CONNECTING THE DOTS: APPLICATION OF FORMAL VERIFICATION FOR SOC CONNECTIVITY

Bin Ju, Staff Application Engineer,
Cadence Design Systems, Inc.
binju@cadence.com

Abstract— **There are several aspects of SoC connectivity which are difficult to exhaustively test through functional simulations. In the past, we would test some arbitrarily chosen subset of such connections using the Verilog force/observe methodology. Formal verification provides a much more scalable solution, finding many more connectivity issues earlier in the process, with fewer verification resources. Formal is also inherently exhaustive, thus addressing the quality of testing.**

This paper will describe in detail the process to apply formal analysis to SoC connectivity verification. As well as showing the methodology and technology that achieves the main benefit of scalability, we will highlight some of the specific bugs found. Finally, we will discuss limitations to the current connectivity verification process, including the current spreadsheet-based connectivity specification method, and highlight improvements to be made.

The paper provides an accurate account of real work carried out by a specific customer supported by the Author, during 2013. Unfortunately, the customer cannot be named.

Keywords— assertions; formal verification; SoC connectivity

I. INTRODUCTION

Each generation of SoC design is more complex than the one before. For a number of design generations, our customer has needed to integrate IP blocks from both third-party suppliers and in-house design at the chip-level. Sometimes these IP blocks connect directly to other blocks or to the interconnect fabric via standardized interface protocols. In other cases the IP block connections need glue logic. There is always some glue logic between block interfaces and the I/O pad ring. Blocks from different design groups often have different assumptions on the interfaces.

All of these connections need thorough verification. Protocol-based interfaces lend themselves more naturally to functional test. Other interfaces are more structural, with no specific timing, but signal connection paths are not necessarily straightforward, with many multiplexers and other components at the top level. Verifying such connections using simulation-based directed testing is time-consuming and inefficient. For each such connection to be tested, verification engineers had to write testbench code incrementally to add the force and observe points. As a result such testing was done very late in the project and often such bugs had to be fixed via ECOs. In addition, the exhaustiveness of this approach was always in question and subject to human error and interpretation. Our customer has found that this kind of testing is better suited to formal connectivity checking, using a spreadsheet-based chip-level connectivity specification.

II. MOTIVATION

Top-level connectivity relies on certain late-arriving information, including programming for both functional operation and test modes. Often, in our customer's design schedules, this information is only complete two months or so before scheduled tapeout, so there is enormous time pressure to complete verification and achieve RTL freeze. After RTL freeze, any bugs found are subject to a closely-controlled ECO flow and are highly visible to management. Hence it is critical that top-level connectivity verification is not only completed quickly, but is also rigorous.

Top-level connectivity is much more complex than engineers who may not have had the pleasure of verifying it might assume. In our customer's typical designs, there are many miscellaneous memory port connections for DFT purposes, and test bus multiplexing. Depending on programming, test signals can be routed differently, connecting to top level I/O pads or memory.

Muxes are also widely used between blocks from different teams, for example the configuration buses between baseband and radio.

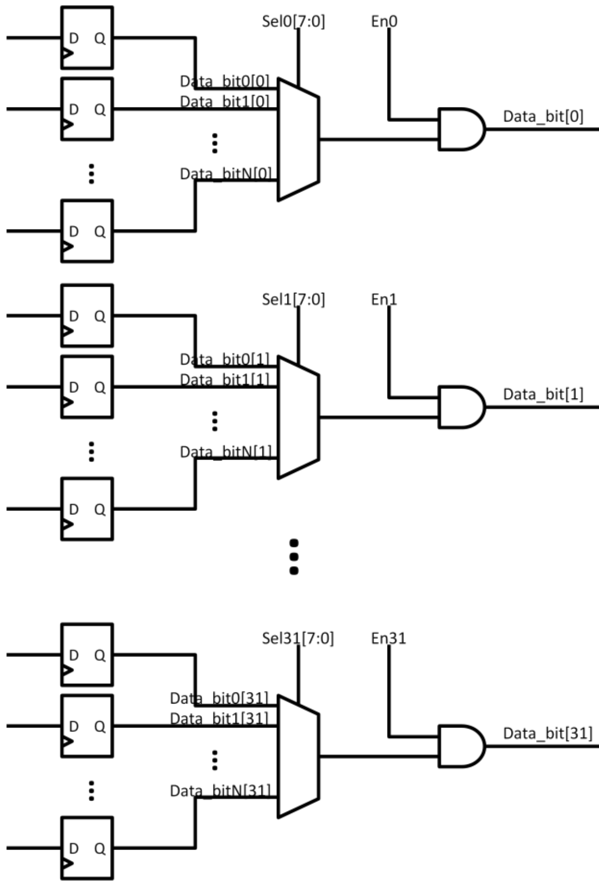


Figure1: Typical Signal Multiplexing for 32-bit SoC Port

For example, Figure 1 shows a 32-bit output interface. Multiplexers are configured in a 32-wide gang and can select one of up to 256 bits each, for each bit of the 32-bit interface.

Additionally, test signals and sequences change between one design and the next. As much as 50% change from one project to the next project is common. So our customer needed a verification solution that made it easy to reuse across projects, and also to easily manage the changing connectivity specifications.

III. Previous Connectivity Verification Methods

Before adopting formal methods, the previous verification approach was to create directed tests that would test some arbitrarily chosen subset of such connections using the Verilog force/observe methodology. For each such connection to be tested, verification engineers had to write testbench code incrementally to add the force and observe points, forcing a value at one end of the signal path, and observing at other.

This customer's previous projects had between 10 and 15 different directed test types. Building the test environment was tedious, taking between 2-3 months – almost one week per test type. Additionally, the force / observe tests were not exhaustive, especially for the degree of signal path complexity illustrated in Figure 1. Furthermore, the existing methods were not reusable from project to project, considering the test bus signals and sequence changes we already described.

As a result such testing was done very late in the project and bugs found had to be fixed via the highly-visible ECO process. More often than not, projects would have a bug needing ECO. In addition, the exhaustiveness of this approach was always in question and subject to human error and interpretation.

One possible improvement our customer considered was to manually create assertions at the interfaces. An obvious downside to this approach is the need for verification engineers to code assertions. Also it would not fully address the need for reuse across changing connectivity specifications. However, while the verification engineers could do this, another problem is that the designers must also sign-off on connectivity specification and verification. In our customer's experience, the designers are not as familiar with using assertions as the verification engineers.

A connectivity specification approach based on a spreadsheet would be easier, and more understandable by other teams. As a baseline improvement, the spreadsheet would be an effective

master documentation of the changing connectivity specification, enabling clearer communication between verification and design engineers, which in itself would eradicate many connectivity issues. Furthermore, if verification could be automated from the same specification, the benefits would be enormous.

IV. FORMAL VERIFICATION FLOW FOR CONNECTIVITY

Such a spreadsheet-based connectivity specification method, with automated assertion creation to check the specified connectivity, is nowadays available with commercially available formal verification tools. The case study we describe here used Incisive® Formal Verifier (IFV) from Cadence® Design Systems, Inc. The connectivity verification flow is illustrated in Figure 2.

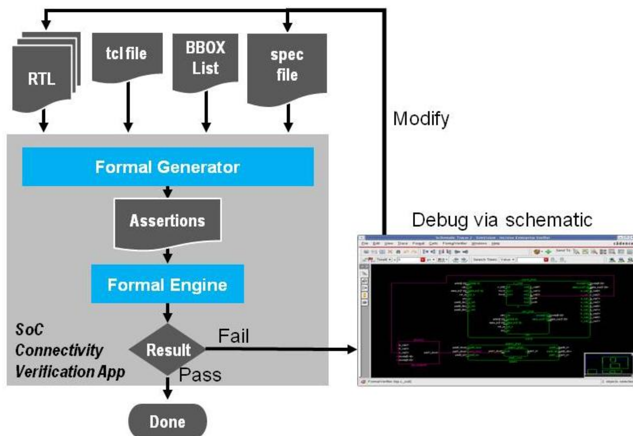


Figure 2: Formal Connectivity Verification Flow

Steps to check connectivity were:

1. The connectivity information was captured in a spreadsheet. See Appendix 1 for an illustration of a typical connectivity spreadsheet. Template and example spreadsheets are provided with the tool. The key fields the designer must complete in the spreadsheet are as follows:

   a. Complete a configuration table: the table is illustrated in Figure 3. The designer sets the assertion language (SVA, PSL or TCL); the design language (Verilog or VHDL); file names; and default parameters for clock, reset, delay and toggle covers. Note that for the assertion language selection, we use TCL as the default. This speeds up connectivity specification update iterations, as changes to the connectivity specification do not require recompilation. PSA and SVA have the advantage that the generated assertions can also be used in simulation if desired.

CONFIG_TABLE	
assertion_language	TCL (PSL / SVA / TCL)
language	
Property File Name	property_file_top
Default Clock	
Default Reset	
Default Delay	
Toggle Checks	enabled
Toggle Checks Clock	
CONFIG_TABLE_END	

Figure 3: Configuration Table

   b. Complete the pad setup table: The I/O pads are specified in a simple table, as illustrated in Figure 4. This information is used to verify there are correct number of rows in the spreadsheet depending on the number of ports associated to the pad that need to be checked. If the number of rows is less than expected, an error issued.

SETUP_TABLE	Pad Type	Number of Sigs	
setup	pad_io	2	
SETUP_TABLE_END			Add Row

Figure 4: Pad Setup Table

   c. Complete the connectivity rows: Next, the source, destination and all intermediate points for Pad-IP and IP-IP connections are specified. The capability to define aliases is useful to minimize repetitive typing of signal names. Connections can be point-to-point, multipoint, multiplexed, or pipelined, with fixed or variable latencies. Figure 5 illustrates a typical connectivity definition table for both muxed and non-muxed types.

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	
			posedge top.hclk		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							

Figure 5: Connectivity Definition Table

2. The Microsoft Excel file created in step 1 was exported as a .csv file.
3. The connectivity information present in the .csv file was converted to a set of assertions by IFV automatically. The types of checks and the code generated are determined by the connectivity specification described in step 1 and there is nothing further the designer needs to do to specify or create the checks.

Some examples are given below, to illustrate the relationship between the specification and the generated code, in this case in the tcl language:

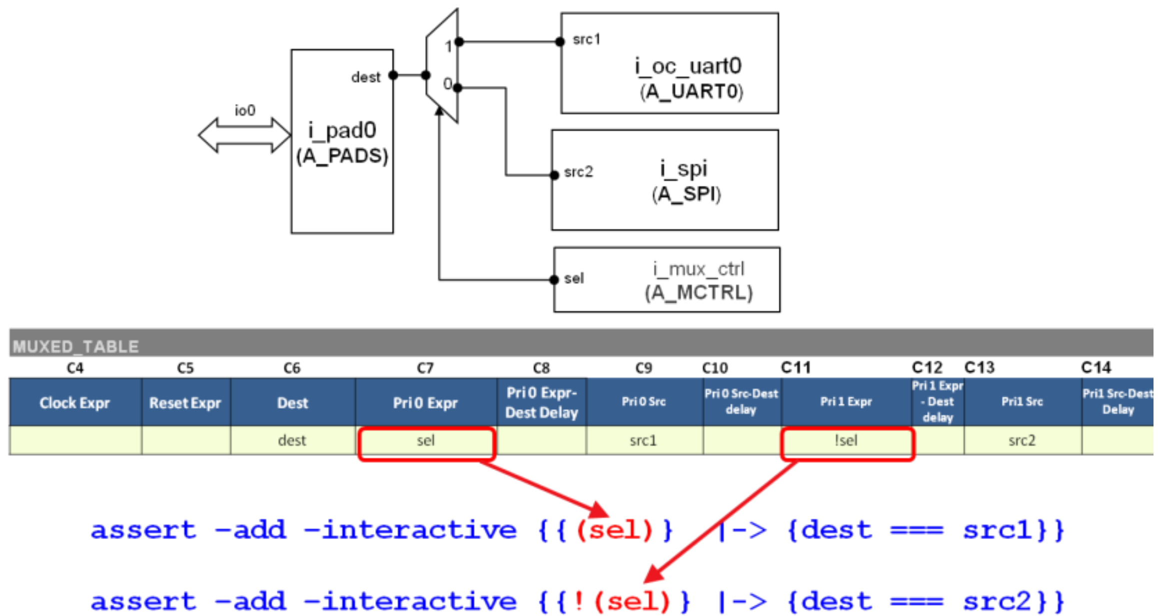
a. Basic connectivity with reset

Figure 6 shows a simple connection with reset definition, which gives the abort construct.

NON_MUXED_TABLE						
C1	C2	C3	C4	C5	C6	C7
Pad Path	Pad Name	Pad Type	Clock Expr	Reset Expr	Src	Dest 0
A_PADS	i_pad0	pad_io		!rst_n	src	dest

```
assertion -add -interactive {{ (dest  === src) }}
-name p2p_mp_example
-abort {!rst_n}
```

Figure 6: Connectivity Assertion with reset



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	

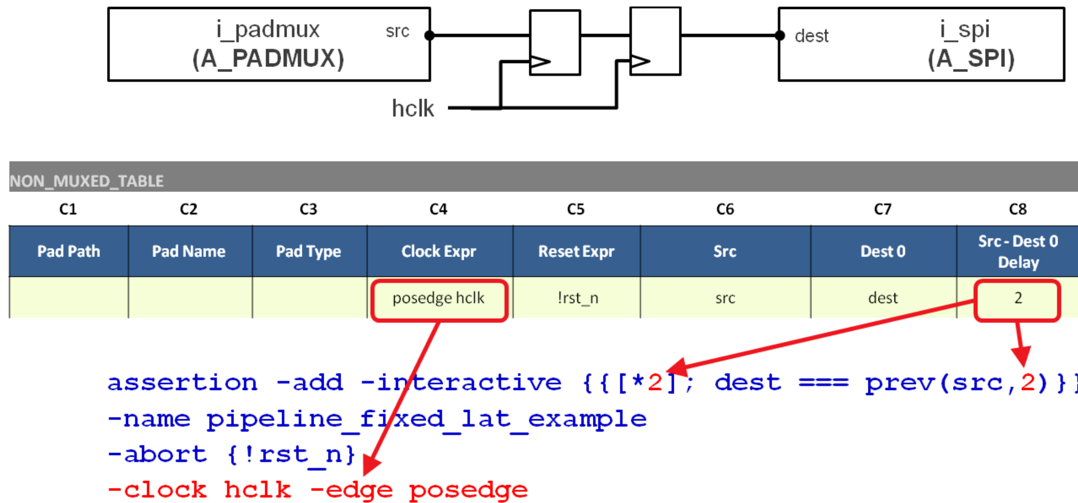
Figure 7: Muxed Connectivity Assertions

b. Muxed connectivity

See Figure 7 above. Note how the select expression is defined.

c. Pipelined connectivity

See Figure 8 below. Note the clock expression and cycle latency definition.



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

Figure 8: Pipelined Connectivity Assertions

4. A list of modules to black box was manually created. Since we are only interested in verifying top level connectivity black-boxing removed irrelevant design logic. The IFV tool now has the ability to freely drive the connectivity endpoints. Benefits are exhaustive connectivity verification and high performance by removing logic irrelevant to verifying the top level connectivity.

5. The IFV tool was run to execute the assertions to verify connections. The tool can be invoked to run through the above-mentioned steps individually, or steps 3, 4 and 5 can be combined in a single invocation command as follows:

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

6. Any connectivity failures encountered were debugged using a schematic tracer. The schematic tracer provides cross-probing with the results log and source code views, and is illustrated in Figure 9.

V. RESULTS

The formal connectivity verification flow proved very easy to setup and use, and highly reusable. From project to project, our customer proved that

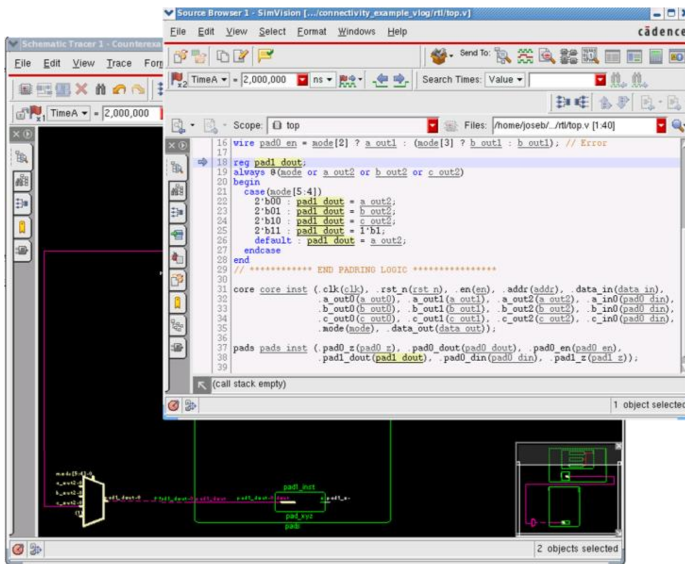


Figure 9: Schematic Debug

they can just update the changes in the spreadsheet, and automatically create the new project's connectivity verification environment. Exhaustive verification could be achieved with only 0.5-1 full-time equivalent verification engineers per project. There was minimal set-up, consisting only of the creation of the .csv file. Normally no constraints required, clock definition is only required for pipelined connections and is configured in the spreadsheet as described earlier. No other constraints or mode setup is required, except as defined in the spreadsheet. We are only concerned with verifying connectivity. No counter-examples tend to be generated for connectivity tests, as one might think would be a result of this loosely-constrained environment. The black-box list is one area that does need careful attention. Black-boxing removes the functional logic so the formal tool is not presented with unnecessary complexity and has access to signals at their endpoints. Initially, it took about 1 day to prepare the first test case, and then our customer was able to progress at a rate of a few test cases per day. For the 15 test cases, they were all done in less than 1 month.

The application of formal connectivity verification method in this case study also showed that verification was more exhaustive. An example of a real connectivity bug that our customer was able to find involved two blocks connected together. Block A's clock was supposed to be connected to clock A, but instead it was connected to clock B. In normal operation, this did not matter since the clocks were identical. However, there was a special case when one clock was off that was not covered. Formal verification caught this, and the directed tests did not.

It is still early days for application of this formal verification method at this customer, but in the projects completed so far no connectivity bugs have escaped connectivity tests, needing to be fixed with the ECO process, since the adoption of the formal flow.

In terms of limitations to the current formal connectivity verification process, our customer made some suggestions for improvements that could be made to improve the spreadsheet specification stage. These changes would enable easier documentation of the connectivity specification earlier in the flow. There was also a suggestion to improve the debug capabilities to save time to close on reasons for failures. Many of the failures experienced were due to errors in the manual black box list. These seemed to occur on larger designs. The tool could be improved to add better debug information to help trace the problems caused by an incorrect black-box list to the right design module.

Once the black box list was correct, we found that connectivity verification was not as demanding on compute resources as many other applications of formal analysis. It was also usual for the assertions created for connectivity verification to attain a definitive pass or fail result in a reasonable time – usually seconds or minutes – even the with the complex wide multiplexed connectivity illustrated in Figure 1. With the number of assertions generated typically in the low thousands for designs of the complexity verified by our customer, and small state-depths, total execution times were much better than simulation. Pipelined connectivity with high

latency can result in higher state depths that increase execution time, but this was not an issue in this case.

VI. CONCLUSIONS

The formal verification flow has reduced the time for connectivity verification down to one month, compared with 3 months previously, for this customer's typical SoCs. This represents a 3X productivity improvement and two months reduced design time at a critical phase of the design flow. Additionally and probably more importantly, the flow has proven to be exhaustive, with no connectivity bugs escaping beyond RTL freeze and invoking the ECO process.

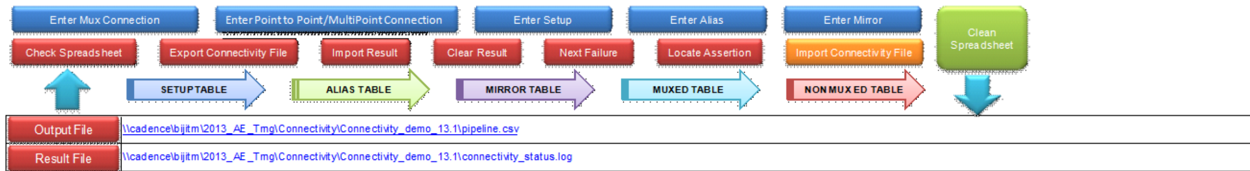
REFERENCES

[1] Verification Apps User Guide, Product Version 13.1, Cadence Design Systems Inc., 2013

© 2014 Cadence Design Systems, Inc. All rights reserved worldwide. Cadence, and Incisive are registered trademarks of Cadence Design Systems, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

Appendix 1: Connectivity Specification in Spreadsheet Form

Incisive Formal
SoC Connectivity Solution (V3.02)



CONFIG_TABLE	
assertion_language	SVA
language	Verilog
Property File Name	pipeline
Default Clock	
Default Reset	
Default Delay	
Property File Scope	top
max_width	32
Bind Instance Name	i_demo
CONFIG_TABLE_END	

SETUP_TABLE	Pad Type	Number of Sigs	
setup	pad_io	2	
SETUP_TABLE_END			Add Row

ALIAS_TABLE	Local Name	Signal Path	
alias	A_CORE	top.i_apb_subsystem	
alias	A_PADS	top.i_padring	
alias	A_BRIDGE	A_CORE.i_ahb2apb	
alias	A_GPIO	A_CORE.i_gpio_veneer	
alias	A_SPI	A_CORE.i_spi	
alias	A_UART0	A_CORE.i_oc_uart0	
alias	A_MCTRL	A_CORE.i_mux_ctrl	
alias	A_PADMUX	top.i_padmux	
ALIAS_TABLE_END			Add Row

MIRROR_TABLE	Local Name	Signal Path	Width	
MIRROR_TABLE_END				Add Row

TCL_COMMANDS_TABLE	
TCL_COMMANDS_TABLE_END	Add Row

PAD-IP/IP-IP_CONNECTIVITY_DEFINITION_TABLE

MUXED_TABLE

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
Pad Path	Pad Name	Pad Type	Clock Expr	Reset Expr	Dest	Pri 0 Expr	Pri 0 Expr-Dest Delay	Pri 0 Src	Pri 0 Src-Dest delay	Pri 1 Expr
A_PADS	i_pad0	pad_io	posedge top.hclk		dout	A_MCTRL.sel0[0]	3	A_UART0.stx_pad_o	3	!(A_MCTRL.sel0[0])
			posedge top.hclk		A_PADMUX.foo	A_MCTRL.sel0[1]	1	A_SPI.mosi_pad_o	1	!(A_MCTRL.sel0[1])
					A_CORE.pclk	A_MCTRL.clk_sel		A_CORE.hclk		!(A_MCTRL.clk_sel)

MUXED_TABLE_END

NON_MUXED_TABLE

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
Pad Path	Pad Name	Pad Type	Clock Expr	Reset Expr	Src	Dest 0	Src - Dest 0 Delay	Dest 1	Dest 1 delay	Dest 2
					A_CORE.pclk	A_UART0.wb_clk_i		A_SPI.wb_clk_i		A_GPIO.pclk
A_PADS	i_pad0	pad_io	posedge top.hclk		din	A_GPIO.gpio_pin_in[0]	2			
					A_BRIDGE.pwdata	A_SPI.wb_dat_i		A_GPIO.pwdata		A_UART0.wb_dat_i
					A_BRIDGE.pwdata[15:0]	A_MCTRL.pwdata				

NON_MUXED_TABLE_END

PAD-IP/IP-IP_CONNECTIVITY_DEFINITION_TABLE_END