RC21963 (98710) 12 February 2001 **Computer Science**

IBM Research Report

Bridging the Domains of High-Level and Logic Synthesis

Reinaldo A. Bergamaschi
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Bridging the Domains of High-Level and Logic Synthesis

Reinaldo A. Bergamaschi
IBM T. J. Watson Research Center, NY, USA

Abstract

High-level synthesis operates on internal models known as control/data flow graphs (CDFG) and produces a register-transfer-level (RTL) model of the hardware implementation for a given schedule. For high-level synthesis to be efficient it has to estimate the effect that a given algorithmic decision (e.g., scheduling, allocation) will have on the final hardware implementation (after logic synthesis). The main problem in evaluating this effect is that the CDFGs are very distinct from the RTL/gate-level models used by logic synthesis. This makes it impossible to estimate hardware costs accurately. Moreover, the fact that high-level and logic synthesis operate on different internal models precludes on-the-fly interactions between these tools. This paper presents a solution to these problems consisting of a novel internal model for synthesis which spans the domains of high-level and logic synthesis. This model is an RTL/gate-level network capable of representing all possible schedules that a given behavior may assume. This representation allows high-level synthesis algorithms to be formulated as logic transformations and effectively interleaved with logic synthesis.

Index terms: high-level synthesis, logic synthesis, control and data flow graphs, register-transfer level, scheduling, allocation, logic optimization.

1 Introduction

High-level synthesis (HLS) is the process which maps a behavioral hardware-description language specification into a register-transfer-level (RTL) network. In most methodologies, this RTL network is then submitted to logic synthesis for gate-level optimization which attempts to produce a design satisfying certain area and delay constraints. Clearly the quality of the final result depends on the quality of the two tools. If the RTL network produced by HLS is excessively inefficient, there is no amount of logic optimization that will produce an acceptable design.

In order to produce an efficient RTL network, HLS has to estimate or compute the effect that a given high-level algorithmic decision will have on the final gate-level network. This effect is translated into *costs* which are used in most HLS algorithms, such as scheduling, allocation and resource sharing. In most scheduling/allocation algorithms [1, 2, 3], the costs are usually based on the number of states and number of resources. These metrics give a rough indication of the complexity and performance of the finite-state machine (FSM) and datapath area of the final design. However, they almost completely ignore important aspects such as the size and delay of the control logic, multiplexers and registers. The inaccuracy of these costs makes it impossible for any scheduling, allocation or resource sharing algorithm to produce optimal results (as measured in the quality of the final hardware).

The main problem in computing these costs accurately lies on the fact that the **internal model** in which HLS operates is too distinct from the final RTL network. In all HLS systems, this internal model is the *Control and Data Flow graph* (CDFG) [4, 5, 6], which is basically a more structured representation of the parse-tree generated by the language parser.

A CDFG represents the specification of the design at a very different level than the final hardware implementation. Although the CDFG may contain nodes representing hardware operators such as adders and subtracters, it usually does not contains any explicit specification of the multiplexers and control logic required by the implementation. An edge (or a node) in the CDFG is used to represent a value, but in hardware this value may become a simple net or a register, depending on the schedule. An adder node in the CDFG may be mapped onto an adder or functional unit in the RTL network, which in turn may be expanded into gates by logic synthesis and optimized with the surrounding logic. Hence, it might be inaccurate to consider simply the area and delay of an adder. These simple examples show that the final implementation (and cost) of a given CDFG node/edge is not really known until after HLS or even after logic synthesis, making it very difficult to measure hardware costs accurately during HLS. The main reason is that these costs are computed on a representation that is closer to the language level than it is to the hardware level that it is trying to measure.

Moreover, the fact that HLS and logic synthesis operate on different representations make it very inefficient for the two domains to interact. For example, during scheduling, it would help to know the exact size and delay of the resulting optimized control logic. In today's systems this would require HLS to finish synthesis completely, and then logic synthesis would process the controller in the RTL network. This is a time consuming and inefficient approach. If the two tools could operate on the same internal representation this problem would be resolved.

This paper presents a novel internal representation for high-level synthesis - called *Behavioral Network Graph* (BNG) - which solves the problems described above. This representation is an RTL/Gate-level network which can represent complete unscheduled behavioral descriptions.

The problem in representing unscheduled behaviors using RTL networks is primarily the determination of the states. In an RTL network the states are the registers and their number and transition relations are known. In a behavioral description, the states are not known *a priori*. An unscheduled behavior may be mapped onto multiple RTL networks. The scheduling task in HLS determines this mapping by placing operations (from the CDFG) into controller states. This defines both the FSM states as well as the datapath states (registers). Hence, the question is how can one represent a behavior, where the states are unknown, using an RTL network where all registers need to be predefined. This paper presents a solution to this problem which consists of an RTL network that can represent all possible schedules that a given unscheduled behavioral specification can assume.

Another goal of this work is to create an unambiguous representation of a given behavior. One long-standing issue in the high-level synthesis community has been the lack of a *formal* representation which could be unambiguously interpreted by different algorithms and systems. This has made it very difficult to share benchmark examples and be able to compare results between different systems. The use of VHDL or Verilog behavioral benchmarks [7] is insufficient because different systems interpret these languages in different ways (not always according to the language semantics). Sharing CDFGs is also impractical because algorithms are dependent on the CDFG semantics which vary from system to system.

In the logic synthesis domain this problem has been solved by using boolean representations (e.g., programmable logic array - PLA - tables [8]) which contain an underlying formalism - boolean algebra. In the case of PLA tables (or other boolean representations) the problem of ambiguity is almost non-existent because they can be uniquely mapped to boolean logic and manipulated using boolean algebra[1]. They can also be translated to a canonical representation using Binary-Decision diagrams (BDDs).

In the behavioral domain this has not been possible so far because there is no CDFG formalism that matches the role of a boolean representation in the logic domain. Nor there is any *behavioral algebra* that can manipulate existing CDFGs.

The representation described in this paper addresses this problem by representing behaviors using a logic network, thus allowing boolean algebra to be used in synthesizing the behavior. Although not part of this paper, this representation allows high-level synthesis algorithms, such as scheduling and allocation to be formulated in terms of logic transformations (similar to existing logic synthesis systems [9, 10, 11]), thus effectively unifying the behavioral and logical domains. To the best of the author's knowledge, this is the first work that demonstrates that behavioral and logical domains can be represented as a single RTL/gate-level model.

This paper is organized as follows. Section 2 presents the relevant high-level synthesis background related to this work. Section 3 explains in detail the algorithms for generating the Behavioral Network Graph. Section 4 describes a number of new algorithms and research approaches that become viable when using the Behavioral Network Graph as internal model for synthesis. Section 5 presents the conclusion.

[1]If don't cares are considered then there may still be some ambiguity in the logic and the mapping may not be unique.

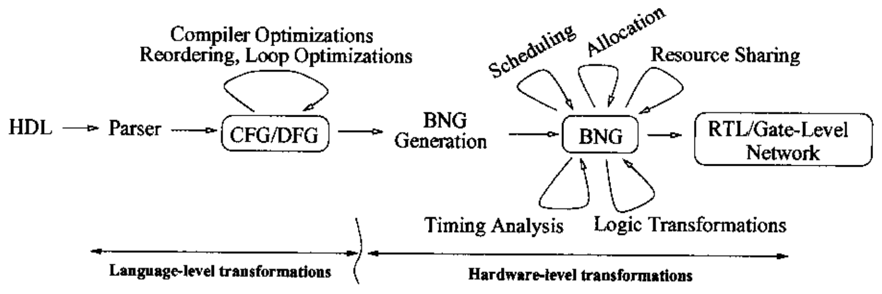


Figure 1: Organization of the BNG-based high-level synthesis system

2 High-Level Synthesis Background

In most HLS systems a parser is used to create a CDFG for a given language description. This CDFG is submitted to compiler-like optimizations such as constant propagation, loop-unrolling and data-flow analysis. The scheduling task operates on the CDFG and generates a finite-state machine (either as a graph or a state/transition table). The allocation and resource sharing tasks create the datapath. Both the FSM and the datapath are then combined to produce one final RTL network.

The system proposed in this paper decouples language-level transformations from hardware generation and transformation tasks, as illustrated in Figure 1. The CDFG is used as an extended parse tree, representing the same semantics as in the hardware language. Besides the compiler-like optimizations, the CDFG can also be submitted to certain domain-specific transformations such as reordering for parallelism extraction [3] and loop unfolding [12]. After these transformations, the order of operations in the CDFG is considered to be fixed. The next step is the mapping of this fixed-order CDFG into the BNG representation, which is an RTL/Gate-level representation of all possible schedules that the fixed-order CDFG can assume.

The tasks of scheduling, allocation and resource sharing are performed on the BNG. Since it is a logic-level representation, one can also perform logic transformations and static timing analysis on the design in order to evaluate accurately the costs involved during high-level synthesis. After these tasks, the BNG itself represents the final RTL/Gate-level network.

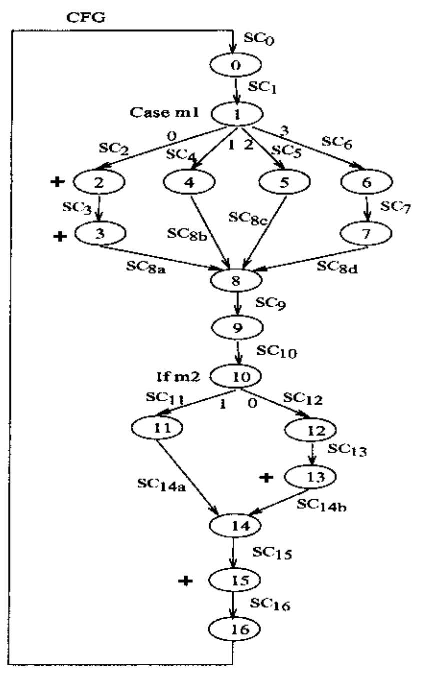
2.1 Control and Data Flow Graphs

Control and Data flow graphs exist in many different forms. This work uses a CDFG similar to [6] consisting of separate control-flow and data-flow graphs. The control-flow graph (CFG) represents the sequencing of operations as described in the language specification, including reordering and loop unrolling and unfolding. The data-flow graph (DFG) represents the data-dependencies among the operations and values. Together CFG and DFG represent all the information in the language

(a)

```
Entity bde is
port ( clock: in bit;
       in1, in2, m1: in integer range 0 to 3;
       m2: in boolean;
       out1, out2, out3, out4: integer range 0 to 3 );
end bde;
Architecture behavior of bde is
Begin
  Process
    Variable A, B : integer range 0 to 3;
  Begin
    Wait until not clock'stable and clock='1';
    B := 0;                                  -- O0
    Case (m1) is                             -- O1
      when  0   =>   B := in1 + in2;         -- O2
                     A := A + B;             -- O3
      when  1   =>   B := 3;                 -- O4
      when  2   =>   A := in2;               -- O5
      when  3   =>   A := 0;                 -- O6
                     B := 2;                 -- O7
    end Case;                                -- O8
    out1 <= A;                               -- O9
    If   (m2)                                -- O10
    then   A := in1;                         -- O11
    else   out2 <= B;                        -- O12
           out3 <= A + in1;                  -- O13
    end if;                                  -- O14
    out4 <= A + B;                           -- O15
  end process;                               -- O16
end behavior;
```

(b)



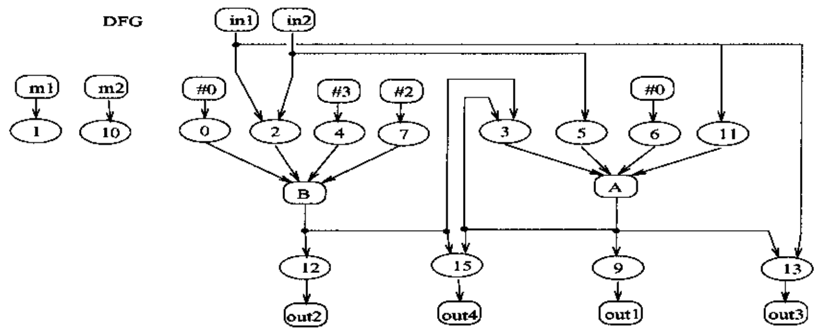


Figure 2: (a) VHDL description; (b) Separate control and data-flow graphs

specification in a format that is more suitable for synthesis purposes.

Figure 2 shows a simple VHDL description and the corresponding CFG and DFG. This example will be used throughout this paper. For each data-flow node there is a corresponding control-flow node. The reverse is not necessarily true, for example, *End_If* nodes have no correspondence in the data-flow graph.

This VHDL description can be synthesized in different ways by HLS, ranging from a solution where no states are inserted (i.e., the description is treated as an RTL specification) to a solution where several states are created by scheduling to satisfy certain constraints (i.e., it is treated as a behavioral specification). The BNG representation presented here allows the full range of schedules to be modeled.

2.2 Data-Flow Analysis

An essential step in language-based synthesis is data-flow analysis (DFA) [13]. Given that data-flow analysis is essential for the BNG generation algorithm, it is important that its main concepts be reviewed here.

DFA is a technique for computing the *definition-use* or *lifetime* of a given value. A *value* is defined as any assignment to a language variable, and two assignments to the same variable count as two values. In VHDL terms, values are defined as any assignment to variables and signals.

DFA computes the exact path in the CFG where a given value is defined, alive and used for the last time. In Figure 2(b), for example, the value assigned to variable A in operation O_3 is alive at operations O_8, O_9, O_{10}, O_{12}, O_{13}, O_{14}, O_{15}, O_{16}, and continues to be alive in the following iteration of the graph (through the feedback edge). This value is not alive at operation O_{11} because it assigns a new value to A, thus terminating the lifetime of the previous value along that path. On the other hand, the value assigned to variable B in operation O_7 is alive at operations O_8, O_9, O_{10}, O_{11}, O_{12}, O_{13}, O_{14} and O_{15}. Note that this value is not alive at operation O_{16} nor in the following iteration of the graph because operation O_{15} is the last use of B prior to being re-assigned (which happens at operation O_0 in the following iteration of the graph).

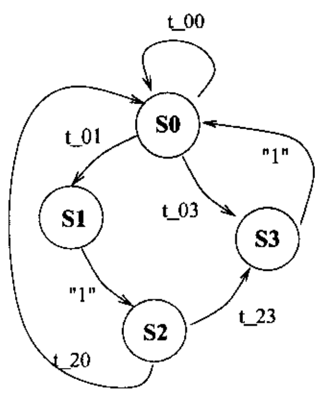
The lifetimes of values determine the possible interconnections between operations that create a value and those using the value. For example, operation O_{12} uses variable B as input. At this operation there are four possible values of B alive, assigned from: (1) O_2 if $M1$ equals 0, or (2) O_4 if $M1$ equals 1, or (3) O_0 if $M1$ equals 2, or (4) O_7 if $M1$ equals 3. Depending on the schedule, these values may come from a register or the operators directly, and may have to be channeled through a multiplexer into the operator implementing operation O_{12}.

2.3 Scheduling Basics

Scheduling decides the controller states in which the CDFG operations will be executed, and indirectly determines the values that will need to be stored in registers. To be able to handle general types of designs it is important that scheduling algorithms be able to handle control and data-flow operations efficiently. This requires a full analysis of all paths in the control-flow graph - such algorithms are called control-flow-based schedulers (e.g., [14, 15, 16]). Control-flow schedulers are capable of analyzing the complete CFG, as opposed to data-flow schedulers which need to partition the graph into basic blocks (sequences of operations containing no conditional operations) [3]. The CFGs considered in this work are general, including conditional operations, loops and non-series-parallel topologies.

Scheduling a CFG implies finding places in the graph where states are going to start and end. The term *state-cut* will be used hereafter to denote these places. In Figure 2(b), if the scheduling goal were to find a solution with only one addition operation per state, one possible solution would

(a)



Transition Conditions
t_00 = [(m1=1)+(m1=2)+(m1=3)].(m2=1)
t_01 = (m1=0)
t_03 = [(m1=1)+(m1=2)+(m1=3)].(m2=0)
t_20 = (m2=1)
t_23 = (m2=0)

State	Scheduled Operation Nodes	Condition
S0	0, 1	"1"
	2	m1=0
	4	m1=1
	5	m1=2
	6, 7	m1=3
	8,9,10	m1=[1,2,3]
	11,14,15,16	(m1=[1,2,3]).(m2=1)
	12,13	(m1=[1,2,3]).(m2=0)
S1	3	"1"
S2	8,9,10	"1"
	11, 14,15,16	m2=1
	12,13	m2=0
S3	14,15,16	"1"

(b)

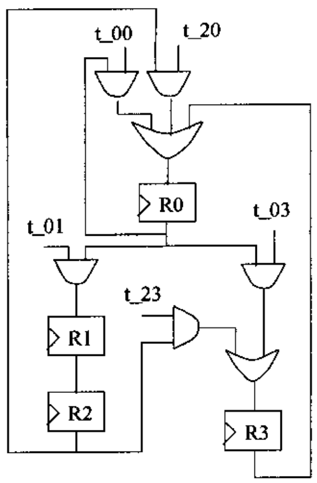


Figure 3: (a) FSM for scheduled CFG in Figure 2(b), (b) Hardware implementation of FSM using one-hot encoding

be to place *state-cuts* between operations $O_2 - O_3$, $O_3 - O_8$, and $O_{13} - O_{14}$, resulting in the FSM shown in Figure 3(a). If this FSM is implemented using *one-hot encoding* the result is the logic network shown in Figure 3(b). There is an implied assumption that the first node in the CFG is also the initial state, which is similar to say that the feedback edge going into the first node has an implicit *state-cut* .

Each *state-cut* has direct implications on the storage elements and interconnections in the datapath. When a *state-cut* is placed inside the lifetime interval of a value (as computed by DFA), it forces that value to be stored in a register since its definition is in one state and its use in another. A value that is used as input to an operation may come directly from the operation creating the value, if there is no *state-cut* between the two operations, or from a register storing the value, if there is a *state-cut* .

It is clear that the position of the *state-cuts* determine the basic control and datapath logic. Hence, for the BNG to represent all schedules, it needs to encompass the different hardware configurations for different choices of *state-cuts* .

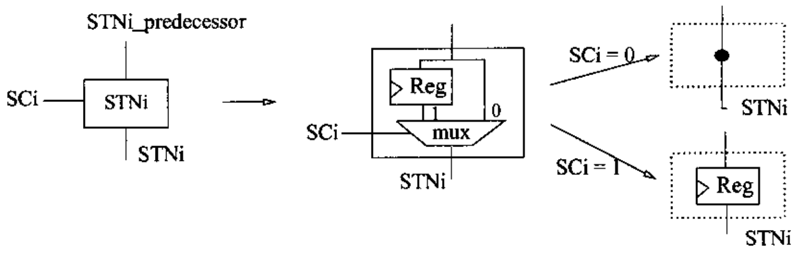


Figure 4: State-Value Node logic representation

3 Behavioral Network Graph

The Behavioral Network Graph is an RTL/gate-level representation of a behavioral specification. The BNG uses the CFG as a starting point for creating a logic network representing the FSMs for all possible schedules. It uses the DFG and the results of data-flow analysis to create a logic network representing the datapaths for all schedules (prior to resource sharing and logic optimizations). This is accomplished by the use of special logic gates called *State-Value Node*, *Register-Value Node* and *Current-Value Node.*

The algorithms describing the generation of the control and data parts of the BNG are given in the next sections.

3.1 Representing All Possible Schedules: Control BNG

As shown in Figure 3, each *state-cut* creates a new state starting at the succeeding operation, hence there is a direct correspondence between *state-cuts* and registers in the one-hot-encoded FSM. *State-cuts* can be placed at any CFG edge.

Let SC_i - denoted *state-cut variable* - be a variable associated with each predecessor edge of control-flow node i. This variable can assume values 0 and 1 depending on whether a *state-cut* is placed at node i (i.e., on the control-flow edge preceeding node i). If a control-flow node has multiple predecessor edges then *state-cut* variables SC_{ia}, SC_{ib},.., are associated with each predecessor edge.

The **State-Value Node** (STN) is a logic structure which represents the choice of either having or not having a *state-cut* on a particular control-flow edge. The STN is a switch which can choose between storing the input value, or passing it through the output immediately, controlled by a *state-cut* variable. The logic for a STN is given in Figure 4.

If SC_i is 0 (no *state-cut* on edge), the STN simplifies to a wire, thus not enforcing a new state. If SC_i is 1 (there is a *state-cut*), the STN simplifies to a register, thus enforcing a state transition.

In the BNG representing all possible schedules, SC_i is a variable. Once the schedule is fixed, the SC_i for each control-flow edge is set to 0 or 1, and the network can then be simplified by means of constant propagation.

The algorithm for Control BNG generation uses the CFG as input and consists of the following steps:

1. Traverse the CFG and associate a SC_i variable with each edge preceeding control-flow node i. If a node has multiple predecessors (a *join* node) then variables SC_{ia}, $SC_{ib}, \ldots$, are associated with each predecessor edge.

2. Traverse the CFG and for each control-flow node i with a single predecessor and a single successor, create a State-Value node STN_i . The net at the output of the STN_i gate, also called STN_i net, represents the control signal *activating* the operation in control-flow node i.

3. For *join* nodes, create a State-Value node STN_{ij} for each predecessor edge and connect all STN_{ij} nets to a single OR gate. The output of the OR gate is called net STN_i .

4. For nodes with multiple successor edges (*fork* nodes), create a State-Value node STN_i and connect its output net to as many AND gates as successor edges. Each AND gate has two inputs: the first input is net STN_i (for the fork node) and the other input is a net representing the condition on the corresponding successor edge. This condition net may be a primary input or a net coming from the datapath. The output of each AND gate is called net STN_{ij} .

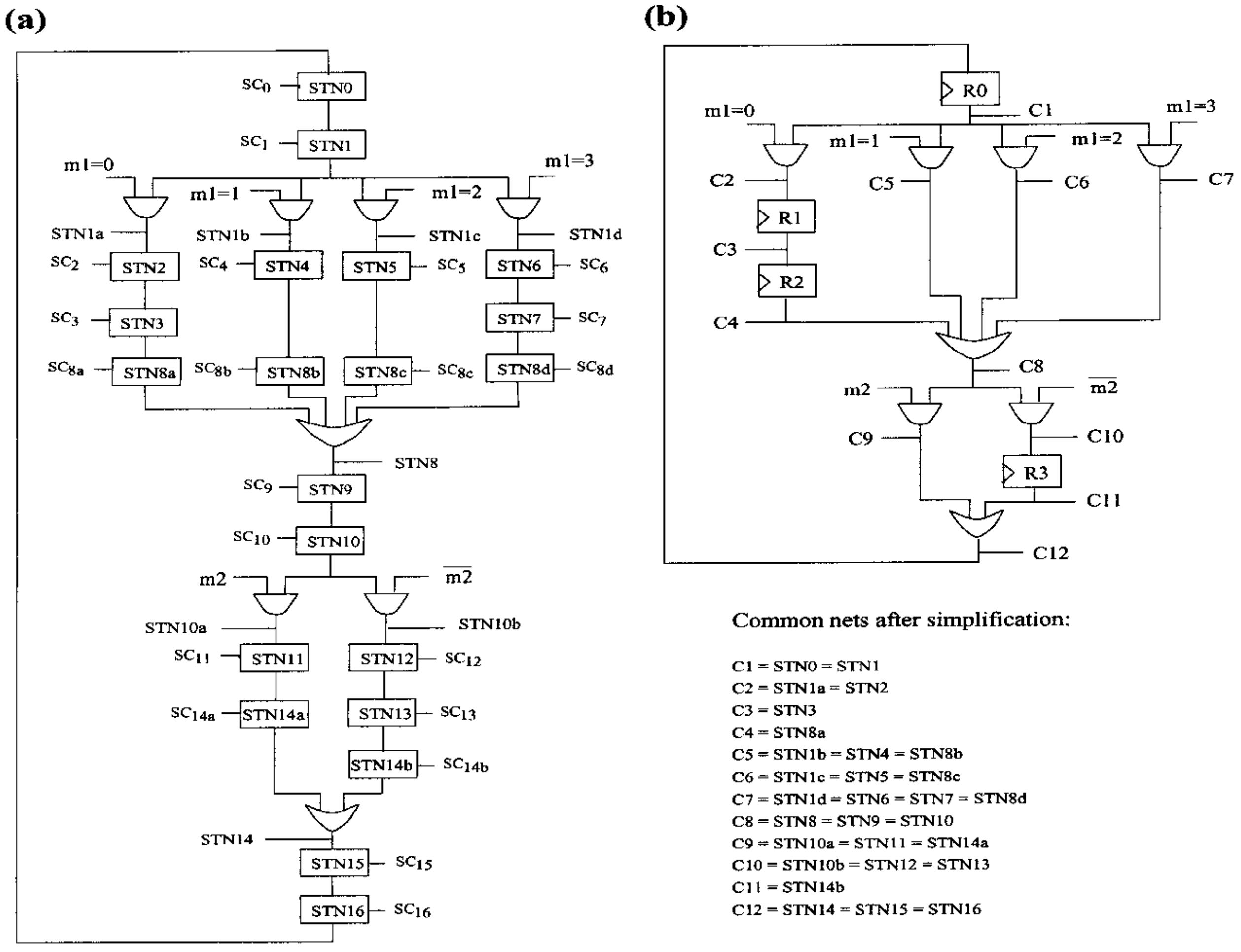
5. Connect the multiple STN_i boxes in the same topology as the CFG.

The resulting Control BNG for the CFG in Figure 2(b) is shown in Figure 5(a). Note that the extra STN_i boxes created for each predecessor edge in a join node are needed in order to allow *state-cuts* to be placed on each edge independently of the other.

Prior to assigning values to all SC_i variables, this BNG network represents all possible schedules for a given fixed-order CFG. By choosing different sets of values for all SC_i variables, one can effectively generate the resulting FSMs for multiple schedules. For example, to implement the same schedule as shown in Figure 3, one would simply set the SC_i variables corresponding to the chosen *state-cuts* to 1 and all other to 0. Hence, variables SC_0 (for the initial state), SC_3, SC_{8a} and SC_{14b} should be set to 1. After constant propagation, the BNG is simplified to the network shown in Figure 5(b), which is logically equivalent to the FSM in Figure 3(b).

The STN_i nets are the controlling conditions of all operations in the CFG. When a net STN_i is 1 it means that operation i is active (i.e., being executed). After scheduling is set and the Control BNG simplifies to a single FSM, several STN_i nets may become the same net, which simply means that the corresponding operations are all scheduled in the same state.

In order to evaluate the FSMs for different schedules, one has only to assign values to all SC_i variables, propagate the constants and evaluate the logic. All of which can be done with simple logic transformations.



Common nets after simplification:

C1 = STN0 = STN1
C2 = STN1a = STN2
C3 = STN3
C4 = STN8a
C5 = STN1b = STN4 = STN8b
C6 = STN1c = STN5 = STN8c
C7 = STN1d = STN6 = STN7 = STN8d
C8 = STN8 = STN9 = STN10
C9 = STN10a = STN11 = STN14a
C10 = STN10b = STN12 = STN13
C11 = STN14b
C12 = STN14 = STN15 = STN16

Figure 5: (a) Control BNG for CFG in Figure 2(b), (b) Control BNG after setting *state-cut* variables $SC_0, SC_3, SC_{8A}, SC_{14b}$ to 1

This process results in a one-hot-encoded FSM, which can be further optimized by means of state-encoding and state minimization [17, 18].

3.2 Representing All Possible Datapaths: Data BNG

The Data BNG is composed of gates representing registers, operators and interconnections, as well as the required control signals. Prior to fixing the schedule, it is unknown whether a value will become a register and therefore it is impossible to derive the final interconnections. As mentioned in Section 2.3, the positions of the *state-cuts* determine the FSM states as well as the values in the DFG that need to be stored in registers. Once these registers are fixed, it is possible to derive all interconnections between datapath operators and from/to registers and operators.

The Data BNG is a logic network which represents all possible value-to-register mappings and all possible resulting interconnection and control structures, for all possible schedules. This flexibility is achieved by the use of two special gates called *Register-Value Node* and *Current-Value Node.* The *Register-Value Node* is a logic structure representing all possible ways under which a variable may be stored under all possible schedules. The *Current-Value Node* represents all possible values that a variable may assume as an input to an operation, under all possible schedules. Necessary definitions and full details are given in the following sections.

3.2.1 Paths and Conditions

A few definitions are required at this point. Let $P = \{O_i, O_j, ..., O_m\}$, be a path in the CFG containing operations $O_i, O_j, ..., O_m$ such that they are all connected in sequence by control-flow edges (possibly including conditional edges).

Given such a path, one can define the following conditions:

Path-Closing Condition - $PCC(P)$ - is the condition under which all operations in the path P are scheduled and executed in the same state.

Path-Breaking Condition - $PBC(P)$ - is the condition under which the first operation in P, O_i is not scheduled and executed in the same state as the last operation O_m.

These two conditions will assume values 0 or 1 depending on the position of the *state-cuts* and on the values of the conditions on the control-flow edges along the path. As shown in Section 3.1, prior to scheduling the *state-cuts* are represented by SC_i variables and the conditions on the edges are represented by the STN_i nets in the Control BNG. Hence it is possible to write the the PCC and PBC equations for all paths in the CFG for all possible schedules in terms of the SC_i variables and STN_i nets along the path.

Computing $PCC(P)$ and $PBC(P)$:

For a basic block $BB_{pt} = \{O_p, O_q, ..., O_t\}$ in the CFG these conditions are represented by the formulas:

$$\begin{aligned} PCC(BB_{pt}) &= STN_p \wedge \overline{\bigvee_{i=p+1}^{t} SC_i} \\ PBC(BB_{pt}) &= STN_p \wedge \bigvee_{i=p+1}^{t} SC_i \end{aligned}$$

where STN_p is the net associated with the control-flow edge from the first operation O_p to the succeeding operation O_q. The formula for PCC implies that if the first operation in the basic block is active ($STN_p = 1$) and all SC_i variables are 0 (no *state-cuts* in the basic block), then $PCC = 1$. Similarly for PBC, if the first operation is active and at least one SC_i variable is 1 (at least one *state-cut*) then $PBC = 1$.

These formulas can be expanded recursively to handle paths spanning multiple basic blocks. For a path $P = \{O_i, ..., O_{f1}, ..., O_{f2}, ..., O_{fn}, ..., O_m\}$, with multiple basic blocks $BB_i, BB_j, ..., BB_m$

connected by *fork* nodes $O_{f1}, O_{f2}, ..., O_{fn}$ these formulas become:

$$\begin{aligned} PCC(P) &= \bigwedge_{t=all\ BB\ in\ P} PCC(BB_t) = STN_i \wedge STN_{f1} \wedge ... \wedge STN_{fn} \wedge \overline{\bigvee_{r=i+1}^{m} SC_r} \\ PBC(P) &= STN_i \wedge \{(\bigvee_{all\ SCj\ in\ BB_i} SC_j) \vee PBC(P - BB_i)\} \end{aligned}$$

The formula for $PCC(P)$ implies that the *Path-Closing Condition* for a path involving multiple basic blocks is the logical *AND* of the *PCC* conditions for the individual basic blocks. The formula for $PBC(P)$ implies that if the *Path-Breaking Condition* for any of the basic blocks in P is true then the condition for P is also true. The term $PBC(P - BB_i)$ denotes the condition for the remainder of path P excluding basic block BB_i.

As an example, consider the path $P = \{O_3, O_8, O_9, O_{10}, O_{12}, O_{13}\}$ in the CFG in Figure 2(b) and the corresponding Control BNG in Figure 5(a). The *PCC* and *PBC* conditions for this path are:

$$\begin{aligned} PCC &= STN_3 \wedge STN_{10b} \wedge \overline{(SC_{8a} \vee SC_9 \vee SC_{10} \vee SC_{12} \vee SC_{13})} \\ PBC &= STN_3 \wedge \{SC_{8a} \vee STN_8 \wedge [SC_9 \vee SC_{10} \vee STN_{10b} \wedge (SC_{12} \vee SC_{13})]\} \end{aligned}$$

Live-Path $LP(x, V_i)$**:**

Let x be a DFG variable and V_i be a value being assigned to x at CFG/DFG operation O_i. Let $LP(x, V_i) = \{O_i, O_j, ..., O_p\}$, denoted *Live-Path*, be a path in the CFG ranging from the operation creating the value (O_i) to the last operation where V_i is alive (O_p).

In Figure 2(b), for example, the live-path $LP(A, V_3)$ associated with the value assigned to variable A by operation O_3 consists of operations $\{O_3, O_8, O_9, O_{10}, O_{12}, O_{13}, O_{14}, O_{15}, O_{16}, O_0\}$. Note that this value is also alive in the next iteration of the graph, which is symbolically represented by adding operation O_0 as the last operation in the path. A given value may have multiple live-paths, starting at the same operation O_i and ending at different operations.

3.2.2 To Store or Not to Store a Value

The basic rule of register inference states that if a value is assigned in one state and used in another then it must be stored. This rule can be formulated in terms of live-paths and path-breaking conditions. Given a *Live-Path* $LP(x, V_i)$ for a variable x and a value V_i, if the path-breaking condition $PBC(LP(x, V_i))$ is true then the value V_i must be stored in a register. By collecting the path-breaking conditions for all assignments to variable x under all live-paths in the CFG, one can create the logic network representing all possible ways in which variable x needs to be stored, as well as the load-enable signals to register x, under all possible schedules.

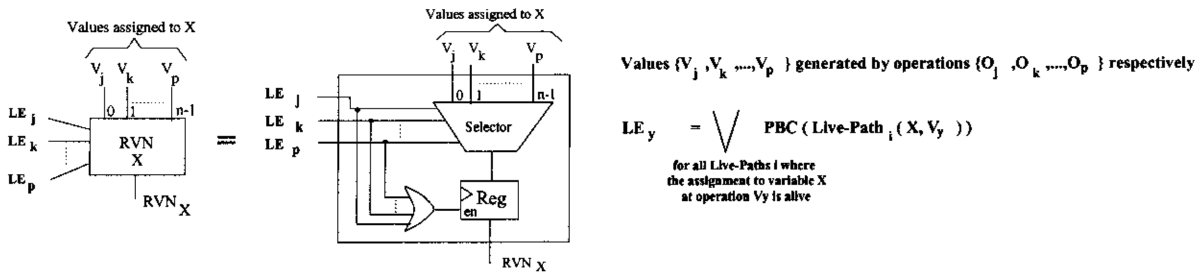

Values assigned to X
V j V k V p
0 1 n-1
LE j
LE k
LE p
RVN X
RVN X
=
Values assigned to X
Selector
Reg
en
RVN X
Values {V j ,V k ,...,V p } generated by operations {O j ,O k ,...,O p } respectively
LE y = ⋁ PBC (Live-Path i (X, V y))
for all Live-Paths i where the assignment to variable X at operation Vy is alive

Figure 6: Register-Value Node

The logic structure representing such configuration is called the **Register-Value Node** (RVN). It consists of a register fed by a Selector gate, which selects among all possible values that may need to be stored in the register (under different schedules). The *Load-Enable* signal for the register is the logical OR of all *path-breaking conditions* for all live-paths associated with the variable being stored. Figure 6 shows the logic structure for a Register-Value Node.

For a given DFG variable x, the set of values $V(x) = \{V_0,\ V_1,\ ...\ V_{n-1}\}$ that can be stored (under any possible schedule) is given by all the values assigned to variable x which have non-empty lifetimes. Given that an assignment value with empty lifetime is redundant and can be eliminated by data-flow analysis, the set of values $V(x)$ will include all values assigned to x.

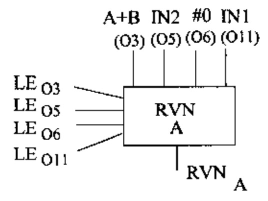
Figures 7(a) and (b) give the RVN for variables A and B (from the CFG/DFG in Figure 2(b)). The inputs are indicated as the operations in the CFG assigning values to A and B, and the load-enable conditions are given in terms of the STN_i and SC_i nets in Figure 5(a). These figures also show the RVN's final simplified hardware structures after the state-cut variables are assigned values 0/1 according to the schedule in Figure 3. In this example, the Load-Enable conditions are not null and variables A and B are in fact stored in registers. Alternatively, if all Load-Enable conditions connected to a register-value node are false, it means that no register is needed for the corresponding variable and the RVN gate can be deleted.

There is an underlying assumption that a single register is used to store a given DFG variable, as opposed to allowing different assignment values to be stored in a distributed way in multiple registers. This is not a limitation of the approach but an implementation choice found to result in more efficient hardware.

3.2.3 Representing All Possible Interconnections

The interconnections between datapath elements are also dependent on the schedule. Consider, for example, operation O_{12} in Figure 2(b) which uses variable B as input. If the design is synthesized without any *state-cuts*, all the assignments to B (from operations O_0, O_2, O_4, O_7) are alive at operation O_{12} and none of the corresponding *path-closing conditions* is false (since all SC_i variables are zero). This means that these values may be used by operation O_{12} in the same state in which

(a)



LE_{O3} = PBC(O3,O8,O9) + PBC(O3,O8,O9,O10,O12,O13) +
+ PBC(O3,O8,O9,O10,O12,O13,O14,O15) +
+ PBC(O3,O8,O9,O10,O12,O13,O14,O15,O16,O0,O1,O2,O3)

LE_{O3} = STN3.(SC8a+SC9) + STN3.(SC8a+SC9+SC10+ STN10b.(SC12+SC13)) +
+ STN3.(SC8a+SC9+SC10+ STN10b.(SC12+SC13+SC14b+SC15)) +
+ STN3.(SC8a+SC9+SC10+ STN10b.(SC12+SC13+SC14b+SC15+SC16+ SC0+SC1+ STN1a.(SC2)))

LE_{O3} = STN3.(SC8a+SC9+SC10+ STN10b.(SC12+SC13+SC14b+SC15+SC16+ SC0+SC1+ STN1a.(SC2)))

LE_{O5} = PBC(O5,O8,O9) + PBC(O5,O8,O9,O10,O12,O13) +
+ PBC(O5,O8,O9,O10,O12,O13,O14,O15) +
+ PBC(O5,O8,O9,O10,O12,O13-014,O15,O16,O0,O1,O2,O3)

LE_{O5} = STN5.(SC8c+SC9+SC10+ STN10b.(SC12+SC13+SC14b+SC15+SC16+ SC0+SC1+ STN1a.(SC2)))

LE_{O6} = PBC(O6,O7,O8,O9) + PBC(O6,O7,O8,O9,O10,O12,O13) +
+ PBC(O6,O7,O8,O9,O10,O12,O13,O14,O15) +
+ PBC(O6,O7,O8,O9,O10,O12,O13-014,O15,O16,O0,O1,O2,O3)

LE_{O6} = STN6.(SC7+SC8d+SC9+SC10+ STN10b.(SC12+SC13+SC14b+SC15+SC16+ SC0+SC1+ STN1a.(SC2)))

LE_{O11} = PBC(O11,O14,O15) + PBC(O11,O14-15,O16,O0,O1,O4,O8,O9) +
+ PBC(O11,O14,O15,O16,O0,O1,O4,O8,O9,O10,O12,O13)

LE_{O11} = STN11.(SC14a+SC15+SC16+ SC0+SC1+ STN1b.(SC4+SC8b+SC9+SC10+ STN10b.(SC12+SC13)))

Final Load-Enable Signals for RVNA using FSM in Figure 3(a)
and BNG Control one-hot implementation in Figure 5(b).
State-cut variables: SC0=SC3=SC8a=SC14b = 1; all other SCi variables = 0

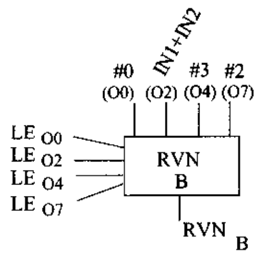
LE_{O3} = STN3 = R1

LE_{O5} = STN5.STN10b = R0.(m1=2).(m2=0)

LE_{O6} = STN6.STN10b = R0.(m1=3).(m2=0)

LE_{O11} = STN11 = (R0.[(m1=1) + (m1=2) + (m1=3)] + R2).(m2=1)

(b)



LE_{O0} = PBC(O0,O1,O5,O8,O9,O10,O11,O14,O15) +
+ PBC(O0,O1,O5,O8,O9,O10,O12) +
+ PBC(O0,O1,O5,O8,O9,O10,O12,O13,O14,O15) +

LE_{O0} = STN0.(SC1+ STN1c.(SC5+SC8c+SC9+SC10+ STN10a.(SC11+SC14a+SC15) +
+ STN10b.(SC12+SC13+SC14b+SC15)))

LE_{O2} = PBC(O2,O3) + PBC(O2,O3,O8,O9,O10,O11,O14,O15) +
+ PBC(O2,O3,O8,O9,O10,O12) + PBC(O2,O3,O8,O9,O10,O12,O13,O14,O15)

LE_{O2} = STN2.(SC3+SC8a+SC9+SC10+ STN10a.(SC11+SC14a+SC15) + STN10b.(SC12+SC13+SC14b+SC15))

LE_{O4} = PBC(O4,O8,O9,O10,O11,O14,O15) +
+ PBC(O4,O8,O9,O10,O12) + PBC(O4,O8,O9,O10,O12,O13,O14,O15)

LE_{O4} = STN4.(SC8b+SC9+SC10+ STN10a.(SC11+SC14a+SC15) + STN10b.(SC12+SC13+SC14b+SC15))

LE_{O7} = PBC(O7,O8,O9,O10,O11,O14,O15) +
+ PBC(O7,O8,O9,O10,O12) + PBC(O7,O8,O9,O10,O12,O13,O14,O15)

LE_{O7} = STN7.(SC8d+SC9+SC10+ STN10a.(SC11+SC14a+SC15) + STN10b.(SC12+SC13+SC14b+SC15))

Final Load-Enable Signals for RVNB using FSM in Figure 3(a)
and BNG Control one-hot implementation in Figure 5(b).
State-cut variables: SC0=SC3=SC8a=SC14b = 1; all other SCi variables = 0

LE_{O0} = STN0.STN1c.STN10b = R0.(m1=2).(m2=0)

LE_{O2} = STN2 = R0.(m1=0)

LE_{O4} = STN4.STN10b = R0.(m1=1).(m2=0)

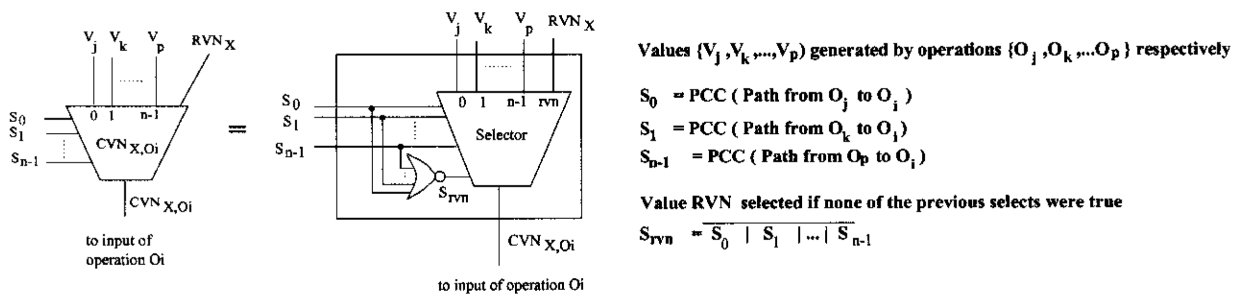
LE_{O7} = STN7.STN10b = R0.(m1=3).(m2=0)

Figure 7: (a) RVN for variable A, (b) RVN for variable B in Figure 2(b)

they are created; therefore, they will need to be multiplexed. If, however, a *state-cut* is placed between operations O_9 and O_{10} then all these values will be stored in a register (for B) and its output will be connected to the input of operation O_{12}. This is the case when all *path-closing conditions* are false.

Given a variable x used as input to operation O_i, the interconnection structure selecting a value for x under all possible schedules will depend on: (1) all values assigned to x and alive at operation O_i, and (2) the *path-closing conditions* for the live-paths from all assignment operations to O_i. The logic element implementing such a structure is called the **Current-Value Node** for variable x at operation O_i (CVN_{x,O_i}). It consists basically of a Selector gate multiplexing among all assignment values plus the *register-value node* for the variable. The *register-value node* is necessary for the case where none of the assignment operations is scheduled in the same state as O_i, in which case the register is used. Figure 8 illustrates the logic structure for a *Current-Value Node*. Figures 9(a) and (b) present the actual *Current-Value Nodes* and *Select* signals for the use of variables A and B by operation O_{15} in the CFG/DFG in Figure 2(b). These figures show the complete equations for the select signals as well as the CVN's final simplified hardware structures after the state-cut variables are assigned values 0/1 according hto the schedule in Figure 3. If a select signal becomes null the corresponding data input can be removed from the CVN .

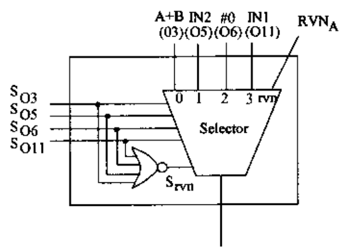
The last element required in the Data BNG is the **Operator Node** which implements a DFG operation. Operators can be anything from multibit adders and multipliers to logic gates. Conditional operations (e.g., If, Case statements) are implemented as decoders whose outputs are connected to the Control BNG. Initially, there is a one-to-one mapping between DFG operations and BNG operators. This mapping can be later modified by resource sharing and binding. The complete Data BNG for the CFG/DFG in Figure 2(b) is given in Figure 10. Control signals are omitted for clarity.



Vj Vk Vp RVNX
S0
S1
Sn-1
0 1 n-1
CVN X,Oi
CVN X,Oi
to input of operation Oi
=
0 1 n-1 rvn
Selector
Srvn
to input of operation Oi
Values {Vj ,Vk ,...,Vp) generated by operations {O j ,O k ,...Op } respectively
S0 = PCC (Path from Oj to Oi)
S1 = PCC (Path from Ok to Oi)
Sn-1 = PCC (Path from Op to Oi)
Value RVN selected if none of the previous selects were true
Srvn = S0 | S1 |...| Sn-1

Figure 8: Current-Value node

(a) Current-Value Node for A at operation O15 - CVN(A,O15)
Operations O3, O5, O6 and O11 assign values to variable A which are alive at O15



to input A of operation O15

S_{O3} = PCC(O3,O8,O9,O10,O12,O13,O14,O15)

$\mathbf{S_{O3}}$ **= STN3.STN10b.**$\overline{\textbf{(SC8a+SC9+SC10+SC12+SC13+SC14b+SC15)}}$

- - - - - - - - - - - - - - - - - - - -

S_{O5} = PCC(O5,O8,O9,O10,O12,O13,O14,O15)

$\mathbf{S_{O5}}$ **= STN5.STN10b.**$\overline{\textbf{(SC8c+SC9+SC10+SC12+SC13+SC14b+SC15)}}$

- - - - - - - - - - - - - - - - - - - -

S_{O6} = PCC(O6,O7,O8,O9,O10,O12,O13,O14,O15)

$\mathbf{S_{O6}}$ **= STN6.STN10b.**$\overline{\textbf{(SC7+SC8d+SC9+SC10+SC12+SC13+SC14b+SC15)}}$

- - - - - - - - - - - - - - - - - - - -

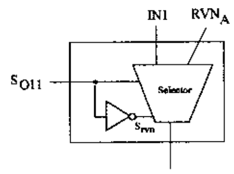
S_{O11} = PCC(O11,O14,O15)

$\mathbf{S_{O11}}$ **= STN11.**$\overline{\textbf{(SC14a+SC15)}}$

Select signals for CVN(A,O15) using FSM in Figure 3(b) and
and BNG Control one-hot implementation in Figure 5(b).
State-Cut Variables: SC0=SC3=SC8a=SC14b = 1, all other SCi variables = 0

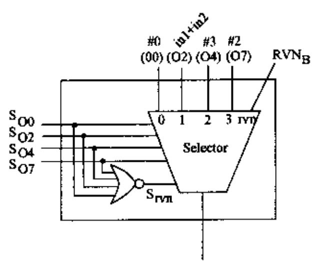
S_{O3} = 0 S_{O5} = 0 S_{O6} = 0

S_{O11} = STN11 = (R0.[(m1=1)+(m1=2)+(m1=3)] + R2).(m2=1)



to input A of operation O15

(b) Current-Value Node for B at operation O15 - CVN(B,O15)
Operations O0, O2, O4 and O7 assign values to variable B which are alive at O15



to input B of operation O15

S_{O0} = PCC(O0,O1,O5,O8,O9,O10,O11,O14,O15) +
+ PCC(O0,O1,O5,O8,O9,O10,O12,O13,O14,O15)

$\mathbf{S_{O0}}$ **= STN0.STN1c.STN10a.**$\overline{\textbf{(SC1+SC5+SC8c+SC9+SC10+SC11+SC14a+SC15)}}$ **+**
+ STN0.STN1c.STN10b.$\overline{\textbf{(SC1+SC5+SC8c+SC9+SC10+SC12+SC13+SC14b+SC15)}}$

- - - - - - - - - - - - - - - - - - - -

S_{O2} = PCC(O2,O3,O8,O9,O10,O11,O14,O15) + PCC(O2,O3,O8,O9,O10,O12,O13,O14,O15)

$\mathbf{S_{O2}}$ **= STN2.STN10a.**$\overline{\textbf{(SC3+SC8a+SC9+SC10+SC11+SC14a+SC15)}}$ **+**
+ STN2.STN10b.$\overline{\textbf{(SC3+SC8a+SC9+SC10+SC12+SC13+SC14b+SC15)}}$

- - - - - - - - - - - - - - - - - - - -

S_{O4} = PCC(O4,O8,O9,O10,O11,O14,O15) + PCC(O4,O8,O9,O10,O12,O13,O14,O15)

$\mathbf{S_{O4}}$ **= STN4.STN10a.**$\overline{\textbf{(SC8b+SC9+SC10+SC11+SC14a+SC15)}}$ **+**
+ STN4.STN10b.$\overline{\textbf{(SC8b+SC9+SC10+SC12+SC13+SC14b+SC15)}}$

- - - - - - - - - - - - - - - - - - - -

S_{O7} = PCC(O7,O8,O9,O10,O11,O14,O15) + PCC(O7,O8,O9,O10,O12,O13,O14,O15)

$\mathbf{S_{O7}}$ **= STN7.STN10a.**$\overline{\textbf{(SC8d+SC9+SC10+SC11+SC14a+SC15)}}$ **+**
+ STN7.STN10b.$\overline{\textbf{(SC8d+SC9+SC10+SC12+SC13+SC14b+SC15)}}$

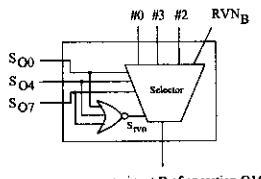
Select signals for CVN(B,O15) using FSM in Figure 3(b) and
and BNG Control one-hot implementation in Figure 5(b).
State-Cut Variables: SC0=SC3=SC8a=SC14b = 1, all other SCi variables = 0

S_{O0} = STN0.STN1c.STN10a = R0.(m1=2).(m2=1)

S_{O2} = 0

S_{O4} = STN4.STN10a = R0.(m1=1).(m2=1)

S_{O7} = STN7.STN10a = R0.(m1=3).(m2=1)



to input B of operation O15

Figure 9: (a) CVN for variable A, (b) CVN for variable B, both at operation O_{15} in Figure 2(b)

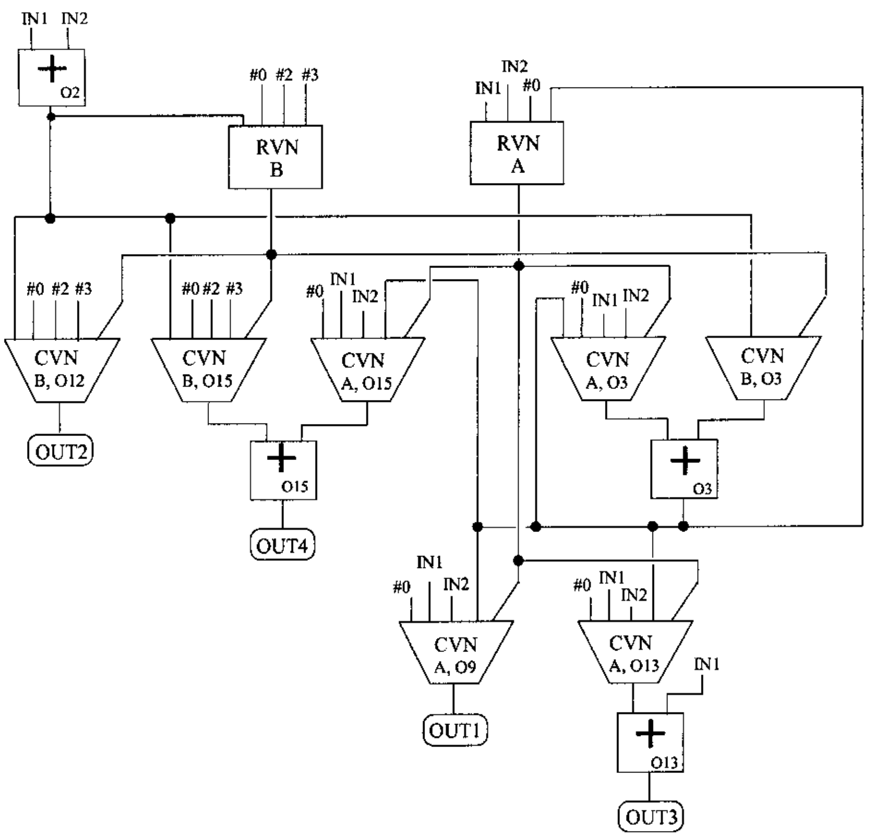


Figure 10: Complete Data BNG for the CFG/DFG in Figure 2(b)

3.3 A Complete BNG Example

This section illustrates the logic simplification process that transforms a BNG into a fixed RTL structure once the *state-cut* variables are assigned values 0 or 1. This process consists of constant propagation, elimination of disconnected gates and simple logic optimization.

As an example, consider the same schedule as used in Section 2.3, that is, *state-cuts* are placed between operations $O_{16} - O_0$, $O_2 - O_3$, $O_3 - O_8$, and $O_{13} - O_{14}$ which result in $SC_0 = SC_3 = SC_{8a} = SC_{14b} = 1$ and all other SC_i variables equal to 0.

Control simplification
After propagating the constant SC_i values through the *state-value nodes*, the STN_i nodes for which $SC_i = 1$ become plain registers, and those for which $SC_i = 0$ become a simple wire. The resulting Control BNG has four states and its one-hot-encoded implementation is shown in Figure 5(b). For the purposes of logic simplication, any AND expression with any two inputs involving state registers $R0, R1, R2, R3$ (in positive polarity) results in NULL since in the one-hot implementation, only one state register can be 1 at any time.

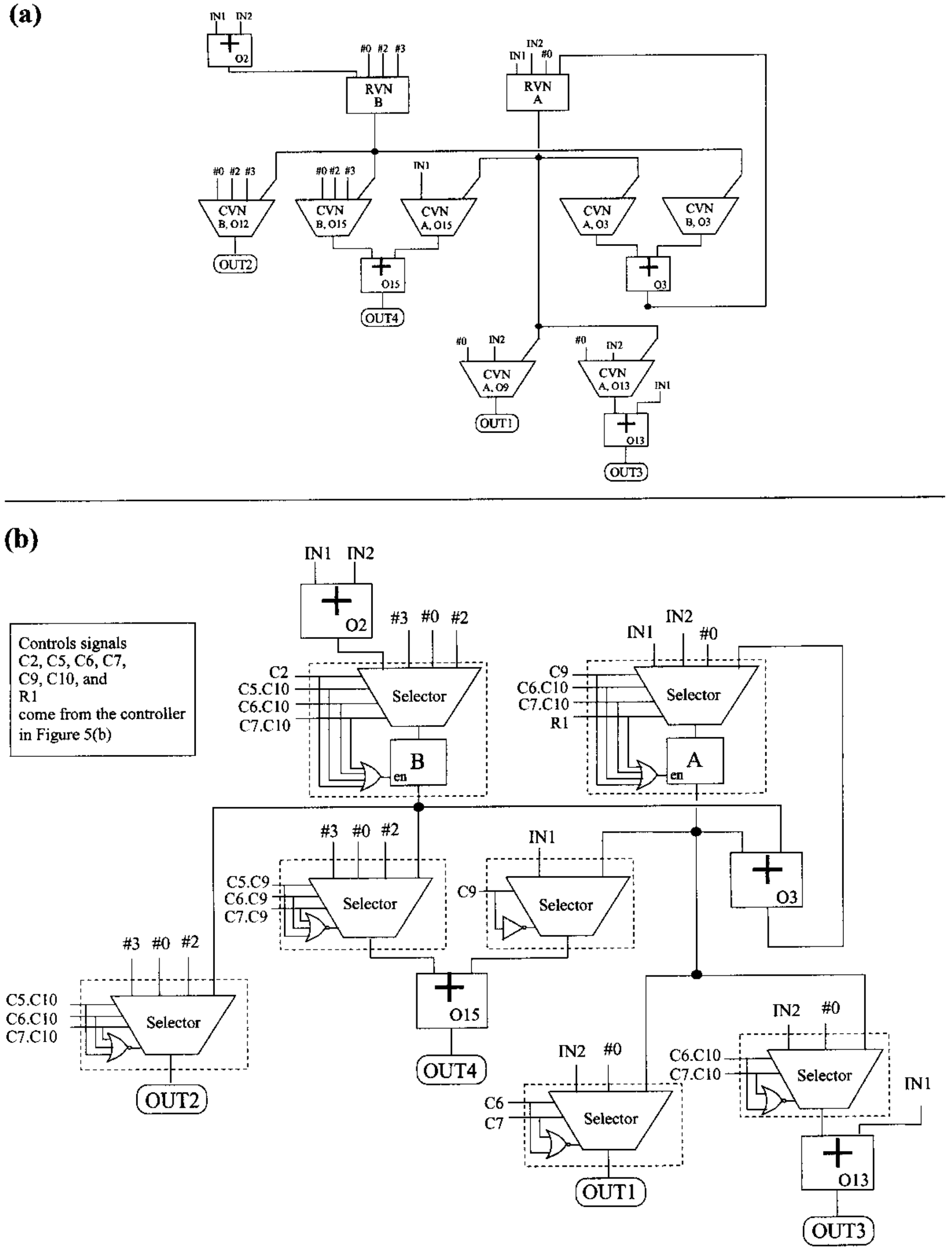


Figure 11: (a) Final BNG after dead connections are removed, (b) Final RTL structure after Data BNG simplification

Datapath simplification

When propagating the constant SC_i values through the *Register-Value* and *Current-Value* nodes, some of the load-enable and select signals may become 0. If the load-enable input in a RVN register is a constant 0, then the register is never active and can be removed by dead-logic elimination. If a select signal in a CVN selector is a constant 0, the corresponding selector input becomes a sinkless net and can be removed.

Figure 11(a) shows the BNG (from Figure 10) after deletion of dead connections due to control signals becoming 0. The resulting RTL structure, after simplifying the Data BNG, is shown in Figure 11(b). The control signals for the selectors in Figure 11(b) can be shown to be equivalent to those derived in Figures 7 and 9. Consider the load-enable signal LE_{O11} in Figure 7(a) whose final value is given by $(R_0.[(m1 = 1) + (m1 = 2) + (m1 = 3)] + R2).(m2 = 1)$. In Figure 11(b) this signal has the value $C9$ which comes from the control BNG in Figure 5(b). It can be easily seen that $C9$ is equivalent to LE_{O11}.

After control and data simplification, the final BGN represents the complete RTL design and, although extensive logic optimizations have not yet been performed, area and delay can be measured with reasonable accuracy and used to determine the quality of the schedule.

Note that, in general, there is no implied assumption that scheduling should be done prior to allocation. As explained later in Section 4, scheduling, allocation, and logic optimization can be done in any order using the BNG.

3.4 Complexity

The algorithm for generating the Control BNG is based on traversing the CFG (see Section 3.1). Its run-time and memory complexities are linear on the number of nodes and edges in the CFG.

The complexity of generating the Data BNG is dominated by the computation of the *Load-Enable* signals for *Register-Value Nodes* and the *Select* signals for *Current-Value Nodes.* Although a lot of specific *path* information is needed, there is no need to enumerate all possible paths in the CFG (which would obviously be impractical).

The process of generating these control signals is similar to the process for data-flow analysis, that is, all paths in the CFG are analyzed in one breadth-first traversal of the CFG. This approach is linear on the number of nodes although it can be expensive in memory usage. At every step of the traversal, the algorithm needs to store and propagate the following information for each variable alive at the current CFG node being visited: (1) set of values alive at the node; (2) the *path-closing* and *path-breaking* conditions for all values reaching the node.

Clearly, the *path-closing* and *path-breaking* conditions can get extremely complex, if generated individually for each value and each path. On the other hand, these conditions can be efficiently generated if decomposed into multi-level equations where each term corresponds to the conditions

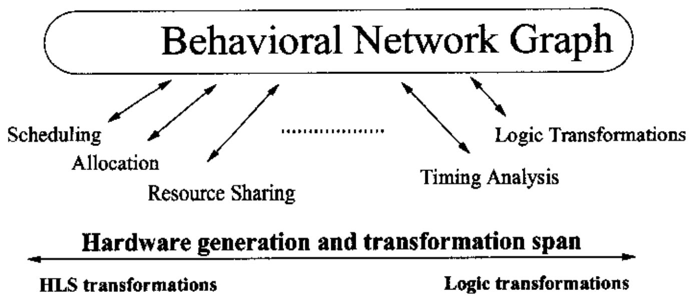


Figure 12: BNG's hardware generation and transformation span

for a basic block. By computing the basic-block conditions first and then combining them as needed for each value/path, the complexity of generating the control conditions become, in the worst case, linear on the number of basic blocks times the number of variables, as opposed to linear on the number of paths (which can grow exponentially).

These conditions are generated in the BNG directly as a multi-level logic network. They can also be efficiently computed using BDDs and then converted to logic gates. Once the scheduling is fixed and the SC_i variables assume values 0 and 1, the logic is simplified by constant propagation and the resulting network is an accurate representation of the control and datapath logic required in the design.

A simplified version of the BNG algorithms presented in this paper has been implemented in the *Hiasynth* system developed at IBM. The theoretical complexity of the algorithms has been observed in practice. Hiasynth's run-time is approximately proportional to the number of nodes in the CFG/DFG. Descriptions with thousands of nodes have been successfully synthesized.

4 BNG-based Algorithms

The Behavioral Network Graph is an RTL network which represents all possible schedules thus allowing the application of high-level synthesis algorithms such as scheduling and allocation. At the same time, it is also a logic network suitable for logic-level transformations. As shown in Figure 12, the BNG encompasses the hardware generation span from high-level to logic-level synthesis. This allows a number of new algorithms and research avenues to be explored. This section discusses some of these approaches.

Merging High-Level and Logic Synthesis

The use of the BNG allows high-level synthesis, specifically scheduling, allocations and resource sharing, to be formulated as a series of logic transformations, similar to logic synthesis. In addition it allows high-level and logic transformations to be interleaved. At each step, the BNG can be evaluated and a decision can be made on whether the transformations are accepted or reverted.

For example, prior to scheduling, one can perform static timing analysis on the whole BNG to determine the worst possible cycle time. A transformation can then gradually set selected SC_i variables to 1 (which effectively inserts registers in the path) and recompute the delays, until the cycle time is acceptable. At the same time, logic transformations may be used to optimize portions of the logic, which may decrease the delays and lead to a different scheduling solution. The scope for design space exploration becomes much larger and more accurate.

Accurate Cost Estimation during High-Level Synthesis

The BNG represents the complete control and datapath for all possible schedules. Even prior to scheduling, an analysis of the BNG can reveal the worst case multiplexers that may be needed at the inputs of registers and operators. Based on that, algorithms can select a specific scheduling or resource sharing solution to minimize the size of a given multiplexer.

Suppose, for example, that the adder at operation O_{15} in Figure 2(b) has a delay just under the target cycle time. The BNG shows that the adder has a 5-input selector gate at each input, selecting the possible values of A and B (see Figure 9). The delay of the selectors chained with the adder would clearly exceed the cycle time. Hence the only possible solution is to eliminate the selectors from the logic, which is obtained by choosing a scheduling solution that simplifies the selectors to a single input. An analysis of the select signals of $CVN_{A,O_{15}}$ and $CVN_{B,O_{15}}$ in Figure 9(a) and (b) reveals that this is achieved by setting SC_{15} to 1, that is, placing a *state-cut* between nodes O_{14} and O_{15}.

Scheduling, Allocation and Logic Optimization as a Unified Problem

Although the SC_i variables were explained in the context of scheduling, setting them to 0 or 1 actually establishes all registers (states and datapath) and interconnections in the design. Hence, the complete synthesis problem can be formulated in terms of choosing values for the SC_i variables which optimize the delay and area for the whole design.

High-Level Formal Verification

The BNG comprises the union of all sets of registers (both control and data) required by all schedules. Thus it represents a super state-machine which is the union of all state machines for all schedules. Given an initial state in the BNG and a corresponding state in the final FSM (for any given schedule), it can be formally proven that for any infinite sequence of states in the FSM there exists a matching sequence of states in the BNG.

Direct Mapping for RTL descriptions

The BNG can be used for mapping an RTL hardware description directly into a logic network. An RTL hardware description (in VHDL or Verilog) is considered here to represent a design where scheduling is not needed. The description may contain one or more states explicitly declared as *"wait until not clock'stable and clock='1';"* statements. Synthesis, in this case, still requires FSM generation (based on the predefined states), register inference, control and datapath generation.

The BNG can be transformed into the final RTL network by simply setting the *state-cut* variables corresponding to the position of the *WAIT* statements to 1, and all other SC_i variables to 0, and applying constant propagation.

4.1 Applying Satisfiability to Resolve the BNG

The problem of transforming a BNG into a final RTL network satisfying certain constraints can be modeled purely as a satisfiability problem. In order for the final RTL network to satisfy the constraints, it is possible that certain interconnections in the BNG may have to be removed, or registers may have to be inserted. These transformations are effected by making certain load-enable or select signals equal to 0 or 1. This would result in a set of Boolean equations dependent on the SC_i variables, to be equated to 0 or 1. The solution to this set of equations can be viewed as finding the minimum number of SC_i variables to be set to 1 so that all equalities are satisfied, which is a satisfiability problem.

5 Conclusions

This paper presented the theoretical foundation and algorithmic details for the Behavioral Network Graph - a novel internal representation for synthesis which effective bridges the gap between high-level and logic synthesis. The BNG is an RTL/gate-level network which represents all possible schedules that a behavioral specification can assume. This representation allows high-level and logic-level transformations to be applied simultaneously. The overall synthesis problem does not have to be decomposed into discrete optimization steps (e.g., scheduling, allocation, logic optimization) - using the BNG, all tranformations are equivalent to setting the SC_i variables to 0 or 1, which allows a more continuous and efficient design space exploration. The advent of the BNG makes it possible a number of new research avenues, including the merging of high-level and logic synthesis, and high-level formal verification. Details on these research avenues were also given.

References

[1] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design*, vol. CAD-8, pp. 661–679, June 1989.

[2] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high-level synthesis," *IEEE Transactions on Computer-Aided Design*, vol. CAD-10, pp. 464–475, April 1991.

[3] R. Bergamaschi and S. Raje, "Control-flow versus data-flow-based scheduling: Combining both approaches in an adaptive scheduling system," *IEEE Transactions on VLSI Systems*, vol. 5, March 1997.

[4] M. C. McFarland, "The Value Trace: A data base for automated digital design," Tech. Rep. DRC–01–4–80, Design Research Center, Carnegie-Mellon University, December 1978.

[5] A. Orailoglu and D. D. Gajski, "Flow graph representation," in *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, (Las Vegas), ACM/IEEE, June 1986.

[6] R. Camposano and R. M. Tabet, "Design representation for the synthesis of behavioral VHDL models," in *Proceedings 9th International Symposium on Computer Hardware Description Languages and Their Applications*, (Washington, D.C.), pp. 49–58, Elsevier Science Publishers B.V., June 1989.

[7] "1995 high-level synthesis design repository," 1995. Available on the WEB at http://www.ics.uci.edu/pub/hlsynth/HLSynth95.

[8] R. K. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis.* The Netherlands: Kluwer Academic Publishers, 1985.

[9] J. Darringer, W. Joyner, C. Berman, and L. Trevillyan, "Logic synthesis through local transformations," *IBM Journal of Research and Development*, vol. 25, July 1981.

[10] R. Rudell, "Tutorial: Design of a logic synthesis system," in *Proceedings of the 33rd ACM/IEEE Design Automation Conference*, (Las Vegas, NV), pp. 191–196, ACM/IEEE, June 1996.

[11] L. Stok, D. S. Kung, A. D. Brand, A. J. Sulivan, L. N. Reddy, N. Hieter, D. J. Geiger, H. H. Chao, and P. J. Osler, "BooleDozer: Logic synthesis for ASICs," *IBM Journal of Research and Development*, vol. 40, pp. 407–430, July 1996.

[12] G. Goosens, J. Vandewalle, and H. De Man, "Loop optimization in register-transfer scheduling for dsp systems," in *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pp. 826–831, ACM/IEEE, June 1989.

[13] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques and Tools.* Reading, MA: Addison-Wesley, 1986.

[14] R. A. Bergamaschi and D. J. Allerton, "A graph-based silicon compiler for concurrent VLSI systems," in *Proceedings of the IEEE CompEuro Conference*, (Brussels), pp. 36–47, IEEE, April 1988.

[15] R. Camposano, "Path-based scheduling for synthesis," *IEEE Transactions on Computer-Aided Design*, vol. CAD-10, pp. 85–93, January 1991.

[16] K. O'Brien, M. Rahmouni, and A. Jerraya, "A VHDL-based scheduling algorithm for control-flow dominated circuits," in *Sixth International Workshop on High-Level Synthesis*, (Dana Point, CA), ACM, November 1992.

[17] S. Devadas, H.-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: State assignment of finite state machines targeting multi-level logic implementation," *IEEE Transactions on Computer-Aided Design*, vol. CAD-7, December 1988.

[18] G. D. Hachtel, J.-K. Rho, F. Somenzi, and R. Jacoby, "Exact and heuristic algorithms for the minimization of incompletely specified state machines," in *Proceedings of The European Conference on Design Automation*, (Amsterdam, The Netherlands), pp. 184–191, IEEE, February 1991.