Bounded Dataflow Networks (BDNs) and Latency-Insensitive (LI) Circuits

Murali Vijayaraghavan and Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

WG 2.8, Chiemsee, Germany
June 12, 2009

Dataflow networks

- Kahn-Dennis networks: A network of computing stations connected by unbounded FIFOs
  - a “get” is blocking but a “put” is not
- Dataflow networks with bounded FIFOs (BDNs)
  - Hard to model as a Kahn-Dennis network
  - Varying the size of a FIFO changes the meaning (may cause a deadlock)
- Several groups are using BDNs for latency-insensitive refinements of *Synchronous Sequential Machines* (SSMs) and often encounter deadlocks

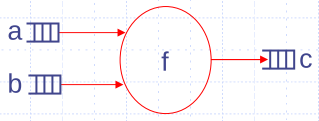
Bounded Dataflow Networks

- Can be modeled accurately in Bluespec
- Can be used a high-level structuring technique for Bluespec designs

what restrictions should be placed on BDNs such that its meaning does not change with respect to a given SSM when we vary the FIFO sizes

Examples of primitive BDNs:

A Combinational Block



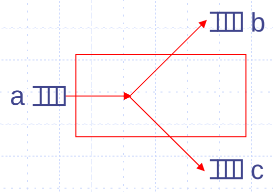
f is a combinational circuit: must accept an input value on each input before producing an output

Behavior

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

Unlike SSMs, the (red) lines only show dataflow and not all the control lines needed to make BDNs function

A fork definition



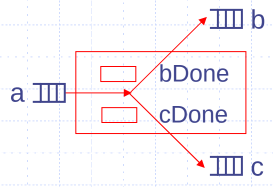
a fork that copies an input value to both its outputs simultaneously

Behavior

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

Examples of primitive BDNs:

Fork



a fork to copy an input value but the input can be dequeued only when both the outputs have accepted the input

Behavior

```
rule FO1 when (¬a.empty∧¬b.full ∧ ¬bDone)
  ⇒ b.enq(a.first); bDone <= True

rule FO2 when (¬a.empty∧¬c.full ∧ ¬cDone)
  ⇒ c.enq(a.first); cDone <= True

rule FI when (¬a.empty∧bDone ∧ cDone)
  ⇒ a.deq; bDone <= False;
     cDone <= False
```

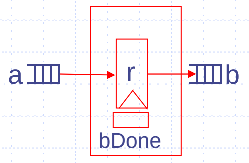
Initial Values

bDone = False
cDone = False

Which one do we want?

Examples of primitive BDNs:

Register



A register whose reads and writes must match

Behavior

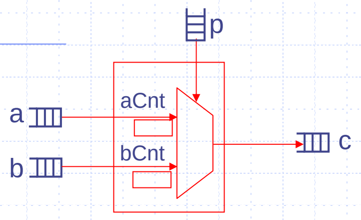
```
rule RO when (¬b.full ∧ ¬bDone)
   ⇒b.enq(r); bDone <= True

rule RI when (¬a.empty ∧ bDone)
   ⇒r <= a.first; a.deq; bDone <= False
```

Initial Values

bDone = False
r = r_0

Examples of primitive BDNs:

Mux



A mux that accepts an input value on each input port but passes only the appropriate value to the output

Behavior

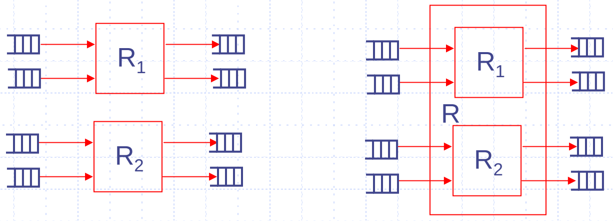
```
rule MuxO when ¬c.full ∧ ¬p.empty
 ⇒ if(p.first ∧ ¬ a.empty)
     then c.enq(a.first); a.deq; bCnt<=bCnt+1
     else if(!(p.first) ∧ ¬ b.empty)
          then c.enq(b.first); b.deq; aCnt<=aCnt+1
rule MuxI1 when aCnt >0 ∧ ¬ a.empty
  ⇒ a.deq; aCnt<=aCnt-1
rule MuxI2 when bCnt >0 ∧ ¬ b.empty
  ⇒ b.deq; bCnt<=bCnt-1
```

Initial values

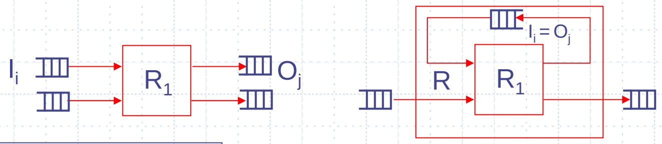
aCnt = 0
bCnt = 0

Composition of BDNs

- If R_1 and R_2 are BDNs then so is the parallel composition of R_1 and R_2 ($R = R_1 \oplus R_2$)

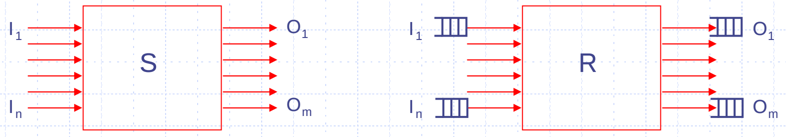


- R1 is a BDN then so is the (Ii ,Oj) iterative composition of R1 (R = (i,j) ⊗ R1) provided Ii ∉ Depends-on(Oj)*



* No direct combinational path

BDN as a refinement of an SSM

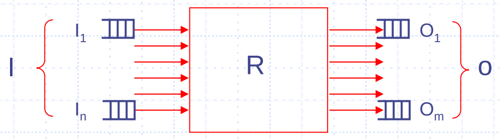


- There is a bijective mapping between the inputs (outputs) of S and R
- for all n > 0,

$$\left.\begin{array}{l} I(k) \text{ matches for S and R } (1 \le k \le n) \\ \Rightarrow O(j) \text{ matches for S and R } (1 \le j \le n) \end{array}\right\} \text{Cycle Accuracy}$$

In general it is difficult to compare an SSM and a BDN because a BDN can deadlock. We will restrict our attention to a class of BDNs with some “desirable properties”

Deadlock-free BDN



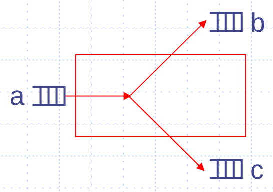
- Assuming an infinite sink, a BDN is deadlock-free if for all n > 0, if n values are enqueued into I then eventually n values will be dequeued from both O and I
  - we need a stronger property for deadlock-freeness to be preserved under composition

NED-BDN:

BDNs with *no extraneous dependencies*

- A BDN is said to have *no extraneous dependencies* if its output O_i is not enqueued n times, assuming it is not full and all the inputs are enqueued n-1 times, then it must be that one of the inputs in Depends-on(O_i) is not enqueued n times
- Note that this is a property of BDN – it is different from the condition for iterative composition

NED-BDN violation 1

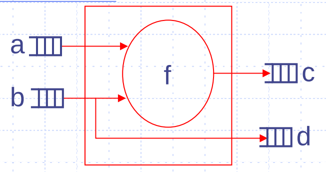


a fork that copies an input value to both its outputs simultaneously

Behavior

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

NED-BDN violation 2



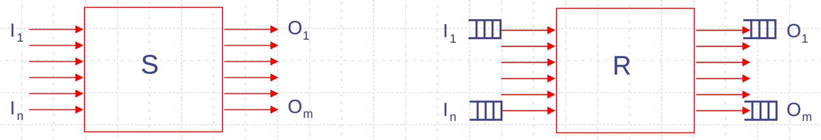
Possible Behaviors

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

rule O1 when (¬a.empty∧¬b.empty∧¬c.full ∧¬cDone)
⇒ c.enq(f(a.first, b.first)); cDone <= True
rule O2 when (¬b.empty∧¬d.full ∧¬dDone)
⇒ d.enq(b.first); dDone <= True
rule In when (cDone ∧dDone)
⇒ a.deq ; b.deq

Latency-Insensitive (LI) BDN

- LI-BDN is an NED-BDN which is refinement of an SSM



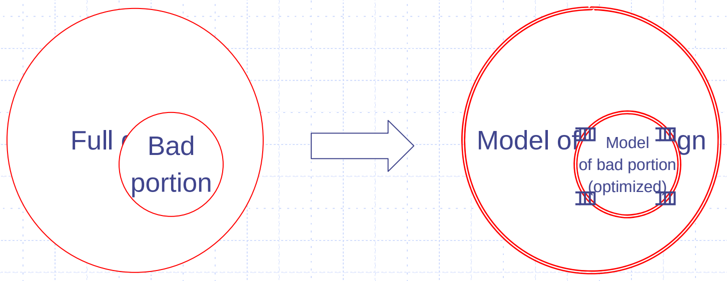
LI refinement Theorem

- LI-BDNs are composable under parallel and sequential composition
  - If R_1, R_2 are refinements of S_1, $S_2 \Rightarrow$
    - $R_1 \oplus R_2$ is the refinement of $S_1 + S_2$
    - $(i, j) \otimes R_1$ is the refinement of $(i, j) \times S_1$
- Basically this ensures that the composition of LI-BDNs are deadlock-free and cycle-accurate w.r.t. the original SSMs

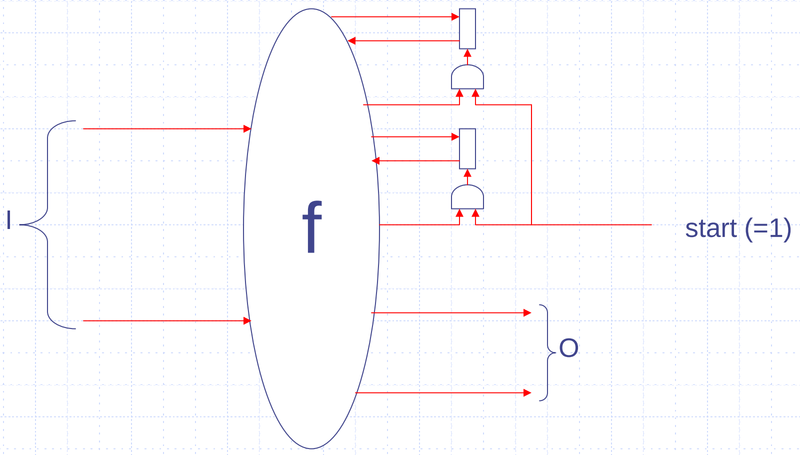
Application: Modeling via RTL prototyping on FPGAs

- Some RTL structures are inefficient to map directly onto FPGAs
  - For example, a 3-ported register file (RF) consumes lot of area as opposed a 1-ported RF used for 3 cycles
  - However, replacing a 3-ported RF naively by a 1-ported RF in a design may loose "cycle-accuracy", even if the high-level functionality "turns out" to be correct

Application: Cycle-accurate modeling



Startable SSMs: SSMs with a “start” signal to update registers



1000 feet view of LI-refinement of an SSM

- FIFOs are introduced in every input and every output of the SSM
- Time cycles of the SSM are converted into enqueues into inputs and dequeues from outputs
  - “Cycle-accurate” w.r.t SSM
- Atomic rules for the operations are defined so that no extraneous dependencies are introduced
  - Ensures deadlock-free operation

Writing the LI-BDN wrapper for an SSM

LI-BDN:

```
rule j (!$o_j$.done)
    $o_j$.done <= True
    $o_j$.enq( $f_j(i_{j1}$.first, … ,$i_{j|j}$.first, s)  )

rule finish ($o_1$.done && $o_2$.done && …)
    $o_1$.done <= False; $o_2$.done <= False; …
    s <= g($i_1$.first, $i_2$.first, … , s)
    $i_1$.deq ; $i_2$.deq ; …
```

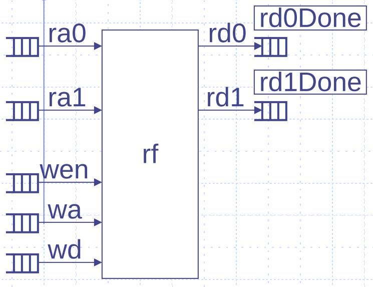
Given the SSM:

$o_j(t) = f_j(i_{j1}(t), \ldots, i_{j|j}(t), s(t))$

// $i_{j1}, i_{j2}, \ldots i_{j|j}$ are in Depends-on(o_j)

$s(t+1) = g(i_1(t), i_2(t), \ldots, s(t))$

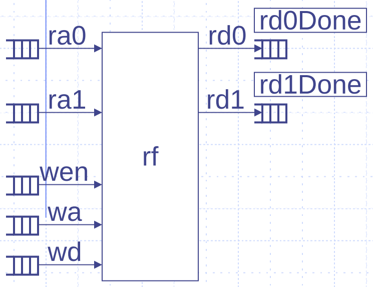
2) Automatically generated LI-BDN for a 3-ported register file



```
rule RD0 when (¬rd0Done)
    rd0.enq(rf_0[ra0.first])
    rd0Done <= True

rule RD1 when (¬rd1Done)
    rd1.enq(rf_1[ra1.first])
    rd1Done <= True

rule finish when (rd0Done ∧ rd1Done)
    ra0.deq; ra1.deq
    wen.deq; wa.deq; wd.deq
    rf_2[wa.first] <= wen.first?
                  wd.first : rf_2[wa.first]
    rd0Done <= False
    rd1Done <= False
```

This again uses 3 ports

Refinement into a one-ported register file LI-BDN



This uses 1 port

```
rule RD0 when (¬rd0Done)
    rd0.enq( rf_0 [ra0.first])
    rd0Done <= True

rule RD1 when (¬rd1Done)
    rd1.enq( rf _0 [ra1.first])
    rd1Done <= True

rule finish when (rd0Done ∧ rd1Done)
    ra0.deq; ra1.deq
    wen.deq; wa.deq; wd.deq
    rf_0 [wa.first] <= wen.first?
                          wd.first : rf_0 [wa.first]
    rd0Done <= False
    rd1Done <= False
```

Thanks