Proving the Correctness of Digital Hardware Designs

Harry G. Barrow
Fairchild Laboratory for Artificial Intelligence Research
4001 Miranda Ave., Palo Alto, CA 94304

Abstract

VERIFY is a PROLOG program that attempts to prove the correctness of a digital design. It does so by showing that the behavior inferred from the interconnection of its parts and their behaviors is equivalent to the specified behavior. It has successfully verified large designs involving many thousands of transistors.

§1 Introduction

When a hardware or software system is designed, there is always the problem of deciding whether the design meets its specification. Research in proving correctness of programs has continued for many years, with some notable recent successes [Schwartz 82]. The correctness of hardware design, in contrast, has received comparatively little attention until recently, indicating perhaps that now the boundaries between hardware and software are becoming increasingly blurred.

In this paper is described an approach to proving the correctness of digital hardware designs. It is based rather squarely upon Gordon's methods, which concentrate on the denotational semantics of designs and their behavior [Gordon 81]. A key principle is that, given the behaviors of components of a system and their interconnections, it is possible to derive a description of the behavior of the system [Foster 81]. The derived behavior can then be compared with specifications. In our current work, behavior of systems or modules is specified in detail, rather than via general assertions (as in [Floyd 67]). The most closely related work, apart from Gordon's, is that of Shostak [Shostak 83]. However, it would appear that the examples described in this paper represent the most complex so far verified automatically.

VERIFY, a PROLOG program, has been implemented and embodies an initial, simplified version of Gordon's methodology (specifically, omitting the notion of behaviors as manipulatable objects). VERIFY has successfully proved correctness of a straightforward but very detailed design, involving thousands of transistors.

VERIFY gains most of its power from exploiting the structural hierarchy (part-whole), which effectively breaks the design into manageable pieces with simply representable behavior (at some level of meaning). This decomposition results in proof complexity that is linear in the number of types of part, rather than the total number of parts. A second, related, source of power is the signal abstraction hierarchy. Signals in a digital system are viewed differently according to the context and conceptual level in which they are considered. At the lowest level, signals are considered as voltages and currents; at the next, as logic levels with various strengths; at the higher levels, as bits, integers, addresses, or even computational objects. The semantics of higher structural and signal levels allow greater leaps of inference, and much suppression of unnecessary detail. Finally, representing and manipulating signals and functions symbolically leads to considerable improvements over the established methods of verifying designs by detailed simulation, involving vast numbers of specific signal values (zeros and ones).

§2 Specifying a Module

Modules are modeled as finite state machines. A module has a set of input ports and a set of output ports. Each port has an associated signal type, which specifies the domain for signals passing through it. A module also has a set of state variables, each with its own signal type. The behavior of a module is specified by two sets of equations: one set gives output signals as functions of inputs and current internal state, and the other gives new internal states as functions of inputs and current state.

As an example, a simple type of module, called 'inc', which has no state variables, and whose output is simply its input plus one, may be declared as follows:

```
% Definition of module type Incrementer

module(inc).

port(inc,in(Aninc),input,integer).
port(inc,out(Aninc),output,integer).

outputEqn(inc, out(AnInc) := 1+in(AnInc)).
```

In this definition, 'AnInc' is a PROLOG variable that stands for any instance of an incrementer, and 'in' and 'out' are functions from an instance to the current values of signals at its ports. ':=' is an infix operator used to define the behavior of an output or state variable (on its left) as an expression (on its right).

Similarly, a type of simple multiplexer, whose output is one of two inputs, according to the value of a control input, may be declared as:

```
% Definition of module type Multiplexer

module(mux).

port(mux,in0(Amux),input,integer).
port(mux,in1(Amux),input,integer).
port(mux,switch(Amux),input,boole).
port(mux,out(Amux),output,integer).

outputEqn(mux, out(AMux) :=
  if(switch(AMux),in1(AMux),in0(AMux))).
```

Here, 'Amux' stands for any instance of type mux, and 'if(<condition>,<true.expr>,<false.expr>)' is a conditional expression with the obvious semantics.

Finally, a module type that involves an internal state variable:

```
% Definition of module type Register

module(reg).
```

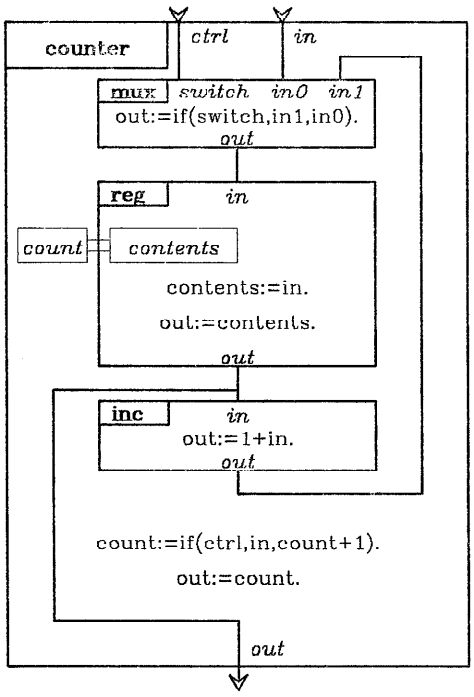

Figure 1. A loadable counter.

```
port(reg,in(Areg),input,integer).
port(reg,out(Areg),output,integer).

state(reg,contents(Areg),integer).

outputEqn(reg,out(AReg) := contents(AReg)).
stateEqn(reg,contents(AReg) := in(AReg)).
```

This describes a simple register, whose current contents are given by the function 'contents'. The equations declare that the output of the register is the current value of its contents, and that the new value of its contents is whatever is the current input. This register thus can be viewed as depending on an implicit clock (which is not necessarily the same as the system clock): at each tick of the clock, the contents are updated, and the output follows the input, but is delayed by one tick.

The incrementer, multiplexer, and register are all primitive modules. That is, their description is assumed to accurately characterize their behavior. Compound modules now can be defined, declaring their parts as instances of the primitives, and specifying the internal interconnections.

We can, for example, specify a loadable counter, shown in Figure 1, as follows:

```
% Definition of module type Counter

module(counter).

port(counter,in(Acounter),input,integer).
port(counter,ctrl(Acounter),input,boole).
port(counter,out(Acounter),output,integer).
part(counter,muxA(Acounter),mux).
part(counter,regA(Acounter),reg).
part(counter,incA(Acounter),inc).

connected(counter,ctrl(Acounter),switch(muxA(Acounter))).
connected(counter,in(Acounter),in1(muxA(Acounter))).
connected(counter,out(muxA(Acounter)),in(regA(Acounter))).
connected(counter,out(regA(Acounter)),in(incA(Acounter))).
connected(counter,out(incA(Acounter)),in0(muxA(Acounter)))
connected(counter,out(regA(Acounter)),out(Acounter)).


% Behavior specification:

state(counter,count(Acounter),integer).
stateMap(counter,count(Acounter),contents(regA(Acounter)))

outputEqn(counter,out(Acounter) := count(Acounter)).

stateEqn(counter, count(Acounter) :=
  if(ctrl(Acounter),in(Acounter),count(Acounter)+1)).
```

In this definition, 'muxA', 'regA', and 'incA' are functions that select a particular component of an instance of a counter. Thus 'muxA(Acounter)' yields the multiplexer of the counter 'Acounter', and hence 'in0(muxA(Acounter))' yields the current value of the signal at its input 'in0'.

Connections are declared between a source and a destination. A source can be an input to the current module, an output of one of its parts, or a state variable. Similarly, a destination can be an input of a part, an output of the module, or a state variable.

The state variable, 'count(Acounter)', declared in the behavior specification above is for the purpose of describing the counter as a black box. It actually corresponds to the state variable of the register, 'contents(regA(Acounter))', and the renaming is accomplished by the 'stateMap' declaration.

The description language supports several additional useful constructs: constants, parameters, part and port arrays, and bit-wise connections.

Constants can be declared in a similar manner to state variables, specifying their name and type, and also their value. Connections can then be specified from the constants to input ports of parts. This approach has the advantage that constants are prominently visible in a description, and their values are not embedded in expressions, which makes editing much more reliable.

Parameters may be provided to module declarations, so that a single description may cover several distinct types of module. For example, one parameter might indicate the width of inputs and outputs in bits, while another might be the value of a hard-wired constant.

Arrays of parts, ports, state variables, and constants are easily represented, adding a subscript to the arguments of selector functions. For example 'in(AModule,I)' refers to the Ith 'in' port of module 'AModule'.

Bit-wise connections are specified by use of a function 'bit', as in 'bit(I,x)', which represents the selection of the Ith bit of a signal, 'x'.

Parameters, constants, arrays, and bit-wise connections can be best understood by considering an example. In the following, an 'adder' is constructed from a collection of 'fullAdder's, each of which has inputs 'inx', 'iny', and 'cin' (the carry in), and outputs 'sum' and 'carry'.

```
% Definition of an (N+1)-bit adder in terms of fullAdders

module(adder(N)).
```

```
port(adder(N),inx(Adder), input,integer(N)).
port(adder(N),iny(Adder), input,integer(N)).
port(adder(N),out(Adder),output,integer(N1))
    :- N1 is N+1.

constant(adder(N),carryin(Adder),0,boole).

part(adder(N),fa(Adder,I),fullAdder) % an array of parts
    :- range(0,I,N).

connected(adder(N),bit(I,inx(Adder)),inx(fa(Adder,I)))
    :- range(0,I,N).
connected(adder(N),bit(I,iny(Adder)),iny(fa(Adder,I)))
    :- range(0,I,N).
connected(adder(N),carryin(Adder),cin(fa(Adder,0))).
connected(adder(N),carry(fa(Adder,I)),cin(fa(Adder,I1)))
    :- range(1,I1,N), I is I1-1.
connected(adder(N),sum(fa(Adder,I)),bit(I,out(Adder)))
    :- range(0,I,N).
connected(adder(N),carry(fa(Adder,N)),bit(N1,out(Adder)))
    :- N1 is N+1.

% Behavior specification:

outputEqn(adder(N),out(Adder) := inx(Adder)+iny(Adder)).
```

In this example, 'N' is a parameter specifying the number of bits in the inputs to the adder (i.e. the most significant bit represents 2^N). The adder is composed of N+1 'fullAdder's, specified in the 'part' declaration, and the construct 'range(0,I,N)' means simply that 'I' may take any value from '0' to 'N'. Individual bits of the inputs are connected to the 'fullAdder's, and their outputs are bit-wise-connected to the adder output. The equation describing the behavior of the adder is the specification, and is concerned only with input/output behavior, not reflecting any of the internal structure.

To facilitate subsequent computations, module descriptions are converted from the "human-convenient", parametrized notation above into a more explicit form. For example, the adder description might be converted into descriptions of 3- and 5-bit adders, if these were required in the course of verification. The conversion is only done once, when needed, and the results are cached.

§3 Inferring Behavior from Structure

Given the behavior of its parts, and their interconnections, it is possible to infer the behavior of a composite module. For each of its output ports, we can trace back a connection to an output port of some part. That port has an equation describing its output as a function of the inputs (and state) of the part. The inputs can themselves be traced back to outputs of other parts, or to an input of the module, and so on. The state variables of parts are replaced by the equivalent state variables of the module. Thus we can construct an expression that describes the module output as a function of the module inputs and state variables (similar to the specification).

This back-tracing process runs into trouble if there are loops in the interconnection of parts, but only if the loops are not broken by a state variable, as in the counter described above. For such cases, we need to introduce new state variables, but this is not done in the current implementation. (See [Shostak 83] for another approach to designs with loops.) Loops do not in practice seem to cause many problems: the Mead-Conway design methodology advocates breaking all loops with clocked dynamic storage elements, thereby conveniently providing the required state variables [Mead and Conway 80].

In constructing the functional expressions for outputs, a certain amount of checking is performed. It is possible to check that no line is driven by two outputs simultaneously (unless they are tri-state outputs), that every line has an output and an input, and that every output and state variable is accounted for. The signal types of connected outputs and inputs are checked for compatibility in the signal abstraction hierarchy, and an appropriate type conversion is made where appropriate, or the user warned where it is not. A special form of 'type-conversion' occurs when bit-wise connections are made: integer outputs can be embedded in bit-selection functions, and bit inputs can be collected into integers as terms in a power series.

In principle, the given behavior specification for a module also should be checked for errors. A simple type-check on the arguments of sub-expressions could be easily performed, but in practice, this has not yet been implemented.

The behavior descriptions (whether given in the specification, or constructed from the structure) are useful for many purposes. Values can be supplied for input signals and state variables, and output values can be determined by the evaluation and simplification mechanism described later. It is not necessary to supply literal values for all variables: values can be left as variables. Thus we can simplify output expressions for such purposes as optimization, or specialization. We can also perform symbolic simulation, propagating symbols for variables which have definite, but unknown, values. For example, by symbolic simulation we can show that in the multiplexer primitive defined above, when 'switch' is true, the output is 'in1', regardless of the actual value of 'in1'. This approach has the potential for much faster checking of a design by simulation than instantiating values for all variables, as is currently done. In this work, however, the major purpose of constructing behavior descriptions is so that they may be compared with the specified behavior and the design verified.

§4 Verifying the design of a module

When presented with a module to be verified, the system recursively verifies each of its parts, constructing a behavior description from its structure, and comparing it with the specified behavior. The behaviors are compared by considering the outputs, and state variables, in turn. Each output has two expressions that purport to describe its functional behavior. These are equated and an attempt is made to prove that the equation is an identity (i.e. holds for all possible values of inputs and states). The module is correct if this can be done for all outputs and state variables.

In work on verification, Shostak has developed a decision procedure for quantifier-free Presburger arithmetic (no multiplication, except by constants) extended to include uninterpreted functions and predicates [Shostak 79]. Such a decision procedure seems to be applicable in many cases of program verification, and could be a useful component of our design verification system, but has not been implemented.

The current strategy for determining whether an equation is an identity involves a sequence of steps. First, the equation is checked to see whether it is a trivial identity, of the form 'X=X' or 'true', or whether it is a trivial non-identity, such as 'a=b', where 'a' and 'b' are constants.

If the equation is not trivial, the size of the domain space of the equation is then determined by finding the cardinality of the domains of the input and state variables occurring in it. If the space size is sufficiently small (less than about 40 combinations of variable values), the equation is tested by enumeration: each combination of variable values is generated and substituted into the equation, which is then simplified to 'true' or 'false'. This "brute-force" approach is surprisingly useful. Complex designs are usually composed of many simple pieces that are amenable to proof by enumeration, but which would re-

quire much additional mathematical knowledge and algebraic manipulation to prove by other means.

When an equation is too complex for enumeration, an attempt is made to prove identity by symbolic manipulation. This strategy incorporates several operations: evaluation, simplification, expansion, canonicalization, and case analysis.

Evaluation involves recursively considering subexpressions, such as '1+2', to see whether they can be replaced by constants, such as '3'. Simplification is an extension of evaluation that can replace a subexpression by a simpler one. For example, 'and(true,x)' is simplified to 'x'. Evaluation and simplification are implemented by sets of transformation rules that are recursively applied to expressions.

Expansion occurs when a function is observed on one side only of the equation. The definition of the function is then substituted for its call by another rewriting rule. Bit-wise connection of signals is a special case: the observation of 'bit(i,x)' on one side of the equation can lead to the expansion of x as a power series of its bits on the other side. Bit-expansion is used fairly frequently in the examples tried so far.

Canonicalization attempts to deal with certain combinatorial problems of representation. In particular, it handles associativity and commutativity of functions (such as '+', '×', 'and', and 'or') and their units and zeros. Nested expressions involving an associative function are flattened, units of the function are discarded, and any occurrence of a zero of the function causes the expression to be replaced by the zero. The terms in the flattened expression are then lexically ordered, and the nested expression rebuilt. Other normalizations, such as putting logical expressions into a normal form, or pulling conditional expressions to the outside, are accomplished by the rewrite rules during simplification.

Case analysis is performed when the equation involves conditional expressions, which by this time are outermost in the expression. The analysis descends recursively through the conditionals, substituting truth for the condition in the true-expression, and falsity for it in the false-expression. For example, if the condition is 'x=1' then '1' can be substituted for 'x' in the true-expression, and 'false' for 'x=1' in the false-expression. In its present form, VERIFY does not deal with all the ramifications of the conditional in both branches.

Evaluation, simplification, expansion, canonicalization, and case analysis can be applied repeatedly until they no longer have any effect. (In practice some care has been taken in the implementation to apply them in ways that avoid large amounts of wasted effort.) If the resulting expression is trivial, identity will have been proved or disproved.

If automatic symbolic manipulation does not manage to produce an answer, an interactive mode is entered. It was originally intended that the interactive mode would be the primary means of proof, with the user specifying the strategy (collect terms, substitute, simplify, etc.) and the system executing the tactics and doing the necessary bookkeeping. The interactive facility has not yet been implemented, partly because surprisingly much progress has been made using the automatic strategy described above! Interaction is currently limited to responding to the system's final, desperate question "Is this an identity?" with "yes", "no", or "unknown", although development of an interactive prover is intended in the near future.

§5 Results

VERIFY is composed of about 2400 lines of PROLOG source code, resulting in 31000 words of compiled code. It has been tested on a small number of example cases. VERIFY can readily handle the example of the counter given above. During execution, it prints a large quantity of monitoring information on the terminal, so that the user can watch its recursive descent through a design.

The most ambitious example that has been attempted is the module known as D74, decomposed in Figure 2. The example was taken from Genesereth [Genesereth 82], which addresses the problem of fault diagnosis. However, our version is enormously more detailed. D74 contains, at the top level, three multipliers and two adders. The multipliers are composed of slices, each of which contains a multiplexer, a shifter, and an adder. The adders are built from full-adders, which are built from selectors, which are built from logic gates. The logic gates are themselves described at two levels: an abstract boolean function level, and a level closer to an underlying NMOS transistor model, involving tri-state signals and stored charge. Enhancement and depletion mode transistors, along with joins and ground, are the primitives upon which the entire design rests. The design specification occupies about 430 lines of PROLOG data.

D74 is parametrized in the number of bits in its input data. An instance that has three 8-bit inputs involves 50 different types of module, from 8-bit multipliers, 16-bit adders, 15-bit adders, and so on, down to transistors, with 9 levels of structural hierarchy. There are 23,448 primitive parts, including 14,432 transistors.

Verification of this design took 8 minutes and 15 seconds of cpu time on a DEC 2060 running compiled PROLOG. Of this time, about 2 minutes is spent in pretty-printing expressions for monitoring (which results in about 6000 lines of output).

Slightly larger designs have been verified (over 18000 transistors) but the current implementation begins to run out of space at about this level.

§6 Conclusion

VERIFY is only a first attempt at implementing a design verification system that is founded on a clear mathematical model, and that can deal with designs of an interesting degree of complexity. The present version is tuned in a few places to handle some special cases, and much must be done before it can be made into a real design tool. However, hints of the potential of this approach, and particularly the virtues of structured designs, are already evident.

Work for the immediate future includes trying the system on real production designs, implementing the interactive proof checker, and extending the representations and methodology to handle conveniently such modules as memories. More ambitious examples can then be tackled.

References

[Floyd 67] Floyd, R. W., "Assigning Meanings to Programs," *Proc. Amer. Math. Soc. Symp. in Applied Math.*, 19 (1967), 19–31.

[Foster 81] Foster, M. J., "Syntax-Directed Verification of Circuit Function," in *VLSI Systems and Computations*, H. Kung, B. Sproull, and G. Steele (ed.), Computer Science Press, Carnegie-Mellon University, Pittsburgh, 1981, 203–212.

[Genesereth 82] Genesereth, M. R., "Diagnosis Using Hierarchical Design Models," in *Proc. National Conference on Artificial Intelligence, AAAI-82*, Pittsburgh, August, 1982.

[Gordon 81] Gordon, M., "Two Papers on Modelling and Verifying Hardware," in *Proc. VLSI International Conference*, Edinburgh, Scotland, August, 1981.

[Mead and Conway 80] Mead, C., and L. Conway, *Introduction to VLSI Systems*, Adison-Wesley, Philipines, 1980.

[Schwartz 82] Schwartz, R. L., and Melliar-Smith, P. M., "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant

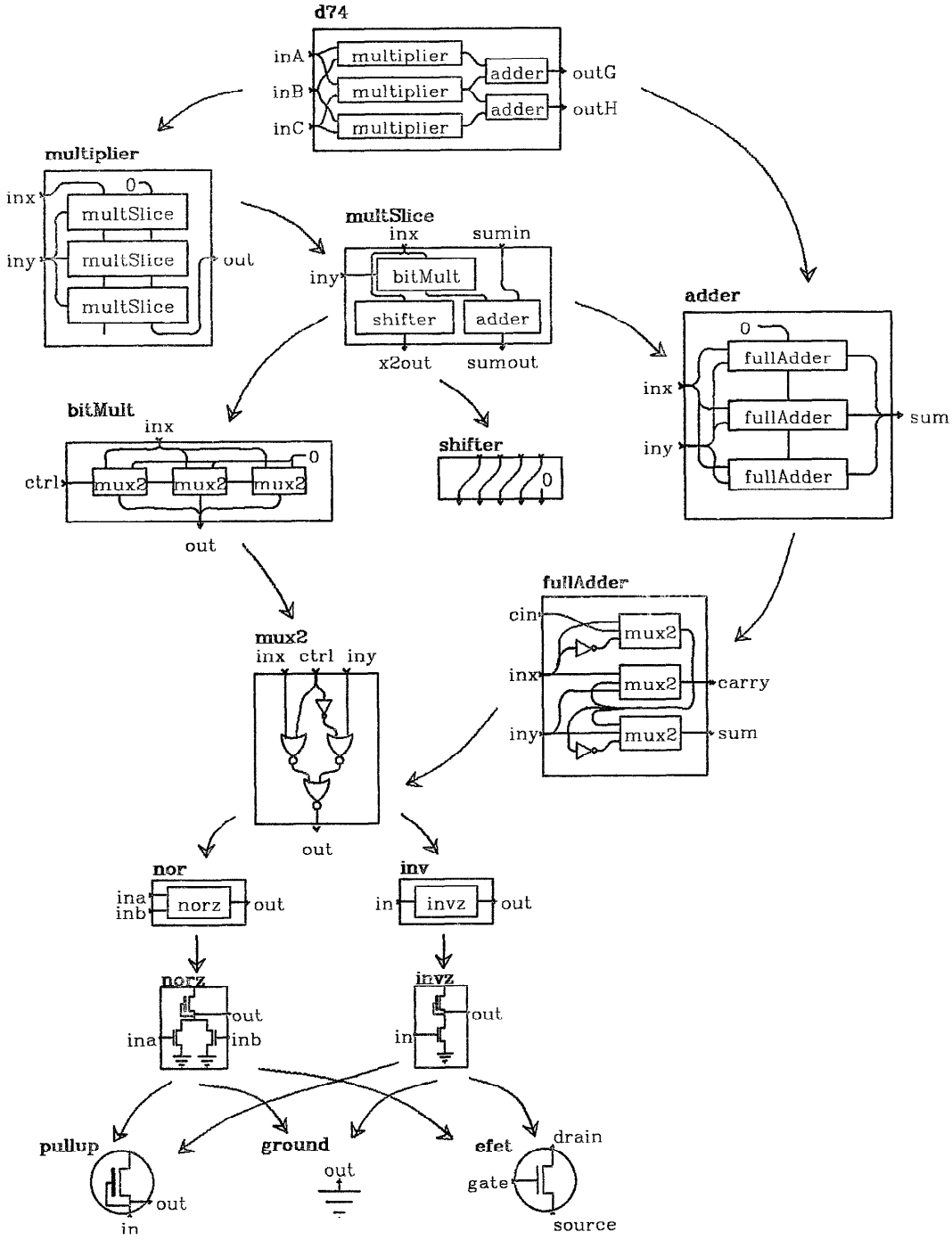

Figure 2. Structural decomposition of D74. Six of the modules have variants differing in the values of their parameters.

Flight Control System," SRI International, Menlo Park, California, TR CSL-133, January, 1982.

[Shostak 79] Shostak, R. E., "A Practical Decision Procedure for Arithmetic with Function Symbols," *Journal of the ACM* **26**, 2 (April, 1979), 351-360.

[Shostak 82] Shostak, R. E., Schwartz, R. L., and Melliar-Smith, P. M., "STP: A Mechanized Logic for Specification and Verification," in *Proc. Sixth Conf. on Automated Deduction*, Courant Institute, New York, June, 1982.

[Shostak 83] Shostak, R. E., "Verification of VLSI Designs," in *Proc. Third Caltech Conf. on VLSI*, Computer Science Press, March 1983.