Focussing Assertion Based Verification Effort for Best Results

Mark Litterick (Verification Consultant)

mark.litterick@verilab.com

Introduction

- Project background
- Overview of ABV
  - including methodology and adoption strategy
- Targeting assertions for maximal return on effort
  - what can be asserted and where
- Example implementations for assertion hot-spots
  - including SVA code examples
- Review of project results
- Brief overview of Questa capability

Project Background

- ABV pilot project
  - aim was to develop and document ABV methodology
  - finding bugs was not the key driver
- Real device
  - close to tape-out
  - with further derivatives planned
  - established VHDL testbench with directed tests
  - FPGA prototype
- ABV proved more useful than expected:
  - detected extra bugs not spotted by simulation
  - pin-pointed bug source for defects visible in FPGA system

What is ABV?

- **Assertion Based Verification** is a *methodology* for improving the *effectiveness* of a *verification* environment
  - define properties that specify expected behavior of design
  - check property assertions by simulation or formal analysis
  - ABV does not provide an alternative testbench or stimulus
- Assertions are used to:
  - clarify specification requirements
  - capture design intent of implementation
  - validate correct operation and usage of design
- Benefits of ABV include:
  - improved error detection and reduced debug time due to observability
  - improved integration due to built-in self-checking
  - improved communication and documentation

Methodology

- ABV involves:
  - **analysis** of design to determine key targets for assertions
  - **implementation** of appropriate properties, assertions and coverage
  - **validation** by formal analysis (static), simulation (dynamic) or mixture
- ABV can be performed during the following project phases:
  - **specification** and planning (both design and verification)
  - **design** architecting and implementation
  - **verification** environment architecting, implementation and execution
- ABV can be applied to:
  - an existing design with known problems
  - an existing design with a planned family of derivatives
  - each new module introduced into a derivative
  - … or a completely new design

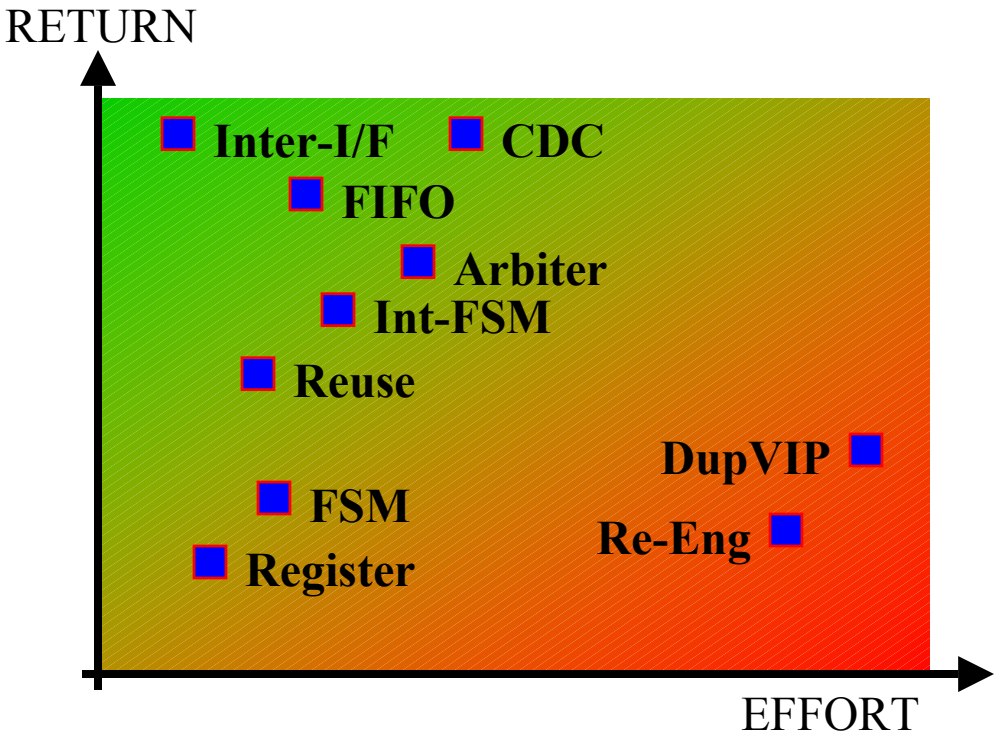
Adoption Strategy

- Apply Assertion-Based Verification to an existing design
  - ideally derivatives planned or late in design cycle
  - use stable testbench to validate assertions by simulation
  - focus on learning assertion coding and capability
- Apply Assertion-Based Design to a new module
  - focus on coding assertions during or before RTL implementation
  - improve testbench environment to maximize synergy with assertions
- Apply formal ABV to a new module development
  - focus on learning strengths and limitations of formal techniques
  - also validate module and assumptions in simulation environment
- Complete Assertion-Based Design and Verification project
  - focus on formal for modules, simulation for chip

What Properties to Assert

- Low effort
  - protocol operation (relationships, order, response, one-hot, etc)
  - unknown values for control (at any time) and data (during transfers)
  - over/underflow (FIFOs, stacks, buffers, shift registers, etc)
  - legal state acquisition and transitions (FSMs)
  - data stability (during transfers, across clock domains, etc)
- Medium effort
  - data integrity (FIFOs, shared memories, across clock domains, etc)
- High effort
  - identifying key assertions for algorithm steps
  - full functionality of complex bus protocol (e.g. PCI, AHB)
  - achieving appropriate assertion density for formal validation

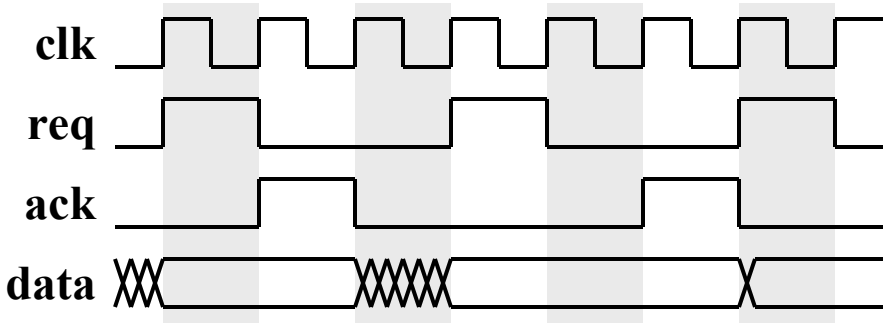
Where to Implement Assertions



- High return on effort
  - inter-module interfaces
  - clock domain crossings
  - temporary storage elements
  - resource sharing logic
  - interactions between FSMs
  - reused components
- Medium return on effort
  - basic FSM functionality
  - simple fundamental components
- Low return on effort
  - duplicated checkers (e.g. HVL Verification IP)
  - reverse engineered design intent

Inter-Module Interfaces

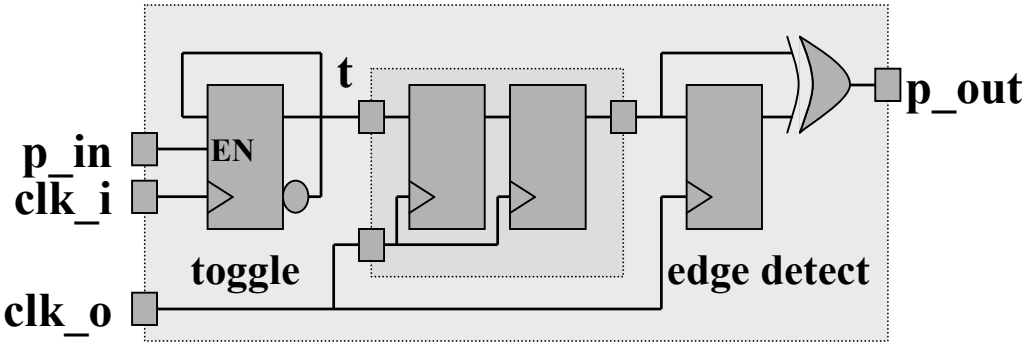
- Example: request-acknowledge handshake interface
  - req and ack never unknown
  - req gets ack before other req
  - no ack without preceding req
  - data valid during transfer
  - data stability during transfer




```
sequence s_transfer;
  @(posedge clk)
    req ##1 !req [*1:max] ##0 ack;
endsequence

property p_req_gets_ack;
  @(posedge clk)
    req |-> s_transfer;
endproperty

property p_ack_had_req;
  @(posedge clk)
    ack |-> s_transfer.ended;
endproperty

assert property(p_req_gets_ack);
assert property(p_ack_had_req);
```

Clock Domain Crossings

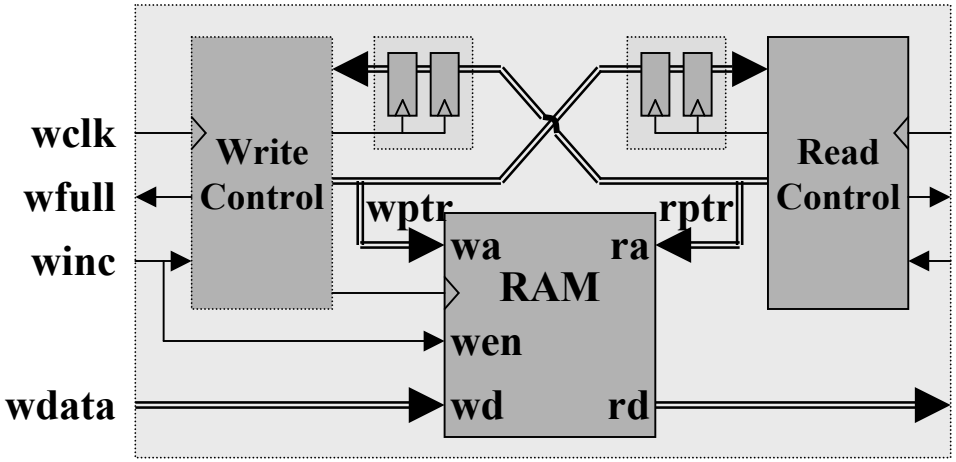
- Design and verification problem
  - structural analysis to ensure synchronizers are connected
  - verify correct operation and use of synchronizers
  - verify correct operation of design in presence of CDC jitter
- Example: pulse synchronizer
  - p_in high for 1 src clock
  - p_in seperation 3 dest clock edges
  - p_in causes p_out pulse




```
property p_in_out;
  @(posedge clk_i)
    p_in |=>
  @(posedge clk_o)
    ##[1:3] p_out;
endproperty

assert property(p_in_out);
```

Temporary Storage

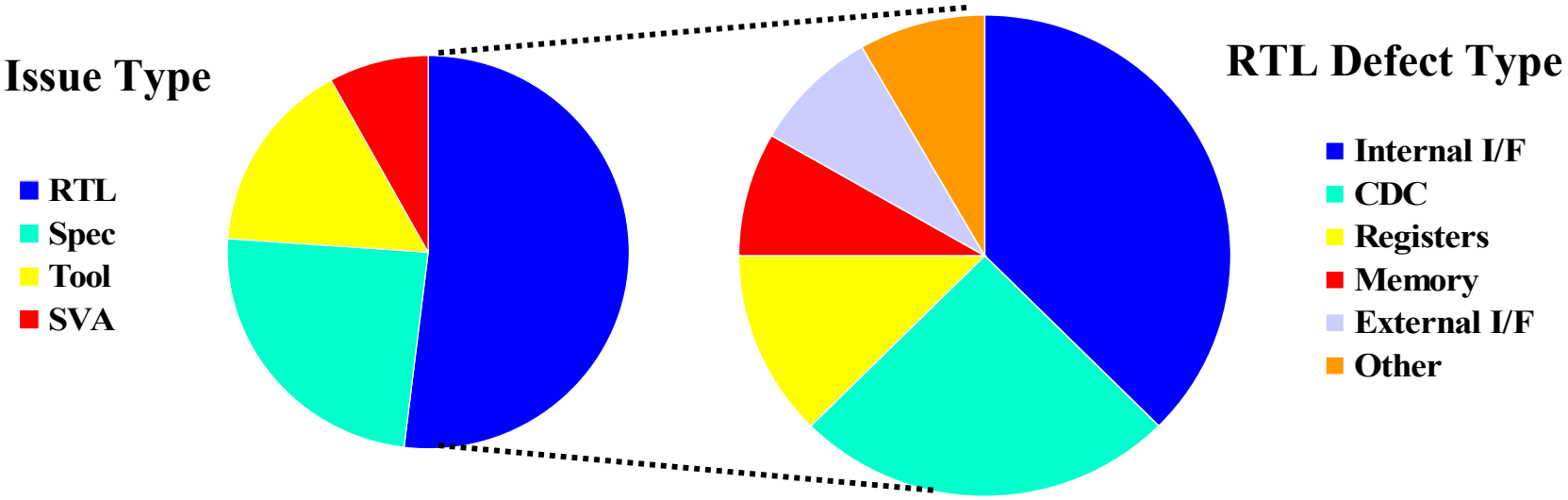
- Example: dual-clock asynchronous FIFO
  - never write when full
  - never read when empty
  - control and status never unknown
  - data not unknown during access
  - pointers must be gray-coded
  - data integrity




```
always @(posedge wclk)
 if (winc) wcnt = wcnt + 1;

always @(posedge rclk)
 if (rinc) rcnt = rcnt + 1;

property p_data_integrity
 int cnt;
 logic [DSIZE-1:0] data;
 @(posedge wclk)
  (winc, cnt=wcnt, data=wdata) |=>
 @(posedge rclk)
  first_match(
    ##[0:$] (rinc & (rcnt==cnt)))
  ##0 (rdata==data);
endproperty
```

Example Project Results

- Hardware graphics accelerator for mobile applications
  - mature VHDL design with modifications and derivatives planned
  - established VHDL testbench environment including FPGA prototype
  - ABV introduced late in design cycle to subsystem level



Specification Payback

- The first thing to benefit from ABV is the specification
- Attempting to write assertions identifies:
  - errors, inconsistencies, omissions and vagueness
  - to a much greater extent than RTL implementation
- Specification can be fixed and enhanced with:
  - clearer description of intent
  - additional text and diagrams
  - assertable English statements
  - executable specification
  - high return for low effort

Concise textual description of intent, including corner cases and exceptions.
• A must cause B within N cycles
• B and C are mutually exclusive

Figure 1: Normal Cycles

Figure 2: Abort Behavior

Appendix A: SVA for Bus Interface

Questa Capability

- Project used *evolving* ModelSim/Questa toolset (5.x … 6.1b)
- Problems with previous releases included:

simulation performance	unsupported SVA constructs
occasional crashes	limitations on binding
inaccurate profiling	formal argument restrictions
incorrect sampling	limited debug support

- Latest release is robust, fully featured and still improving:
  - performance with VHDL still not great, but much better
  - debug support for local variables in multi-threaded required
  - merging of coverage databases underway
- Questa is totally useable and supports SVA today!

Conclusion

- ABV gets results with minimal effort
  - select appropriate entry point
  - validation using simulation is effective
  - provides a path to formal
- SVA language is a pleasure
  - concise, logical and clear
  - relatively easy to learn for H/W designers
- Tools (from MTI and others) are ready
  - Questa is among the very best
- ABV fulfills its promise and improves:
  - effectiveness of verification environment
  - quality of design

mark.litterick@verilab.com