Silicon C:
A Hardware Backend for SUIF

C. Scott Ananian

May, 1998

High-level languages have much to offer the hardware designer. Freed of the tyranny of gates, it is possible to approach circuit function in terms of the algorithm or protocol it implements. Design rule checking and gate-level optimizations are becoming impossible for large designs without computer assistance anyway, the argument goes, so why not delegate all low-level design and synthesis to the machine, and free the humans to work on the high-level tasks the machine is incapable of?

In addition, a successful hardware compiler for a high-level language allows for more flexible hardware-software co-design and simulation. Ideally, a single high-level language could be used for both the application software and hardware. The model can be compiled completely in software for simulation or debugging, then easily partitioned to implement a subset of functions in hardware. If the compiler is retargettable, prototypes can be implemented using FPGA technology, and then the same source used to layout an ASIC when the design is finalized.

This work will explore the use of the general-purpose C programming language for hardware description, focusing on compiler issues. The implementation of a C-to-structural VHDL compiler is discussed, and we attempt to assess quantitatively the effect of C language features on hardware synthesis.

1 Hardware Description Languages

The two most common hardware description languages used are VHDL (the Very High Speed Integrated Circuit — VHSIC — Hardware Description Language) [8] and Verilog [12]. These are roughly equivalent: VHDL is mandated for Department of Defense work, while Verilog is more common in industry [2].

David Galloway writes:

> There is one compelling reason why a C language based hardware description language may prove successful: there are millions of C programmers in the world who might be quick to adopt the language. This is in contrast to the agony suffered by a C programmer trying to learn moden hardware description languages such as VHDL or Verilog. [4]

In Galloway's FPGA synthesis context, other benefits of C as an HDL are evident as well; flexible codesign/partitioning and integrated simulation foremost. In the FPGA-coprocessor context, it is very useful to be able to swap out arbitrary portions of a C-coded application to be compiled into custom hardware for speed. If the C program expresses the entire application process, a conventionally-compiled executable of identical sources may be used for debugging or execution on a machine without an FPGA coprocessor.

These properties prove desirable for more general hardware targets. In developing an application-specific integrated circuit or custom DSP, for example, it is very useful to be able to completely simulate both the hardware and the driving software in the same framework during development. This is facilitated by using a single general purpose language for application software and hardware description. Similarly, if testing of the preliminary design proves unsatisfactory, it is convenient to be able to repartition the software/hardware division of the system without having to recode portions of the combined model.[1]

The main drawback to such an approach is the expressivity of C. It is desirable to obtain a straightforward mapping between specification and hardware synthesis, but C contains several features which prove difficult to support on a hardware target. It may be noted, however, that C is not alone in this regard: both VHDL and Verilog contain "unsynthesizable" language constructs, as well. There is an IEEE working group (1076.3) charged with the creation of a standard synthesizable subset of VHDL; it is not unreasonable for us to define a similar subset of C for use with hardware targets.

In other cases, however, a *superset* of C's semantics seems warranted. The particular case this paper will address concerns non-standard bit-width integer types. In synthesis contexts, it is often known at specification time exactly how wide the representation of a given signal need be. Using a 16- or 32-bit C datatype to represent a 12-bit datapath seems wasteful, foreboding an inefficient translation to hardware. It appears that we ought to define an `int12_t` type for this case to correspond to C's standard `int8_t`, `int16_t` and `int32_t` types. We have resisted the temptation to add features to C, in order to preserve the ability to compile and simulate a unified hardware/software model with standard C tools.[2]

[1] These views are echoed by many in the FPGA community, for example [7]. Incidentally, the VHDL community agrees, but proposes VHDL as the new unified application/hardware language [11, xiii]. However, the slowness of current VHDL simulators/compilers makes this suggestion untenable.

2 Semantics of C HDL

When writing a hardware compiler for C, we must deal with the traditional bugaboo of any advanced compiler: the necessity of reconstructing an author's *intent* from their code. C is a sequential programming language designed for a general computational model; mapping it to hardware is often non-trivial. The standard tools of the parallelizing compiler must be deployed to reconstruct loop structure, induction variables, aliasing and data dependencies from the source, and perform loop unrolling and other optimizations to regenerate parallelism which the sequential structure of the input has hidden. Nevertheless, we can come up with a reasonably straightforward semantics for the translation of C code to hardware. This semantics is a superset of that described in [4].

First, we declare that all straight-line code executes in zero time. In the example of figure 1, the externally visible value of signal `a` jumps directly to 5 when the block is executed; it does not transition through 3. Conceptually one can imagine that there is an entirely separate variable representing the externally visible value — `a_ext`, say — which is assigned the value of `a` just once, at the end of the block. This rule allows us to synthesize straight-line code as a combinational logic block.

Branching and looping code then naturally define sequence points, where we can determine an order-

[2] Object-oriented techniques prove to be useful (see section 6), but here also standard C++ can be used. Compare the work in [7], which shares our unified modelling goals.

```
{
  /* 'a' is an externally visible signal */
  a = 3;

  ...straight-line code...

  a = 5;
}
```

Figure 1: Example C code illustrating concurrent execution semantics.

```
{
  /* 'a' is an externally visible signal */
  int i;
  a = 0;                  /* pre-loop    */
  for (i=1; i<5; i++)
    a += i;               /* in-loop     */
  a = 20;                 /* post-loop   */
}
```

Figure 2: Example C code illustrating sequence point semantics.

ing to the combinational blocks. These sequence points delimit separate states in a synthesizable state machine. Thus, the hardware described by figure 2 corresponds to a three-state machine, with states corresponding to pre-loop, in-loop, and post-loop conditions. We remain in the in-loop state until we exit the loop, either via `break` or the loop test. The externally visible values of `a` are: 0, while in the pre-loop state; 1, 3, 6, 10, and 15, while in the in-loop state; and 20 while in the post-loop state. It takes 6 synchronous clock cycles for `a` to transition from 0 to 20.[3]

Note that unrolling this loop will change the externally visible states of the machine, and so unrolling should proceed cautiously. For predictability, it is best not to unroll any loops by default, and allow the programmer to specially annotate functions which they desire to be unrolled. Performance considerations, however, may suggest rather that all internal states be opaque, allowing the compiler maximum scheduling flexibility. Our work inclines toward the latter view.

Our implemented semantics map functions to hardware objects. The input and output ports of the synthesized hardware correspond to the function parameters and return value, respectively. Nested functions correspond to hierarchical hardware composition; recursive functions thus refer to infinitely extended hardware, and are disallowed.[4] Using a function for a frequently reused subprogram does not save hardware in the same way it saves code size, but can serve as a useful hint to an optimizing hardware compiler that this particular logic block may be profitably multiplexed on its idle cycles.

Static variables and their associated state map in an obvious way to registers. Arrays, pointers, and their associated memory model can sometimes be localized and mapped to a RAM structure in hardware (especially for small fixed-size arrays), but often pointer arithmetic requires an full-fledged memory interface to implement properly. The behavior of these constructs is implementation dependent, but discouraged. The general rule is that non-obvious mappings are disallowed: if the programmer cannot

[3]Note that the externally-visible behavior in the current-implementation is somewhat different: the first and last loop iterations execute concurrently with the pre-loop and post-loop code, respectively, so that the externally-visible states for the example would be: 1, 3, 6, 10, and 20. See section 4 and figure 6 for details.

[4]A more refined, if complex, semantics would translate recursive functions into their synthesizable equivalent by rewriting recursion as loops whenever possible. This strategy suffers from somewhat unpredictable behavior.

easily visualize how a construct will map into hardware, then it is very difficult for the compiler and programmer to agree on the correctness of the result.

3 Limitations

The above semantics show some inherent limitations of C as a hardware description language. At the most basic level, C is defined with a general memory and computation model that does not hold for hardware. This leads to difficulties implementing, for example, pointer arithmetic. Its operator and type set are overrich for the target; also, ingrained conventions of C coding (the use of strings and `char *`) are essentially meaningless in a hardware context.

More serious, though, is the lack of a timing grammar to define input/output behavior. These constraints must be specified in some manner outside the scope of the language, or by strong reliance on the inherently synchronous definition of our looping constructs. We would prefer to eliminate the notion of external visibility at sequence points and allow flexible unrolling, retiming, register insertion/deletion, and other sequence optimizations subject only to user-defined time/cycle contraints. This requires some external timing specification in addition to the C source.

Also, function calls are very inflexible models of hardware. The single return value restriction is especially difficult. This can in large measure be alleviated by moving to an object-oriented programming language/style; this is discussed further in section 6.

Other limitations are specific to the current implementation, although many are likely to be found in most or all implementations of these semantics. Floating point math is extremely expensive in hardware and very difficult to do combinationally, as our semantics mandate. Therefore we have omitted support for floating point types in our implementation.[5]

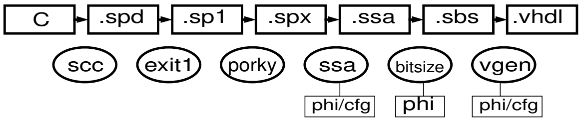


Figure 3: Hardware compiler pipeline stages

Integer addition and subtraction are easy, but multiplication and division are, again, very expensive to perform combinationally. Division is not supported in this implementation, but multiplication is, with the warning that it is not generally a good idea.

This implementation does not attempt to support arrays and pointer arithmetic in any form, although the previous section suggested ways in which it might be done. Similarly, there is no support for static variables in this implementation, for reasons of simplicity only.

4 Implementation

The C hardware compiler discussed in this paper was implemented using the SUIF (Stanford University Intermediate Format) compiler system [13]. Translation from C to structural VHDL takes place in a six stage pipeline, using two code libraries to extend the base SUIF system. Except for the first and last, every module reads and writes the SUIF intermediate representation, annotated with the various pieces of information collected at each stage. Figure 3 shows the pipeline schematically.

The standard SUIF front-end, `scc`, is used to generate a structured intermediate-format file from C source code. The output from this stage is given an `.spd` suffix.

[5] See [10] for information on what an implementation would entail.

The next module is named `exit1`. Its job is to create single-entry, single-exit functions from the general SUIF code output by `scc`. It creates a new label and a return opcode at the end of each function, using a new "return value" variable. It then replaces return statements through the code with `cpy` and `jmp` opcodes. This code restructuring makes it easier for the VHDL-generation stage to synthesize the function output port. The output of `exit1` is given an `.sp1` extension.

The SUIF optimizer, `porky`, is then used to convert the `.sp1` file into low-SUIF by breaking down high-level for, loop, if, block, and mbr structures. This simplifies the VHDL generation phase by reducing the number of different opcodes it must translate. We also perform constant propagation, common subexpression elimination, and dead-code elimination to reduce the amount of generated logic. These optimizations could all be performed by a aggressize logic optimization after synthesis, but at a greatly increased cost. Finally, `porky` does jump optimization to remove any inefficiencies introduced by the simple-minded `exit1` pass. The optimized low-SUIF output is given an `.spx` extension.

The `ssa` module then translates the intermediate representation into Static Single Assignment (SSA) form [3]. A control flow graph of basic blocks is generated from the input, using a version of the `cfg` library from the `machsuif` package highly modified to handle low-SUIF instead of machine-SUIF. The dominator tree and dominance frontier are then computed and used to place phi functions on the control flow graph according to [1]. The variables in the program are renamed and the phi function information written out as annotations to the SUIF instruction objects. The `phi` library manages the phi function information, which is reused in all following stages. The output of this stage is given the extension `.ssa`.

Modules to optimize the intermediate representation to generate more efficient hardware fit into the pipeline after the conversion to SSA form. One such module was implemented, which attempts to narrow the bit-widths of variables in order to generate more efficient datapaths. This `bitsize` module will be discussed in section 5.

```
a = b + c;
d = a * (e - f);
```

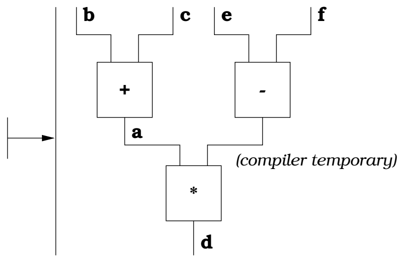


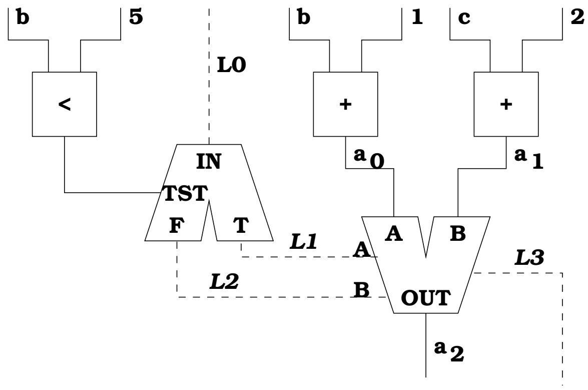
Figure 4: Hardware generated for straight-line code.

The final module, `vgen`, generates structural VHDL code from an SSA-annotated input file. It is worth noting that the structural VHDL produced is merely a convenient notation for describing logic at a module and netlist level; VHDL is a language much more capable than the use to which we are putting it.

The rules for generating logic from straight-line SSA form input are illustrated in figure 4. Each basic three-register instruction maps directly to a hardware functional unit; its source and destination registers indicate connections to the ports of the unit.

Each basic block has an additional signal associated with it, which we call the "execution bit." This bit is set to pass the flow of control, and corresponds to control flow graph edges. Branch instructions switch the bit according to their target, and phi functions become multiplexers controlled by the bits. Figure 5 illustrates.

Back edges in the control flow graph have registers added in-line, as figure 6 illustrates. Note that the input execution bit (L_0) is only active for one clock cycle; consequently the output execution bit (L_2) is also active for only one cycle. This is the minimum

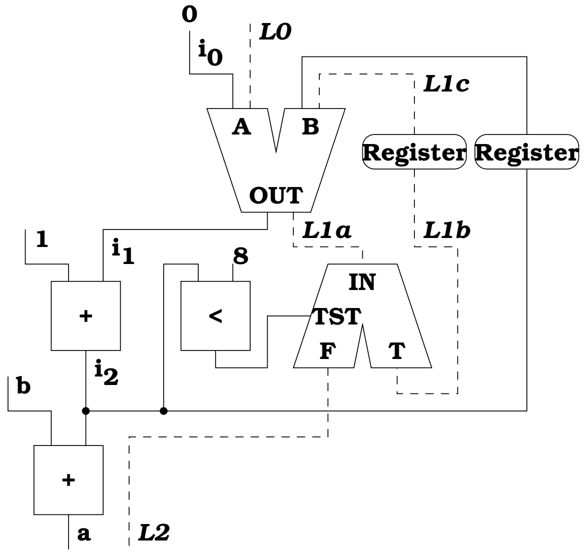



```
L0:  if (b < 5)
L1:       a0 = b + 1;
     else
L2:       a1 = c + 2;
L3:  a2 = φ(a0,a1);
```

Figure 5: Hardware translation of an if-else statement in SSA form, using execution bits (dashed lines).

cycle implementation of the simple loop structure of the figure; the first loop iteration is overlapped with the evaluation of pre-loop code, and the post-loop code executes on the same clock cycle as the final loop iteration. The execution bits form a one-hot encoding [6] of the loop state in a simple finite-state machine.

The function prototype becomes an entity and port description for the generated logic.




```
L0:  i0 = 0;
L1:  i1 = φ(i0,i2);
     i2 = i1 + 1;
     if (i2 < 8) goto L1;
L2:  a = i2 + b;
```

Figure 6: Hardware translation of a simple for-loop, illustrating the handling of control-flow backedges. L_{1a} is the logical OR of L_0 and L_{1c}.

5 Optimizations

Hardware generated from a high-level language can be optimized in a number of ways. Many high-level optimizations on the source language will translate to optimized hardware, but some high-level optimizations may be hardware-neutral or even generate less-efficient hardware. Jump optimizations, for example, are hardware-neutral because jump instructions do not create hardware at the synthesis step: they merely control the naming and routing of execution bits. Similarly, optimizations designed to ease register pressure or other register-renaming optimizations will have no effect once the code has been translated to SSA form. Common subexpression elimination, on the other hand, has the same effect on hardware as it does on the source language: it reduces the number of computations necessary to achieve a result.

Certain optimizations may be done at either a high or low level. Source-language common subexpression elimination may alternatively be performed as part of a logic-reuse algorithm at a low level, and dead-code elimination can be done by pruning unused logic. In fact, the low-level optimizations may be able to take advantage of bit-level structure or the logic implementation of certain operators to optimize details invisible to a high-level analysis. However, the low-level optimizations will invariably be much slower than their high-level equivalents, due to the larger dataset they must process and the often-exponential running time of logic optimization algorithms. Hence, we want to perform our optimizations at a high level whenever possible. We also hope to take advantage of source-language information which may be lost in translation; induction-variable transformations, for example, may be very difficult to perform at a logic level.

As mentioned in section 4, the SUIF optimizer `porky` was used to perform a variety of general-purpose high-level optimizations on our input source. We were also interested in examining high-level *hardware-specific* optimizations, and in particular whether some of the disadvantages of C as a hardware description language could be overcome by intelligent optimization.

The `bitsize` module implemented in this work was designed to address C's inflexible type system. In hardware design, the exact bitwidth of various datapaths is often known exactly, and inefficiencies are introduced when this datapath is expressed using one of C's fixed-width integer types. The `bitsize` module attempts to recreate the bitwidth information using the arithmetic properties of the computation performed. All constants are given exact bitwidths, and the result of combining variables and constants with a given operator can often be assigned a maximum bitwidth as well. The SUIF type information for all variables was then modified to represent the newly computed restricted bitwidths.

The current implementation does not take advantage of all possible bit-width information; it implements only a def-use analysis on the SSA-format input. Further optimization is often possible by examining the control flow graph. For example, analysis of the code in figure 6 could determine that i_2 was always 3 bits wide if the condition at the bottom of the loop was true and i_2 was an unsigned type. Since i_0 is known to be a 1 bit type, this could be used to assign i_1 a width of 3 bits as well. i_2 would be pushed up to 4 significant bits by the increment statement, but its value at L_1 would remain at 3 bits because of the loop condition. This optimization requires splitting variables at branches to differentiate between the width of i_2 at L_1 (three bits) and its width at L_2 (four bits). If the width of i_0 was unknown, the loop may be rewritten to allow the branch information to narrow the variable width in all but the first iteration. This and other sophisticated width optimizations are possible improvements that were not implemented in the current codebase.

In an attempt to evaluate the `bitsize` module on a realistic hardware description, it was fed the main routine of a PDP-8 simulator written for an earlier project. The PDP-8 has a twelve bit datapath, and various methods were used in the code to truncate results to obtain an accurate simulation. The unmodified code also contains string operations to output the machine state to the console during simulation; it is to be expected that these operations would not benefit from the `bitsize` module, but they would also not typically be synthesized into hardware.

The `bitsize` module managed to trim 315 of the 1,466 compiler variables in the `ExecuteNextInstruction` function; this

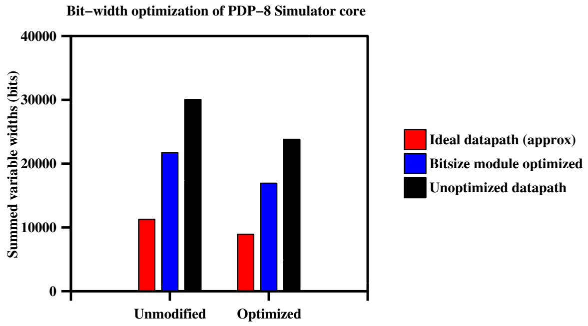


Figure 7: Performance of `bitsize` optimization.

corresponded to reducing the summed datapath width of all variables from 21,715 to 30,056 bits (eliminating 28% of the bit paths). Elimination of the string manipulation code changed these numbers to 261 of 1,129 variables, for a new sum datapath width of 16,932 bits from 23,808 bits (eliminating 41% of the bits). The 12-bit datapath was defined using C 32-bit `int` types, so a 'perfect' reduction would have eliminated 60% of the datapath bits. See figure 7, and note that the actual machine state variables were outside the analyzed routine, so they propagated their full width through the calculations. By adding analysis of static variables, it is expected that compiler bitwidth optimization can approach the ideal closely.

6 Conclusions and future work

Although C is still somewhat expressivity-limited as a hardware description language, it is anticipated that most of its limitations can be overcome by a combination of well-defined semantics and intelligent high- and low-level optimization. In particular, optimizations targetting C's inflexible type system seem to show that most of the inefficiencies of fixed-width data types can be optimized away using a fast al-

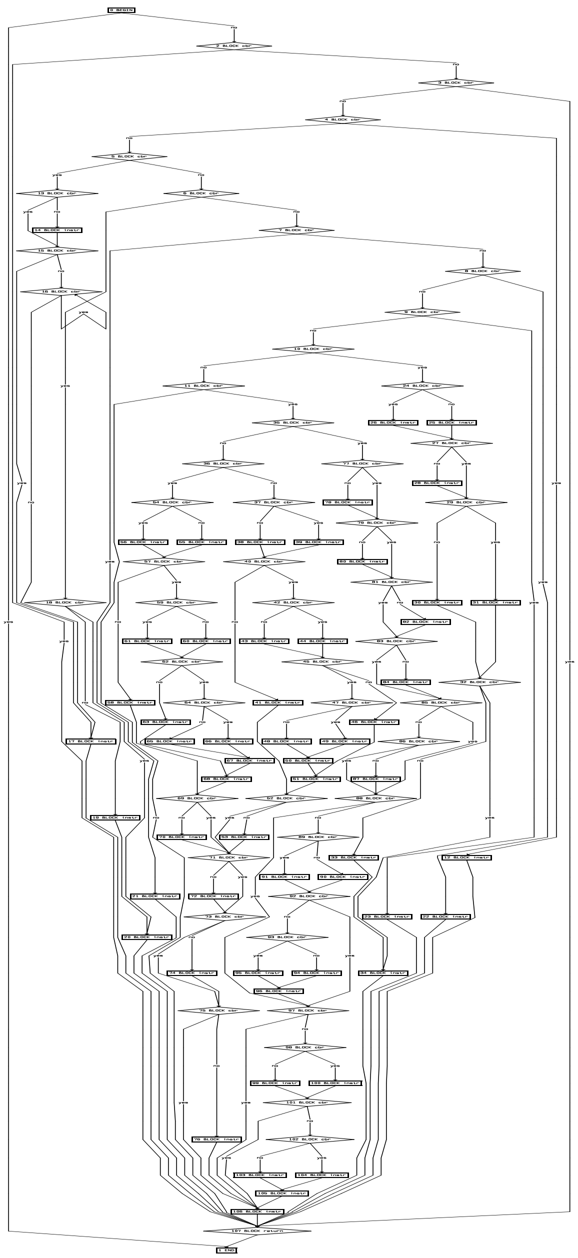


Figure 8: Sample graph output from the `cfg` library for the optimized `ExecuteNextInstruction` function.

gorithm. Standard C optimization techniques, such as constant-propagation and common subexpression elimination, can be used profitably to generate optimized hardware as well.

Certain C features still hamstring its expressivity as an HDL. Foremost of these in the present work is C's limiting single-value-return function syntax. Hardware blocks often have multiple outputs, but it is difficult to express this using C syntax. However, the current syntax *does* provide a clean in/out marking of signals. However, it has difficulty dealing with bidirectional signals.

The solution to these problems seems to lie in a more object-oriented approach. If the basic hardware description unit is a *class*, instead of a function, we can define multiple interface functions to internal state maintained in class member variables. The added semantics make hierarchical composition easier, as well. The semantics are an easy superset of the ones described for C in this paper, and are the subject of current work. However, the SUIF system has limited support for object-orientation at the present time, so the continuation of this work is currently being implemented in Java. Open questions on the semantics still remain, among them the expression of timing and interface constraints in C or its cousins. At the moment, an auxillary interface description language is being used to express these constraints; it would be better if this information could be integrated into the source language.

A Graphing extensions to the `cfg` library

As part of the modifications to the control-flow graph library, an existing output routine was extended to produce input suitable for the `vcg` graph-layout tool [9]. Illustrative output from the optimized `ExecuteNextInstruction` code evaluated in section 5 is included as figure 8.

References

[1] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1997.

[2] `comp.lang.verilog` alternate FAQ. `http://www.comit.com/~rajesh/verilog/faq/alt_FAQ.html`, April 1998. Version 6.1.

[3] Ron Cytron et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] David Galloway. The Transmogrifier C hardware description language and compiler for FPGAs. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 136–144, April 1995.

[5] Maya Gokhale and Brian Schott. Data-parallel C on a reconfigurable logic array. *Journal of Supercomputing*, 9(3):291–313, 1995.

[6] Steve Golson. State machine design techniques for Verilog and VHDL. *Synopsys Journal of High-Level Design*, September 1994.

[7] Christian Iseli and Eduardo Sanchez. A C++ compiler for FPGA custom execution units synthesis. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 173–179, Napa Valley, CA, April 1995.

[8] Andrew Rushton. *VHDL for Logic Synthesis: An Introductory Guide for Achieving Design Requirements*. McGraw-Hill, 1995.

[9] Georg Sander. Graph layout through the VCG tool. In *Proceedings of the 1994 DIMACS International Workshop on Graph Drawing*, pages 194–204, Princeton, NJ, October 1994. Springer-Verlag.

[10] Nabeel Shirazi, Al Walters, and Peter Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 155–162, Napa Valley, California, April 1995.

[11] Stefan Sjoholm and Lennart Lindh. *VHDL for Designers*. Prentice Hall, 1997.

[12] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic, 3rd edition, June 1996.

[13] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, December 1994.

[14] Cliff Young. *The SUIF Control Flow Graph Library*. Harvard University. Included with the `machsuif` library.