IBM Corporation

Formal Verification of Correctness and Performance of Random Priority-based Arbiters

Krishnan Kailas (IBM T J Watson Research Center, NY)
Viresh Paruthi (IBM Systems & Technology Division, TX)
Brian Monwai (University of Washington, Seattle)

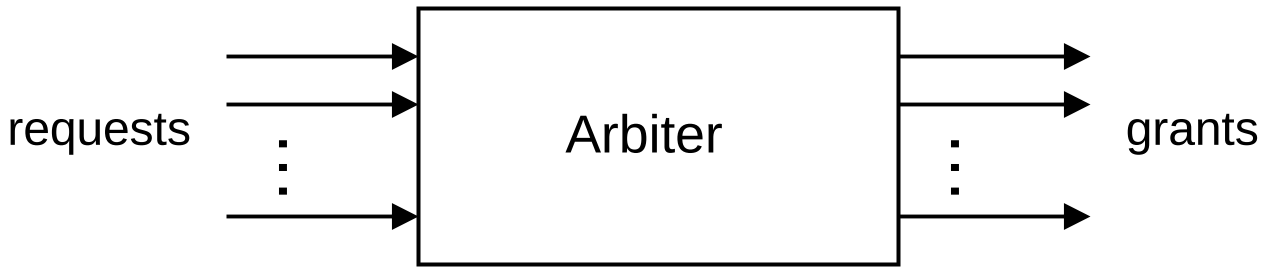
November 17, 2009
Formal Methods in Computer Aided Design (FMCAD 2009), Austin, TX

Outline

- **Arbiters**
- **Correctness properties and performance**
- **Related work**
- **Random priority based arbiters**
- **Complete random sequences**
- **Verification method**
- **Results**

Arbiters

- **Used for restricting access requests to shared resources**
  - when there are more number of requests than the maximum number requests that can be satisfied concurrently.
- **Eg: access to cache directory, shared bus, etc.**



- **Arbitration of large number of requests is sometimes done in multiple stages as reduction trees**

Correctness Properties of Arbiters

- **Mutual exclusion property**
  – At most N requests can be granted concurrently
- **Liveness property**
  – Any request should *eventually* get a grant
  – Can be used to prove arbitration logic is deadlock free

Correctness and Performance

- **Proving liveness property is not sufficient**
- **Example:**
  - Design specs: worst-case cache access latency = 100 cycles
  - Cache latency = directory access time + cache array access time
  - a cache directory access request will *eventually* get a grant after 500 cycles
  - Satisfies the liveness property, but violates the design specification!

- **Request-to-grant delay is an important design specification**

Correctness and Performance

- **Request-to-grant delay is an important design specification**
- **Must prove that the arbiter satisfies the design specifications**
- **Accurate request-to-grant delay bounds are crucial for**
  - timer models used for performance verification
  - avoiding “performance bugs”
- **Solution: bounded liveness property checking**

Correctness Properties of Arbiters

- **Mutual exclusion property**
  - At most N requests can be granted concurrently
- **Liveness property**
  - Any request should *eventually* get a grant
  - Can be used to prove arbitration logic is deadlock free
- **Bounded liveness property**
  - Any request should get a grant within a *bounded time*
  - Subsumes liveness property

Related Work (liveness property checking)

- **Large body of research on specifying liveness properties using temporal logic and model checking**
- **Practical approaches to liveness checking by casting the problem as a safety property checking**
  - Biere, Artho, and Schuppan, “Liveness Checking as Safety,” FMICS’02, 2002.
  - Baumgartner and Mony, “Scalable Liveness Checking via Property-Preserving Transformations,” DATE 2009
- **Formal notion of bounded fairness**
  - kTL and k-fairness: Dershowitz,Jayasimha,Park, “Bounded Fairness”, LNCS 2772, 2003.

Related Work (arbiter verification)

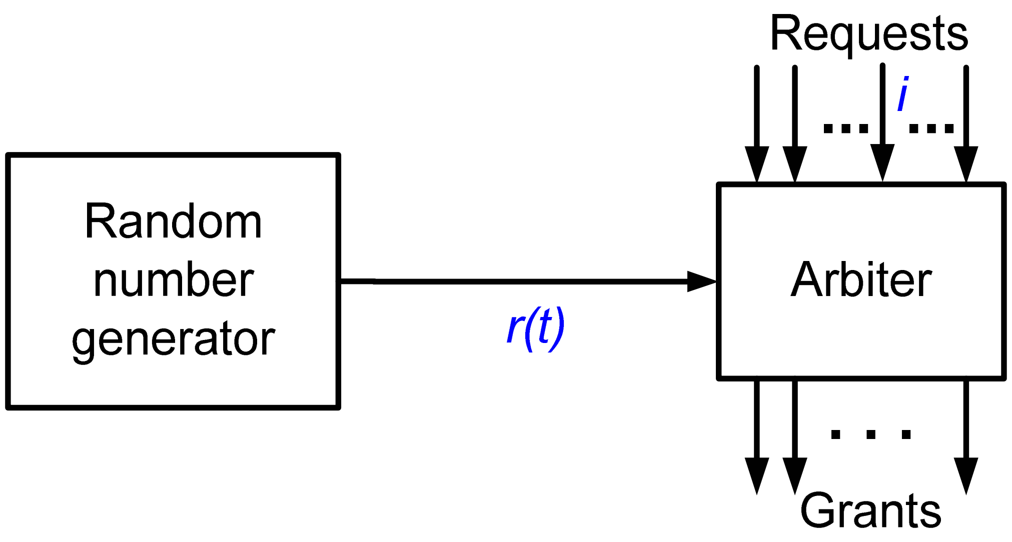
- **Formal verification of high-level models of arbiters, and RTL designs containing arbiters using model checking tools (VIS, RuleBase, SixthSense).**
  - Wasaki, “A Formal Verification Case Study for IEEE-P.896 Bus Arbiter Using A Model Checking Tool,” IJCSNS, 2007.
  - Goel and Lee, “Formal Verification of an IBM CoreConnect Processor Local Bus Arbiter Core,” DAC 2000.
  - Le, Gloekler, and Baumgartner, “Formal Verification of a Pervasive Interconnect Bus System in a High-Performance Microprocessor,” DATE 2007.
- **None of the previous approaches…**
  - verify random-priority based arbitration
  - were not tackling performance aspects, nor reasoning about the fairness scheme.

Arbiters and priority functions

- **Which request should get grant first?**
- **Fairness of arbitration policy is determined by a priority function**
- **Several priority functions:**
  - **Fixed**: certain requests always have higher priority than others
  - **FIFO**: requests are prioritized based on arrival time
  - **Round-robin**: strict rotation of priority assignment
  - **Random**: any request can have the highest priority at random

Random priority arbitration

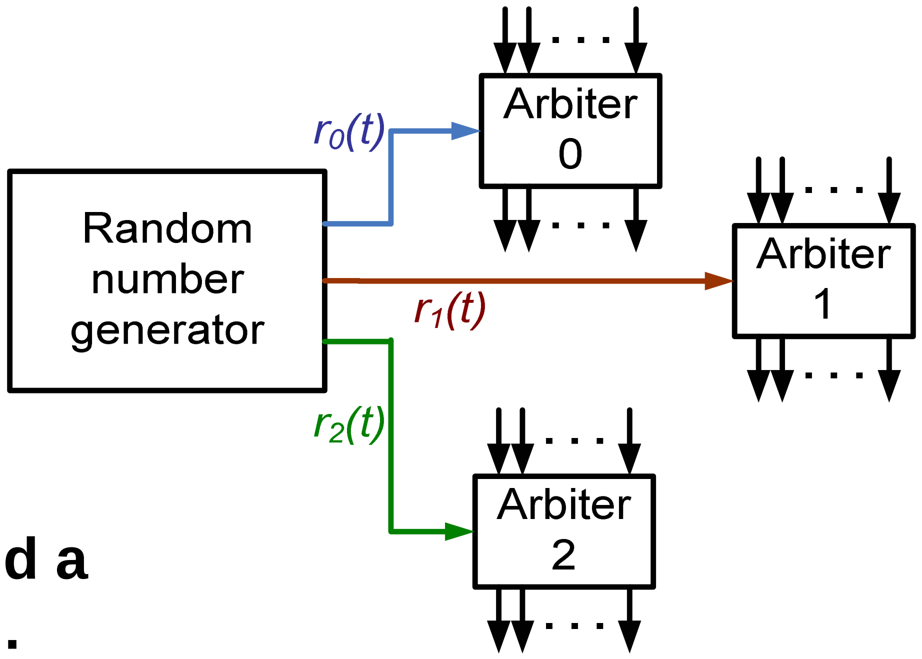
- **Any request can become the highest priority request at random**
- **Eg: request *i* gets its turn at time *t* when random number *r(t) = f(i)***



- **Goal: provide <u>unbiased</u> service to <u>all</u> requests**
- **Unfortunately random number generators have a large state space**
  - makes the verification problem challenging / hard

Why random priority-based arbiters?

- **Relatively less logic compared to FIFO and round-robin arbiters**
- **Amortization of logic: one random number generator catering several local arbiters on chip**



- **Sometimes it is hard to find a "good" arbitration policy…**

Random numbers

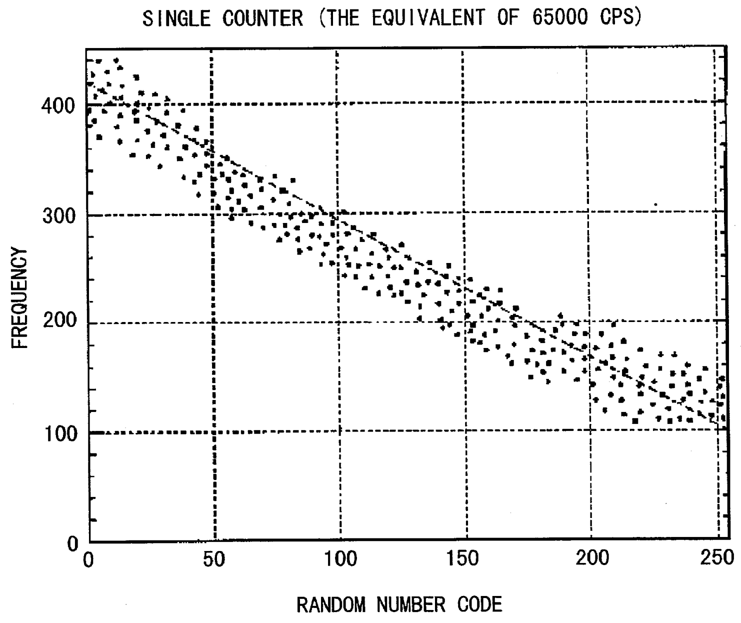
- **Main properties of random numbers**
  - Predictability
    - Must be highly unpredictable
    - No repeating sequences
  - Frequency distribution
    - Uniform frequency distribution: Each random number must have the same frequency in an infinitely long sequence

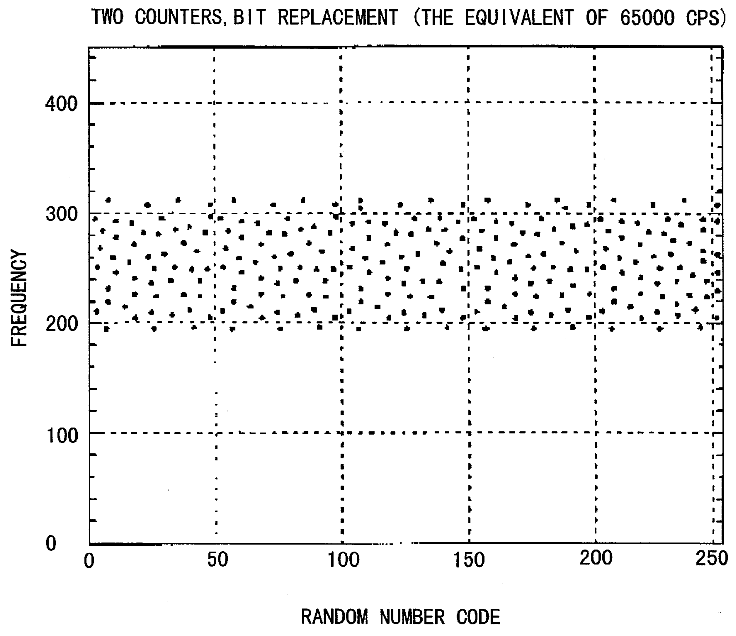
- **True random numbers are hard to generate → pseudo-random number generators**
  - LFSR (linear feedback shift register)
- **Pseudo-random number generators typically focus on the predictability and distribution properties of random numbers**

“Desirable” properties of random-numbers

- “bad” vs. “good” random number generator characteristics

J.Ikeda,“Random number generator which can generate random number based on a uniform distribution,” US22159590A1



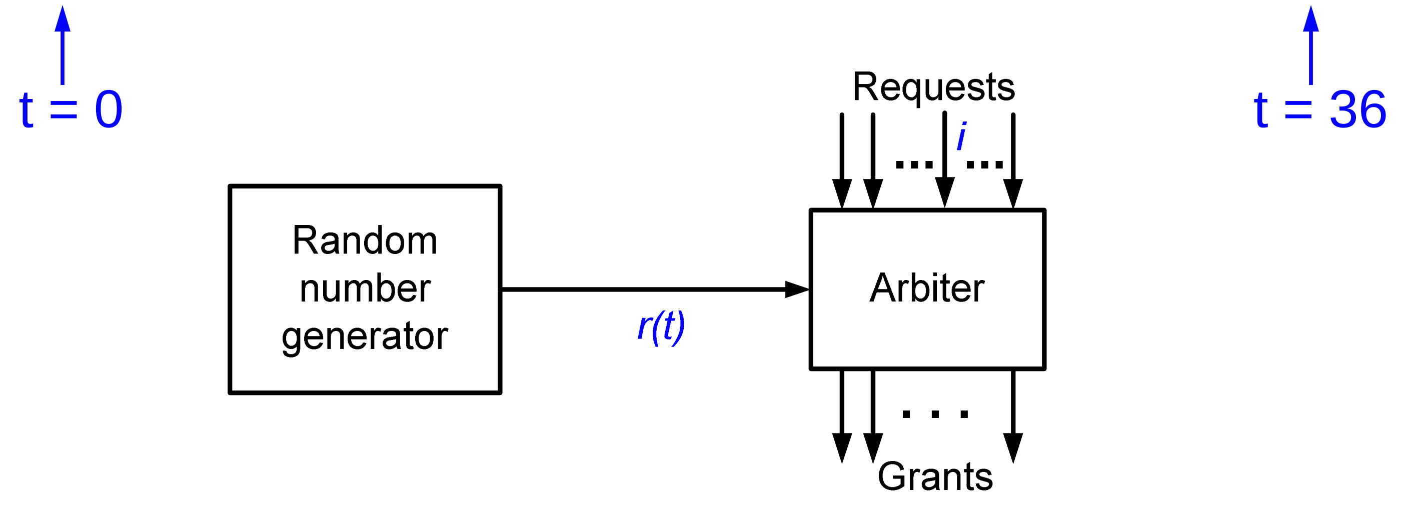


- Different applications (communication, cryptography, games, arbitration) requires different notions of randomness
- **What property of random number generators is important for arbiters?**

Fairness property of random number sequence

- **A “missing” random number in the sequence can delay a request until it shows up**
- **Request-to-grant delay is determined by how long the request needs to wait until the “missing” random number arrives**

..7521421425252563142525210046314252567214…



Fairness property of random number sequence

- **A “missing” random number in the sequence can delay a request until it shows up**
- **Request-to-grant delay is determined by how long the request needs to wait until the “missing” random number arrives**

$$..7\underbrace{5214214252525631425252100463142525567}_{\textbf{complete random sequence}}214\ldots$$

Complete Random Sequence

- A **Complete Random Sequence (CRS)** in a random number sequence is a contiguous sequence of random numbers that has all the possible unique random numbers at least once.

Complete Random Sequence

- A **Complete Random Sequence (CRS)** in a random number sequence is a contiguous sequence of random numbers that has all the possible unique random numbers at least once.
  - **Shortest CRS** has 2^N numbers – exactly one copy of each unique number
  - **Longest CRS** may be infinitely long
  - CRS may be used to characterize the fairness property of random number sequence and the random number generator (eg. LFSR)

Complete Random Sequence

- A **Complete Random Sequence (CRS)** in a random number sequence is a contiguous sequence of random numbers that has all the possible unique random numbers at least once.
  - **Shortest CRS** has 2^N numbers – exactly one copy of each unique number
  - **Longest CRS** may be infinitely long
  - CRS may be used to characterize the fairness property of random number sequence and the random number generator (eg. LFSR)
- Bounded fairness property can be specified in terms of the length of CRS (or the number of cycles needed to generate a CRS)
  - Eg: if the max length of CRS is ***L***, then any requests will be granted within ***L*** cycles
- A request may be starved for multiple CRSes – a property of arbiter

Bounded liveness in terms of CRS

- **Request-to-grant delay** is bounded by:

  C * max(length of a CRS), and

  C * min(length of a CRS)

  where ***C*** is the number of CRSes are needed for any request to get grant

- In order to satisfy bounded liveness property, **request-to-grant delay** must be less than a constant, the worst-case request-to-grant delay (a design specification).

Checking for bounded liveness (1)

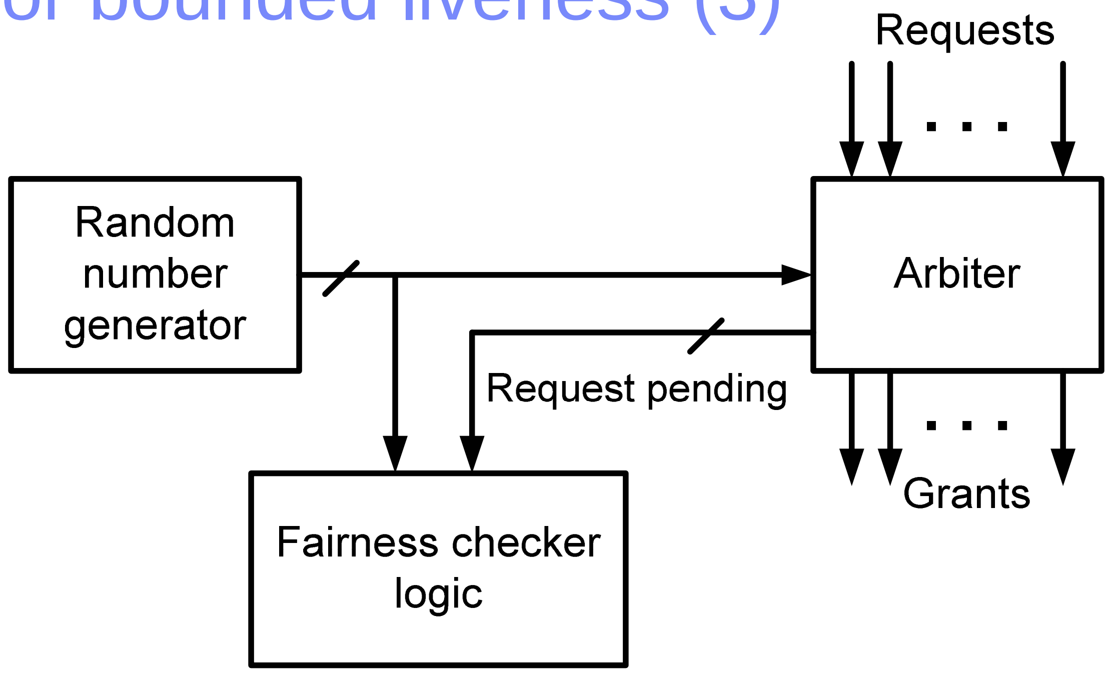
- **Number of CRSes needed to get grant is a property of the arbiter design**
  - Nondeterministic input can be used in lieu of the random number generator logic
  - Reduces the complexity of FV testbench (no LFSR logic)
  - Easy to obtain proofs or counter-examples

Checking for bounded liveness (2)

- **Number of CRSes needed to get grant is a property of the arbiter design**
  - Nondeterministic input can be used in lieu of the random number generator logic
  - Reduces the complexity of FV testbench (no LFSR logic)
  - Easy to obtain proofs or counter-examples

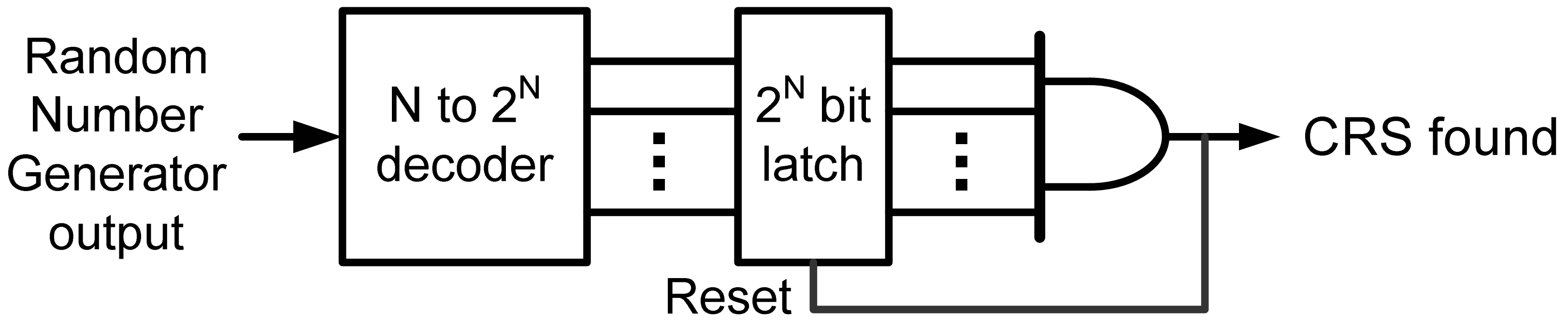
- **Max/min length of CRS is a property of the random number generator logic**
  - this parameter can be independently measured
    - Reduces the complexity of FV testbench (no arbiter logic)
  - Detect CRS in the sequences generated by LFSR logic
  - Iteratively determine longest and shortest CRS

Checking for bounded liveness (3)



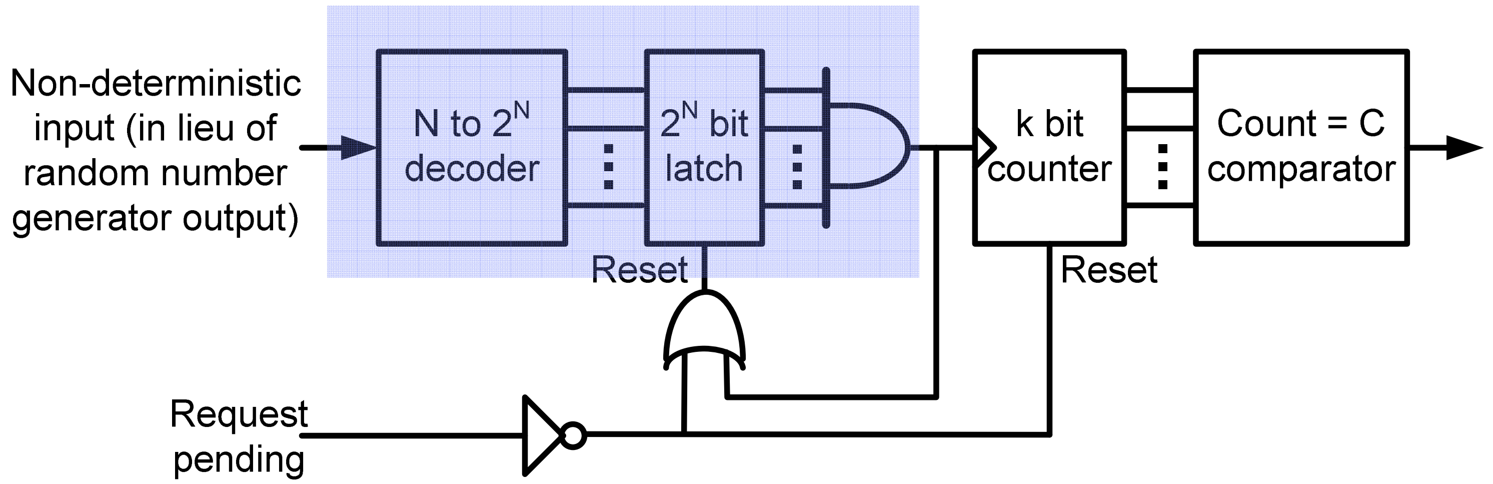
- **Step 1**: Determine request-to-grant delays in terms of CRSes
  – Basic idea: prove that request cannot be starved longer than ***C*** CRSes
- **Step 2**: Determine the length of (or time taken to generate) a CRS
- **Step 3**: compute request-to-grant delays bounds in terms of clock cycles
  – Request-to-grant delay = (largest value of ***C***) x (length of a CRS in cycles)

Detecting CRSes in random number sequences



- Set ***j*** th bit of the latch when the random number ***j*** is the input of the decoder
- ***CRS found = 1*** when all **2^N** bits of the latch are set

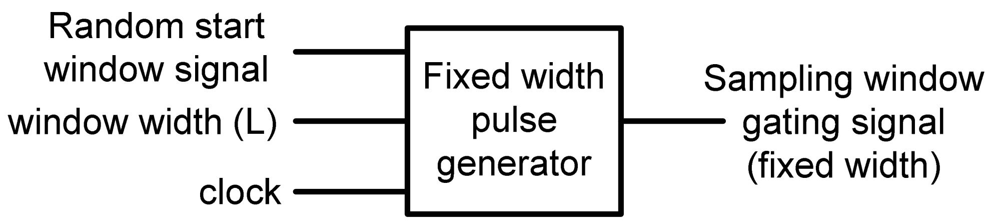
Step 1: Determining request-to-grant delays in CRS

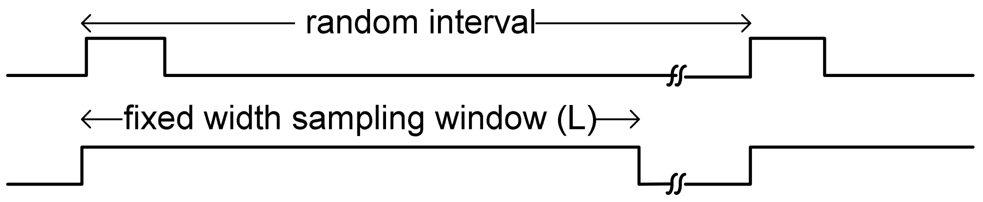


- **Count the number CRSes while the request is pending**
- **Bounded liveness as an iterative safety property check:**
  - ***request_pending AND (number_of_CRSes = C) == TRUE***
  - Find the largest value of ***C*** iteratively, where ***C* < *(2^k^ – 1)*** $C < (2^k - 1)$
  - ***k*** iterations (binary search for selecting values of ***C***)

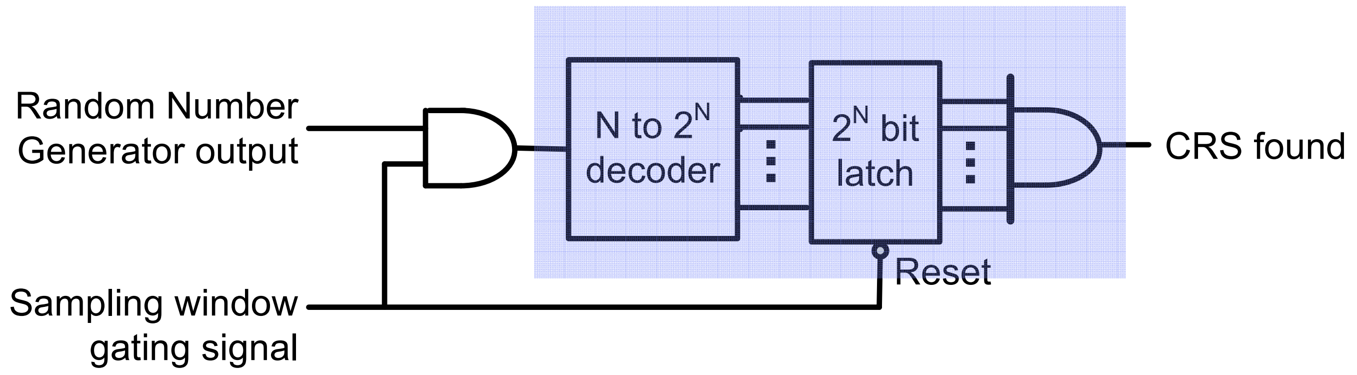
Step 2: Determine bounds of the CRS length (1)

- Basic idea:
- Take random *fixed length (L)* samples of random number sequences
  - A nondeterministic signal starts the sampling window
- Use a decision procedure to prove that the any sample of fixed length *L* contains exactly one CRS
- Vary the length of sampling window *L*


Random start window signal
window width (L)
clock
Fixed width pulse generator
Sampling window gating signal (fixed width)



Step 2: Determine bounds of the CRS length (2)



Logic for checking the fairness of random number generator

- The smallest and largest values of *L* can be determined by iteratively checking for "CRS found" is TRUE / FALSE
- Length of sampling window *L* is varied in each proof step

- For more accurate results, find length of C * CRSes instead of one CRS.
- A potential pitfall: some arbiters may not use the random numbers in *consecutive* cycles. Must look for CRSes in the "sampled sequence" as seen by the arbiter.

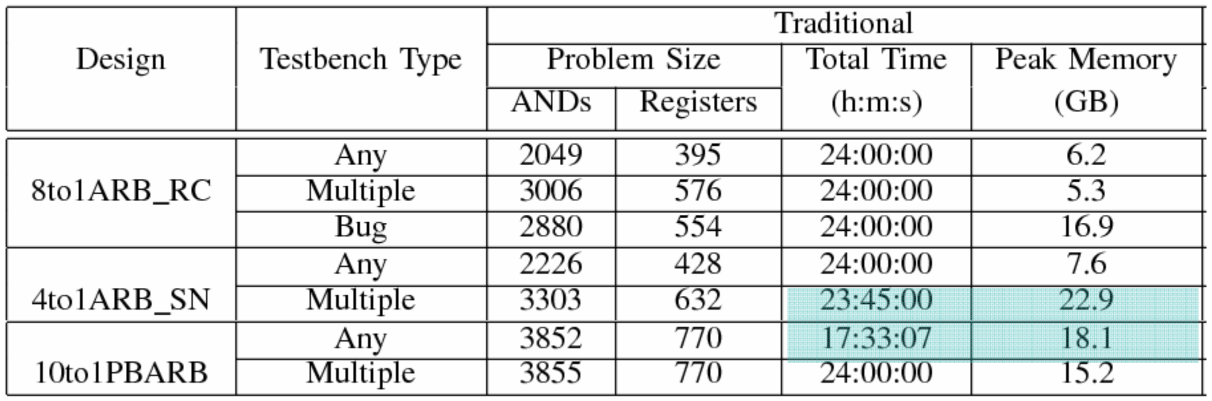
Experimental evaluation

- Arbiter designs used as benchmarks are taken from a set of industrial designs
  - cache directory port arbitration logic
  - command arbitration logic of an on-chip interconnection network controller
- 3 types of FV testbenches for bounded fairness checking
  - "Any": Can *any* request be starved?
  - "Multiple": Can *multiple* requests be starved?
  - "Bug": an arbiter design with a bug
- Used a 2-bit counter for counting the number of CRSes in all cases except in "bug" (larger counter)

Experimental Setup

- IBM's SixthSense (semi-) formal verification tool
  - Transformation-based verification approach
- Machine: 1.65 GHz POWER5+ processor, 384 GB memory
- Time out of 24 Hrs. for all experiments
- "Traditional" runs
  - Included the both arbiter and LFSR logic in the FV testbench
  - Used a 6 or 9-bit counter to check for request-to-grant delays:
    - (count < max_request_to_grant_delay) == TRUE

Experimental results

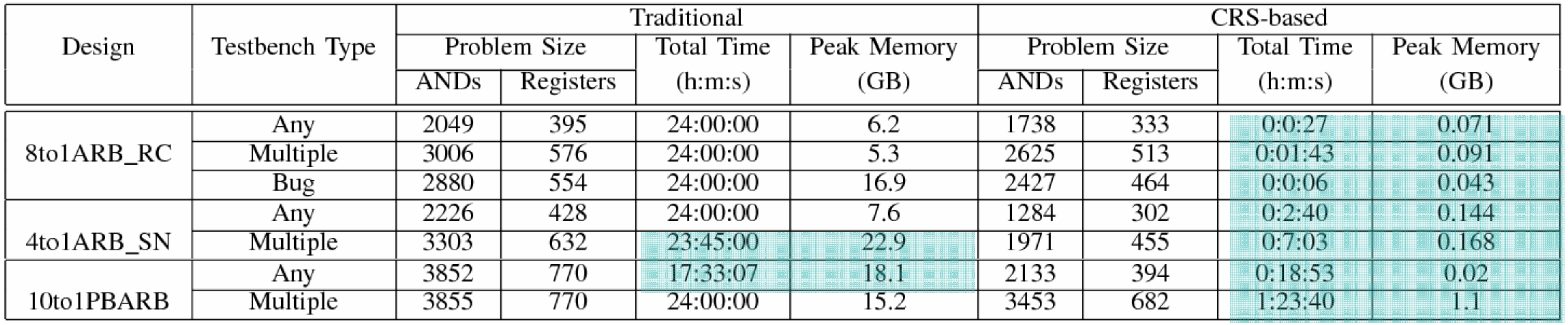


Design	Testbench Type	Traditional			
		Problem Size		Total Time	Peak Memory
		ANDs	Registers	(h:m:s)	(GB)
8to1ARB_RC	Any	2049	395	24:00:00	6.2
	Multiple	3006	576	24:00:00	5.3
	Bug	2880	554	24:00:00	16.9
4to1ARB_SN	Any	2226	428	24:00:00	7.6
	Multiple	3303	632	23:45:00	22.9
10to1PBARB	Any	3852	770	17:33:07	18.1
	Multiple	3855	770	24:00:00	15.2

- Problem size in AIG representation after initial COI reduction and before unrolling the design
- FV testbench consists of DUV, driver, and checker

- Number of CRSes: 1 – 7
- Length of CRS: 10’s – 100’s of cycles
- 16-stage LFSR: 2^{16} “random” states; many more state transitions…

Experimental results



Design	Testbench Type	Traditional				CRS-based			
		Problem Size		Total Time	Peak Memory	Problem Size		Total Time	Peak Memory
		ANDs	Registers	(h:m:s)	(GB)	ANDs	Registers	(h:m:s)	(GB)
8to1ARB_RC	Any	2049	395	24:00:00	6.2	1738	333	0:0:27	0.071
	Multiple	3006	576	24:00:00	5.3	2625	513	0:01:43	0.091
	Bug	2880	554	24:00:00	16.9	2427	464	0:0:06	0.043
4to1ARB_SN	Any	2226	428	24:00:00	7.6	1284	302	0:2:40	0.144
	Multiple	3303	632	23:45:00	22.9	1971	455	0:7:03	0.168
10to1PBARB	Any	3852	770	17:33:07	18.1	2133	394	0:18:53	0.02
	Multiple	3855	770	24:00:00	15.2	3453	682	1:23:40	1.1

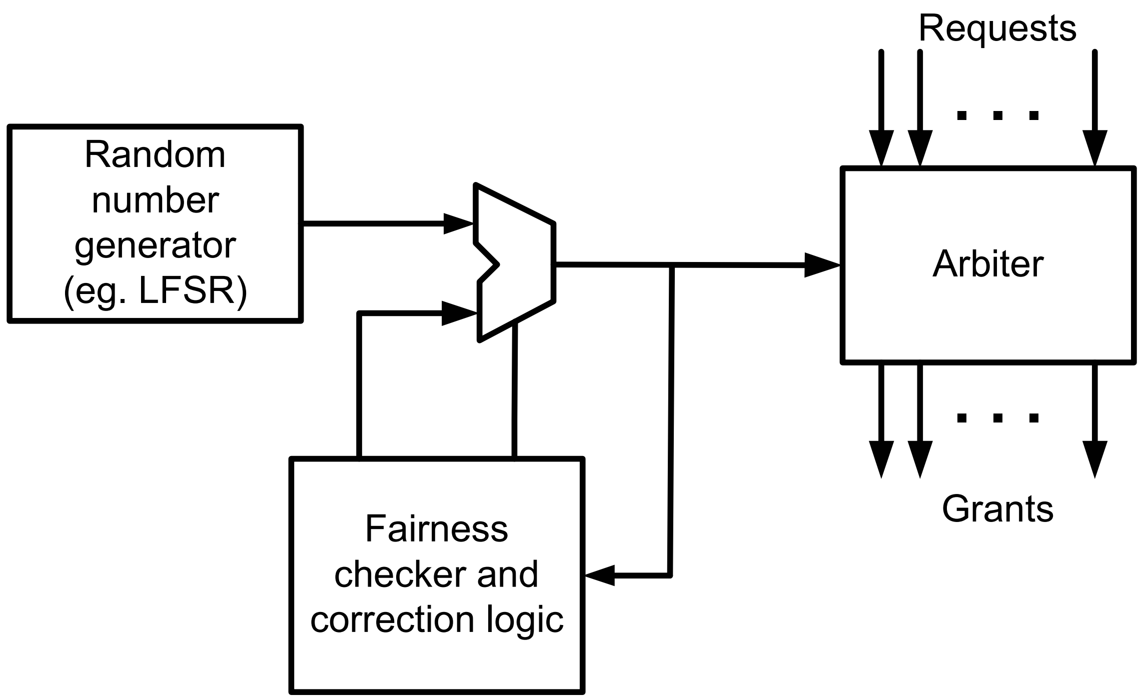
- Number of CRSes: 1 – 7
- Length of CRS: 10’s – 100’s of cycles
- 16-stage LFSR: **2^{16}** “random” states; many more state transitions…
- FV testbench for computing length of CRS (verification step 2)
  - Problem size (approx): 3000 ANDs, 600 REGs
  - Run time: 1.2 hrs avg. (few Secs to 6+ hrs)
- **CRS-based method significantly reduces the FV testbench complexity**
  - **Made the problem amenable for FV**

Other applications

- **Accurate modeling of request-to-grant delays for performance verification**

- **Tuning pseudo random number generators**
  - Selecting different set of tap points to reduce CRS length
  - Avoiding “interferences” due to tap points derived from same random number generator

Provably fair random number generator


Requests
Random number generator (eg. LFSR)
Arbiter
Grants
Fairness checker and correction logic

- An “add-on logic” for *any* random number generator to provide bounded fairness guarantees
- Basic idea: insert “missing” numbers to complete CRS in bounded time
- A “byproduct” of formal verification of arbiter designs! ☺

Summary

- **Our method effectively decouples the fairness logic from the actual arbitration logic, allowing checking the bounded liveness and fairness properties of each independently.**
- **Uses the notion of CRS to quantify the fairness properties of pseudo-random number generators and arbiters**
- **Can be applied to the verification of RTL directly ensuring correctness of the real logic, and does not require building any specialized models.**
  - Used to verify arbiters in IBM microprocessor designs
  - Detected a number of bugs and verified the fixes
- **Can be used for proving deadlock-free operation of arbiters as well as accurately computing request-to-grant delays**

Thank you!

Randomness, Random number generators, …

“Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number– there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method.”

– “Various techniques used in connection with random digits” by **John von Neumann** in *Monte Carlo Method* (1951).