Productivity Engineering in the UNIX† Environment

An Advanced Silicon Compiler in Prolog

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871

†UNIX is a trademark of AT&T Bell Laboratories

An Advanced Silicon Compiler in Prolog

William R. Bush, Gino Cheng, Patrick C. McGeer, Alvin M. Despain

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley 94720

The Advanced Silicon Compiler in Prolog (ASP) is a full-range synthesis system based on Prolog. It produces VLSI masks from instruction set architecture specifications written in Prolog. The system is composed of several hierarchical components that span behavioral, circuit, and geometric synthesis. While small microprocessors have been synthesized, ASP is still under development. In general, Prolog appears adequate for both specification and implementation.

1. Introduction

We have two goals with the ASP project. One is to generate good microprocessor designs rapidly, as a tool for the architectural research being pursued by the Aquarius project [1]. The other is to understand the benefits and liabilities of using Prolog for large software systems in general and CAD in particular.

This paper describes the approach and components of the ASP system, and our experience with Prolog.

2. Decomposition of Silicon Compilation

A full behavior-to-silicon compiler is a complex undertaking. We decompose the silicon compilation problem into three abstract problem domains, ordered hierarchically (see [2] and [3], for other similar decompositions).

The top level of our system is the behavioral domain. This level generates a data path (a set of functional units), controlled by a finite state machine, from an input specification written in Prolog (see Figure 1). Both standard compiler techniques and hardware-specific knowledge are used in this process. This behavioral synthesis task is performed by the Viper component of ASP.

The second level is the circuit or functional domain. The purpose of this domain is to present the behavioral component with abstract components (for example, see Figure 2). Hence, this level attempts to synthesize and connect the finite state machine and functional units generated by the behavioral level. This level encompasses the traditional tasks of state assignment, logic synthesis, transistor sizing, placement and routing, and module generation. The core of this level is module generation, which is done by the Topolog component. We also have a CMOS PLA generator called Plague, a transistor sizer named Most [4], and a channel router called Char.

The third level is the geometric domain. The purpose of this domain is to present the programs of the functional domain with an abstract class of idealized elements, or a sticks-and-elements virtual-grid abstraction of the actual mask layers involved in fabricatable design (for an example, see Figure 3). This domain encompasses the traditional tasks of compaction and device-level simulation. These tasks are accomplished by the Sticks-Pack component of ASP. See Figures 4 and 5 for example final layout.

Clearly there is some interaction between the levels. No layout generator can ignore the constraints inherent in technology, such as, for example, the richer connectivity of two layers of metal. Similarly, the data path constructor can only use functional units that the module generator can generate.

3. Viper

Viper generates structural hardware descriptions from instruction-set level specifications written in standard Prolog. It performs two basic functions. It translates Prolog constructs into hardware equivalents, and it creates and allocates hardware resources within various constraints.

It uses a combination of compiler analysis and hardware knowledge. Algorithmic compiler techniques -- dependency analysis, register allocation, and dependency-based scheduling -- are used to produce a basic design with constraints. Hardware specific heuristics and knowledge about the characteristics of functional units are then used to generate a design within the constraints.

Viper operates in four phases: register allocation, translation of Prolog into an RTL-based form, data path construction, and structural description generation.

The first phase operates on an input specification written in Prolog and constrained to a style illustrated in Figure 1. First, the microprocessor must be a finite state machine as indicated by the first clause. Second, the model of memory is assumed to be external to the microprocessor, and is realized in Prolog with assert and retract. The first phase transforms an input specification into an equivalent Prolog program in which variable references have been replaced by assertions involving global data structures that model registers. As with the original specification, the transformed specification can be executed directly. It also transforms assert and retract into memory references, while providing a system-defined memory interface.

The second phase converts Prolog goals to register transfers, assigns transfers to FSM states, and produces a state transition table. The operations appearing in transfers are Prolog operators, such as '+', not yet bound to functional unit operations. The schedule of transfers is maximally parallel, based only on dependencies between values and not on resource constraints.

The third phase produces a constrained data path, mapping abstract operators to functional units and minimizing the connections between units. If the system cannot find an available functional unit it tries to extend the functionality of an existing one, for example by converting a register used in an increment expression into a counter (providing enabling conditions are met).

Knowledge about functional units is packaged in a library, which also serves as the interface to lower synthesis levels. Each member of the library contains knowledge, in the form of Prolog assertions, about when and how it should be synthesized. This approach is similar in spirit to [5], but is not object-oriented in implementation. Each library member also contains the logic equations and other information necessary for it to be realized as a circuit.

The fourth phase generates a structural description containing a connected data path and control path. Figure 2 presents the data path derived from the specification in Figure 1, consisting of named instances of functional unit types along with connected input and output buses and control signals. Functional unit implementation is deferred to Topolog.

4. Topolog

Topolog is the module generator, layout engine, and circuit database manager. It takes in a description of a circuit to be generated, constraints on the bounding box, and a set of ports, and outputs a sticks-based layout description which can be converted to a fabricatable form by the mask-level design environment, Sticks-Pack.

Topolog combines the functions of a module generator and layout engine in the hope that a combination of these tools may solve problems specific to each. In particular, the availability of a layout engine permits the module generators to specify a module as a collection of functional blocks rather than pieces of geometry, which significantly simplifies the problem of specifying components of a module. The module generator is freed from most concerns of geometry, routing and placement, secure that the layout engine will solve the routing and placement problem. Similarly, the collection of circuit elements into modules provides valuable information to those automated placement tools which either implicitly or explicitly partition a circuit into connected subcircuits.

Topolog is designed around the basic abstraction of a *block*. A block represents a primitive circuit element. A block has a *p-side* and an *n-side*. Topolog's basic function is to group blocks into rows, and to route signals between the blocks. A single *routing channel* runs between the p- and n-side of any row; a power bar runs above the p-side of every row, and a ground bar runs beneath the n-side of any row. Odd rows are flipped about the horizontal axis so that power and ground bars may be shared between rows. It is tempting to consider Topolog as a standard cell layout program, but this is quite misleading. Since blocks can be anything which shares the characteristics mentioned here, it is more accurate to describe Topolog as a gate matrix style layout engine.

Topolog has a six stage pipeline. After inputs are parsed, a preliminary generation of all the blocks is done. The blocks are then grouped into rows, and placed within rows. During this placement phase, compound blocks are expanded into their primitive component blocks. Detailed generation of blocks is done; the blocks are fleshed out into a sticks-and-elements description, and the pins for channel routing are defined. The channel is then routed. Finally the package is output. An example is shown in Figure 3, which is a bit slice derived from the data path description in Figure 2. Our existing logic blocks are all designed by the well-known Uehara-Van Cleemput procedure [6]. The UVC algorithm has been shown to derive near-minimal-width single-diffusion-strip static CMOS arrays.

Topolog currently supports four types of blocks: static CMOS and-or-invert gates, domino CMOS gates, pass gates and transmission gates. Topolog is designed to support any circuit style or technology that can be expressed in the style described above. The terms *p-side* and *n-side* refer to p- and n-diffusion regions, reflecting our primary concern with CMOS technology; however, there is no reason, in principle, to use these regions specifically for these purposes. One can imagine, for example, using Topolog for NMOS designs using the p-side for the enhancement device. The addition of a new circuit type is quite easy, due to Prolog's clause-based programming style. The library routines have so far proved powerful enough to make the addition of new circuit types almost automatic: the addition of domino CMOS required only 30 lines of new Prolog code.

5. Sticks-Pack

The Sticks-Pack environment consists of a technology independent compactor that creates spaced layout and simulation files from sticks-and-elements descriptions, a joiner that joins together cells generated by the compactor, and a simulator that simulates sticks-based cells.

The Sticks-Pack compactor takes a cell defined in the sticks-and-elements representation used by Topolog (see Figure 3), and creates a mask level representation for the cell. A new compaction technique is employed which is both algorithmic and rule based. An algorithm similar to zone refining [7] is used to perform a rough spacing of the elements. Floor and ceiling profiles for each layer of material are maintained. Elements from the ceiling are moved directly across the molten region to the floor, where spacing requirements are calculated, and diagonal constraints are noted. Rules are used to shift the elements to better fit their environment. For each cell, a connectivity file containings nodal connectivity, resistivity and capacitance information is generated for the switch-level simulator and for the Spice circuit simulator. The Sticks-Pack compactor is relatively technology independent. It supports an arbitrary number of layers, and elements such as transistors and contacts are defined from a set of primitives. A design rule file and a set of technology dependent rules are specified for each technology.

Large layouts in Sticks-Pack are realized by joining small cells together. Leaf cells (cells of the lowest level consisting of transistors and wires) are compacted individually and constitute the building blocks for larger modules. Previous tilers have either pitchmatched or river routed cells [8]. The joiner program connects signals between cells by either pitchmatching or river routing, whichever is more area efficient. The joiner operates in the physical domain rather than the virtual grid domain for tighter results. This also allows cells of various virtual grid heights and widths to be joined.

6. Other Components

We have a boolean equation generator that takes the finite state machine description produced by Viper and does state assignment and generates the equations used by Plague. We hope in the future to take advantage of current logic synthesis work [9].

Plague is our CMOS PLA generator, which creates AND-OR sticks-and-elements PLA's from boolean equations.

We have a left-edge-first channel router, called Char, for connecting the major blocks of the system, primarily the data path and control path.

In an effort to improve the performance of our designs, we have a Prolog-based transistor sizer name Most [4], which currently runs standalone, but which will be integrated with Topolog.

7. The Use of Prolog

The use of Prolog for both specification and implementation arose from experience using and implementing Prolog in both a compiler and new execution engine. Our experience with Prolog in ASP has in general been positive.

7.1. The Use of Prolog for Implementation

We have observed several benefits in using Prolog for implementation.

(1) Prolog's database properties have aided the production and processing of information. The relations that the system generates are much better expressed in that form than in the usual compiler hash table structures. Prolog itself is therefore the database manager for our low-level cell design environment Sticks-Pack, which gives us a simple solution to what is, for most systems, a major part of the silicon compiler design and implementation effort.

(2) Prolog's rule-based environment has made heuristics easy to implement. Most of the system is in fact algorithmic, and a general heuristic approach has been avoided, but heuristics are used in a few local contexts.

(3) Prolog's unification of the concepts of data and procedure call lets us use module libraries in a natural way; it also leads to a very simple and elegant mechanism for user-programmability of (for example) our module generator.

On the other hand, without a sophisticated debugger, Prolog, with its failure and backtracking semantics, has been hard to debug. Similarly, Prolog code is hard to modify without careful redesign. We have found that these difficulties may be overcome by the use of appropriate extensions embedded in the language. In particular, we have found that the implementation of a data structuring package and primitives which simulate backtrackable assignment have made the implementation and modification of Prolog programs much easier.

7.2. The Use of Prolog for Specification

Prolog is used for specification because of its logical basis and declarative nature [10]. Specifications are executable in Prolog, and thus can be simulated without a simulator. Since Prolog does not have explicit hardware constructs, both hardware structures and parallelism information must be derived by the system. The microprocessor focus of the system has allowed us to ignore some specification issues -- we are not concerned with the specification or synthesis of multichip, asynchronous, bit serial, or analog designs. For clarity and implementation simplicity we require Prolog specifications to be determinate (without backtracking); we only implement determinate FSM's.

Specification in Prolog has turned out well so far, for a number of reasons [11].

(1) Control in Prolog is simple (ignoring backtracking), and maps easily into hardware. The user's conceptualization and the system's realization are very similar.

(2) The derivation of information (such as concurrency constraints and register bindings) that in another language might be explicit has not been difficult.

(3) Clauses tend to be short and well modularized, lending themselves to easy translation.

(4) Prolog's simple structure and syntax facilitate automatic generation of Prolog specifications.

8. Conclusions

ASP is still being developed. It currently consists of about 10,000 lines of Prolog code. We have so far been surprised at how effective Prolog has been for specification and synthesis. The system has been relatively easy to develop and modify. We hope to use it to produce the next generation of our Prolog microprocessor engines.

9. Acknowledgements

This work was sponsored in part by Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871, monitored by Space & Naval Systems Warfare Command under Contract No. N00039-84-C-0089. We are indebted to Carlo Sequin and Glenn Adams, the authors of the Topogen and Topogate programs, of which Topolog is a direct intellectual descendant [12].

References

[1] 'Aquarius – A High Performance Computing System for Symbolic/Numeric Applications'; A.M. Despain, Y.N. Patt; *COMPCON 85.*

[2] 'Synthesizing Circuits from Behavioral Level Specifications'; W. Rosenstiel, R. Camposano; *Computer Hardware Description Languages and their Applications (CHDL 85 Proceedings)*; 1985; pp. 391-403.

[3] 'OCCAM to CMOS: Experimental Logic Design Support System'; T. Mano, F. Maruyama, K. Hayashi, T. Kakuda, N. Kawato, T. Uehara; *Computer Hardware Description Languages and their Applications (CHDL 85 Proceedings)*, 1985.

[4] 'Delay Reduction Using Simulated Annealing', J. Pincus, A.M. Despain; *23rd Design Automation Conference, 1986.*

[5] 'Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions'; M.C. McFarland; *23rd Design Automation Conference, 1986.*

[6] 'Optimal Layout of CMOS Functional Arrays'; T. Uehara, W. van Cleemput; *IEEE Transactions on CAD*, 1981.

[7] 'Two Dimensional Compaction by Zone Refining'; H. Shin, A. Sangiovanni-Vincentelli, C. Sequin; *23rd Design Automation Conference, 1986.*

[8] 'Virtual Grid Symbolic Layout', N. Weste, *18th Design Automation Conference, 1981.*

[9] 'Efficient, Stable Algebraic Operations on Logic Expressions'; P.C. McGeer; *VLSI 87.*

[10] *Programming in Prolog*; W.F. Clocksin, C.S. Mellish; Springer-Verlag, 1981.

[11] 'Experience with Prolog as a Hardware Specification Language'; W.R. Bush, G.Y. Cheng, P.C. McGeer, A.M. Despain; *Symposium on Logic Programming, 1987.*

[12] 'Design and Layout Generation at the Symbolic Level'; C.H. Sequin; in *Proceedings of the Summer School on VLSI Tools and Applications*; W. Fichtner and M. Morf, editors, Kluwer Academic Publishers, 1986.


```
% main tail-recursive run clause
run(AC, PC) :-
      fetch(PC, P1, OP, X),
      execute(OP, X, AC, A, P1, P),
      run(A, P).
run(_, _).

% instruction fetch clause
fetch(PC, P1, OP, X) :-
      mem(PC, OP, X),
      P1 is PC + 1.

% instruction-specific execute clauses
execute(halt, _, _, _, _, _) :- !,
      fail.
execute(add, X, AC, A, PC, PC) :- !,
      mem(X, T),
      A is T + AC.
execute(stor,X, AC, AC,PC, PC) :-
      mem(X, _), !,
      retract( mem(X, _) ),
      assert( mem(X, AC) ).
execute(stor,X, AC, AC,PC, PC) :- !,
      assert( mem(X, AC) ).
execute(brn ,X, AC, AC,PC, X ) :-
      AC < 0, !.
execute(brn ,X, AC, AC,PC, PC ).
```

Figure 1: A Simple Microprocessor Specification

```
functionalUnit(regAC,reg,
      [bus3],[bus2],[regACFn,clock],[regACsign]).
functionalUnit(regPC,reg,
      [bus1],[bus1],[regPCFn,clock],[]).
functionalUnit(regOP,reg,
      [bus1],[],[regOPFn,clock],[]).
functionalUnit(regX,reg,
      [bus2],[bus1],[regXFn,clock],[]).
functionalUnit(adder1,adder,
      [bus1,bus2],[bus3],[adder1Cin],[]).
functionalUnit(memAR,reg,
      [bus1],[],[memARFn,clock],[]).
functionalUnit(memDR,reg,
      [bus2],[memDRdBus],[memDRFn,clock],[]).
functionalUnit(memDRdecoder,decoder,
      [memDRdBus],[bus1,bus2],[memDRDecode],[]).
```

Figure 2: The Symbolic Data Path

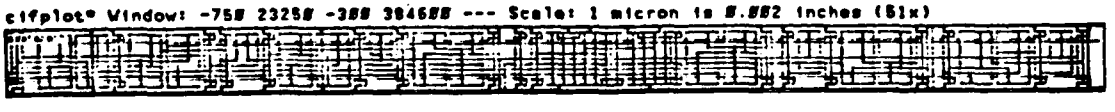

Figure 3: A Sticks-Form Data Path Bit Slice

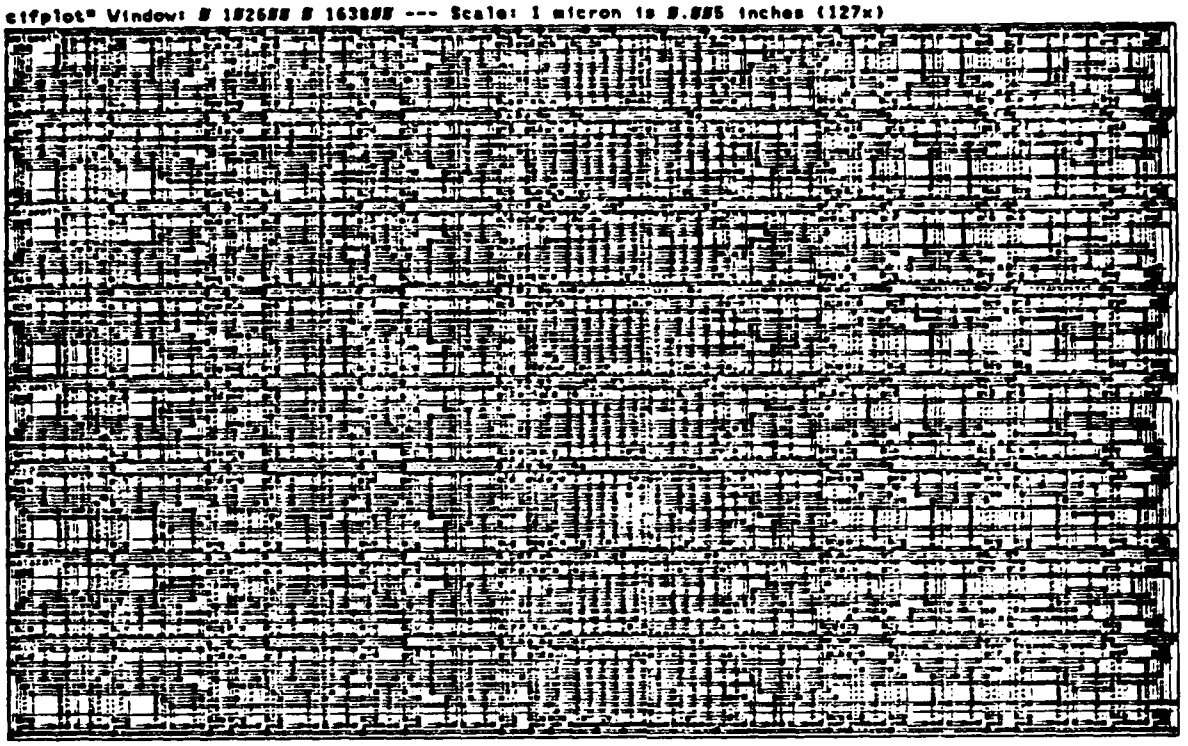

Figure 4: The Data Path Layout

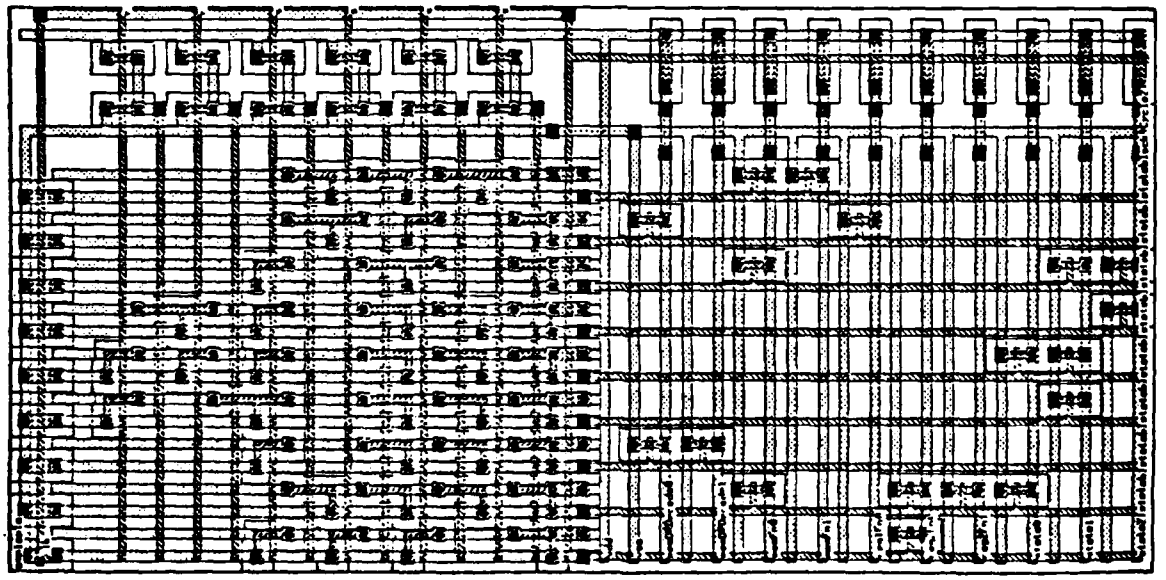

Figure 5: The Controlling PLA Layout

END

DATE

FILMED

9-88

DTIC