High-level Symbolic Construction Techniques for High Performance Sequential Synthesis

Andrew Seawright and Forrest Brewer
ECE Dept. University of California, Santa Barbara 93106.

***Abstract* — Techniques for constructing synchronous sequential machines with associated data paths from an input format consisting of high-level non-deterministic productions are described. These construction techniques rely on recent work in symbolic Boolean representation and manipulation to produce an intermediate machine representation that is not impacted by state explosion.**

I. Introduction

In conventional register-transfer-level and high-level design languages, the control structure is represented by conditional branching and procedure constructions which utilize explicitly specified program state variables during execution. However, for many problems, the specification of the machine behavior in this format is unnecessarily lengthy and complex. This is especially true for problems in which the time sequence behavior is complex or the control state space is large or difficult to describe explicitly. Example machines include protocol controllers, communication devices, and computer interface subsystems.

To address these specification problems, a non-deterministic production-based specification for high-level synthesis was introduced in [15] building on efforts in the areas of software engineering [1, 8, 9, 12] and hardware specification [6, 10]. Jackson introduced a grammar-based methodology for specification of software programs and software interfacing between programs. These ideas are manifest in the successful compiler specification and constructions tools "yacc" and "lex". Ullman et al. used regular expression based specifications for hardware compilation of minimal area PLA-based controllers.

Conventional sequential synthesis techniques are usually based on deterministic FSM manipulations. Using these techniques requires conversion from a non-deterministic production model to a deterministic FSM. Since the FSM model is typically a state transition graph, this conversion often leads to an exponential number of deterministic states. In the formulation presented here, the machine is instead, represented using symbolic Boolean Decision Diagrams which can be rapidly manipulated and from which a deterministic machine can be directly derived. These construction techniques rely on the recent work in symbolic Boolean representation and manipulation [2, 3, 5, 13, 14, 16]. It is important to note that the complexity of this derivation and the resulting implementation is not impacted by the potentially exponential growth of the deterministic state space. This is due to the fact that exponential growth of the deterministic state space does not necessarily imply the exponential growth of a circuit implementation.

In the work described herein, the high-level specification form has been expanded with the addition of productions which can represent arbitrary Boolean functions of the interface signals and additional Boolean production connective operators. These additions greatly expand the expressive power of the language and allows an efficient reformulation of the machine construction process.

II. The Production Specification

The high-level specification form described here, specifies the control structure of a synchronous design using a hierarchical set of *productions*. These productions describe the desired "protocol" of the design at the high-level, and "recognition" of these productions effect the design's behavior. A production is a named composition of symbols, operators, and action clauses. In this production specification, there are two kind of productions, those specifying sequential behaviors, and those specifying combinational Boolean functions. The symbols in a *sequential production* are either references to other sequential productions or are *tokens*. A token is a reference to a *Boolean production* in a sequential production. The symbols in a Boolean production are either references to other Boolean productions or atomic symbols which represent the input interface signals or are other language defined symbols.

Composition operators are used to compose the productions. They build more abstract or complex productions from simpler productions. The composition operators are similarly grouped into sequential and combinational types for use in the two kinds of productions. The sequential operators include the classical regular expression [1, 7, 11] operators: *concatenation*, *or*, and *closure* ("`,`", "`||`", "`*`"), generalizations such as the *sequential-and* operator "`&&`", useful for specifying synchronization, the *sequential-not* operator "`!`", and many specialized operators mainly used for specifying exception behaviors. The combinational operators include the usual Boolean operators represented by: "`&`", "`|`", and "`~`".

Recursion of productions is not allowed, since the intent is to produce a specification representable by a finite state machine. Although simple tail recursion doesn't jeopardize the finite machine requirement, it is still not allowed. This is not a problem, since, tail recursive behavior can be concisely described using the closure operator.

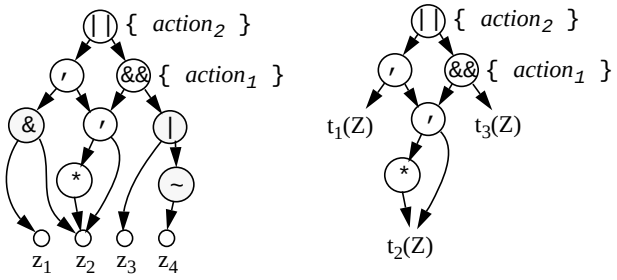
A token is "accepted" or "recognized", if its Boolean function is satisfied in the cycle the token is referenced through execution of the productions. A production is accepted when the time sequence of behaviors dictated by its *composition* is satisfied. Symbols, compositions, and productions are annotated with *action clauses* or *actions* for short. An action is a specified data-flow behavior which is executed when its antecedent symbol, composition, or production is recognized. In general, any number of productions may be active or simultaneously in acceptance, reflected by the composition of the specification. To make these ideas clear, consider the set of five productions shown below:

```
p1 -> p2 || p3; { action2 }
p2 ->(z1 & z2), p4
p3 -> p4 && p5; { action1 }
p4 -> z2+;
p5 -> z3 | ~z4;
```

In this example, the symbols, `z1`, `z2`, `z3`, and `z4` represent interface signals. Productions `p1`, `p2`, `p3`, `p4` are all sequential productions because they contain sequential operators or references to sequential productions. Production `p5` is a Boolean production, "`(z1`

`&z2)`" is a Boolean composition, and "`z2+`" is a sequential composition using the *sequential one-or-more* operator "+" and is equivalent to the composition "`z2*,z2`". Two actions denoted by curly brace clauses are attached to two of the productions.

When the production specification is compiled, a directed acyclic graph (DAG) structure representing the collapsed production structure is constructed. This DAG is called *the production DAG*. Each node in the DAG represents a composition operator. The production DAG representation for the example productions is illustrated in the left-hand figure below. Here, the sequential composition operator nodes



are represented by unshaded nodes, while the Boolean composition nodes are shaded. It is illegal for sequential productions or compositions to be used in Boolean compositions. Thus, the Boolean (shaded) nodes must be only present in the lower portion of the DAG structure.

Each sub-DAG represents a sub-machine. Actions clauses from the productions are associated with respective sequential composition nodes. An action clause associated with the top node in a sub-DAG is executed after initiation and recognition of the respective sub-machine. Often, several actions will be required to execute in the same cycle due to the behavior specified in the productions. The conceptual execution order of the actions within the cycle is *important.* For example, if the two actions "*{ x := 0; }*" and "*{ x := x + 1; }*" were to execute in the same cycle their execution ordering matters. *Action precedence* determines the conceptual ordering of actions executing within the same clock cycle and is determined from the production DAG structure which enforces a *partial ordering* on the actions. The precedence stems from the refinement of simultaneously accepting productions from primitive to abstract. For example, in the production DAG above, $action_1 < action_2$.

The right-hand figure shows the structure with the Boolean nodes removed. Instead, the arcs from the sequential nodes to Boolean composition nodes are replaced by arcs to the representative Boolean functions. For example, $t_1(Z) = z_1 \wedge z_2$, and $t_2(Z) = z_2$. These references to the Boolean functions are token references. Practically speaking, these arcs are implemented as ROBDD [2] node pointers representing the token functions. A machine representation can be constructed from this production DAG structure.

The next section describes the intermediate machine representation. An ensuing section describes the process which constructs the intermediate form from the production DAG structure.

III. Intermediate Machine Representation

The intermediate machine representation consists of two parts, a state transition function and an output function for the machine. In what follows, B represents the set {0, 1}. The transition function Δ is a function mapping: $B^n \times B^k \rightarrow B^n$. This mapping is written:

Δ: $\{(x_1, x_2, x_3, \ldots, x_n)\} \times \{(z_1, z_2, z_3, \ldots, z_k)\} \rightarrow \{(y_1, y_2, y_3, \ldots, y_n)\}$,

where X, Y, and Z are Boolean vectors. X represents the present state of the machine, Z represents the input interface signals, and Y the next state of the machine. The transition function Δ represents a deterministic state transition function. Each x_i, however, represents a particular control point in the machine. The control points represent the activity of non-deterministic states in the machine, for example, many x_i's may be simultaneously active. The function $y_i = f_i(X, Z)$ represents excitation for the next state of control point x_i. This representation of the transition function allows two views: Viewed as a whole, Δ represents the transition function of a single deterministic FSM, while each $f_i(X, Z)$ in Δ represents the excitation of an individual non-deterministic control point.

The output function Λ maps $B^n \rightarrow U$. In this mapping, written:

Λ: $\{(x_1, x_2, x_3, \ldots, x_n)\} \rightarrow \{ u_1, u_2, u_3, \ldots, u_l \}$.

X represents the present state of the machine, and each $u_i \in U$ represents, symbolically, a *sequence* of actions to be executed within a single clock cycle. An *equivalent* output function Λ': $B^n \rightarrow B^m$ is defined, mapping:

Λ': $\{(x_1, x_2, x_3, \ldots, x_n)\} \rightarrow \{(a_1, a_2, a_3, \ldots, a_m)\}$.

Again, X is the present state, however, the elements $a_i \in A$ represent each of the individual actions. Each action is "activated" by the condition $a_i = c_i(X)$. Because many actions may be activated simultaneously, action precedence determined by the production structure enforces the execution sequence. The ordering of the a_i's in the vector A satisfy these partial order action relations implied by the production DAG. The functions Λ and Λ' are related as follows: each u_i symbolically represents a particular action sequence which is an element of the range of Λ'. We focus on the function Λ', Λ was described simply to introduce Λ'.

The output function described previously is the Moore machine form of the intermediate machine representation. The Mealy machine representation form, as expected, is: $\Lambda : B^n \times B^k \rightarrow U$, Λ': $B^n \times B^k \rightarrow B^m$, and with the individual action conditions a function of X and Z, e.g. $c_i(X, Z)$.

IV. Construction of the Intermediate Machine

This section describes a procedure to construct the intermediate machine form introduced in the previous section. The construction is a recursive process which builds, from the production DAG structure, either a Moore or Mealy intermediate machine representation. This recursive construction procedure applies a particular construction rule at each composition node of the DAG, based on the node's type. The process proceeds in a fashion similar to Thompson's construction [1, 7] for creating a NFA state graph from a regular expression. Here, however, we are symbolically constructing the intermediate machine representation. Recall, the intermediate machine representation contains the excitation functions for individual non-deterministic control points. In this representation, there is no analog of ε transitions which are generally present in NFA machines. The construction necessarily produces an ε-free implementation directly. Furthermore, the construction process allows generalization of the regular expression operator construction rules to other very useful composition operators.

The construction is performed in a recursive bottom-up fashion over the production DAG. The structure of the recursive routine **Build()** which implements construction process is illustrated in the pseudocode below. At each level of the recursion, the routine is passed a pointer to a node of the production DAG and a Boolean function

representing a partial excitation function f(X) passed from other recursion levels. The routine returns a Boolean function h(X). Depending on the type of node, different operations are performed. At leaf nodes, control points are allocated and their excitation functions are set. At intermediate nodes, left and right sub-machines are composed via operations on the passed and returned functions. The construction process is initiated by allocating an initial control point $\mathbf{x_1}$ and calling: **Build(n=*top-level-node*, f(X)=$\mathbf{x_1}$)**.

```
Build (node: *n, Boolean function: f(X)) {
    if (n is a terminal function t_j(Z)) {
        create new control point x_i;
        set h(X) = f_i(X, Z) = y_i = f(X) ∧ t_j(Z);
    } else if (n is "concatenation" node) {
        g(X) = Build(node->left, f(X));
        h(X) = Build(node->right, g(X));
    } else if (n is "sequential and" node) {
        g(X) = Build(n->left, f(X));
        h(X) = Build(n->right, f(X));
        h(X) = g(X) ∧ h(X);
    } else if (n is "sequential not" node) {
        g(X) = Build(n->left, f(X));
        h(X) = ¬ g(X);
    } else if (...) {
        ... other cases ...
    }
    if (action a_k attached to node)
        set c_k(X) = c_k(X) ∨ h(X);
    return h(X);
}
```

The time complexity of this algorithm depends on the representation used for Boolean functions. If we assume constant time for two operand Boolean operations, which is certainly possible for a factored representation, then, for a DAG representing a regular expression, the time complexity of this construction is linear in the size of the regular expression. Using BDDs, a pseudo-linear algorithm results as the growth of the variable support of the BDDs is typically slow.

The construction for the closure operator is more complex. A temporary variable x_{tmp} is allocated and passed down for construction of the operand sub-machine. This is done because the excitation of the sub-machine depends on the returned function g(X), which is unknown until the machine is constructed. On the return of g(X), after the sub-machine is constructed, the function $h(X) = f(X) \vee g(X)$ is calculated. At this point, this function may be substituted for x_{tmp} in all functions in which x_{tmp} appears in the structure of the sub-machine. This completes the closure construction and removes the extraneous variable x_{tmp}. These substitutions are nicely performed by composing BDD functions e.g. $f(x=g()) = ite(g(), f_x, f_{\bar{x}})$ [2]. Note, a unique x_{tmp} variable must be used for each simultaneously open closure in the construction process.

Special sequential operators called *exception operators* are implemented. In an exception construction, a handler machine M_h is initiated when its associated sub-machine *M*, once initiated, will enter a state in the next cycle from which it can *never* accept. Note this is a different notion than in the *sequential not* operator where not accepting includes both the cases of "active but not presently accepting" and "will never accept". The function e(X, Z) represents the excitation that triggers M_h. Consider the following equation for $e_x(X, Z)$, which is used to calculate e(X, Z):

$$e_x(X, Z) = \overline{g(X)} \wedge \prod_{f_i \in M} \overline{f_i(X, Z)} \qquad \text{(EQ 1)}$$

This equation describes the conditions in which *M* is not in a state of recognition and in the next cycle will contain no active control points. However, this relation is not enough to describe e(X) as we need knowledge that *M* was first initiated. This information can be computed as summation of the present control points in *M* and *M*'s excitation. However, this logic can be substantially reduced if an extra control point is allocated to store this information. Let x_h represent this control point. Then,

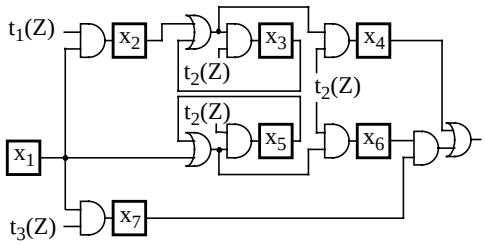
$$e(X, Z) = (f(X) \vee x_h) \wedge e_x(X, Z) \qquad \text{(EQ 2)}$$

The excitation of x_h is $f_h(X, Z)$ and can be computed as follows:

$$f_h(X, Z) = (f(X) \vee x_h) \wedge \overline{e_x(X, Z)} \wedge \overline{h(X)} \qquad \text{(EQ 3)}$$

Other related exception operators are implemented, as well, in a similar fashion. One further point concerning the exception operators, since the exception operators operate on a general sub-machine, they may be nested hierarchically.

To clarify the construction process, a circuit implementation of the constructed machine for the example is illustrated below:



In the pseudocode, the action execution conditions $c_K(X)$'s are created using h(X) at the nodes where actions are attached. This creates the Moore output function Λ'. The Mealy output function may be created from the Moore output function after the construction is complete. This is done by substituting $f_i(X,Z)$ for all x_i's in $c_k(X)$ forming $c_k'(X, Z)$. Again, substitutions can be easily done by composing BDD functions. Note, this conversion from Moore to Mealy does not create an identically equivalent Mealy machine — the results of action executions will lag by a cycle in the Moore machine vs. the Mealy machine. The designer is aware of this in choosing the machine model and will structure the actions accordingly. The following are the action execution conditions for the example:

- Moore: $c_1(X) = x_7x_6$, $c_2(X) = x_7x_6 + x_4$.
- Mealy: $c_1(X, Z) = t_2t_3x_1(x_5 + x_1)$,
  $c_2(X, Z) = t_2t_3x_1(x_5 + x_1) + t_2(x_3 + x_2)$.

The constructed intermediate machine is intended to be a base representation to which further synthesis tasks and manipulations are applied. It is a convenient form, because both classical FSM techniques as well as more recent symbolic methods may be applied. A further useful property is that the elements of intermediate machine can be directly linked, during the construction, to the high-level operators that

created them. This is useful for high-level optimization, debugging, and design information tracking in a synthesis system.

V. Implementation

A production compiler using these construction techniques was implemented. This compiler is part of a high-level synthesis system coded in approximately 8000 lines of C++. The output of the compiler is VHDL code describing the synthesized machine architecture. The VHDL is composed of processes that describe the logic structure of the machine and processes that implement the register transfers and data-path logic required by the actions. The structure of the VHDL action processes also satisfy the partial ordering required by action precedence. The VHDL output is tailored for further hardware synthesis by specific adherence to an appropriate synthesis policy.

The tool is able to perform reachable state analysis on the compiled intermediate machines. This type of analysis is an example of the symbolic techniques possible on the intermediate form. The reachable state analysis is based on the work in [5, 13, 16]. *Reachable state analysis is not required for the synthesis of the intermediate machine,* but it is an important task, useful in several ways. Knowledge of the set of reachable states aids in further RTL optimizations, particularly, determining action conflict resolution. This information may be used to simplify elements of the machine structure. For example, simplifying (EQ 1) using sequential don't care information. Finally, it is useful for design analysis and providing feedback to the designer of possible problems.

The tool uses BDDs for all Boolean and symbolic manipulations. The BDD nodes are allocated as the construction proceeds. It should be pointed out that the construction process naturally develops a reasonable heuristic variable ordering based on circuit topology arguments [14]. In addition, the different kinds of BDD variables are divided into classes and are interleaved. The following three-way ordering is used: $z_1 < x_1 < q_1 < z_2 < x_2 < q_2 < z_3 < x_3 < q_3 < ...$ The q_i's represent an additional set of state variables used by the reachable state analysis. The reachable states are computed based on implicit enumeration techniques. Specifically, the procedure is based on the heuristics in [16]. The observation that the variable support of each $f_i(X, Z)$ is generally localized in a range close to and prior to i allows further tailoring of the heuristics to this application.

VI. Experiments

Several example designs were compiled and studied using the tool. The characteristics of these designs are tabulated in Table I. The **mouse** examples are quadrature decoder machines used in computer pointing devices. These designs continuously update a 1-dimensional position based on the quadrature encoding of signals from external motion sensors. Both versions recognize the same quadrature signals, however, the difference is that the **mouse(b)** design places more constraints on the input waveforms than the **mouse(a)** design.

The **xymouse** designs are 2-dimensional versions of the respective 1-D mouse decoder examples and are specified as a single set of productions using the expressive power of the Boolean representation in the language. Using the earlier version of language, described in [15], which only modeled tokens as an enumerated set, the **xymouse** would be impossible to specify as concisely in a single set of productions.

count0 counts sequential zero's in a valid input frame format. This example is based on the procedural design in [4]. **qr42** is an asynchronous handshake conversion protocol implemented as a sequential machine. The machine connects two devices together, one side operating with two-phase (non-return-to-zero) signaling and the other with four-phase (return-to-zero) signaling. This machine makes use of the synchronization (sequential and) operator.

The **i8251ar** example implements the asynchronous receiver portion of the i8251. This example makes use of Boolean *qualification operators*. The **midi** example is a machine that forms a MIDI interface controller which interprets the MIDI music protocol for a digital synthesizer chip controller. This design included an exception operator to reset the machine in case of invalid input. **mismatch** is the pathological regular expression introduced in [10] which detects mismatches between the first 8 symbols and the last 8 symbols of a string. This example is expected to produce very large numbers of deterministic states.

VII. Results and Conclusions

The results for compiling the example designs to the intermediate machine form and generating direct VHDL implementations from the intermediate machine is shown in the second part of the table. The variable support of the transition and output functions is also equivalent to their depth, in these designs. The number of ite() calls represents the total number of calls to this fundamental BDD function and indicates the relative complexity of the constructions. The generated designs were simulated with VHDL simulator.

The third portion of the table shows the results for further hardware synthesis of the output VHDL descriptions of the designs. Gate level circuit implementations of the intermediate machine representation were synthesized using the Synopsys® VHDL and logic synthesis tools. In this process, *no additional sequential optimizations such as state-assignment, re-timing, or re-encoding were invoked*. Only the logic synthesis and data-path allocations were performed. In a few cases, boundary control point registers were eliminated due to lack of fan-out. The logic synthesis was directed to use LSI 10k gate array library cells and to optimize for speed. The relative area, total number of cells, and number flip-flops includes the logic for both the control and the data-path portions of the designs.

The final portion of the table contains the results of the optional reachable state analysis to better illustrate the designs. The reachable states represent the total number of deterministic states reachable from the start state in the intermediate machine. The diameter denotes the path length to the furthest reachable states from the initial state. The ite numbers in this table include both the machine synthesis as well as the reachable state analysis.

A few conclusions can be drawn from the results. In the **mouse** and **xymouse** machines, the number of productions and control points roughly doubles while the state space of the machine is squared. It is clear that the construction complexity is not proportional to the growth of the machine's state space as might be expected from conventional algorithms. The speed of the two designs (which includes the data-path delay as well as the control delay) is nearly the same. The **i8251ar** design had a small number of control points but a relatively large number of states due its numerous modes. The final design is shown in figure 1. Finally, the **midi** design was much more complicated in its behavior and included an exception handling routine so that any valid data imbedded in arbitrary invalid data would be correctly interpreted. The exception operator lead to the larger variable support numbers and relatively slower cycle time.

It is of interest to note the relatively high performance of the designs derived directly from the intermediate form. These designs have many more registers than conventional designs but generally have very simple excitation logic between the control points. This is due to the constructive synthesis of the machine from the high-level specification. In effect, the control points provide a set of signals from which their excitation functions can be derived with very small literal support. In future work, optimization of the intermediate machine to reduce the number of registers under performance constraints will be studied.

The authors thank Emil Girczyc and Margaret Marek-Sadowska for helpful suggestions and discussion. This research was made possible through the generous support of Synopsys Inc. and the California MICRO program — #92-019.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*. Reading: Addison-Wesley 1988.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package," 27th DAC, pp. 40-45, June 1990.

[3] R. E. Bryant, "Graph Based Algorithms for Boolean Function Manipulation," IEEE T-CAD, pp. 677-691, Aug. 1986.

[4] S. Carlson, *Introduction to HDL-Based Design Using VHDL*. Mountain View: Synopsys, 1990.

[5] O. Coudert, J. C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits," ICCAD-90, pp. 126-129, Nov. 1990.

[6] R. W. Floyd, and J. D. Ullman, "The Compilation of Regular Expressions into Integrated Circuits," Jo. ACM. 29:2, 1982.

[7] J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Reading: Addison Wesley, 1986.

[8] M. A. Jackson, "Constructive Methods of Program Design," *Lecture Notes in Computer Science*, Vol. 44, Springer-Verlag, pp. 236-262, 1976.

[9] S. C. Johnson, "Yacc: Yet Another Compiler Compiler," Computing Science Tech. Rep. 32, AT&T Bell Labs, Murray Hill, NJ 1975.

[10] A. R. Karlin, H. W. Trickey, and J. D. Ullman, "Experience with a Regular Expression Compiler," ICCD, pp. 656-665, 1983.

[11] Z. Kohavi, *Switching and Finite Automata*. New York: McGraw-Hill, 1978.

[12] M. E. Lesk, "Lex -A Lexical Analyzer Generator," Computing Science Tech. Rep. 39, AT&T Bell Labs, Murray Hill, NJ 1975.

[13] B. Lin, *Synthesis of VLSI Designs with Symbolic Techniques*, Ph. D. Thesis, Univ. of Calif., Berkeley, UCB/ERL M91/105, Nov. 1991.

[14] S. Malik, A. R. Wang, R. K. Brayton, A. Sangiovanni-Vincentelli, "Logic Verification Using BDDs in a Logic Synthesis Environment," ICCAD-88, pp. 6-9 Nov. 1988.

[15] A. Seawright, F. Brewer, "Synthesis from Production-Based Specifications," 29th DAC, pp. 194-199, June 1992.

[16] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDD's," ICCAD-90 pp. 130-133, Nov. 1990.

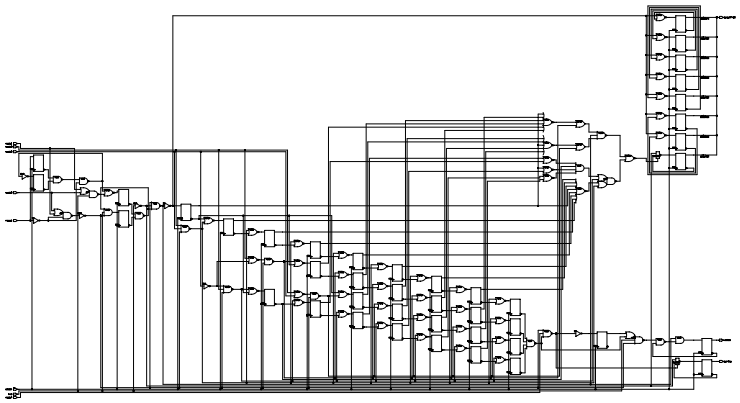


Figure 1.

Table I	design characteristics				**intermediate machine synthesis**					direct circuit synthesis				optional reachable state analysis			
example	#P	#A	#in	#out	**POC**	Δ_{sup}	Λ'_{sup}	**time**	**ite calls**	t_{cyc}	area	#cells	#FF	RS	D	RST	ite calls
mouse(a)	5	2	4	8	8	3/4	4/4	0.1	285	7.88	299	130	14	8	2	0.5	10,211
mouse(b)	8	2	4	8	14	3/4	4/4	0.1	660	6.88	334	132	20	14	4	1.5	33,854
xymouse(a)	8	4	6	16	14	3/4	4/4	0.1	892	8.91	465	198	26	50	2	3.4	79,763
xymouse(b)	15	4	6	16	27	3/4	4/4	0.2	2292	7.69	634	258	39	170	4	17.9	444,780
count0	6	3	3	4	10	3/5	4/5	0.1	500	5.47	170	62	13	8	4	0.6	13,160
qr42	4	3	4	2	21	3/6	8/10	0.3	3349	6.09	251	90	23	62	12	6.2	146,568
i8251ar	14	4	8	10	33	4/8	8/12	0.5	6309	6.87	407	115	42	814	12	94.7	2.20e6
midi	29	12	3	16	182	14/166	10/27	12.7	224,707	9.22	2223	683	210	310	40	3363	6.84e7
mismatch	7	1	4	1	74	2/4	5/5	0.5	6419	3.94	788	229	74	8062	16	8620	2.05e8

Legend: #P - number of productions, #A - number of actions, #in - number of inputs including clock & reset, #out - number of outputs, POC - # of control points, Δ_{sup} - transition function variable support avg./max., Λ'_{sup} - output function variable support, t_{cyc} - minimum cycle time nS, area - total relative area (LSI 10k), #cells - number of cells (LSI 10k), #FF - DP & control, RS - number of reachable states, D - diameter, RST - time to compute RS — Times are CPU seconds for Sparcstation® 1.