COMS31700 Design Verification:

Block-level Case Study

with a demonstration of Formal Verification

Kerstin Eder

(Acknowledgement: I gratefully acknowledge the support from Cadence who provide the licenses for the Formal Verification Tool demonstration. Special thanks also to Anton Klotz from the Cadence Academic Network.)

University of BRISTOL

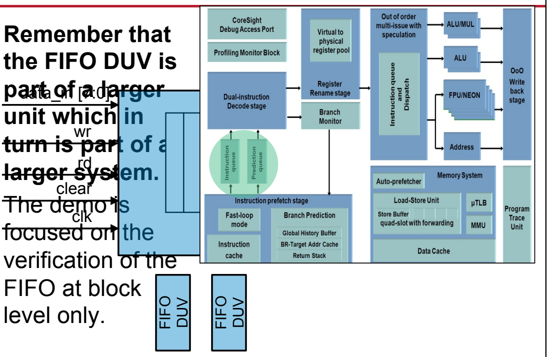
Department of COMPUTER SCIENCE

Case Study

Specification
Verification Plan
Directed Testing
(Code Coverage)
Functional Coverage
Assertion-based Verification
Formal Property Checking

Case Study: FIFO DUV

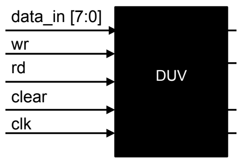
- **Remember that the FIFO DUV is ~~part of a larger unit which in turn is part of a larger system.~~**
- ~~The demo's focused on~~ the verification of the FIFO at block level only.



Specification

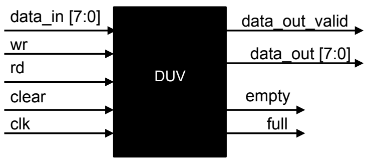
FIFO DUV as for ABV session

Example DUV Specification - Inputs



- Inputs:
  - wr indicates valid data is driven on the data_in bus
  - data_in is the data to be pushed into the DUV
  - rd pops the next data item from the DUV in the next cycle
  - clear resets the DUV

Example DUV Specification - Outputs



- Outputs:
  - data_out_valid indicates that valid data is driven on the data_out bus
  - data_out is the data item requested from the DUV
  - empty indicates that the DUV is empty
  - full indicates that the DUV is full

DUV Specification

- High-Level functional specification of DUV
  - The design is a FIFO.
  - Reading and writing can be done in the same cycle.
  - Data becomes valid for reading one cycle after it is written.
  - No data is returned for a read when the DUV is empty.
  - Clearing takes one cycle.
  - During clearing read and write are disabled.
  - Inputs arriving during a clear are ignored.
  - The FIFO is 8 entries deep.

7

Verification Plan

Those who fail to plan, plan to fail.

The Verification Plan

- Functions to be verified:
  - For each level in the design hierarchy, list all the functions that will be verified at that level.
  - In particular, identify corner cases for the design.
- Methods of verification:
  - Define which verification methods to use, e.g. simulation (directed or random), formal, etc.
- Completion criteria:
  - Define the measurements/metrics that indicate that verification is complete.
  - In particular, define coverage models and targets.
- Resources required (people) and schedule details:
  - Integrate the verification plan into the overall design plan and estimate the cost of verification.
- Required tools:
  - List the software and hardware necessary to perform verification.

9

The Verification Plan

- Functions to be verified:
  - For each level in the design verified at that level.
  - In particular, identify corner
- Methods of verification:
  - Def irected or ran
- Comp
  - Def omplete.
  - In p
- Resources required (people) and schedule details:
  - Integrate the verification plan into the overall design plan and estimate the cost of verification.
- Required tools:
  - List the software and hardware necessary to perform verification.

Verification Plans are "live" documents. They change as our understanding of the DUV increases.

The Verification Plan is the **Specification** for the Verification Process

10

Test Scenarios Matrix - Basic

Test #	Description
1.1	Check the write functionality
1.2	Check the read functionality
1.3	Check the reset functionality
1.4	Check …

Develop a Test Scenarios Matrix for our Case Study DUV.

NOTE: *"Check that X"* should be read as "Create a scenario that allows checking X".

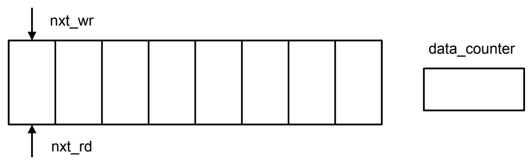
- These generic tests should be broken to more specific tests
  - Test case 1.1.1: Check write when **empty**
  - Test case 1.1.2: Check write when **full**
  - Test case 1.1.3: Check write during **reset**
  - …

11

White Box View DUV Implementation

Example DUV Implementation

- Implementation based on a circular buffer
  - nxt_wr and nxt_rd pointers indicate where the next entry will be written to or read from.
  - data_counter indicates the number of valid data items in the FIFO.
  - Complex control logic for pointers and counter.



13

Functional Coverage

Cross-Product Functional Coverage

[O Lachish, E Marcus, S Ur and A Ziv. Hole Analysis for Functional Coverage Data. In proceedings of the 2002 Design Automation Conference (DAC), June 10-14, 2002, New Orleans, US.]

A cross-product coverage model is composed of the following parts:

1. A **semantic description** of the model (**story**)
2. A list of the **attributes** mentioned in the story
3. A set of all the **possible values** for each attribute (**the attribute value domains**)
4. A **list of restrictions** on the **legal combinations** in the cross-product of attribute values

15

FIFO Cross Product Coverage Model

- From a “White Box” verification perspective:
  - The FIFO is implemented using a circular buffer.
  - The circular buffer implementation is based on the following control signals:
    - nxt_rd, nxt_wr, data_counter
  - These signals are used to control the data flow and also the empty and full signals.
  - **Verification Plan:**
    - *“Interactions of read and write transactions can create complex and unexpected conditions. All combinations need to be verified to gain confidence in the correctness of the FIFO.”*
- This is the story.

16

FIFO Cross Product Coverage Model

- Attributes relevant to the coverage model:
  - nxt_rd, nxt_wr, data_counter, empty, full signals
- Attribute value domains:
  - nxt_rd ϵ {0,1,2,3,4,5,6,7}
  - nxt_wr ϵ {0,1,2,3,4,5,6,7}
  - data_counter ϵ {0,1,2,3,4,5,6,7,8}
  - empty ϵ {0,1}
  - full ϵ {0,1}
- Full coverage space:
  - 8*8*9*2*2 = **2304 coverage tasks**
  - Format: (nxt_rd, nxt_wr, data_counter, empty, full)
  - Find some legal and some illegal examples

17

FIFO Cross Product Coverage Model

- Attributes relevant to the coverage model:
  - nxt_rd, nxt_wr, data_counter, empty, full signals
- Attribute value domains:
  - nxt_rd ϵ {0,1,2,3,4,5,6,7}
  - nxt_wr ϵ {0,1,2,3,4,5,6,7}
  - data_counter ϵ {0,1,2,3,4,5,6,7,8}
  - empty ϵ {0,1}
  - full ϵ {0,1}
- Full coverage space:
  - 8*8*9*2*2 = **2304 coverage tasks**
  - Format: (nxt_rd, nxt_wr, data_counter, empty, full)
  - Find some legal and some illegal examples

18

FIFO Cross Product Coverage Model

- Attributes relevant to the coverage model:
  - nxt_rd, nxt_wr, data_counter, empty, full signals
- Attri
  - nxt_
  - nxt_
  - dat
  - emp
  - full ϵ {0,1}
- Full coverage space:
  - 8*8*9*2*2 = **2304 coverage tasks**
  - Format: (nxt_rd, nxt_wr, data_counter, empty, full)
  - Find some legal and some illegal examples

Legal Coverage Tasks:
(0,0,0,1,0)
(2,2,8,0,1)
(3,7,4,0,0)

Illegal Coverage Tasks:
(0,0,0,1,1)
(1,1,4,0,0)
(1,5,0,1,1)

19

FIFO Cross Product Coverage Model

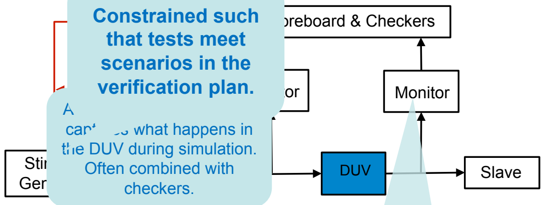
- Restrictions
  - data_counter is only important when nxt_rd==nxt_wr so that one can tell the difference between empty and full
  - 64 combinations of nxt_rd and nxt_wr
  - 56 cases are for nxt_rd!=nxt_wr, so er
  - 8 cases are for nxt_rd==nxt_wr
  - one set of 8 is for data_counter==0, sc
  - one set of 8 is for data_counter==8, sc
  - empty and full are not both asserted at
- Revised format of coverage model:
  - (nxt_rd, nxt_wr, empty, full)
  - total size of coverage space: 8*8*2*2 = 256 coverage tasks
  - Encode assumptions into properties
  - Only 56+8+8 = 72 coverage tasks are legal and meaningful.
- It is worth noting the close link of the above coverage model to the properties for formal verification.

Defining meaningful functional coverage requires design understanding and engineering skill.

20

Constraint Pseudo Random Test Generation

Advanced TB Architecture



Constrained such that tests meet scenarios in the verification plan.

A
car‌ ‌ what happens in the DUV during simulation. Often combined with checkers.

Captures complete interface protocols by combining several checkers.

- Constrained pseudo-random stimulus generation
- Self-checking TB
  - Monitors, Scoreboarding
- Coverage Collection and Analysis

Promote a Coverage-Driven Verification Methodology

22

Test Scenarios Matrix – Advanced

Topic	Test #	Description
Data checking	2.1	Data is not modified
	2.2	
	2.3	
	2.4	

23

Test S... Matrix – Advanced

Basic scoreboard functionality.

Topic		Description
Data checking	2.1	Data is not modified
	2.2	Data is not lost
	2.3	Data is not duplicated
	2.4	Data order is maintained

24

Test Scenarios Matrix – Advanced

Topic	Test #	Description
Data checking	2.1	Data is not modified
	2.2	Data is not lost
	2.3	Data is not duplicated
	2.4	Data order is maintained
Corner cases	2.3	Reading and writing at the same time
	2.3.1	
	2.3.2	
	…	
	…	
	…	

Test Scenarios Matrix – Advanced

Topic	Test #	Description
Data checking	2.1	Data is not modified
	2.2	Data is not lost
	2.3	Data is not duplicated
	2.4	Data order is maintained
Corner cases	2.3	Reading and writing at the same time
	2.3.1	
	2.3.2	
	…	
	…	
	…	

Test Scenarios Matrix – Advanced

Topic	Test #	Description
Data checking	2.1	Data is not modifie
	2.2	Data is not lost
	2.3	Data is not duplica
	2.4	Data order is maintained
Corner cases	2.3	Reading and writing at the same time
	2.3.1	Reading and writing at the same time when empty
	2.3.2	Reading and writing at the same time when full
	…	... at the same time as clearing
	…	
	…	

These make interesting functional coverage scenarios.

How?

Should we rely on random generation by constraining stimulus to make these rare events happen more often?

Bug Hunting

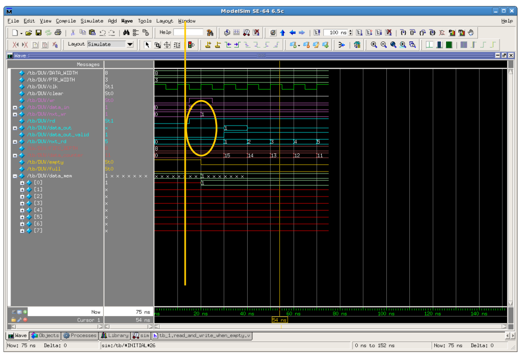
Given the following bug...

- Concurrent reading from and writing to FIFO
  - Should the data counter change its value?

Given the following bug...

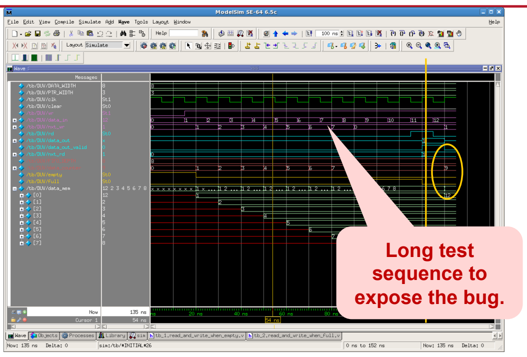
- Concurrent reading from and writing to FIFO
  - ok, the data counter should not change its value
  - unless reading from and writing to the same data slot
  - this only happens when the FIFO is
    - **empty:** When the FIFO is empty and there is a write at the same time as a read (from empty), then the read should be ignored.
    - **full:** When the FIFO is full and there is a read at the same time as a write, then the write (to full) should be ignored.
  - But the logic that controls the value of the data counter does not distinguish these special cases.
  - **What do we need to do to find these bugs?**

Read/Write when FIFO is empty



31

Read and Write when FIFO is full



32

ABV

FIFO DUV as for ABV session

Revision - Properties of the DUV

Black box view:

- Empty and full are never asserted together.
- After clear the FIFO is empty.
- After writing 8 data items the FIFO is full.
- Data items are moving through the FIFO unchanged in terms of data content and in terms of data order.
- No data is duplicated.
- No data is lost.
- data_out_valid only for valid data, i.e. no x's in data.

An invariant property.

34

Revision - Properties of the DUV

- White box view:
  - The value range of the read and write pointers is between 0 and 7.
  - The data_counter ranges from 0 to 8.
  - The data in the FIFO is not changed during a clear.
  - For each valid read the read pointer is incremented.
  - For each valid write the write pointer is incremented.
  - Data is written only to the slot indicated by nxt_wr.
  - Data is read only from the slot indicated by nxt_rd.
  - When reading and writing the data_counter remains unchanged provided the DUV is neither empty nor full.

35

Property Formalization

Formalization of key DUV Assertions

- **System Verilog Assertions (SVA) for:**
  - Empty and full are never asserted together.

```
property not_empty_and_full;
@(posedge clk) !(empty && full);
endproperty
assert property (not_empty_and_full);
```

  - After clear the FIFO is empty.

```
property empty_after_clear;
@(posedge clk) (clear |=> empty==1);
endproperty
assert property (empty_after_clear);
```

  - On empty after one write the FIFO is no longer empty.

```
property not_empty_after_write_on_empty;
@ (posedge clk) (empty && wr |=> !empty);
endproperty
assert property (not_empty_after_write_on_empty);
```

Assertions can be monitored during simulation.

Assertions can also be used for formal property checking.

37

Formalization of key DUV Assertions

- **System Verilog Assertions (SVA) for:**
  - Empty and full are never asserted together.

```
property not_empty_and_full;
@(posec
endprop
assert
```

  - After c

```
propert
@(posec
endprop
assert property (empty_after_clear);
```

  - On empty after one write the FIFO is no longer empty.

```
property not_empty_after_write_on_empty;
@ (posedge clk) (empty && wr |=> !empty);
endproperty
assert property (not_empty_after_write_on_empty);
```

ertions can be nitored during simulation.

ertions can also used for formal perty checking.

Challenge:
There are many more interesting assertions.

38

Corner case properties

- **FIFO empty:** When the FIFO is empty and there is a write at the same time as a read (from empty), then the read should be ignored.

```
property empty_write_ignore_read;
@(posedge clk){empty && wr && rd |=>
                    data_counter == $past(data_counter)+1};
endproperty
assert property (empty_write_ignore_read);
cover property (empty_write_ignore_read);
```

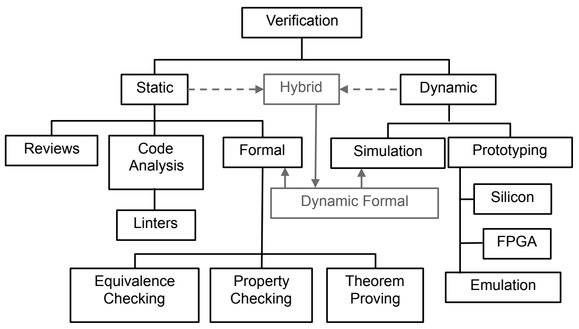
- **FIFO full:** When the FIFO is full and there is a read at the same time as a write, then the write (to full) should be ignored.

```
property full_read_ignore_write
@ (posedge clk) {full && rd && wr |=>
                    data_counter == $past(data_counter)-1};
endproperty
assert property (full_read_ignore_write);
cover property (full_read_ignore_write);
```

39

Formal Property Checking

Functional Verification Approaches



41

Formal Property Checking

Properties of a design are formally proven or disproved.

- Used to check for generic problems or violations of user-defined properties of the behaviour of the design.
- Usually employed at **higher levels** of abstractions.

Give a reconvergence model for formal property checking!

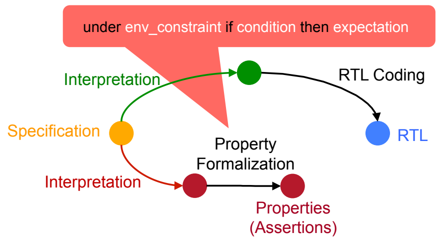
A **reconvergence model** is a conceptual representation of the verification process. It helps us understand **what is being verified.**

42

Formal Property Checking

Properties of a design are formally proven or disproved.

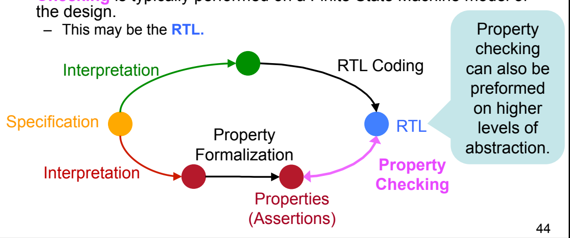
- Used to check for generic problems or violations of user-defined properties of the behaviour of the design.
- Usually employed at **higher levels** of abstractions.
- **Properties** are derived from the specification. (interpretation step)
- **Properties** are expressed as formulae in some (temporal) logic.



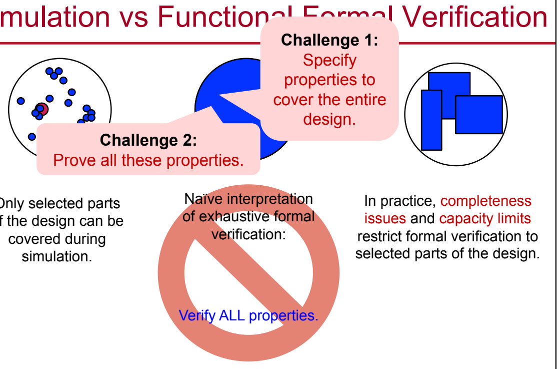
Formal Property Checking

Properties of a design are formally proven or disproved.

- Used to check for generic problems or violations of user-defined properties of the behaviour of the design.
- Usually employed at **higher levels** of abstractions.
- **Properties** are derived from the specification. (interpretation step)
- **Properties** are expressed as formulae in some (temporal) logic.
- **Checking** is typically performed on a Finite State Machine model of the design.
  - This may be the **RTL.**



Simulation vs Functional Formal Verification


Challenge 1: Specify properties to cover the entire design.
Challenge 2: Prove all these properties.
Naïve interpretation of exhaustive formal verification:
Verify ALL properties.

Only selected parts of the design can be covered during simulation.

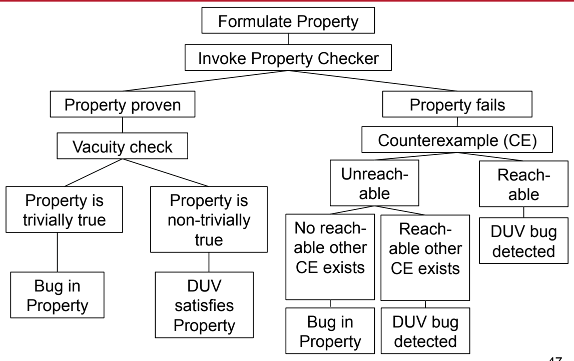
In practice, completeness issues and capacity limits restrict formal verification to selected parts of the design.

[B. Wile , J.C. Goss and W. Roesner, "Comprehensive Functional Verification – The Complete Industry Cycle", Morgan Kaufman, 2005]

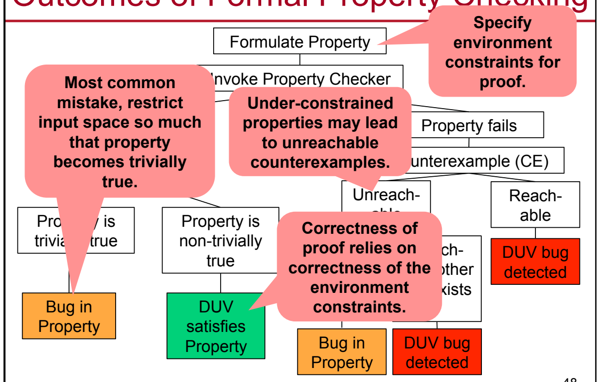
The Role of Formal Property Checking

- Property Checking is the most common form of high-level formal verification used in practice.
  - Property checking is fully automatic.
    - Requires the properties to be written.
  - It performs exhaustive verification of the design ***wrt the specified properties.***
  - It provides proofs and can demonstrate the absence of bugs.
  - A ***counterexample*** is presented for failed properties.
  - Used for critical, well specified parts of the design
    - Cache coherence protocols, Bus protocols, Interrupt controllers
- Formal Methods can suffer from **capacity limits**
  - There are tried and trusted techniques to overcome these:
    - Start with narrow focus on block level, work up towards higher levels in the design hierarchy turning assertions into assumptions
    - Restrict property checking to work over finite small time windows.
    - Limit environment behaviour by strengthening constraints.
    - Case splits over a set of properties, partitioning and black boxing.

Outcomes of Formal Property Checking



Outcomes of Formal Property Checking


Formulate Property
Specify environment constraints for proof.
Invoke Property Checker
Most common mistake, restrict input space so much that property becomes trivially true.
Under-constrained properties may lead to unreachable counterexamples.
Property fails
Counterexample (CE)
Property is trivially true
Property is non-trivially true
Unreach-able
Reach-able
Correctness of proof relies on correctness of the environment constraints.
DUV bug detected
Bug in Property
DUV satisfies Property
Bug in Property
DUV bug detected

How do you know you've encoded the property right?

- Keep properties and sequences simple.
  - Build complex properties from simple, short properties

- **Peer review properties you write.**
- **If the property fails, you can investigate the counterexample:**
  - Is it reachable or not?
- **But if the property succeeds, how do you know whether you've encoded the property right?**

49

Formal Property Checking

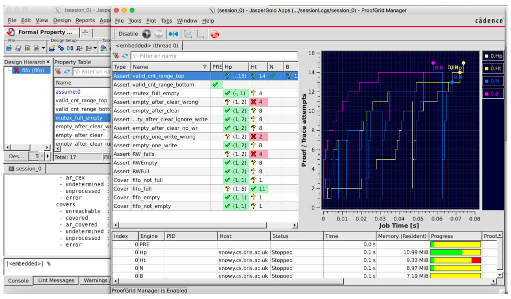
- Property checking tools can formally verify assertions.
  - **Basic properties (visualize):**
    - Basic functionality
    - Range checks
  - **Re-use SV Assertions as properties (check):**
    - Empty and full are never asserted together.
    - After clear the FIFO is empty.
    - On empty after one write the FIFO is no longer empty.
  - Understanding counterexamples:
    - Debug a failed property

Note:
- Closely related to functional coverage.
- Link from env_constraints to simulation assertions.

50

Formal Property Checking

- Jasper DEMO
  - Formal verification of selected FIFO properties from ABV



Name
assume:0
valid_cnt_range_top
valid_cnt_range_bott
mutex_full_empty
empty_after_clear_wr
empty_after_clear
empty_after_clear_ig

Type	Name	PRE	Hp	Ht	N	B
Assert	valid_cnt_range_top		? ...15)	? 14	✓	? 1
Assert	valid_cnt_range_bottom	✓				
Assert	mutex_full_empty		✓ (-, 1)	? 4		
Assert	empty_after_clear_wrong		? (1, 2)	✗ 4		
Assert	empty_after_clear		✓ (1, 2)	? 8		
Assert	...ty_after_clear_ignore_write		✓ (1, 2)	? 8		
Assert	empty_after_clear_no_wr		✓ (1, 2)	? 8		
Assert	empty_one_write_wrong		? (1, 2)	✗ 2		
Assert	empty_one_write		✓ (1, 2)	? 8		
Assert	RW_fails		? (1, 2)	✗ 4		
Assert	RWEmpty		✓ (1, 2)	? 8		
Assert	RWFull		✓ (1, 2)	? 8		
Cover	fifo_not_full		✓ (1, 1)	? 1		
Cover	fifo_full		? (1, 5)	✓ 11		
Cover	fifo_empty		✓ (1, 1)	? 1		
Cover	fifo_not_empty		✓ (1, 1)	? 1		

```
session_0
    - ar_cex              :
    - undetermined        :
    - unprocessed         :
    - error               :
  covers                  :
    - unreachable         :
    - covered             :
    - ar_covered          :
    - undetermined        :
    - unprocessed         :
    - error               :
[<embedded>] %
```

Index	Engine	PID	Host	Status	Time	Memory (Resident)	Progress	Proof
0	PRE				0.0 s			
0	Hp		snowy.cs.bris.ac.uk	Stopped	0.1 s	10.99 MiB		
0	Ht		snowy.cs.bris.ac.uk	Stopped	0.1 s	9.33 MiB		
0	N		snowy.cs.bris.ac.uk	Stopped	0.1 s	8.97 MiB		
0	B		snowy.cs.bris.ac.uk	Stopped	0.1 s	7.19 MiB		

51

How big is Exhaustive?

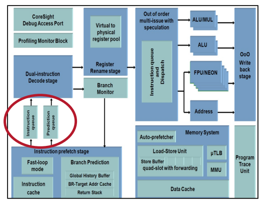
- Consider simulating a typical CPU design
  - 500k gates, 20k DFFs, 500 inputs
  - 70 billion sim cycles, running on 200 linux boxes for a week
  - **How big: 2^{36} cycles**
- Consider formally verifying this design
  - Input sequences: cycles $2^{(\text{inputs+state})} = 2^{20500}$
  - What about X's: 2^{15000} (5,000 X-assignments + 10,000 non-reset DFFs)
  - **How big: 2^{20500} cycles** (2^{15000} combinations of X is not significant here!)
- That's a big number!

– Cycles to simulate the 500k design:	$\mathbf{2^{36}}$	(70 billion)
– Cycles to formally verify a 32-bit adder:	2^{64}	(18 billion billion)
– Number of stars in universe:	2^{70}	(10^{21})
– Number of atoms in the universe:	2^{260}	(10^{78})
– Possible X combinations in 500k design:	2^{15000}	(10^{4515} x 3)
– Cycles to formally verify the 500k design:	$\mathbf{2^{20500}}$	(10^{6171})

52

Summary

- Block-level Case Study
  - Specification
  - Verification Plan
  - Directed Testing
  - (Code Coverage)
  - Functional Coverage
  - Assertion-based Verification
  - Formal Property Checking



- **No single method is adequate to cover a whole design in practice.**
  - Carefully select the verification methods that maximize ROI.
  - Complement simulation with formal: Integrated approach

53

Merry Christmas and a Happy New Year

Why red wine is so important for Christmas

54

[redacted]