A Prototype Silicon Compiler in Prolog

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

The Advanced Silicon Compiler in Prolog (ASP) is a full-range hardware 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. This report describes the prototype ASP system and its major components.

ACKNOWLEDGEMENT

This research was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 6501, and monitored by the Office of Naval Research under Contract No. N00014-88-K-0579.

A Prototype Silicon Compiler in Prolog

The Advanced Silicon Compiler in Prolog (ASP) is a full-range hardware 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.

This report describes the prototype ASP system and its major components. The system is currently being completely reimplemented, based on our experience with the prototype, to make it faster and more general, and produce higher quality output. The report first gives an overview of the prototype system, then discusses in detail its three major components, and concludes with remarks about the new version of the system.

1. ASP Overview

The ASP effort is part of the Aquarius Project [Aquarius], which is aimed at producing high-performance Prolog engines, realized in part with specialized high-quality microprocessors. Thus the focus of ASP is microprocessor synthesis, with a design domain of single synchronous chips with a single data path and control path. ASP is also meant to test Prolog as an implementation language for design automation.

The general ASP approach is hierarchical and automatic. The input to the system is an abstract specification of an instruction set, and the output is a specification in CIF suitable for submission to a VLSI foundry.

An early design of the system ([CHS]) used a common unifying data structure; this approach was abandoned because we did not have the resources to both develop tools and a data base system.

ASP operates instead in a transformational manner, each level of the system transforming its input into sets of facts about the developing design. Each level brings the design closer to layout with more detailed facts, reflecting design decisions made at that level. Each level is autonomous, using the facts generated by previous stages.

Since ASP is implemented in Prolog, it is naturally a multi-paradigm system, using both algorithmic and rule-based techniques. In general the system is algorithmic, with rule-based local optimizations. It does not use goal-directed planning or have a single well-isolated rule set.

1.1. Decomposition of Silicon Compilation

Because a full behavior-to-silicon compiler is a complex undertaking, we decompose the silicon compilation problem into three major abstract problem domains, ordered hierarchically (see [CADDY] and [OCCAM], 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 Appendix 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 circuit components (for example, see Appendix 10). 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, module generation, transistor sizing, placement, and routing. The core of this level is module generation, which is done by the Topolog component. We also have a CMOS PLA generator and a channel router.

The third level is the geometric domain. The purpose of this domain is to present the programs of the functional domain with idealized geometric elements, in the form of a sticks-and-elements virtual-grid abstraction of actual mask layers (for an example, see Appendix 13). This domain encompasses the traditional tasks of compaction and device-level simulation. These tasks are accomplished by the Sticks-Pack component of ASP. See Appendices 14, 15, and 16 for example 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 compared to a single layer. Similarly, the data path constructor can only use functional units that the module generator can generate.

1.2. 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 while satisfying various constraints.

Viper 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 the Prolog specification 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 Appendix 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 by a Prolog interpreter. 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 '+', and are 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 [BUD], 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. Appendix 8 presents the data path derived from the specification in Appendix 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. Appendix 9 presents the finite state machine control path.

1.3. 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 an attempt to solve, in combination, 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 every row. Odd rows are flipped about the horizontal axis so that power and ground bars may be shared between rows. Topolog can be thought of as a standard cell layout program, but since blocks can be anything which has 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 Appendix 16, which is a bit slice derived from the data path description in Appendix 8. Our existing logic blocks are all designed by the Uehara-Van Cleemput procedure [UVC]. The UVC algorithm has been shown to derive near-minimal-width single-diffusion-strip static CMOS arrays.

Topolog 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 complementary device. The addition of a new circuit type is 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.

1.4. 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 Appendix 13), 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 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 containing 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. 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.

1.5. 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 our CMOS PLA generator (see Appendix 11), which then creates AND-OR sticks-and-elements PLAs from those boolean equations.

We have a left-edge-first channel router 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 investigated transistor sizing with a Prolog-based transistor sizer named Most [Most], which runs standalone.

1.6. 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 a new execution engine. Our

experience with Prolog in ASP has in general been positive.

1.6.1. 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 Sticks-Pack cell design environment, 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 simple 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.

1.6.2. The Use of Prolog for Specification

Prolog is used for specification because of its logical basis and declarative nature [Prolog]. 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 [Viper].

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

2. Viper

Viper is the high-level synthesis component of the Advanced Silicon in Prolog (ASP) system ([ASP]). This section summarizes the organization of Viper, and then presents the operation of individual Viper stages in detail (some of which appeared in [Viper]).

2.1. Organization

Viper performs the same basic tasks that other synthesis systems do. It translates specifications into an intermediate representation, schedules operations, allocates registers, creates functional units, binds operations to functional units, and creates interconnect. In order, the detailed tasks it performs are:

(1) realization of Prolog variables as architected registers,

(2) translation of Prolog goals into an intermediate representation containing register transfer operations and control information,

(3) dependency analysis,

(4) scheduling of operations,

(5) global analysis of data path resource needs,

(6) functional unit allocation and binding of critical operations to functional units,

(7) binding of the remaining operations and creation of interconnect,

(8) data path construction, and

(9) control path construction.

These tasks are grouped into four stages.

(1) Stage one consists of task 1. The model of storage in an input specification is changed from using write-once Prolog variables to global write-many registers.

(2) Stage two consists of tasks 2, 3, and 4. These are essentially bookkeeping activities that translate Prolog into a tractable intermediate form.

(3) Stage three consists of tasks 5, 6, and 7. This is the critical stage in which a data path of functional units is allocated (including, for example, ALUs) and operations in the specification (such as + and -) are mapped onto (bound to) functional units.

(4) Stage four consists of tasks 8 and 9. These again are bookkeeping tasks, which translate the internal design generated by Viper into a form usable by lower synthesis levels.

Viper performs two additional tasks that are needed to create proper input to the available lower level ASP tools, but that are not part of high-level synthesis. The control path definition is translated into PLA logic equations, and topological constraints are added to the data path definition.

2.2. Hardware Specification using Prolog

The microprocessor specification domain of ASP makes standard Prolog [Prolog] a reasonable choice as a specification language. Multiple asynchronous finite state machines, explicit parallelism, and detailed off-chip interface descriptions need not be supported. Instead, concurrency information can be derived by the system, and standard interfaces (design frames) can be supplied. The result has been to put considerable responsibility for the final quality of the design on the ASP system.

The specification domain is also constrained by ASP's pragmatic purpose (and reason for existence) as a synthesis system. Specifications must be effectively realizable in hardware.

2.3. Instruction Set Level Specification

ASP takes as its input, specifications that define the operation of microprocessor instructions. Individual instruction-specific clauses are contained in a recursive instruction-executing definition.

For example, consider a simple example, a fragment of a microprocessor specification.

```
SM1(AC, PC) :-
        fetch(PC, P1, OP, X),
        execute(OP, X, AC, A, P1, P),
        SM1(A, P).
SM1(_, _).
```

This is a definition of a Von Neumann machine, the SM1 (Simple Machine 1), which has two explicit registers, an accumulator (AC) and a program counter (PC). The machine is composed of a fetch cycle and an execute cycle, which are recursively evaluated until one fails.

The fetch cycle is defined as a clause that retrieves an instruction from memory and increments the PC.

```
fetch(PC, P1, OP, X) :-
        mem(PC, OP, X),
        P1 is PC + 1.
```

An add instruction is defined with an execute clause.

```
execute(add, X, AC, A, PC, PC) :- !,
        mem(X, T),
        A is T + AC.
```

A complete specification of this simple machine appears in Appendix 1. From this example a few observations can be made.

First, the specification is abstract. Bit widths and values, explicit concurrency, timing, and hardware entities (such as buses) are not present. Nonetheless, the basic specification is complete, without detail, in that it is an executable Prolog program, which provides a complete high-level simulation of the microprocessor.

Second, some details can be derived, such as concurrency from dependency analysis. Other details, such as bit widths, can be declared in auxiliary assertions, but default values are provided (32-bit data paths, for example).

Third, simulation at this level is also abstract. To execute the above specification in Prolog, abstract memory must be defined.[1] For example, the facts

---

[1]Memory could be defined in other ways. For example, each clause could have two additional variables, one bound to the state of memory when the clause is entered, and another bound to the state of memory on exit. Memory could be represented as a structure containing all valid addresses. This model of state is used in some theories of program semantics. It is logically clean but practically inefficient.

```
mem(1000, load, 2000).
mem(1001, add, 2001).
mem(1002, stor, 2002).
mem(1003, halt, _).
mem(2000, 2).
mem(2001, 3).
```

define a program and its data. Starting at location 1000, the SM1 adds two numbers, 2 and 3, and stores the result in location 2002. Actual binary images of programs must be simulated at a lower level.

Fourth, no particular level of abstraction is enforced. Memory and its referencing are, for example, quite abstractly defined, while the realization of the AC and PC variables (as registers) is obvious. Various stages of ASP synthesis will define, or require the definition of, many specific details.

Fifth, only a semantic subset of Prolog is supported. Backtracking must be avoided, since we do not want to implement non-deterministic finite state machines. We also do not implement truly recursive hardware.

2.4. Register-Based Transformation

The first stage of high-level synthesis in ASP introduces register-like storage into Prolog specifications. State in a basic Prolog specification is contained in Prolog variables, while state in a machine is held in registers that are global value holders. The first stage moves all state -- all value storage -- into global assertions. It performs a source to source transformation, producing a new specification, equivalent in functionality to the original one, in which register value assertions are used to store values instead of Prolog variables.

2.4.1. Register Conversion

In detail, values are stored in assertions of the form

<register-name>(<register-value>).

and are referenced by *set* and *access* goals.[2] Prolog variables carry values (and can be thought of as buses) but do not store them.

For example, the add clause above becomes

```
execute(add) :- !,
  access(regX, X),
  mem(X, T),
  access(regAC, AC),
  A is T + AC,
  set(regAC, A).
```

Prolog's tail-recursive single-assignment style, evident in the SM1 definition clause of the example microprocessor specification, is the main motive for introducing registers at this

---

[2]The access and set goals are defined as

```
access(X, Y) :- Z =.. [X, Y], Z.
```

and

```
set(X, Y) :- abolish(X, 1), Z =.. [X, Y], assert(Z).
```

point in synthesis. Since Prolog does not have destructive assignment, the variables in a Prolog program are equivalent to arcs in a data flow graph representation of it; the process of assigning registers to variables is essentially data flow optimization of storage. ASP need not initially translate the specification into a data flow graph, as, for example, the CMU-DA system does ([CMU-DA]). Because some analysis is needed to remove tail-recursive storage from specifications, and because specifications are already in data flow form, register allocation is done first. In addition, making registers visible through source to source transformation permits the user to analyze the transformation.

Conversion operates in two phases. The analysis phase associates registers with variables, optimizing by sharing. The transformation phase uses the analysis information to generate a new register-based specification.

2.4.2. The Variable Analyzer

The analyzer assigns registers to all Prolog variables in a specification. Different variables are made to share the same register under two basic circumstances, argument passing and value assignment.

Argument passing almost always causes sharing. In the original specification, values are passed between a goal and its matching clause head via argument variables. The analyzer preserves this result by assigning the same register to variables in the same positions in invocation and head. Thus in

... g(A, B), ...

and

g(X, Y) :- ...

A and X share one register, and B and Y share another. In the transformed specification, assigning a value to A's register makes the value available to X.

One case where argument passing may not cause sharing involves unification. Consider the general execute goal from above,

... execute(jump, X, ... P), ...

and the jump instruction clause (which sets the PC),

execute(jump, ADR, ... ADR).

The X and P variables should not be assigned the same registers.

A special case of argument passing is tail recursion; different variables in the *same* clause are assigned the same registers. The clause head variables (representing the values of the current loop iteration) share the storage of the variables in the recursive invocation (representing the values of the next iteration).[3]

Value assignment often causes sharing. In particular, the destination variable and one source variable of an *is* operator can be assigned the same register when the old source value is not used after the new destination value is computed. Sometimes the analyzer has a choice of source variables. Consistency with tail-recursive argument sharing usually drives the

[3]This sharing is correct only if the next iteration values are defined after all uses of the current-iteration values.

choice.

The analyzer takes a goal as its input argument, and analyzes the (depth-first) transitive closure of clauses reachable from that initial goal. It generates a database of relations containing variable and register information.

2.4.3. The Register Transformer

The transformer produces new Prolog clauses, adding access and set goals, and removing variables from clause heads and associated goal invocations.

Not all variable arguments can be removed. For example, constants appear in clause heads for clause selection, and the variables in corresponding goals must be retained. Consider the execute clauses in the example specification; the halt, add, and load symbols must be retained, with the corresponding goal in the SM1 clause becoming execute(OP). These control flow variables will later be mapped into next state selection logic in the control path.

Variables must also be retained when they return values from facts. For example, the instruction memory location

mem(1000, load, 2000).

when referenced by

... mem(PC, OP, ADR), ...

with PC bound to 1000, retrieves the load operator and the operand 2000. The memory reference is transformed into

... access(regPC, PC),
mem(PC, OP, ADR),
set(regOP, OP),
set(regADR, ADR), ...

An appendix shows part of the analysis data base for the microprocessor example. It contains the facts that the analyzer generates for the fetch clause. The nameBindings associate variable names with variable positions in clause heads; the indexBindings relate indices to storage information; and the storage bindings bind classes of storage to registers. Note that PC and P1 are assigned to the same register.

2.4.4. Register-Based Constructs

The memory example above illustrates a problem with introducing registers into a specification, that of mixed levels of detail. At this point, after variables have been converted, all registers should be defined. The memory system, however, is still abstract. Memory address and data registers, in particular, are needed.

Returning to the add clause at the beginning of this section, with memory registers it should become

```
execute(add) :- !,
  access(regX, X),
  set(memAR, X),
  mem_read,
  access(memDR, T),
  access(regAC, AC),
  A is T + AC,
  set(regAC, A).
```

This makes the memory registers explicit. The complete memory-based microprocessor is shown in Appendix 2.

Knowledge of the complete memory subsystem is currently built into Viper. After register analysis, and as part of transformation, abstract memory references are converted into register-based ones. Addition of such microarchitectural features as the memory subsystem could be done in a separate later stage, but is instead part of register transformation because the information and analysis needed are readily available. Subsystem addition should be more parameterized than it is in Viper, and to achieve this a separate stage may be necessary, in which case the abstract version above would serve as an intermediate form.[4]

In general the system must support the specification of implementations of hardware subsystems. This is equivalent to allowing the user to define microarchitectural detail.

2.5. Prolog to Register Transfer Translation

The second stage of synthesis converts register-based Prolog into a form suitable for data and control path construction. It translates Prolog goals into register transfers, which are then used for dependency analysis and scheduling.

Each transfer collects, from different goals, information related to a single hardware time step. In particular, each transfer, represented as a four-element[5] structure, contains value sources (registers or constants), an operation on those values, and a destination register for the result value, and has the form

transfer(<source1>, <source2>, <operation>, <destination>)

A transfer is constructed out of source, operation, and destination goals. The transfers are abstract because the operations they contain are Prolog operators (such as +) not yet bound to any hardware implementation.

For example, the register-based add goals

---

[4]The abstract interface is signal-based -- values are passed by bus-like Prolog variables. The concrete interface is register-based -- values are passed in registers.

[5]This is a simplification. Each transfer also has a unique name and identifies the FSM state to which it belongs. See below.

```
access(regX, X),
set(memAR, X),
mem_read,
access(memDR, T),
access(regAC, AC),
A is T + AC,
set(regAC, A).
```

are converted into the sequence

```
transfer(regX, none, none, memAR)
transfer(none, none, mem_read, none)
transfer(memDR, regAC, +, regAC)
```

A transfer is constructed out of source, operation, and destination goals; as individual goals are processed information about them is recorded.

Abstract transfers fit between register-based Prolog and synthesized hardware. Since registers have been allocated by this stage, and ASP does not currently synthesize pipeline computations, atomic register transfers are appropriate units for analysis and hardware generation. Dependencies between transfers constrain scheduling, and resources must be allocated on the basis of transfers.

This stage also generates a control flow graph, which divides abstract transfers into a collection of basic block linear sequences; each basic block is realized as a state of a finite state machine. Clause selection is the fundamental conditional construct in Prolog, and maps straightforwardly into finite state machine transitions when the control path is constructed.

The relations produced by this stage are a complete representation of the specification. They could serve as input to a simulator that evaluated the control flow graph and associated transfers.

2.5.1. Transfer Analysis

This stage scans Prolog specifications, converting each goal into part of an abstract transfer operation. Each transfer operation is associated with a basic block of transfers.

Each transfer is stored in a relation and has the form

```
transfer(<identifier>, <block>,
        <source1>, <source2>, <operation>, <destination>).
```

The identifier is generated by the system and uniquely identifies the transfer.

Prolog goals divide into three classes: sources, operations, and destinations. When a source or operation goal is processed, information about it is recorded. When a destination goal is encountered, the relevant source and operation information is retrieved and the complete transfer constructed. All source goals are access goals. Destination goals are set goals and certain computation goals, such as comparisons, that affect control. Prolog variables are used to connect the pieces of goal information. For example, the add goals

access(memDR, T),
access(regAC, AC),
A is T + AC,
set(regAC, A).

are represented by the fragments

srcVar(memDR, T).
srcVar(regAC, AC).
expVars(T, AC, +, A).
dstVar(regAC, A).

By following the chain of Prolog variables back from the dstVar entry, the operation and source registers can be found and assembled into a single transfer.

The data base of fragments for the example processor can be found in Appendix 3. The complete set of transfers is in Appendix 4.

2.5.2. Control Flow Analysis

As the analyzer processes goals it also accumulates state transition information. It only records transitions that alter normal linear control flow. These transitions can be conditional or unconditional.

Consider the simple processor example. It consists of a case dispatch to a collection of instruction-specific goals. The dispatch is a conditional transition; the return from a case arm is an unconditional one.

Unconditional branches are stored as

branch(<from-block>, uncond, <to-block>).

Conditional branches have the form

branch(<from-block>, cond, <test>).

where *<test>* is the source of the value that will drive the dispatch. Each arm is stored as

case(<from-block>, <value>, <to-block>).

For example, the execute dispatch example is represented as

branch(block1, cond, regOP).

and

case(block1, add, block3).
case(block1, load, block4).
...

and the end of the case arms appear as

branch(block3, uncond, block1).
branch(block4, uncond, block1).
...

From these relations a controlling finite state machine can be constructed.

The relations produced by this stage are a complete representation of the specification. They could serve as input to a simulator that evaluated *transfer*, *branch* and *case* entries.

The branch relations for the example processor are in Appendix 4, along with the transfers.

2.6. Transfer Scheduling

After the system generates abstract transfers it schedules them. It assigns time steps to transfers in an as-soon-as-possible manner, with concurrency limited only by inter-transfer dependencies. The data path construction stage has the capability to modify this schedule, based on resource constraints discovered in that stage.

Dependency is defined to be the conflicting use of any register or restricted resource (such as memory). Most inter-transfer dependencies are explicit, involving register uses, and are much like dependencies between variables in software. Dependencies can be implicit, however, because some actions cause side effects. For example, a memory read loads the memory data register; use of memory data requires waiting for the read to complete. The system allows for the definition of implicit dependencies between operations and registers.

The concurrent schedule is easily generated. Transfers are scanned in the order in which they were created -- in the serial order of the original Prolog goals. A transfer is assigned to the time step immediately following that of the latest transfer upon which it depends. Part of the memory subsystem definition includes assertions specifying its implicit dependencies, such as

```
implicitDependent(mem, memAR).
implicitDependent(memDR, mem).
```

To aid the designer, and guide later rescheduling, the stage also creates a dependency data base. It records dependencies between pairs of transfers and the resources involved.

Appendix 5 contains the dependency data base for the example processor; Appendix 6 contains its schedule. Note that the cycle numbers assigned to transfers are relative within a block -- the first cycle of any block is cycle 1.

2.7. Data Path Generation

The third synthesis stage defines data paths based on the requirements of abstract transfers and their associated schedule. It generates both static information (symbolic functional units and bus connectivity) and dynamic information (functional unit use and bus use).

For example, the add transfer and schedule fragment

```
transfer(op8, block3, memDR, regAC, +, regAC).
cycle(op8, block3, 3).
```

produce the data path elements

```
elementType(memDR, register).
elementType(regAC, register).
elementType(dpalu, alu).
elementFn(dpalu, add).
```

and the dynamic binding

elementUse(dpalu, add, op8, block3, 3).

In addition, the buses

busSrc(bus1, memDR).
busDst(bus1, dpaluPort1).
busSrc(bus2, regAC).
busDst(bus2, dpaluPort2).
busSrc(bus3, dpalu).
busDst(bus3, regAC).

are created, as well as the bus bindings

busUse(bus1, memDR, dpaluPort1, op8, block3, 3).
busUse(bus2, regAC, dpaluPort2, op8, block3, 3).
busUse(bus3, dpalu, regAC, op8, block3, 3).

The stage allocates functional units based on the requirements of each time step, creating enough units to execute all operations assigned to that step. It also creates enough buses and connections.

The complete data path data base for the example processor is found in Appendix 7.

2.7.1. Functional Unit Allocation

Information generated by the system about functional units can be divided into two categories, static and dynamic. Static information defines data path structure. Dynamic information is time step dependent and binds the operations of abstract transfers to data path elements.

An operation in a transfer is a Prolog operator (such as +). A functional unit has a type (ALU, for example) and a set of functions it performs (such as add and subtract). Every Prolog operator the system can process has at least one associated functional unit type and function.

To allocate functional units, the system first scans all transfers, noting all the operations that the designs will have to support. It notes operations that can be treated as special cases (such as adding 1 to a register), and operations that are performed in parallel. It then uses heuristics to select an efficient set of functional units. It next it binds individual operations in transfers to specific functional units, and then creates and schedules buses.

2.7.2. Connectivity

Buses are created and scheduled in a manner similar to functional units. The system produces both static structural information and dynamic binding information. It uses existing bus resources when possible. It considers buses to be bidirectional, but connections (multiplexers and decoders) to be unidirectional.

Given a collection of functional units and a schedule, the system attempts to generate only the connectivity necessary to implement that schedule. It examines in turn each time step's transfers. For each transfer, if its associated registers and functional unit are connected by buses unused in that time step, those buses are used. Alternatively, if unused buses exist but are not connected to the relevant functional units, the necessary connections are created. Finally, if unused buses are needed they are created and connected.

To keep the prototype system simple, it does not modify the functional unit schedule during bus creation. Also, the number of buses is not constrained, nor is bus regularity (connecting all registers to the same buses, for example) a factor considered by the system.

2.8. A Structural Description Mechanism

After data path generation, the data path and control path are completely defined. The information exists, however, in several incrementally generated relations. The final act of high-level synthesis translates that information into a structural hardware description[6] that the lower levels of the ASP system can use. This translation collects various elements from various relations and packages them into a sequence of data path element declarations and a finite state machine definition, in both of which all interconnections are explicit and named. Prolog structures and lists are used to package this information.

2.8.1. The Data Path

Instances of element types are created and given names. In addition (unlike variable definition), the connectivity between elements must be established.

A structural data path element has a type, a name, and four lists of connections -- inputs from other data path elements, outputs to data path elements, inputs from the control path, and outputs to the control path.

In detail, each element declaration has the form

functionalUnit(<type>, <name>,
[<list-of-data-input-signals>],
[<list-of-data-output-signals>],
[<list-of-control-input-signals>],
[<list-of-control-output-signals>]).

The lists of signals indicate connections to be made with other parts of the design. For example,

functionalUnit(alu, dpalu,
[bus1, bus2], [bus3],
[dpaluFn dpaluCin], [dpaluSign, dpaluCout]).

creates an ALU and binds it to *dpalu*.

For every control input signal mentioned in an element statement, a declaration of the form

controlIn(<signal>, <default-input>, [<list-of-inputs>]).

is required. Control input signals are connected to and driven by the control path. This declaration defines the signal's default value and other possible values it can have. The number of values defines the bit width of the signal. For example,

controlIn(dpaluFn, pass, [add, ...]).

would appear with the ALU element definition above.

---

[6] A structural description explicitly represents connections between hardware elements. The OCCAM to CMOS project ([OCCAM]) uses DDL as an intermediate form, similar in function to our structural description mechanism; DDL is not, however, strictly structural.

In a similar manner,

controlOut(<signal>, [<list-of-outputs>]).

defines the outputs of a control output signal. Such signals serve as input to the control path.

The complete data path definition for the example processor is in Appendix 8.

2.8.2. The Control Path

Control information is specified in finite state machine style. Associated with each state are the control lines to be driven and conditional next state transitions.[7]

For example, a state definition using the ALU for addition could appear as

state(state1,
[output(dpALUfn, add), ...],
state2).

The state contains additional outputs for loading and storing registers and gating values to and from buses.

In particular, each state has the form

state(<name>, [<list-of-outputs>], <next-state>).

The *<name>* is the name of the state. The list of outputs consists of pairs of the form

output(<value>, <signal>)

where the *<value>* is the value to be output, and *<signal>* designates the signal to be driven. Both *<value>* and *<signal>* must be defined in a controlIn statement. The *<next-state>* can either be a state name or a conditional branch of the form

branch(<test-signal>, [<list-of-cases>])

where *<test-signal>* is an output control signal. Each element in the *<list-of-cases>* has the form

case(<value>, <state>)

where *<state>* is the next state if *<test-signal>* is equal to *<value>*. Both the signal and all its values must be defined in a controlOut statement.

The complete control path definition for the example processor is presented in Appendix 9.

All the state, control, and element statements are passed to the lower level parts of ASP for synthesis.

2.8.3. The Library of Functional Units

As the synthesizer allocates, binds, and outputs a data path, its basic building block is the functional unit. It is a fundamental link between high-level synthesis and lower synthesis levels. Its characteristics are important to behavioral synthesis; its contents are important to

[7]Multi-phase clocks are not supported. They could be, either by dividing a state into phases for control line purposes or by defining multiple phase-conditioned states.

logic and geometrical synthesis. In ASP those characteristics and contents are collected in a library of functional units.

The characteristics of a functional unit are its type and the functions it implements. Its contents are logic equations used by the ASP module generator. From the behavioral point of view the purpose of a functional unit's characteristics is to guide functional unit selection.

The library also contains implementation details about functional units, in particular the control signal bit patterns used to stimulate specific functions; this information is used by the PLA equation generator. Not all information about functional units is contained in the library; the heuristics that allocate functional units contain knowledge about some functional unit types, and knowledge about topology is contained in the topology constraint layer discussed below.

The library of functional units can be found in Appendix 10a. The corresponding logic equations used by the lower level module generator can be found in Appendix 10b. (This is not the complete library, but only that part needed for the example processor.)

2.8.4. Lower Level Interfaces

Two lower level interfaces are not strictly part of the Viper system, but they are necessary to interface with the rest of ASP, and interact with the library of functional units.

One interface generates and/or logic equations for the ASP PLA generator from control path state statements. Enable signals for individual control path outputs are accumulated by scanning all the state statements, and converted into logic equations for specific control bits. Common and and or terms are eliminated from the equations. The equations for the example processor are shown in Appendix 11.

The other interface generates topological constraints for the data path module generator, indicating how control and data lines should be placed. Signal lines are also decomposed into individual bit lines that can be connected to the PLA. The topologically constrained data path for the example processor is found in Appendix 12.

3. Topolog

Topolog is the module generator and layout engine for ASP. It takes as input a circuit description and constraints, and outputs sticks-based layout.

3.1. General Approach

A module generator is a program which, given a description of a circuit as a collection of blocks, or subcells, returns a constructed cell. The subcells may be modules in their own right, or elementary pieces of silicon called leaf cells. A layout engine is a program which, when given a description of a circuit either as a collection of gates or as a list of transistors and connections, returns a piece of silicon which implements the circuit.

Topolog combines the functions of a module generator and layout engine in an attempt to solve, in combination, problems specific to each. Typical module generation systems [Allende] manipulate pieces of geometry rather than circuit elements, which means that most module generation programs and parameters simply direct the manipulation of pieces of wire rather than function. Further, if a module consists of submodules, the choice of which submodule to instantiate first has a large effect on the resultant circuit for purely geometric reasons. Folding a layout program into a module generator permits the generator to concentrate on the functional design of circuits, rather than on their geometry, which in practice yields much more concise module descriptions. Further, if the submodules are expanded as blocks

and jointly placed and routed, the second problem disappears.

Topolog takes as input logic equations and port locations, and produces a virtual-grid static CMOS layout in the gate matrix style popularized by Lopez and Law [G-Matrix]. Topolog, like the Berkeley tools Topogate and GEM, produces layouts featuring a single pair of diffusion rows between power lines. This practice we found to halve the spacing between polysilicon columns, at a small penalty in vertical dimension. This penalty is bounded above by 27% in the MOSIS scalable CMOS rules, and approaches 0 in most practical cases as most penalty area may be used for horizontal buses.

Unlike Topologizer or GEM, which consider transistors individually in placement, Topolog uses the Uehara-van Cleemput algorithm [UVC] to lay out blocks. Blocks are then placed using a min-cut algorithm and routed using a left-edge-first algorithm.

3.2. Description of the Program

Typical layout engines are flat ([SWAMI], [GENIE]), that is, a single long list of transistors is used to describe the function to be generated. This both is tedious from the point of view of users (who must enter their circuits as long sequences of logic equations, rather than using circuit hierarchy) and robs the layout engine of inherent partitioning of most logic circuits. This is onerous since most automated placement tools either implicitly or explicitly partition a circuit into connected subcircuits. The class of placement tools which do such partitioning is broad indeed, including clustering, min-cut, force-directed and clique-based placement tools. Even simulated annealing, which specifically does not work by circuit partitioning, derives its name and its original motivation from the formation of metal into disjoint clusters.

Topolog is designed around the basic abstraction of a block. A block represents a primitive circuit element, and it is defined by the fields it contains and the routines which generate it. A block has a p-side and an n-side, both of which have a maximum height and minimum height, a set of elements, a set of sticks, and a set of pins. In addition, the blocks have a set of net names, a maximum width and minimum width, and various fields used only by Topolog itself. 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-side and the 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. Although Topolog can be used as a standard cell layout program, since a block can be anything which has the characteristics mentioned above, it is more accurate to describe Topolog as a Gate Matrix [G-Matrix] style layout engine.

Topolog has a six stage pipeline.

(1) Inputs are parsed and a preliminary generation of all blocks is done. In this pass, the maximum height, minimum height, maximum width, and minimum width of the blocks are fixed.

(2) The blocks are then grouped into rows.

(3) The blocks are placed within rows. During this placement phase, macroblocks (modules) are expanded into their primitive components.

(4) 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.

(5) The channel is then routed.

(6) Finally, the complete circuit is converted to a sticks form and the package is output.

An abbreviated Topolog pipeline is available when only one row needs to be placed. This abbreviated pipeline omits placement into rows and vertical channel routing.

3.3. Description of the Algorithms

Topolog is a package consisting of ten modules and about 3000 lines of Prolog code. Of the ten modules, six implement algorithms used in the package, one is a rule-based module to connect the outputs of logic functions formed in the wells to buses in the channel between the wells, one generates the sticks description from the internal data structures, one forms the declarations and generic routines for the data structures used, and one is used to simulate the extensions to the Prolog language that we found were required to implement the algorithms we wished to use.

Topolog first reads in and parses a set of facts in Prolog's database which describe the blocks to be laid out. The parsed blocks are then passed to the Uehara-van Cleemput package, which determines transistor order and separation zones within the blocks. The blocks are then passed to the placement routine, which separates them into rows using a min-cut algorithm modified to consider block size when determining the cut. Once placed, the logic specifications with transistor placement are translated into a pair of diffusion strips for each block. Metal routing is then done over the strips using a left-edge-first channel router. A simple router is all that is required, since pins are on only one edge of the channel.

This routing must be dense, since it is a prime determinant of the vertical pitch of the block. Further, vias must be minimized, since they contribute heavily to parasitic capacitance in the wells. Finally, diffusion must be used as little as possible for routing, since it is highly capacitive.

The channel router therefore uses metal-1 for horizontal routing, and vertical routing where the proposed vertical run does not cross a horizontal metal line. Metal-2 is used for vertical routing but not horizontal routing, since it requires a double contact to go down to diffusion. Diffusion is used for other vertical runs.

Once the wells are routed, a rule-based program is invoked to route the output of the gate from the p-well and the n-well into the channel. This program first attempts to ensure that no track must be added to either well to route the output of the gate into metal-2, as required. Its second function is to ensure that the same column is used by both the p-side and the n-side to route the output to the channel.[8]

The horizontal channels are then routed, again using the simple left-edge first router. The assignment of numbers to rows is then made, and the entire package is output.

3.4. Input Format

The Topolog input format is a collection of logic equations, each having one of the following forms:

---

[8]Once the outputs are routed, the full internal coverage of metal-2 in each row of blocks is known. Channels are defined for routing between channels. A modified left-edge-first router is used to run lines between the rows, attempting to minimize channel density in the horizontal channels.

Output = pass(Input, Control)
Output = transmit(Input, Control)
Output = compl(Expr)

where Expr is an and-or tree in an arbitrary number of variables, whose value is the complement of Output.

Optionally, one may add a sequence of statements of the form:

{left, right, top, bottom}Edge(X)

which indicates that signal X has a port at the left, right, top, or bottom edge, respectively.

An example for a one-bit adder is given below.

x = compl(or(and(c,or(a,b)),and(a,b))).
y = compl(or(and(x,or(a,b,c)),and(a,b,c))).
sum = compl(y).
carry = compl(x).
leftEdge(a).
leftEdge(b).
leftEdge(c).
rightEdge(sum).
rightEdge(carry).

3.5. Output Format

Topolog generates a description of the circuit in virtual-grid symbolic coordinates, as a database of Prolog facts. These facts are then read by the compactor and converted into Caltech Intermediate Form.

The database consists of several kinds of clauses. A wire is described by

wire(Material, FromPt, ToPt, Width, Signal).

with the fields having the obvious meanings. A transistor is described by

trans(Type, PtSrc, PtGate, PtDrain,
Width, Length, SrcSig, GateSig, DrainSig).

where PtSrc, PtGate, and PtDrain are the positions of the source, gate, and drain of the transistor, and SrcSig, GateSig, and DrainSig are their source, gate, and drain signals, respectively. A contact is described by

cont(Type, Center, Offset, Signal)

where Offset (e, n, w, s) defines an offset of the transistor from the center point.

Finally, maxrow and maxcol describe the positions of the maximum row and column in the layout. An example of the output format is given below.

```
wire(p,pt(2,2),pt(9,2),_,a).
wire(p,pt(2,4),pt(9,4),_,b).
wire(p,pt(2,6),pt(9,6),_,c).
wire(p,pt(2,8),pt(9,8),_,b).
trans(pd,pt(2,30),pt(2,31),pt(2,32),1,1,vdd,x,carry).
trans(nd,pt(9,30),pt(9,31),pt(9,32),1,1,gnd,x,carry).
cont(m1m2,pt(5,27),no,sum).
cont(m1m2,pt(7,9),no,x).
cont(m1m2,pt(7,11),no,x).
node(10,34,gnd).
maxrow(10).
maxcol(34).
```

Further discussion of these formats can be found in the next section.

3.6. Extensibility: Technology Independence and Block Generation

Our existing logic blocks are designed by the Uehara-Van Cleemput [UVC] procedure, because the UVC algorithm has been shown to derive near-minimal-width single-diffusion-strip static CMOS arrays. It minimizes vertical dimension as well, given a single diffusion strip, and it is unlikely that any multiple-strip layout style can approach the UVC single-strip style in area minimization for either static or dynamic CMOS.

We are not restricted to pure UVC blocks, however. It is easy to customize Topolog to produce and place other blocks -- indeed, we use such customization to produce and place pass and transmission gates along with static CMOS AOI gates. We did not originally intend that Topolog be this versatile; it has This versatility is a result of using Prolog as our implementation language and a consequence of the modularity of the Topolog pipeline.

The only algorithms within Topolog that are specific to static CMOS AOI blocks are the Uehara-Van Cleemput procedure, and the procedures to wire up the rows, route the wells, and route block outputs. The other algorithms deal with blocks as abstract objects, and a block is merely an object that contains certain features.

The addition of a new circuit type is easy, due to Prolog's clause-based programming style. It is possible in Prolog to write polymorphic procedures -- that is, procedures which take one of several types of inputs as clauses. Hence it is possible to write clauses as specializations of general procedures to perform operations on special purpose data structures. If these clauses simply fail because their inputs diverge from those for which the clause was designed then such clauses have no effect on the rest of the procedure.

In order to customize Topolog to produce a specific type of block, users must write a clause for the procedure *parseInputs*, which produces a data structure describing their block; such a block must contain fields *blockSize* (the horizontal pitch of the block, in some standard size -- the only standard is that used for AOI gates, which is integer multiples of the horizontal pitch of two polysilicon columns). The user may also write a procedure for *minimalInterlaceBlock*, the main routine of the Uehara-Van Cleemput algorithm; this is unnecessary, as a catch-all do-nothing clause is defined which will simply pass the block through the algorithm. The user must then write a clause for procedure *extractBlock*, which takes the user's original block as an argument and defines an extracted block, which contains a list of the rows used in the two wells, the columns used in the block, the sticks and circuit elements defined, and a pair of nodes for output routing. Such extracted blocks are presumed to define wires in diffusion or metal layers only in the well regions, are presumed to have defined distinguished

wires for Vdd (at the top of the block) and GND (at the bottom of the block), and are presumed to have obeyed the constraints given by the *horizontalWire* and *verticalWire* procedures; unless modified, these are *horizontalWire(metal1)*, *verticalWire(metal2)*; the static CMOS *extractedBlock* procedure assumes this restriction.

3.7. Extensibility: Module Generation

It is convenient for users to define modules as collections of blocks or other modules. As a result, *buildBlock* has a catch-all clause; if it cannot build a block any other way, it calls a procedure defined by its first argument. Specifically:

```
buildBlock(X, Block) :-
  X =.. [BlockType|BlockArgs],
  concat(BlockArgs, [Block], FunctionArgs),
  Call =.. [BlockType|FunctionArgs],
  Call.
```

Hence a request in Topolog's input file of the form:

alu(x, y, z).

would result in a call to the Prolog procedure:

alu(x, y, z, Block).

The user must write a clause for the procedure *buildBlock(Input, Block)*, where *Input* is the input for the block; for example, the clause header for AOI blocks is *buildBlock(Output = aoi(Expr),Block)*. This clause must return a *Block*, which is a data structure with the fields mentioned above. Some of these fields (in particular, the *max_height and min_height* fields of the two sides and the *max_width* and *min_width* fields) must be filled in, since these are used by the placement code. In addition, the user probably wishes to store a parse form of *Expr* for later use. We have designed a a variety of library routines to assist in the construction of this clause.

buildBlock calls must be used to build the various component blocks (including other modules, which would be invoked by the same mechanism). A final call

buildCompositeBlock([Block1,...,Blockn], Block)

must appear as the last call in the *alu* procedure. Here, *Block1,...,Blockn* are the blocks built by the call to *buildBlocks* in the *alu* procedure.

Of course, the alu procedure must be known to Topolog at the time of invocation; the request:

use(file).

loads the procedures defined in *file*.

buildBlock only does the first pass at generation of a block. In the second pass, the block must become an object with a full set of elements and sticks. The procedure *generate_block(Block, PRows, NRows, Columns)* is called to instantiate a block on the rows and columns given; these columns are guaranteed to be in the range given by height and width. Again, a large set of modules is available to aid in the construction of this routine.

No other clauses are required for module construction, since the placement routines break modules into their component parts before the blocks are actually generated; hence *generateBlock* clauses need only be supplied for primitive blocks.

3.8. Performance

The one-bit adder example given above was generated by Topolog in 72.15 CPU seconds on a Sun 3/75. The output from the compactor is shown here.

Procedure	Time (sec)	% Execution
input	1.0	1.4
uvc	1.3	1.8
placement	10.5	14.8
making rows	5.9	8.3
extracting blocks	17.0	23.9
channel routing	12.7	17.8
output	23.2	32.6
Total	71.1	100.6

Thus far, the largest example that we have run on Topolog is a pair of bit slices of a simple microprocessor, the SM-1. The total time to generate the bit slices is broken down as follows[9]:

Task	Time (sec)
Input	0.4
Build Blocks	3.6
Place Blocks	486.7
Generate Blocks	50.8
Channel Definition	24.4
Channel Routing(bit 0)	56.8
Output (bit 0)	83.6
Channel Routing (bit 1)	57.2
Output (bit 1)	83.0
Total	846.5

The first four stages of the pipeline are held in common between bit 0 and bit 1; channel routing and output is separate. The details of this economy are due to a little trick involving Prolog's backtracking semantics.

These performance figures are by no means optimal; we expect that an improvement by a factor of three is possible without any change to the underlying substrate of our Prolog interpreter, of type access and definition code. The critical path here is clearly our placement algorithm.

3.9. Extensions to Prolog Useful for Topolog

In implementing Topolog we found certain aspects of Prolog to be restrictive.

[9]These figures were obtained on a Vax 11/785 running 4.3 BSD Unix and C-Prolog version 1.5.

3.9.1. Structural Replacement

The major problem we encountered was the assign-once nature of Prolog. The Kernighan-Lin min-cut algorithm works by exchanging blocks across a partition; in order for the algorithm to function, then, each block must contain a component which indicates which side of the partition a block is currently on. Further, in order for the cost of an exchange to be computed quickly and accurately, each net must contain a list of the blocks it is incident upon and each block must contain the list of nets incident upon it. When a block is moved across the partition, the component indicating which side it is on must be changed. This requires generating a new block. This block is contained in some set of nets, each of which must be regenerated. These nets in turn are contained in some set of blocks, each of which must be regenerated. Potentially, this may continue until each block and each net has been regenerated, all to adjust one field in one block.

The solution we adopted simulates multiple assignment in an assign-once language. In each component of a data structure, instead of storing the actual value we store a value structure, the first field of which is the value of the component, and the second an unbound variable. The value of the component is set by the following code:

```
setVal(U, X) :-
  var(U), !,
  U = valStruct(X, _).
setVal(valStruct(_, U), X) :-
  setVal(U, X).
```

and the value is accessed by the following code:

```
accessVal(valStruct(U, X), U) :-
  var(X),
  !.
accessVal(valStruct(_, X), Y) :-
  accessVal(X, Y).
```

Broadly, *setVal* chases recursively through the *valStructs* until it reaches an unbound variable, which it sets to the *valStruct* of the new value and an unbound variable; *accessVal* chases through the *valStructs* until it finds one with an unbound variable as the second argument; it then returns the first argument of the *valStruct*.

The effect of this storage method is the provision of multiple assignment in a single-assignment language, and it permits the efficient implementation of standard CAD algorithms in Prolog. There are two principal costs of this method. First, assignment or access to a structure component becomes an $O(n)$ rather than an $O(1)$ operation, where n is the number of assignments to the component. In practice, this is not too onerous a cost; measurements on Topolog have shown that the median depth of a *valStruct* is 1, and the mean slightly over 1; the maximum in our programs has been 5.

The second disadvantage is that unification cannot be used to build or access structures that contain *valStructs*, since unification will not return the value of a component but rather a *valStruct*. Since we prefer the use of the *field* macro described above, it was easy to write *accessField* and *setField*, a straightforward combination of the *field* macro with the two procedures described above.

We would prefer a weak form of destructive assignment, which we call structural replacement, over value structures. In particular, as we have shown [rplacarg], in a

networked data structure (a data structure in which some node is shared by two or more other nodes), modification of the data structure without structural replacement can cost up to *O(log n)*, where *n* is the number of nodes in the data structure, no matter how the data structure is stored. Since we can generate one node in a single data structure for each step of any algorithm, the performance penalty is bounded below by *O(log n)* for any algorithm implemented without structural replacement.

The form of the structural replacement operator that we prefer is simple. We would like an operator that would replace transparently only arguments of structures (since the lack of a destructive assignment operator for atomic variables is not only benign, but, given the logical variable, necessary for any reasonable semantics of a Prolog program), and whose work would be undone on backtrack, since we feel that any operation not undone on backtrack is destructive of Prolog semantics. The SICStus Prolog *setarg* operator meets these requirements [SICStus].

3.9.2. Arrays

Multidimensional arrays are required for some of the algorithms used within Topolog, and hence we sought a method of array implementation. Once the value structure and data structure code above were in place, implementation of array code became relatively straightforward. An array is merely a structure of size equal to the number of elements of the array, and a small associated data structure which maps a given index vector to an array element. The difficulties in implementing arrays in Prolog have traditionally been a desire to avoid recopying the entire array when any element is changed; this is precisely the purpose of *setVal* and *accessVal*, and hence this difficulty is solved for us.

3.9.3. Circular Data Structures

Topolog manipulates both circuit elements (blocks) and their connections (nets). Each net contains a list of all blocks incident upon the net, and each block contains a list of all nets incident upon it, giving rise to a circular data structure.

In C-Prolog, however, every attempt to create this structure resulted in an infinite loop in the unification routine; eventually, we gave up, and stored only the net names in the blocks, and looked up the actual nets in a balanced tree sorted by net name -- a cost of *O(log n)* for each (logical) pointer traversal.

3.9.4. Data Types

Unification is used in Prolog to create and access data structures. When programs are small, or the data structures that they create or access are small, or each data structure is used only within a single module, this is straightforward. We found, however, that the most convenient way to program Topolog was to create a single data structure, the block, with a large number of fields; each module selectively filled in fields of the block. This organization meant that whenever a field was added to the block definition (a common occurrence in program development), the field had to be added in every clause where the block structure appeared, an onerous task, and one that led to the introduction of many bugs.

The solution we adopted was to add a *typedef* procedure, called when a file is loaded. *typedef* takes a structure as its argument, and defines a clause in the procedure makeStruct, which builds an instance of the data structure and clauses in the procedure *field*, which in turn, when given an instance of a data structure and the name of a field within the structure, returns the value of that field. Once *typedef* was implemented, data structures proved easy to modify, and a major difficulty in programming was removed. *field* proved to be the

procedure most called in Topolog; almost 600 lines of code directly reference it. Sixteen major data types are defined in Topolog, with the number of fields varying from 2 to 19. These data types are often widely shared among various procedures.

4. Sticks-Pack

In this section we present Sticks-Pack (SP), a design environment for VLSI circuit layout generation written exclusively in Prolog. Not only does Prolog provide a relational database for VLSI objects, it also provides a syntax well suited for expressing both algorithms and rules. Although SP is a component of ASP, it can also be used by human designers. The SP environment consists of a technology independent compactor that creates spaced layout and simulation data files from symbolic sticks, a joiner that joins together cells generated by the compactor, and a switch level simulator.

4.1. The System

Current layout systems are composed of programs that have been written independently of each other. This often results in a duplication of work and a need for conversion programs. The programs within the SP system have been designed to work together. For example, while spacing the elements from a cell file, the compactor saves all the elements on the border of the cell into a border file for the joiner. The joiner can then space cells properly without again searching through each cell for border elements. This is in contrast to other systems where the program that compacts cells is written independently of the program that joins cells.

Previous approaches to integrated VLSI design environments were generally based upon conventional programming languages using custom data managers with strict data formats ([OCT], [Symbolic-IC]). Objects in these data managers can only be generated through a fixed data field. For example, many databases group wires by layer. To find wires of the same layer, one simply calls a generator that returns instances of wires that are of the queried layer. However, if one wants to find all the wires of an electrical node, one cannot simply call a generator to generate the wires of the node. One must first generate the wires by layer and then filter out the wires that are not of the desired node [OCT]. By using the relational database inherent in Prolog, SP allows generation of objects by any arbitrary number of data fields. Furthermore, individual data fields may be represented by objects. This allows specific fields to be parameterized. For example, the W/L ratio of an output transistor in a cell can be expressed as a parameter and modified without any knowledge of the location of the transistor. This gives the CAD designer a simple but powerful method of accessing data.

Topolog generates male and female single tier cells (cells composed of one p-strip and one n-strip). These cells are individually compacted by the SP compactor, joined so that the n-well from the male cell and the n-well from the female cell share a ground rail, and then arrayed by the joiner.

4.2. The Compactor

The SP compactor takes a cell defined in the Sticks In Prolog (SIP) language and creates a mask level representation for the cell using a new compaction technique that is both algorithmic and rule based. An algorithm similar to zone refining [Zone] is used to perform a rough spacing of the elements. For each compaction pass, a floor and ceiling profile for each layer of material is maintained. In zone refining each element is moved from the ceiling profile to an optimum site on the floor. The SP compactor moves elements directly across the 'molten region' to the floor, where spacing requirements are satisfied, and diagonal

constraints are noted. Rules are then employed to shift elements for a better fit within their environment. By resolving diagonal constraints after the horizontal and vertical compaction passes have completed, the compactor can relieve each constraint by adding space in either the vertical or the horizontal direction, whichever costs less. By treating each layout element as an object, the compactor can easily interpret new layout objects such as bipolar transistor elements to suit mixed technology processes.

For each cell, a connectivity file containing nodal connectivity, resistivity and capacitance information is generated for the simulator and for Spice. The SP compactor is relatively technology independent. A design rule file and a set of technology dependent rules are specified for each technology.

4.3. The Joiner

Large layouts in SP are realized by joining small cells together with the joiner. Leaf cells (cells of the lowest level consisting only of transistors and wires) are compacted individually and are the building blocks for larger modules. There are two methods for joining cells, pitchmatching and river routing. Pitchmatching causes expansion in one axis, while river routing causes expansion in the other. Previous tilers have either exclusively pitchmatched or river routed cells together [V-Grid]. The joiner program connects signals between a given pair of cells by either pitchmatching or river routing, whichever is more area efficient. Directional constraints can override the joiner (that is, if a horizontal constraint is placed, the joiner will river route all signals joined vertically, and pitchmatch all signals joined horizontally). 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.

4.4. The Simulator

The built-in switch level simulator simulates the operation of cells compacted by the compactor or cells joined by the joiner. The simulator asserts given input values at the input nodes and propagates those values to all the other nodes throughout the circuit. Feedback paths are noted and their nodal values are saved for calculation of the next state. The simulator is unique in that it makes extensive use of Prolog backtracking for determining the value of nodes within a circuit.

4.5. The Design Environment

There are many characteristics of CAD elements that make them difficult to represent in a database [CAD-DB], [VLSI-DB]. Each element has many features that associate it with other elements. For example, a wire may be related to other wires by node, by layer, and by location. A CAD tool should be able to select elements by any features as well as assign new features and relations.

A VLSI database must:

- Provide a method for representing objects and structures as well as relations between objects.
- Provide an abstraction to allow the user to access data efficiently without burdening the user with details of operation.
- Interface well with the programming environment. The programming language must be powerful enough to manipulate data efficiently.

The relational database inherent in Prolog is well suited for meeting these requirements.

4.5.1. Prolog as a Database

To model the many complex CAD structures as well as the relationships between structures, many CAD environments use object oriented databases. CAD elements, whether they be geometry for a compactor, transition states for a simulator, or logic expressions for a logic minimizer, can all be expressed in terms of objects. Relationships between the elements can be expressed in terms of groups. For example, elements in a cell can be grouped by node or by location as well as by layer. Current object oriented databases for CAD have strict set relations [OCT]. For example, many databases categorize wires by layer but not location. To find wires of the same layer, one simply calls a generator that returns instances of wires that are of the queried layer. But to find wires of the same grid, one cannot simply generate wires based upon the grid information, but must generate wires by layer and filter out the wires that are not of a common grid. Data in Prolog can be linked by both structure and value. Thus the procedure for generating all wires on the metal-1 layer is the same as the procedure for generating all wires on row 5, or all wires of node vdd, or all wires of row 5 and node vdd in metal-1. Data can also be stored in structures (such as binary trees or sorted lists) for faster access. These constructs provide the ASP Prolog database with a flexible syntax. Elements ranging from behavioral descriptions to logic equations to an ALU layout are all directly expressed in and referenced through Prolog.

4.5.2. Sticks in Prolog

Sticks in Prolog (SIP) is a grid based sticks representation in Prolog that supports hierarchy and parameterized elements. Module generators or human designers generate SIP files which are converted to mask geometry by the Sticks-Pack compactor. In SIP, VLSI elements are modeled as facts. Attributes for the elements are represented as atoms within the facts. There are four types of facts in SIP:

wire(Layer, pt(X1, Y1), pt(X2, Y2), Width, Net).
cont(Type, pt(X1, Y1), Offset, Net).
transistor (Type, pt(SX1, SY1), pt(GX2, GY2), pt(DX3, DY3),
Width, Length, Nets, Netg, Netd).
pin(Layer, pt(X1, Y1), Element).

Layer can be one of the atoms *m1, m2, p, pd, nd.* These represent the physical layers of the element (metal-1, metal-2, poly, p-diffusion, and n-diffusion).

Contact types can be one of the atoms *m1m2, m1pd, m1nd, m1p* (metal-1-to-metal-2, metal-1-to-p-diff, metal-1-to-n-diff, metal-1-to-poly). Contact offsets can be one of the atoms *nw, nn, ne, ee, se, ss, sw, ww, nof* (northwest, north, northeast, east, southeast, south, southwest, west, none).

Width, Length, and X and Y coordinates are integers. *pt(X, Y)* represents a point location at (X, Y). The *Net* field is an atom that is the node name of the element, representing its connectivity. Elements of the same node are electrically connected. Nodal information is supplied by the cell generator or can be extracted by a net extractor.

Transistors have 3 point locations, one for the source, one for the gate, and one for the drain. They also have three nodes, one for each terminal.

The *Element* field for pins contains the element that the pin is attached to.

For example, the following defines an inverter in SIP:

```
wire(m1, pt(0,0), pt(0,5),2,vdd).
wire(m1, pt(0,1), pt(2,1),2,vdd).
wire(m1, pt(10,0), pt(10,5),2,vss).
wire(m1, pt(10,1), pt(8,1),2,vss).
wire(m1, pt(8,3), pt(2,3),2,out).
wire(m1, pt(6,3), pt(6,5),2,out.
wire(p, pt(8,2), pt(2,2),2,in).
wire(p, pt(6,0), pt(6,2),2,in).
trans(nd, pt(2,1), pt(2,2), pt(2,3), 4, 2, vdd, in, out).
trans(pd, pt(8,1), pt(8,2), pt(8,3), 2, 2, vss, in, out).
cont(m1pd, (2,1), nof, vdd).
cont(m1pd, (2,3), nof, out).
cont(m1pd, (8,1), nof, vss).
cont(m1pd, (8,3), nof, out).
```

CAD applications frequently generate specific sets of elements. For example, in SP the simulator generates all the elements of nodes adjacent to a given node, the compactor generates all of the elements of the same grid and layer, and the spacer generates all of the terminals of a given cell side. With the SIP representation, data elements can be easily generated by combinations of characteristics. For example, all of the wires that are of m1 of node vdd which have a width greater than 3 can be generated with two lines of Prolog:

```
wire(m1, Pt1, Pt2, Width, vdd),
Width > 3, ...
```

This representation also allows fields to be easily parameterized within a cell. For example, in a cell definition an output transistor can be parameterized with the statement

```
parameter(outputtrans, pt(2, 3)).
```

A call to the following clause would modify the W/L ratio of any transistor that has been parameterized.

```
modtsize(Name, Neww, Newl):-
  parameter(Name, pt(Xloc, Yloc)),
  retract(trans(Layer, pt(Sy, Sy), pt(Xloc, Yloc), pt(Dx, Dy),
        _, _, Ns, Ng, Nd)),
  assert(trans(Layer, pt(Sy, Sy), pt(Xloc, Yloc), pt(Dx, Dy),
        Neww, Newl, Ns, Ng, Nd)), !.
modtsize(Name, Neww, Newl):-
  write('transistor not found'), !.
```

This flexibility allows tools to address and modify specific elements within any context. For example, a program that tries to optimize the performance of a circuit containing many cells can do so by adjusting the W/L ratio of the output transistors. With the output transistors parameterized, the program can reference the output transistors from any cell simply as *outputrans* regardless of where the transistor is or what the transistor is attached to.

SIP provides an excellent abstraction of VLSI layout for an automated module generator. For example, the following clause

```
makeinvert(Vddgrid, Vssgrid, Ingrid, Outgrid, Pw, Pl, Nw, Nl):-
  Pdgrid is Vddgrid - 1,
  Ndgrid is Vssgrid + 1,
  assert(wire(m1, pt(2, Vddgrid), pt(2, Pdgrid), 1, unk)),
  assert(wire(m1, pt(2, Vssgrid), pt(2, Ndgrid), 1, unk)),
  assert(wire(m1, pt(1, Vddgrid), pt(5, Vddgrid), 1, unk)),
  assert(wire(m1, pt(1, Vssgrid), pt(5, Vssgrid), 1, unk)),
  assert(wire(m1, pt(4, Pdgrid), pt(4, Ndgrid), 1, unk)),
  assert(wire(m1, pt(4, Outgrid), pt(5, Outgrid), 1, unk)),
  assert(wire(p, pt(3, Pdgrid), pt(3, Ndgrid), 1, unk)),
  assert(wire(p, pt(0, Ingrid), pt(3, Ingrid), 1, unk)),
  assert(cont(m1d, pt(2, Pdgrid), nof, unk)),
  assert(cont(m1d, pt(2, Ndgrid), nof, unk)),
  assert(cont(m1d, pt(4, Pdgrid), nof, unk)),
  assert(cont(m1d, pt(4, Pdgrid), nof, unk)),
  assert(trans(pd, pt(1, Pdgrid), pt(2, Pdgrid), pt(3, Pdgrid),
       Pw, Pl,unk, unk, unk)),
  assert(trans(nd, pt(1, Ndgrid), pt(2, Ndgrid), pt(3, Ndgrid),
       Nw, Nl,unk, unk, unk)), !.
```

will generate an arbitrarily sized inverter with variable input and output locations. Nodal information is deduced by the extractor. ROMs, PLAs, and other modular layout styles can be generated in a similar fashion.

4.5.3. Language and Data Integration in Prolog

Most CAD applications rest upon a database substrate. They do not, however, rest uniformly on the substrate. The line defining the facilities of the CAD data manager would, in an ideal world, be drawn differently for each application. In the real world of separate databases and application programs the capabilities of the CAD data manager must be defined once, and not vary by application.

In the Prolog world, the line between database and application is hazy, and often transparent. In Prolog there is no syntactic or semantic difference between a procedure call and a database query.

Design rules specify the distances between elements. In the compactor these rules are expressed as facts, such as

```
space(poly, pd, 2).
space(poly, nd, 2).
space(pd, nd, 2).
```

The order in which the layers appear does not matter (for instance, the poly to pd spacing is the same as pd to poly spacing). A simple procedure referencing the space facts will make them order independent.

```
findspace(Layer1, Layer2, Dist):-
   space(Layer1, Layer2, Dist).
findspace(Layer1, Layer2, Dist):-
   space(Layer2, Layer1, Dist).
findspace(Layer1, Layer2, notfound).
```

Facts can in general be easily retrieved and processed. One method of iteration in Prolog that can be used with SIP data is the explicit fail loop. For example, with the layers

```
layer(m1).
layer(m2).
layer(p).
layer(nd).
layer(pd).
```

the following procedure generates all wires by layer:

```
getwirebylayer:-
   layer(Layer),
   wire(Layer, Pt1, Pt2, Width, Node),
   {process the wire...}
   fail.
getwirebylayer.
```

When the fail is encountered, Prolog backtracks over the processing goals and gets another wire instance if it exists. If it does not, Prolog gets a new layer value. If that fails, Prolog drops to the next clause, which is always true.

It is easy to construct custom data managers in Prolog. Much data in SP is passed from clause to clause through lists of *bundles*, which are in turn lists. A bundle is a list of data elements created by a clause such as:

```
Field1, Field2, Field3, Bundle):-
   Bundle = [Field1, Field2, Field3].
```

A bundle (the first in a list) can be selected and disassembled with

```
processBundle([Head|ListOfBundles]):-
   listio(Field1, Field2, Field3, Head),
   ...
```

Lists of bundles allow Sticks-Pack to manipulate data as list elements. Other custom data managers are also employed to provide constructs such as sorted trees for efficiency and simplicity.

4.5.4. Prolog Programming for CAD

There has been a growing trend in CAD to develop tools that use both algorithmic and rule-based programming styles [Expert], [Rules]. Algorithms are generally fast, but are inefficient at handling problems that have many special cases. Rule-based systems are well suited for solving problems with many special cases or problems that are not well defined. Rule-based systems have generally been slow. The rules must be looked up and efficient management systems have not yet been developed. Some CAD problems have algorithmic

solutions (such as simulation), but most are computationally expensive (such as routing and logic minimization), and can be solved by a host of approximation techniques, including rule-based heurisics.

Prolog provides an environment for both algorithmic and rule-based programming styles. Its clausal nature allows rules to be easily updated or modified. Algorithms can be expressed quickly and easily, which makes Prolog an ideal language for rapid prototyping.

A current philosophy in CAD systems is to develop CAD tools that are 'technology independent' or 'technology insensitive'. Tools have been developed with information regarding technology expressed as a set of parameters, with the data for a certain technology loaded from a technology file. Because of this, the tools have not been able to utilize fully benefits that certain technologies have to offer. For example, in the automatic generation of random logic metal-1 to metal-2 vias are expensive in area. In certain technologies, the area of diffusion between two series transistors is an ideal site for the via as metal-1 and metal-2 are both routable to the site and the spacing between the transistors is about the same as the size of the via. Such a condition is difficult to express algorithmically and is very technology dependent, but would be useful in minimizing area. The SP compactor supplements its set of technology parameters with a set of rules that allow the compactor to compact cells more tightly.

4.5.5. Prolog Programming Methodology Employed by Sticks-Pack

Three basic formats for Prolog clauses are employed by SP:

Deterministic Clauses. These clauses work to achieve a certain value or state without failing.

Filtering Clauses. These clauses, given a set of data elements, interpret each element differently depending upon the values of certain data fields. If-then and case constructs can be expressed through these clauses.

Generator Clauses. These clauses generate an element or set of elements through backtracking.

An example of a deterministic clause is the *mindist* procedure. It finds the minimum spacing distance between two objects of specified layer and width. The *space* procedure returns the minimum spacing distance between two layers, and the *width* procedure determines the minimum width of a layer.

```
mindist(Layer1, Width1, Layer2, Width2, Distbetwnobjcts):-
  space(Layer1, Layer2, Distance),
  width(Layer1, Widthspace1),
  Widthmod1 is Width1*Widthspace1,
  width(Layer2, Widthspace2),
  Widthmod2 is Width2*Widthspace2,
  Distbetwnobjcts is Widthmod1 + Widthmod2 + Distance.
```

An example of filtering clauses is the *checkconstr* procedure. It determines how to space two elements. Each clause filters out a certain condition. If the elements are on the same row, the spacing is irrelevant. If the elements are contacts, they cannot be stacked upon each other and must be spaced accordingly. If the elements are not contacts and are of the same node, the spacing does not matter. Otherwise the elements must be spaced.

```
checkconstr(Layer1, Width1, Node1, Row1, Layer2, Width2, Node2) :-
  Row1=Row2.
checkconstr(Layer1, Width1, Node1, Row1, Layer2, Width2, Node2) :-
  contacts(Layer1, Layer2), ...
checkconstr(Layer1, Width1, Node1, Row1, Layer2, Width2, Node2) :-
  Node1=Node2.
checkconstr(Layer1, Width1, Node1, Row1, Layer2, Width2, Node2) :-
  ...
```

An example of generator clauses is the *makebox* procedure. It creates boxes from various elements, first processing wires, followed by contacts and transistors.

```
makebox :-
  wire(Layer, pt(X1, Y1), pt(X2, Y2), Wid, Node),
  ...
  fail.
makebox :-
  cont(Type, pt(Row, Y), Oset, _),
  ...
  fail.
makebox :-
  trans(Type, pt(Sx, Sy), pt(Gx, Gy), pt(Dx, Dy), W, L, Sn, Gn, Dn),
  ...
  fail.
makebox.
```

All of the procedures in SP employ a combination of these three basic formats.

5. The New Version of ASP

The new version of ASP currently being completed is aimed at solving three problems with the prototype: generality, maintainability, and speed.

Generality was a problem with the prototype system because it was not designed to support complex designs. In particular, input specifications were constrained to have a single loop and a single case dispatch. Furthermore, the system could only generate data paths in which all bit slices were identical. These limitations required new Prolog translation code and a new data path generator.

Maintainability was a problem because some of the code would not port from C-Prolog to Quintus Prolog, which was necessary to take advantage of Quintus garbage collection. In particular, the Topolog module generator had to be replaced because of this problem.

Execution speed was a problem in general with the lower level tools, which had to deal with thousands of geometric elements, and were orders of magnitude slower than equivalent C programs. In particular, we have rewritten the compactor to use lists instead of assert and retract, in a successful effort to improve its performance.

In addition to addressing the above problems, the reimplementation also caused a change in the system's structure. In the prototype system behavioral and boolean synthesis were combined. In the current system they have been split, which clarifies the separate issues raised at each level.

6. References

[Allende]
'Allende: A Procedural Language for the Specification of VLSI Layouts'; J.M. da Mata; *22nd Design Automation Conference*, 1985.

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

[ASP]
'An Advanced Silicon Compiler in Prolog'; William R. Bush, Gino Cheng, Patrick C. McGeer, Alvin M. Despain; *1987 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1987, pp. 27-31.

[BUD]
'Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions'; Michael C. McFarland; *23rd Design Automation Conference*, June 1986; pp. 474-480.

[CAD-DB]
New Features for a Relational Database System to Support Computer Aided Design; A. Guttman; Ph.D. thesis, U.C. Berkeley, 1984.

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

[CHS]
'Design Considerations for a Prolog Silicon Compiler'; Patrick C. McGeer et alia; Internal Memorandum, November 1985.

[CMU-DA]
'Automatic Data Path Synthesis'; Donald E. Thomas, Charles Y. Hitchcock, Thaddeus J. Kowalski, Jayanth V. Rajan, Robert A. Walker; *IEEE Computer*; December 1983; pp. 59-70.

[Expert]
'An Expert-System Paradigm for Design'; F.D. Brewer, D.D. Gajski; *23rd Design Automation Conference*, June 1986.

[G-Matrix]
'A Dense Gate Matrix Layout Method for MOS VLSI'; A.D. Lopez, H-F.S. Law; *IEEE Transactions on Electronic Devices*, Vol ED-27, August 1980.

[GENIE]
'GENIE: A Generalized Array Optimizer for VLSI Synthesis'; S. Devadas, A.R. Newton; *23rd Design Automation Conference*, June 1986.

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

[OCCAM]
'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; pp. 381-390.

[OCT]
'Data Management and Graphics Editing in the Berkeley Design Environment'; D. Harrison, P. Moore, R. Spickelmeier, A.R. Newton; *Proceedings of the IEEE International Conference on CAD*, November 1986.

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

[rplacarg]
Data Structures and Destructive Assignment in Prolog; P.C. McGeer, A.M. Despain; UCB Technical Report, 1987.

[Rules]
'Rule Based and Algorithmic Approach for Logic Synthesis'; T. Yoshimura, S. Goto; *Proceedings of the IEEE International Conference on CAD*, November 1986.

[SICStus]
SICStus Prolog Documentation; Swedish Institute of Computer Science; 1987.

[SWAMI]
'SWAMI: A Flexible Logic Implementation System'; C. Rowen, J. Hennessy; *22nd Design Automation Conference*, 1985.

[Symbolic-IC]
'A Symbolic Design System for Integrated Circuits'; K.H. Keller, A.R. Newton; *19th Design Automation Conference*, 1982.

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

[V-Grid]
'Virtual Grid Symbolic Layout', N. Weste, *18th Design Automation Conference*, 1981, pp. 225-233.

[Viper]
'Experience with Prolog as a Hardware Specification Language'; William R. Bush, Gino Cheng, Patrick C. McGeer, Alvin M. Despain; *Fourth Symposium on Logic Programming*, September 1987, pp. 490-498.

[VLSI-DB]
'A Database Approach for Managing VLSI Design Data'; R.H. Katz; *19th Design Automation Conference*, 1982.

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

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

% 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 ).
```

Appendix 1: A Simple Microprocessor Specification in Prolog

```
sml :-
    fetch,
    lookup(reg4, OP),
    execute(OP),
    sml.
sml :-
    true.
fetch :-
    access(reg2, PC), set(memAR, PC),
    mem_readinst,
    access(memDR1, OP), set(reg4, OP),
    access(memDR2, X), set(reg5, X),
    access(reg2, PC),
    P1 is PC+1,
    set(reg2, P1).
execute(halt) :-
    !, fail.
execute(add) :-
    !,
    access(reg5, X), set(memAR, X),
    mem_read,
    access(memDR, T), access(reg1, AC),
    A is T+AC,
    set(reg1, A).
execute(stor) :-
    access(reg5, X), set(memAR, X),
    access(reg1, AC), set(memDR, AC),
    mem_write, !.
execute(brn) :-
    access(reg1, AC),
    AC<0,
    !,
    access(reg5, T), set(reg2, T).
execute(brn) :-
    true.
```

Appendix 2a: The Memory Register Version

```
access(X, Y) :-
    Z =.. [X, Y],
    Z.
lookup(X, Y) :- access(X, Y).
set(X, Y) :- abolish(X, 1),
    Z =.. [X, Y],
    assert(Z).

mem_read :-
    access(memAR, Loc),
    mem(Loc, Data),
    set(memDR, Data).
mem_readinst :-
    access(memAR, Loc),
    mem(Loc, Data1, Data2),
    set(memDR1, Data1),
    set(memDR2, Data2).
mem_write :-
    access(memAR, Loc),
    mem(Loc, _), !,
    retract(( mem(Loc, _) )),
    access(memDR, Data),
    assert(( mem(Loc, Data) )).
mem_write :-
    access(memAR, Loc),
    access(memDR, Data),
    assert(( mem(Loc, Data) )).
```

Appendix 2b: The Memory System

```
srcVar(reg2,var1).
srcVar(memDR1,var2).
srcVar(memDR2,var3).
srcVar(reg2,var1).
srcVar(reg5,var5).
srcVar(memDR,var6).
srcVar(reg1,var7).
srcVar(reg5,var9).
srcVar(reg1,var10).
srcVar(reg1,var11).
srcVar(reg5,var12).

expVars(var1,1,+,var4).
expVars(var6,var7,+,var8).
expVars(var11,0,<,none).

dstVar(memAR,var1).
dstVar(reg4,var2).
dstVar(reg5,var3).
dstVar(reg2,var4).
dstVar(memAR,var5).
dstVar(reg1,var8).
dstVar(memAR,var9).
dstVar(memDR,var10).
dstVar(reg2,var12).
```

Appendix 3: Transfer Fragments

```
transfer(op1,block1,reg2,none,transfer,memAR).
transfer(op2,block1,none,none,mem,readinst).
transfer(op3,block1,memDR1,none,transfer,reg4).
transfer(op4,block1,memDR2,none,transfer,reg5).
transfer(op5,block1,reg2,1,+,reg2).
transfer(op6,block3,reg5,none,transfer,memAR).
transfer(op7,block3,none,none,mem,read).
transfer(op8,block3,memDR,reg1,+,reg1).
transfer(op9,block4,reg5,none,transfer,memAR).
transfer(op10,block4,reg1,none,transfer,memDR).
transfer(op11,block4,none,none,mem,write).
transfer(op12,block5,reg1,0,<,none).
transfer(op13,block6,reg5,none,transfer,reg2).

branch(block1,cond,reg4).
branch(block3,uncond,block1).
branch(block4,uncond,block1).
branch(block5,cond,<).
branch(block6,uncond,block1).

case(block1,halt,halt).
case(block1,add,block3).
case(block1,stor,block4).
case(block1,brn,block5).
case(block5,<,block6).
case(block5,=>,block1).
```

Appendix 4: The Transfer-Based Version

```
implicitDependent(mem,memAR).
implicitDependent(memDR,mem).
implicitDependent(memDR1,mem).
implicitDependent(memDR2,mem).

dependent(op5,op1,reg2).
dependent(op2,op1,mem,memAR).
dependent(op3,op2,memDR1,mem).
dependent(op4,op2,memDR2,mem).
dependent(op7,op6,mem,memAR).
dependent(op8,op7,memDR,mem).
dependent(op11,op9,mem,memAR).
```

Appendix 5: The Dependency Data Base

```
cycle(op1,block1,1).
cycle(op2,block1,2).
cycle(op3,block1,3).
cycle(op4,block1,3).
cycle(op5,block1,2).
cycle(op6,block3,1).
cycle(op7,block3,2).
cycle(op8,block3,3).
cycle(op9,block4,1).
cycle(op10,block4,1).
cycle(op11,block4,2).
cycle(op12,block5,1).
cycle(op13,block6,1).
```

Appendix 6: The Transfer Schedule

```
elementType(reg2,reg).
elementType(memAR,reg).
elementType(mem,memory).
elementType(memDR,reg).
elementType(reg4,reg).
elementType(reg5,reg).
elementType(reg1,reg).
elementType(adder1,adder).
elementType(bus1,bus).
elementType(bus2,bus).
elementType(bus3,bus).

elementFn(mem,read).
elementFn(mem,write).
elementFn(adder1,add).
elementFn(adder1,inc).

elementUse(mem,read,block1,2,op2).
elementUse(mem,read,block3,2,op7).
elementUse(mem,write,block4,2,op11).
elementUse(reg2,src,block1,1,op1).
elementUse(memAR,dst,block1,1,op1).
elementUse(memDR,src,block1,3,op3).
elementUse(reg4,dst,block1,3,op3).
elementUse(memDR,src,block1,3,op4).
elementUse(reg5,dst,block1,3,op4).
elementUse(adder1,inc,block1,2,op5).
elementUse(reg2,sad,block1,2,op5).
elementUse(reg5,src,block3,1,op6).
elementUse(memAR,dst,block3,1,op6).
elementUse(adder1,add,block3,3,op8).
elementUse(memDR,src,block3,3,op8).
elementUse(reg1,sad,block3,3,op8).
elementUse(reg5,src,block4,1,op9).
elementUse(memAR,dst,block4,1,op9).
elementUse(reg1,src,block4,1,op10).
elementUse(memDR,dst,block4,1,op10).
elementUse(reg5,src,block6,1,op13).
elementUse(reg2,dst,block6,1,op13).

elementTest(sign,block5,reg1,reg1sign,op12).
elementTest(switch,block1,reg4,reg4out,none).
```

Appendix 7a: The Data Path Data Base -- Units

```
busSrc(bus1,reg2).
busSrc(bus1,memDR).
busSrc(bus2,memDR).
busSrc(bus2,adder1).
busSrc(bus1,reg5).
busSrc(bus2,reg1).
busSrc(bus3,adder1).

busDst(bus1,memAR).
busDst(bus1,reg4).
busDst(bus2,reg5).
busDst(bus1,adder1port1).
busDst(bus2,reg2).
busDst(bus2,adder1port2).
busDst(bus3,reg1).
busDst(bus2,memDR).
busDst(bus1,reg2).

busUse(bus1,reg2,memAR,block1,1,op1).
busUse(bus1,memDR,reg4,block1,3,op3).
busUse(bus2,memDR,reg5,block1,3,op4).
busUse(bus1,reg2,adder1port1,block1,2,op5).
busUse(bus2,adder1,reg2,block1,2,op5).
busUse(bus1,reg5,memAR,block3,1,op6).
busUse(bus1,memDR,adder1port1,block3,3,op8).
busUse(bus2,reg1,adder1port2,block3,3,op8).
busUse(bus3,adder1,reg1,block3,3,op8).
busUse(bus1,reg5,memAR,block4,1,op9).
busUse(bus2,reg1,memDR,block4,1,op10).
busUse(bus1,reg5,reg2,block6,1,op13).
```

Appendix 7b: The Data Path Data Base -- Buses

```
functionalUnit(reg1,reg,
        [bus3],[bus2],[reg1Fn],[reg1sign]).
functionalUnit(reg2,reg,
        [reg2mBus],[bus1],[reg2Fn],[]).
functionalUnit(reg2mux,mux,
        [bus1,bus2],[reg2mBus],[reg2Mux],[]).
functionalUnit(reg4,reg,
        [bus1],[],[reg4Fn],[reg4out]).
functionalUnit(reg5,reg,
        [bus2],[bus1],[reg5Fn],[]).
functionalUnit(memAR,reg,
        [bus1],[],[memARFn],[]).
functionalUnit(memDR,reg,
        [bus2],[memDRdBus],[memDRFn],[]).
functionalUnit(memDRdecoder,decoder,
        [memDRdBus],[bus1,bus2],[memDRDecode],[]).
functionalUnit(adder1,adder,
        [bus1,bus2],[adder1dBus],[adder1Fn],[adder1Cout]).
functionalUnit(adder1decoder,decoder,
        [adder1dBus],[bus2,bus3],[adder1Decode],[]).

controlIn(regFn,reg1Fn,hold).
controlIn(regFn,reg2Fn,hold).
controlIn(muxFn,reg2Mux,[bus1,bus2]).
controlIn(regFn,reg4Fn,hold).
controlIn(regFn,reg5Fn,hold).
controlIn(regFn,memARFn,hold).
controlIn(regFn,memDRFn,hold).
controlIn(decodeFn,memDRDecode,[bus1,bus2]).
controlIn(adderFn,adder1Fn,pass).
controlIn(decodeFn,adder1Decode,[bus2,bus3]).

controlOut(switch,reg4out,[add,brn,halt,stor]).
controlOut(sign,reg1sign,block5).
```

Appendix 8: The Data Path

```
%fetch
state(block1Cycle1,
        [output(memARFn,dst), output(reg2Fn,src)],
        block1Cycle2).
state(block1Cycle2,
        [output(adder1Decode,bus2), output(reg2Mux,bus2),
         output(adder1Fn,inc), output(memFn,read), output(reg2Fn,sad)],
        block1Cycle3).
state(block1Cycle3,
        [output(memDRDecode,bus1), output(memDRDecode,bus2),
         output(memDRFn,src), output(reg4Fn,dst), output(reg5Fn,dst)],
        switch(reg4out,
           [case(add,block3Cycle1), case(brn,block5Cycle1),
            case(halt,haltCycle1), case(stor,block4Cycle1)])).
%add
state(block3Cycle1,
        [output(memARFn,dst), output(reg5Fn,src)],
        block3Cycle2).
state(block3Cycle2,
        [output(memFn,read)],
        block3Cycle3).
state(block3Cycle3,
        [output(memDRDecode,bus1), output(adder1Decode,bus3),
         output(adder1Fn,add), output(memDRFn,src), output(reg1Fn,sad)],
        block1Cycle1).
% stor
state(block4Cycle1,
        [output(memARFn,dst), output(memDRFn,dst),
         output(reg1Fn,src), output(reg5Fn,src)],
        block4Cycle2).
state(block4Cycle2,
        [output(memFn,write)],
        block1Cycle1).
%brn
state(block5Cycle1,[],
        switch(reg1sign,
           [case(gezero,block1Cycle1), case(ltzero,block6Cycle1)])).
state(block6Cycle1,
        [output(reg2Mux,bus1), output(reg2Fn,dst), output(reg5Fn,src)],
        block1Cycle1).
```

Appendix 9: The Control Path

```
opFnMapping('+', add, arlog).

%memory
signalValue(memFn, read, 1).
signalValue(memFn, write, 3).
%register

signalValue(regFn, src, 0).
signalValue(regFn, dst, 1).
signalValue(regFn, sad, 1).

lib(adder).
twoPortType(adder).
signalValue(adderFn, add, 0).
signalValue(adderFn, inc, 1).

%sign bit output
signalValue(sign, gezero, 0).
signalValue(sign, ltzero, 1).
```

Appendix 10a: The Functional Unit Library

```
reg([In], [Out], [Load, Clock], _, Block) :-
    buildBlock(Refresh = aoi(Load),B1),
    buildBlock(Masterin = transmit(Out,Load,Refresh),B2),
    buildBlock(Masterin = transmit(In,Refresh,Load),B3),
    buildBlock(Masterout = aoi(Masterin),B4),
    buildBlock(Slavein = pass(Masterout, Clock),B5),
    buildBlock(Out = aoi(Slavein),B6),
    buildCompositeBlock([B1,B2,B3,B4,B5,B6], Block).

reg([In], [Out, Top], [Load, Clock], _, Block) :-
    buildBlock(Refresh = aoi(Load),B1),
    buildBlock(Masterin = transmit(Out,Load,Refresh),B2),
    buildBlock(Masterin = transmit(In,Refresh,Load),B3),
    buildBlock(Masterout = aoi(Masterin),B4),
    buildBlock(Slavein = pass(Masterout, Clock),B5),
    buildBlock(Out = aoi(Slavein),B6),
    buildBlock(Top = aoi(Slavein),B7),
    buildCompositeBlock([B1,B2,B3,B4,B5,B6,B7], Block).

decoder2([Input], [Output1, Output2], [Control], _, Block) :-
    buildBlock(not(Control) = aoi(Control), B1),
    buildBlock(Output1 = transmit(Input, Control, not(Control)), B2),
    buildBlock(Output2 = transmit(Input, not(Control), Control), B3),
    buildCompositeBlock([B1,B2,B3], Block).

mux2([Input1, Input2], [Output], [Control], _, Block) :-
    buildBlock(CBar = aoi(not(Control)), B1),
    buildBlock(Output =
        aoi(or( and(Input1, CBar), and(Input2, Control) )), B2),
    buildCompositeBlock([B1,B2], Block).

adder([A, B], [Sum], [Cin], [Cout], Block) :-
    buildBlock(X = aoi(or(and(Cin,or(A,B)),and(A,B))),B1),
    buildBlock(Y = aoi(or(and(X,or(A,B,Cin)),and(A,B,Cin))),B2),
    buildBlock(Sum = aoi(Y),B3),
    buildBlock(Cout = aoi(X),B4),
    buildCompositeBlock([B1,B2,B3,B4], Block).
```

Appendix 10b: The Logic Equation Library

```
plavar(state(0)).
plavar(state(1)).
plavar(state(2)).
plavar(state(3)).
plavar(reg4out(0)).
plavar(reg4out(1)).
plavar(reg1sign(0)).

alias(reg2Mux0,adder1Fn0).
alias(memDRDecode0,reg4Fn0).
alias(memDRDecode0,reg5Fn0).
alias(adder1Decode0,reg1Fn0).
```

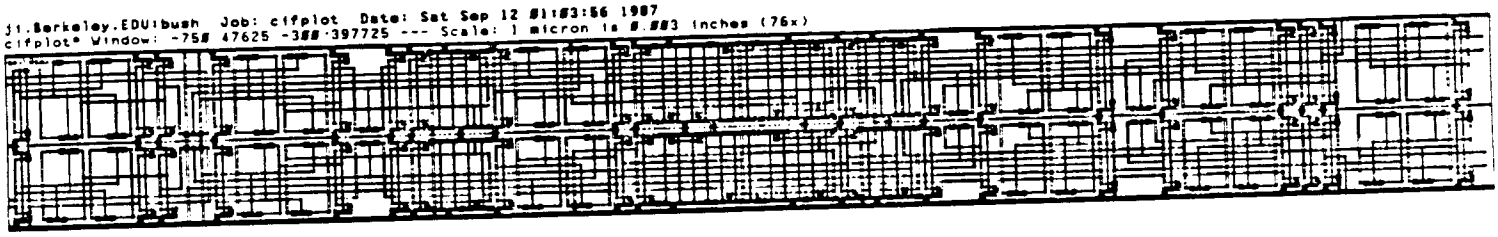
Appendix 11a: The PLA Equations -- Inputs and Aliases

```
pterm(stateblock1Cycle1,[inv(state(0)),inv(state(1)),inv(state(2)),state(3)]).
pterm(stateblock3Cycle1,[inv(state(0)),state(1),inv(state(2)),inv(state(3))]).
pterm(stateblock4Cycle1,[state(0),inv(state(1)),state(2),inv(state(3))]).
pterm(stateblock1Cycle2,[inv(state(0)),inv(state(1)),inv(state(2)),
       inv(state(3))]).
pterm(stateblock3Cycle2,[inv(state(0)),state(1),state(2),inv(state(3))]).
pterm(stateblock1Cycle3,[state(0),inv(state(1)),inv(state(2)),inv(state(3))]).
pterm(stateblock3Cycle3,[state(0),state(1),state(2),inv(state(3))]).
pterm(stateblock1Cycle3reg4outadd,[state(0),inv(state(1)),inv(state(2)),
       inv(state(3)),reg4out(0),inv(reg4out(1))]).
pterm(stateblock1Cycle3reg4outbrn,[state(0),inv(state(1)),inv(state(2)),
       inv(state(3)),reg4out(0),reg4out(1)]).
pterm(stateblock1Cycle3reg4outhalt,[state(0),inv(state(1)),inv(state(2)),
       inv(state(3)),inv(reg4out(0)),inv(reg4out(1))]).
pterm(stateblock1Cycle3reg4outstor,[state(0),inv(state(1)),inv(state(2)),
       inv(state(3)),inv(reg4out(0)),reg4out(1)]).
pterm(stateblock6Cycle1,[inv(state(0)),state(1),inv(state(2)),state(3)]).
pterm(stateblock5Cycle1reg1signgezero,[state(0),state(1),inv(state(2)),
       inv(state(3)),inv(reg1sign(0))]).
pterm(stateblock4Cycle2,[state(0),inv(state(1)),inv(state(2)),state(3)]).
pterm(stateblock5Cycle1reg1signltzero,[state(0),state(1),inv(state(2)),
       inv(state(3)),reg1sign(0)]).

oterm(state(0),[stateblock1Cycle2,stateblock1Cycle3reg4outbrn,
       stateblock1Cycle3reg4outstor,stateblock3Cycle2,stateblock4Cycle1]).
oterm(state(1),[stateblock1Cycle3reg4outadd,stateblock1Cycle3reg4outbrn,
       stateblock3Cycle1,stateblock3Cycle2,stateblock5Cycle1reg1signltzero]).
oterm(state(2),[stateblock1Cycle3reg4outhalt,stateblock1Cycle3reg4outstor,
       stateblock3Cycle1,stateblock3Cycle2]).
oterm(state(3),[stateblock3Cycle3,stateblock4Cycle1,stateblock4Cycle2,
       stateblock5Cycle1reg1signgezero,stateblock5Cycle1reg1signltzero,
       stateblock6Cycle1]).
oterm(memARFn0,[stateblock1Cycle1,stateblock3Cycle1,stateblock4Cycle1]).
oterm(reg2Mux0,[stateblock1Cycle2]).
oterm(reg2Fn0,[stateblock1Cycle2,stateblock6Cycle1]).
oterm(memDRDecode0,[stateblock1Cycle3]).
oterm(adder1Decode0,[stateblock3Cycle3]).
oterm(memDRFn0,[stateblock4Cycle1]).
```

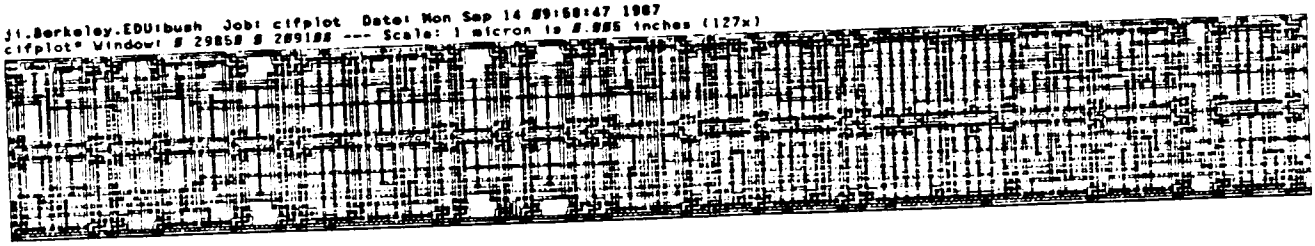
Appendix 11b: The PLA Equations -- Product and Or Terms

```
reg([bus3],[bus2,reg1sign],[reg1Fn0,clock],[]).
reg([reg2mBus],[bus1],[reg2Fn0,clock],[]).
mux2([bus1,bus2],[reg2mBus],[reg2Mux0],[]).
reg([bus1],[reg4out],[reg4Fn0,clock],[]).
reg([bus2],[bus1],[reg5Fn0,clock],[]).
decoder2([memDRdBus],[bus1,bus2],[memDRDecode0],[]).
adder([bus1,bus2],[adder1dBus],[adder1Fn0],[adder1Cout0]).
decoder2([adder1dBus],[bus2,bus3],[adder1Decode0],[]).

mirror.

feed(reg2Fn0).
feed(reg2Mux0).
feed(reg4Fn0).
feed(reg5Fn0).
feed(reg1Fn0).
feed(memDRDecode0).
feed(adder1Decode0).
top(adder1Cout0).
bottom(adder1Fn0).
pairedSignals(adder1Fn0,adder1Cout0).
leftEdge(bus1).
rightEdge(bus2).
rightEdge(memDRdBus).
```

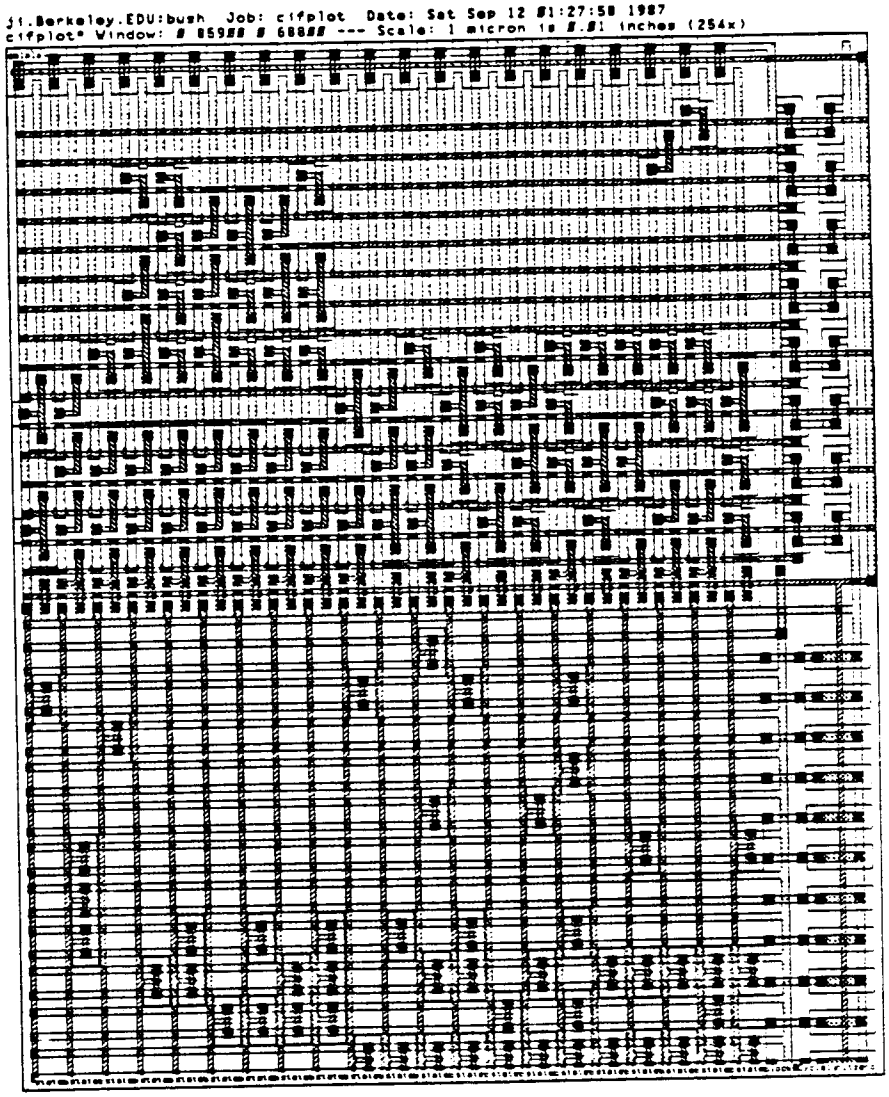
Appendix 12: The Topological Data Path



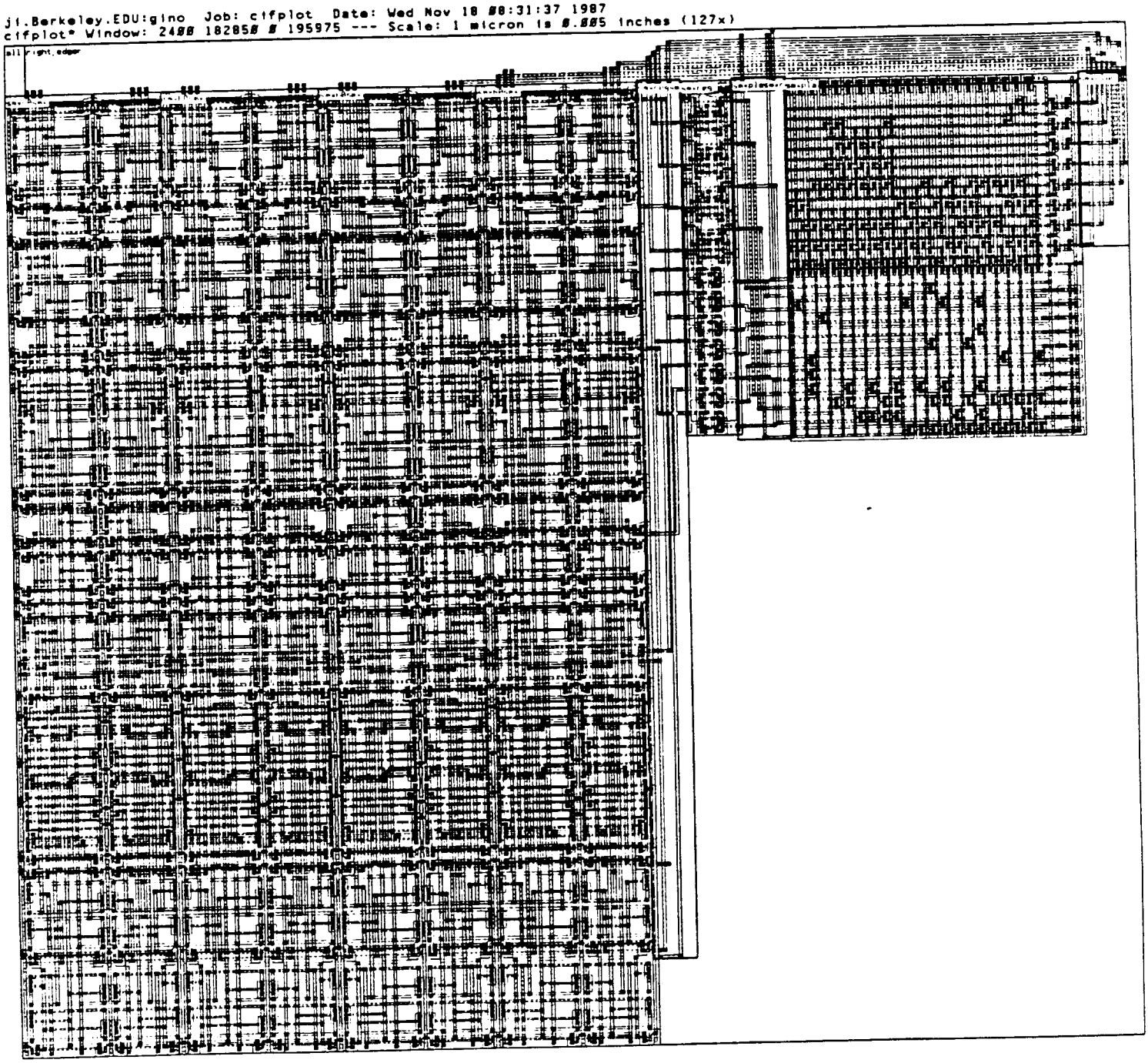
Appendix 13: The Sticks-Based Data Path Bit Slice



Appendix 14: The Compacted Data Path Bit Slice



Appendix 15: The Compacted PLA



Appendix 16: The Final Layout