The Challenges of Synthesizing Hardware from C-Like Languages

Stephen A. Edwards
Columbia University

Editor's note:
This article presents one side of an ongoing debate on the appropriateness of C-like languages as hardware description languages. The article examines various features of C and their mapping to hardware, and makes a cogent argument that vanilla C is not the right language for hardware description if synthesis is the goal.
—Sandeep K. Shukla, Virginia Polytechnic and State University

The main reason people have proposed C-like languages for hardware synthesis is familiarity. Proponents claim that by synthesizing hardware from C, we can effectively turn every C programmer into a hardware designer. Another common motivation is hardware-software codesign: Designers often implement today's systems as a mix of hardware and software, and it's often unclear at the outset which portions can be hardware and which can be software. The claim is that using a single language for both simplifies the migration task.

I argue that these claims are questionable and that pure C is a poor choice for specifying hardware. On the contrary, the semantics of C and similar imperative languages are distant enough from hardware that C-like thinking might be detrimental to hardware design. Instead, successful hardware synthesis from C seems to involve languages that vaguely resemble C, mostly its syntax. Examples of these languages include Celoxica's Handel-C[1] and NEC's Behavior Description Language (BDL).[2] You can think of executing C code on a traditional sequential processor as synthesizing hardware from C, but the techniques presented here strive for more highly customized implementations that exploit greater parallelism, hardware's main advantage. Unfortunately, the C language has no support for user-specified parallelism, and so either the synthesis tool must find it (a difficult task) or the designer must use language extensions and insert explicit parallelism. Neither solution is satisfactory, and the latter requires that C programmers think differently to design hardware.

My main point is that giving C programmers tools is not enough to turn them into reasonable hardware designers. Efficient hardware is usually very difficult to describe in an unmodified C-like language, because the language inhibits specification or automatic inference of adequate concurrency, timing, types, and communication. The most successful C-like languages, in fact, bear little semantic resemblance to C, effectively forcing users to learn a new language (but perhaps not a new syntax). As a result, techniques for synthesizing hardware from C either generate inefficient hardware or propose a language that merely adopts part of C syntax.

Here, I focus only on the use of C-like languages for hardware synthesis and deliberately omit discussion of other important uses of a design language, such as validation and algorithm exploration. C-like languages are far more compelling for these tasks, and one in particular, SystemC, is now widely used, as are many ad hoc variants.

A short history of C

Dennis Ritchie developed C in the early 1970s,[3] based on experience with Ken Thompson's B language, which had evolved from Martin Richards' Basic Combined Programming Language (BCPL). Ritchie described all three as "close to the machine" in the sense that their abstractions are similar to data types and operations supplied by conventional processors.

A core principle of BCPL is its memory model: an

Performance or bust

Throughout this article, I assume that optimizing performance—for example, speed under area and power constraints—is the main goal of hardware synthesis (beyond, of course, functional correctness). This assumption implicitly shapes all my criticisms of using C for hardware synthesis and should definitely be considered carefully.

On the one hand, performance optimization has obvious economic advantages: An efficient circuit solves problems faster, is cheaper to manufacture, requires less power, and so forth. Historically, this has been the key focus of logic synthesis, high-level synthesis, and other automated techniques for generating circuits.

On the other hand, optimization can have disadvantages such as design time and nonrecurring engineering costs. The distinction between full-custom ICs and ASICs illustrates this. A company like Intel, for example, is willing to invest an enormous number of hours in designing and hand-optimizing its next microprocessor's layout because of the volume and margins the company commands. A company like Cisco, however, might implement its latest high-end router on an FPGA because it doesn't make economic sense to design a completely new chip. Both approaches are reasonable.

A key question, then, is: What class of problems does hardware synthesis from C really target? This article assumes an audience of traditional hardware designers who want to design hardware more quickly, but other articles target designers who would otherwise implement their designs in software but need faster results. The soundness of my conclusions may well depend on which side of this fence you're on.

undifferentiated array of words. BCPL represents integers, pointers, and characters all in a single word; the language is effectively typeless. This made perfect sense on the word-addressed machines BCPL was targeting, but it wasn't acceptable for the byte-addressed PDP-11 on which C was first developed.

Ritchie modified BCPL's word array model to add the familiar character, integer, and floating-point types now supported by virtually every general-purpose processor. Ritchie considered C's treatment of arrays to be characteristic of the language. Unlike other languages that have explicit array types, arrays in C are almost a side effect of its pointer semantics. Although this model leads to simple, efficient implementations, Ritchie observed that the prevalence of pointers in C means that compilers must use careful dataflow techniques to avoid aliasing problems while applying optimizations.

Ritchie listed a number of infelicities in the language caused by historical accident. For example, the use of *break* to separate cases in switch statements arose because Ritchie copied an early version of BCPL; later versions used *endcase*. The precedence of bitwise-AND is lower than the equality operator because the logical-AND operator was added later.

Many aspects of C are greatly simplified from their BCPL counterparts because of limited memory on the PDP-11 (24 Kbytes, of which 12 Kbytes were devoted to the nascent Unix kernel). For example, BCPL allowed the embedding of arbitrary control flow statements within expressions. This facility doesn't exist in C, because limited memory demanded a one-pass compiler.

Thus, C has at least four defining characteristics: a set of types that correspond to what the processor directly manipulates, pointers instead of a first-class array type, several language constructs that are historical accidents, and many others that are due to memory restrictions.

These characteristics are well-suited to systems software programming, C's original application. C compilers have always produced efficient code because the C semantics closely match the instruction set of most general-purpose processors. This also makes it easy to understand the compilation process. Programmers routinely use this knowledge to restructure source code for efficiency. Moreover, C's type system, while generally very helpful, is easily subverted when needed for low-level access to hardware.

These characteristics are troublesome for synthesizing hardware from C. Variable-width integers are natural in hardware, yet C supports only four sizes, all larger than a byte. C's memory model is a large, undifferentiated array of bytes, yet hardware is most effective with many small, varied memories. Finally, modern compilers can assume that available memory is easily 10,000 times larger than that available to Ritchie.

C-like hardware synthesis languages

Table 1 lists some of the C-like hardware languages proposed since the late 1980s (see also De Micheli[4]). One of the earliest was Cones, from Stroud et al.[5] From a strict subset of C, it synthesized single functions into combinational blocks. Figure 1 shows such a function. Cones could handle conditionals; loops, which it unrolled; and arrays treated as bit vectors.

Ku and De Micheli developed HardwareC[6] for input to their Olympus synthesis system.[7] It is a behavioral hardware language with a C-like syntax and has extensive support for hardware-like structure and hierarchy.

Table 1. C-like languages for hardware synthesis.

Language	Comment
Cones	Early, combinational only
HardwareC	Behavioral synthesis centered
Transmogrifier C	Limited scope
SystemC	Verilog in C++
Ocapi	Algorithmic structural descriptions
C2Verilog	Comprehensive
BDL	Many extensions and restrictions (NEC)
Handel-C	C with CSP (Celoxica)
SpecC	Resolutely refinement based
Bach C	Untimed semantics (Sharp)
CASH	Synthesizes asynchronous circuits
Catapult C	ANSI C++ subset (Mentor Graphics)

```
INPUTS: IN[5];
OUTPUT: OUT[3];
rd53()
{
     int count, i;
     count = 0;
     for (i=0 ; i<5 ; i++)
       if (IN[i] == 1)
         count = count + 1;
     for (i=0 ; i<3 ; i++) {
       OUT[i] = count & 0x01;
       count = count >> 1;
     }
}
```

Figure 1. A function that returns a count of the number of 1's in a five-bit vector in Cones. The function is translated into a combinational circuit.

```
#define SIZE 8
process gcd (xi, yi, rst, ou)
     in port xi[SIZE], yi[SIZE];
     in port rst;
     out port ou[SIZE];
{
     boolean x[SIZE], y[SIZE];

     write ou = 0;
     if ( rst ) <
       x = read(xi);
       y = read(yi);
     >

     if ((x != 0) & (y != 0))
       repeat {
         while (x >= y)
           x = x - y;
         <
           x = y; /* swap x and y */
           y = x;
         >
       } until (y == 0);
     else
       x = 0;
     write ou = x;
}
```

Figure 2. Greatest common divisor algorithm in HardwareC. Statements within a < > block run in parallel; statements within a { } block execute in parallel when data dependencies allow.

Figure 2 shows the greatest common divisor (GCD) algorithm in HardwareC.

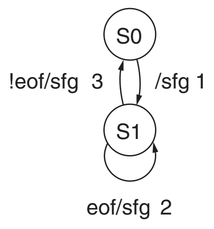
Galloway's Transmogrifier C is a fairly small C subset that supports integer arithmetic, conditionals, and loops.[8] Unlike Cones, it generates sequential designs by inferring a state at function calls and at the beginning of while loops. Figure 3 shows a decoder in Transmogrifier C.

```
#pragma intbits 8
seven_seg(x)
#pragma intbits 4
int x;
{
#pragma intbits 8
     int result;
     x = x & 0xf; result = 0;
     if (x == 0x0) result = 0xfc;
     if (x == 0x1) result = 0x60;
     if (x == 0x2) result = 0xda;
     if (x == 0x3) result = 0xf2;
     if (x == 0x4) result = 0x66;
     if (x == 0x5) result = 0xb6;
     if (x == 0x6) result = 0xbe;
     if (x == 0x7) result = 0xe0;
     if (x == 0x8) result = 0xfe;
     if (x == 0x9) result = 0xf6;
     return(~result);
}

twodigit(y)
int y;
{
     int tens;
     int leftdigit, rightdigit;
     outputport(leftdigit,
       37, 44, 40, 29, 35, 36, 38, 39);
     outputport(rightdigit,
       41, 51, 50, 45, 46, 47, 48, 49);

     tens = 0;
     while (y >= 10) {
       tens++;
       y -= 10;
     }
     leftdigit = seven_seg(tens);
     rightdigit = seven_seg(y);
}
```

Figure 3. Two-digit decimal-to-seven-segment decoder in Transmogrifier C. Output-port declarations assign pin numbers.

```
#include "systemc.h"
#include <stdio.h>

struct decoder : sc_module {
     sc_in<sc_uint<4> > number;
     sc_out<sc_bv<7> > segments;

     void compute() {
       static sc_bv<7> codes[10] = {
         0x7e, 0x30, 0x6d, 0x79, 0x33,
         0x5b, 0x5f, 0x70, 0x7f, 0x7b };
       if (number.read() < 10)
         segments = codes[number.read()];
     }

     SC_CTOR(decoder) {
       SC_METHOD(compute);
       sensitive << number;
     }
};

struct counter : sc_module {
     sc_out<sc_uint<4> > tens;
     sc_out<sc_uint<4> > ones;
     sc_in_clk clk;

     void tick() {
       int one = 0, ten = 0;
       for (;;) {
         if (++one == 10) {
           one = 0;
           if (++ten == 10) ten = 0;
         }
         ones = one;
         tens = ten;
         wait();
       }
     }

     SC_CTOR(counter) {
       SC_CTHREAD(tick, clk.pos());
     }
};
```

Figure 4. A two-digit, decimal-to-seven-segment decoder in SystemC. The decoder produces combinational logic; the counter produces sequential logic.



```
fsm f;
initial s0;
state s1;

s0 << always    << sfg1 << s1;

s1 << cnd(eof) << sfg2 << s1;

s1 << !cnd(eof)<< sfg3 << s0;
```

Figure 5. FSM described in Ocapi. This is a declarative style executed to build data structures for synthesis rather than compiled in the traditional sense.

SystemC is a C++ dialect that supports hardware and system modeling.[9] Its popularity stems mainly from its simulation facilities (it provides concurrency with lightweight threads), but a subset of the language can be synthesized. SystemC uses the C++ class mechanism to model hierarchical structure and describes hardware through combinational and sequential processes, much as Verilog and VHDL do. Cynlib, from Forte Design Systems, is similar. Figure 4 shows a decoder in SystemC.

The Ocapi system from IMEC (the Interuniversity Microelectronics Center in Belgium) is also C++ based but takes a different approach.[10] Instead of being parsed, analyzed, and synthesized, the C++ program is run to generate in-memory data structures that represent the hardware system's structure. Supplied classes provide mechanisms for specifying data paths, finite-state machines (FSMs), and similar constructs. These data structures are then translated into languages such as Verilog and passed to conventional synthesis tools. Figure 5 shows an FSM in Ocapi.

The C2Verilog compiler developed at CompiLogic (later called C Level Design and, since November 2001, part of Synopsys) is one of the few compilers that can claim broad support of ANSI C. It can translate pointers, recursion, dynamic memory allocation, and other thorny C constructs. Panchul, Soderman, and Coleman hold a broad patent covering C-to-Verilog-like translation, which describes their compiler in detail.[11]

NEC's Cyber system accepts BDL.[2] Like HardwareC, Cyber is targeted at behavioral synthesis. BDL has been in industrial use for many years and deviates greatly from ANSI C by including processes with I/O ports, hardware-specific types and operations, explicit clock cycles, and many synthesis-related pragmas.

Celoxica's Handel-C is a C variant that extends the language with constructs for parallel statements and Occam-like rendezvous communication.[1] Handel-C's timing model is uniquely simple: Each assignment statement takes one cycle. Figure 6 shows a four-place buffer in Handel-C.

Gajski et al.'s SpecC language[12] is a superset of ANSI C, augmented with many system- and hardware-modeling constructs, including constructs for FSMs, concurrency, pipelining, and structure. The latest language reference manual lists 33 new keywords.[13] SpecC imposes a refinement methodology. Thus, the entire language is not directly synthesizable, but a series of manual and automated rewrites can refine a SpecC description into one that can be synthesized. Figure 7 shows a state machine described in a synthesizable RTL dialect of SpecC.

```
const dw = 8;

void main(chan (in) c4 : dw, chan (out) c0 : dw)
{
     int d0, d1, d2, d3;
     chan c1, c2, c3;

     void e0() { while (1) { c1 ? d0; c0 ! d0; } }
     void e1() { while (1) { c2 ? d1; c1 ! d1; } }
     void e2() { while (1) { c3 ? d2; c2 ! d2; } }
     void e3() { while (1) { c4 ? d3; c3 ! d3; } }

     par {
       e0(); e1(); e2(); e3();
     }
}
```

Figure 6. Four-place buffer in Handel-C. The ? and ! operators are CSP-inspired receive and transmit operators.

```
behavior even(
     in event clk,
     in unsigned bit[1] rst,
     in bit[31:0] Inport,
     out bit[31:0] Outport,
     in bit[1] Start,
     out bit[1] Done,
     out bit[31:0] idata,
     in bit[31:0] iocount,
     out bit[1] istart,
     in bit[1] idone,
     in bit[1] ack_istart,
     out bit[1] ack_idone)
{
     void main(void) {
       bit[31:0] ocount;
       bit[31:0] mask;
       enum state { S0, S1, S2, S3 } state;

       state = S0;

       while (1) {
         wait(clk);
         if (rst == 1b) state = S0;
         switch (state) {
         case S0:
           Done = 0b;
           istart = 0b;
           ack_idone = 0b;
           if (Start == 1b) state = S1;
           else state = S0;
           break;
         case S1:
           mask = 0x0001;
           idata = Inport;
           istart = 1b;
           if (ack_istart == 1b)
              state = S2;
           else state = S1;
           break;
         case S2:
           istart = 0b;
           ocount = iocount;
           if (idone == 1b) state = S3;
           else state = S2;
           break;
         case S3:
           Outport = ocount & mask;
           ack_idone = 1b;
           Done = 1b;
           if (idone == 0) state = S0;
           else state = S3;
           break;
         }
       }
     }
};
```

Figure 7. State machine in a synthesizable RTL dialect of SpecC. The wait(clk) statement denotes a clock cycle boundary.

Like Handel-C, Sharp's Bach C is an ANSI C variant with explicit concurrency and rendezvous communication.[14] However, Bach C only imposes sequencing rather than assigning a particular number of cycles to each operation. Also, although it supports arrays, Bach C does not support pointers.

Budiu and Goldstein's CASH compiler is unique among the C synthesizers because it generates asynchronous hardware.[15] It accepts ANSI C, identifies instruction-level parallelism (ILP), and generates an asynchronous dataflow circuit.

Mentor Graphics' recent (2004) Catapult C performs behavioral synthesis from an ANSI C++ subset. Because it is a commercial product, details of its features and limitations are not publicly available. However, it appears to be a strict subset of ANSI C++ (that is, with few, if any, language extensions).

Concurrency

The biggest difference between hardware and software is its execution model. Software follows a sequential, memory-based execution model derived from Turing machines, whereas hardware is fundamentally concurrent. Thus, sequential algorithms that are efficient in software are rarely the best choice in hardware. This has serious implications for software programmers designing hardware—their familiar toolkit of algorithms is suddenly far less useful.

Why is so little software developed for parallel hardware? The plummeting cost of parallel hardware would make such software appear attractive, yet concurrent programming has had limited success compared with its sequential counterpart. One fundamental reason is that humans have difficulty conceiving of parallel algorithms, and thus many more sequential algorithms exist. Another problem is disagreement about the preferred parallel-programming model (for example, shared memory versus message passing), as demonstrated by the panoply of parallel-programming languages, none of which has emerged as a clear winner.

Rather than exposing concurrency to the programmer and encouraging the use of parallel algorithms, the more successful approach has been to automatically expose parallelism in sequential code. Because C does not naturally support user-specified concurrency, such a technique is virtually mandatory for synthesizing efficient hardware from plain C. Unfortunately, these techniques are limited.

Finding parallelism in sequential code

There are three main approaches to exposing parallelism in sequential code, distinguished by their granularity. Instruction-level parallelism (ILP) dispatches groups of nearby instructions simultaneously. Although this has become the preferred approach in the computer architecture community, programmers recognize that there are fundamental limits to the amount of ILP that can be exposed in typical programs.[16] Adding hardware to approach these limits, usually through speculation, results in diminishing returns.

The second approach, pipelining, requires less hardware than ILP but can be less effective. A pipeline dispatches instructions in sequence but overlaps them—the second instruction starts before the first completes. Like ILP, interinstruction dependencies and control-flow transfers tend to limit the maximum amount of achievable parallelism. Pipelines work well for regular loops, such as those in scientific or signal-processing applications, but are less effective in general.

The third approach, process-level parallelism, dispatches multiple threads of control simultaneously. This approach can be more effective than finer-grained parallelism, depending on the algorithm, but process-level parallelism is difficult to identify automatically. Hall et al. attempt to invoke multiple iterations of outer loops simultaneously,[17] but unless the code is written to avoid dependencies, this technique might not be effective. Exposing process-level parallelism is thus usually the programmer's responsibility. Such parallelism is usually controlled through the operating system (for example, Posix threads) or the language itself (for example, Java).

Approaches to concurrency

The C-to-hardware compilers considered here take either of two approaches to concurrency. The first approach adds parallel constructs to the language, thereby forcing the programmer to expose most of the concurrency. SystemC, BDL, and Ocapi all provide process-level parallel constructs. HardwareC, Handel-C, SpecC, and Bach C additionally provide statement-level parallel constructs. SystemC's parallelism resembles that of standard hardware description languages (HDLs) such as Verilog, in which a system is a collection of clock-edge-triggered processes. HardwareC, Handel-C, SpecC, and Bach C's approaches are more like software, providing constructs that dispatch collections of instructions in parallel.

The other approach lets the compiler identify parallelism. Although the languages that provide parallel constructs also identify some parallelism, Cones, Transmogrifier C, C2Verilog, Catapult C, and CASH rely on the compiler to expose all possible parallelism. The Cones compiler takes the most extreme approach, flattening an entire C function with loops and conditionals into a single two-level combinational function evaluated in parallel. The CASH compiler takes an approach closer to compilers for VLIW processors, carefully examining interinstruction dependencies and scheduling instructions to maximize parallelism. None of these compilers attempts to identify process-level parallelism.

Both approaches have drawbacks. The latter approach places the burden on the compiler and therefore limits the parallelism achievable with normal, sequential algorithms. Although carefully selecting easily parallelized algorithms could mitigate this problem, such thinking is foreign to most software programmers and may be more difficult than thinking in an explicitly concurrent language.

The former approach, by adding parallel constructs to C, introduces a fundamental and far-reaching change to the language, again demanding substantially different thinking by the programmer. Even for a programmer experienced in concurrent programming with, say, Posix threads, the parallel constructs in hardware-like languages differ greatly from the thread-and-shared-memory concurrency model typical of software.

A good hardware specification language must be able to express parallel algorithms, because they are the most efficient for hardware. Its inherent sequentiality and often undisciplined use of pointers make C a poor choice for this purpose.

Which concurrency model the next hardware design language should employ remains an open question, but the usual software model—asynchronously running threads communicating through shared memory—is clearly not the one.

Timing

The C language is mute on the subject of time. It guarantees causality among most sequences of statements but says nothing about the amount of time it takes to execute each sequence. This flexibility simplifies life for compilers and programmers alike but makes it difficult to achieve specific timing constraints. C's compilation technique is transparent enough to make gross performance improvements easy to understand and achieve, and differences in efficiency of sequential algorithms is a well-studied problem. Nevertheless, wringing another 5% speedup from an arbitrary piece of code can be difficult.

Achieving a performance target is fundamental to hardware design. Miss a timing constraint by a few percentage points and the circuit will fail to operate or the product will fail to sell. Achieving a performance target under power and cost constraints is usually the only reason to implement a particular function in hardware rather than using an off-the-shelf processor. Thus, an adequate hardware specification technique needs mechanisms for specifying and achieving timing constraints.

This disparity leads to yet another fundamental question in using C-like languages for hardware design: where to put the clock cycles. Figure 8 shows a program fragment that is interpreted in at least three different ways by different compilers. Most of the compilers described here generate synchronous logic in which the clock cycle boundaries have been defined. There are only two exceptions: Cones and CASH. Cones only generates combinational logic; CASH generates self-timed logic.

Compilers use various techniques for inserting clock cycle boundaries, which range from fully explicit to fully implicit. Ocapi's clocks are the most explicit. The designer specifies explicit state machines, and each state gets a cycle. At some point in the SpecC refinement flow, the state machines are also explicit, although clock boundaries might not be explicit earlier in the flow. The clocks in the Cones system are also explicit, but in an odd way—because Cones generates only combinational logic, clocks are implicit at function boundaries. SystemC's clock boundaries are also explicit; as in Cones, the clock boundaries of combinational processes are at the edges, and in sequential processes, explicit wait statements delay a prescribed number of cycles. BDL takes a similar approach.

HardwareC lets the user specify clock constraints, an approach common in high-level synthesis tools. For example, the user can require that three particular statements should execute in two cycles. This presents a greater challenge to the compiler and is sometimes more subtle for the designer, but it allows flexibility that can lead to a better design. Bach C takes a similar approach.

```
for (i = 0 ; i < 8 ; i++) {
     a[i] = c[i];
     b[i] = d[i] || f[i];
}
```

Figure 8. It is not clear how many cycles it should take to execute this (contrived) loop written in C. Cones does it in one (it is combinational), Transmogrifier-C chooses eight (one per iteration), and Handel-C chooses 25 (one per assignment). Others, such as HardwareC, allow the user to specify the number.

Like HardwareC, the C2Verilog compiler also inserts cycles using fairly complex rules and provides mechanisms for imposing timing constraints. Unlike HardwareC, however, these constraints are outside the language.

Transmogrifier C and Handel-C use fixed implicit rules for inserting clocks. Handel-C's are the simplest: Each assignment and delay statement takes one cycle; everything else executes in the same clock cycle. Transmogrifier C's rules are nearly as simple: Each loop iteration and function call takes a cycle. Unfortunately, such simple rules can make it difficult to achieve a particular timing constraint. To speed up a Handel-C specification, assignment statements might require fusing, and Transmogrifier C might require loops to be manually unrolled.

The ability to specify or constrain detailed timing in hardware is another fundamental requirement. Whereas slow software is an annoyance, slow hardware is a disaster. *When* something happens in hardware is usually as important as *what* happens. This is another big philosophical difference between software and hardware, and again hardware requires different skills.

A good hardware specification language needs the ability to specify detailed timing, both explicitly and through constraints, but should not demand the programmer to provide too many details. The best-effort model of software is inadequate by itself.

Types

Data types are another central difference between hardware and software languages. The most fundamental type in hardware is a single bit traveling through a memoryless wire. By contrast, each base type in C and

C++ is one or more bytes stored in memory. Although C's base types can be implemented in hardware, C has almost no support for types smaller than a byte. (The one exception is that the number of bits for each field in a *struct* can be specified explicitly. Oddly, none of these languages even mimics this syntax.) As a result, straight C code can easily be interpreted as bloated hardware.

Compilers take three approaches to introducing hardware types to C programs. The first, and perhaps the purest, neither modifies nor augments C's types but allows the compiler or designer to adjust the width of the integer types outside the language. For example, the C2Verilog compiler provides a GUI that lets the user set the width of each variable in the program. In Transmogrifier C, the user can set each integer's width through a preprocessor pragma.

The second approach is to add hardware types to the C language. HardwareC, for instance, adds a Boolean vector type. Handel-C, Bach C, and BDL add integers with an explicit width. SpecC adds all these types and many others that cannot be synthesized, such as pure events and simulated time.

The third approach, used by C++-based languages, is to provide hardware-like types through C++'s type system. C++ supports a one-bit Boolean type by default, and its class mechanism makes it possible to add more types, such as arbitrary-width integers, to the language. The SystemC libraries include variable-width integers and an extensive collection of types for fixed-point fractional numbers. Ocapi, because it is an algorithmic mechanism for generating structure, also effectively takes this approach, letting the user explicitly request wires, buses, and so on. Catapult C presumably has a similar library of hardware-like types.

Each approach, however, is a fairly radical departure from C's call-it-an-integer-and-forget-about-it approach. Even the languages that support only C types compel a user to provide each integer's actual size. Worrying about the width of each variable in a program is not something a typical C programmer does.

Compared with timing and concurrency, however, adding appropriate hardware types is a fairly easy problem to solve when adapting C to hardware. C++'s type system is flexible enough to accommodate hardware types, and minor extensions to C suffice. A larger question, which none of the languages adequately addresses, is how to apply higher-level types such as classes and interfaces to hardware description. SystemC has some facilities for inheritance, but the inheritance mechanism is simply the one used for software; it is not clear that this mechanism is convenient for adding to or modifying the behavior of existing hardware. Incidentally, SystemC has supported more high-level modeling constructs such as templates and more elaborate communication protocols since version 2.0, but they are not typically synthesizable.

A good HDL needs a rich type system that allows precise definition of hardware types, but it should also assist in ensuring program correctness. C++'s type system is definitely an improvement over C's in this regard.

Communication

C-like languages are built on the very flexible RAM communication model. They implicitly treat all memory locations as equally costly to access, but this is not true in modern memory hierarchies. At any point, it can take hundreds or even thousands of times longer to access certain locations. Designers can often predict the behavior of these memories, specifically caches, and use them more efficiently. But doing so is very difficult, and C-like languages provide scant support for it.

Long, nondeterministic communication delays are anathema in hardware. Timing predictability is mandatory, so large, uniform-looking memory spaces are rarely the primary communication mechanism. Instead, hardware designers use various mechanisms, ranging from simple wires to complex protocols, depending on the system's needs. An important characteristic of this approach is the need to understand a system's communication channels and patterns before it is running because communication channels must be hardwired.

The problem with pointers

Communication patterns in software are often difficult to determine a priori because of the frequent use of pointers. These are memory addresses computed at runtime, and as such are often data dependent and cannot be known completely before a system is running. Implementing such behavior in hardware mandates, at least, small memory regions.

Aliasing, when a single value can be accessed through multiple sources, is an even more serious problem. Without a good understanding of when a variable can be aliased, a hardware compiler must place that variable into a large, central memory, which is necessarily slower than a small memory local to the computational units that read and feed it.

One of C's strengths is its flexible memory model, which allows complicated pointer arithmetic and essentially uncontrolled memory access. Although very useful for system programs such as operating systems, these abilities make analyzing an arbitrary C program's communication patterns especially difficult. The problem is so great, in fact, that software compilers often have an easier time analyzing a Fortran program than an equivalent C program.

Any technique that implements a C-like program in hardware must analyze the program to understand all possible communication pathways; resort to large, slow memories; or do some combination of the two.

Séméria, Sato, and De Micheli applied pointer analysis algorithms from the software compiler literature to estimate the communication patterns of C programs for hardware synthesis.[18] Although this is an impressive body of work, it illustrates the difficulty of the problem. Pointer analysis identifies the data to which each pointer can refer, allowing memory to be divided. Solving the pointer analysis problem precisely is undecidable, so researchers use approximations. These are necessarily conservative and hence might miss opportunities to split memory regions, leading to higher-cost implementations.

Finally, pointer analysis is a costly algorithm with many variants.

Communication costs

Software's event-oriented communication style is another key difference from hardware. Every bit of data communicated among parts of a software program has a cost (that is, a read or write operation to registers or memory), and thus communication must be explicitly requested in software. Communicating the first bit is very costly in hardware because it requires the addition of a wire, but after that, communication is actually more costly to disable than to continue.

This difference leads to a different set of concerns. Good hardware communication design tries to minimize the number of pathways among parts of the design, whereas good software design minimizes the number of transactions. For example, good software design avoids forwarding through copying, preferring instead to pass a reference to the data being forwarded. This is a good strategy for hardware that stores large blocks of data in memory, but is rarely appropriate in other cases. Instead, good hardware design considers alternate data encodings, such as serialization.

Communication approaches

The languages considered here fall broadly into two groups: those that effectively ignore C's memory model and look only at communication through variables, and those that adopt the full C memory model.

Languages that ignore C's memory model don't support arrays or pointers. Instead they look only at how local variables communicate between statements. Cones is the simplest; all variables, arrays included, are interpreted as wires. HardwareC and Transmogrifier C don't support arrays or memories. Ocapi also falls into this class, although arrays and pointers can assist during system construction. BDL is perhaps the richest of this group, supporting multidimensional arrays, but it doesn't support pointers or dynamic memory allocation.

Languages in the second group go to great lengths to preserve C's memory model. The CASH compiler takes the most brute-force approach. It synthesizes one large memory and puts all variables and arrays into it. The Handel-C and C2Verilog compilers can split memory into multiple regions and assign each to a separate memory element. Handel-C adds explicit constructs to the language for specifying these elements. SystemC also supports explicit declaration of separate memory regions.

Other languages provide communication primitives whose semantics differ greatly from C's memory style of communication. HardwareC, Handel-C, and Bach C provide blocking, rendezvous-style (unbuffered) communication primitives for communicating between concurrently running processes. SpecC and later versions of SystemC provide a large library of communication primitives.

Again, the difference between appropriate software and hardware design is substantial. Software designers usually ignore memory access patterns. Although this can slow overall memory access speed, it is usually acceptable. Good hardware design, in contrast, usually starts with a block diagram detailing every communication channel and attempts to minimize communication pathways.

So, software designers usually ignore the fundamental communication cost issues common in hardware. Furthermore, automatically extracting efficient communication structures from software is challenging because of the pointer problem in C-like languages. Although pointer analysis can help mitigate the problem, it is imprecise and cannot improve an algorithm with poor communication patterns.

A good hardware specification language should make it easy to specify efficient communication patterns.

Metadata

A high-level construct can be implemented in many different ways. However, because hardware is at a far lower level than software, there are many more ways to

Table 2. The big challenges for hardware languages.

Challenge	Comment
Concurrency model	Specifying parallel algorithms
Specifying timing	How many clock cycles?
Types	Need bits and bit-precise vectors
Communication patterns	Need isolated memories
Hints and constraints	How to implement something

implement a particular C construct in hardware. For example, consider an addition operation. A processor probably has only one useful addition instruction, whereas in hardware there are a dizzying number of different adder architectures—for example, ripple carry, carry look-ahead, and carry save.

The translation process for hardware therefore has more decisions to make than translation for software. Making many decisions correctly is difficult and computationally expensive. Furthermore, the right set of decisions varies with design constraints. For example, a designer might prefer a ripple-carry adder if area and power are at a premium and speed is a minor concern, but a carry-look-ahead adder if speed is a greater concern.

Much effort has gone into improving optimization algorithms, but it remains unrealistic to expect all these decisions to be automated. Instead, designers need mechanisms that let them ask for exactly what they want. Such designer guidance takes two forms: manual rewriting of high-level constructs into the desired lower-level ones (for example, replacing a "+" operator with a collection of gates that implement a carry-look-ahead adder) or annotations such as constraints or hints about how to implement a particular construct. Both are common RTL design approaches. Designers routinely specify complex data paths at the gate level instead of using higher-level constructs. Constraint information, often supplied in an auxiliary file, usually drives logic optimization algorithms.

Although it might seem possible to use C++'s operator-overloading mechanism to specify, for example, when a carry-look-ahead adder should implement an addition, using this mechanism is probably very difficult. C++'s overloading mechanism uses argument types to resolve ambiguities, which is natural when you want to treat different data types differently. But the choice of algorithm in hardware is usually driven by resource constraints (such as area or delay) rather than data representation (although, of course, data representation does matter). Concurrency is the fundamental problem. In software, there is little reason to have multiple implementations of the same algorithm, but it happens all the time in hardware. Not surprisingly, C++ doesn't support this sort of thing.

The languages considered here take two approaches to specifying such metadata. One group places it within the program itself, hiding it in comments, pragmas, or added constructs. The other group places it outside the program, either in a text file or in a database populated by the user through a GUI.

C has a standard way of supplying extra information to the compiler: the #pragma directive. By definition, a compiler ignores such lines unless it understands them. Transmogrifier C uses the directive to specify integer width, and Bach C uses it to specify timing and mapping constraints. HardwareC provides three language-level constructs: timing constraints, resource constraints, and arbitrary string-based attributes, whose semantics are much like a C #pragma. BDL has similar constructs.

SpecC takes the other approach; many tools for synthesizing and refining SpecC require the user to specify, using a GUI, how to interpret various constructs.

Constructs such as addition, which are low level in software, are effectively high level in hardware. Thus, there must be a mechanism for conveying designer intent to any hardware synthesis procedure, regardless of the source language. A good hardware specification language needs a way of guiding the synthesis procedure to select among different implementations, trading off between, say, power and speed.

WHY BOTHER generating hardware from C? It is clearly not necessary, because there are many excellent processors and software compilers, which are certainly the cheapest and easiest way to run a C program. So why consider using hardware? Efficiency is the logical answer. Although general-purpose processors get the job done, well-designed customized hardware can always do it faster, using fewer transistors and less energy. Thus, the utility of any hardware synthesis procedure depends on how well it produces efficient hardware specialized for an application. Table 2 summarizes the key challenges of a successful hardware specification language.

Concurrency is fundamental for efficient hardware, but C-like languages impose sequential semantics and require the use of sequential algorithms. Automatically exposing concurrency in sequential programs is limited in effectiveness, so a successful language requires explicit concurrency, something missing from most

C-like languages. Adding such a construct is easy, but teaching software programmers to use concurrent algorithms is difficult.

Careful timing design is also required for efficient hardware, but C-like languages provide essentially no control over timing, so the language needs added timing control. The problem amounts to where to put the clock cycles, and the languages offer a variety of solutions, both implicit and explicit. The bigger problem, though, is changing programmer habits to consider such timing details.

Using software-like types is also a problem in hardware, which wants to manipulate individual bits for efficiency. The problem is easier to solve for C-like languages. Some languages add the ability to specify the number of bits used for each integer, for example, and C++'s flexible type system allows hardware types to be defined. The type problem is the easiest to address.

Communication also presents a challenge. C's flexible global-memory communication model is not efficient for hardware. Instead, memory should be broken into smaller regions, often as small as a single variable. Compilers can do so to a limited degree, but efficiency often demands explicit control over this. A fundamental problem, again, is that C programmers generally don't worry about memory, and C programs are rarely written with memory behavior in mind.

A high-level HDL must let the designer provide constraints or hints to the synthesis system because of the wide semantic gap between a C program and efficient hardware. There are many ways to implement a construct such as addition in hardware, so the synthesis system needs a way to select an implementation. Constraints and hints are the two main ways to control the algorithm, but standard C has no such facility.

Although presenting designers with a higher level of abstraction is obviously desirable, presenting them with an inappropriate level of abstraction—one in which they cannot effectively ask for what they want—is not much help. Unfortunately, C-like languages, because they provide abstractions geared toward the generation of efficient software, do not naturally lend themselves to the synthesis of efficient hardware.

The next great hardware specification language won't closely resemble C or any other familiar software language. Software languages work well only for software, and a hardware language that does not produce efficient hardware is of little use. Another important issue will be the language's ability to build systems from existing pieces (known as IP-based design), which none of these languages addresses. This ability appears necessary to raise designer productivity to the level needed for the next generation of chips.

Looming over all these issues, however, is verification. What we really need are languages that let us create correct systems faster by making it easier to check for, identify, and correct mistakes. Raising the abstraction level and facilitating efficient simulation are two well-known ways to achieve this, but are there others? ■

Acknowledgments

Edwards is supported by the National Science Foundation, Intel, Altera, the SRC, and New York State's NYSTAR program.

■ References

1. *Handel-C Language Reference Manual*, RM-1003-4.0, Celoxica, 2003.
2. K. Wakabayashi and T. Okamoto, "C-Based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, Dec. 2000, pp. 1507-1522.
3. D.M. Ritchie, "The Development of the C Language," *History of Programming Languages-II*, T.J. Bergin Jr. and R.J. Gibson Jr., eds., ACM Press and Addison-Wesley, 1996.
4. G. De Micheli, "Hardware Synthesis from C/C++ Models," *Proc. Design, Automation and Test in Europe* (DATE 99), IEEE Press, 1999, pp. 382-383.
5. C.E. Stroud, R.R. Munoz, and D.A. Pierce, "Behavioral Model Synthesis with Cones," *IEEE Design & Test*, vol. 5, no. 3, July 1988, pp. 22-30.
6. D.C. Ku and G. De Micheli, *HardwareC: A Language for Hardware Design*, Version 2.0, tech. report CSTL-TR-90-419, Computer Systems Lab, Stanford Univ., 1990.
7. G. De Micheli et al., "The Olympus Synthesis System," *IEEE Design & Test*, vol. 7, no. 5, Oct. 1990, pp. 37-53.
8. D. Galloway, "The Transmogrifier C Hardware Description Language and Compiler for FPGAs," *Proc. Symp. FPGAs for Custom Computing Machines* (FCCM 95), IEEE Press, 1995, pp. 136-144.
9. T. Grötker et al., *System Design with SystemC*, Kluwer Academic Publishers, 2002.
10. P. Schaumont et al., "A Programming Environment for the Design of Complex High Speed ASICs," *Proc. 35th Design Automation Conf.* (DAC 98), ACM Press, 1998, pp. 315-320.
11. Y. Panchul, D.A. Soderman, and D.R. Coleman, *System for Converting Hardware Designs in High-Level*

Programming Language to Hardware Implementations, US patent 6,226,776, Patent and Trademark Office, 2001.

12. D.D. Gajski et al., *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, 2000.
13. R. Dömer, A. Gerstlauer, and D. Gajski, *SpecC Language Reference Manual*, Version 2.0, SpecC Consortium, 2001.
14. T. Kambe et al., "A C-Based Synthesis System, Bach, and Its Application," *Proc. Asia South Pacific Design Automation Conf.* (ASP-DAC 01), ACM Press, 2001, pp. 151-155.
15. M. Budiu and S.C. Goldstein, "Compiling Application-Specific Hardware," *Proc. 12th Int'l Conf. Field-Programmable Logic and Applications* (FPL 02), LNCS 2438, Springer-Verlag, 2002, pp. 853-863.
16. D.W. Wall, "Limits of Instruction-Level Parallelism," *Proc. 4th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 91), Sigplan Notices, vol. 26, no. 4, ACM Press, 1991, pp. 176-189.
17. M.W. Hall et al., "Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler," *Proc. Supercomputing Conf.*, IEEE Press, p. 49.
18. L. Séméria, K. Sato, and G. De Micheli, "Synthesis of Hardware Models in C with Pointers and Complex Data Structures," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 6, Dec. 2001, pp. 743-756.

Stephen A. Edwards is an associate professor in the Computer Science Department of Columbia University. His research interests include embedded-system design, domain-specific languages, and compilers. Edwards has a BS from the California Institute of Technology and an MS and a PhD from the University of California, Berkeley, all in electrical engineering. He is an associate editor of *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. He is a senior member of the IEEE.

■ Direct questions and comments about this article to Stephen A. Edwards, Dept. of Computer Science, Columbia University, 1214 Amsterdam Ave. MC 0401, New York, NY 10027; sedwards@cs.columbia.edu.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.

ADVERTISER INDEX SEPTEMBER/OCTOBER 2006

FUTURE ISSUE:

November-December 2006

Process Variation and Stochastic Design and Test

Advertising close date: 01 October 06

Advertising Personnel

Marion Delaney
IEEE Media, Advertising Director
Phone: +1 415 863 4717
Email: md.ieeemedia@ieee.org

Marian Anderson
Advertising Coordinator
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: manderson@computer.org

Sandy Brown
IEEE Computer Society,
Business Development Manager
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: sb.ieeemedia@ieee.org

Advertising Sales Representatives

Product Display
John Restchack
Phone: +1 212 419 7578
Fax: +1 212 419 7589
Email: j.restchack@ieee.org

Recruitment Display
Mid Atlantic (recruitment)
Dawn Becker
Phone: +1 732 772 0160
Fax: +1 732 772 0164
Email: db.ieeemedia@ieee.org

New England (recruitment)
John Restchack
Phone: +1 212 419 7578
Fax: +1 212 419 7589
Email: j.restchack@ieee.org

Southeast (recruitment)
Thomas M. Flynn
Phone: +1 770 645 2944
Fax: +1 770 993 4423
Email: flynntom@mindspring.com

Midwest/Southwest (recruitment)
Darcy Giovingo
Phone: +1 847 498-4520
Fax: +1 847 498-5911
Email: dg.ieeemedia@ieee.org

Northwest/Southern CA (recruitment)
Tim Matteson
Phone: +1 310 836 4064
Fax: +1 310 836 4067
Email: tm.ieeemedia@ieee.org

Japan (recruitment)
Tim Matteson
Phone: +1 310 836 4064
Fax: +1 310 836 4067
Email: tm.ieeemedia@ieee.org

Europe (recuirtment)
Hilary Turnbull
Phone: +44 1875 825700
Fax: +44 1875 825701
Email: impress@impressmedia.com