2013-05-11

Techniques for LI-BDN Synthesis for Hybrid Microarchitectural Simulation

Tyler S. Harris
Brigham Young University - Provo

Techniques for LI-BDN Synthesis for Hybrid Microarchitectural Simulation

Tyler S. Harris

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

David A. Penry, Chair
James K. Archibald
Doran K. Wilde

Department of Electrical and Computer Engineering

Brigham Young University

May 2013

ABSTRACT

Techniques for LI-BDN Synthesis for Hybrid Microarchitectural Simulation

Tyler S. Harris
Department of Electrical and Computer Engineering
Master of Science

Computer designers rely upon near-cycle-accurate microarchitectural simulation to explore the design space of new systems. Unfortunately, such simulators are becoming increasingly slow as systems become more complex. Hybrid simulators which offload some of the simulation work onto FPGAs can increase the speed; however, such simulators must be automatically synthesized or the time to design them becomes prohibitive. Furthermore, FPGA implementations of simulators may require multiple FPGA clock cycles to implement behavior that takes place within one simulated clock cycle, making correct arbitrary composition of simulator components impossible and limiting the amount of hardware concurrency which can be achieved.

Latency-Insensitive Bounded Dataflow Networks (LI-BDNs) have been suggested as a means to permit composition of simulator components in FPGAs. However, previous work has required that LI-BDNs be created manually. This paper introduces techniques for automated synthesis of LI-BDNs from the processes of a System-C microarchitectural model. We demonstrate that LI-BDNs can be successfully synthesized. We also introduce a technique for reducing the overhead of LI-BDNs when the latency-insensitive property is unnecessary, resulting in up to a 60% reduction in FPGA resource requirements.

Keywords: hybrid simulation, latency insensitivity, microarchitectural simulation, SPRI, synthesis

ACKNOWLEDGMENTS

The submission of my thesis, while not the end of my educational journey, does represent the close of the formal educational period of my life. As such, it is only fitting to look back in retrospect in gratitude at the immense quantity of aid I have received in this undertaking. I am deeply grateful to the professors and educators who have made the process of learning so enjoyable. I am in particular grateful to my advisor Dr. David A. Penry, who not only stuck with me through the long processes of researching, writing, and at times procrastinating my thesis, but who also taught me how to utilize my understanding to stretch my ambitions and attempt things I thought where impossible. I am also grateful to my parents, who raised me to see the value of education and hard work, and my twin brother Mitchell for his example and the comfort he provided me. I am additionally grateful to my friends Jessica, Gretchen, and Gordon; all three of whom encouraged me when I was doubtful, cheered for me when I was successful, and tolerated me when I was crazy.

Table of Contents

List of Figures

Chapter 1

Introduction

Humanity needs faster microprocessors. We may find solutions to many of the issues that will face humanity in the twenty first century using high performance computational techniques. In its report on High-End Capability Computing (HECC) the National Research Council claimed that "...high-end capability computing is distinguished by its ability to enable science and engineering investigations that would otherwise be infeasible." and "...computational approaches are essential to continued progress and will play an integral and essential role in much of twenty-first century science and engineering" [1]. Diverse fields, from astrophysics, to atmospheric sciences, to evolutionary biology depend upon higher performing computers to fuel new breakthroughs. Such fields may give us answers to problems that have plagued humanity. From the ability to accurately predict hurricanes, to even the cure to cancer, humanity has an immense benefit to reap from faster, higher performing microprocessors.

The performance of a microprocessor represents the balance of a large number of complex variables such as power, circuit size, frequency, latency, and pipeline control features. When evaluating a potential enhancement to a microprocessor it is very difficult to predict exactly how that enhancement will affect the performance of the processor. A computer architect could iteratively evaluate designs by fabricating test chips and measuring the performance characteristics of the chip; however, this approach is made intractable by the extreme cost, manpower, and time commitment necessary for microprocessor fabrication. Instead, computer architects and designers rely upon simulations when evaluating designs. Simulators allow designers to rapidly and cheaply test new ideas, explore the microarchitectural design space, and validate the behavior of a proposed system.

Moore's Law has made more transistors available on a silicon die, which micro architects have utilized to create more powerful, and more complex microprocessors. As microprocessors

have become increasingly complex the simulators that are used to model them have also become more complex. With the advent of multi-core microprocessors the work load of the simulator has further multiplied. Modern microarchitectural simulators have become too slow to permit an extensive exploration of complex future multi-core systems. Simulation of a single multi-core benchmark can require more than a week [2].

Researchers have proposed using FPGAs to accelerate simulators [2, 3, 4, 5, 6]. These FPGA-based *hybrid simulators* contain a software portion and a hardware portion which communicate through an interface. Such hybrid simulators can provide two orders of magnitude of speedup [2].

Designing such simulators manually has proved to be time-consuming. The process for such a design begins with the microarchitect partitioning the simulator into the software and hardware regions. Then the hardware sections must be developed in a hardware description language (HDL), a process that can take many man-months. When the hardware region has been developed a communication interface must be designed to allow the software region to invoke the FPGA accelerator. If at anytime the designers change their minds concerning the partition, or chooses to change the simulator in the course of design space exploration, the process must largely be repeated. The interface and hardware regions must be redesigned to accommodate the newly partitioned or eliminated elements.

As a result of the complexity of designing and maintaining hybrid simulators it has been proposed [7] that hybrid simulators be synthesized from simulation models written in structural software simulation frameworks such as SystemC [8], Unisim [9], and the Liberty Simulation Environment (LSE) [10]. A structural framework encourages the decomposition of a microarchitectural model to take place along structural boundaries. For example, a model's floating point unit would be modeled in a floating point module. The data that flows into and out of a module is expressed as signals, and the execution behavior of the module is expressed as a code sequence called a process. The advantages of this decomposition are two fold. First, it allows the microarchitectual simulation model to be arranged in a manner similar to how the implementation will be arranged. Second, it makes the communication between modules explicitly defined by the signals, which greatly simplifies the job of the chosen synthesis engine.

When creating a hybrid simulator a designer must consider how to control the elements that have been partitioned into the hardware section. A naive approach to managing synthesized components is to rely upon software to control them and simply use a minimal interface in hardware. This control requires communicating across the hardware/software interface for every individual execution of every synthesized simulator component. Such communication can quickly negate the speed up gains due to hybrid execution [11].

Instead, it is desirable to compose (internally connect) components in hardware. This scheme allows synthesized components that only communicate with other synthesized components to forgo the necessity of synchronizing with software. Only components that receive or transmit data to components in the software partition would be required to communicate across the hardware/software interface.

This desire to compose synthesized components conflicts with a fundamental limitation of FPGA implementations. This limitation arises because the FPGA must be used to model architectural constructs such as content-addressable memories and multi-ported array structures which are convenient to model using state machines and multiple clock cycles in the FPGA. However, in general, state machines which require multiple clock cycles are not composable because they can fall out of synchronization with each other. Some solution must be found to allow these state machines to resynchronize with each other if components are to be internally composed.

One proposed solution to this dilemma is to place FIFOs between the state machines implementing individual components. Simulation time is then represented by counting enqueue and dequeue operations. This approach has been taken in [12] and [3] and simplified and formalized as the theory of Latency-Insensitive Bounded Dataflow Networks (LI-BDNs) [13]. When LI-BDNs are used, the state machines in a simulation model communicate with each other through FIFOs. Each state machine is "wrapped" with logic for controlling these FIFOs and the state machines. As long as the wrapping and interconnection obey certain properties, the wrapped state machines may be composed.

Reference [13] describes a procedure to generate the wrappers which LI-BDNs require. However, this description assumes that one state transition of the state machine equals one clock cycle of simulation time and can be computed in a single FPGA clock cycle. An example is given in [13] of an LI-BDN which can take multiple FPGA cycles to model a single cycle of simulation

time, but this example cannot be derived from the stated procedure because the clean abstraction of a wrapper around a state machine is lost. If synthesizers are to automatically generate hybrid simulators, this abstraction must be maintained.

Hybrid simulator synthesis tools will not always be able to generate state machines in which one state transition equals one simulation clock cycle because of the FPGA implementation limitations previously mentioned. Synthesis tools therefore require a new procedure to wrap such state machines into LI-BDNs. This work makes the following contributions:

1. A procedure for wrapping multi-cycle state machines modeling a single cycle of simulation time into LI-BDNs.

2. An implementation of this procedure within SPRI, a hybrid simulator synthesis tool which can synthesize LI-BDNs from a SystemC architectural model.

3. A simple technique which removes FIFOs from the synthesized LI-BDN when latency-insensitivity is not required, resulting in a savings of up to 60% of FPGA resources.

As a result of this work, hybrid simulator synthesizers will be able to compose the SystemC modules that they chooses to place in the FPGA fabric. The resulting hybrid simulators will enjoy less communication overhead and more concurrency, resulting in faster simulators and allowing designers to explore a greater portion of the design space, leading to improved designs. Such designs will permit the creation of higher performing computing systems, which will be required to solve many of the engineering challenges of our day.

Chapter 2

Related Work

The SPRI synthesis engine yields Finite State Machines (FSMs) implemented in HDL. State machines, of which FSMs are a subclass, are described by a branch of mathematics known as automata theory. Automata theory describes many classes of state machine composition including direct, parallel, and hierarchical composition. This thesis is mainly concerned with the direct class of composition, where the output of one FSM is used as the input of another. While automata theory permits direct composition, it does not elaborate the techniques necessary to do so. State machines in automata theory have no notion of time and are able to consume inputs and produce outputs instantly. State machines in the real world do not behave this way; they may consume inputs and produce outputs completely asynchronously to each other.

Kahn Process Networks (KPNs) address the asynchronous nature of physical FSMs by using unbounded FIFOs with blocking reads and non-blocking writes to synchronize real world state machines [14]. In a KPN an FSM that is producing a token into an output must enqueue the token into a FIFO, where a consuming FSM may consume the token as an input whenever it is prepared to do so, or else wait for a token to appear. This allows the boundless FIFOs to be decoupled in time, yet still pass data between them. Of course, in a real, physical system FIFOs cannot be boundless, but by placing certain restrictions on the network writes may be made blocking, eliminating the need for bounded FIFOs [15]. KPNs have become widely accepted in the computing community, but have proved to be especially useful in implementations of data flow networks.

One example of a data flow network implemented with KPNs is the RAMP project [16]. RAMP is a hand designed hybrid microarchitectural simulator. The hardware portions of RAMP are composed of compute blocks that are synchronized to each other and software using A-Ports, which themselves are an implementation of a FIFO [12]. The designer of a RAMP compute block

must manually determines when to dequeue inputs and enqueue outputs. While A-Ports proved extremely useful for implementing hybrid simulators, they lacked the formal theory necessary for use in a general network, and therefore cannot be used in a hybrid simulator synthesis engine. Arvind recognized that RAMP represented a combination of Kahn Process Networks and Carloni's theory of latency insensitive design [17]. Latency insensitive design provides a theory for allowing a state machine to delay update, which allows the state machine to wait for its inputs to arrive, and therefore behave in a latency insensitive manner. Arvind provided a proof for the general case of combining latency insensitivity and KPNs in [13], which introduced LI-BDNs, upon which this work expands.

LI-BDNs are nearly sufficient for hybrid synthesis composition, but lack a method for composing multi-cycle components. Additionally, the SPRI SystemC synthesis engine cannot leverage LI-BDNs without a formal method for interfacing SystemC synthesized components with the LI-BDN interface.

Chapter 3

Background

The techniques presented in this thesis allow synthesized SystemC processes to be composed in hardware by making them compatible with and extending Latency Insensitive Bounded Dataflow theory. This chapter provides an overview of the relevant technical details of SystemC, synthesized processes, and LI-BDN theory in order to establish a common basis for discussing these techniques.

3.1 Latency-Insensitive Bounded Dataflow Networks

This section explains Latency-Insensitive Bounded Dataflow Networks and how Latency-Insensitive Bounded Dataflow Networks can be said to implement state machines. The formal definitions and proofs given in [13] are not repeated here, but rather an informal explanation is provided to encourage understanding. The reader is encouraged to consult the formal definitions if a more thorough explanation is desired.

3.1.1 LI-BDN Fundamentals

Bounded dataflow networks (BDNs) are dataflow networks [14] whose nodes are connected by bounded FIFOs of size ≥ 1. The individual nodes, called *primitive BDNs*, implement *patient* synchronous sequential machines (SSMs). A patient SSM merely means that there is a single enable signal controlling state update for the entire SSM. A primitive BDN is shown in Figure 3.1. FIFOs can be enqueued only when they are not full and dequeued only when they are not empty. All FIFOs are empty to start with. A FIFO's output is connected to a single primitive BDN and its input is also connected to only a single primitive BDN. Fanout is accomplished by creating "tee" nodes that take in an input and replicate it into two or more outputs. Such a tee node is in itself a primitive BDN.

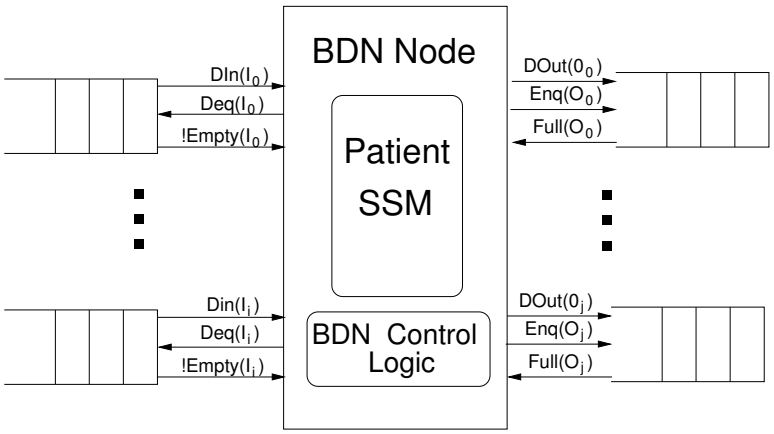


Figure 3.1: A primitive BDN

FIFOs which are inputs to the entire network are assumed to be driven by infinite sources which have an infinite supply of inputs and are always able to supply an input to input FIFOs whenever a FIFO is not full. FIFOs which are outputs from the entire network are assumed to drive infinite sinks which are able to remove values from output FIFOs at any time.

Bounded dataflow networks are able to implement SSMs if the notion of time is changed from a "wall clock" measurement to a "sampling-period-based" measurement. Sampling periods in the SSM are represented by enqueue and dequeue operations on FIFOs of the BDN. In particular, a BDN is said to *implement* an SSM if and only if:

- There exists bijective mappings between the outputs of the BDN and the outputs of the SSM and between the inputs of the BDN and the inputs of the SSM;
- the output histories of the SSM (i.e. the sequence of values which its outputs take at the end of each sampling period) and the output histories of the BDN (i.e. the sequence of values which are enqueued into its output FIFOs) match whenever the input histories match; and
- the BDN is deadlock-free.

This redefinition of time as enqueue/dequeue operations on FIFOs provides latency-insensitivity because the implementation of primitive BDNs can take any number of FPGA cycles to execute, but the simulation time of the simulated SSM increments only when enqueues and dequeues are performed. Note also that there is no need for a global logical time nor global synchronization; individual primitive BDNs are decoupled and may slip time with respect to each other.

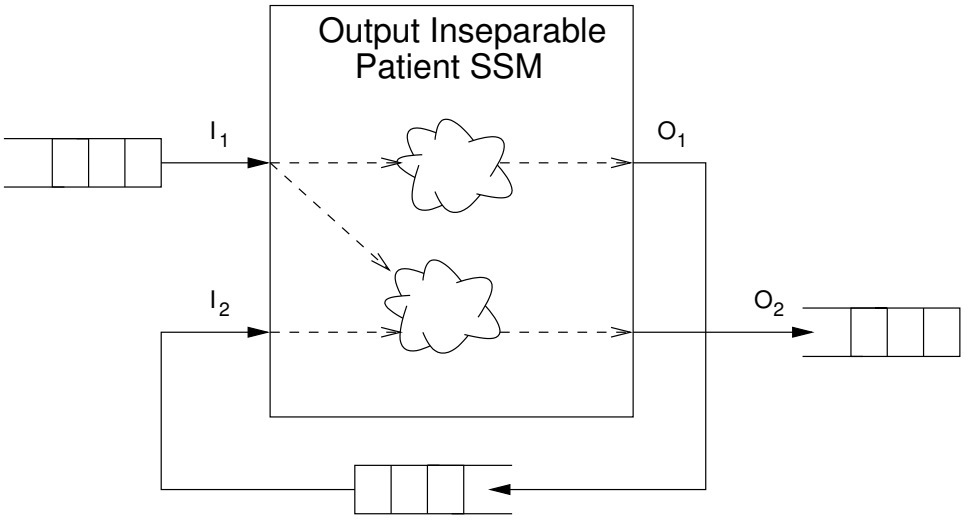


Figure 3.2: A BDN that would deadlock if the NED properties were not followed

3.1.2 Latency-Insensitive Properties

Arbitrary combinations of primitive BDNs may not be deadlock-free; however, if the primitive BDNs have two properties, deadlock may be prevented in many situations. These two properties force outputs to be produced and inputs to be consumed in a timely manner. They are:

No Extraneous Dependency (NED)

An output value must eventually be produced if all the inputs to which it is combinationally-connected (i.e. all the inputs in its fan-in cone) are available. This property ensures that there are no deadlocks in which outputs are not enqueued because input FIFOs are empty in a cycle.[1]

Self-Cleaning (SC)

If all outputs for a logical timestep have been produced, all the inputs for the logical timestep must eventually be dequeued. This property ensures that there are no deadlocks in which no output can be enqueued because output FIFOs are full in a cycle.

In order to demonstrate the necessity of the NED property, consider Figure 3.2 where output O_1 loops back into input I_2. If O_1 waited on both inputs I_1 and I_2 to be available before producing

[1] There is a technical condition that the property only needs to hold if all other output and input FIFOs have "caught up" to the previous simulated cycle.

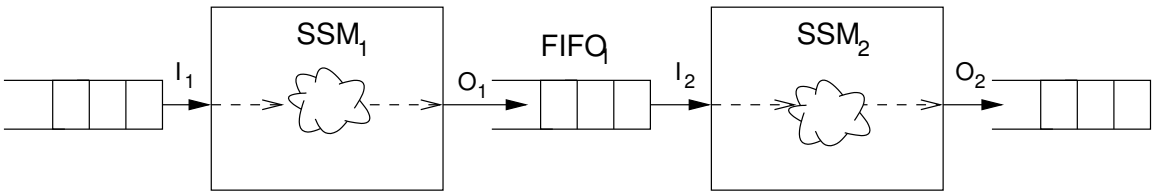


Figure 3.3: A BDN that would deadlock if the SC property where not followed

its output and the FIFO between O_1 and I_2 were ever empty, deadlock would occur because O_1 would would be waiting on itself to produce I_2. When NED is followed and O_1 depends only on I_1, the SSM would eventually produce O_1 whenever I_1 is available, independent of the availability of I_2. I_2 would then become available and O_2 could be produced since both I_1 and I_2 are available and deadlock will result.

The necessity of the SC property is much simpler to envision. For a trivial example consider Figure 3.3. If SSM_2 never dequeued $FIFO_1$ then eventually SSM_1 would be unable to proceed, due to the full FIFO ahead of it, and would be unable to advance simulation time.

When a primitive BDN possesses both the NED and the SC properties, it is known as a *primitive Latency-Insensitive BDN (primitive LI-BDN)*. Primitive LI-BDNs can be connected together (with FIFOs between them) to form LI-BDNs as long as their composition does not create a combinationally-connected cycle.

Reference [13] provides a simple procedure for wrapping an SSM into a primitive LI-BDN which implements that SSM:

1. Transform the SSM to a patient SSM by adding enable signals to all state elements and ANDing existing internal enable signals with the external enable signal.

2. Add enqueue and dequeue signals for outputs and inputs, and maintain a "done" flag for each output, as shown in Figure 3.4. Enqueue occurs when an output FIFO is not empty, it has not already been enqueued in this logical timestep, and all the inputs upon which the output depends are available. All inputs are dequeued when all outputs have been enqueued and all inputs are available; the enable signal for the patient SSM is asserted in the same FPGA cycle that all dequeue signals are asserted to the input FIFOs.

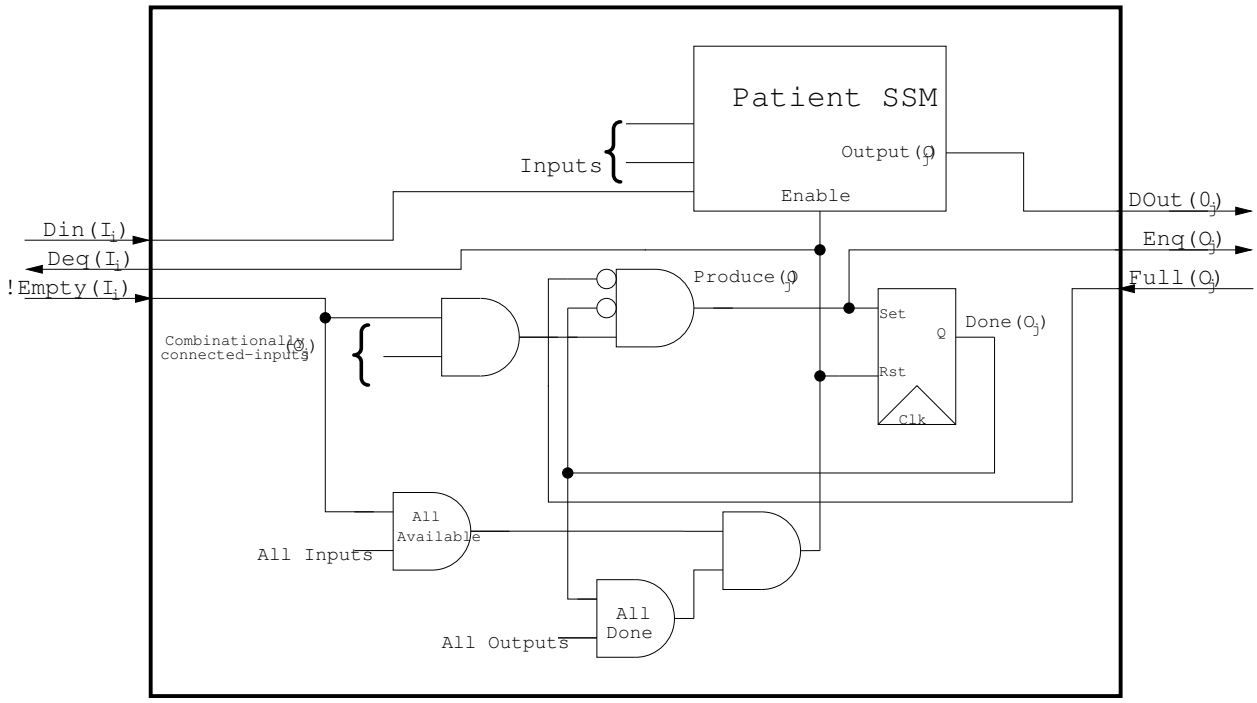


Figure 3.4: LI-BDN wrapper from [13]

Reference [13] does not formally prove that this procedure is correct, but it is easily observed that both properties are maintained: an output is enqueued whenever all its combinationally-connected inputs are available (NED) and all the input queues are dequeued once they are all available and all of the outputs have been enqueued (SC). Furthermore, the enable signal prevents the state from changing until all inputs are available and outputs are created, allowing the output histories to match.

3.1.3 Limitations of the LI-BDN Wrapping Procedure

The procedure introduced in [13] assumes that the SSM to be wrapped is an SSM whose behavior is to be literally modeled by the LI-BDN. That is to say, one state transition of the SSM equals one cycle of logical time. This in turn becomes one set of enqueue and dequeue operations of the LI-BDN. This forces SSMs under [13] to produce outputs and calculate their next state in a single cycle. An observation of Figure 3.4 shows that there is no provision for an SSM that produces outputs in multiple cycles.

Synthesized hybrid simulator components are SSMs, however, these SSMs *simulate* a cycle of logical time. Multiple transitions of the SSM may be required to compute a single logical cycle. As a result, the procedure of [13] is not applicable. Reference [13] does go on to argue that an LI-BDN could take multiple FPGA cycles to compute its outputs; indeed, this is part of the argument for using LI-BDNs. However, no general procedure for forming such an LI-BDN is given. There is one example given of refining an LI-BDN into one which uses multiple FPGA cycles in computation, but this example is manually generated and loses the clean abstraction of a wrapper around a state machine. Instead, the state machine and the LI-BDN control state are fused into one state machine which is then refined for multi-cycle behavior. This manual technique is not useful when synthesizing an LI-BDN network. Either a clear procedure for fusing the LI-BDN control state to the SSM state must be invented, or the LI-BDN control state must be modified to support SSMs that take multiple states to produce outputs and generate their next states.

3.2 SystemC

SystemC [8] is a library built on top of C++ that models a design as a collection of functions known as processes which are interconnected by directed signals. When fired, processes read their input signals and write to their output signals. The signal values form the *environment* in which processes run. SystemC processes have the following properties that are important to hardware synthesis:

Edge-triggered A process is fired, or run, by the SystemC framework when any event upon which it is waiting occurs. The process is said to be sensitive to these events, which are usually changes in input signal values.

Non-preemptive A process runs until it either returns or explicitly waits on an event in any firing of the process.

Output-inseparable Any firing of the process updates outputs based on the current values of the inputs and state. It is impossible to update only the outputs that are affected by the inputs that have changed.

Non-concurrent Processes may not execute concurrently unless the behavior appears identical to non-concurrent execution. Most notably this means that inputs to a process may not change unless the process yields control.

In addition to these process properties, signals must maintain delta-delay semantics; new signal values cannot be read in the same timestep in which they are written. In conjunction with non-preemptive execution and non-concurrency, the implication is that inputs to a process may not change while a process is firing, and outputs from a process do not change the environment until the process returns or waits.

```
1  #include <systemc.h>
2
3  SC_MODULE(ComboModule) {
4    sc_in<bool> MultEn;
5    sc_in<int> A;
6    sc_in<int> B;
7    sc_in<int> C;
8    sc_out<int> Product;
9    sc_out<bool> Comp;
10   void Process();
11
12   SC_CTOR(ComboModule) : MultEn("MultEn"), A("A"), B("B"), C("C"),
13   Product("Product"), Comp("Comp") {
14     SC_METHOD(Process);
15     sensitive << MultEn << A << B << C;
16   }
17 };
18
19 void ComboModule::Process() {
20   Comp = A < C;
21   if(MultEn)
22     Product = A * B;
23   else
24     Product = A + B;
25 }
```

Figure 3.5: An example of a SystemC combination module.

Figure 3.5 and Figure 3.6 both show simple modules written in SystemC, named `ComboModule` and `SeqModule` respectively. Each process has a number of input and output ports and a single

```
1  #include <systemc.h>
2
3  SC_MODULE(SeqModule) {
4    sc_in_clk Clk;
5    sc_in<bool> J;
6    sc_in<bool> K;
7    sc_out<bool> Q;
8    bool currentQ;
9    void Process();
10
11   SC_CTOR(SeqModule) : Clk("Clk"),J("J"), K("K"), Q("Q") {
12     currentQ = false;
13     SC_METHOD(Process);
14     sensitive << Clk.pos();
15   }
16 };
17
18 void SeqModule::Process() {
19   if(J and K)
20     currentQ = !currentQ;
21   else if(J)
22     currentQ = true;
23   else if (K)
24     currentQ = false;
25   Q = currentQ;
26 }
```

Figure 3.6: An example of a SystemC sequential module.

process. Line 15 in Figure 3.5 and line 14 in Figure 3.6 declare the signals to which each process is sensitive. `ComboModule` is sensitive to all of its inputs while SeqModule is only sensitive to an input clock edge. When the ComboModule process is fired its `Product` output is updated with either the product or sum of the `A` and `B` inputs, depending on the value of the `MultEn` input. The `Comp` output is also updated with the result of a comparison between `A` and `C`. SeqModule simply models a JK flip flop by setting the `Q` output based on the `J`, `K`, and current value of `Q` when there is a rising edge on its `Clk` input. It is important to note that these SystemC modules are far simpler than a typical SystemC module used in micro-architectural simulations. However, for the purpose of clear illustration, the examples in this thesis use only minimal modules.

SystemC processes may execute arbitrary code, however not all code nor design styles can be readily synthesized into hardware. Exactly what is considered synthesizable depends upon the capabilities of the synthesizer used. In this work, we will assume that processes to be synthesized

obey a simple set of rules similar to those supported by commercial vendors [18, 19] and proposed in the draft SystemC Synthesizable Subset standard [20]:[2]

- A process may only be sensitive to its inputs.
- A process may be either combinational or sequential. A combinational process must be sensitive to all of its inputs, and a sequential process may only be sensitive to the clock. Latches are not supported
- A combinational process must produce all of its outputs whenever it is fired.
- A combinational process must produce outputs that are only a function of its inputs, i.e.: it may not have internal state.
- A process may only alter its outputs and internal state and may not have side-effects.

3.3 SystemC Process Synthesis

Hybrid simulator synthesizers transform SystemC processes into FPGA hardware. We will call the generated hardware *FPGA-implemented Processes (FIPs*). FIPs inherit the properties of SystemC processes. They may also require multiple cycles to execute because of structures or operations that either cannot be synthesized as purely combinational elements, or doing so would incur prohibitive costs.

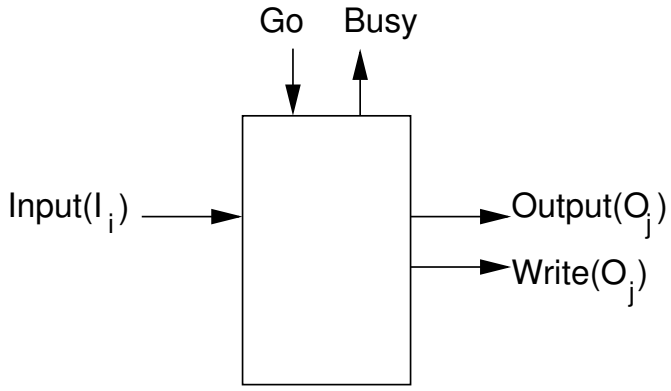


Figure 3.7: FIP Interface

[2]Processes remaining in software have no such restrictions.

The properties of a SystemC process, and hence a FIP, imply an interface like that shown in Figure 3.7. The signals are as follows:

Signal	Type	Function
Go	input	Signals to a FIP when to fire. Asserted when inputs have changed. Cannot be asserted while `Busy` is asserted.
Busy	output	Signals that the FIP is firing. In order to preserve the appearance of delta-cycle semantics for signals, the inputs may not change and new values produced by the FIP may not be seen by other FIPs while the FIP is busy. The FIP asserts `Busy` the cycle after `Go` is asserted and deasserts when the FIP has completed. If only a single FPGA cycle is required to complete the firing, `Busy` never need be asserted.
Input(I_i)	input	The data for a given input.
Output(O_j)	output	The data for a given output.
Write(O_j)	output	Signals to the environment that the corresponding output is being written in this FPGA cycle. The output signal is valid only while this signal is asserted.

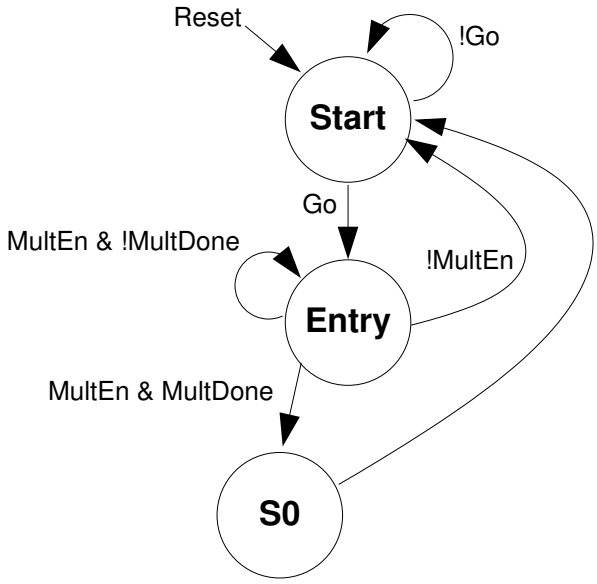
The environment of a FIP is made up of the software and hardware which maintains communication between FIPs by retaining signal values and instructing when FIPs should execute. When the software detects that a SystemC signal has changed it communicates with the hardware to write the inputs of a FIP to the hardware environment and then fire the FIP by asserting the `Go` signal, thus maintaining the edge-triggered property of a SystemC process. The software then waits to read the outputs until the hardware communicates that the `Busy` signal has de-asserted, which occurs when the FIP is finished, allowing non-preemptive behavior to be maintained. The output-inseparable property is maintained because all outputs are read by the software every time the `Go` signal is asserted. The environment must maintain the appearance of non-concurrency by ensuring that the inputs do not change.

FIPs do not need to provide state elements for outputs which are driven directly from state because in SystemC this state is maintained by the signals in the environment. After synthesis, the

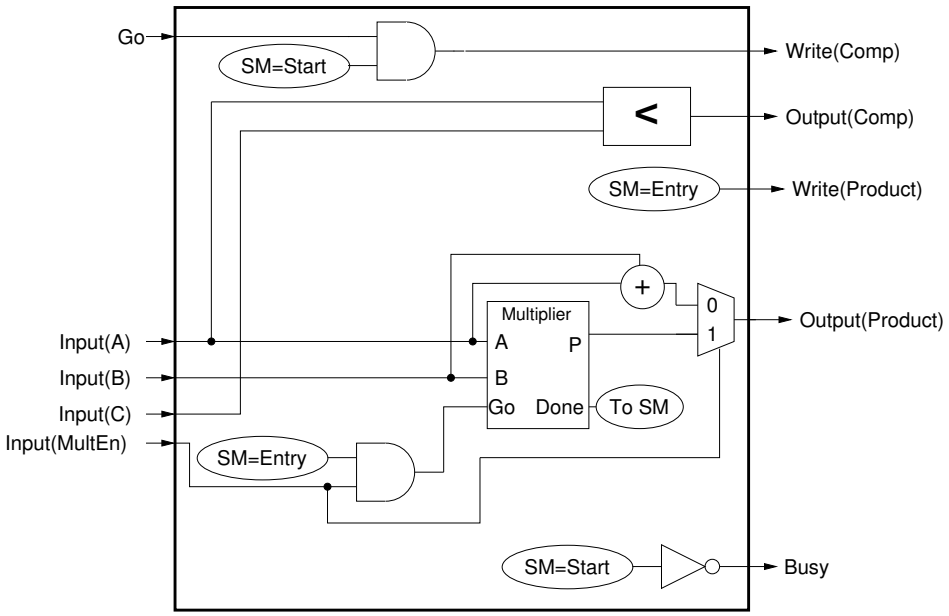
environment (i.e., the logic outside of FIPs) retains this responsibility. Thus the output signals of FIPs derived from sequential processes are actually the "next state" values of those signals.

We will call FIPs derived from combinational processes *combinational FIPs* and FIPs derived from sequential processes *sequential FIPs*. This nomenclature does *not* imply that the FIP itself is implemented as purely combinational or sequential logic.

The SPRI hybrid simulator synthesis tool uses its own internally built, open source HDL synthesizer to generate FIPs from SystemC processes. Figure 3.8 shows the SPRI FIP synthesis engine output for the `ComboModule` process. In this case the synthesis engine has chosen to encode the multiplier as a Booth multiplier with a variable output latency. The state machine waits for the `Go` signal in the `Start` state. Since the `Comp` output can be produced as soon as A and C are available, the synthesis engine has chosen to assert `Write(Q)` in the `Start` state, as soon as `Go` is asserted. In the `Entry` state the block either writes out the sum, which can be calculated in the `Entry` state since the add has a zero cycle latency, or it enables the multiplier and waits for it to complete, while writing out the product every cycle until the result has emerged. Only the last write of the product will have the valid output, but since unlimited writes to an output are permitted the synthesizer may choose to write the incomplete value every cycle as long as the last value written is the correct value. The block then returns to the `Start` state to wait for another firing, either through the `S0` state or directly, depending on if the multiplier was used. The `S0` state is not strictly necessary, but is an artifact of how the SPRI synthesis engine calculates states. In the future a more sophisticated synthesis engine may be capable of eliminating this state, but the interface to the hardware environment need not change to do so.



(a) State machine



(b) Logic diagram

Figure 3.8: The `ComboModule` process synthesized as a FIP.

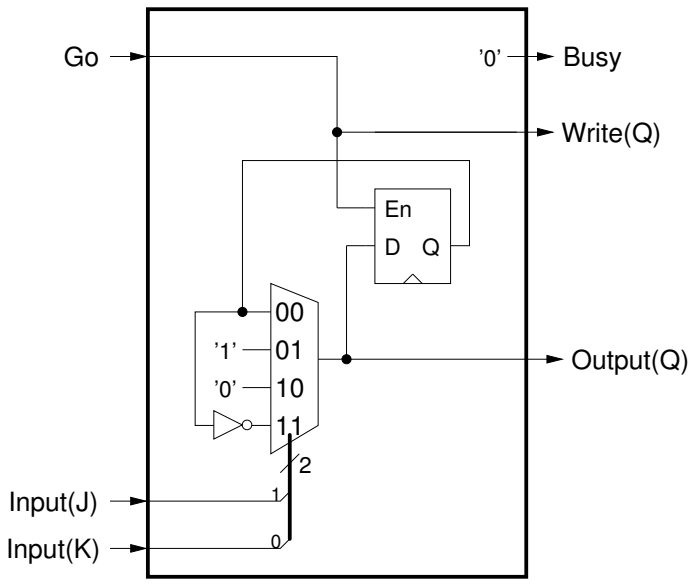


Figure 3.9: The `SeqModule` process synthesized as a FIP.

Figure 3.9 shows the SPRI FIP synthesis engine output for the `SeqModule` process. Since this FIP is capable of computing its outputs in a single cycle no state machine is necessary and the FIP is never busy. The output of the FIP represents the next state value of the `Q` output, not the state for the current cycle. However, there is an inferred flip flop due to the dependence of the current state on the next state. The synthesis engine infers this activity from the writes to the module instance variable `currentQ`. If the block had no dependence on current state to calculate next state (e.g. a D-type flip flop with no clock enable) the synthesis engine would need not infer the state element.

Chapter 4

Composing FIPs

Composition is useful in a hybrid simulator because it reduces round-trip communication from the host to the FPGA. FIPs cannot be directly composed because of the variable completion time for each FIP. As FIPs complete at different times they update their outputs, which in a composed network may be the input to another FIP. This causes FIPs to see changing values of inputs during their execution, which is a violation of the SystemC non-concurrent property. Wrapping a FIP into a primitive LI-BDN, however, can allow us to create a composable network.[1]

In order to achieve LI-BDN wrappers for FIPs two steps are necessary. First, FIPs must be transformed to be compatible with LI-BDNs in much the same way that SSMs need to be transformed into patient SSMs to be compatible with LI-BDNs in order to create a standard interface for wrapping the process in a BDN node. Second, appropriate control signals must be generated in the LI-BDN wrapper. FIPs which have undergone transformation to become LI-BDN-compatible are called *FPGA-implemented LI-BDN-compatible Processes (FLIPs)*.

4.1 Transforming FIPs into FLIPs

FIPs can not be directly inserted into a BDN node because the SystemC process behavior inherently conflicts with BDN node behavior. In order to resolve this conflict three transformations are necessary.

4.1.1 NED vs Output-inseparable Transformation

First, FIPs must be transformed by addressing the inherent conflict between the NED property of LI-BDNs and the output-inseparable property of SystemC processes. The NED property

[1] Once LI-BDNs have been composed, delta-cycle accuracy of the simulator is not maintained. However, microarchitectural simulators generally do not require delta-cycle accuracy and neglecting it is a common optimization technique, e.g. [21].

requires that outputs that have their inputs available must be allowed to produce their values eventually, even if other outputs of a node are not yet ready. The output-inseparable property however requires processes to produce all of their outputs together. In software, the SystemC environment simply keeps firing the process when the inputs change until all of the inputs have stabilized, at which point the values of the outputs represent the final values for that simulation cycle. This means a combinational FIP may fire many times, or even not at all if its inputs do not change, and worse, may write a signal multiple times even in a single firing. This mechanism presents a dilemma for LI-BDNs because a given output must be enqueued exactly once per simulated clock cycle.

The dilemma is resolved by allowing the FIP to be fired multiple times, when the inputs for a given output are ready, but only enqueue an output that has both its inputs ready and has not yet been enqueued. This requirement is satisfied through a requirement in the FLIP interface and additional control added to the FLIP wrapper.

4.1.2 Sequential FLIP Transformation

Second, FLIPs must be modified to remove certain dependencies on the SystemC environment which are not yet supported in hardware. Many of these dependencies remain synthesizable for the present generation of synthesizers. However, the dependence of a sequential process on the environment to maintain current state is trivial to resolve.

The sequential FLIP transformation changes the outputs of a sequential FIP to reflect the current state instead of the next state. As a result, the FLIP must maintain the state of the output internally instead of relying on the environment. Additionally, state elements in a sequential FLIP must be told when it is safe to recalculate state for the next cycle, in the same way that patient SSMs must be told when their state can update. A sequential FLIP has effectively two phases: the production of outputs and state update. The production phase occurs on a new simulation cycle and proceeds until all outputs have been produced and enqueued into FIFOs, which may occur at different times due to FIFO availability. The update phase is essentially what was formerly the 'firing' of the FIP, which may take multiple cycles and begins after all outputs have been produced and all inputs are available. Analysis of the original LI-BDN wrapper shows that patient SSMs have essentially the same behavior; however, their production and update phases may only last a

single cycle. This transformation is accomplished by making a simple change to the synthesizer to produce sequential FLIPs which capture their outputs with state elements when the FLIP is updated and to produce those outputs on the following simulated clock cycle.

4.1.3 Multiple Writes Transformation

The final transformation is necessary due to a conflict in the way in which LI-BDNs and SystemC processes write values to their outputs. A SystemC process is permitted to write to its outputs a limitless number times in a single firing with only the last written value becoming part of the actual simulation state. LI-BDNs however, may only enqueue a single value per simulation cycle. Therefore, if a FIP writes an output multiple times in a firing, then the FLIP must either buffer the values to be written or determine which write post dominates the other writes. Either way, only the last written value may be communicated through the FLIP interface.

It is possible for this conflict to be resolved as part of the LI-BDN wrapper by having the wrapper buffer the writes; however, it is convenient to include this logic as part of the FLIP itself for two reasons. First, doing so presents a more minimal interface to the LI-BDN wrapper. And second, the synthesis engine is more aware of the internal workings of the FLIP than the wrapper, and therefore is in a better position to determine when buffering is necessary and when it is redundant.

4.2 FLIP Interface

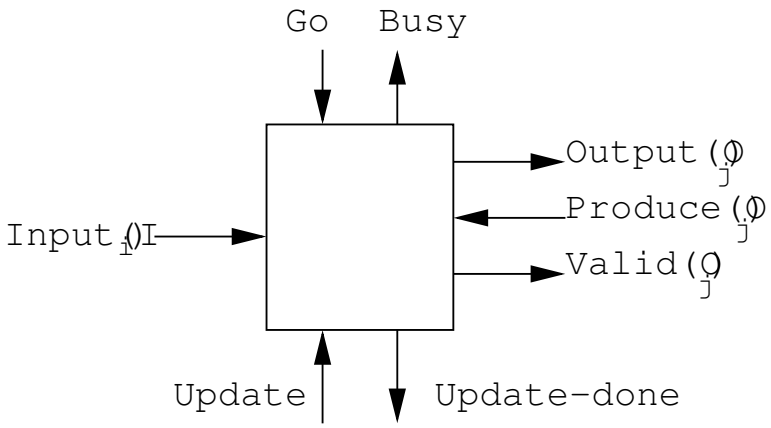


Figure 4.1: FLIP interfaces

The above requirements imply an interface like that shown in Figure 4.1. The new signals are as follows:

Signal	Type	Function
Update	input	Signals the FLIP to update itself to the next state. Asserted when all outputs have been produced and all inputs are available and will remain asserted until `Update-done` is asserted. Only one update should be triggered per assertion.
Update-done	output	A single cycle pulse that signals that the FLIP has completed its transition to the next state.
Produce(O_j)	input	Asserted simultaneously with `Go` to command that the corresponding output should be produced. Cannot be asserted when inputs required to compute the output are not yet available. The value may be arbitrary whenever `Go` is not asserted.
Valid(O_j)	output	Asserted when the corresponding output is ready to be enqueued. May only be asserted once per firing and then only if `Produce` was asserted when the FLIP was fired. Replaces the Write(O_i) signal from the FIP interface. The given output signal is valid only in the cycle this signal is asserted.

The process of calculating the next state of a FLIP is called ”updating” the FLIP and is managed by the `Update`/`Update-done` signals. Updating a FLIP is the analog of enabling a patient SSMs, but updates are allowed to take multiple cycles to calculate the next state. Since a combinational FLIP has no state to update it may simply tie `Update-done` to `Update`. For a sequential FLIP, the calculation of new state, which was previously its firing behavior, is now triggered by `Update`/`Update-done` instead of `Go`/`Busy`. For a sequential FLIP the `Go`/`Busy` signals are used solely for the new firing behavior of the FLIP, which is simply driving outputs from the current state.

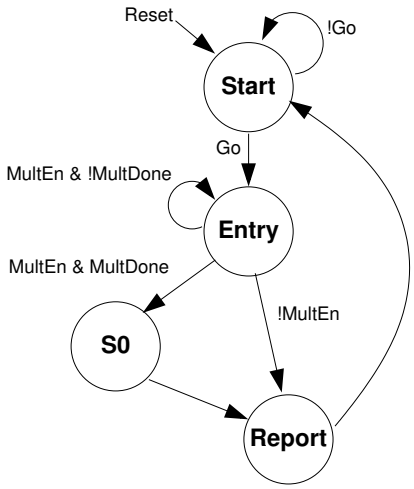
The `Produce`/`Valid` signals are the key to overcoming the NED and output-inseparability dilemma. By requiring that `Valid` be asserted only when `Produce` was previously asserted at `Go`, the LI-BDN wrapper is given the ability to control the production of each output individually.

A FLIP may still compute the value of each output; indeed, output-inseparability implies that it always will. However, outputs which are invalid for this firing of the FLIP are masked out (ignored) because the `Valid` signal will not be asserted. It is possible for the synthesis engine to use the de-assertion of the `Produce` signal during a firing to avoid computing outputs with long latencies since the output will never be valid during the firing.

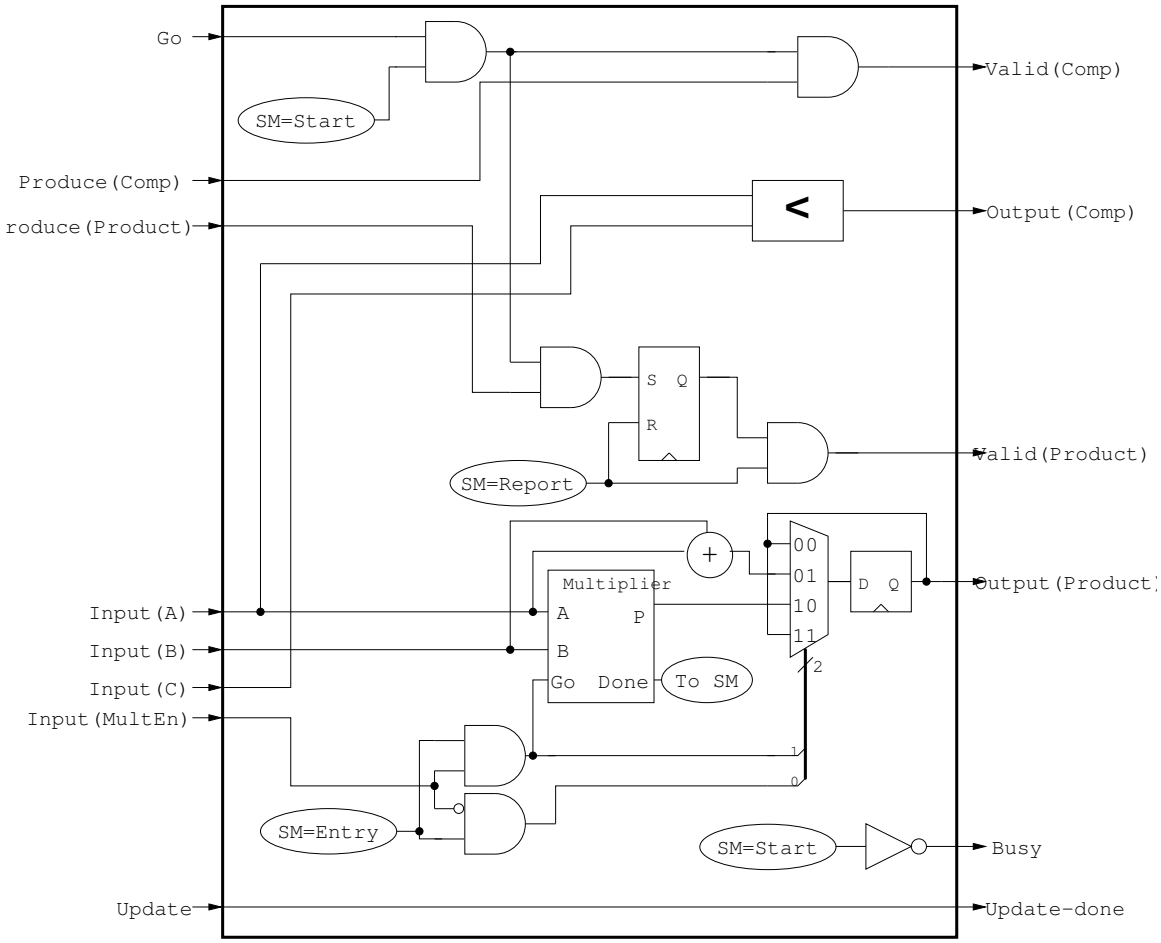
The `Produce` signal also allows the non-concurrency property of SystemC processes to be slightly relaxed in a FLIP. Inputs that are not needed to calculate any of the outputs that are being produced in this firing may change during the firing of a FLIP because these changes, by definition, do not propagate to an output, and therefore do not effect the state of the FLIP. Inputs needed for outputs that are being produced must remain unchanged, and all inputs must be stable and unchanged while updating a FLIP.

Note that this interface forces the FLIP to be responsible for tracking which outputs have been masked and producing `Valid` accordingly. It is alternately possible to place this responsibility on the LI-BDN controller, and just allow the FLIP to assert `Valid` once per firing when it computes the corresponding output. The decision to make the FLIP contain the masking state was made to simplify the LI-BDN controller circuit with only a moderate change to the synthesis engine, to enable shorter block latency by allowing FLIPs to ignore operations that are masked and have long latency, and to allow easy reduction of LI-BDN resource requirements, as will be discussed in Section 5.

Figure 4.2 shows the synthesis engine output for the `ComboModule` process when set to produce FLIPs. The synthesis engine has added the `Report` state in order to simplify its analysis of the multiple write transform. Like the `S0` state from both the FIP and this FLIP implementation, it is expected that further development of the synthesis engine will remove these superfluous states. One potential optimization the synthesis engine could perform is to skip the `Entry`, `S0`, and `Report` states entirely if the FLIP is fired when `Produce(Product)` is not asserted, since these states are only necessary to calculate the `Product` output. The state element for `Valid(Product)` is necessary because `Valid(Product)` may only be asserted when `Produce(Product)` was asserted when the FLIP was fired, so this condition must be retained until `Valid(Product)` is asserted. Additionally since `Valid(Product)` is asserted in a different cycle than the associated output is actually calculated, the output value for



(a) State machine



(b) Logic diagram

Figure 4.2: The `ComboModule` process synthesized as a FLIP.

`Output(Product)` must be retained in a state element until it is formally written out of the block (by asserting `Valid(Product)`) in the `Report` state. No update is necessary for the `ComboModule` FLIP because it is a combinational FLIP so `Update-done` is driven directly from `Update`.

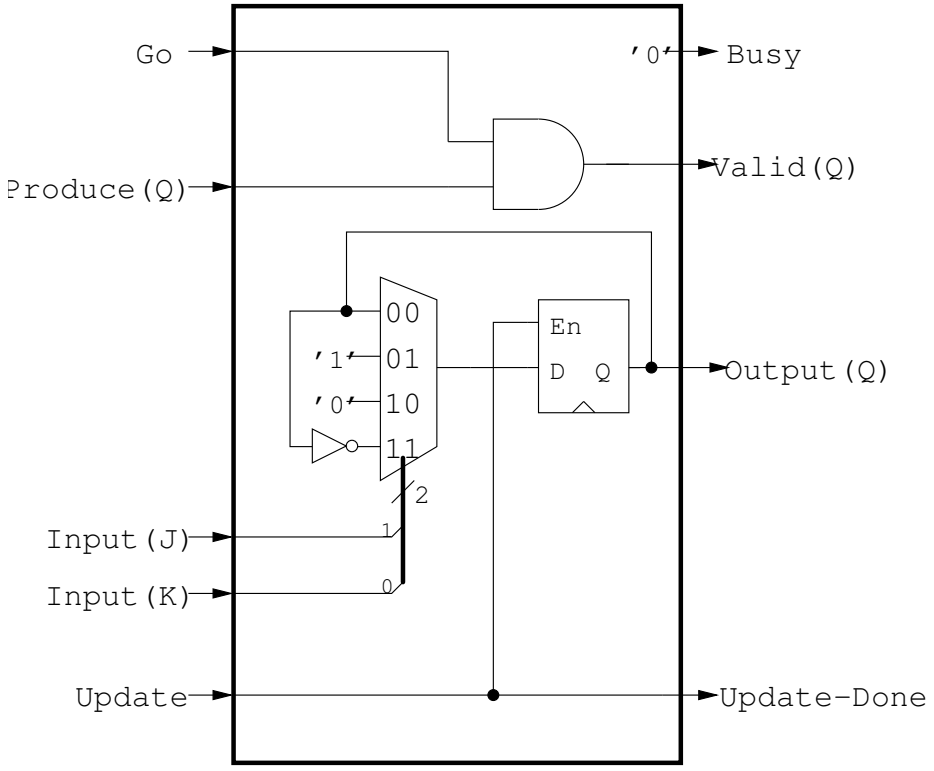


Figure 4.3: The `SeqModule` process synthesized as a FLIP.

Figure 4.3 shows the FLIP implementation of the `SeqModule` process. The most significant change from the FIP implementation is due to the sequential flip transformation. `Output(Q)` now outputs the current state of `Q` instead of the next state. The next state of `Q` is calculated during the update phase of the FLIP, which is only initiated after the current `Q` has been produced. Note that neither `Input(J)` nor `Input(K)` need be available to produce `Output(Q)` because, as a sequential FLIP, `SeqModule` does not require its inputs to report the current state of outputs. Since the output production phase merely drives the output value from the state element, only a single cycle is required and the FLIP never asserts `Busy`. Similarly the update phase requires only

a single clock cycle, so the FLIP can signal it has completed update by asserting `Update-done` in the same cycle it is directed to update.

4.3 LI-BDN Wrappers for FLIPs

The LI-BDN wrapper which surrounds a FLIP to form a primitive LI-BDN must do five things:

1. it must ensure that the NED property is maintained,
2. that outputs are enqueued once and only when they are valid,
3. that the FLIP is triggered until all outputs have been enqueued,
4. that state is updated when a simulated clock cycle is finished, and
5. that the SC property is maintained.

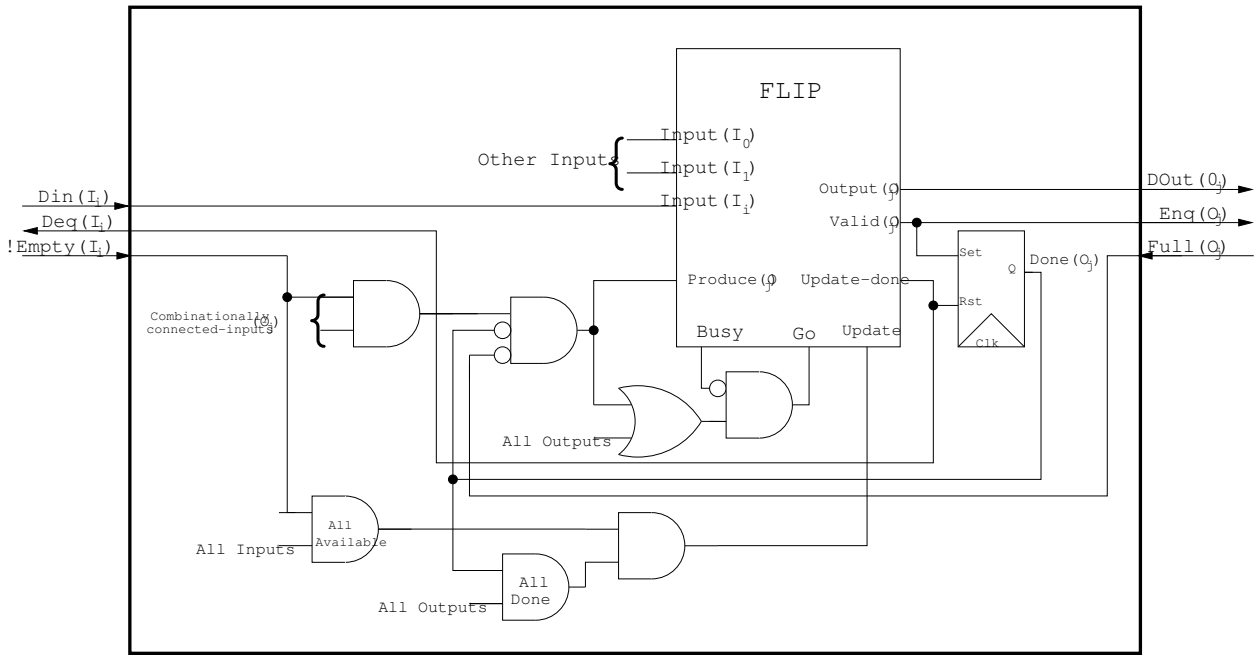


Figure 4.4: An extension to [13]'s LI-BDN control circuit that supports FLIPs.

Figure 4.4 shows the LI-BDN wrapper for a FLIP. The wrapper contains a `Done` flag for each output and combinational logic to generate all the control signals. As in [13], no formal proof of the wrapper's correctness is provided, but instead informal arguments are given.

The first two requirements are met by 1) asserting the `Produce` signal for an output only when the combinationally-connected inputs for the output are available, the output FIFO is not full, and the output has not been previously enqueued in this simulation cycle; and 2) connecting the `Valid` signal directly to the enqueue signal of the output FIFO. The conditions for asserting `Produce` and the FLIP's rules for asserting `Valid` imply that the output will only be enqueued once per simulated clock cycle.

Computing the combinationally-connected relation requires that the synthesis tool know which outputs depend upon which inputs. This knowledge can either be supplied by user annotation of the dependences or by analysis of the SystemC process function.[2] Sequential processes have no combinationally-connected inputs for any output because they do not require any inputs in order to report the output for the current simulation cycle, which was computed as a result of the state update of the previous simulation cycle.

The triggering requirement is met by asserting `Go` whenever the FLIP is not busy and an output can be produced which has not yet been produced. The state update requirement is met by asserting `Update` once all outputs have been enqueued and all inputs are available. The final (SC) requirement is maintained by dequeuing all inputs and clearing all `Done` flags when the `Update-done` signal is completed.

Note that unlike SystemC processes, the inputs to a FLIP may change while it is firing because these changes do not affect the enqueued output values. This is true for sequential FLIPs because outputs do not depend on inputs and state is not updated until all inputs are already available. For combinational FLIPs, outputs are ignored until all inputs needed for their computation are available; inputs which become available while the FLIP is firing do not cause outputs to become unmasked because the `Produce` signal is specifically considered valid only when `Go` is asserted.

[2]Hand-annotation can be tedious, as the user must think about the dependence relation, but the cost can be amortized when modules are reused. Analysis requires compiler techniques for dependence analysis which are beyond the scope of this thesis; development of these techniques is a subject of ongoing investigation.

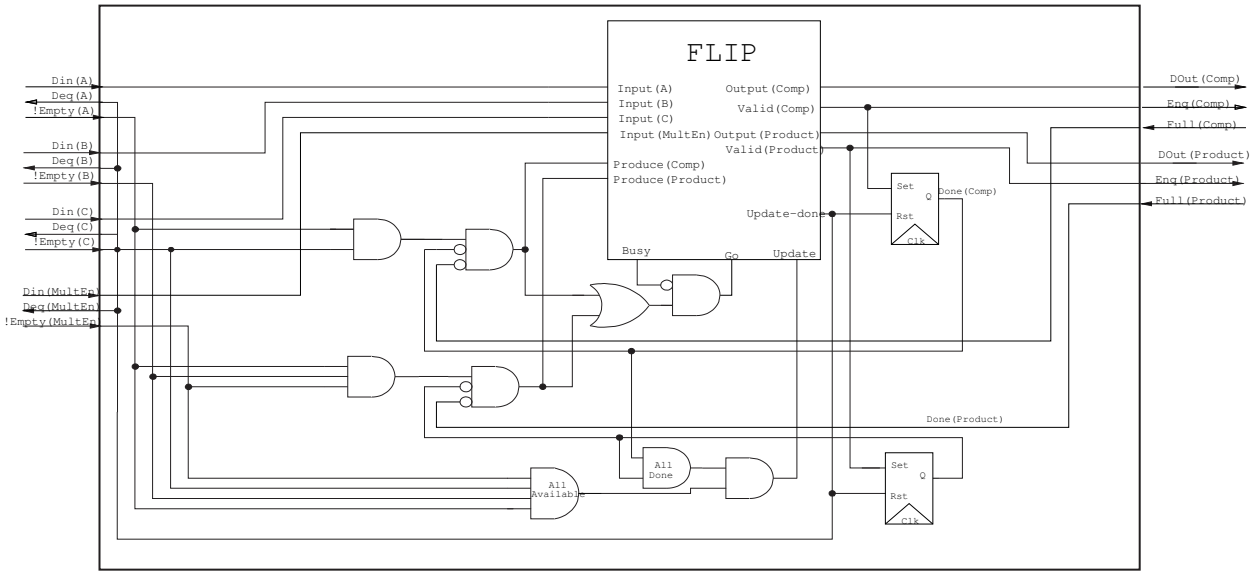


Figure 4.5: The LI-BDN wrapper for the `ComboModule` FLIP

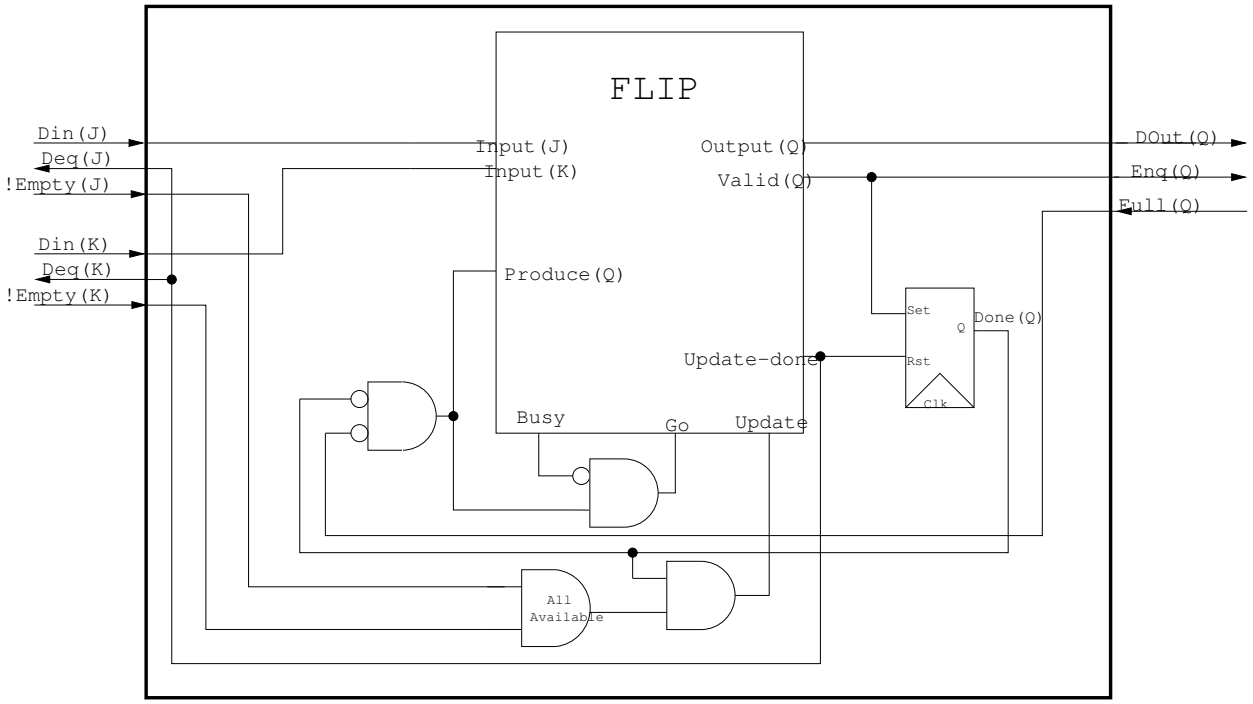


Figure 4.6: The LI-BDN wrapper for the `SeqModule` FLIP

Figure 4.5 and Figure 4.6 show the final LI-BDN wrapper for the `ComboModule` process and `SeqModule` process, respectively. Notice that the `SeqModule` FLIP does not need any inputs to produce `Output(Q)`, and therefore can produce that output anytime the output FIFO has capacity and the output has not already been produced.

The FLIP interface allows the LI-BDN formation stage of a hybrid simulator synthesis tool to make decisions about how to create the LI-BDN network independent of the SystemC to HDL synthesis engine. This separation of concerns is useful because optimization in either section does not require the other section to change, and therefore makes using multiple LI-BDN formation and HDL synthesis techniques for different kinds of FLIPs and networks relatively easy. For example, if designers are particularly concerned with a certain SystemC process they may use a more aggressive synthesis tool, or even manually create the FLIP for that process. As long as it subscribes to the FLIP interface, the LI-BDN formation stage will be able to seamlessly pull in the custom FLIP, form the LI-BDN control logic and interface it with the other FLIPs in the system. On the other hand, if an innovative method for creating the LI-BDN network is produced, any FLIP that subscribes to the interface, no matter how they were created, may be used in this network. The next chapter discusses just such an innovative technique.

Chapter 5

FIFOless Composition

LI-BDNs provide composability, but also require resources for FIFOs and wrapper logic. In some areas of the network nodes may be inherently composable, in which case the addition of the LI-BDN interface may be unnecessary and costly. Resources might be saved if composition can be done without using FIFOs. Such composition would trade off FPGA resources with both clock cycle period and the ability for the primitive LI-BDNs to slip time relative to each other.

In micro-architectural simulation the blocks are modeled after physical circuits, which typically leads to synthesized FLIPs with a manageable number of levels of logic, which in turn leads to a practical expectation for clock rate. Additionally, the capability to slip time may not be very useful if the hybrid simulator must communicate between hardware and software in each simulated clock cycle. Therefore, the advantages to FIFOless composition make it extremely attractive for micro-architectural hybrid simulators because the recovered resources may come at no cost to either clock rate or time slip.

The proposed FLIP interface has been designed to allow FIFOless composition in many situations. However, in order to be composed, the primitive LI-BDN nodes must contain FLIPs that subscribe to the following criteria:

- Composed FLIPs must never assert `Busy`.
- From the former criteria it follows that a composed FLIP's outputs must always be valid on the same FPGA cycle that `Go` and `Produce` are asserted.
- Additionally, a FLIP must be able to update in a single cycle.
- A composed FLIP must be capable of being fired and updated in the same cycle. When a FLIP is simultaneously fired and updated the outputs should produce their pre-update values, i.e., the value it would have produced if it was not being updated this cycle.

These requirements allow all the composed FLIPs to remain in lock step, producing outputs and updating state all together in a single cycle.

The FLIPs are connected together without FIFOs and a new LI-BDN wrapper is formed around them, creating a *composite primitive LI-BDN* which obeys all the properties of a primitive LI-BDN.

Consider Figure 5.1(a) which contains 3 primitive LI-BDNs: A, B, and T. If the FLIPs inside these primitive LI-BDNs can be composed, then FIFOs 3, 4, and 6 may be eliminated and the controller circuits merged, resulting in Figure 5.1(b).

The following rules are used for forming the composite control signals in the LI-BDN wrapper:

1. A composed FLIP has its `Busy` signal ignored (since it is always unasserted) and is constantly being fired by tying its `Go` signal high.

2. In the composite primitive LI-BDN, only FLIP outputs that go to FIFOs retain their `Done` (O_i) register. The removed `Done` flags are replaced with a logic high into the `All Done` AND gate and a logic low into the `Produce` AND gate. An internally-connected FLIP output is thus continuously produced while its inputs are ready, allowing this output to be available as the input to another FLIP.

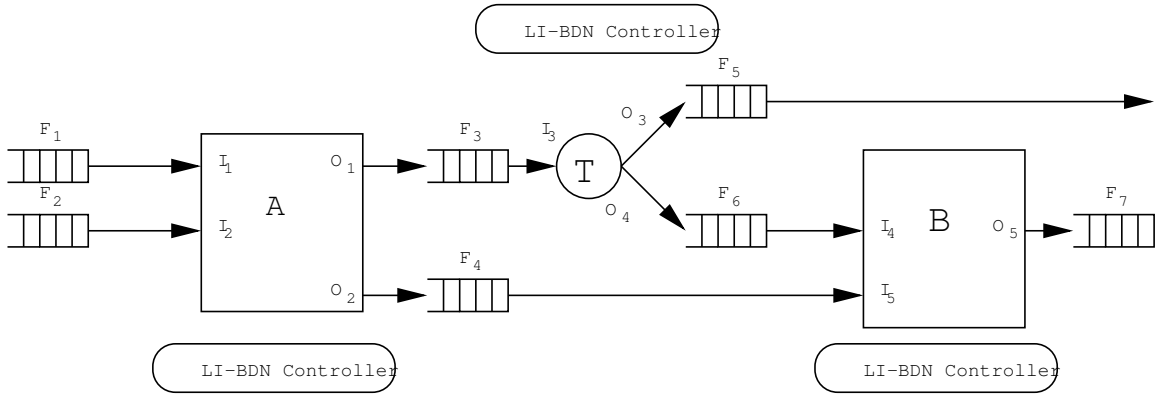
3. For any eliminated FIFO, all uses of the `!Empty` signal are replaced with the `Valid` signal of the output that feeds that FIFO. All uses of the `Full` signal are replaced with logic low.

4. The `All Done` and `All Available` signals in the wrapper must be formed using all the remaining `Done` registers and all the remaining input FIFOs' `!Empty` signals, respectively.

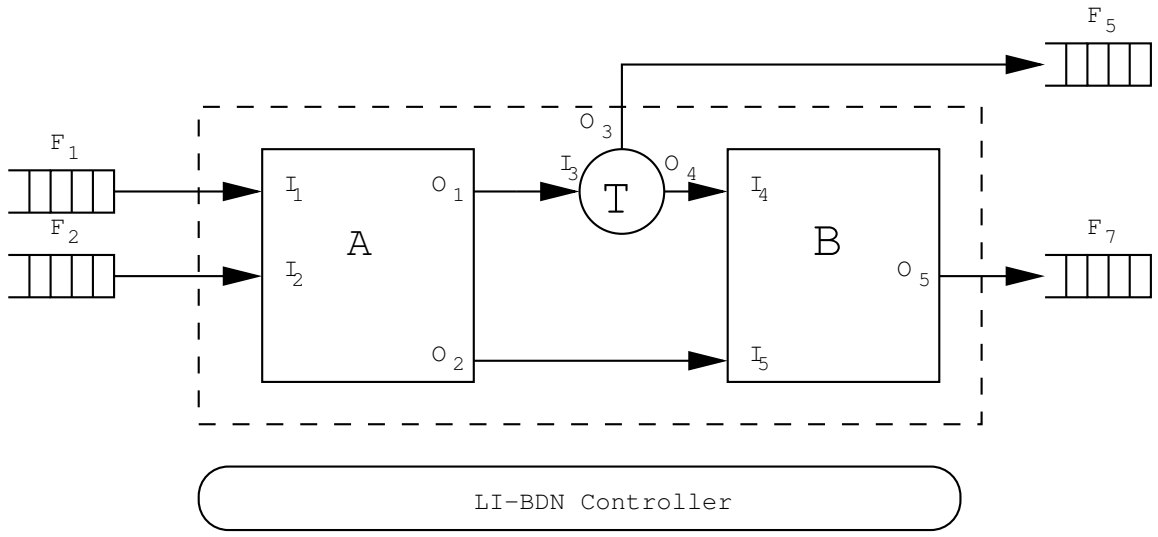
5. The signal which dequeues all input FIFOs and clears all `Done` registers is the AND of all of the `Update-done` signals. [1]

The NED property is maintained via simple signal connectivity without having to reanalyze combinational connectivity. If it were to be reanalyzed, the `Produce` signal for each

[1] Since composed FLIPs must update in a single cycle, the `Update-done` signal on these FLIPs is equivalent to the `Update` signal, which is the same for all composed FLIPs. The HDL synthesizer will reduce the Input dequeue signals of the FIFOs to this value, saving the AND gate. I choose to form the composite node in this way in order to preserve the same basic form as other nodes.



(a) Three independent primitive LI-BDNs



(b) A composite primitive LI-BDN.

Figure 5.1: FIFOless LI-BDN composition

output of the composed FLIPs would need to be asserted when the output FIFO is not full, the output has not already been enqueued in this simulation cycle, and all the inputs in the transitive closure of the combinationally-connected relation are available. However, consider the `Produce` signals for FLIP outputs which are internally connected. The outgoing FIFO has been eliminated and the `Produce` signal only computes whether all combinationally-connected inputs are available. If these inputs are in turn internally connected, then they are available if and only if their combinationally-connected inputs are available. Thus a simple inductive proof shows that the `Produce` and `Valid` signals for internal FLIP outputs are simply the ANDed availability of the transitive closure of the combinationally-connected relation. For the outputs of the composed

FLIPs the `Produce` signals are therefore asserted under the same conditions that a re-analysis would have derived.

The single cycle produce and update restrictions make satisfying the non-concurrency property of SystemC processes trivial. Such FLIPs, when composed, will settle their outputs from the their inputs in a single cycle, and hence will only see a single state for any input, allowing them to be directly composed without violating the non-concurrency property. The same restriction applies to the update phase because during this phase FLIPs depend on their inputs to form their next state outputs. If FLIPs were allowed to take multiple cycles to update then downstream FLIPs in the composed nodes may see intermediate state or the next state too early. This would violate the portion of the non-concurrency property that requires the inputs of a process to not change while it is being fired.

A composed node is guaranteed to be fired at least twice, once to produce outputs for FIFOs and once to produce outputs for downstream updates. Additional firings may occur as Input FIFOs become available at different times. Such action is safe because each output will still only be based on a single state of the inputs, and side effects in combinational FLIPs are illegal. However, the update will only occur once, when all outputs have been enqueued and all inputs are available.

As an example let us consider the FLIPs implied in Figure 5.1 to be defined as the FLIPs shown in Figure 5.2. The T FLIP is a FLIP whose only purpose is to replicate an input into multiple outputs. This is required because a BDN may only drive a single input from a given output. The user does not need to create special "tee" SystemC modules because the synthesis engine can detect fanout situations and automatically generate "tee" nodes to handle these cases.

The fully composed BDN formed by these FLIPs is shown in Figure 5.3. Rule one has caused `Go` to be hard coded to '1' which causes all FLIPs to be fired every cycle and each output's `Valid` signal to be merely the `Produce` signal for that output. From rule two we see that since there are only two output FIFOs in the composite node only two `Done` flags are needed to track when these FIFO have been enqueued, one for the `Output(`O_3`)` FIFO and one for the `Output(`O_5`)` FIFO. The other removed done flags are considered "never done" for the sake of producing the output, and "always done" for the sake of determining when the FLIP can begin updating. Rule two, combined with rule three causing removed FIFOs to never be considered full, leaves the `Produce` signal of each composed output to simply be the "and" of the "availability

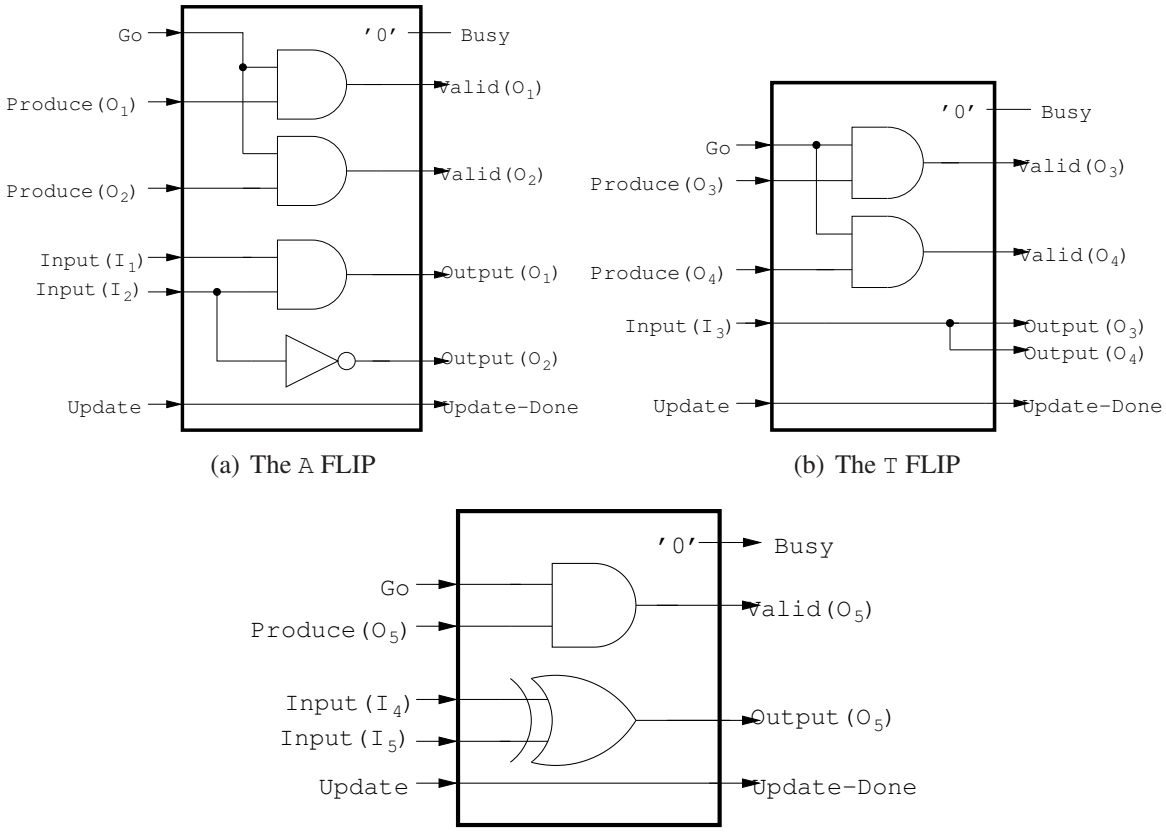


Figure 5.2: An example of FLIPs that are very easy to compose

signal" of the inputs on which the output depends. If the input signal is an input FIFO, the availability signal is the `!Empty` signal for that FIFO. In the case where the input signal is another FLIP output rule 2 makes the availability signal the `Valid` signal of that output. These first 3 rules are what allow the transitive closure of the combinationally-connected relations to propagate through the `Produce` and `Valid` signals.

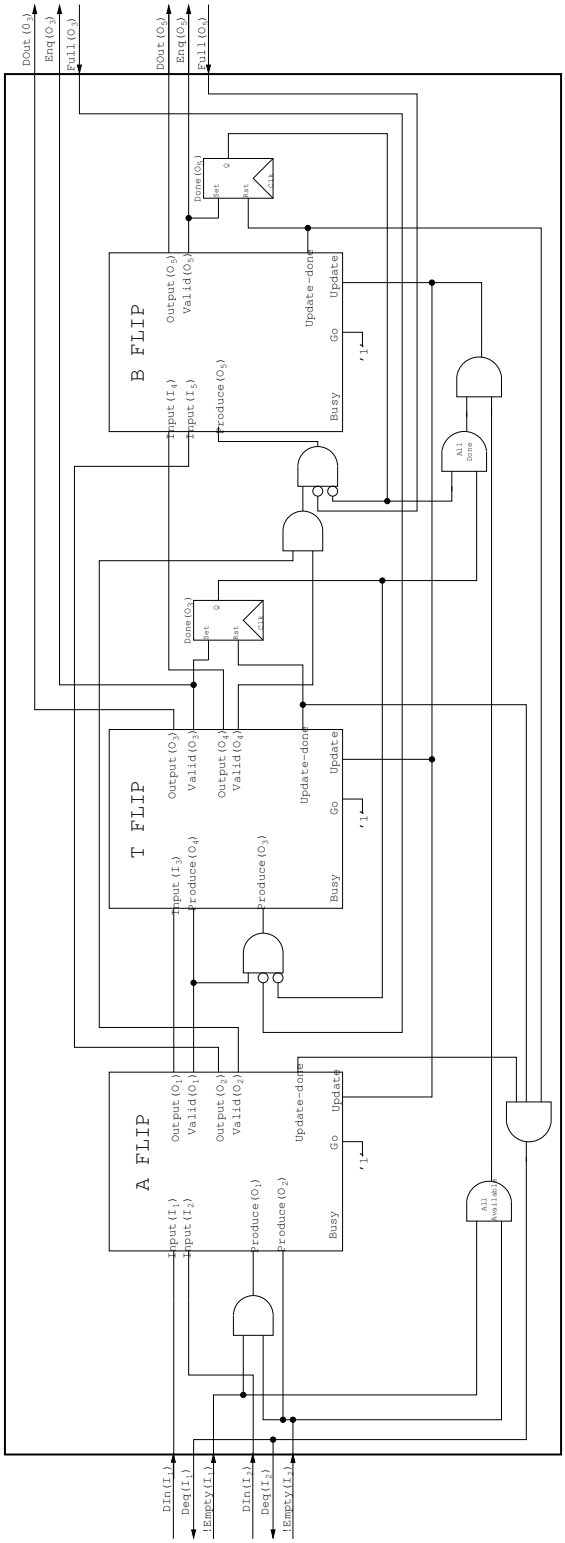


Figure 5.3: The fully composed BDN formed from the merging given in Figure 5.1

We can see the dependency propagating in Figure 5.3. In this composed node we can trace the dependencies through the `Produce` and `Valid` signals, which creates the following system of equations:

$$\text{Valid}(O_5) = \text{Produce}(O_5), \tag{5.1}$$

$$\text{Produce}(O_5) = \neg\text{Done}(O_5) \wedge \neg\text{Full}(O_5) \wedge \text{Valid}(O_4) \wedge \text{Valid}(O_2), \tag{5.2}$$

$$\text{Valid}(O_4) = \text{Produce}(O_4), \tag{5.3}$$

$$\text{Produce}(O_4) = \text{Valid}(O_1), \tag{5.4}$$

$$\text{Valid}(O_1) = \text{Produce}(O_1), \tag{5.5}$$

$$\text{Produce}(O_1) = !\text{Empty}(I_1) \wedge !\text{Empty}(I_2), \tag{5.6}$$

$$\text{Valid}(O_2) = \text{Produce}(O_2), \tag{5.7}$$

$$\text{Produce}(O_2) = \text{!Empty}(I_2). \tag{5.8}$$

By back substituting we find that the condition under which `Output(`O_5`)` can be produced and enqueued is

$$\text{Produce}(O_5) = \neg\text{Done}(O_5) \wedge \neg\text{Full}(O_5) \wedge \text{!Empty}(I_1) \wedge \text{!Empty}(I_2). \tag{5.9}$$

Retaining "tee" nodes in composed FLIPs is still necessary in composed nodes because they allow the `Valid` signals for the outputs to be independently tracked depending on the destination of the output. In this example `Valid(`O_3`)` is dependent on the status of the `Done(`O_3`)` flag as well as the availability of the output FIFO as denoted by the `Full(`O_3`)` signal. `Output(`O_4`)` on the other hand has had its FIFO removed, and therefore must be made available anytime its inputs are valid.

FIFOless composition may have an impact on the minimum clock cycle period because the FIFOs that broke timing paths between combinational FLIPs have been removed. This means that composing all the FLIPs that can be merged may not be the most optimal technique for all LI-BDN networks. There are likely many types of innovative LI-BDN formation stages that could be created that differ only in the FLIPs that they choose to compose. For example, it may be possible to analyze a single cycle combinational FLIP to estimate the delay through the FLIP. This estimation then could be used in a composition algorithm that restricts the total maximum delay through a composed node to be less than some user specified target. However, in the microarchitectural models on which this work was run, such a heuristic was unnecessary because even when all single-cycle nodes were merged, the minimum clock period was not affected. As such, development of a clock cycle period aware composition algorithm is beyond the scope of this thesis.

The FLIP interface has allowed the LI-BDN formation stage to use an innovative technique to form the LI-BDN control logic without modifying the internal behavior of the FLIP. As other HDL synthesis techniques and LI-BDN formation methods are developed they too can use the same interface to ensure compatibility and inter-operability.

Chapter 6

Evaluation

We demonstrate the new LI-BDN wrapping procedure by adding it to the SPRI hybrid simulator synthesis tool flow [11] and then synthesizing a hybrid simulator which uses FLIPs and LI-BDNs to implement SystemC processes. We then run the hybrid simulator and compare its running time to that of a software-only simulator. Note that it is not possible to compare results directly with those of [13], as that work only introduced a procedure without implementing or evaluating it. The current work represents the first attempt to actually create LI-BDNs in a simulator synthesis tool chain.

The original software-only simulator uses SystemC to model a 16-core chip multiprocessor. Each core is a simple five-stage in-order pipeline implementing the PowerPC instruction set. The cache hierarchy is extremely simple and there are no shared caches. The simulator uses a speculative functional-first organization [22]: a single SystemC module calls a functional simulator to simulate instruction-set behavior; this module then communicates information such as branch results, effective addresses, and register specifiers to other SystemC modules which compute the timing by modeling the hardware.

Figure 6.1 shows the modified SPRI synthesis tool flow. The SystemC model and a partitioning specification are the input to the flow. We used a partitioning specification which assigns the functional simulator module to software and the rest of the model to hardware. The LLVM IR-to-VHDL synthesizer produces FLIPs from SystemC processes. The LI-BDN wrapper generator produces primitive LI-BDNs that wrap the controller, FLIP, and FIFOs into a valid network.

We validated the synthesized hybrid simulator by running a multi-threaded benchmark – the FFT kernel from the SPLASH-2 benchmark suite [23] with arguments `-p16` – on the simulator and comparing both the program results and the number of simulated cycles with those reported

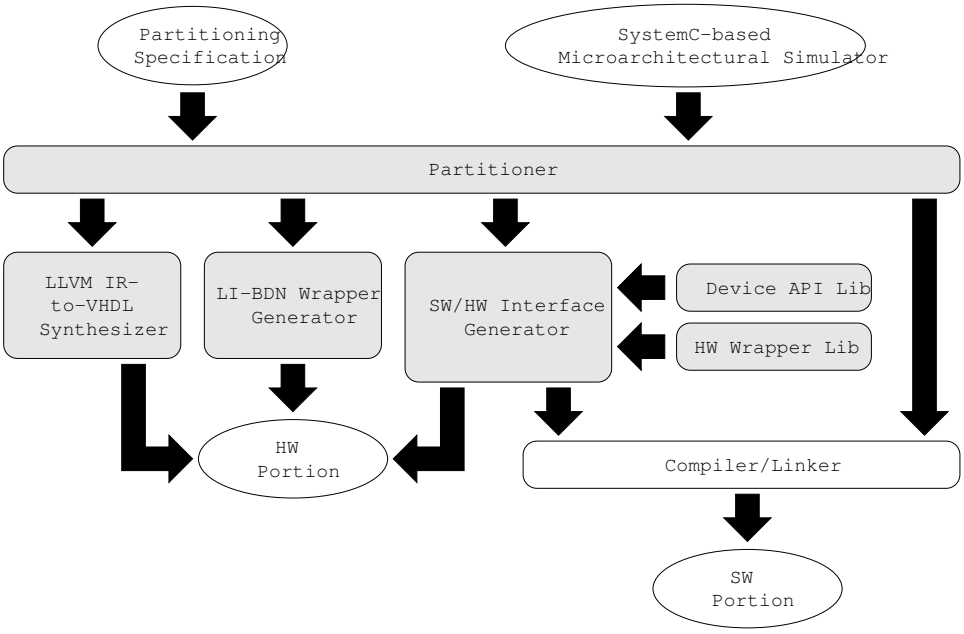


Figure 6.1: Modified SPRI synthesis flow for hybrid microarchitectural simulators

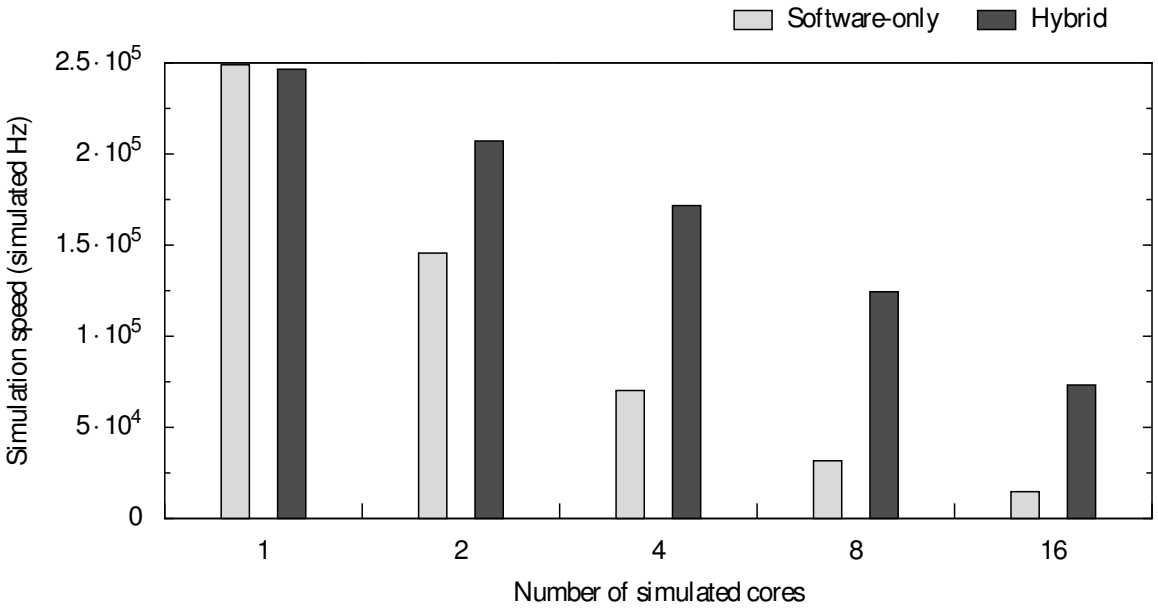


Figure 6.2: Simulation Speeds

by the software-only simulator.[1] Both simulators were run on a DRC 1000 system with a dual-

[1]The cycle counts match to within 0.03%; an exact match is not possible because the benchmark output is expected to differ slightly from run to run.

core AMD Opteron-275 CPU running at 2.1 GHz with 2 GB of system memory and a Xilinx XC4VLX60-11 FPGA fitted on the HyperTransport bus as a coprocessor.

The hybrid simulator achieves a simulation speed of 73.2 KHz while the software-only simulator achieves a speed of 14.7 KHz, for a speedup of 4.97. While this speedup is not particularly large, it is limited by the size and complexity of the model, which in turn limit the amount of computation which can be moved into hardware. As models become larger, there is often more parallelism available to be taken advantage of in the hardware. As they become more complex, the execution time of a software-implemented process usually grows more rapidly than the execution time of its hardware implementation.

We demonstrate the effects of model size on speedup by creating a family of hybrid chip multiprocessor simulators modeling varying numbers of cores. Figure 6.2 shows the simulation speed achieved by these simulators when running the FFT benchmark. As the number of cores increases, the margin between the software-only simulator and hybrid simulator increases, yielding higher speedups.

The primary bottleneck is the communication latency from hardware to software, which is quite high in the DRC 1000 system because all communication from hardware to software requires that the host driver provide a DMA read command to the FPGA and then poll the FPGA's status registers until the DMA completes. For the single-core hybrid simulator, communication occupies a staggering 80% of the execution time and no speedup is achieved. However, as the models become larger, the synthesized HW/SW interface code batches communication whenever possible. Thus communication cost does not grow nearly as rapidly in this family of models as do either SystemC overhead or the aggregate execution time of the models' processes. For the 16-core hybrid simulator, communication is down to 45% of execution time. The net result is that as models become larger, the speedup increases.

Hardware capacity eventually limits the speedup of large models: in this case, a 32-core chip multiprocessor simulation does not fit within the available hardware. Platforms with multiple large FPGAs and/or better-organized communication (e.g., allowing FPGA-initiated transfers) will be necessary to achieve truly impressive speedups.

To evaluate FIFOless composition, we synthesized a version of the simulator where all possible FIFOs which could be removed were. The original simulator without FIFOless composition

utilized 26,118 4-input LUTs and 26,062 slice flip flops. When FIFOless composition was added, the simulator utilized 9,429 4-input LUTs and 13,854 slice flip flops. This represents a 63.9% reduction of LUT resources, and 46.8% reduction in slice flip flops. FPGA clock cycle time was not affected, as the FPGA's critical path remained in the FPGA's interface with the HyperTransport bus.

Chapter 7

Conclusions

Computer architects and designers need fast near-cycle-accurate simulation to evaluate new ideas and guide their exploration of the design space of new systems. Such simulators will be critical in designing the new high-end capability computing systems that scientists will look to in order to solve many of the challenges facing our world. Synthesized hybrid simulation promises to produce such simulators without requiring excessive simulator design effort. However, FPGA implementations of simulator components require composability in order to achieve the best simulator performance.

We have demonstrated a procedure for forming LI-BDNs from the multi-cycle state machines for modeling a single simulation cycle which arise in hybrid simulators. We have demonstrated this procedure within SPRI, a hybrid simulator synthesis framework. We have furthermore shown that a simple technique for composing LI-BDNs without intermediate FIFOs can reduce FPGA resource usage by 60% which will allow even larger and more complex models to be created.

Future work will allow the SPRI synthesizer to further increase the synthesizable subset of SystemC while still maintaining LI-BDN composability. For example side-band channels could be added to the LI-BDN network which would allow FLIPs to communicate with each other, a shared resource, or the host asynchronously. Operations that are impossible or expensive to implement in hardware, such as the `printf` function, could be offloaded to the host or another centralized resource. Additionally, techniques for analyzing SystemC processes for combinationally-connected relations between inputs and outputs will further increase the productivity gains of hybrid simulator synthesizers.

As a result of this work, hybrid simulator synthesizers will be able to provide both timing flexibility and composability in the FPGA implementations. The resulting hybrid simulators

will enjoy less communication overhead and more concurrency, resulting in faster simulators and allowing designers to explore a greater portion of the design space, leading to improved designs.

Bibliography

[1] *The Potential Impact of High-End Capability Computing on Four Illustrative Fields of Science and Engineering*. The National Academies Press, 2008. [Online]. Available: http://www.nap.edu/openbook.php?record_id=12451 1

[2] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson, "A case for FAME: FPGA architecture model execution," in *Proceedings of the 37th International Symposium on Computer Architecture*, 2010, pp. 290–301. 2

[3] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. H. Reinhart, D. E. Johnson, and Z. Xu, "The FAST methodology for high-speed SoC/computer simulation," in *Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 295–302. 2, 3

[4] E. S. Chung, J. C. Hoe, and B. Falsafi, "ProtoFlex: Co-simulation for component-wise FPGA emulator development," in *Proceedings of the 2nd Annual Workshop on Architecture Research using FPGA Platforms*, 2006. 2

[5] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "Quick performance models quickly: Closely-coupled partitioned simulation on FPGAs," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2008, pp. 1–10. 2

[6] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors, "Exploiting parallelism and structure to accelerate the simulation of chip multi-processors," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, February 2006, pp. 29–40. 2

[7] Z. Ruan, K. Rehme, and D. A. Penry, "SPRI: Simulator partitioning research infrastructure," in *3rd Workshop on Architectural Research Prototyping*, 2008. 2

[8] *IEEE Std 1666-2005: IEEE Standard SystemC Language Reference Manual*. IEEE, 2005. 2, 12

[9] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani, "UNISIM: An open simulation environment and library for complex architecture design and collaborative development," *IEEE Computer Architecture Letters*, vol. 6, no. 2, pp. 45–48, July–December 2007. 2

[10] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural exploration with Liberty," in *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, November 2002, pp. 271–282. 2

[11] Z. Ruan and D. A. Penry, "Partitioning and synthesis for hybrid architectural simulators," in *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems*, 2010. 3, 39

[12] M. Pellauer, M. Vijayaraghavan, M. A. Arvind, and J. Emer, "A-Ports: An efficient abstraction for cycle-accurate performance models on FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays '08*, 2008. 3, 5

[13] M. Vijayaraghavan and Arvind, "Bounded dataflow networks and latency-insensitive circuits," in *Proceedings of the 7th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, July 2009. vi, 3, 6, 7, 10, 11, 12, 27, 28, 39

[14] G. Kahn, "The semantics of a simple language for parallel processing," in *Information Processing '74: Proceedings of the IFIP Congress*, 1974, pp. 471–475. 5, 7

[15] E. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May. 5

[16] Arvind, K. Asanović, D. Chiou, J. C. Hoe, C. Kozyrakis, S. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek, "CRI: RAMP: Research accelerator for multiple processors - a community vision for a shared experimental parallel HW/SW platform," RAMP Project, Tech. Rep., 2005. 5

[17] L. P. Carloni, K. L. McMilland, and A. L. Sangiovanni-Vicentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, Sep 2001. 6

[18] S. Sundralingram, "SystemC modeling, synthesis, and verification in CATAPULT C," Mentor Graphics, Tech. Rep., February 2010. 15

[19] "ESL design with the agility compiler for SystemC," Celoxica Software-compiled System Design, 2004. [Online]. Available: www-ti.informatik.uni-tuebingen.de/~systemc/../Presentation-10-SF_3_sullivan.pdf 15

[20] *SystemC Synthesizable Subset 1.3 draft*. OSCI, 2009. 15

[21] D. Gracia Pérez, G. Mouchard, and O. Temam, "A new optimized implementation of the SystemC engine using acyclic scheduling," in *Proceedings of the Design Automation and Test in Europe Conference*, February 2004, pp. 552–557. 20

[22] D. Chiou, H. Angepat, N. A. Patil, and D. Sunwoo, "Accurate functional-first multicore simulators," *IEEE Computer Architecture Letters*, vol. 8, no. 2, July 2009. 39

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, June 1995, pp. 24–36. 39