow, but little memory use */
                  sub = lcsl_return("dsp_return sigma_mainl(in);",&ops);
                  lcsl_compile_localspace(sub, 0);
                  lcsl_compile_runorder(sub, 1);
                  break;
            }
      }
      else
      /* If array is variable-length, figure it out on-the-fly */
      {
            out = sprintf("(float[[]])* local_in; float temp; \
                              if (dsp_length(in) > %d) \
                                    dsp_return sigma_main(in); \
                              else \
                              { \
                                    local_in = local_temp_load(in); \
                                    temp = sigma_local(local_in); \
                                    local_temp_free(local_in); \
                                    dsp_return temp; \
                              }",
                        lcsl_mem_maximum()/sizeof(float));
            lcsl_return(out, &ops);
            break;
      }

}
```