CONNECTING THE DOTS: APPLICATION OF FORMAL VERIFICATION FOR SOC CONNECTIVITY

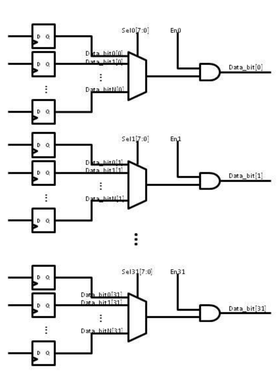
Bin Ju, Staff Application Engineer, Cadence Design Systems, Inc.

Introduction

- Real design case study with unnamed customer in 2013
  - Customer personally supported by the author
- Customer experiencing greatly increased complexity in each SoC generation
- SoC top-level design integrates increasing number (instances and types) of IP blocks
  - Third-party and in-house IP
  - Some connected directly to protocol-based on-chip bus
  - Others connected via glue logic
  - Always glue logic between IP interfaces and IO pad ring

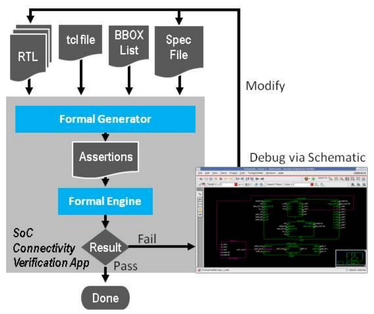
Motivation and Previous Methods

- Top-level connectivity relies on late-arriving design information
  - Info only complete 2 months before tape-out
  - ECO flow for bugs found after RTL freeze subject to management scrutiny
- Top-level connectivity is complex
  - Up to 256-wide multiplexing on each IO
  - Test signal routing differs in each design
- Simulation methods are too time-consuming and inefficient
  - Previously used directed tests with Verilog force/observe methodology
  - Building test environment took 2-3 months for typically 15 different test types
  - About 1 week per test type to execute
- Problem lends itself well to formal verification
  - But customer found manual creation of assertions hard

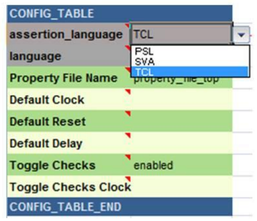


Formal Verification Flow for Connectivity

1. Capture connectivity specification (spreadsheet)
2. Export as .csv, import with RTL to Incisive® Formal Verifier (IFV)
3. Generate connectivity assertions
4. Generate black box module list
5. Execute assertions in IFV
6. Debug connectivity failures

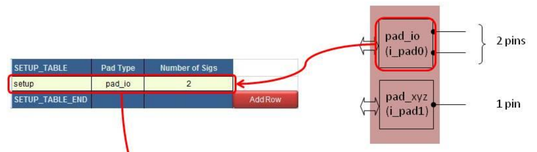


1a: Configuring Property Generation



- Assertion Language
  - SVA/PSL/TCL
- Language (at top-level of design)
  - Verilog/VHDL
- Property File Name
  - Defines the Property File Name
- Default Clock
  - Sampling clock for properties
- Default Reset
  - Reset Condition (used for disable iff/abort)
- Default Delay
  - Delay value for pipelined connections
  - Needed for pipelined connections only
- Toggle Checks
  - Enable/Disable Toggle Cover generation
- Toggle Checks Clock
  - Sampling clock for Toggle covers

1b: Setup Table – Specifying Pads



- Used to define pad types and associated number of ports that will need to be checked per type
- The goal is to ensure that for each type, the correct # of rows exist in the connectivity section
- If number of rows are less than expected, an error will be issued

C1	C2	C3	C4	C5	C6	C7	C8	C9
Pad Path	Pad Name	Pad Type	Clock Expr	Reset Expr	Dest	Pri 0 Expr	Pri 0 Expr-Dest Delay	Pri 0 Src
A_PADS	i_pad0	pad_io	posedge top.hclk		dout	A_MCTRL.sel0[0]	3	A_UART0.stx_pad_
A_PADS	i_pad0	pad_io	posedge top.hclk		din	A_GPIO.gpio_pin_in[0]	2	

1c: Connectivity Definition Table

- Single Table for PAD-IP and IP-IP Connections
  - Table for Muxed connections
  - Table for Non-Muxed connections
- Connectivity Data Entry
  - Manual
  - Through built-in forms

PAD-IP/IP-IP_CONNECTIVITY_DEFINITION_TABLE

MUXED_TABLE

C1	C2	C3	C4	C5	C6	C7	C8
Pad Path	Pad Name	Pad Type	Clock Expr	Reset Expr	Dest	Pri 0 Expr	Pri 0 Expr-Dest Delay
					A_CORE.pclk	A_MCTRL.clk_sel	
					A_PADMUX.foo	A_MCTRL.sel0[1]	1
A_PADS	i_pad0	pad_io	posedge top.hclk		dout	A_MCTRL.sel0[0]	3

MUXED_TABLE_END

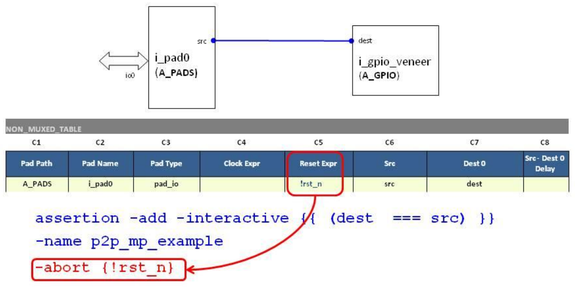
NON_MUXED_TABLE

C1	C2	C3	C4	C5	C6	C7	C8
Pad Path	Pad Name	Pad Type	Clock Expr	Reset Expr	Src	Dest 0	Src - Dest 0 Delay
					A_BRIDGE.pwdat	!A_MCTRL.pwdata	
					A_BRIDGE.pwdat	!A_SPI.wb_dat_i	
A_PADS	i_pad0	pad_io	posedge top.hclk		din	A_GPIO.gpio_pin_in[0]	2
					A_CORE.pclk	A_UART0.wb_clk_i	

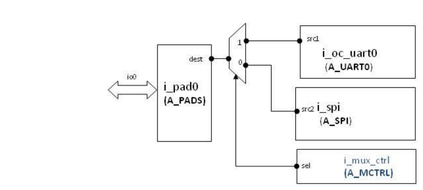
NON_MUXED_TABLE_END

PAD-IP/IP-IP_CONNECTIVITY_DEFINITION_TABLE_END

3a: Basic Connectivity Assertion With Reset



3b: Muxed Connectivity Assertions

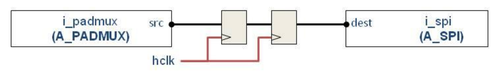


MUXED_TABLE

C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14
Clock Expr	Reset Expr	Dest	Pri 0 Expr	Pri 0 Expr-Dest Delay	Pri 0 Src	Pri 0 Src-Dest delay	Pri 1 Expr	Pri 1 Expr-Dest delay	Pri1 Src	Pri1 Src-Dest Delay
		dest	sel		src1		!sel		src2	

```
assert -add -interactive {{(sel)} |-> {dest === src1)}
assert -add -interactive {{!(sel)} |-> {dest === src2)}
```


3c: Pipelined Connectivity Assertions



NON_MUXED_TABLE

C1	C2	C3	C4	C5	C6	C7	C8
Pad Path	Pad Name	Pad Type	Clock Expr	Reset Expr	Src	Dest 0	Src - Dest 0 Delay
			posedge hclk	!rst_n	src	dest	2

```
assertion -add -interactive {{[*2]; dest === prev(src,2)}}
-name pipeline_fixed_lat_example
-abort {!rst_n}
-clock hclk -edge posedge
```


Black-boxing and Execution

- Black-boxing modules reduces complexity and run-time
  - Removes functional logic that does not affect connectivity from formal analysis

Manually create list of **modules** to be black boxed and pass to IFV

```
blockA
blockB
blockC
.....
.....
```

- Manual black-boxing

```
% irun -ifv -connectivity <spec file> -f ifv_top.f -bb_list <bb_file>
-input top.tcl
```

  - Manual black-boxing was used by this customer

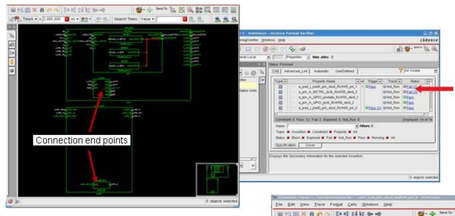
Automatically generates black box list `auto_gen_bblist.txt` based upon specification file

- Automatic black-boxing

```
% irun -ifv -connectivity <spec file> -f ifv_top.f -bb_gen -input
top.tcl
```

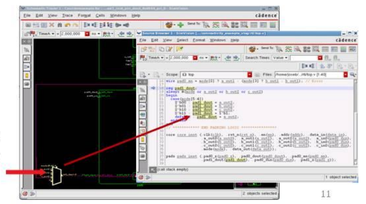
  - Automatic black-boxing is new and was not validated in this case study

Debug



Cross-probing between assertion results in SimVision and connection path in schematic browser

Cross-probing between schematic browser and source browser



Results and Conclusions

- Formal SoC Connectivity flow was easy to setup and use
  - Highly reusable from design to design
  - Verification set-up time reduced to typically 1 day
- Exhaustive verification was achieved
  - With between 0.5-1 full-time verification engineer per project
  - 15 test types completed in under 1 month
- Real bugs found that directed tests would not have caught
  - E.g. block supplied by wrong clock, which was identical to the right clock in all but one corner-case
  - No bugs have escaped, no highly-visible ECO needed, since formal flow adopted

3X verification productivity improvement
2 months reduced design time
No connectivity bug escapes

cadence®