Behavioral Network Graph
Unifying 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). Currently, this effect cannot be measured accurately because the CDFGs are very distinct from the RTL/gate-level models used by logic synthesis, precluding interaction between high-level and logic synthesis. This paper presents a solution to this problem 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.

1 Introduction

High-level synthesis (HLS) is the process which maps a behavioral hardware-description language specification into an 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.

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* (usually based on the number of states and number of resources) which are used in most HLS algorithms, such as scheduling, allocation and resource sharing (e.g., [1, 2]). These cost 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 is 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* [3, 4], which is basically a more structured representation of the parse-tree generated by the language parser.

Permission to make digital/hardcopy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 99, New Orleans, Louisiana
(c) 1999 ACM 1-58113-109-7/99/06..$5.00

A CDFG represents the specification of the design at a very different level than the final hardware implementation. Although the CDFG may contain edges and nodes representing values and 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.

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 makes it very inefficient for the two domains to interact.

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 into 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. 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 [5, 6, 7]), 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.

2 High-Level Synthesis Background

The system proposed in this paper is organized as shown 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 [2] and

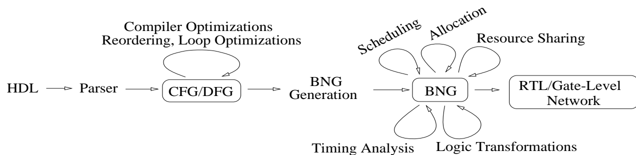

Figure 1: Organization of the BNG-based high-level synthesis system

loop unfolding [8]. 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

This work uses a CDFG similar to [4] consisting of separate control-flow and data-flow graphs. Figure 2 shows a simple VHDL description and the corresponding CFG and DFG. This example will be used throughout this paper.

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) [9]. 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.

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}.

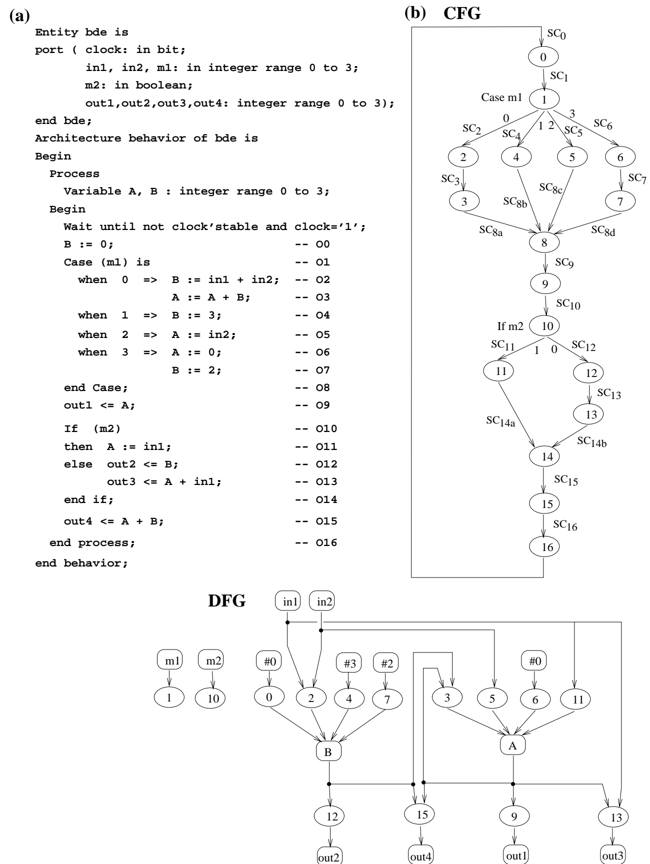

Figure 2: (a) VHDL description; (b) Separate control and data-flow graphs

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., [10, 11, 12]). 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 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

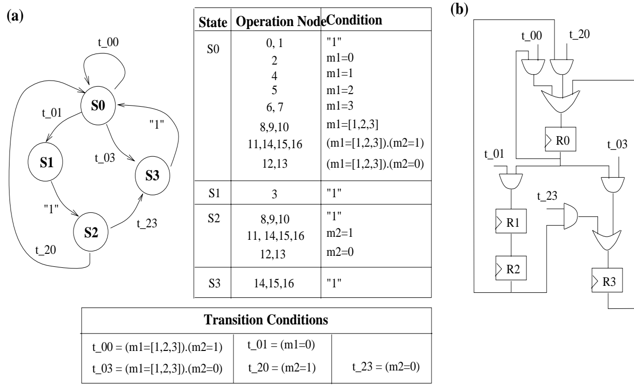

Figure 3: (a) FSM for scheduled CFG in Figure 2(b), (b) Hardware implementation of FSM using one-hot encoding

value, if there is a *state-cut* .

It is clear that the positions 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* .

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 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} 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.

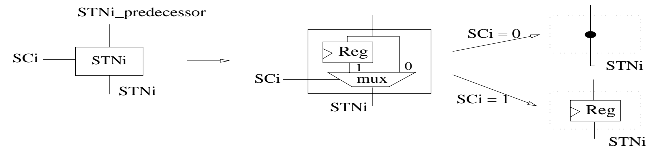

Figure 4: State-Value Node logic representation

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.

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

3.2 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

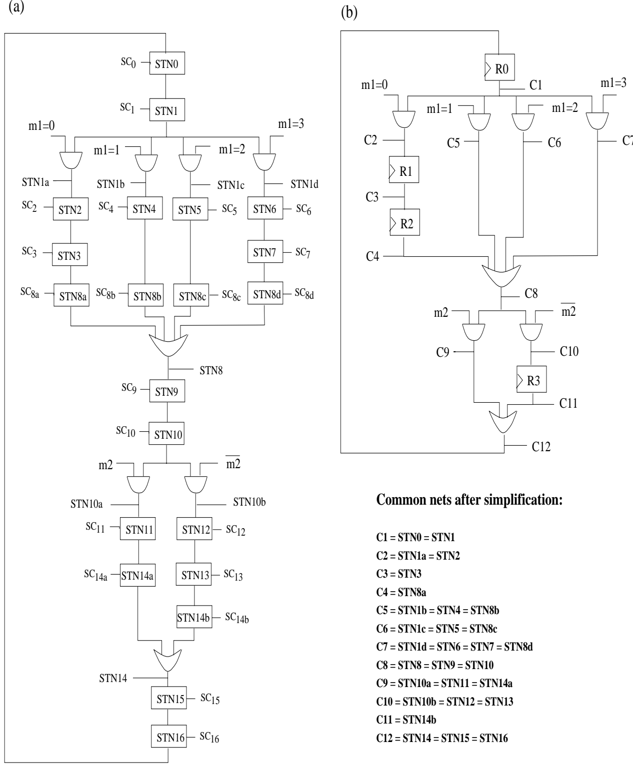

Figure 5: (a) Control BNG for CFG in Figure 2(b), (b) Control BNG after setting *state-cut* variables SC_0, SC_3, SC_{8A}, SC_{14b} to 1

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*.

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 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 formulae:

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

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 formulae 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 formulae 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:

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

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). A given value may have multiple live-paths, starting at the same operation O_i and ending at different operations.

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.

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(a) 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.

Figure 6(b) and (c) give the RVN for variables A and B (from the CFG/DFG in Figure 2(b)). The inputs are

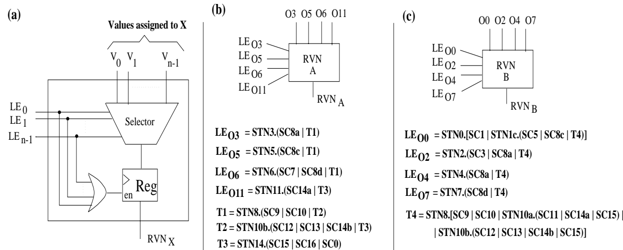

Figure 6: (a) Register-Value Node, (b) RVN for variables A, (c) RVN for variable B in Figure 2(b)

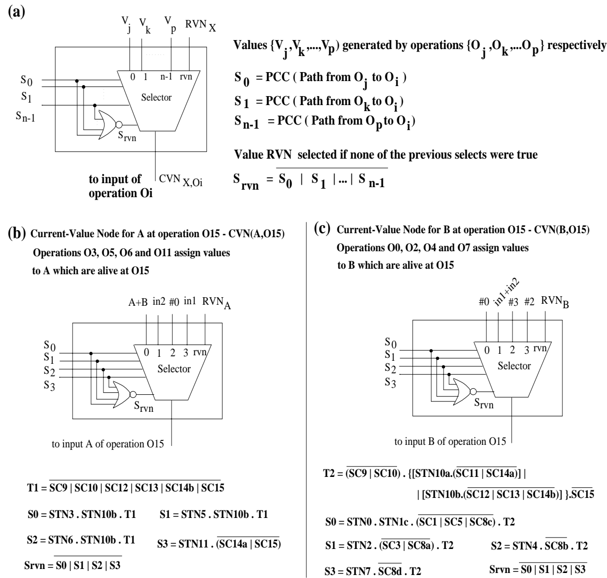

Figure 7: (a) Current-Value node, (b) Current-Value nodes for operations using variables A and B in Figure 2(b)

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).

After the scheduling is fixed, 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.

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 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 7(a) illustrates the logic structure for a *Current-Value Node*. Figures 7(b) and (c) 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).

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.

3.3 A Complete BNG Example

This section illustrates the logic simplification process that transforms a BNG into a fixed RTL structure once scheduling is defined. 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 $O_2 - O_3$, $O_3 - O_8$, 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 for which $SC_i = 1$ become a plain register, and those for which $SC_i = 0$ become a simple wire. The resulting Control BNG has four states, as shown in Figure 5(b).

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. In such cases, the corresponding input is disconnected and the selector gate simplified. The resulting RTL structure, after simplifying the Data BNG, is shown in Figure 8.

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.

4 BNG-based Algorithms

The existence of the Behavioral Network Graph - an RTL network representing all possible schedules - 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.

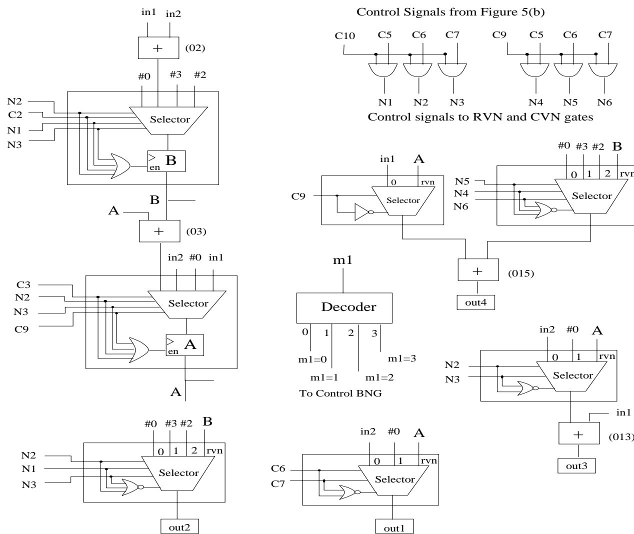

Figure 8: Final RTL structure after Data BNG simplification

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 7(b)). 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 to $CVN_{A,O_{15}}$ and $CVN_{B,O_{15}}$ in Figure 7(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.

5 Conclusions

This paper presented the Behavioral Network Graph - a novel internal representation for high-level 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. Using the BNG, high-level synthesis algorithms can accurately evaluate the effect of high-level decisions on the final hardware and perform 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. A simplified version of these algorithms have been implemented in the *Hiasynth* system in IBM and examples with thousands of operations have been successfully synthesized.

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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.