Symbolic Execution of Data Paths*

Chuck Monahan, Forrest Brewer

Department of Electrical and Computer Engineering
University of California, Santa Barbara

Abstract

We present a data-path model which concisely captures the path constraints imposed by a data-path, such as bus hazards, register constraints, and control encoding limitations. A process for expressing arbitrary data-paths in terms of this model's base components and techniques for systematic translation into Boolean functions are described. Finally, this model is expanded to represent the limitations of generating as well as moving operands by incorporating dataflow graphs. The power of this representation is demonstrated by applying the path-constrained model to scheduling on 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 synthesis of behavior, 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 the capabilities of the existing designs. For complex data-path networks, efficient exploitation of the resources offers 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.

To provide support for these synthesis and scheduling applications, the model must meet several requirements. The model must correctly restrict communications to those that are simultaneously feasible on the defined interconnection network. It must model the behavior of the storage elements including the limited capacity of register file. It must model the behavior and exclusive use of function units in the network. Finally it must model the effects of the control encoding on the execution of the datapath.

We choose to formulate this model as a symbolic Boolean relation [3][13]. This relation represents a state machine whose states encompass all possible activities of the data path constrained by a DFG (data flow graph) of operations to preform. Recent advances in Boolean representations (BDD's) have made the canonical representation of Boolean relations in a compressed, executable form feasible. This allows the rapid determination of sets of solutions meeting predefined constraints. For example, it allows us to rapidly determine all simultaneously feasible communications on a pre-defined data-path. Note that this task is much more complex than simply determining feasible paths for operand transfers. We must provide simultaneous paths for several operands potentially competing for limited connectivity resources.

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][9][10][11]. This model is typically represented as a connection graph and conventional graph search and matching techniques are applied. Recent model accommodate register files but restrict the connectivity [14]. The second type of system used a pre-defined data-path and generated microcode or control for the structure [4][8]. In these systems, the network is specially designed to simplify assignments of communications. Finally, a related research line of formal verification systems is being developed [1][2][6]. In contrast to the verification modeling, our model is intended for synthesis us. Thus, we are free to introduce heuristics which do not seriously detract from the power of the system but greatly enhance the speed.

In Section 2, the path constrained model is developed and shown to be capable of representing the behavior of a wide variety of data-path switching element styles. In Section 3, the Boolean relation for this model is formu-

* This work has been supported in part by fellowship donation from Mentor Graphics Corp.

lated, with special attention paid to efficient construction and execution. Finally a number of applications for this model are discussed and related results are presented in Section 4.

2 Path Constrained 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, in which operands are conditionally transferred to different wires, is distinguished from combinational logic which creates new operands. Figure 1 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.

Each component connects to a network of buses via a set of uni-directional ports. Bidirectional ports are modeled by combinations of uni-directional ports, switching elements, and switching control restrictions. The model places restrictions upon wire and component interfaces. First, each wire of the network contains a single source represented by a specified output port. Also each output port is connected to a single wire. While a wire may fan out to many destinations, each input port must be connected to a single wire. In the case where multiple wires converge at an input port, a switching element interfaces the wire set and the port.

A wide variety of data-paths can be represented by the path-constrained model. More complex components are built by combining the set of base components listed in Figure 1. Each complex component is partitioned into its operand storage, movement, and generating elements to enhance the component's behavioral specification. These partitions are directly modeled by the memory, switching, and function unit base components with appropriate control and wire interfaces, as shown in Figure 2. For example, an arithmetic logic unit capable of transmitting either of its operands or creating new ones is represented as the structure in Figure 2(a). Figure 2(b) depicts the partitioning of the storage and routing element implied by a register. Register files could be represented by a network of multiplexers and a latch, but this would cause the representation to contain information on the permutation order of stored data. For this reason an alternative formulation is

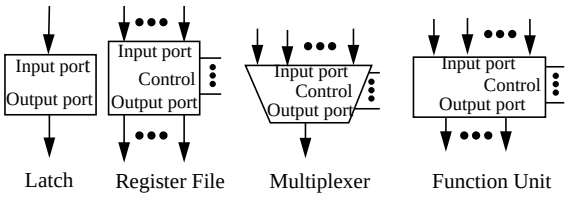


Figure 1. Base component set.

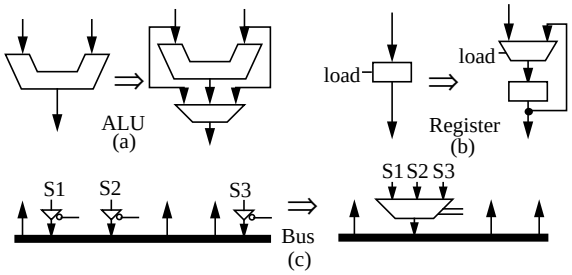


Figure 2. Representing larger structures in the model.

described in Section 3.2.1. The final example, Figure 2(c), shows how a tri-stated buffer based bus model is represented by our multiplexer based model.

Although this model assumes buses of uniform bit-width, it can model multiple bit-width buses. Figure 3 displays how multiple input and output ports can be utilized to accommodate operands of wider bit-widths. Note, that bit-wise operations, such as shifts, rotates and selections are modeled as function units since they alter the values of operands.

A *communication* consists of the transfer of an operand over a connection. A *connection* is a path over an arbitrary number of switching elements which connects two component ports on the data-path. The restrictions placed on the port and wire interfaces enable a connection to be equivalently defined as a path from a source wire to a destination wire. This definition implies a connection between two component ports since the source wire has an associated output port and the destination wire has a list of input ports.

A data-path has restrictions on the connections and communications that are simultaneously feasible. Buses restrict paths between output and input ports. Multiplexers generate restrictions since a multiplexer only transfers one source to its output. Similarly, function units and memory elements can be viewed as imposing restrictions upon communications, since they can produce only a single operand on each of their output ports. Lastly, control encoding constraints can further restrict the data-path activities.

3 Boolean Symbolic Formulation

This section formulates the Boolean symbolic representation for the path-constrained model. Conceptually,

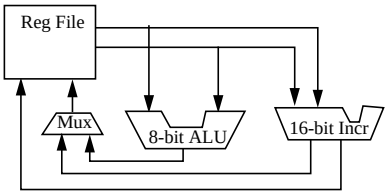


Figure 3. An example variable bit-width data path

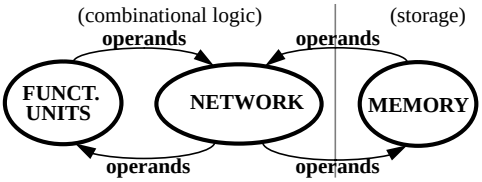


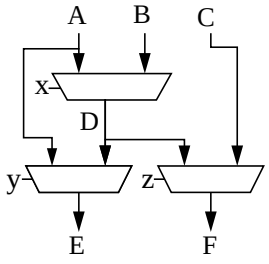
Figure 4. Path Constrained Model Organization

we wish to represent the behavior of the modeled data-path implicitly as a symbolic state machine. Isolating the memory elements of the modeled data-path transforms the model into a standard Huffman model of a state machine as depicted in Figure 4. In our machine, the state is represented by the set of operands stored in each of the memory components of the model. The next-state functions represent communications of operands on connections in the network and creation of new operands in the function units. Connection constraints model the behavior of the interconnection network while operand constraints model the behavior of the function units and memory elements.

3.1 Connection Constraints

The representation of a connection consists of an encoding of the source and destination wires and the switching control bits required to realize the connection on the network. Each wire in the network is labeled with a unique Boolean encoding. In this paper, upper case characters represent wire labels and lower case characters represent control bits. An example of the symbolic representation of connections is shown in the switching network in Figure 5. The set of possible communications are listed in the accompanying table.

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. 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).



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 5. An example data-path and the set of represented connections.

Initial wire function:

$W(D) = xA + \bar{x}B$

$W(E) = yA + \bar{y}D$

$W(F) = zD + \bar{z}C$

Resulting wire functions:

$W(E) = yA + \bar{y}(xA + \bar{x}B) = yA + \bar{y}xA + \overline{yx}B$

$W(F) = z(xA + \bar{x}B) + \bar{z}C = zxA + z\bar{x}B + \bar{z}C$

Figure 6. Wire function construction.

The construction technique for W(X) for an arbitrary networks relies on the acyclic nature of connections in switching networks[1]. 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 Figure 5 is shown in Figure 6.

3.2 Operand Constraints

In a given cycle, the operation of a data-path consists of a set of *communications*. A communication is the transfer of a specific operand on a connection. There are essentially two basic cases for operand constraints, one in which old operands are retrieved from memory elements, and the other in which new operands are created in function units.

It is useful to list the possible ways a data-path can supply specific operands. An *operand function*, F(W, opX), is defined as the sum of communications which supply operand *opX* to wire *W*. These equations elaborate the previous wire equations by adding the control signals required to generate the operand as shown in Figure 7.

3.2.1 Memory Constraints

Memory elements in the data-path provide "state" information for the symbolic machine execution. Thus,

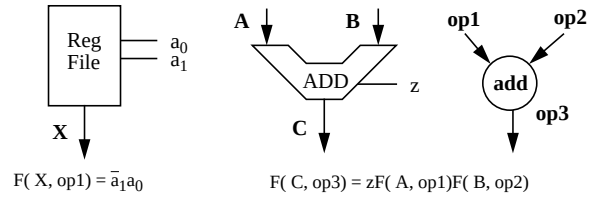


Figure 7. Example operand functions

1. Combinational connections must be acyclic to prevent races.

for each clock cycle, operands stored in the various memory elements are represented by the present state of the machine. Since a latch contains only a single operand, constraints for retrieving an operand from a latch are only dependent upon the state variables. By contrast, a register file contains multiple operands requiring additional control variables related to operand selection. It is possible to constrict register files 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.

3.2.2 Function Unit Constraints

A recursive construction technique is used to build a set of operand functions for arbitrary data-paths and data-flow. Dependencies are determined for each operand and function unit pair. Each dependency is a combination of input operands and function unit control signals. One can traverse the data-flow in order, evaluating an operand, only after its operand dependencies have been met. Each function unit that can construct the selected operand uses the operand functions of its predecessors to construct the total operand function.

Figure 8 depicts the combination of an example data-path and data-flow. For simplicity, this example ignores any permutation of operands to the function units. Initial functions are formulated by listing the conditions for the creation of each operand. The initial functions are expanded in Figure 9 by substituting operand functions into each dependent operand instance. 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. Finally, operand memory access functions are substituted for those operands coming from memory storage.

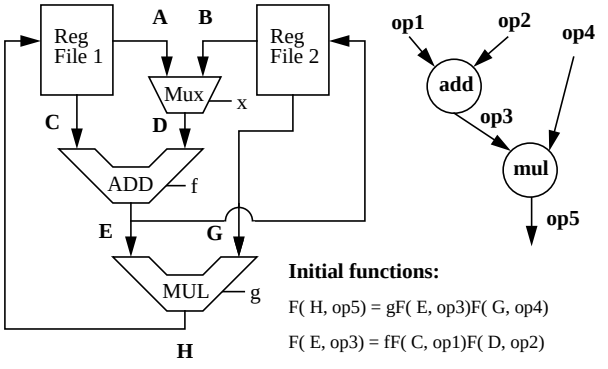


Figure 8. Example data-path and data-flow

Operand relations:

R(E, op3) = fF(C, op1)(xF(A, op2) + $\bar{x}$F(B, op2))Mem(RegFile2, op3)

R(H, op5) = gfF(C, op1)(xF(A, op2) + $\bar{x}$F(B, op2))F(G, op4) Mem(RegFile1, op5)

Transform relations:

R(H, op5) = gfF(C, op1)(xF(A, op2) + $\bar{x}$F(B, op2))F(G, op4)Mem(ReeFile2, op3)Mem(RegFile1, op5)

Figure 10. Construction of transform relation.

3.3 Transform Relation

A *transform relation* describing the relation between present and next states, is systematically built from the operand functions in a two step process. First, a set of *operand relations* are constructed from the operand functions. It is sufficient to only analyze operand functions pertaining to input ports for the memory components. Each operand function identifies the storage of a particular operand in a specific memory device. Multiplying the operand function by the state encoding corresponding to the operand and memory device creates an operand relation between the operands of the present state and the operands available at the next state. Figure 10 shows operand relations derived from the operand function of Figure 9.

The second step uses Eq.1 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 which operand relations for the set of memory input ports are compatible. If two communications require incompatible control signals, the product will be null. Figure 10 shows resulting transform for the example data path.

$$transform = \bigcap_{i=0}^{memoryinputports} \left(\sum_{j=0}^{operands} R(i,j) \right) \qquad (1)$$

A number of additional constraints may be added to the general transform relation. Control encoding constraints may be formulated as a Boolean function. Taking the product of the transform relation and this Boolean function constructs the set of permissible communications

Furthermore, external chip interfaces can be added to the general transform relation. The interface ports are modeled as function units as shown in Figure 11. For timed external I/O signals, operand functions pertaining

Converting wire functions to operand functions:

W(D) = xA + xB

Final operand functions:

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

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

Figure 9. Deriving operand functions.

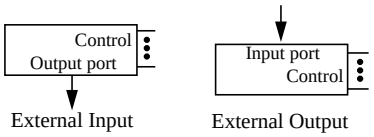


Figure 11. External I/O Interfaces

to specified data transfers are used to qualify the transform relations on specific states. This is equivalent to a time dependent encoding requirement. Alternatively, if the inputs do not contain timing restrictions, operand relations are built directly from the operand functions.

4 Applications

Many applications can be formulated around this symbolic Boolean data-path model.

4.1 Connectivity

A user of an existing data-path may desire to confirm whether a set of communications are feasible during a single state. Alternatively, they may desire to know which subsets from a list of communications may be simultaneously executed. These types of questions are directly answered by the model which builds all sets of simultaneous communications for a single clock period. One would simply intersect the specified list of communications which the transfer relation to determine the desired results.

We have built three data-paths using techniques presented in this paper. For simplicity, the models listed in this section do not utilize a specified DFG. Instead they are constructed from the set of wire equations and represent the set of feasible simultaneous connections supported by the data-paths. Table 1 lists the construction size for the Texas Instrument TMS32010 and 74888 data-paths and for the data and control path of the Intel 8085. The column "#terms" indicates the number of sets of simultaneously feasible communications for the example. The column "#nodes" list the number of BDD nodes required to represent the relation.

Table 1: Representation size

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

4.2 Feasible Schedule Verification

Another application is to verify the feasibility of a schedule for a given datapath. This application is a logical extension of the previous one, where the effects of multiple cycles are considered. This is a powerful technique since it will check the mapping of operands between states

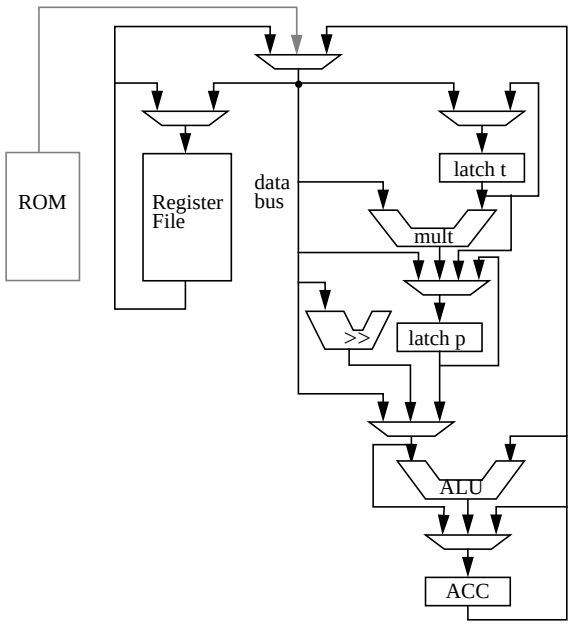


Figure 12. TMS32010 based data-path model

as well as confirming that the data-path can support the required communications.

Schedules can be specified in many forms. They may list when operations are done, the function unit bindings, and/or memory unit bindings. The application then symbolically executes the machine. At each state, the transform relation is restricted by the schedule and binding information supplied. Information left unspecified can be explored exhaustively or by heuristics to enhance the speed. Conventional FSM state enumeration techniques can be used to determine feasibility and all matching solution bindings.

4.3 Scheduling

If only a DFG is specified, an application could schedule the complete program. In general, however, the transform relation grows rapidly as the size of the DFG and data-path increase.

An application of this technique was constructed to schedule the *elliptic wave filter* and *differential equation* benchmarks for the TMS32010 design depicted in Figure 12. The *elliptic wave filter* example was scheduled in 54 states. Other techniques using general data-path models construct solutions in 27 states with equivalent register and bus constraints [7][12]. The wide variance in the scheduling time is based upon the restriction an existing data-path places upon the routability of operands.

For *differential equation*, multiplication by the constants 3 and 5 can be implemented by shifting and adding. We formulated three forms of the data-flow in regards to these options: using just the multiplier (*), just the shifter

(>>), and using both (*>>). Schedules for all three examples consisted of 20 states. This reflects the fact that the scheduling was restricted by the connection network. Adding a second global "data bus" to increase freedom decreased the time to 14 cycles for all sets of available units.

Because of the lack of restrictions, the execution times for these schedules varied between twenty minutes to two hours.

4.4 Engineering Change

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

5 Conclusion and Future Work

This paper introduces a concise and flexible data path model. Algorithms necessary to generate the Boolean relation for the data path are presented. The constraints were used in communication mapping and scheduling applications to demonstrate the feasibility of the model. Future work will expand the model to support multi-phase clocking schemes and data flow which contains loops and nested conditionals, as well as typing and bus bit-width extensions.

6 References

[1] A. Aziz, F. Balarin *et al.*, "HSIS:A BDD-Based Environment for Formal Verification", 31st ACM/IEEE Design Automation Conference, 1994.

[2] J. R. Burch, E. Clarke *et al.*, "Sequential Circuit Verification Using Symbolic Model Checking", 27th ACM/IEEE Design Automation Conference, 1990.

[3] O. Coudert, J. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits", Pro ICCAD-90, November 19990, pp. 126-129

[4] C. Ewering, "Automated High Level Synthesis of Partitioned Busses" Pro. ICCAD-90, November 1990, pp. 304-307.

[5] B.S. Haroun, M.I. Elmasry, "Architectural Synthesis for DSP Silicon Compiler", IEEE Trans on Computer Aided Design, April, 1989, pp. 431-447.

[6] A. Hu, D. Dill, et al. "Higher Level Specification and Verification with BDD's" Computer-Aided Verification: Fifth International Conference, 1993, published in Lecutre Notes in Computer Science v.697, Springer-Verlag, 1993.

[7] C.T. Hwang, J.H. Lee, and Y.C. Hsu, "A Formal Approach to the Scheduling Problem in High Level Synthesis" Transactions on Computer Aided Design, Vol 10 No.4 April 1991, pp. 464-475.

[8] S. Note, F. Catthoor, G. Goossens, H. De Man, "Combined Hardware Selection and Pipelining in High-Performance Data-Path Design" IEEE Transactions on Computer Aided Design, Vol 11 No.4 April 1992, pp. 413-423.

[9] B. M. Pangrle, D. D. Gajski, "Design Tools for Intelligent Silicon Compilation", IEEE Trans on Computer Aided Design, November 1987.

[10] N. Park, F. Kurdahi, "Module Asignment and Interconnect Sharing in Register Transfer Synthesis of Pipelined Data-Paths" Proc. ICCAD-89, November 1989, pp. 16-19.

[11] P.G. Paulin, J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", IEEE Trans on Computer Aided Design, June, 1989, pp. 661-679.

[12] I. Radivojevic, F. Brewer, "Ensemble Representation and Techniques for Exact Control-Dependent Scheduling", 7th International Symposium on High-Level Synthesis, May 1994, pp. 60-65.

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

[14] F. S. Tsai, Y.C. Hsu, "Data-Path Construction and Refinement", Proc. ICCAD-90, November 1990.