# An Integrated FPGA, PCB, Verification, Software, and Testing Design Flow


## ABSTRACT

This paper describes an integrated design environment that connects FPGA designs with the hardware PCB design flow, allows for software co-simulation, enables simulation of multi-board systems, and helps with the lab testing and integration of the overall system. By using this flow, a 32-layer PCB with 11 large FPGAs that took nine months from concept to gerbers, was integrated into the rest of the system less than one week after it arrived from the board manufacturer. The one barnacle was due to an error in a datasheet that was found after the board was built.



## Categories and Subject Descriptors

M2.1 [**Design Methodology and Case Study**]: Overall design flow

## General Terms

Design flow, methodology

## Keywords

FPGA design flow, PCB design flow, system integration


## 1. INTRODUCTION

The design of a large network switching system requires the implementation of many complex interacting printed circuit boards (PCBs). Each PCB will have commercial and proprietary devices that are typically ASICs or FPGAs. There is also typically a control processor that communicates with each of the devices on the card. Building such a system involves a number of different design flows that can be roughly divided into the PCB, ASIC/FPGA, and the software design flows. Each of these flows are complex in their own right and they will have their own integrity, verification, and design rule checks.

Many of the integration problems will occur at the boundaries of these flows. An obvious example is that often the first time the software compatibility with the hardware can be tested is when the real hardware arrives. Problems will also occur at the ASIC/FPGA and PCB boundary. A simple example occurs when the pin assignments are not consistent between the chip design and the PCB design.

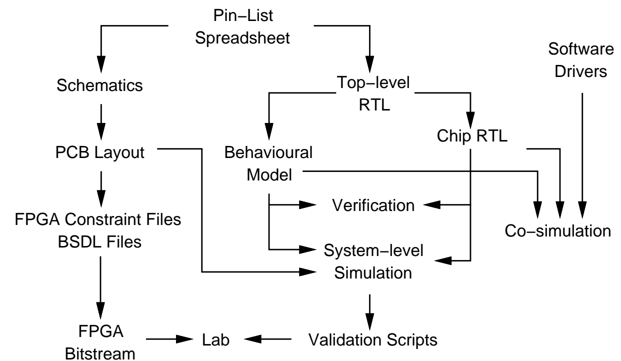


**Figure 1: The overall design flow.**

Another problem area is checking the connectivity of the components on a PCB and the overall functionality of the system. In our case, we had boards with over ten large FPGAs plus other off-the-shelf components. With the large number of interacting parts, it was desirable to be able to test the system-level functional interaction of the FPGAs being developed along with the third party components.

In this paper, we describe the design flow that was developed at XYZ Networks, an optical networking startup company, that integrated our FPGA, PCB, and software driver designs. Figure 1 shows an overview of the flow.

There were three goals: To achieve first-time success for each PCB design; To provide a co-simulation flow for the software team to debug their drivers for the FPGAs; To provide a means to generate scripts for the initial board testing and system integration.

Some of the important glue that ties the overall flow together is the naming convention described in Section 2. The root of the design flow centres around the FPGA flow, which is discussed in Section 3. The links to the PCB design flow are described in Section 4. Section 5 outlines how schematic checking was done and the simulation environment is described in Section 6. Connecting the design environment to the lab testing is explained in Section 7 and results and conclusions are given the final two sections.

## 2. NAMING CONVENTION

The use of common names for all signals in all flows was how linkages between the various flows were implemented.


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.


*DAC* '04 San Diego, California USA


Copyright 2004 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

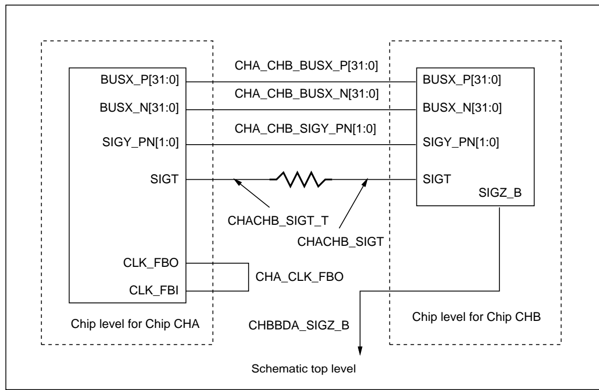


**Figure 2: Two-level signal naming convention.**

For example, source and destination names were used in connectivity checking. Differential signal naming was used for pin placement and DRC rule checking. The names also gave some indication about the properties of the signal. The main requirements and features of the convention are highlighted here.

Two levels of naming were used. One level was a chip-level name and the other was the schematic top-level name. The chip-level names applied to the chip-level schematic and the FPGA ports. The top-level names were used for the chip-to-chip connections and incorporated the chip-level name as well as being prepended by additional information. The two-level hierarchy was sufficient to allow for multiple instances of the same device.

The key requirements of the naming convention included an indication of the source and destination, dealing with active low signals, names for differential pairs, names for active low differential pairs, handling signals that go to connectors and backplanes, handling loop-back signal names such as a clock feedback, allowing for different instances of the same device to talk to each other, a prefix to indicate the signals that are part of the same group such as a bus, not using an in/out direction reference, handling bidirectional signals/buses, showing source termination, and handling stubs and buffers. The electrical signal type, such as HSTL or SSTL, were not included to keep the names short and to avoid having to change the name if the pad type changed.

Figure 2 shows the basic scheme. A number of signals originating in chip *CHA* and received at chip *CHB* are shown as examples of signals using the convention. The top two signals represent a differential bus named `BUSX` where the `_P` and `_N` represent the positive and negative senses of the signal. The signal `SIGY` is a single differential pair where `_PN[1]` is the positive sense and `_PN[0]` is the negative sense. The series-terminated signal `SIGT` shows how `_T` is appended to the part of the net that drives the termination. The naming of a clock feedback signal is shown for chip *CHA*, where `_FBO` is the output and `_FBI` is the input. At the top level, an arbitrary decision was made to use `_FBO` to indicate the feedback nature of the signal. On chip *CHB* the signal `SIGZ_B` is a bidirectional signal that goes from chip *CHB* to the connector for signals going to the board called `BDA`.

# 3. THE FPGA FLOW

The basic FPGA design flow was not much different from what would be used in many environments. The HDL used was Verilog, revision control was done using CVS, PERL was used for scripting, makefiles were used to run the various commands, and VERA from Synopsys [1] was used as the testbench language. All test cases were regressionable and self-checking so that they could be launched in batch mode. Synplicity [2] was used for synthesis and our target technology was Xilinx [3] Virtex-II FPGAs. The simulator we used was Synopsys VCS [1] and load balancing of our simulations was done using LSF [4].

## 3.1 The Pin-List Spreadsheet

The root of the overall flow was the use of an Excel [5] spreadsheet to capture all of the top-level information about the specific FPGA being developed. This pin-list spreadsheet template was originally created because we were developing a large number of FPGAs and there was a desire to automate many of the tasks having to do with specifying the functions of the various pins on each FPGA. Eventually, this spreadsheet evolved to be the key to linking all of the other design flows together.

The pin-list spreadsheet captured the following information:

- signal name, determined by the naming convention discussed in Section 2;
- the source signal and/or destination signal. If a signal had a multiple fanout, all destinations were specified, and for devices on that net only the source name was used, and no destination was specified;
- I/O direction, signaling standard (HSTL, LVDS, etc.) to be configured for that pin, reference voltages;
- clocking domain for the signal, especially for I/Os that used flip flops in the I/O pads;
- termination type if the internal Xilinx I/O terminations were used;
- FPGA bank. Xilinx I/Os are partitioned into eight different banks with rules about I/O types allowed in each bank;
- package information, as in type, number of pins;
- pin (actually ball) number for pins that had to be locked to a specific pin. Pins that could float within the bank were given an initial assignment during symbol generation.

A number of scripts were developed to process the information captured in the pin-list spreadsheet. These scripts could generate other representations of the data, do design rule checks, and generate the initial pin placements.

The first processing step was to convert the spreadsheet into a *csv* (comma-separated values) file so that it could be processed by various PERL scripts. For the RTL and verification designers, a script was used to generate the top-level Verilog module with the appropriate ports declared from the pin-list *csv* file. This module could then be easily regenerated as the pin definitions changed.

There were also a number of design-rule checks to ensure that various Xilinx rules were not violated. Some were scripts and some were built into the spreadsheet. These included checks for the number of pins per bank, compatibility of I/O standards to the voltage references in a bank, valid clocking of I/O flip flops, and cross-checking the consistency of the data in the spreadsheet.

The pin-list spreadsheet was not used just for the FPGAs developed. A simplified version was also used for all other

symbols. This was the best way to maintain the consistency of the naming convention in the schematic symbols by just generating the symbols from a single database. However, when using third party devices, the spreadsheet had to use the names from the behavioural model provided by the vendor to make sure that the signals could be found when doing the simulations.

The pin-list spreadsheet was also important for linking the FPGA flow to the PCB flow.

# 4. LINKS TO THE PCB FLOW

The PCB design was done using the Cadence [6] *Concept HDL* suite of tools for schematic entry and Cadence *Allegro* for layout. There were two main links from the FPGA flow to the PCB flow: symbol generation and PCB layout reporting. The symbol generation was the link into the PCB flow and the PCB layout reporting was a back annotation process that also included some consistency checking. The most important information to make consistent between the two flows was the information relating to the pins, such as the physical pin assignments and the characteristics of the pins.

## 4.1 Pin Specifications

The allocation of the signals to each of the banks in the FPGA was the responsibility of the FPGA designers. The decisions were made based on the internal floorplan of the chip, the floorplan of the PCB, and the allowable I/O standards based on the voltage references used in each bank. The FPGA designer could also lock signals to specific pins when necessary.

We also used the property that moving pins within a bank of pins would not significantly affect the timing characteristics of the FPGA. This gave some flexibility during PCB layout to move pins or flip buses within a bank if this would make the layout easier. Allowing and using this flexibility worked quite well. This meant that the final pin assignments would have to be extracted from the layout report files and merged back into the pin-list spreadsheet in case new symbols had to be generated due to a change. This back annotation process for the pins was part of the connection of the PCB layout flow back to the FPGA flow.

## 4.2 Symbol Generation

The flexibility and complexity of the FPGA designs required quite a number of symbols to be created for the schematic entry. The FPGAs we used were mostly in an 1152-pin BGA package so there were a large number of pins to track. It was important to automate this process to minimize the chance of introducing errors in the symbol creation.

The FPGA symbols were created in a two-level hierarchy. The top-level symbol captured the names of all of the signals without any of the physical attributes, such as pin numbers. The second level of symbols were partitioned according to the allocation of pins to the eight different banks as implemented on the FPGA by Xilinx. These were called the bank symbols. An important benefit of this hierarchy was that the remainder of the schematics could first proceed using only the top-level symbol before the final allocations of the signals to the various banks was done. This enabled an overlap that could save several weeks in the overall design time.

Figure 3 shows an overview of the symbol generation flow.

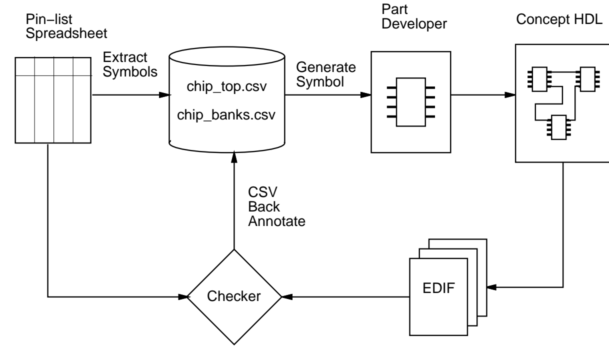


**Figure 3: Automatic symbol generation flow.**

By using a script, symbol information was extracted from the *csv* file of the pin-list spreadsheet. Two new *csv* files were created, one for the top-level symbol, called *chip_top.csv*, and the other to define the banks, called *chip_banks.csv*. These *csv* files were then converted via a script into the symbol files that could be imported into Cadence *Part Developer*, and then made available for use in the schematics.

## 4.3 PCB Layout Checking

After the PCB layout was complete a number of checks were done to ensure that no FPGA design rules had been violated and that the design requirements had been properly implemented.

The PCB layout information was extracted in the form of an EDIF netlist of the board. A number of scripts were used to process the EDIF netlist. They are represented by the *Checker* in Figure 3.

The main reason the *Checker* was required was to check the correctness of the pins after the swapping that was done during the PCB layout. The tasks of the *Checker* included:

- check that all the I/O types on a particular bank had compatible voltages for the output drivers, the input voltage references and the clamping voltages;
- check that all the signals in the pin-list spreadsheet were assigned to an I/O in the schematic symbols;
- check that all symbol pins were assigned to a valid I/O;
- check that all signals were paired with a compatible signal in terms of clocking and pin assignments. This addressed a Xilinx design requirement related to how clocks were distributed in the I/Os;
- check that all pins that were locked in the pin-list spreadsheet were not moved in the layout.

In addition to the design rule checks performed on the PCB layout, the pin locations used in the layout needed to be recorded and used to generate the pin assignment constraint file (UCF) for the Xilinx place and route tool.

## 4.4 PCB Layout Backannotation

The pin location back annotations were also done starting with the EDIF netlist extracted from the PCB layout. This file was converted into a *csv* file, called *backannotated.csv*. The new pin locations in the *backannotated.csv* file were then merged back into the *chip_banks.csv* file.

Note that the pin locations were not merged back into the original pin-list spreadsheet. Once the initial *chip_banks.csv* file was created, it became the central database for the pin

locations. If new pins were added into the pin-list spreadsheet, then a forward annotation was done to update the *chip_banks.csv* file.

## 4.5 BSDL Files for JTAG Testing

Once the final pin locations were determined and backannotated, all of the pin information was available to create the BSDL files needed for JTAG testing of the boards. A script was used to generate all of the BSDL files from the database, again without any worry about having inconsistent or incorrect data about the pins.

## 4.6 Further Pin Assignment Checking

To add another level of confidence in the FPGA pin-assignment process, a trial place and route of the FPGA was done as soon as enough RTL was available. This would make sure that the documentation of the pin-assignment rules corresponded to the rules implemented in the place and route tool. Passing the place and route using the generated *chip.ucf* file was a key goal before the PCB was sent for fabrication.

# 5. CONNECTIVITY CHECKING

In any PCB design flow, an important step after the design entry is to check that all of the connections are correct and that other miscellaneous components are correctly specified and connected such as termination resistors. When any of the components are designed in-house, such as the FPGAs, then it is best that the FPGA designers actually check the schematics to ensure that their FPGA has been correctly connected with respect to the signals, the power supplies, and the terminations required for the specified FPGA I/Os. The PCB designers must review the connections of all of the other components.

Schematic checking by hand is extremely tedious, time-consuming, and error-prone. We developed a *Connectivity Checker*, called *CAD*, written in C++ that would produce reports to make it much easier for inspecting the schematic. The input to *CAD* included the Verilog netlist extracted from the schematic, an extraction list, where users specified the modules that were to be extracted from the schematic and the expected pin lists of the modules. For example, to extract all of the FPGAs on a PCB, the names of all of the FPGAs would be included in the list.

*CAD* built the entire circuit in its database using a circuit synthesizer engine. Once the circuit was synthesized from the netlist, an initial electrical value was propagated to all the nets from the *power* and *ground* pins. *CAD* then generated several output files.

An extracted Verilog netlist for both component-level and system-level simulation was created. The original netlist could not be simulated so *CAD* would remove components or modify them, as well as remove the components that were not in the extraction list. Pull-ups and pull-downs that were in the schematic were replaced with a pull-up and pull-down module that could be simulated. Capacitors and resistors were replaced with modules that could be simulated. Power and ground pins and all their electrically equivalent nets were replaced by `1'b1` and `1'b0`, respectively.

The module list was used to compare the extracted pins with the pins in the spreadsheets. For example, for a chip called *mpp*, the *mpp_top* file that was generated from the spreadsheet was compared to the *mpp_top* file from the schematic. If there were differences between the spreadsheets and the schematics, *CAD* would generate an error report and indicate the missing or mismatched pins in the schematic.

```
**********************************************************

Pin/Bus Name  : ad_data           (OUTPUT)

**********************************************************
ad_data[3] INOUT  OTHER     bms_banks    ad_data3
           WIRE   OTHER     t            bmsdpc_ad_data[3]
           INPUT  CHIP      dpc_top (1)  ad_data[3]
           INOUT  OTHER     dpc_banks    ad_data3
           INOUT  R-H-0v75  rpac4        a[0]
..............
R-H-0v75   1
```

**Figure 4: Example of the connections for one pin from a Connectivity Report.**

Some of the most important outputs from *CAD* were the reports that were generated to facilitate schematic checking. A report was generated for each module on the extraction list. There were different levels of the report. One would just show the start and end point of a net and the type of pull-up, if any. Another would show a more detailed description of the path that it would take. This allowed the designer to selectively turn on or off varying amounts of detail.

An example item from a schematic report is shown in Figure 4. The example design report shows the path of the net `ad_data[3]` on the schematic and its endpoint. The first symbol that the net appears on is called `bms_banks`, and it is called `ad_data3` on that symbol. The next place that we see the signal is at the top level and it is called `bmsdpc_ad_data[3]`. It then goes into the `dpc_top` and the `dpc_banks` symbols. This path indicates that the signal goes from the `bms` chip to the `dpc` chip. Also, it is showing that it branches off to resistor pack `rpac4`. Below the path, is the indication `R-H-0v75`, which shows that the signal has a pull-up resistor tied to 0.75v on it.

In addition to tracing the path, there are two other columns of information. The second column shows the Verilog port type declaration from the extracted netlist and the third column indicates whether the signal was found on a chip-level view of the schematic, whether it is a resistor module or some other view.

The report generated by *CAD* made it much easier to trace the connection of each net on the schematic. The designer did not have to go through multiple pages of the schematic and try to follow possible name changes. The FPGA designers were extremely happy with this report as they could check the schematic with much greater accuracy in only about an hour, instead of taking over a day of concentrated effort.

There were cases of errors found in the reports that the board designers could not find on the schematic at first. They were reading what they wanted, instead of what they had done, such as crossing pairs of chip selects. A more insidious problem was a stub of a power net shorting a data line. The stub was not visible on the schematic.

Once the Connectivity Reports had been completely checked for the first time, any further changes to the schematic could be very quickly reviewed by simply generating a new set of reports and using the Unix *diff* utility to check for file dif-

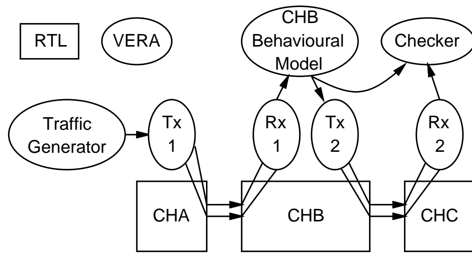


**Figure 5: Example of the testbench environment.**

ferences. Any changes in the schematic would be quickly highlighted for checking.

# 6. SIMULATION

There were several levels of simulation done to verify different aspects of the design, ranging from checking an FPGA by itself, co-simulation to allow software to do some checking, and system-level simulation to check the functionality of many interacting components.

Behavioural models of all of the FPGAs being designed were built and the models for third party devices, such as framers or memories, were also obtained.

All models could be automatically connected using netlists generated from the schematics.

## 6.1 Testbench Environment

Figure 5 shows an example of our testbench environment where chip *CHB* is being tested. The *Tx* and *Rx* VERA blocks are bus functional models (BFMs).

Using the capabilities of the object-oriented programming methodology, universal *Tx* and *Rx* BFM object classes were created to reduce the effort of building the large number of BFMs required. These universal BFMs were able to work using any bus naming and width. An internal hash table was used to load the pin-mapping table and the generic pin names of the universal BFM were dynamically bound to the actual pin names at run time.

Several different kinds of tests could be carried out, each using the same netlist generated by *CAD* from the schematics. In the *DUTLESS* simulation mode, the netlist itself could be tested before any RTL was completed using just the VERA behavioural models as connected in Figure 5. In this mode, *Tx2* could also be configured to inject errors to help debug the *Checker* prior to using the RTL. If the RTL for *CHB* was to be tested, then *Tx2* was removed and the RTL for *CHB* was included. To test the interfaces of *CHB* with the RTL for *CHA* and *CHC* then the RTL for those chips would be included and *Tx1* would not be required.

Using this environment, it was straightforward to test a single device or to simulate a number of devices, which we called system-level simulation.

## 6.2 System-Level Simulation

The system-level simulation could be used in many ways. Given the board netlist and the behavioural models of all of the components, it was possible to simulate the functionality of the entire card. In fact, it was even possible to simulate multiple interacting cards using our framework. The entire line card could be simulated from the input to the framer chip, through the input datapath, looped back at the switch fabric interface to the output datapath and through the output of the framer. With behavioural models it took about two minutes to send about 100 24-byte packets and about 10 minutes using RTL. To test the signaling in the scheduler system, an RTL system simulation of two fabric scheduler cards, two line card controller cards, and one line card for 15 fabric ticks (about 3 million simulation steps) took about ten hours on a 2 GHz P4 with 2 GB of RAM running Linux.

System-level simulation also exposed protocol problems between our chips and third party devices due to incorrect configurations and misundertanding the data sheets.

Many of our FPGAs had insertion and capture buffers at the interfaces that were software accessible. Interface testing at the system level was often done by utilizing these buffers in some form of system-level model. These tests were a key part of the testing when the hardware arrived and could be used to automatically generate scripts to test the hardware. This is discussed in Section 7.

## 6.3 Co-simulation

To be able to get the software group into the early testing loop a simple co-simulation interface was developed. Instead of using commercial tools, we created a simple link between the processor platform used for the software development and the Synopsys VERA testbench environment. This link took advantage of the structure of the software and was implemented using sockets to open TCP/IP network connections. The software drivers had a layer where it was possible to extract the lowest level reads and writes to memory without affecting the application code. Only some code below the programmer visible interface was touched.

The reads and writes were converted to accesses to a socket that was created over the network between the processor platform being used to test the software and the system running the chip simulation. At the simulation end, the accesses received on the socket were converted to look like accesses from the on-board processor from the perspective of the simulation.

Although this methodology does not exactly model the interface it was quite adequate for testing the software. The only part not tested would be the actual physical interface between the processor bus and the access to the chip.

Co-simulation was mostly used to test that the drivers accessed the registers properly with the proper memory map. In some cases, diagnostic features on a chip would be exercised using the software to test some higher-level features.

With the ability to do system-level simulations with multiple boards it was conceivable that large portions of the entire system could have been simulated and run with the actual software though this was never attempted.

# 7. VALIDATION SCRIPT GENERATION

After the first smoke tests to determine if new boards would power up correctly, the first task was to get the on-board processor up and running. The processor would run a board-support package (BSP) that included a monitor that allowed basic memory accesses and other functions. *Validation scripts* written in PERL could be run to carry out functions like inserting a packet at a transmitter interface, capturing it at a receiver interface, and checking the result against an expected value generated during simulation. The scripts used a handle to a telnet object that would telnet to the board and execute monitor commands.

Most of the tests done on the system-level simulations were done using only the microprocessor interfaces to the chips, so many of these tests could also be done on the actual hardware. As a result it was possible to generate a large number of scripts automatically while running test cases on a system-level simulation. This was done by running a test case and capturing the memory transactions on the processor that accessed the various chips. This memory transaction log was converted to a script that could be used to validate the actual hardware. The only caveat was that in the real world the interactions obviously happened at different time scales so that some timing-related activities had to be accounted for in the scripts. To handle real-time delays, a delay statement was inserted in the simulation test case that had no effect on the test case, but added a delay in the script. The other case to handle was the polling of status bits. A dummy polling statement was placed in the test script after the simulation polling loop. It would generate a polling loop in the validation script. Events asynchronous to register accesses could not be validated with these scripts, such as the use of hardware packet generators, since they would run at much different rates than in the simulation.

This process could be extended to system integration when a new card was added to the system. For example, a system-level simulation could be run for the multiple cards in the system. Packets could be injected on one card and then captured and observed on another card. Scripts to do this on the hardware would be automatically generated, significantly simplifying the task of generating the scripts and debugging them. If the test case on the simulation worked, then the actual test on the real hardware using the automatically generated script had a high likelihood of working, or at least when problems were observed, it was unlikely to be a problem in the script.

Automatic script generation for hardware validation and system integration was extremely useful and showed the merits of having a fully integrated tool flow.

# 8. RESULTS

The true measure of success can only be demonstrated by the first-time success of a complex PCB. There were two large boards that give the necessary proof.

The first board was the datapath card for a linecard. It was the first board to go through our complete flow from the beginning as well as the first to use the Cadence PCB tool flow. It contained nine 1152-pin Xilinx Virtex-II FPGAs on a 28-layer PCB with 5400 components, 7082 nets, 41317 pins, 30740 connections, 25180 inches (0.64 km) of etched copper, and 200 constraint/topology classes. The design took six months from drawing board to gerbers and had one barnacle due to a problem not detected by our tools. One of the clocks had an incorrect pullup and the requisite check was quickly added to the flow.

The second board was the scheduler card for the switching fabric. It had 11 Xilinx Virtex-II FPGAs, most of which were in 1152-pin packages. The PCB had 32 layers, 5500 components, 7433 nets, 37411 pins, 25800 connections, 26930 inches (0.68km) of etched copper, and had the characteristic of high routing density and just being a very large design. The design nine months from drawing board to gerbers, including four months of routing. The only barnacles were due to a specification sheet that had incorrect information on it. No barnacles were due to the design being improperly captured.

To add even stronger evidence to the success of the scheduler card using our flow, the expectation was that it would take several weeks to get the card up and running and longer to get it integrated into the system. Instead, it took less than a week before the card was installed into a system and running all of our validation scripts.

## 8.1 Cultural Observations

It is interesting to add a note about design cultures that was observed. There is no claim here that this is true everywhere, but it is worthy to mention that a difference in cultures can make a large effort like this more than a technical challenge.

The board designers tended to be more *visual* and reluctant to accept the flow at the beginning. For example, there were problems getting the board designers to accept the naming convention that was developed. They still preferred to view the schematics rather than using the reports generated by *CAD*. The FPGA designers were much more scripting and automation driven and they were the main drivers for the development of the overall environment. However, the success of the flow on a number of early cards got everyone on board, and the success of the most complex cards was the final convincing evidence.

# 9. CONCLUSIONS

The attempt to connect all of the design flows can be deemed to be highly successful based on the results from the implementation of two complex boards. To be first-time successful when implementing a PCB, extensive checking, double-checking, and simulation is required between all of the individual design flows within the system. In our system, a consistent naming convention and a multitude of scripts and custom tools were required to pull all the flows together, but the results were well worth the time invested.

With further effort, other interesting extensions could be considered. If the pin-list spreadsheets were extended to include all of the connectivity for each device, then most of the schematics could likely have been generated automatically. In the FPGA verification environment, much of the testing is similar to the tests that the software drivers would perform. Assuming that the drivers were written in C++ and if a C++ verification language was used, like TestBuilder [7], then *methods* for testing each device could be directly imported into the device drivers.

# 10. ACKNOWLEDGMENTS

D and W also made significant contributions to this environment. P and M provided the board data. Finally, we want to recognize the efforts of the entire amazing XYZ team. . .

# 11. REFERENCES

[1] www.synopsys.com.
[2] www.synplicity.com.
[3] www.xilinx.com.
[4] www.platform.com.
[5] www.microsoft.com.
[6] www.cadence.com.
[7] www.testbuilder.net.