Bounded Dataflow Networks and Latency-Insensitive Circuits

Muralidaran Vijayaraghavan, and Arvind

Computation Structures Group
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
{vmurali, arvind}@csail.mit.edu

***Abstract*—We present a theory for modular refinement of Synchronous Sequential Circuits (SSMs) using Bounded Dataflow Networks (BDNs). We provide a procedure for implementing any SSM into an LI-BDN, a special class of BDNs with some good compositional properties. We show that the Latency-Insensitive property of LI-BDNs is preserved under parallel and iterative composition of LI-BDNs. Our theory permits one to make arbitrary cuts in an SSM and turn each of the parts into LI-BDNs without affecting the overall functionality. We can further refine each constituent LI-BDN into another LI-BDN which may take different number of cycles to compute. If the constituent LI-BDN is refined correctly we guarantee that the overall behavior would be cycle-accurate with respect to the original SSM. Thus one can replace, say a 3-ported register file in an SSM by a one-ported register file without affecting the correctness of the SSM. We give several examples to show how our theory supports a generalization of previous techniques for Latency-Insensitive refinements of SSMs.**

I. Introduction

Synchronous designs or clocked sequential circuits are very rigid in their timing specifications because the behavior of the system at every clock cycle is specified. Modular refinement of such systems is difficult; if the timing characteristic of a single module is changed then the functional correctness of the whole system has to be re-established. Architects often talk about the benefits of latency-insensitive or decoupled designs. The benefits include greater flexibility in physical implementation because the latency of communication or the number of clock cycles a particular module takes can be changed without affecting the correctness of the whole design. Once the overall design is set up as a collection of latency-insensitive modules, different people or teams can do refinements of their modules independently of others. In this paper we will show how a synchronous specification can be implemented in a latency-insensitive manner using *Bounded Dataflow Networks* (BDNs). In the rest of the paper we will refer to both synchronous specifications and their implementations as *Synchronous Sequential Machines* (SSMs).

BDNs are a class of circuits representing dataflow networks [1], [2] where the nodes of the network are connected by bounded FIFOs. Nodes can enqueue into a FIFO only when the FIFO is not full and dequeue from a FIFO only when it is not empty. In contrast to the cycle-by-cycle synchronous input-output behavior of an SSM, the behavior of a BDN is characterized by the sequence of values that are enqueued in the input FIFOs and the sequence of values that are dequeued from the output queues. We will define what it means to implement an SSM as a BDN, and show how a BDN implementation of an SSM relaxes the timing constraints of the SSM, while preserving its functionality. The theory which we have developed can be used to solve several important implementation problems:

1. The timing-closure problem: Carloni et. al [3]–[6] have proposed a methodology which provides flexibility in changing the communication latency between synchronous modules. Their approach is to start with an SSM and identify some wires whose latency needs to be changed without affecting the overall correctness of the SSM. The circuit is essentially cut in two parts such that the cut includes the wire. Each of these parts is treated as a black box and a wrapper is created for each black box. The wrappers contain shift registers (similar to bounded FIFOs) for each input and output wire, and effectively allow the wire latencies to be changed without affecting the overall correctness. We will show that our approach is a generalization of Carloni's work in two ways. First, in addition to changing the latencies of wires, we can change the timing behavior of any module without affecting the functionality of the original SSM. Second, we allow arbitrary cuts for decomposing SSMs while Carloni's method restricts where cuts can be made.

2. The multiple FPGA problem: The predominant model for programming FPGAs is RTL, *e.g.*, Verilog. There are a number of tools that can generate good implementations for an FPGA provided the design fits in a single FPGA. When the design does not fit in a single FPGA then either the designer modifies the design to reduce its area at the expense of fidelity with respect to the timing of the original design or he tries to decompose the design to run on multiple FPGAs. The latter often involves serious verification issues, in addition to the timing fidelity issues. The implementation on multiple FPGAs is not difficult if the design itself is *latency insensitive* and the modules that are mapped onto different FPGAs are connected using latency insensitive FIFOs. BDNs can be used to implement an SSM in a manner that makes it straightforward to map the resulting BDN onto a multi-FPGA platform and preserve its functional and timing characteristics.

978-1-4244-4807-4/09/$25.00 ©2009 IEEE

3. The cycle-accurate modeling of processors on FPGAs: Our development of BDNs was inspired by HASim [7], [8], an ongoing project at Intel to develop cycle-accurate FPGA simulators for synchronous multi-core processors. The goal is to develop FPGA based simulators that are three orders of magnitude faster than software simulators of comparable fidelity. There are similar efforts underway at other institutions (University of Texas at Austin [9], Berkeley [10] and at IBM Research). So far the Intel group has demonstrated cycle-accurate simulators for in-order and out-of-order pipelines for a single-core with the Alpha ISA. The modules in the simulator communicate via *A-Ports* which represent a fixed-latency communication link. The behavior of the modules is dictated by the following rule: first, all the input A-Ports are read; second, the module does some processing; and finally, all the output A-Ports are written. This rule is identical to Carloni's firing rule. In order to avoid deadlocks, an A-Port network is not allowed to contain 0-latency cycles. This restriction is similar to Carloni's restrictions on cuts. Other groups have experienced deadlocks and have avoided them in an ad-hoc manner [11]. In our work, we focus on developing the rules for the behavior of the modules that guarantees the absence of deadlocks. These rules are captured as restrictions on BDNs and can be enforced by a tool or by the designer of the BDN.

The collective experience of aforementioned projects has shown that one always has to modify the design to implement it efficiently on FPGAs. Some hardware structures such as multi-ported register files and content addressable memories (CAMs) are not well matched for FPGAs. Others, such as low-latency multiply and divide can take up enormous FPGA resources. In order to conserve FPGA resources, there is a need to implement hardware structures which take one cycle in the target model to take multiple clock cycles on FPGAs by time-multiplexing the resources. The BDN theory developed in this paper can help one implement designs to run efficiently on FPGAs and still be able to reconstruct the timing of the original model. The problem essentially translates into being able to make latency-insensitive modular refinements.

As an alternative to starting with a rigid SSM like specification, BDNs can also be used to express a design directly. The specifications of complex synchronous digital systems have a built-in tension. On one hand one wants precise timing specifications to study performance but one also wants the flexibility of changing the timing to study its effect. We think that the specification of a design in terms of BDNs may help alleviate this tension and may be a much better starting point than RTL or other formal and informal ways of describing SSMs. *In this paper, however, we focus exclusively on the narrow technical question of implementation of SSMs in terms of latency-insensitive BDNs such that the timing characteristics of the original SSM can be reconstructed accurately.*

Paper Organization: We give a brief recap of SSMs in Section II. We then introduce BDNs in Section III. We give several examples to show what it means for a BDN to implement an SSM. We discuss some of the properties of nodes of BDN networks and also give a procedure to convert any SSM into a BDN. In Section IV, we discuss LI-BDNs which are composable, deadlock-free, latency-insensitive implementations of SSMs. In Section V, we describe a methodology for developing latency insensitive designs and give examples modular refinements. We offer some conclusions in Section VI.

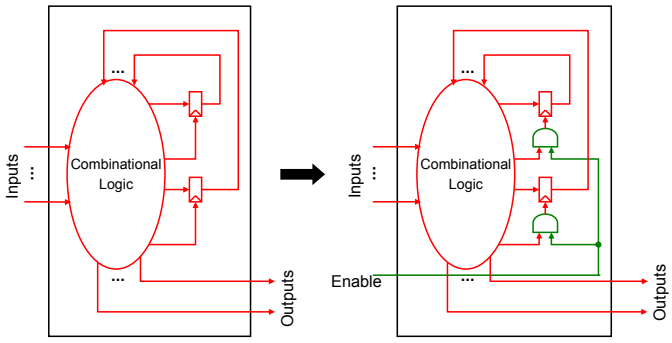


Fig. 1: Converting a normal SSM into a patient SSM

II. Synchronous Sequential Machines

We begin with some definitions and notations.

Definition 1. (Synchronous Sequential Machine (SSM))
An SSM is a network of *combinational operators* or gates such as AND, OR, NOT, and *state elements* such as registers, provided the network does not contain any cycles which has only combinational elements. □

Notation: $I = \{I_1, I_2, \ldots, I_{k_I}\}$, $O = \{O_1, O_2, \ldots, O_{k_O}\}$ and $s = \{s_1, s_2, \ldots, s_{k_s}\}$ represent the inputs, outputs and states (registers) of an SSM, respectively.

$I_i(n)$ represents the value of input I_i during the n^{th} cycle. $I(n)$ represents the values of all inputs during the n^{th} cycle. Similarly, $O_j(n)$ represents the value of output O_j during the n^{th} cycle and $O(n)$ represents the values of all the outputs in the n^{th} cycle.

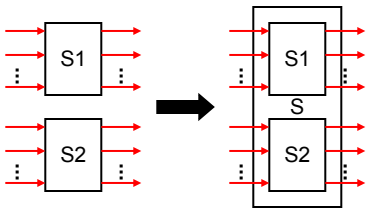
$s(n)$ represents the value of all the registers during the n^{th} cycle. $s(1)$ represents the initial value of all these registers, *i.e.*, the value of the states during the first cycle. □

A. Operational Semantics of SSMs

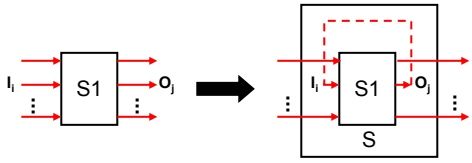
Assume that the initial values for all the registers of the SSM are given. An SSM computes as follows: the outputs of a combinational block at time $n \geq 1$ is determined by the value of its inputs at time n, and the value of a register at time n is determined by its inputs at time $n-1$. Mathematically, it is assumed that combinational gates compute in zero time.

B. Patient SSMs

We will show later that in order to implement these SSMs in a latency-insensitive manner, we need a global control over the update of all the state elements of an SSM. One often represents registers so that they have a separate enable signals and the state of the register changes only when the enable signal is high. We can provide global control over an SSM



(a) Parallel composition of SSMs ($S_1 + S_2$)



(b) Iterative composition of SSMs ($(I_i, O_j) \cdot S_1$)

Fig. 2: Composition of SSMs

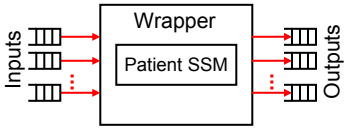


Fig. 3: A primitive BDN

by conjoining the enable signal of each register with a global enable signal. We will call an SSM with such a global enable as a *patient SSM* [4]. No state change in a patient SSM can take place without the global enable signal. Any SSM can be transformed into a patient SSM as shown in Figure 1.

C. A structural definition of SSMs

Any SSM can be defined structurally in terms of parallel and iterative compositions of SSMs. The starting point of the recursive composition is the set of combinational gates, forks and registers. Since we do not allow pure combinational cycles in SSMs we need to place some restrictions on the iterative composition.

Definition 2. (Structural definition of SSMs)

1) Combinational gates, forks and registers are SSMs.
2) If S_1 and S_2 are SSMs, then so is the *parallel composition* of S_1 and S_2, written as $S_1 + S_2$, (Figure 2a).
3) If S_1 is an SSM, then so is the (I_i, O_j) *iterative composition* of S_1 written as $(I_i, O_j) \cdot R_1$, provided there is no combinational path from I_i to O_j, (Figure 2b). □

We will use structural definitions in our proofs.

III. Bounded Dataflow Networks

A. Preliminaries

Bounded Dataflow Networks (BDNs) are Dataflow Networks [1], [2] where nodes are connected by bounded FIFOs of any size ≥ 1. The nodes of the network, which we refer to as *primitive* BDNs, implement SSMs (Figure 3). A FIFO can be enqueued only when it is not full and dequeued only when it is not empty (Figure 4). We do not draw the control wires associated with the FIFOs in figures to avoid unnecessary clutter. All FIFOs start out empty. At most one node can enqueue in a given FIFO and at most one node can dequeue from a given FIFO. BDNs also cannot contain the equivalent of combinational loops; a restriction that we define formally later.

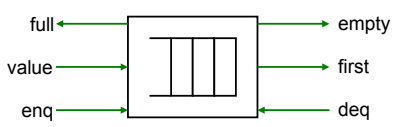


Fig. 4: A FIFO interface

Throughout the paper we will make the assumption of *infinite source* for each input FIFO and *infinite sink* for each output FIFO. The *infinite source* assumption implies that there is an infinite supply of inputs and an input FIFO can be supplied a value whenever it is not full. The *infinite sink* assumption implies that whenever a value is present in an output FIFO it can be dequeued anytime.

Definition 3. (Deadlock-free BDN)
Assuming all input FIFOs are connected to infinite sources and all outputs are connected to infinite sinks, a BDN is said to be *deadlock-free* iff a value is enqueued into each input FIFO then eventually a value will be enqueued into each output FIFO and dequeued from each input FIFO. □

We now define what it means for a BDN to implement an SSM.

Notation: $I_i(n)$ represents the n^{th} value enqueued into I_i. $I(n)$ represents the n^{th} values enqueued into all inputs. Similarly, $O_j(n)$ represents the n^{th} value enqueued into O_j. $O(n)$ represents the n^{th} values enqueued into all outputs. n does not correspond to the nth cycle in the BDN. For example, values $I(n)$ and $I(n+1)$ can exist simultaneously in a BDN unlike an SSM. □

Definition 4. (BDN partially implementing an SSM)
A BDN R partially implements an SSM S iff

1) There is a bijective mapping between the inputs of S and R, and a bijective mapping between the outputs of S and R.
2) The output histories of S and R matches whenever the input histories matches, *i.e.*,

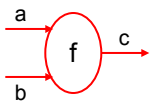
$$\forall n > 0,$$
$$I(k) \text{ for } S \text{ and } R \text{ matches } (1 \leq k \leq n)$$
$$\Rightarrow O(j) \text{ for } S \text{ and } R \text{ matches } (1 \leq j \leq n)$$

□

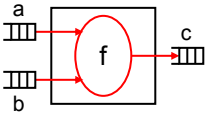
Definition 5. (BDN implementing an SSM)
A BDN R *implements* an SSM S iff R partially implements S and R is deadlock-free. □

Note that there may be many BDNs which implement the same SSM.

One often thinks of a large SSM in terms of a composition of smaller SSMs. We will first tackle the problem of implementing an SSM as a single BDN node and then later discuss the parallel and iterative compositions of BDNs in a manner similar to the composition of SSMs (Section II-C).

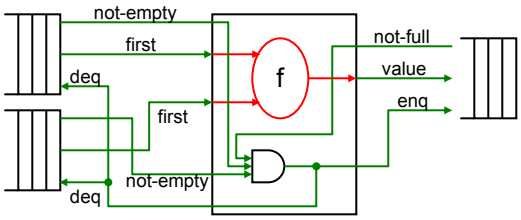


(a) SSM for a Gate




```
rule GateOutC
  when (¬a.empty ∧ ¬b.empty ∧ ¬c.full)
  ⇒ c.enq(f(a.first, b.first)); a.deq; b.deq
```

(b) BDN for a Gate



(c) The synchronous circuit implementing the BDN gate

Fig. 5: SSM and a BDN for a Gate

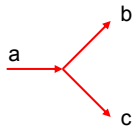
B. Definition and Operational semantics of primitive BDNs

We specify the semantics of primitive BDNs (and later, of all BDNs) using *rules* or *guarded atomic actions* [12], [13]. Given a set of state elements (*e.g.*, registers, FIFOs), a rule specifies a guard, *i.e.*, , a predicate, and the next values for some of the state elements. Execution of a rule means that if the guard is true then the specified state elements can be updated. However, it is not necessary to execute a rule even if its guard is true. Several rules can be used to describe the behavior of one BDN. Rules are atomic in the sense that when a rule is executed, all the state elements whose next state values the rule specifies must be updated before any other rule that updates the same state can be executed. The behavior of a BDN described using rules can always be explained in terms of some sequential execution of rules. Algorithms and tools exist to synthesize the set of atomic rules into efficient synchronous circuits [14].

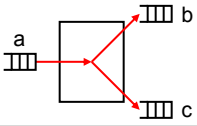
We next show BDN implementations of some SSMs.

C. Examples of primitive BDNs

1) Gate: Figure 5b shows a primitive BDN to implement the SSM gate shown in Figure 5a. `f` is a combinational circuit. The primitive BDN must consume an input value from each input FIFO before producing an output in its output FIFO. The behavior of the gate is defined formally using the rule which says that the rule can execute only when neither input FIFO is empty and the out FIFO is not full. When the rule executes it dequeues one input from each of the input FIFO and enqueues the value $f(a_1, b_1)$ in the output FIFO. All this testing and updates have to be performed atomically. The reader can test that this primitive BDN definition implements the gate correctly according to Definition 5.

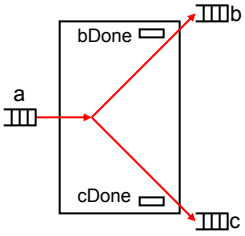


(a) SSM for a Fork




```
rule ForkOutBAndC
  when (¬a.empty ∧ ¬b.full ∧ ¬c.full)
  ⇒ b.enq(a.first); c.enq(a.first); a.deq
```

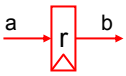
(b) BDN for a fork




```
rule ForkOutB
  when (¬a.empty ∧ ¬b.full ∧ ¬bDone)
  ⇒ b.enq(a.first); bDone ← True
rule ForkOutC
  when (¬a.empty ∧ ¬c.full ∧ ¬cDone)
  ⇒ c.enq(a.first); cDone ← True
rule ForkFinish
  when (¬a.empty ∧ bDone ∧ cDone)
  ⇒ bDone ← False; cDone ← False; a.deq
Initial
  bDone ← False; cDone ← False
```

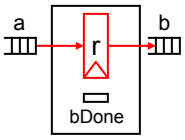
(c) Another BDN for a fork

Fig. 6: SSM and BDNs for a fork




```
r(0) = r0
```

(a) SSM for a register




```
rule RegOutB
  when (¬b.full ∧ ¬bDone)
  ⇒ b.enq(r); bDone ← True
rule RegFinish
  when (¬a.empty ∧ bDone)
  ⇒ r ← a.first; a.deq
    bDone ← False
Initial
  r ← r0; bDone ← False
```

(b) BDN for a register

Fig. 7: SSM and BDN for a register

A circuit to implement this rule is shown in Figure 5c. Notice this whole circuit is itself an SSM, and can be generated by the Bluespec compiler from the atomic rule specification. *Whenever we talk about an SSM corresponding to a BDN, we mean the SSM a BDN implements, and not the SSM which is the synchronous implementation of the BDN.*

2) Fork: Figure 6a shows an SSM fork and Figures 6b and 6c show two different primitive BDNs that implement the fork. The rule for the fork in Figure 6b will not execute if

either of the output forks is full. While the rules for the fork in Figure 6c can enqueue in either of the output forks as and when that output becomes non-full. The `Done` flags ensure that enqueuing in each output FIFO can happen only once for each input and the input can be dequeued only when both the outputs have accepted the input.

Each of these primitive BDNs implements the SSM fork correctly but exhibit different operational properties - for example the fork in Figure 6c can tolerate more slack and may result in better performance.

3) Register: Figure 7a shows an SSM register and Figure 7b shows a primitive BDN that implements it. In the primitive BDN, the value in input `a` is written into register `r` and consumed only after the output `b` has accepted the previous value in the register.

In an SSM, at every clock cycle all the state elements in the SSM get updated, and all the output values are obtained. We can see in the above examples that the clock cycle of the SSM is transformed into dequeue of all the inputs and enqueue of all the outputs in a primitive BDN, and the state elements in the primitive BDN corresponding to the SSM are updated exactly once during this period. This ensures that each primitive BDN in the examples implements the corresponding SSM according to Definition 5.

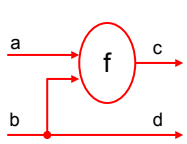
We will later describe a procedure to implement any SSM as a primitive BDN.

D. Properties of primitive BDNs

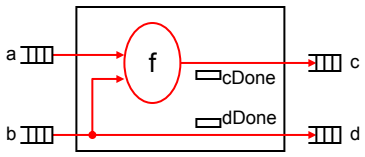
We have already seen that several different primitive BDNs can implement an SSM. Besides their differences in performance, these primitive BDNs behave differently when composed with other BDNs to form larger BDNs. We illustrate this via some examples.

1) The No-Extraneous Dependency (NED) property: Figure 8 shows an SSM and four different primitive BDN implementations for the SSM. Implementation #1 is perhaps the most straightforward; it waits for all the inputs to become available and all the output FIFOs to have space and then simultaneously produces both outputs and consumes both inputs. Contrast this with implementation #2 which is most flexible operationally; for each output it only waits for the inputs actually needed to compute that output. It dequeues the inputs after both the outputs have been produced. We need the `Done` flags to ensure that each output is produced only once for each set of inputs. Implementations #3 and #4 are variants of this and contain some extraneous input or output dependencies as compared to implementation #2. For example, implementation #3 unnecessarily waits for the availability of input `a` to produce output `d` while implementation #4 unnecessarily waits for the availability of space in output FIFO `d` to produce output `c`.

Such extraneous dependencies can create deadlocks when this primitive BDN is used as a node in a bigger BDN. Consider the composition shown in Figure 9b which implements the SSM shown in Figure 9a which can be formed by connecting `d` to `a` in Figure 8a. The BDN in Figure 9b will deadlock if



(a) An example SSM



(b) Figure for BDN implementations #1 to #4

```
rule Out
  when (¬a.empty ∧ ¬b.empty ∧ ¬c.full ∧ ¬d.full)
  ⇒ c.enq(f(a.first, b.first));
    d.enq(b.first); a.deq; b.deq
```

(c) BDN implementation #1 (Does not use `Done` registers)

```
rule OutC
  when (¬a.empty ∧ ¬b.empty ∧ ¬c.full ∧ ¬cDone)
  ⇒ c.enq(f(a.first, b.first)); cDone ← True
rule OutD
  when (¬b.empty ∧ ¬d.full ∧ ¬dDone)
  ⇒ d.enq(b.first); dDone ← True
rule Finish
  when (¬a.empty ∧ ¬b.empty ∧ cDone ∧ dDone)
  ⇒ a.deq; b.deq; cDone ← False; dDone ← False
Initial
  cDone ← False; dDone ← False
```

(d) BDN implementation #2

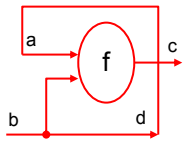
```
rule OutD
  when (¬b.empty ∧ ¬a.empty ∧ ¬d.full ∧ ¬dDone)
  ⇒ d.enq(b.first); dDone ← True
```

(e) BDN implementation #3 (Same as #2 except for rule `OutD`

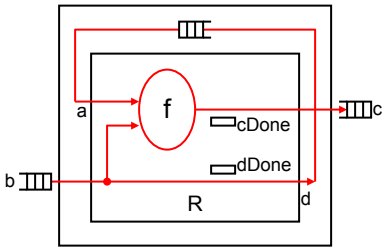
```
rule OutC
    when (¬a.empty ∧ ¬b.empty ∧
          ¬c.full ∧ ¬cDone ∧ ¬d.full)
    ⇒ c.enq(f(a.first, b.first)); cDone ← True
```

(f) BDN implementation #4 (Same as #2 except for rule `OutC`)

Fig. 8: An example to illustrate the NED property



(a) SSM composition



(b) BDN implementing the composition

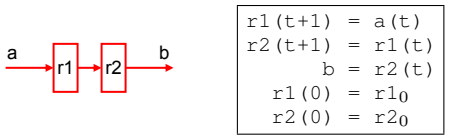
Fig. 9: Illustrating deadlock when composing BDNs without NED property

R uses implementation #1 or #3 because `a` becomes available only when `d` is produced. Using implementation #4 for R will also cause a deadlock if `d` is a single element FIFO, since FIFO `d` has to have space for rule `Out` to consume `a`. If we do not want our BDN implementations to depend upon the size of various FIFOs then even implementation #4 is not satisfactory.

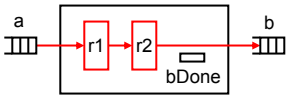
We now define the NED property formally:

Definition 6. (Combinationally-connected relation for primitive BDNs)

For any output O_i of a primitive BDN R which implements SSM S, Combinationally-connected(O_i) is the inputs of R

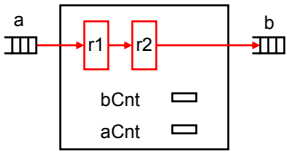


(a) An example SSM




```
rule OutB
  when (¬b.full ∧ ¬bDone)
  ⇒ b.enq(r2); bDone ← True
rule Finish
  when (bDone ∧ ¬a.empty)
  ⇒ r2 ← r1; r1 ← a.first; a.deq;
      bDone ← False
Initial
  r1 ← r1₀; r2 ← r2₀;
  bDone ← False
```

(b) BDN implementation #1




```
rule Out1
  when (¬b.full ∧ bCnt = 0)
  ⇒ b.enq(r2); bCnt ← 1
rule Out2
  when (¬b.full ∧ bCnt = 1)
  ⇒ b.enq(r1); bCnt ← 2
rule In1
  when (¬a.empty ∧ bCnt = 2 ∧ aCnt = 0)
  ⇒ r2 ← a.first; a.deq; aCnt ← 1
rule In2
  when (¬a.empty ∧ bCnt = 2 ∧ aCnt = 1)
  ⇒ r1 ← a.first; a.deq; bCnt ← 0; aCnt ← 0;
```

(c) BDN implementation #2

Fig. 10: An example illustrating the SC property

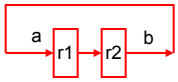
corresponding to those inputs of S that are combinationally connected to the output O_i in S. □

Definition 7. (No-Extraneous Dependencies (NED) property for primitive BDNs)

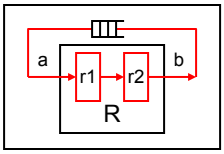
A primitive BDN has the NED property if all output FIFOs have been enqueued at least $n-1$ times and for each output O_i, and if all the FIFOs for the inputs in Combinationally-connected(O_i) are enqueued n times, and all other input FIFOs are enqueued at least $n-1$ times, then O_i FIFO must be enqueued n times. □

According to this definition only implementation #2 satisfies this property.

2) The Self-Cleaning (SC) property: Figure 10 shows an SSM and two different primitive BDN implementations. Both the implementations obey the NED property. In implementation #1, after an output is produced, the `Done` flag is set. Then the input is consumed and the state is updated accordingly. In implementation #2, two outputs have to be produced, and



(a) SSM composition



(b) BDN implementing the composition

Fig. 11: Illustrating deadlock when composing BDNs without SC property

then two inputs are consumed. The `Cnt` counters keeps track of how many outputs are produced and how many inputs are consumed, and they get reset once two outputs are produced and two inputs are consumed.

Implementation #2 does not dequeue its inputs every time an output is produced. This can create deadlocks when this primitive BDN is used as a node in a bigger BDN. Consider the composition shown in Figure 11b, which implements the SSM shown in Figure 11a which can be formed by connecting `b` to `a`. If `b` is a single element FIFO, then the implementation #2 for R will cause a deadlock as it can not dequeue inputs unless two outputs are produced, but there is no space to produce two outputs.

We now define the SC property formally:

Definition 8. (Self-Cleaning (SC) property for primitive BDNs)

A primitive BDN has the SC property, if when all the outputs are enqueued n times, all the input FIFOs must be dequeued n times, assuming an infinite source for each input. □

According to this definition only implementation #1 is self-cleaning.

Definition 9. (Primitive Latency-Insensitive (LI) BDNs)

A primitive BDN is said to be a primitive LI-BDN if it has the NED and the SC properties. □

Note that according to this definition of primitive LI-BDNs, the fork in Figure 6b is not an LI-BDN while the fork in Figure 6c is. As it turns out both the BDN forks work well under composition because both the outputs have the same Combinationally-Connected inputs. We could have given a more complicated definition of the NED property which would admit the fork in Figure 6b also as an LI-BDN. For lack of space we won't explore this more complicated definition of the NED property.

E. Implementing any SSM as a primitive LI-BDN

We will now describe a procedure to implement any SSM as a primitive LI-BDN. Let the SSM be described as follows:

$$\begin{aligned} O_i(t) &= f_i(s(t),\ ICC_{i_1}(t),\ldots,\ ICC_{i_k}(t)) \\ s(t+1) &= g(s(t),\ I_1(t),\ldots,\ I_{n_I}(t)) \\ s(1) &= s_0 \end{aligned}$$

where $\{ICC_{i_1}, \ldots ICC_{i_k}\}$ are inputs combinationally connected to O_i

We associate a `done`$_i$ flag with every output O_i. These flags initially start out as *False*. We have a rule for each output O_i

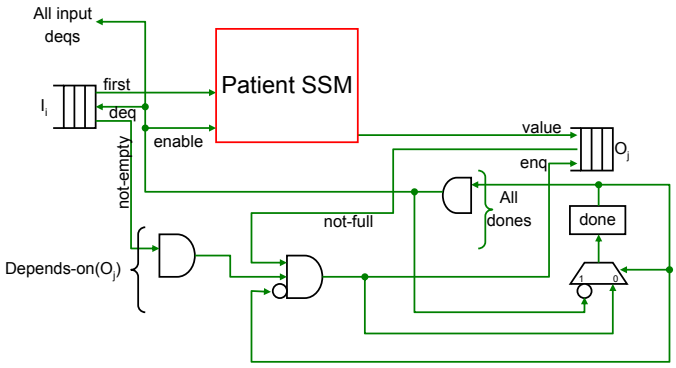


Fig. 12: A wrapper to turn a patient SSM into an LI-BDN

as follows:

```
rule OutO_i
  when (¬ICC_{i_1}.empty ∧ ... ∧ ¬ICC_{i_k}.empty ∧
        ¬O_i.full ∧ ¬done_i)
  ⇒ O_i.enq(f_i(s, ICC_{i_1}.first,..., ICC_{i_k}.first));
    done_i ← True
```

Finally we have a rule to dequeue all the inputs:

```
rule Finish
  when (done_1 ∧ ... ∧ done_{n_O} ∧
        ¬I_1.empty ∧ ... ∧ ¬I_{n_I}.empty)
  ⇒ s ← g(s, I_1.first, ..., I_{n_I}.first);
    I_1.deq; ...; I_{n_I}.deq;
    done_1 ← False; ...; done_{n_O} ← False
Initial
  done_1 ← False; ...; done_{n_O} ← False; s ← s_0
```

The BDN described above can be implemented as a synchronous hardware circuit by converting the original SSM into a Patient SSM, and creating a wrapper around it, treating it as a black-box. Figure 12 shows the circuit representing the wrapper.

IV. Theory for Modular Refinement of BDNs

Large SSMs are often designed by composing smaller SSMs. In this section we develop the theory needed to support a design methodology where each smaller SSM can be implemented as a primitive LI-BDN and the large SSM can be constructed simply by composing these LI-BDNs. We will further show that any constituent primitive LI-BDN can be refined to improve some implementation aspect such as area, timing, etc without affecting the overall correctness of the BDN.

A. Preliminaries

Definition 10. (Bounded Dataflow Network (BDN))
A BDN is a network of primitive BDN nodes such that

1) At most one node can enqueue in a given FIFO and at most one node can dequeue from a given FIFO.
2) For any FIFO X in the network, the transitive closure of Combinational-connected(X) is well defined, i.e., X is not in the transitive closure of Combinationally-connected(X). □

The transitive closure restriction formally says that a BDN does not contain any "combinational loops". Note the similarity of this definition with that of SSMs (Definition 1).

Lemma 1. *If an SSM is formed as a network of smaller SSMs, then a BDN can be formed as a corresponding network of primitive BDNs that implement the smaller SSMs.*

Proof: Since an SSM formed from smaller SSMs does not contain a combinational loop, neither does the corresponding network of the primitive BDNs; thus it forms a BDN. ■

We now extend several definitions that we had for primitive BDNs to larger BDNs.

Definition 11. (Depends-on relation for BDNs)
For any output O_i of a BDN R, Depends-on(O_i) is the inputs of R that are in the transitive closure of Combinationally-connected(O_i). □

Definition 12. (NED property for BDNs)
A BDN has the NED property if all outputs have been enqueued atleast $n-1$ times and for each output O_i, all the inputs in Depends-on(O_i) are enqueued n times, and all other inputs are enqueued at least $n-1$ times, then O_i must be enqueued n times. □

Definition 13. (SC property for a BDN)
A BDN has the SC property, if all the outputs are enqueued n times, then all the inputs must be dequeued n times, assuming an infinite source for each input. □

Definition 14. (Latency-Insensitive (LI) BDNs)
An LI-BDN is a BDN which has the NED and SC properties. □

A natural equivalence relation between BDNs can be defined in terms of their input-output behavior as follows:

Definition 15. (Equivalence of BDNs)
BDNs R_1 and R_2 are said to be equivalent iff

1) There is a bijective mapping between the inputs of R_1 and R_2, and a bijective mapping between the outputs of R_1 and R_2.
2) The output histories of R_1 and R_2 matches whenever the input histories matches, *i.e.*,

$\forall n > 0,$
$I(k)$ for R_1 and R_2 matches $(1 \le k \le n)$
$\Rightarrow O(j)$ for R_1 and R_2 matches $(1 \le j \le n)$ □

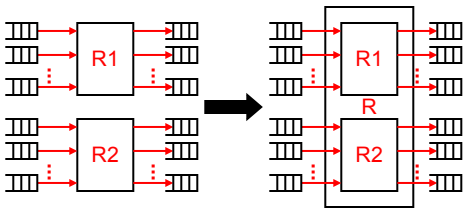
Our theory rests on the fact that this equivalence relation becomes a congruence for LI-BDNs, *i.e.*, if two LI-BDNs are equivalent then there is no way to tell them apart regardless of the context.

B. Structural composition of BDNs

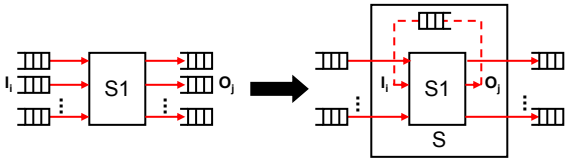
Just like SSMs, any BDN can be formed inductively using parallel and iterative compositions of BDNs starting with primitive BDNs.

Definition 16. (Structural definition of BDNs)

1) Primitive BDNs are BDNs
2) If R_1 and R_2 are BDNs, then so is the *parallel composition* of R_1 and R_2, written as $R_1 \oplus R_2$ (Figure 13a). The semantics of parallel composition of two BDNs is defined



(a) Parallel composition of BDNs



(b) Iterative composition of BDNs

Fig. 13: Composition of BDNs

by the union of the two sets of disjoint rules (*i.e.*, rules with no shared state) for the two BDNs.

3) If R_1 is an BDN then so is the (I_i, O_j) *iterative composition* of R_1 written as $(I_i, O_j) \odot R_1$ (Figure 13b), provided that $I_i \notin$ Depends-on(O_j). The semantics of the iterative composition of a BDN is defined by aliasing the name of a pair of an input and an output FIFO in the set of rules associated with the BDN. □

The following two theorems are the main theorems of this paper:

Theorem 1. *(Modular Composition Theorem)*
If R_1 and R_2 are LI-BDNs implementing SSMs S_1 and S_2, respectively, then

- *$R = R_1 \oplus R_2$ is an LI-BDN that implements $S = S_1 + S_2$.*
- *$R = (I_i, O_j) \odot R_1$ is an LI-BDN that implements $S = (I_i, O_j) \cdot S_1$.*

Theorem 2. *(Modular Refinement Theorem)*
Equivalence of two LI-BDNs is preserved under parallel and iterative composition, i.e. if R_1 and R_2 are LI-BDNs equivalent to BDNs R_1' and R_2', then

- *$R = R_1 \oplus R_2$ is an LI-BDN equivalent to $R' = R_1' \oplus R_2'$.*
- *$R = (I_i, O_j) \odot R_1$ is an LI-BDN equivalent to $R' = (I_i, O_j) \odot R_1'$.*

The proofs of the two theorems are very similar; we only give the proof of Theorem 1.

Proof:

1) Compositions of LI-BDNs are LI-BDNs. (Lemma 2)
2) LI-BDNs are deadlock-free. (Lemma 3)
3) If R is a composition of LI-BDNs R_1 and R_2 then R partially implements the composition of S_1 and S_2 where S_1 and S_2 are the SSMs corresponding to R_1 and R_2, respectively. (Lemma 4)

■

Lemma 2. *(Closure property of LI-BDNs) If R_1 and R_2 are LI-BDNs then so are $R_1 \oplus R_2$ and $(I_i, O_j) \odot R_1$.*

Proof: The proof is obvious for the parallel composition.

For the iterative composition, we show by induction on n, O_j will be enqueued and dequeued n times if all the inputs are enqueued n times and all the outputs are enqueued n times. This is trivial as O_j will not wait for I_i (as R_1 has the NED property). All output FIFOs of R_1 can now be enqueued and so all input FIFOs of R_1 can be dequeued (R_1 has the SC property).

We now prove again by induction on n that for an output $O_k \neq O_j$, O_k will be enqueued n times if all the inputs in Depends-on(O_k) are enqueued n times, and all the other inputs are enqueued at least $n-1$ times, and all the other outputs are enqueued at least $n-1$ times (NED property). There are two cases to consider:

1) $I_i \notin$ Depends-on(O_k): trivial as R_1 is an LI-BDN.
2) $I_i \in$ Depends-on(O_k): O_j is not full because of the SC property of R_1 and it will be enqueued n times because of NED property of R_1, which makes I_i and hence O_k enqueued n times.

By a similar induction on n we can show that if all the inputs are enqueued n times, and all the outputs are enqueued n times, then all the inputs will be dequeued n times (SC property). ■

Lemma 3. *(Deadlock-free Lemma) LI-BDNs are deadlock-free.*

Proof: Given infinite sinks for outputs and infinite source for inputs, by induction on the number of inputs and outputs enqueued it can be shown using the NED and SC properties of an LI-BDN that it will not deadlock. ■

Lemma 4. *If R_1 and R_2 are LI-BDNs then $R_1 \oplus R_2$ partially implements $S_1 + S_2$ and $(I_i, O_j) \odot R_1$ partially implements $(I_i, O_j) \cdot S_1$*

Proof: The proof is obvious for the parallel composition. For iterative composition, we first show by induction on n $O_j(n)$ matches for R and S whenever the input histories for R and S match upto n values. $O_j(n)$ does not depend on $I_i(n)$ in S_1. So $O_j(n)$ matches in R_1 and S_1 whenever the rest of the inputs match upto $n-1$ as R_1 implements S_1.

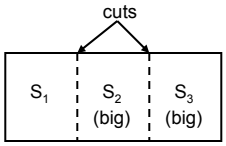
We show by induction on n, for an output $O_k \neq O_j$, O_k will have the same n^{th} values in the SSM and the LI-BDN if the input histories match upto n values. There are two cases to consider:

1) $I_i \notin$ Depends-on(O_k): trivial as R_1 implements S_1.
2) $I_i \in$ Depends-on(O_k): $O_j(n)$ matches. Since all the n^{th} inputs are the same for R_1 and S_1, all the n^{th} outputs will be the same in both.

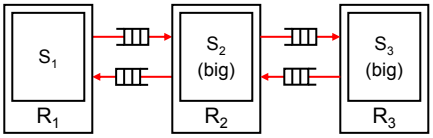
■

V. Cycle-accurate refinements of SSMs using LI-BDNs

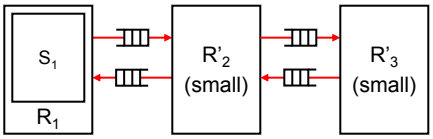
One way to apply the theory that we have developed is to refine parts of an existing SSM (referred to as the model SSM in this section) into more area-efficient hardware structures. Such modular refinements, in general, are quite difficult because the refined module may take a different number of cycles



(a) Model SSM



(b) Turning the model SSM into an LI-BDN



(c) Refining R_2 and R_3

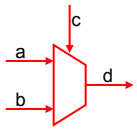
Fig. 14: Modular Refinement

than the original module. Under such circumstances even if one proves the functional correctness of the refined module with respect to the original module, there is no guarantee that the functionality of the model SSM would be preserved by the modularly refined design. This problem is faced by many designers who are trying to use FPGAs.

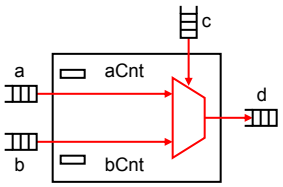
Based on the theory we have presented, a designer can make one or more cuts in his model SSM to separate the modules he wishes to change. Each such module can be converted into an LI-BDN using the procedure given in Section III-E. Once the SSM is implemented as a network of LI-BDN nodes, then each node can be refined into a different but equivalent LI-BDN (Figure 14). There is no need to verify the entire LI-BDN again for correctness, just the changed node has to be validated. Thus one can do true modular refinements using LI-BDNs. In this section, we give examples of some useful refinements of LI-BDNs.

Example 1: Optimizing the multiplexor

Figure 15a shows an SSM for a multiplexor and Figures 15b and 15c show primitive BDNs which implement the multiplexor. Figure 15b is the standard primitive BDN while in Figure 15c the rule waits only for the input indicated by the predicate. The output will be enqueued by the rule even if the other input takes longer to get enqueued. When the other input finally arrives, it will be discarded; the counters keep track of how many inputs to discard. A similar multiplexor was considered using A-Ports [15]. The BDN is an LI-BDN because it obeys the NED and SC properties. In general, this idea can be used to "run-ahead" in any LI-BDN, without waiting for all the inputs, thereby improving the performance of the whole system. This is shown in Figure 15d. If function g is implemented as a multi-cycle function, then the output in d has to wait till g is computed every time if we use the multiplexor in Figure 15b. If we use the multiplexor in Figure 15c, then the multiplexor has to wait for g only if the predicate c is false.



(a) Multiplexor SSM

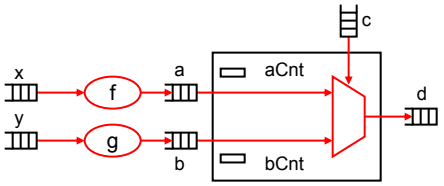



```
rule OutD
  when (¬d.full ∧ ¬c.empty ∧ ¬a.empty ∧ ¬b.empty)
  ⇒ if(c.first) then d.enq(a.first);
     else d.enq(b.first);
     a.deq; b.deq; c.deq
```

(b) A multiplexor BDN (Does not use the `Done` flags)

```
rule OutD
  when (¬d.full ∧ ¬c.empty)
  ⇒ if(c.first ∧ ¬a.empty ∧ bCnt ≠ max) then
       d.enq(a.first); a.deq; c.deq; bCnt ← bCnt+1;
    else if(¬b.empty ∧ aCnt ≠ max)
       d.enq(b.first); b.deq; c.deq; aCnt ← aCnt+1;
rule DiscardA
    when (aCnt > 0 ∧ ¬a.empty)
    ⇒ a.deq; aCnt ← aCnt-1
rule DiscardB
    when (bCnt > 0 ∧ ¬b.empty)
    ⇒ b.deq; bCnt ← bCnt-1
Initial
    aCnt ← 0; bCnt ← 0
```

(c) A performance-optimized multiplexor BDN

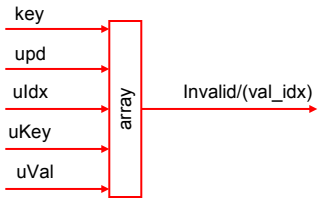


(d) Illustrating the performance optimized multiplexor

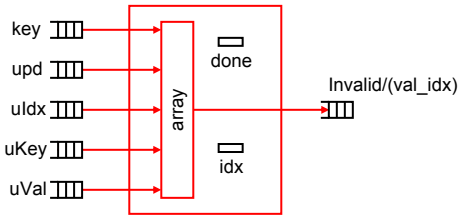
Fig. 15: Optimizing the multiplexor BDN

Example 2: Multicycle implementation of Content Addressable Memory (CAM)

Figure 16a shows a synchronous CAM lookup. The CAM is an array of n elements, where each element stores a (key, value) pair. Every cycle, the synchronous CAM returns a value and the index of the *search-key* in the CAM-array if the key is found; otherwise it returns an *Invalid*. If the *upd* signal is enabled, then the CAM array is updated with *uKey* and *uVal* at the position given by *uIdx*. Single cycle CAMs are expensive structures in terms of area and critical path. If the CAM is implemented as an LI-BDN, then it can be refined to do a sequential lookups taking several cycles to lookup the value corresponding to a key (Figure 16b). See [16] for other multi-cycle implementations of CAM. This keeps the rest of the system using the CAM unchanged, while resulting in a circuit with lesser area and lesser critical path than the original unrefined LI-BDN. If the CAM lookup is not done frequently,



(a) A single cycle CAM lookup SSM




```
rule OutValueIndex
  when (¬key.empty ∧ ¬val_idx.full ∧ ¬done)
  ⇒ if(idx = LastIdx+1)
      val_idx.enq(Invalid); idx ← 0;
      done ← True
    else if(array[idx].key = key.first)
      val_idx.enq(array[idx].value, idx);
      idx ← 0; done ← True
    else idx ← idx + 1
rule Finish
  when (¬key.empty ∧ ¬upd.empty ∧
        ¬uKey.empty ∧ ¬uVal.empty ∧
        ¬uIdx.empty ∧ done = True)
  ⇒ if(upd.first)
      array[uIdx.first].key = uKey.first;
      array[uIdx.first].val = uVal.first;
    key.deq; upd.deq; uKey.deq; uVal.deq;
    uIdx.deq; done ← False
```

(b) Multicycle LI-BDN implementing the CAM

Fig. 16: CAM lookup as a multicycle LI-BDN

then using the multiplexor that we discussed above we can ensure that even the performance degradation is minimized.

VI. Conclusions

We have presented a theory using Bounded Dataflow Networks (BDNs) that can help in modular latency-insensitive refinements of synchronous designs. The theory should help implementers avoid deadlocks in implementing latency insensitive circuits, especially when refinements involve changes in design that affect the timing.

In future we wish to explore the use of BDNs even in the absence of a clearly specified model SSM. For example, it is not clear how the timing requirements for a modern complex processor should be specified. One can use the RTL of the microprocessor as a timing specification but usually RTL is not available at the time when modelers do their architectural explorations. Furthermore, architects want to be able to modify their micro-architectures and study its effect on performance (*i.e.*, the number of cycles it takes to execute a program) without having to worry about the correctness of various models. We doubt that RTL for the many variants of the models that an architect wants to study can be provided to the architect. It is quite common to set up the design in a way that latency insensitive aspects of the design are clear to the modeler. However, a modeler probably would not want to make any changes in its design to accommodate the vagaries of an FPGA implementation because mixing of these two concerns may destroy his intuition about the performance of the machine to be studied. We think BDNs can provide the required separation between these two types of timing changes.

Acknowledgement

The authors would like to thank Intel Corporation and NSF grant Generating High-Quality Complex Digital Systems from High-level Specification (No. 0541164) for funding this research. The discussions with the members of Computation Structures Group at MIT, especially Joel Emer, Mike Pellauer, Asif Khan and Nirav Dave have helped in refining the ideas presented in this paper.

References

[1] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing '74: Proceedings of the IFIP Congress*, J. L. Rosenfeld, Ed. New York, NY: North-Holland, 1974, pp. 471–475.

[2] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ISCA '75: Proceedings of the 2nd annual symposium on Computer architecture*. New York, NY, USA: ACM, 1975, pp. 126–132.

[3] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency insensitive systems," in *DAC '00: Proceedings of the 37th conference on Design automation*. New York, NY, USA: ACM, 2000, pp. 361–367.

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

[5] L. P. Carloni, K. L. Mcmillan, and A. L. Sangiovanni-vincentelli, "Latency insensitive protocols," in *in Computer Aided Verification*. Springer Verlag, 1999, pp. 123–133.

[6] L. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli, "A methodology for correct-by-construction latency insensitive design," *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on*, pp. 309–315, 1999.

[7] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-ports: an efficient abstraction for cycle-accurate performance models on fpgas," in *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2008, pp. 87–96.

[8] ——, "Quick performance models quickly: Closely-coupled partitioned simulation on fpgas," April 2008, pp. 1–10.

[9] 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 *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*. Piscataway, NJ, USA: IEEE Press, 2007, pp. 295–302.

[10] K. Asanovic. (2008, January) RAMP Gold. RAMP Retreat. [Online]. Available: http://ramp.eecs.berkeley.edu/Publications/RAMP%20Gold%20(Slides,%201-16-2008).ppt

[11] D. Chiou, private Communication.

[12] J. Hoe and Arvind, "Operation-centric hardware description and synthesis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 23, no. 9, pp. 1277–1288, Sept. 2004.

[13] J. C. Hoe and Arvind, "Synthesis of operation-centric hardware descriptions," in *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*. Piscataway, NJ, USA: IEEE Press, 2000, pp. 511–519.

[14] Bluespec System Verilog. Bluespec Inc. [Online]. Available: http://www.bluespec.com

[15] M. Pellauer, private Communication.

[16] K. Fleming and J. Emer, "Resource-efficient fpga content-addressable memories," in *WARP*, 2007.