Symbolic Modeling and Evaluation of Data Paths*

Chuck Monahan Forrest Brewer

Department of Electrical and Computer Engineering
University of California, Santa Barbara, U.S.A.

Abstract

We present an automata model which concisely captures the constraints imposed by a data-path, such as bus hazards, register constraints, and control encoding limitations. A set of uniform base components for depicting general data-paths and techniques for systematic translation of such depictions into Boolean functions are described. Finally, this model is expanded to represent the limitations of generating as well as moving operands by incorporating data-flow graphs. The benefits of this representation are demonstrated by modeling a commercial DSP microprocessor.

1. Introduction

The goal of this work is to provide an accurate and concise executable representation of a data-path for synthesis and scheduling applications. Although extensive work exists in behavioral synthesis, relatively little work exists for resynthesizing designs where large portions are already constructed and an upgrade or engineering change is required. These tasks require the ability to rapidly determine and exercise the capabilities of existing designs. For complex data-path networks, efficient exploitation of the resources poses many challenges. In particular, assignment of operands into memory elements and the scheduling of operations may have greater dependence on the limitations of the data-path interconnection network than on the algorithm's critical path, even when resource limits are included.

* This work has been supported in part by UC MICRO 94-024 and with the generous support of Mentor Graphics Corp.

Conventional data-path models abstract the interconnection and locality constraints into simple bounds based on the number of resources (Fig. 1a). These bounds aid the synthesis of an unspecified data-path. However, such bounds fail to represent the tightly constrained connections and local storage for a design with an assigned interconnection network (Fig. 1b). Moreover, it is often the case that adopting pre-existing designs is more economical than starting anew, especially when such designs have been verified for timing or are completed layouts. Furthermore, changing design requirements might require rescheduling or rebinding, yet still must make efficient use of the existing resources.

Fig. 2 exposes a vulnerability of the conventional data-path model. The figure lists a schedule for the small data-flow on the given data-path. Using conventional constraints, ASAP reports that the data-flow should take four cycles. In fact, the feasible schedule takes over twice as long. The difference results from two main factors: (1) the schedule requires time to move operands to the proper place, and (2) there are insufficient resources to store all of the operands after generating operand *o2*. Although the conventional register constraint of four operands is never violated, the operand *o1* must be recomputed since the data-path can not simultaneously use and overwrite operand *a2*. Clearly, limiting the avail-

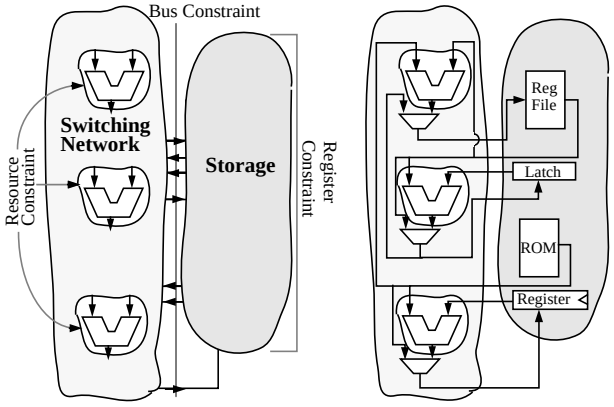
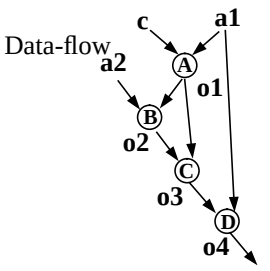


Figure 1. Data-path models



Schedule

State	ALU	ACC	Reg
S0			{a_1, a_2, c}
S1		a_1	{a_1, a_2, c}
S2	A	o_1	{a_1, a_2, c}
S3	B	o_2	{a_1, c}
S4		o_2	{a_1, o_2, c}
S5		a_1	{a_1, o_2, c}
S6	A	o_1	{a_1, o_2, c}
S7	C	o_3	{a_1, c}
S8	D	o_4	{c}
S9			{o_4, c}

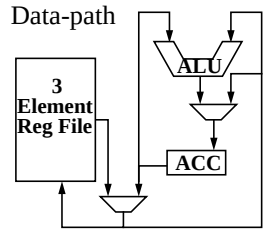


Figure 2. scheduling example

ability of operands compromises fundamental scheduling analysis techniques.

This paper presents an automata based data-path model which captures the constraints required to correctly model temporal behavior. The model restricts communications to those that are simultaneously feasible on the defined interconnection network. It models the behavior and exclusive use of function units, as well as the behavior of storage elements, including the limited capacity of register files. Finally, it enforces the control encoding restrictions on the data-path switching, storage and function units. We present techniques for systematically constructing this model as a Boolean symbolic representation.

First, we describe in Section 3 a uniform set of components from which formal models of behavior can be constructed. The subsequent section presents techniques for generating these models. Initially, Section 4.1 describes methods for generating the constraints of a general switching network. From here, techniques for modeling the restricted availability of an operand and for transforming them into a general transform are specified. Finally, in Section 5 we outline a number of applications employing this model including results for representative algorithms constrained by the TMS32010 data-path.

2. Previous Work

Previous efforts in data-path synthesis used models that can be divided into two major types: In the first type, a register and multiplexer bus transaction model is derived for the particular communications of the designs [5][10][11][12]. This model is typically represented as a connection graph and conventional graph search and matching techniques are applied. More recent models accommodate register files but restrict the connectivity [15]. The second type uses a pre-defined data-path and generates microcode or control for the structure [4][9]. In these systems, the network is specifically designed to simplify assignment of communications. Finally, current research in formal system verification uses similar concepts [1][3][6]. In contrast to formal verification modeling, our model is intended for synthesis use. Thus, we are free to introduce heuristics which do not seriously detract from the power of the system but greatly enhance the speed.

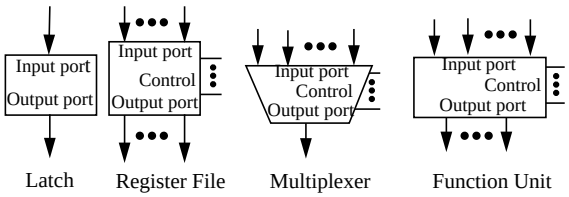


Figure 3. Base component set.

3. Data-Path Model

We model a data-path as a network of memory elements, switching logic, and combinational logic connected by a set of wires. Switching logic, used to conditionally transfer existing operands to different wires, is distinguished from combinational logic which creates new operands. All activities of a data-path are determined by its set of control lines during each clock cycle. Furthermore, the control is currently modeled under the assumption that the data-path uses a single-phase clocking structure. Fig. 3 lists the basic blocks of the model. Memory elements are represented by either latches or register files, switching elements are modeled as multiplexers, and combinational logic elements are referred to as function units. A wide variety of data-paths can be specified with this basic set of components [8].

Each component connects to the wires via a set of unidirectional ports.[1] No bounds are placed on the number of ports, so function units and register files may have multiple outputs. A few restrictions are placed upon the wire connections. First, each wire of the network must have a single source represented by a specified output port. Also, each output port must connect to a single wire. Although, a wire may fan out to many destinations, each input port must be connected to an individual wire.

Components may have a control field which determines their activity during a clock cycle. As this model is not intended for timing verification, it is assumed that the control signals are well-defined and consistent over the span of a clock cycle. Control fields may have logical constraints needed to model complex interconnect or control word encoding. For example, certain types of complex switching components constrain the values of the control bits to ensure proper operation of the data-path. An exam-

1. Bidirectional ports are modeled by combinations of unidirectional ports, switching elements, and switching control restrictions.

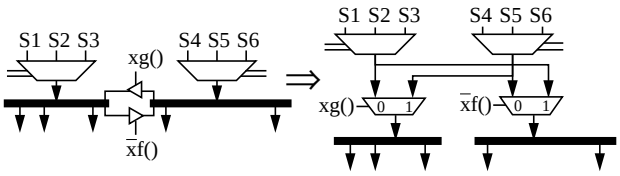


Figure 4. Partitioned bus modeling

ple of this is shown in the partitioned bus depicted in Fig. 4. A similar situation occurs when the control bits of the data-path are heavily encoded. This situation is characterized by a further set of restrictions on the allowed connections and is easily accommodated.

A *communication* consists of transfer of an operand over a connection. A *connection* is a path between two component ports on the data-path. Connections may traverse an arbitrary number of switching elements but not other components. Because of the restrictions placed on the port and wire interfaces, a connection can be equivalently defined as a path from a source wire to a destination wire.

4. Boolean Symbolic Formulation

We wish to represent the behavior of the modeled data-path implicitly as a Boolean symbolic finite state machine. This allows direct use of efficient BDD based representation and traversal algorithms. Isolating the memory elements of the data-path model transforms it into a standard Huffman model of a state machine as depicted in Fig. 5. In our construction, the state is represented by the set of operands stored in each of the data-path's memory components. The next-state function represents the limitations of generating new operands from the function units as well as moving them and existing operands over the network. We restrict the automata to model operations and operands specific to a pre-specified data-flow graph. The next state function is represented by a constructed Boolean relation.

4.1. Wire Functions

A connection between a source and a destination wire is constrained by the control bits of the network's switching elements. Fig. 6 depicts a example switching network and lists the possible connections as well as their control requirements in the accompanying table. In this paper, upper case characters represent wire labels and lower case characters represent control bits. To aid a Boolean formulation, each wire in the network is assigned a unique Boolean encoding.

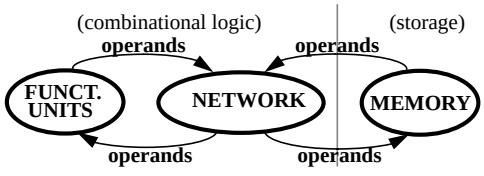
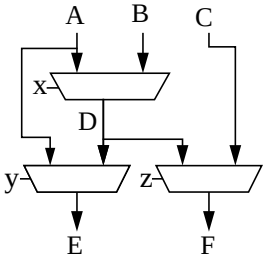


Figure 5. Automata Model Organization



control			*connection*	
x	y	z	E	F
0	0	0	A	A
0	0	1	A	C
0	1	0	A	A
0	1	1	A	C
1	0	0	A	B
1	0	1	A	C
1	1	0	B	B
1	1	1	B	C

Figure 6. An example data-path and the set of connections.

It is useful to denote the set of connections that can be achieved for the destination wire *X*. A *wire function* W(X) is defined as a sum of all such connections. Each connection is represented by the required switching values and labeled with the Boolean encoding of the source wire. Therefore, the function operates over the set of variables as depicted in Eq. 1. To describe the behavior of a switching network, it is sufficient to list a wire function for each of the network's primary outputs (destinations).

$$W(X) = f(\text{switch vars, wire vars}) \quad (1)$$

The construction technique for W(X) for an arbitrary networks relies on the acyclic nature of connections in switching networks.[2] This construction process recursively generates W(X) for each switching component. An initial set of wire functions is constructed to represent the switching function of each multiplexer. This set of wire functions is then combined to represent a set of wire functions for the network of multiplexers. Symbolic substitution of a wire symbol by the associated wire function constructs the set of wire functions systematically. The construction of the wire functions associated with the network shown in Fig. 6 is shown in Fig. 7.

The ensemble behavior of the switching network may be formulated as a single function by combining the set of wire functions of the primary outputs. This function will represent the dependencies that each of the wire functions have upon the common set of switching variables and is generated through a two-step process. First the source encodings are remapped to a unique encoding representing both the source and destination wires. This re-encoding is symbolically depicted by using subscripts to denote the destination wire. At this point, the intersection of these functions generates the ensemble switching function. Fig. 8 displays the construction process for the ensemble network function for the example data-path.

Initial wire function:	**Resulting wire functions:**
$W(D) = \bar{x}A + xB$	$W(E) = \bar{y}A + y(\bar{x}A + xB) = \bar{y}A + y\bar{x}A + yxB$
$W(E) = \bar{y}A + yD$	$W(F) = \bar{z}(\bar{x}A + xB) + zC = \bar{z}\,\bar{x}A + \bar{z}xB + zC$
$W(F) = \bar{z}D + zC$	

Figure 7. Wire function construction.

2. Acyclic switching networks simplify timing and race analysis. Well defined cyclic switching networks can also be represented unless used as a mechanism for storage of state.

Wire functions:

$W(E) = \bar{y}A + y\bar{x}A + yxB$

$W(F) = \bar{z}\,\bar{x}A + \bar{z}xB + zC$

$\Rightarrow$

After remapping:

$\bar{y}A_E + y\bar{x}A_E + yxB_E$

$\bar{z}\,\bar{x}A_F + \bar{z}xB_F + zC_F$

Ensemble network function:

$\bar{z}\,\bar{x}A_EA_F + \bar{z}x\bar{y}A_EB_F + z(\bar{y}+\bar{x})A_EC_F + \bar{z}xyB_EB_F + xyzB_EC_F$

Figure 8. Network function construction.

4.2. Operand Functions

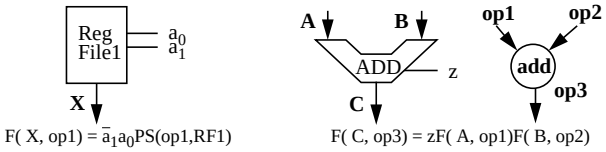
A correct data-path representation models the limitations placed on an operand existence at a specific location. Towards this goal, our automata model constructs operands functions. An *operand function*, F(W, opX), is defined as the sum of communications which supply operand *opX* to wire *W*. Since the availability of an operand is dependent upon the state information, operand functions operate over the set of variables in Eq. 2. Operands are supplied to the source wire of a connection by either a memory element or function unit.

$$F(W, OpX) = f(\text{control vars, present state}) \qquad (2)$$

The availability of an operand from a memory unit is constrained by the operand binding of the present state and by potential control bits. Fig. 9a depicts the constraints for an example memory access including control variables and an abstract present state mapping function, PS(op1, RF1), corresponding the operand *op1* being bound in the device *RegFile1*. A control field is required only for selecting operands from register files. It is possible to constrict register file access using network constraints defined previously. However, this is not efficient since we do not care in what order the operands are stored in register files. We therefore simply list all values available in the register file. The control variables then select which operand to use instead of which register. We must, however, limit the *number* of such operands to that allowed by the register file.

The existence of an operand on a function unit's output port is constrained by a set of operand functions as well as a set of control bits. Fig. 9b depicts how the data-path and data-flow identify the wires and operands for constructing the set of dependencies. Additionally, the required value of the function unit control lines are specified.

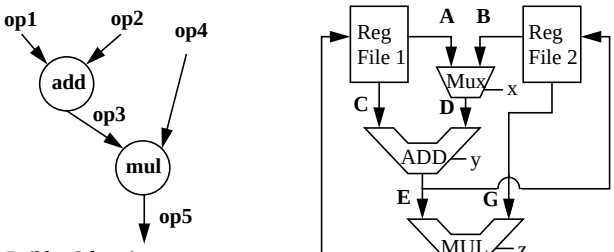
A recursive construction technique is used to build a set of operand functions for an arbitrary data-path and data-flow. An example construction is depicted Fig. 10; for simplicity, this example ignores any permutation of operands to the function units. The construction uses an ordered traversal of the data-flow graph, evaluating each node only after its dependencies are analyzed (hence, op5 follows



$F(X, op1) = \bar{a}_1a_0PS(op1,RF1)$ (a)

$F(C, op3) = zF(A, op1)F(B, op2)$ (b)

Figure 9. Example operand functions



Build op3 functions:

$F(E, op3) = yF(C, op1)F(D, op2)$

$W(D) = \bar{x}A + xB \Rightarrow F(D, op2) = \bar{x}F(A, op2) + xF(B, op2)$

$F(E, op3) = yF(C, op1)(\bar{x}F(A, op2) + xF(B, op2))$

Build op5 functions:

$F(H, op5) = zF(E, op3)F(G, op4)$

$F(H, op5) = zyF(C, op1)(\bar{x}F(A, op2) + xF(B, op2))\,F(G, op4)$

Figure 10. Example data-path and data-flow

op3). For each node, an initial operand function is generated for each function unit capable of producing the corresponding operand. Then operand functions, identified by the wire and operand pairs, are substituted to formulate a function based solely upon memory access and control bits. If an operand function must traverse a switching network as in F(D,op2), the operand function is derived from the network's wire function. Eventually, the memory access functions are replaced by the proper state encoding and control bits.

4.3. Transform Relation

A *transform relation*, describing the relation between present and next states, is systematically built from the set of operand functions in a two step process. First, a set of *operand relations* are constructed from the operand functions. Each operand function pertaining to a memory component's input port identifies the storage of a particular operand in that device. Adding the corresponding next state operand binding to these functions creates a relation between the operand bindings of the present and next state. Fig. 11 shows operand relations derived from the operand functions of Fig. 10 using general next state binding functions.

The second step uses Eq. 3 to combine the operand relations into a general transform relation. The inner term lists the set of operands which may be loaded in each memory device. The product of the resulting terms lists compatible operand relations for the set of memory input ports. Fig. 11 shows resulting transform for the example data-path.

Operand relations:

$R(E, op3) = yF(C, op1)(\bar{x}F(A, op2) + xF(B, op2))NS(op3, RF2)$

$R(H, op5) = zyF(C, op1)(\bar{x}F(A, op2) + xF(B, op2))\,F(G, op4)NS(op5, RF1)$

Transform relations:

$zyF(C, op1)(\bar{x}F(A, op2) + xF(B, op2))\,F(G, op4)NS(op3, RF2)NS(op5, RF1) +$
$\bar{z}yF(C, op1)(\bar{x}F(A, op2) + xF(B, op2))NS(op3, RF2)$

Figure 11. Construction of transform relation.

$$transform = \prod_{i \in memory\_input\_ports} \left(\sum_{j \in operands} R(i,j) \right) \quad (3)$$

A number of additional constraints may be added to the general transform relation. Specifically, control encoding constraints may be formulated as a Boolean function. The product of the transform relation and this Boolean function identifies the set of permissible communications.

5. Applications

The model's capacity for symbolic execution is utilized to construct a number of data-path specific applications. Initially a tool to generate a comprehensive list of the set of simultaneously feasible connections was developed. This tool was then expanded to create a data-path constrained scheduler and a schedule feasibility checker (verifier). Through their symbolic execution of the data-path, both applications preform binding for both function and memory units.

These applications use the transform relations developed in Section 4.3 to define the data-path activity. This technique constructs the set of the next states using an image relation as in [14]. Because the size of the general transform relation can grow very large, we construct the transform dynamically. Since the transform is constructed by relations based on operands, we partition operands into those that are obtainable and those that are unobtainable based on the operands in the current set of states. Constructing the transform relation from the set of active operand relations drastically reduces the relation size.

Fig. 12 depicts how the wire labels, control variables, and state information are combined for our applications. The connection and switching requirements are represented using a common set of switching bits and an encoded source field for each of the *k* destination wires. The present and next state fields list the content of memory components. Each of the *l* latches specifies the operand that it stores, and each of the *m* register files specifies the subset of the *n* operands that are present. The encoding of latch operands require less space since they can utilize binary encoding to identify an operand, while registers use one-hot encoding which aids the representation of arbitrary subsets.

Finally, this model shows promise in the field of engineering change. Specifically, we are interested in analyzing the effects that small changes in the data-flow or data-path has on prescheduled algorithms on existing machines. Changing the timing constraints on an external signal or

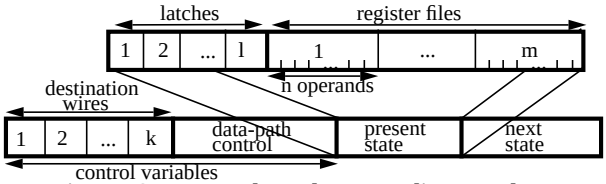


Figure 12. Data-path Boolean encoding template

TABLE 1. Representation size

	# terms	# nodes
TMS32010	80	79
7488	96	48
8085	608	579

deleting a previously existing wire are two examples of changes an engineer might face. Our model can incorporate the existing algorithm, including all binding information and then rapidly perform local rescheduling.

5.1. Feasible-Path Generation

The feasible-path tool was built to construct the wire functions and the ensemble network function of a data-path. The tool utilized the techniques discussed in Section 4.1 to build an OBDD function. The function represents a set of terms, where each term is a unique, legal set of communications.

Table 1 lists the cost for representing the detailed data-path infrastructure of three data-path models. All three models were based on existing commercial data-paths: Texas Instruments TMS32010 DSP processor, Texas Instrument's 74888 and Intel's 8085. Fig. 13 shows the data-path model of the TMS32010 using the base components described in Section 3. The number of legal configurations are listed ("# terms") as well as the number of BDD nodes required to represent the network function ("# nodes"). The smaller size of the TMS32010 and 74888 designs are due to our ability to isolate the data-path from the control-path.

5.2. Scheduling

The scheduler explores the state space linking a specified initial and final state. These states are specified by an initial and final binding of operands to memory elements. Using the automata model, the state space is explored cycle by cycle until a state equivalent to the final state is

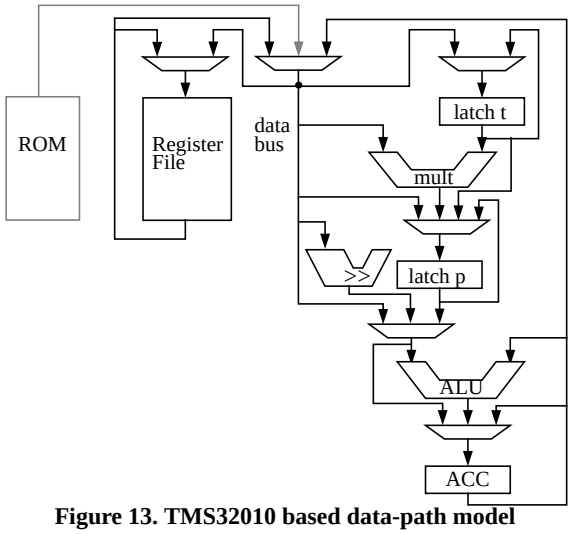


Figure 13. TMS32010 based data-path model

generated. Finally, the history of intermediate states is parsed to identify the set of schedules and memory bindings which generated the solution.

This scheduler explores many options not implemented in traditional techniques. Multiple operation binding is allowed to accommodate problems which require the operand recomputation as seen in Fig. 2. Additionally, operands are permitted multiple memory bindings to increase their availability. Support for multiple register files is inherent in the scheduling model. Finally, an arbitrary mapping between operations and function units is permitted, allowing function units to support multiple operations.

A variety of pruning techniques can be employed to reduce the number of states. For large problems, the number of states can grow rapidly and critically effect CPU requirements of the program. A useful pruning technique is to maintain a list of previously traversed states and only propagate unencountered states on each cycle. Additionally, heuristics can be employed to reduce the number of states. For example, a greedy technique can identify the most active schedules. At each step, only those states which created the maximum number of operands are preserved.

Table 2 lists the results for scheduling the *differential equation* and *elliptic wave filter* benchmarks. The benchmarks were scheduled on two data-paths. The first, based on the TMS32010, is depicted in Fig. 13. Then, an expanded version of the first data-path was constructed to explore the improved performance resulting from an additional global bus. The posted register constraints correspond to the size of the register file. Two values are listed: the first assumes constants are stored in a coefficient ROM and the second, in parentheses, assumes that both operands and constants use the register file. The benchmarks only benefit from a relaxed register constraint on the two-bus data-path. Schedule execution times are indicated in the column "# cycles." The data-path-constrained results generated by our system ("automata") are compared to results of exact scheduling techniques (e.g., [7][13], "exact"). These exact execution times correspond to the same functional unit constraints, but with global bus structure and no latches in data-path. A comparison of these values underscores the dramatic effects that the data-path constraints of a commercial microprocessor have on the execution time.

TABLE 2. Application results

	bus #	register constraint	# cycles		run time (sec/cycle)	
			automata	exact[a]	schedule	verify
elliptic wave filter	1	9 (17)	54	27	61.0	14.6
	2	8 (16)	42		18.7	1.0
		9 (17)	40		33.5	0.8
differential equation	1	4 (8)	20	7	26.7	1.0
	2	4 (8)	15		9.6	0.3
		5 (9)	14		11.6	0.3

a. assumes no bus constraints. ewf(2 bus) = 34 and diff_eq(2 bus) = 11 cycles.

5.3. Feasible Schedule Verification

Another application is to verify the feasibility of a schedule for a given data-path. This application investigates the binding freedom for memory and function units by symbolically executing the automata model for a given sequence of operations. The feasibility of a schedule is verified by the successful automaton execution subject to data-flow and data-path constraints.

The CPU execution times of the scheduler and verifier are listed in the final columns of Table 2. Although the two bus data-path is more complicated, its increased data-path activity enables our heuristic to identify productive schedules more efficiently. As expected, the smaller search space of the verifier leads to faster execution.

6. Conclusion

We present an automata model which concisely captures the constraints imposed by a data-path. A process for expressing data-paths in terms of the base components is presented and techniques for systematic translation of such specifications into Boolean functions are described. The benefits of this representation are demonstrated by applying the automata model to a commercial DSP microprocessor. Future work will expand the model to support variable-width bus structures and multiphase clocking schemes.

7. References

[1] A. Aziz, *et al.*, "HSIS:A BDD-Based Environment for Formal Verification", *Proc. 31st DAC*, 1994.

[2] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. Computers*, pp. 199-213, Aug. 1986.

[3] J. R. Burch, *et al.*, "Sequential Circuit Verification Using Symbolic Model Checking", *Proc.* 27th *DAC*, 1990.

[4] C. Ewering, "Automated High Level Synthesis of Partitioned Busses", *Proc. ICCAD*, 1990.

[5] B.S. Haroun, M.I. Elmasry, "Architectural Synthesis for DSP Silicon Compiler", *IEEE Trans. CAD/ICAS*, pp. 431-447, April 1989.

[6] A. Hu, *et al.* "Higher Level Specification and Verification with BDD's", *Computer-Aided Verification: Fifth International Conf.*, 93, Lecture Notes in Comp. Sci. v.697, Springer-Verlag, 1993.

[7] C.T. Hwang, *et al.*, "A Formal Approach to the Scheduling Problem in High Level Synthesis", *IEEE Trans. CAD/ICAS*, pp. 464-475, April 1991.

[8] C. Monahan, F Brewer, "Symbolic Execution of Data-Paths", *Proc. 5th Great Lakes Symp. on VLSI*, 1995.

[9] S. Note, *et al.*, "Combined Hardware Selection and Pipelining in High-Performance Data-Path Design", *IEEE Trans. CAD/ICAS*, pp. 413-423, April 1992.

[10] B. M. Pangrle, D. D. Gajski, "Design Tools for Intelligent Silicon Compilation", *IEEE Trans. CAD/ICAS*, pp. 1098-1112, Nov. 1987.

[11] N. Park, F. Kurdahi, "Module Asignment and Interconnect Sharing in Register Transfer Synthesis of Pipelined Data-Paths", *Proc. ICCAD*, 1989.

[12] P.G. Paulin, J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Trans. CAD/ICAS*, pp. 661-679, June 1989.

[13] I. Radivojević , F. Brewer, "Incorporating Speculative Execution in Exact Control-Dependent Scheduling", *Proc. 31st DAC*, 1994.

[14] H. J. Touati, *et al.*, "Implicit State Enumeration of Finite State Machines using BDD's", *Proc ICCAD*, 1990.

[15] F. S. Tsai, Y.C. Hsu, "Data-Path Construction and Refinement", *Proc. ICCAD*, 1990.