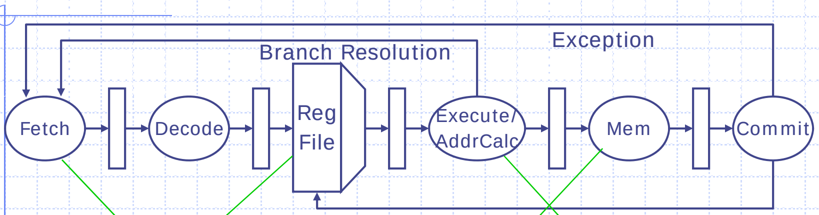
Bounded Dataflow Networks and Latency Insensitive Circuits

Arvind
Computer Science and Artificial Intelligence Laboratory
MIT

Based on the work of Murali Vijayaraghavan and Arvind[MEMOCODE 2009]

Modeling of a processor on an FPGA



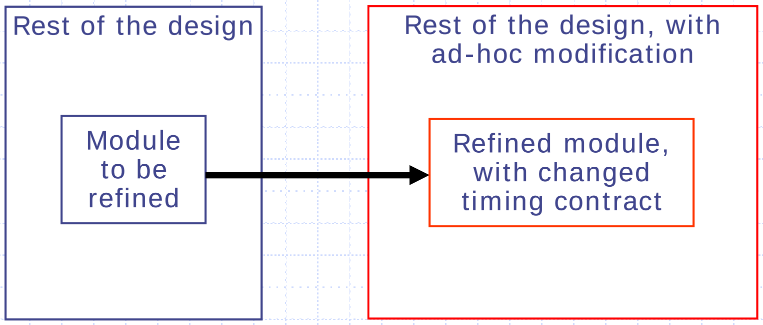
- Multiported register file maps poorly on FPGAs
- Can we map it as a multicycle operation into BRAMs?

- Divide and multiply are resource hogs
- Can we pipeline or implement them as a multicycle operation

- CAM for TLBs map poorly on FPGAs
- Can we implement CAMs as sequential search using BRAMs?

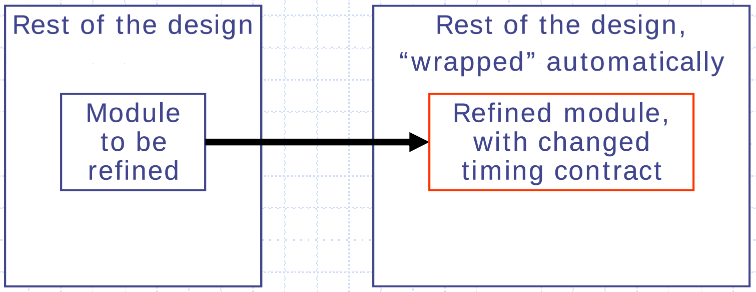
How to do these refinements to Synchronous Sequential Machines (SSMs) without affecting the overall correctness

Conventional “modular refining” methodology


Rest of the design
Module to be refined
Rest of the design, with ad-hoc modification
Refined module, with changed timing contract

- Requires re-verification
- Besides, in our processor example, after the ad-hoc changes, what are we modeling?

True modular refinement


Rest of the design
Module to be refined
Rest of the design, “wrapped” automatically
Refined module, with changed timing contract

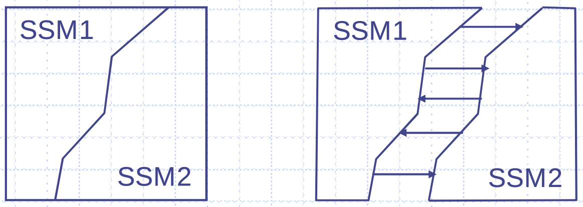
Ability to replace any module by an “equivalent” module without affecting the overall correctness

Theory of Latency Insensitive Designs by Carloni *et. al*

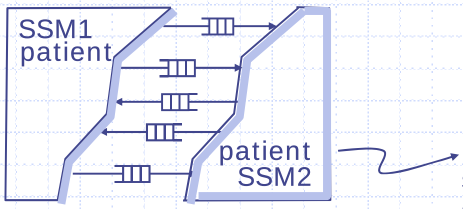
[ICCAD99, IEEE-TCAD01]

- A method to reduce critical wire delays by adding buffers
  - Module is treated as a black-box and wrapped to make it latency-insensitive to input/output wire latencies
- Our goal is to also permit refinements that may change the timing of a module

Carloni’s method

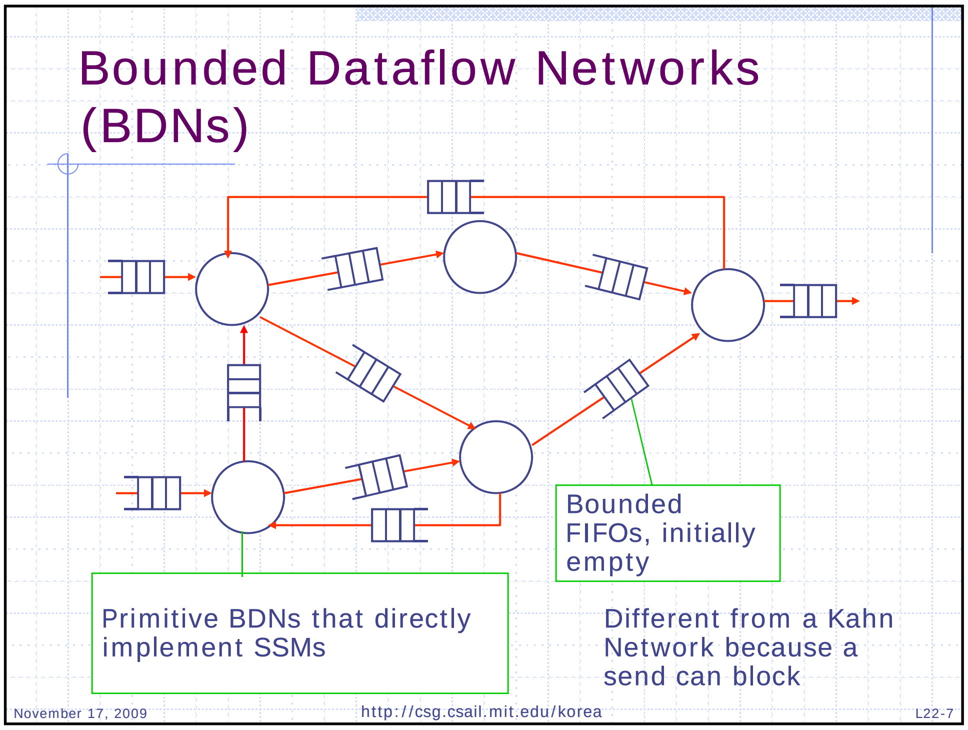


Make a cut to include the wires of interest (some restrictions on cuts)

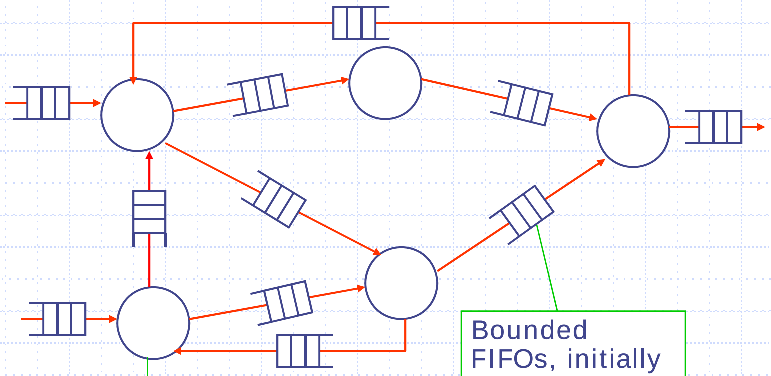


Create wrappers and insert buffers or FIFOs

In patient SSMs the state change can be controlled by an external wire



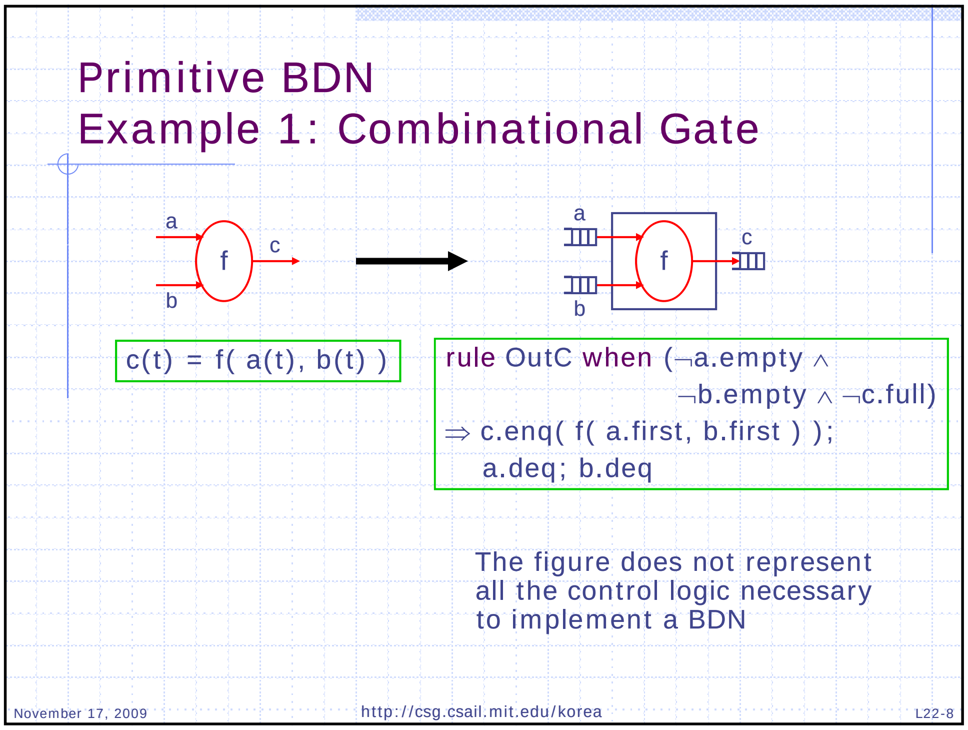
Bounded Dataflow Networks (BDNs)



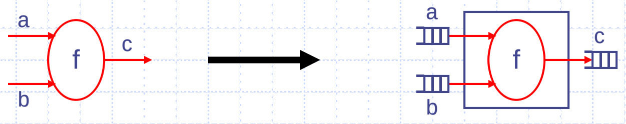
Bounded FIFOs, initially empty

Primitive BDNs that directly implement SSMs

Different from a Kahn Network because a send can block



Primitive BDN Example 1: Combinational Gate

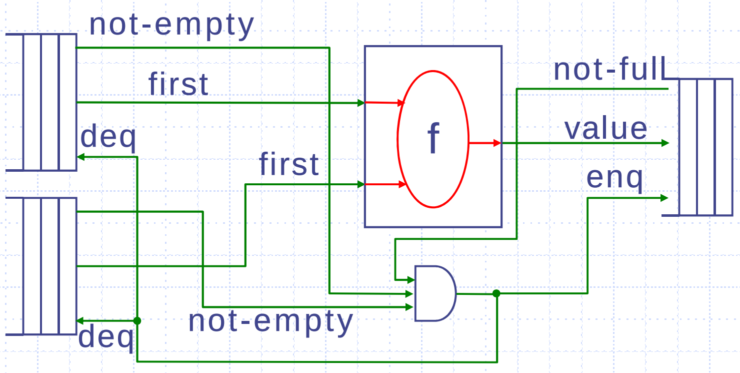


$c(t) = f(a(t), b(t))$

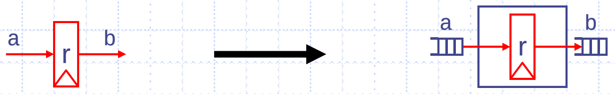
```
rule OutC when (¬a.empty ∧
                  ¬b.empty ∧ ¬c.full)
⇒ c.enq( f( a.first, b.first ) );
   a.deq; b.deq
```

The figure does not represent all the control logic necessary to implement a BDN

Circuit generated for the Gate BDN



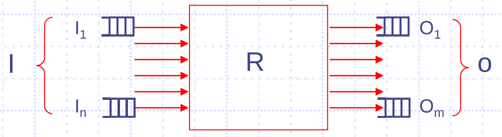
Primitive BDNs Example 2: Register



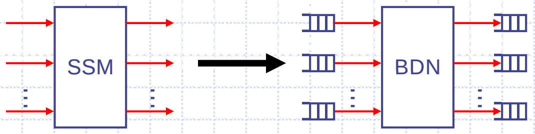
Initial: r ← r0
rule OutB when(¬a.empty ∧ ¬b.full)
⇒ b.enq(r); r ← a.first; a.deq

BDN Input/Output notation

- $I_i(n)$ represents the nth values enqueued in input buffer I_i
  - $I(n)$ represents the nth values enqueued in all input buffers
- $O_j(n)$ represents the nth values dequeued from output buffer O_j
  - $O(n)$ represents the nth values dequeued from all output buffers



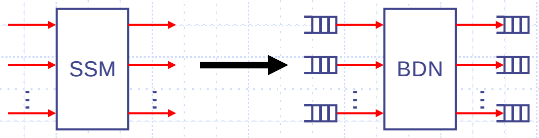
Implementing an SSM as a BDN



- Time is converted into enqueues into input FIFOs and dequeues from output FIFOs
  - $I(t)$ input into an SSM corresponds to the tth enqueues in the input FIFOs of the BDN
  - $O(t)$ output of an SSM corresponds to the tth dequeues from the output FIFOs of the BDN

This separates the timing and functionality in a BDN
$\Rightarrow$ makes BDNs an asynchronous framework

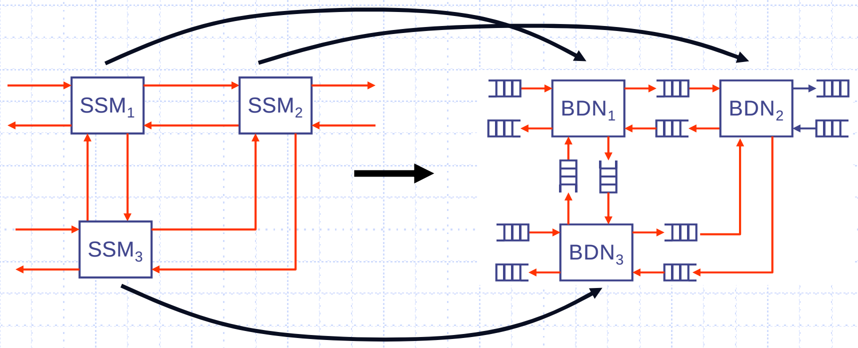
BDN *Implementing* an SSM



A BDN is said to *implement* an SSM iff

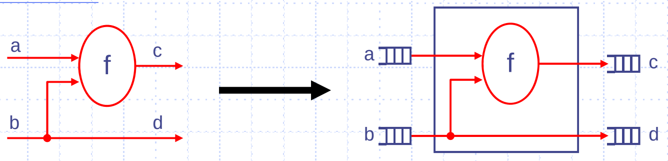
1. There is a bijective mapping between inputs (outputs) of the SSM and BDN
2. The output histories of the SSM and BDN match whenever the input histories match
3. The BDN is *deadlock-free*

Implementing a network of SSMs



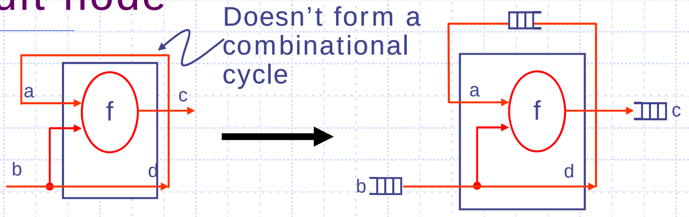
Is this transformation correct?

Implementing an SSM as a BDN Example 3: A combinational circuit

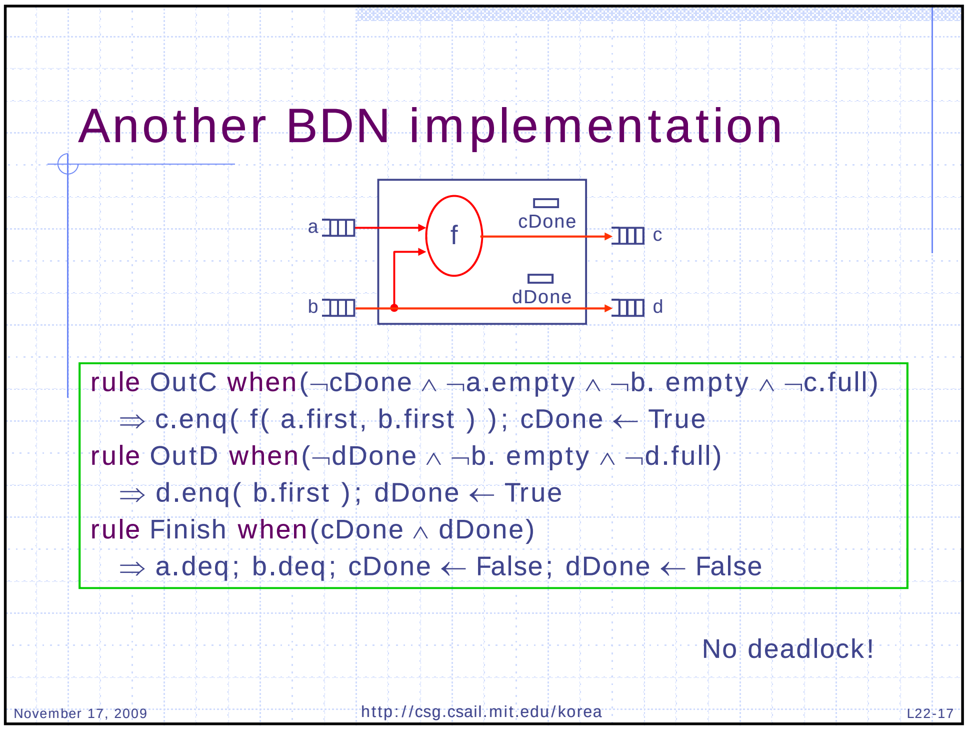


rule OutCD when(¬a.empty ∧ ¬b. empty ∧ ¬c.full ∧ ¬d.full)
 ⇒ c.enq(f(a.first, b.first)); d.enq(b.first); a.deq; b.deq

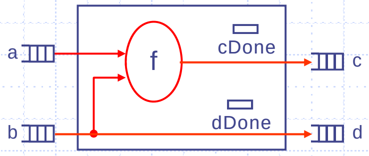
Network with the combinational circuit node



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

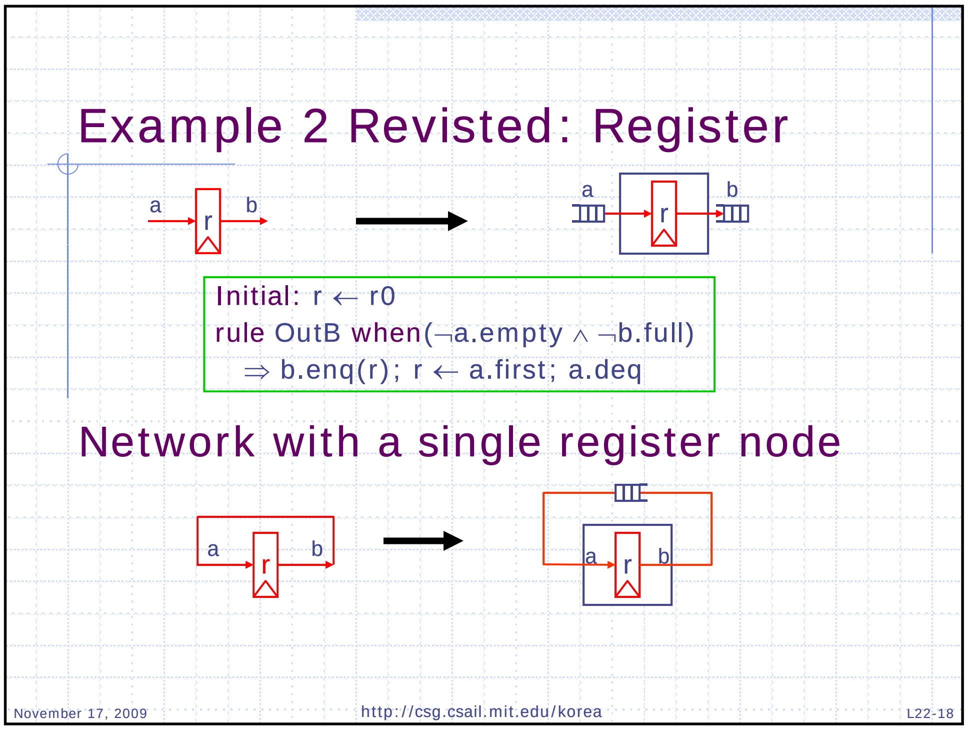


Another BDN implementation

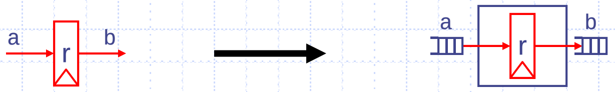


```
rule OutC when(¬cDone ∧ ¬a.empty ∧ ¬b. empty ∧ ¬c.full)
   ⇒ c.enq( f( a.first, b.first ) ); cDone ← True
rule OutD when(¬dDone ∧ ¬b. empty ∧ ¬d.full)
   ⇒ d.enq( b.first ); dDone ← True
rule Finish when(cDone ∧ dDone)
   ⇒ a.deq; b.deq; cDone ← False; dDone ← False
```

No deadlock!

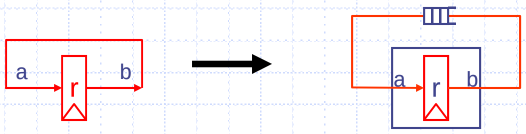


Example 2 Revisted: Register

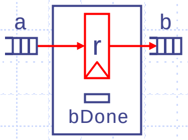


```
Initial: r ← r0
rule OutB when(¬a.empty ∧ ¬b.full)
   ⇒ b.enq(r); r ← a.first; a.deq
```

Network with a single register node

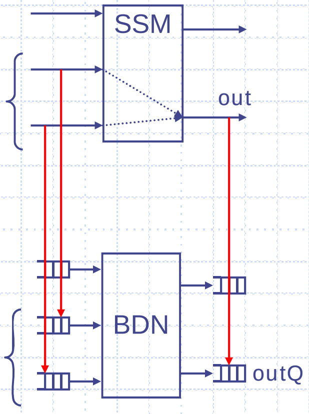


BDN for a register avoiding deadlocks



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

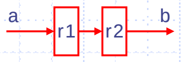
No-Extraneous Dependency (NED) property



Inputs combinationally connected to out

Production of outQ waits only for these input FIFOs

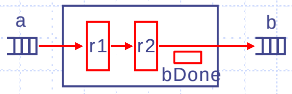
Example 3: A shift register



$r1(t+1) = a(t);\ r2(t+1) = r1(t);$
$b(t) = r2(t)$
initially $r1(0)=r1_0$; $r2(0)=r2_0$

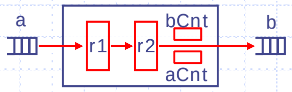
Two BDN implementations

A shift register: Two Implementations



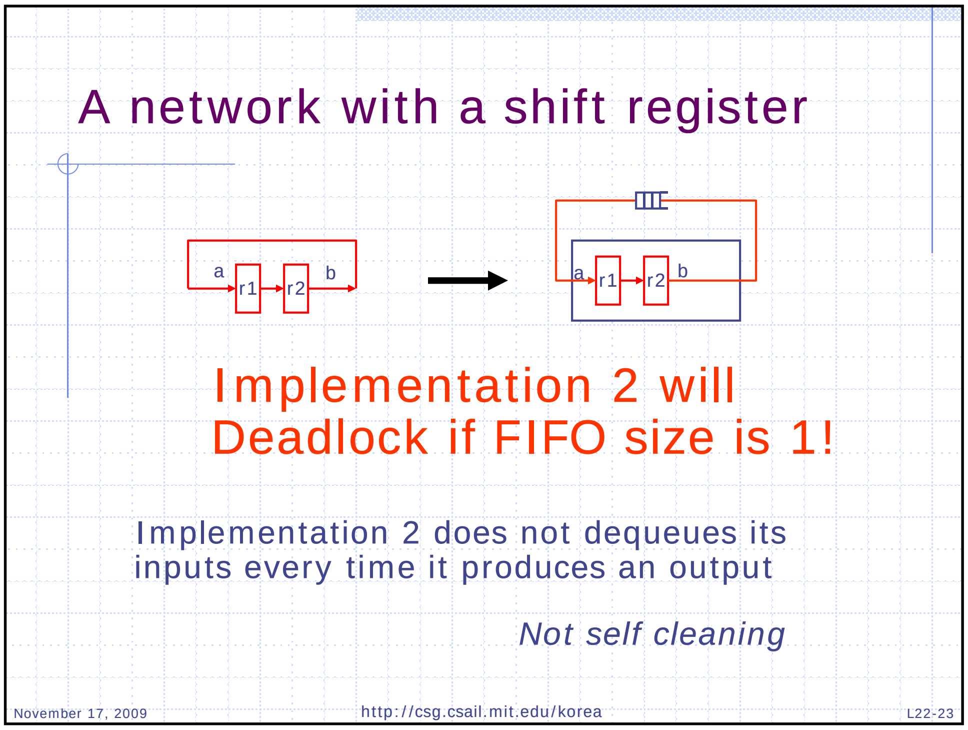
Implementation 1

Initial: $r1 \leftarrow r1_0$; $r2 \leftarrow r2_0$; bDone $\leftarrow$ False
rule OutB when($\neg$bDone $\wedge$ $\neg$ b.full)
$\Rightarrow$ b.enq(r2); bDone $\leftarrow$ True
rule Finish when(bDone $\wedge$ $\neg$ a.empty)
$\Rightarrow$ r1 $\leftarrow$ a.first; r2 $\leftarrow$ r1; a.deq; bDone $\leftarrow$ False

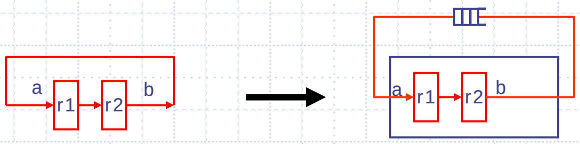


Implementation 2

Initial: $r1 \leftarrow r1_0$; $r2 \leftarrow r2_0$; aCnt $\leftarrow$ 0; bCnt $\leftarrow$ 0;
rule Out1 when(bCnt=0 $\wedge$ $\neg$ b.full)
$\Rightarrow$ b.enq(r2); bCnt $\leftarrow$ 1
rule Out2 when(bCnt=1 $\wedge$ $\neg$ b.full)
$\Rightarrow$ b.enq(r1); bCnt $\leftarrow$ 2
rule In1 when(bCnt=2 $\wedge$ aCnt=0 $\wedge$ $\neg$ a.empty)
$\Rightarrow$ r2 $\leftarrow$ a.first; a.deq; aCnt $\leftarrow$ 1
rule In2 when(bCnt=2 $\wedge$ aCnt=1 $\wedge$ $\neg$ a.empty)
$\Rightarrow$ r1 $\leftarrow$ a.first; a.deq; aCnt $\leftarrow$ 0; bCnt $\leftarrow$ 0;



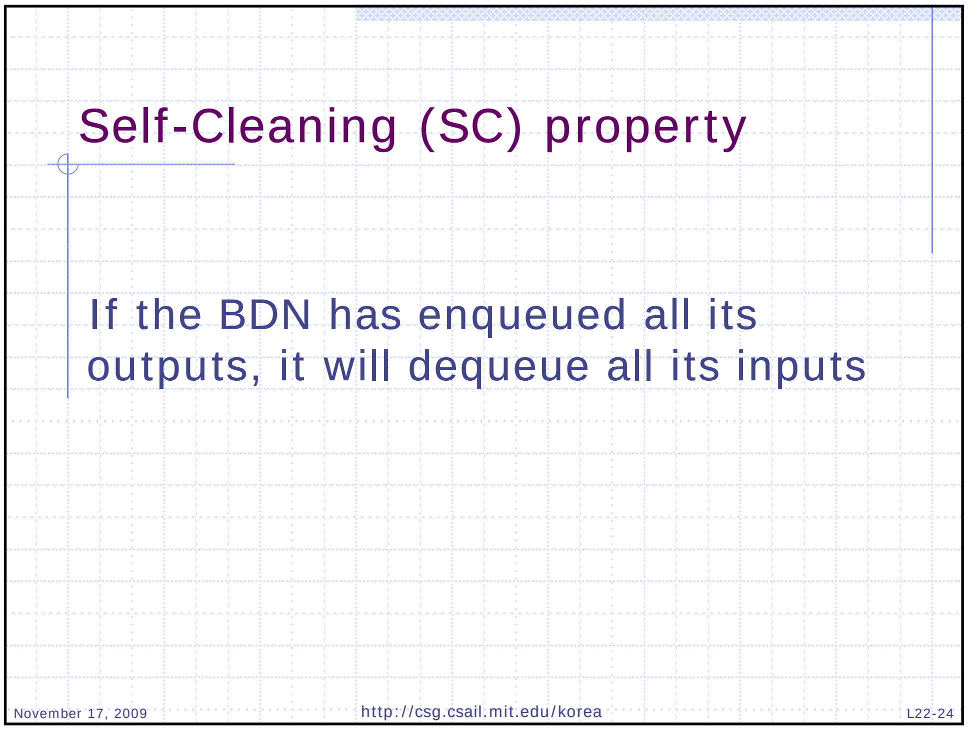
A network with a shift register



Implementation 2 will
Deadlock if FIFO size is 1!

Implementation 2 does not dequeues its inputs every time it produces an output

Not self cleaning



Self-Cleaning (SC) property

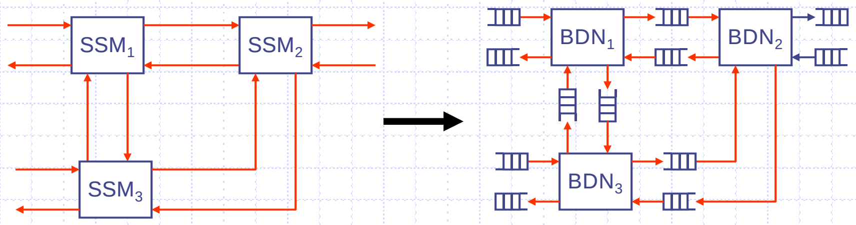
If the BDN has enqueued all its outputs, it will dequeue all its inputs

Latency-Insensitive BDN (LI-BDN)

- A BDN implementing an SSM is an LI-BDN iff it has
  - No extraneous dependencies property
  - Self cleaning property

Theorem: A BDN where all the nodes are LI-BDNs will not deadlock

Implementation of a network of SSMs - revisited



This transformation is correct if each BDN_i *implements* SSM_i and is *latency insensitive*

Next lecture – how to do modular refinement using BDNs