System On Chip Design And Modelling

Second half slides (LG 5-8)

CST Part II, 12 lectures

Lent Term 2009

Dr. David Greaves

David.Greaves@cl.cam.ac.uk

LG5: Assertion-Based Design (ABD)

Topics: Assertion-Based Design, Assertion Synthesis.

LG5.1 - ABD - Assertion-Based Design
LG5.2 - ABD - The alternative: Simulation
LG5.3 - ABD - Formally Synthesised Bus Monitor
LG5.4 - ABD - PSL Assertion, General Structure
LG5.5 - ABD - PSL Extended Regular Expressions
LG5.6 - ABD - PSL Properties and Macros
LG5.7 - ABD - Naive pattern to RTL Automaton
LG5.8 - ABD - SERES Pattern Matching Example
LG5.9 - ABD - PSL Temporal Layer Operators
LG5.10 - ABD - PSL Overall Layered Architecture
LG5.11 - ABD - Sequence Constraint as a Suffix Implication
LG5.12 - ABD - A Simple Model Checker
LG5.13 - ABD - Boolean Equivalence Checker
LG5.14 - ABD - Sequential Logic Equivalence: Example
LG5.15 - ABD - Sequential Equivalence Checker
LG5.16 - ABD - Sequential Logic Simplification
LG5.17 - ABD - Sequential Logic Equivalence
LG5.18 - ABD - Automated Stimulus Generation

LG5.1 - ABD - Assertion-Based Design

ABD $\rightsquigarrow$ Writing assertions *before* coding.

ABD $\rightsquigarrow$ Writing assertions as coding progresses.

ABD $\rightsquigarrow$ Structuring testing around assertions.

Assertions are (combinations of):

- Imperative safety checks (like assert.h in C++)
- Declarative safety properties.
- Strong properties about liveness and deadlock.

All three can potentially be proved by formal methods.

Simulation can sometimes find safety violations and sometimes find deadlock but it cannot prove liveness.

ABD $\rightsquigarrow$ Make the designer/programmer think!

Assertions can be imported from previous designs or other parts of the same design for global consistency.

ABD $\rightsquigarrow$ shows up corner case problems not encountered in simulation.

A formally-verified result may be required by the customer.

LG5.2 - ABD - The alternative: Simulation

Extensive simulation: overnight regression testing.

RTL yes/no automaton as part of the test bench.

Write output to file and diff against golden version with perl script.

Simulation problems: non-exhaustive, time consuming.

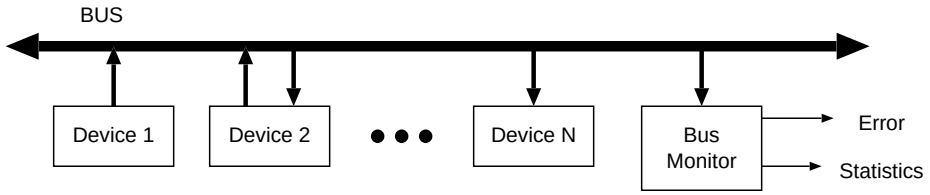
Simulations can be used for some production test vectors.

Partly formal: Bus monitors, Specman/VERA constrained pattern generators.

Full formal check benefit: Completeness, Scale, Overlapping occurrences can be tracked.

Simulations are needed for performance analysis and general design confidence.

LG5.3 - ABD - Formally Synthesised Bus Monitor



A bus monitor connects to the net-level bus.

NB: No tri-states in a SoC: everything is multiplexors.

Monitor can keep statistics as well as detect protocol violations.

Can it be synthesised from a formal spec ?

(`http://www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/transactors`)

Coverage: What percentage of rule disjuncts held as dominators (on their own) ?

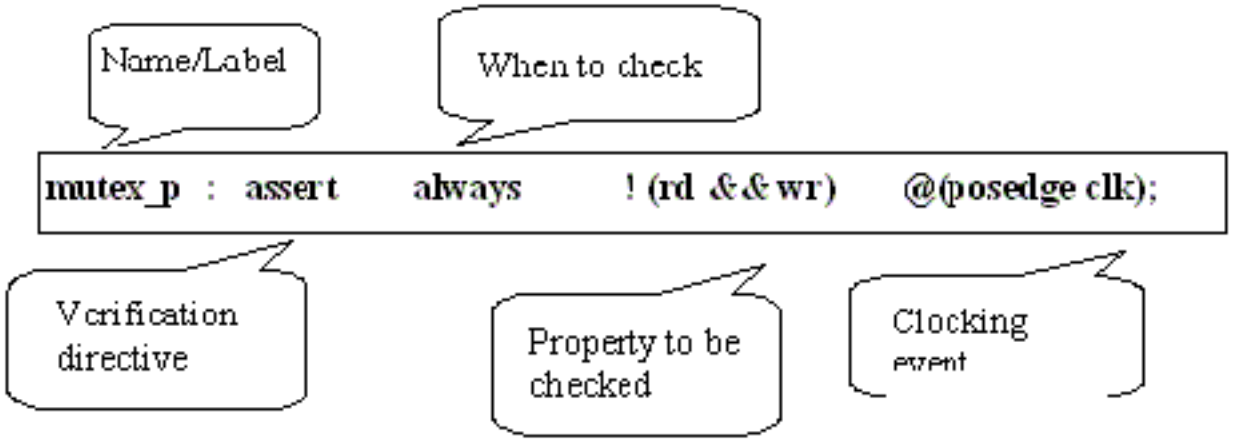
Coverage: What (inverse log) percentage of reachable state space was spanned?

There is no clear definition of 100 percent coverage!

LG5.4 - ABD - PSL Assertion, General Structure

From `http://www.project-veripage.com/psl_tutorial_2.php`

General structure of a PSL assertion



- always
- never
- eventually!
- next (family of them).

The `always` operator is the most frequently used one and it specifies that the following property expression should be checked every clock.

LG5.5 - ABD - PSL Extended Regular Expressions

PSL Sequences: curly braces: SERES (Sugar Extended Regular Expression)

Sequence elements are state predicates from Modelling and Boolean layers.

Core operators: Disjunction, Concatenation, Arbitrary repetition.

As a temporal logic: interpret concatenation as a time sequencing.

- **A;B** Semicolon denotes sequence concatenation
- **A[*]** Postfix asterisk for arbitrary repetition
- **A|B** Vertical bar (stile) for alternation.

Make richer with additional operators:

- **A[+]** One or more occurrences: { **A;A[*]** }
- **A[*n]** Repeat `n` times
- **A[=n]** Repeat `n` times non-consecutively
- **A:B** Fusion concatenation (last of A occurs during first of B)

Further repetition operators denote repeat count ranges.

Repeat counts must be compile-time constant.

LG5.6 - ABD - PSL Properties and Macros

PSL defines some simple path to state macros

- `rose(X)` means { **!X;X** }
- `fell(X)` means { **X; !X** }

Others are easy to define:

- `stable(X)` can be defined as { **X; X** } $\vee$ { **!X; !X** }
- `changed(X)` can be defined as { **X; !X** } $\vee$ { **!X; X** }

LG5.7 - ABD - Naive pattern to RTL Automaton

```
fun gen_pattern_matcher g (seres_statexp e) = gen_and2(g, gen_boolean e)

|   gen_pattern_matcher g (seres_diop(diop_seres_alternation, l, r)) =
    let val l' = gen_pattern_matcher g l
        val r' = gen_pattern_matcher g r
        in gen_or2(l', r') end

|   gen_pattern_matcher g (seres_diop(diop_seres_catenation, l, r)) =
    let val l' = gen_dff(gen_pattern_matcher g l)
        val r' = gen_pattern_matcher l' r
        in r' end

|   gen_pattern_matcher g (seres_diop(diop_seres_fusion, l, r)) =
    let val l' = gen_pattern_matcher g l
        val r' = gen_pattern_matcher l' r
        in r' end

|   gen_pattern_matcher g (seres_monop(mono_arb_repetition, l)) =
        let val nn = newnet()
            val l' = gen_pattern_matcher nn l
            val r = gen_or2(l', g)
            val _ = gen_buffer(nn, r)
            in r end

|   gen_pattern_matcher g (seres_diop(diop_n_times_repetition, l,
                          seres_statexp(x_num n))) =
        let fun f (g, k) = if k=0 then g else
            gen_pattern_matcher (f(g, k-1)) l
            in f (g, n) end
```

However: temporal operators are harder: e.g. length-matching conjunction.

LG5.8 - ABD - SERES Pattern Matching Example

Suppose four events are supposed to always happen in sequence:

Consider `always` { **true[*]; A; B; C; D** }

Basic pattern matcher applied to { **A;B;C;D** } generates:

```
  DFF(g0, A, clk);
  AND2(g1, g0, B);
  DFF(g2, g1, clk);
  AND2(g3, g2, C);
  DFF(g4, g3, clk);
  AND2(g5, g4, D);
> val it = x_net "g5" : hexp_t
```

Although correct, distributive laws mean that all four signals must hold constantly (ignoring start up).

Hence SERES on their own are not much use. Need to embed in temporal layer.

Also, let's create an output that holds when a safety predicate fails.

LG5.9 - ABD - PSL Temporal Layer Operators

- `P |-> Q` P is followed by Q (one state overlapping)
- `P |=> Q` P is followed by Q (immediately afterwards)
- `P && Q` P and Q occur at once (length matching)
- `P & Q` P and Q succeed at once
- `P within Q` P occurred at some point during Q
- `P until Q` P held at all times until Q started
- `P before Q` P held before Q held

LG5.10 - ABD - PSL Overall Layered Architecture

MODELLING LAYER : Create state properties using RTL.

BOOLEAN LAYER

```
not (rd and wr); -- rd, wr are design signals in the RTL.
```

TEMPORAL LAYER

```
-- Sequence definition
sequence s1 is {pkt_sop; (not pkt_xfer_en_n [*1 to 100]); pkt_eop};

sequence s2 is {pkt_sop; (not pkt_xfer_en_n [*1 to 100]); pkt_aborted};

-- Property definition
property p1 is reset\_cycle\_ended |=> {s1; s2};

-- Property p1 uses previously defined sequences s1 and s2.
```

VERIFICATION LAYER

```
assert p1;
```

LG5.11 - ABD - Sequence Constraint as a Suffix Implication

Earlier example: try phrasing using suffix implication:

Perhaps this will serve as a good `always` assertion?

`always` { **A;B** `|=>` **C;D** } now gives

```
  DFF(g0, A, clk);
  AND2(g1, g0, B);
  DFF(g2, g1, clk);
  INV(g3, C);
  AND2(g4, g3, g2);  // Holds if C missing
  DFF(g5, g2, clk);
  INV(g6, D);
  AND2(g7, g5, g6);  // Holds if D missing
  OR2(g8, g7, g4);
> val it = x_net "g8" : hexp_t   // Holds on error
```

Even this is not very specific: C and D might occur at other times.

PSL has a `onehot` operator: can use it to stop more than one of these values at once.

Data conservation: At an interface we commonly want to assert that data is not lost or duplicated. Is PSL any help?

LG5.12 - ABD - A Simple Model Checker

Formal checker for a state safety property:

Algorithm: '**Find reachable state space**': Add successors of current set until closure.

1. S := $\{ q_0 \}$ // initial state
2. S := S $\cup \{q' \mid \exists\ \sigma \in \Sigma, q \in S\ .\ NSF(q, \sigma) = q' \}$
3. If safety property does not hold in any $q \in S$ then flag error.
4. If S increased in step 2 then goto step 2.

S can be held explictly in bit map form or symbolically as a BDD.

Variation: property to check might be a path property:

- Compile it to a checking automaton (becomes a state property of expanded NSF).
- Expand it as we go (using modal μ calculus).

LG5.13 - ABD - Boolean Equivalence Checker

Often we have two implementations to check for equivalence:

- RTL version: pre-synthesis,
- Gate-level version: post-synthesis or post-layout.

Error sources: manual implementation, manual edits to synthesiser outputs and EDA tool fault.

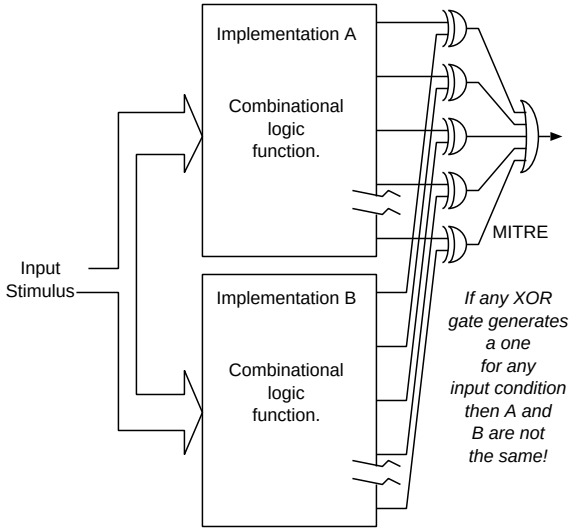
Boolean equivalence: do the two functions produce the same output?

- For all input combinations ?
- For a subset of input combinations (don't cares present).

Method: Create a *MITRE* of the two designs using a disjunction of XOR gates.

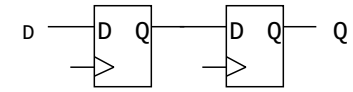
Then: feed negation of mitre to a SAT solver.

Result: if there are no solutions, designs are equivalent.



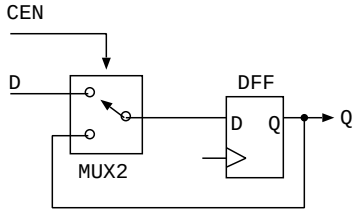
LG5.14 - ABD - Sequential Logic Equivalence: Example

Two-bit conventional shift register



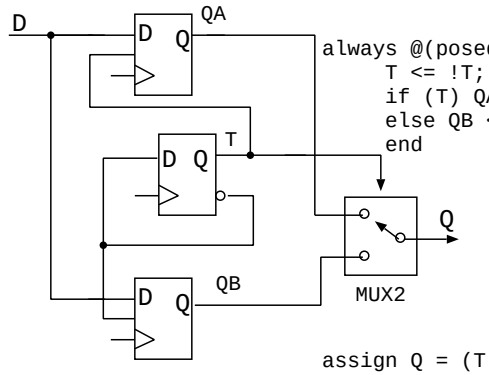

```
always @(posedge clk) begin
      q1 <= D;
      Q <= q1;
      end
```

Flip-flop with synchronous clock enable.




```
always @(posedge clk)
   if (cen) Q <= D;
```

Equivalent to a two-bit shift register: ping-pong style (two clock delay)




```
always @(posedge clk) begin
      T <= !T;
      if (T) QA <= D;
      else QB <= D;
      end
```

```
assign Q = (T) ? QA:QB;
```

Two implementations of a two-bit shift register.

Different amounts of internal state.

Equivalent observable behaviour (ignoring glitches).

LG5.15 - ABD - Sequential Equivalence Checker

Do a pair of designs follow the same state trajectory ?

- Considering the values of all state variables ?
- Considering a re-encoding of the state variables ?
- For an observable subset of the state (e.g. at an interface) ?
- When interfacing with a given reactive automaton ?

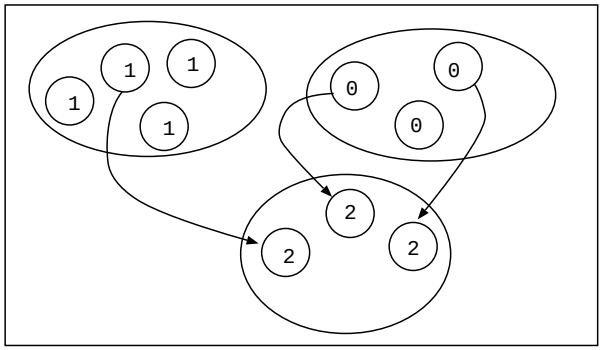
Other variations:

- Synchronous or asynchronous (turn-taking) composition.
- Strong or weak bi-simulation (stuttering equivalence).

Practical problem: Designs may only be equivalent in the used portion of the state space.

LG5.16 - ABD - Sequential Logic Simplification

A finite state machine may have more states than it needs to perform its observable function.



A Moore machine can be simplified by the following procedure:

```
1. Partition all of the state space into blocks of
    states where the observable outputs are the same
    for all members of a block.

2. Repeat until nothing changes (i.e. until it closes)
     For each input setting:
       2a. Chose two blocks, B1 and B2.
       2b. Split B1 into two blocks consisting of those
           states with and without a transition from B2.
       2c. Discard any empty blocks.

3.  The final blocks are the new states.
```

LG5.17 - ABD - Sequential Logic Equivalence

Alternative algorithm: start with one partition per state and repeatedly conglomerate.

Recent algorithms use a mixture of the two approaches.

Bi-simulation simplification: remove un-observable internal complexity.

Could de-pipeline a processor to give high-level model ?

Common core algorithms for : Bi-simulation checking and unobservable simplification.

`http://en.wikipedia.org/wiki/Formal_equivalence_checking`

CADP package: developed by the VASY team at INRIA.

Products: Conformal by Cadence, Formality by Synopsys, SLEC by Calypto.

LG5.18 - ABD - Automated Stimulus Generation

Testbench automation: generate pseudo-random input under constraining assertions.

```
struct frame {
  llc: LLCHeader;
  destAddr: uint (bits:48);
  srcAddr: uint (bits:48);
  size: int;
  payload: list of byte;
  keep payload.size() in [0..size];  };
```

The frame structure is accepted at an input port.

Testing will be inside envelope defined by '`keep`' statements.

An heirarchy of specifications and constraints:

```
extend frame { keep size == 0;  };
```

Products: Verisity's Specman Elite, Synopsys Vera.

`http://www.verisity.com/products/specman.html` `http://www.open-vera.com`