# Scalable RTL in Design and Verification

Unisys Corporation

Ross Weber

Session # 2.5

Presented at


# 1. Abstract

While verification tools have made great strides in capacity over the last few years, there are still limits in their effectiveness with large designs. However, such limits can be compensated for with scalability. Some of the reasons that designs are large, such as deep FIFOs and memories, may actually have little to do with essential functionality or common design bugs. In fact, it is rather common to abstract such structures down to smaller sizes for the purposes of more focused verification. This paper discusses the application of an implementation and verification strategy that calls for scalable RTL in which the size (depth, width, number of ports, etc.) of certain design structures are made easily adjustable. This design approach has several clear advantages, including fostering easier reuse and enabling late-stage design changes without having to re-enter the RTL. Ease of abstraction for formal analysis is another advantage of scalable RTL. This paper discusses the experiences of one Unisys development team on past and current projects using this design technique and the attendant benefits. It includes specific examples of design structures easily transformed from difficult to tractable for verification. RTL scalability is a technique sometimes recommended in vendor formal training classes; this talk discusses the challenges for designers to use this approach and presents the real-world results achieved. It is appropriate for anyone doing verification of large or complex designs. No specific knowledge of assertions, properties, or formal analysis is required.

# 2. Introduction

This paper discusses the introduction of scalability into a design's implementation and verification strategy. It describes how past designs were implemented and verified and the hardships incurred because of the lack of scalability. The paper explains the benefits of scalability to the RTL implementation as well as the benefits to different aspects of verification including formal analysis, simulation, and acceleration. In addition, the difficulties designers had with the new implementation style and verification methods are examined.

# 3. Prior Design Work

## *3.1 General Design Description*

The past designs referred to in this paper include several chips that allow for scaling servers, i.e. adding in more processors, memory, and I/O. The major components of this type of design are highlighted in Figure 1. They include processor units, I/O units, memory units, remote interface units, and a crossbar unit connecting all of the other units together. Assume that the units within the chip communicate via a common protocol and that this same protocol is used for communication with the remote nodes as well. Further, assume that the external processor, I/O, and memory components all communicate to the chip with their own distinct protocols.

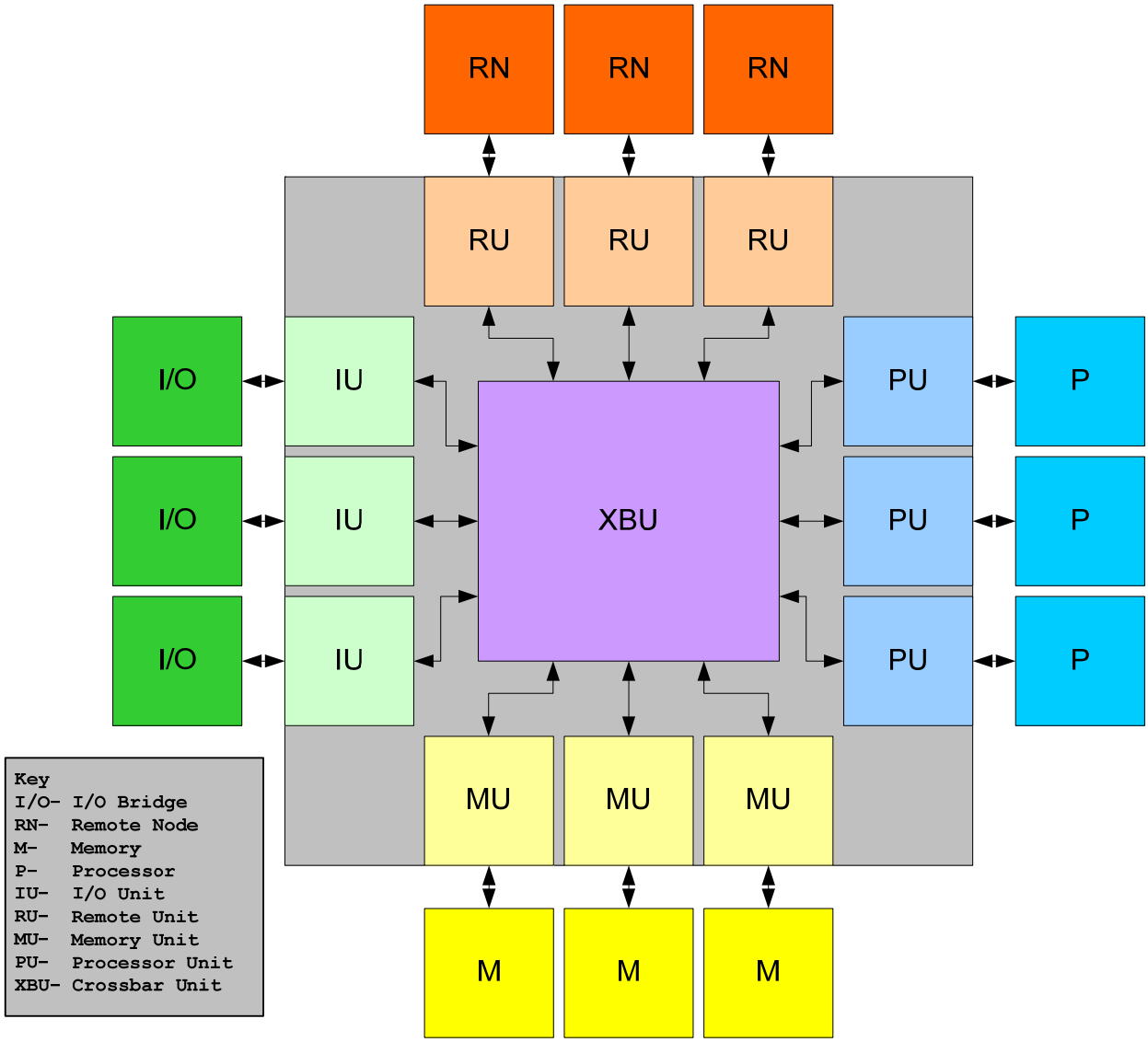


Figure 1

Figure 2 shows one embodiment of multiple chips being connected together to enable growth of processing power, I/O devices, and memory capacity.

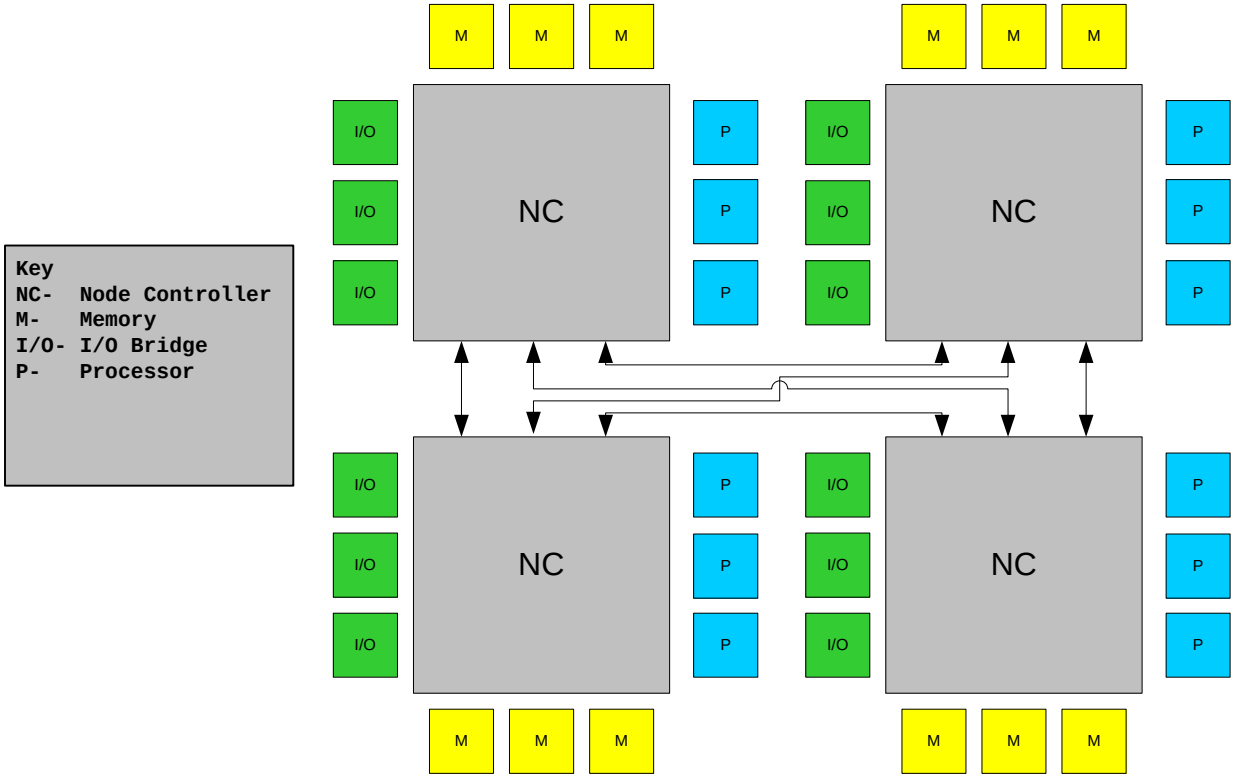


Figure 2

### *3.2 Implementation and Verification Strategies*

The implementation and verification strategies applied to these designs included the goal of quickly developing the sub-modules of each unit so they could be built into a unit-level testbench as soon as possible. Using this approach instead of applying a widespread effort to focus on smaller testbenches to verify the sub-modules independently had several advantages. It allowed for the focused development of only a few models for the major chip interfaces including a processor model, I/O model, memory model, and a model for an entire remote node. As mentioned, the remote unit communicates with the remote node controllers with the same communication protocol as the other units do with one another over the crossbar. Therefore, the model developed for the remote node was not only able to emulate another entire remote node, but also each component individually; that is, it could model each processor unit and processor, or each I/O unit and I/O bridge, etc. Figure 3 shows the models replacing the actual processors, I/O bridges, memory, and remote nodes in a chip-level testbench.

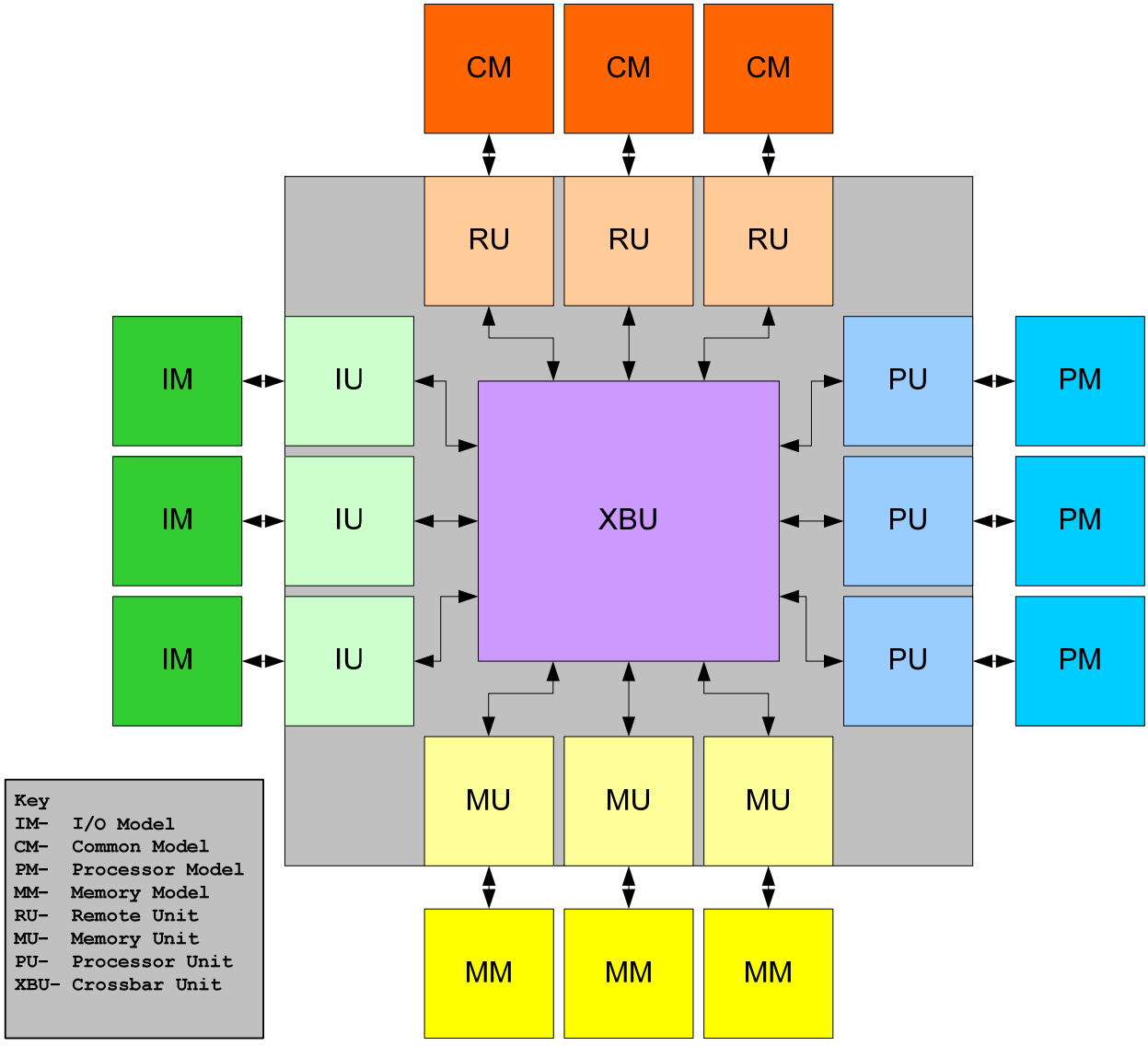


Figure 3

Figure 4 shows one embodiment of a unit-level testbench. Note how the Common Model replaces nearly all of the other components in the chip allowing for a more compact and focused unit-level testbench. Unit design teams generally relied on 1 or 2 workhorse testbenches to accomplish unit-level verification of their design.

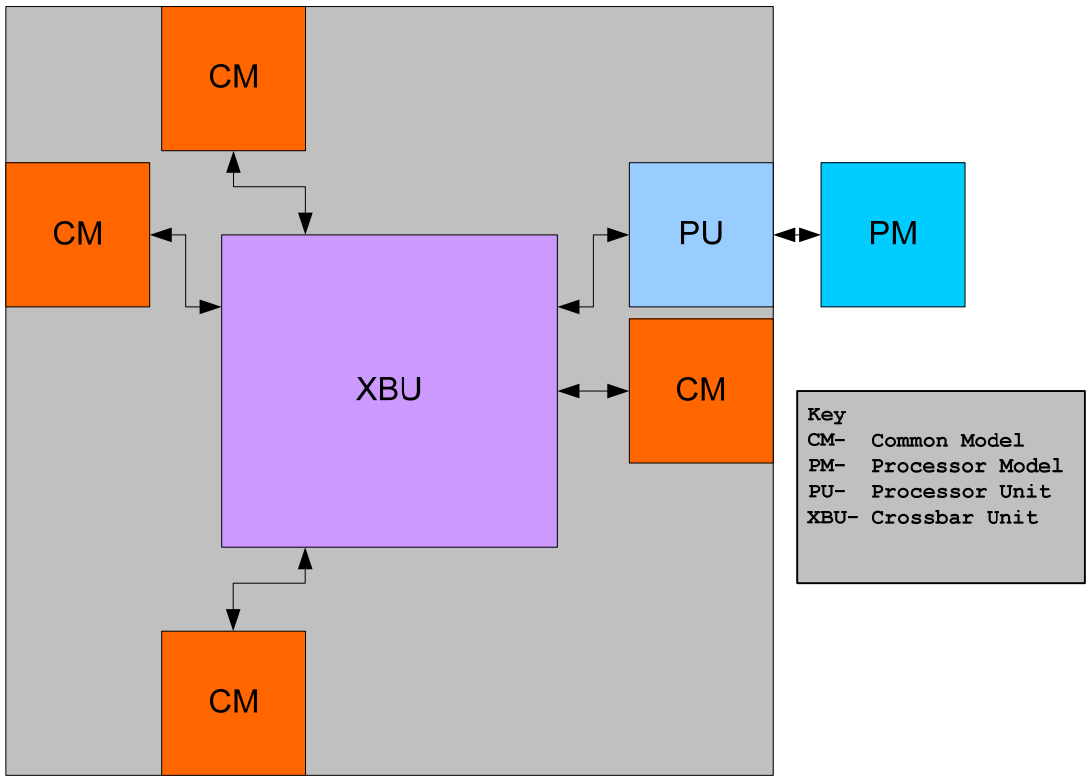


Figure 4

The quick development of unit-level testbenches enabled early detection of module-to-module communication problems. These types of problems are very common when modules are implemented by different designers who may have had slightly different interpretations of either a specification or other agreed upon implementation ideas. The early unit bring-up also had advantages physically as it allowed for early looks at unit sizing, congestion, and timing problems.

The focused model development and the focused development of the unit designs allowed for the early development of system verification testbenches. These system-level testbenches included multi-unit, chip, or multi-chip topologies. They attacked the unit-to-unit communication problems, the system performance characteristics, and the overall verification of system algorithms including coherency, address mapping, and routing.

Figure 5 shows one embodiment of a system-level testbench connecting design units and models together in a partial multi-chip testbench.

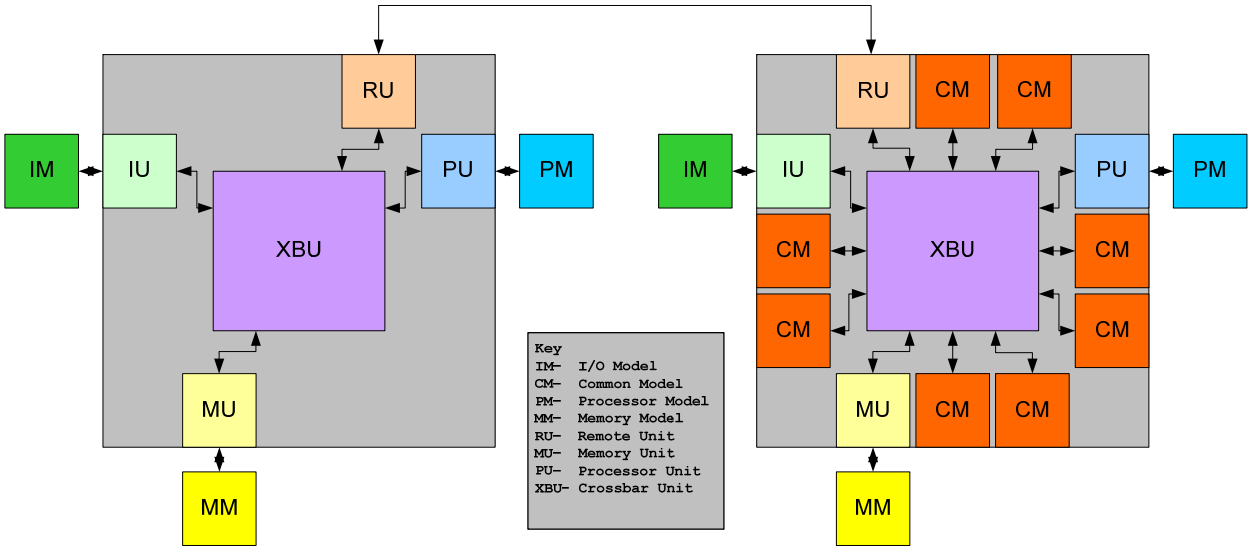


Figure 5

For thoroughness, it should be mentioned that the workhorse testbenches mainly operated in a constrained random environment where self-checking properties ensured legal behavior while coverage sequences made sure interesting functions were being performed and provided data to drive future verification efforts and indicate verification progress.

Although not the main focus of the verification effort, some sub-module verification was also done. Some designers found the time and necessity to create module-level testbenches where they had more control over how the module interfaces sequenced allowing for more focused verification. For many designers, however, the reward of a more focused verification environment did not outweigh the significant costs associated with such a task including the difficulty in creating models to drive a module's complex interfaces, the extra time needed to debug the models, and the ongoing need to update the models as design requirements changed. Also, designers struggled with the fact that the work developing such testbenches would not be reused as the module was reused in other testbenches.

The advent of formal technologies could have helped to mitigate some, but not all, of the difficulties and concerns designers had when developing the module-level testbenches. However, the technology did not yet have widespread adoption because of several factors. The designers did not have significant training or experience with the tools or technology. Many of the design groups did not have people championing the growth of formal analysis looking for opportunities to apply the technology and to help others learn how to apply the technology. The designs lacked a comprehensive implementation and verification strategy that included formal analysis as part of the total verification suite. The design schedules could not afford the additional learning curve time associated with adopting formal analysis. These factors, in addition to designer inertia, instead continued to pull the teams down the simulation only path.

There were individuals and small groups of people who did apply formal technologies as part of their verification of the modules they developed. In particular, a library of shared modules used by all design units was developed. This library consisted of arbiters, FIFO controllers, timers, counters, encoders,

decoders, and other common chip specific functions like address mapping functions. These well defined functions were perfect candidates for easily applying formal analysis and some of the individuals developing the modules successfully applied formal technologies to their designs.

Another interesting aspect of the implementation and verification strategy was creating fully synthesizable testbenches, including all the models previously mentioned. This approach enabled entire testbenches to be mapped to gates and put into a Palladium hardware accelerator box. The Palladium accelerated the performance of the simulation by approximately 1000 times over the performance of the NC-Verilog software simulator for a typically sized unit-level testbench. Early on in the development of the design units, the performance boost was used to run many short tests to go after the seemingly endless possible configuration settings and modes of operation available in the design units and models in the testbenches. Then, as the design stabilized, the Palladium was used to troll for very deep problems. In every project, the Palladium was beneficial in finding very deep or very rare problems that the NC-Verilog simulation runs had not hit yet.

### *3.3 Detailed Design Description*

To handle typical system loads, the designs mentioned are built with sufficiently large buffering structures to meet system-wide peak bandwidth requirements. One example of a large amount of buffering is in the crossbar unit. The crossbar unit buffers up transactions when several source units are trying to communicate with a single destination unit and arbitrates amongst the sources. The crossbar has several communication channels and each channel has individual buffers for each source and destination pair. The buffering is necessary since the destination units typically have many less input ports compared to the possible number of units trying to communicate with it.

Another structure common to several of the units in these designs is a transaction tracking structure. The transaction tracking structures are designs that collect and hold information eventually used to respond to a device that made a request. An example transaction flow is shown in Figure 6. The I/O device makes a memory request via an I/O bridge to the I/O unit. The I/O unit stores the request, shown by the black bar, in its transaction tracker and forwards another request on to the memory unit. The memory unit stores the request in its transaction tracker and forwards another request on to the processor unit whose processor has a copy of the requested data. At the same time, the memory unit makes a request to the memory. The processor unit stores the request in its tracker and forwards a request off to the processor. The processor responds to the processor unit and the processor unit's tracker responds back to the memory unit and deallocates its tracker entry. Once the memory unit has collected responses for all its outstanding requests, it responds back to the I/O unit and deallocates its tracker entry. Based on the response received, the I/O unit then forwards an appropriate response back to the I/O device via the I/O bridge and finally clears its tracker entry. Each unit's specified tracker depth, i.e. the number of transactions it can track, is usually set based on the system's peak bandwidth requirements.

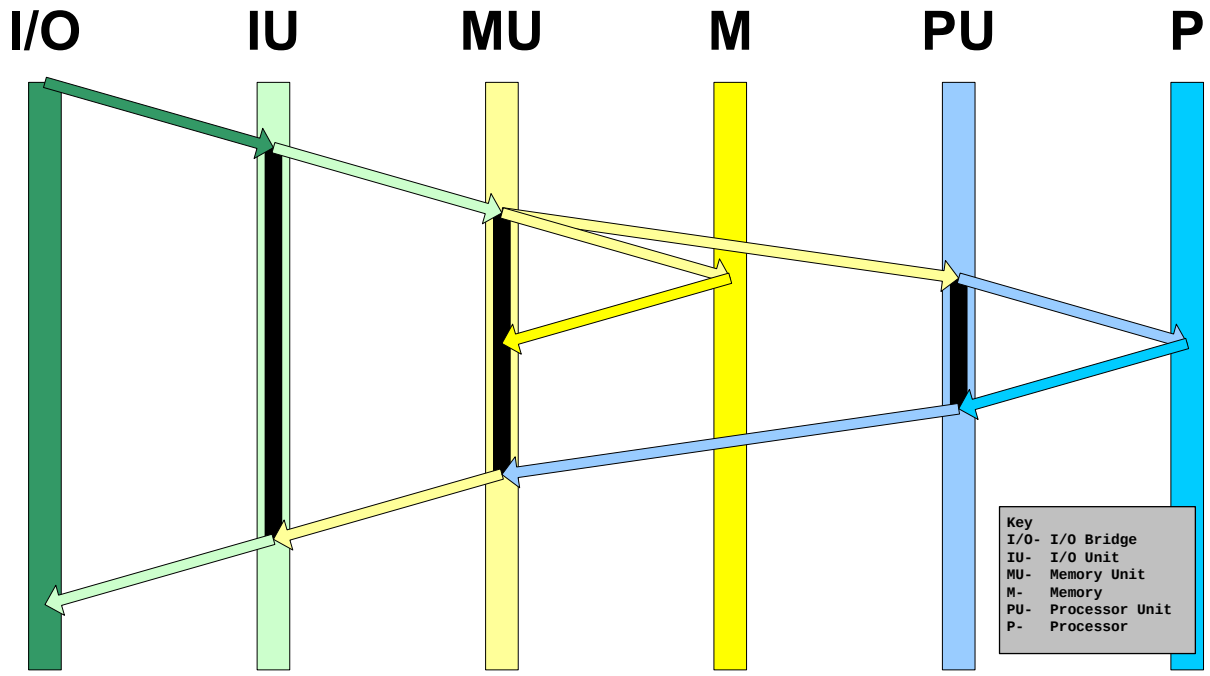


Figure 6

The aforementioned structures are not the only buffering structures in the chip, there are also numerous input smoothing FIFOs, FIFOs to cross clock domains, and other control oriented buffers used to track interface protocols in each of the units.

Other common pieces of logic in several of the design units are counters used in fairness algorithms. Counters become important to fairness algorithms when the protocol uses a retry scheme to handle transaction conflicts. Assume that a transaction conflict is any situation when a unit cannot handle a particular transaction at the time it is processed. Then, assume that the protocol uses a retry scheme to respond back to the requester telling it to try again later. Without some kind of counting mechanism to either give some indication of how long a requester has been retrying a transaction or to give some indication of how many times the transaction has been retried, that requester could theoretically get locked out with the design unit always handling other requests. This is called a livelock. Instead, the counting mechanisms indicate to the fairness algorithm when these potential livelock situations arise and appropriate actions are taken to prevent the livelock. One action may be to retry all other requesters except the one that has been locked out for a while.

## *3.4 Verification Hardships*

The design descriptions given do not fully describe the details of the design, but instead give an indication of the potentially deep state buried in the design. Also, the descriptions highlight that sometimes critical actions occur when the deep state of the design is reached. Getting the design into these deep states are critical to ensuring successful verification. The design descriptions help to explain why such large buffering structures are needed. It should be clear that the large buffering structures are frequently in place to prevent the actions that occur when the buffers become full. Thus, the very nature of the design is working against the goal of verifying the critical boundary condition actions.

Another hindrance to the verification effort occurs when design units are put together in chip-level and system-level testbenches. The large structures in each unit multiplied by the numerous instantiations of each unit again multiplied by several instantiations of the chip all contribute to very large simulation snapshots. They are so large in fact that they can grow out of 32-bit simulators, leaving only 64-bit machines to run on. Even when the testbenches do not grow large enough to limit the machines they can be run on, their large size contributes to slower simulation speeds. As mentioned previously, many testbenches are put into a hardware accelerator. The hardware accelerator resource is very expensive; filling the machine with large buffers that do little to contribute to the verification of the complicated control algorithms is a very poor use of a costly resource.

Designers generally do a good job of putting in logic to allow for programmatically reducing the logical size of buffers and counters. They accomplish this by putting in programmable thresholds to cause early full and timeout indications. This programmability allows for better focus on the control algorithm functionality. However, it does not do anything for the physical, simulator, and hardware accelerator size issues.

## *3.5 Physical Design Hardships*

Large buffer structures not only create verification hardships but also impact physical design. The estimations done to size buffer structures in a design are not an exact science. The estimations are decided upon by iteratively considering many factors such as system-level modeling and die size. However, no matter how thorough the process, the specified sizes are just estimates and subject to change as units on the chip grow unexpectedly due to tight timing constraints, routing congestion, feature creep, and other unaccounted for logic additions. The growth results in the need for units to reduce or remove logic and features. In all of the designs represented in this paper, there has been a need to significantly reduce major buffering structures in the design.

To better anticipate the need to shrink designs, logic implementers have become better at parameterizing their designs. However, sometimes the implementation of scalability is incomplete in many ways. Individual designs may fully implement a parameterized design, but the modules that instantiate the design may only handle a specific parameter value. Designers may implement their design to use a parameter in most of their logic, but because of complexity, not in all of it. Using a parameter to make the logic easier to change when a different parameter is needed helps the implementer. However, the fact that even a small amount of logic needs to be changed to allow for scaling makes a design less flexible when physical design requires experimentation with sizing options. Additionally, the more structures in a design available for scaling, the more options available to consider when physical design requires the design to shrink.

## 4. Current and Future Design Work

The current and future generations of the chips discussed thus far have a slightly new architecture. As shown in Figure 7, new processor and I/O interfaces required a new design to communicate with them and the new communication protocol made it necessary to add crossbars between several interface units (CIUs) and transaction handling units (ITU, PTU). The focus of the remainder of the paper is on a new implementation and verification strategy applied during the design of the Common Interface Unit, the CIU block.

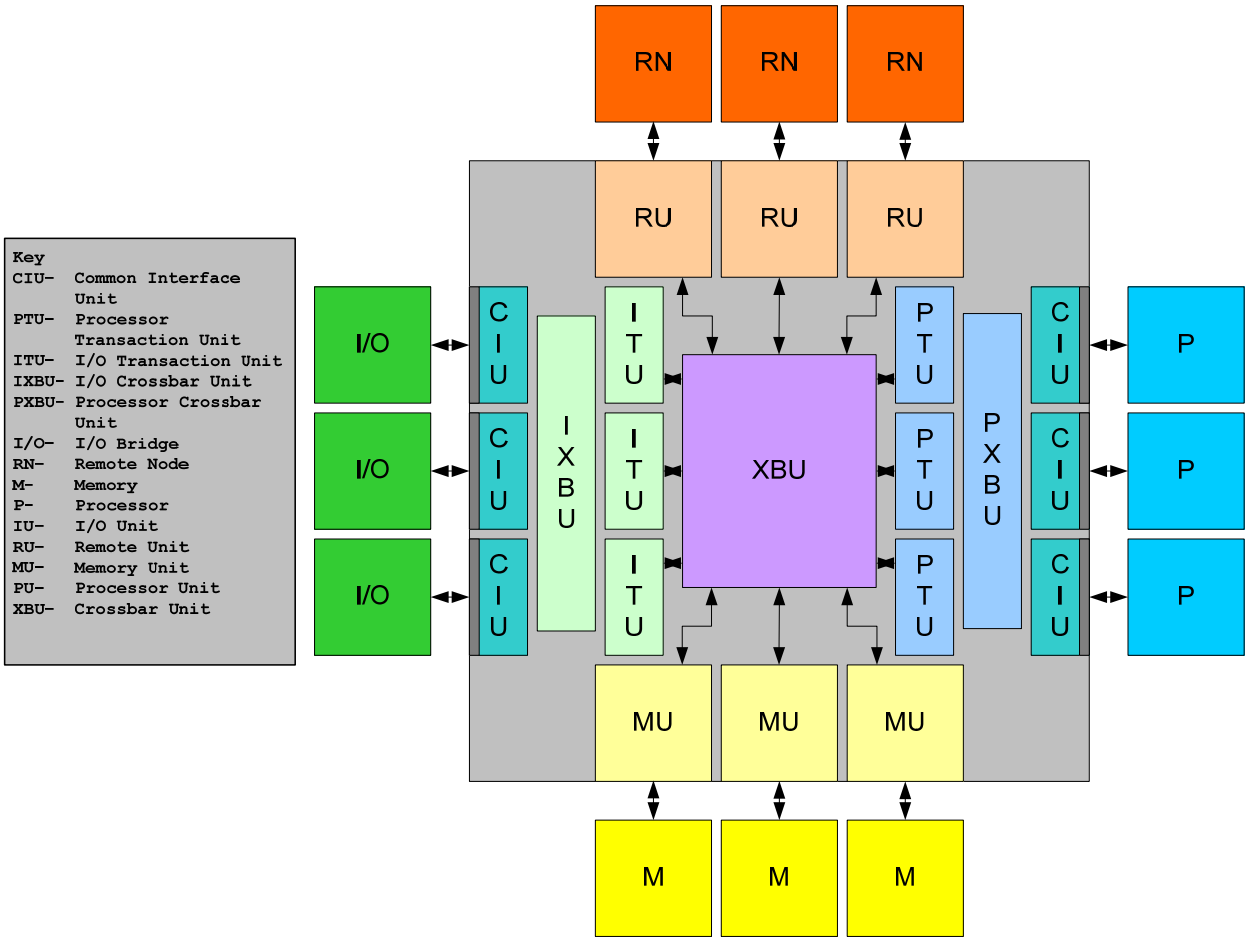


Figure 7

## *4.1 Detailed Description of Focus Block*

The teal CIU block shown in Figure 7 is instantiated on both the processor side and the I/O side, once for each processor and I/O bridge interface. The same communication protocol is used over both off-chip interfaces.

Figure 8 shows a more detailed, but still high-level, depiction of the CIU block. The bottom of the figure is the physical layer which converts high speed serial input data into wider packets of information and presents them to the CIU block on the receive (RX) interface as shown in the figure. The physical layer also performs the opposite function of converting the packets from the CIU block on the transmit (TX) interface into serial output data. The CIU design performs the link layer functions for the protocol including ensuring reliable data transmission and proper link initialization. It also implements the buffering and flow control logic to provide consistent, deadlock, and livelock free transaction flow from the RX interface to the inbound (IB) interface and from the outbound (OB) interface to the TX interface.

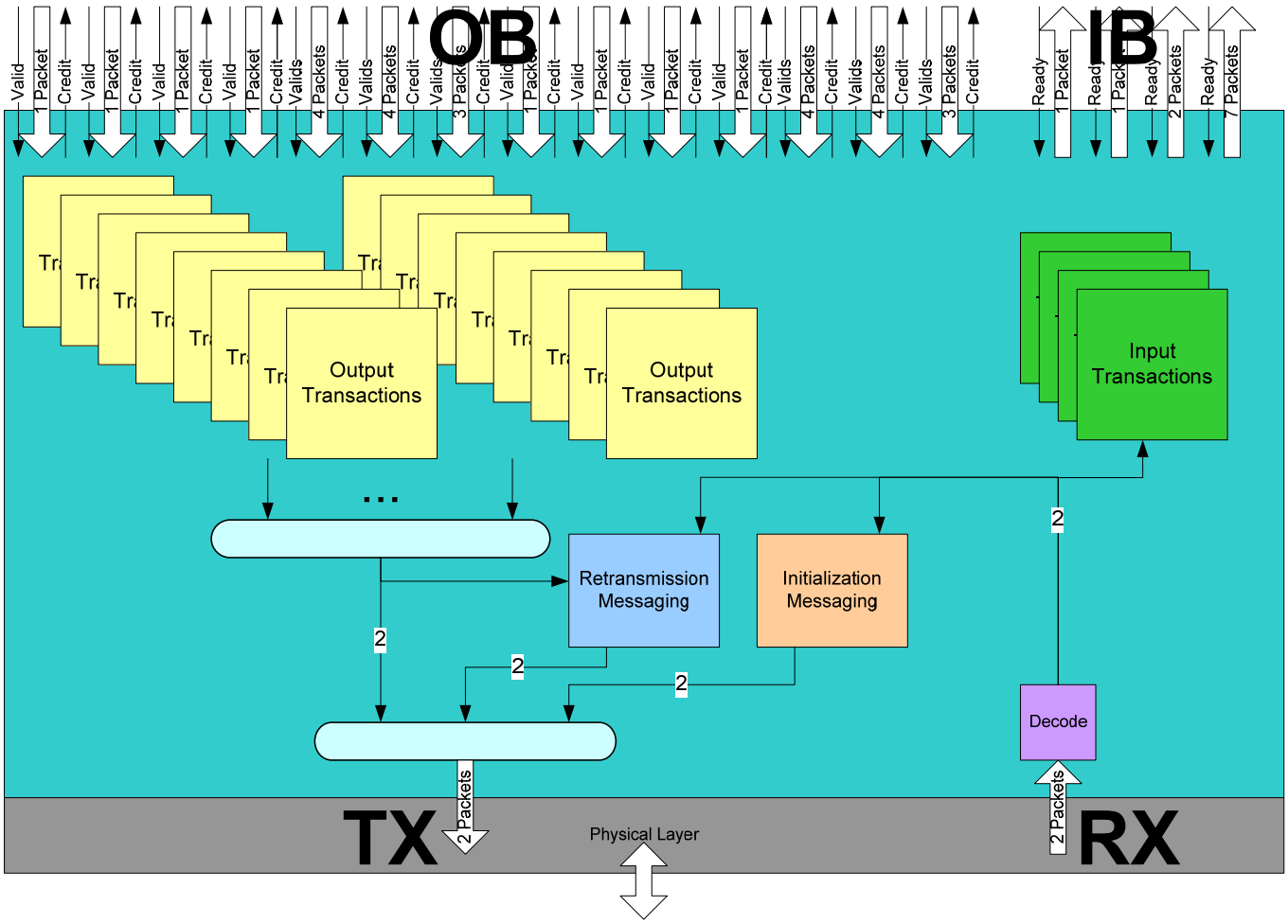


Figure 8

The TX and RX interfaces are specified as 2 packet interfaces. The packets on these interfaces carry transactions for several virtual channels. Each virtual channel in this protocol has their own set of credits to allow them to flow independently of one another. There is also a shared set of credits that all virtual channels can use. The CIU design maps the narrow 2 packet virtual channels on the RX side into several,

multi-packet physical channels on the IB side. Similarly, it maps the physical channels on the OB side back to the virtual channels on the TX side. In each direction, the design provides a significant amount of buffering to meet protocol requirements and to smooth traffic during peak bandwidth times.

To ensure reliable transmission, the protocol performs retries when CRC errors are detected. The retry protocol requires the TX side of the link to maintain a history of previously sent transactions in anticipation of possible CRC errors being detected. When the RX side detects a CRC error, it requests retransmission. Once a request for retransmission is received, the link acknowledges the request and then begins transmitting from the retry buffer from where the request for retransmission indicated until it is empty.

Upon reset, the link goes through a complicated initialization sequence to communicate with the other side of the link in order to exchange information allowing each side to appropriately prepare for normal operations and initialize.

To complicate both the retry protocol and initialization protocol state machines, the messages used to communicate do not go into the retry buffer and thus each algorithm needs its own form of retry when the messages are lost due to CRC errors.

## *4.2 Introduction of Scalability into the Implementation and Verification Strategy*

The implementation and verification strategies employed in previous designs were effective. They allowed for early detection of critical module-to-module and unit-to-unit communication problems. The strategies enabled early looks at sizing and timing problems. The strategies were comfortable for designers, both implementers and verifiers. However, as explained, there have also been some difficulties. The verification of deep boundary condition logic has always been difficult. Most designs did not utilize powerful formal technologies to find problems that simulation cannot and to gain confidence in a design by proving the design's properties. Some simulation snapshots for large testbenches were so large that they had grown out of 32-bit simulators. The Palladium accelerator box was being filled with large buffers that were not contributing to the verification of the boundary condition algorithms. Late-breaking sizing changes because of physical limitations were difficult to re-implement and had not been previously verified. The lack of scalability options limited the choices when considering how to reduce the size of the designs.

Moving forward, it was decided that the implementation and verification strategy would call for adding significantly more scalability to the design and, at the same time, still continue to realize the benefits of previous strategies. It was believed that having a design that could easily shrink would allow for more effective verification. Full testbenches could be scaled to target critical boundary condition logic. In the software simulator, NC-Verilog, this meant that the testbench was more likely to fit into a 32-bit simulator and would more likely be probing critical logic areas. In the hardware accelerator, a scaled testbench was more likely to fit in the box and the costly resource was more likely to be targeting critical logic. Also, scalable testbenches allow formal technologies to be applied to a larger set of modules. Physically, a scalable design gives designers the tools to react appropriately and confidently when sizing changes are required.

### 4.3 Applicability of Design to New Strategy

The CIU design offers many opportunities for scaling. Some obvious candidates are the input and output transaction buffer sizes. The specification sized these buffers to be quite large to reduce the possibility of them filling during normal operation which would cause back pressure leading to system-wide performance degradation. However, when one considers that the sizing of the buffers is just an estimate and may change, that the design may grow out of its agreed upon footprint and need size reductions, and that verification efforts would like to focus on the control algorithms on the boundary conditions, it becomes obvious that these buffers needed the ability to be scaled down in size. Similarly, the retry buffer mentioned previously was also sized to protect against performance degradation and is a good candidate to be made scalable. Furthermore, note that the OB and IB interfaces in Figure 8 are specified to be anywhere in the range from 1 to 7 packets wide. It was clear that the packet counts on the interface were sized based on throughput estimates for each channel. Because of past routability problems with large crossbars, it was apparent that the packet counts on these interfaces were subject to change in the future and thus needed to be scalable.

## 5. Application of Scalable Design Strategy

### 5.1 Language Constructs

Implementing a design in a way to make it scalable requires a set of skills that is new to some designers. The language constructs, coding techniques, and code flow used to implement a scalable design can be very different than what many of the designers are familiar with. Instead of implementing the specification in a straight-forward manner, the designers have to generalize the specification and come up with algorithms that work for variable sized structures. Some of the verilog language constructs valuable to a scalable design include for loops, index part-select, generate blocks, multidimensional wires and regs, and constant functions. The example in Appendix A is a multiple read and multiple write port FIFO and illustrates how many of the language constructs are used. Note, the module is just an illustration and is not actually used anywhere in the design, although similar structures are used.

Lines 1-40 show the interface to the module. The parameters allow for any number of read and write ports as well as any depth and width of the internal memory. Verilog has a shortcoming in that it does not allow multidimensioned input and output ports. So, to pass data through the interface for a parameterized number of ports, it is necessary to make a single data port wide enough for the data of all ports. These flat, wide ports are immediately converted into multidimensioned nets to be used by the rest of the design. This conversion can be considered an extension of the actual interface. The index part-select language construct is used to select the appropriate data field.

Lines 41-94 show the write port logic of the module. An important construct is the constant function used on line 42 to calculate the base 2 log of the depth of the FIFO. The result is the width of the vector needed to address the memory. The always block containing the write port logic, lines 55-82, loops over the number of write ports updating the memory with the appropriate write data and write address when the port indicates that a write is needed. The loop also produces a full indication for each port. Lines 83-94 are the write address and memory state assignments.

Lines 95-129 contain the rest of the functional code. The read port logic is similar to the write port. There is also the definition of the constant function referenced previously. A base 2 log constant function is a highly used function when creating orthogonal parameters in a scalable design.

Lines 130-210 contain some verification code for the FIFO module. The verification code is not comprehensive but is available to provide examples for discussion. There are properties to ensure correctness on the interface, end-to-end properties verifying data integrity, and coverage sequences detecting interesting functional conditions. Further discussion of how these properties and sequences are used in formal analysis is in the next section. Also, more details about how the properties and sequences reacted when formal analysis was applied to this scalable design are in section 6 describing the results.

## *5.2 Applicability of Formal Analysis*

Formal analysis has shown to be effective in fully verifying well defined modules that do not lead formal tools into exploding state space. As modules are put together and built into larger testbenches, it becomes more difficult to avoid state space explosion. In general, the more scalable a design is, the more likely that it is going to be able to avoid such difficult problems. Like simulation, a consolidated design allows formal engines to attack critical boundary condition logic in the design. Once a design and testbench become tractable for formal analysis, then the significant benefits of formal analysis can be fully taken advantage of.

Formal analysis has some significant advantages over simulation. For instance, where a simulation testbench would drive a design with legal weighted random sequences, a similar formal testbench can drive that design with all legal sequences. The exhaustive nature of formal analysis offers more than just confidence that a design property is being followed; it can prove that it is being followed. At the same time that the formal engines are attempting to prove the properties in the design, they are also showing that interesting coverage sequences are reachable. When a feature's associated properties are proven and coverage sequences shown to be reachable, it is a good indication of verification completeness.

In addition, a formal testbench, with IFV, can be easier to create and be more effective than a simulation testbench. Some of the reasons are deficiencies in the simulators, but none the less, they are valid. The IFV tool automatically creates a default square wave clock and has simple clocking commands for creating different clocks. This seemingly simple thing is a nice time saver when creating many testbenches. A formal tool does not need random number generators to drive nets randomly. Instead, all legal values and sequences of values are considered when a net is undriven. In the FIFO example in Appendix A, the rd, wr, and, wr_data ports are all undriven and thus the tool will attempt all legal values and sequence of values. The rd and wr ports are constrained by the assumptions on lines 136-143. These assumptions highlight the fact that a formal tool can use assumptions to constrain nets. Usually, writing assumptions is much easier than creating similar verilog to perform the same function. Many of these assumptions can be reused as checkers when the module is integrated into a higher level testbench. The assumptions also work as documentation of the rules of the interface.

A formal tool can find problems that a random simulation cannot. A formal tool considers repeated sequences when checking liveness properties. These repeated sequences can find livelocks in a design. Repeated sequences are the antithesis of random simulation and, therefore, make it very difficult or impossible for simulation to find such problems. In addition, liveness properties in simulation will not fail until a simulation is complete. The actual cause of the problem may have occurred many cycles before the end of the simulation. On the other hand, formal tools will show the shortest sequence to the failing condition and also indicate the shortest looping sequence highlighting the problem. Lines 169-172 and lines 181-184 are examples of liveness properties.

The possibilities of scalability in the CIU design make it an ideal candidate for formal analysis. Also, the fact that there are some distinct stand-alone functions that the design performs makes those pieces of logic

ideal formal candidates. If sized as specified, the powerful end-to-end properties in each of the individual components of the design could not be proven. However, scaling allows for not only the end-to-end properties in the individual components to be proven, but also some of the end-to-end properties of the entire design.

## *5.3 Implementation Hardships*

Changing a designer's coding style is extremely challenging. A common difficulty when writing code with scalability in mind is the readability of the code being entered. When generalizing an algorithm, names are lost and replaced by numbers. Vector sizes are not known unless the parameter values given at time of instantiation are known. Another factor can be unfamiliarity with the coding style. The code looks more script-like with many loops, multidimensional arrays, temporary variables, and in general much more procedural code. There are less bit ranges, assign statements, and specialized code. While experience with the coding style can help to overcome some of the readability and understanding difficulties, the fact that the code is not a straight-forward implementation of the specification will always make the code slightly more confusing.

Developing generalized algorithms instead of simply implementing as specified results in longer code entry time which is probably the largest drawback of creating a scalable design. The main issue is that it can directly affect when the unit-level testbenches can start exercising the unit's functions and finding the most common bugs which are module-to-module communication issues and unit-wide algorithm problems. Again, experience will contribute to a quicker entry period, but even when a similar experience level is reached, the entry time will still be longer. The longer entry time must be built into the design schedule.

## 5.4 Verification Hardships

As discussed, a scalable design can do a lot to aid verification. However, there are some trade-offs that come with scalability when debugging a generalized design. When debugging in a generalized, reusable module, specific names are replaced by generalized code and numbers. This abstraction can potentially cause some misunderstandings.

A powerful debugging tool is a source browser that can annotate source code with values at a selected point in time. Annotated code in a procedural loop only shows the settled values of nets. However, typically when debugging it is interesting to know the values of the nets after each iteration of a loop. This problem can be seen in Figure 9. Notice how all the left hand side net values are displayed shown transitioning to their final values. Without stepping through the code, there is no way to see the values of the nets after any previous iterations of the loop.

```
95  // Read ports
              'd 2
96  integer rdi;
97  always @* begin
98    // Assign initial values
      'h 2 -> 'h 4
                'h 0 -> 'h 2
99    rd_addr=rd_addr_q;
100
101   // Main loop
                  'd 2
                    'd 2
                      'd 2
          'd 2                'd 2
102   for(rdi=0;rdi<RD_P;rdi=rdi+1) begin
103     // Empty based on current rd_addr and clocked wr_addr
        [index out of range]
                        'h 2 -> 'h 4
            'd 2                  'h 2 -> 'h 4
104     empty[rdi] = (rd_addr==wr_addr_q);
105
106     // Mem addr can only be as large as the depth of the FIFO
        'h 1 -> 'h 3
                    'h 2 -> 'h 4
                                'd 8
107     mem_rd_addr=rd_addr % D;
        {'h 0,'h 0}
                    'd 2
                          [8 x 2 bits]
                                'h 1 -> 'h 3
108     rd_data_md[rdi]=mem_q[mem_rd_addr];
109
110     // Wrap rd_addr when it reaches 2x the depth.
        'h 2 -> 'h 4
                    'h 2 -> 'h 4
                          'h 3
                      'd 2          'd 8
111     rd_addr=(rd_addr+rd[rdi]) % (2*D);
112   end
113 end
```

Figure 9

Although effective at finding obscure bugs, deep formal analysis at the module level required long run times and extensive debug efforts. The extra time designers spent performing formal analysis during module development in addition to the extra time spent creating a scalable design negatively impacted the other project goal of quickly developing initial unit-level designs to enable early looks at higher level verification and physical design problems.

# 6. Results

When all tallied, the CIU design had 19 logical buffers that were made scalable, all the transaction buffers and the retry buffer. Additionally, 7 of the 14 output transaction buffers could be entirely removed based on a parameter. The packet interface count on each of the major design interfaces could be changed as needed. The testbench models had similar buffer structures and interfaces and those models could be scaled as well. This flexibility allowed for more focused verification efforts. Smaller sized buffer structures were used when verification was targeting critical boundary condition logic and other potentially deep control algorithms. The actual specified sizes were used when tests targeted throughput and bandwidth functioning. Shrunken buffer structures were used when large system-level testbenches became slow or too large for 32-bit simulations. Also, shrinking designs to more effectively utilize a hardware accelerator was another advantage.

Physically, the flexibility of a scalable design allowed for easy integration of architectural changes that were made throughout the project. Including changing data buffer sizes to help meet fluctuating sizing requirements, updating interface widths to help alleviate routing congestion issues, and resizing structures as performance considerations were revisited. The design changes were made with confidence because verification had already been performed on scaled versions of the design making new implementation bugs unlikely.

The physical structure of logical memories could also be changed. A logical buffer that required several read and/or write ports had the choice of being implemented with any number of physical memories. This flexibility was important when vendor memory cells limited the number of ports available. The scalability also allowed for the easy exploration of the trade-offs of using vendor memory cells versus flip-flops.

Many of the stand-alone functions in the design had formal analysis applied to them. Without scalability some of the proofs achieved would not have been possible. Also, many of the bugs found may have never been found by simulation alone or at the very least it would have taken a long time to hit them. The advantages scalability brings to formal analysis can be shown by comparing some of the raw results from IFV on the FIFO module.

As a baseline, a very small version of the FIFO was used. The depth parameter (D) was set to 3, the width of the data (W) was set to 1, and the number of read and write ports (RD_P and WR_P) were set to 1. A powerful end-to-end property (assert_data_in_data_out) checks that data written into the module is eventually read out of the module. For the baseline design, IFV was able to prove this property true in well under 1 second of CPU time. However, Figure 10 shows some examples of increasing the depth of the FIFO, increasing the width of the data, and increasing the number of read and write ports. All of them, especially the depth, significantly increased the proof times. When combined, the exploding state space issue becomes clear.

| assert_data_in_data_out | | | |
|---|---|---|---|
| FIFO Parameters | Inputs | FFs | CPU Time |
| D= 3, W=1, RD_P=1, WR_P=1 | 5 | 9 | 0.27 |
| D= 8, W=1, RD_P=1, WR_P=1 | 5 | 16 | 2.63 |
| D= 9, W=1, RD_P=1, WR_P=1 | 5 | 19 | 452.7 |
| D=10, W=1, RD_P=1, WR_P=1 | 5 | 20 | 3987.66 |
| D=16, W=1, RD_P=1, WR_P=1 | 5 | 26 | 5497 |
| | | | |
| D= 3, W=1, RD_P=1, WR_P=1 | 5 | 9 | 0.27 |
| D= 3, W=1, RD_P=2, WR_P=2 | 8 | 9 | 0.68 |
| D= 3, W=1, RD_P=3, WR_P=3 | 11 | 9 | 19.63 |
| | | | |
| D= 3, W=1, RD_P=1, WR_P=1 | 5 | 9 | 0.27 |
| D= 3, W=4, RD_P=1, WR_P=1 | 8 | 18 | 1.24 |
| D= 3, W=16, RD_P=1, WR_P=1 | 20 | 54 | 12.26 |
| | | | |
| D= 3, W=1, RD_P=1, WR_P=1 | 5 | 9 | 0.27 |
| D= 8, W=1, RD_P=2, WR_P=2 | 8 | 16 | 258.44 |
| D= 8, W=2, RD_P=1, WR_P=1 | 6 | 24 | 881.16 |
| D= 8, W=2, RD_P=2, WR_P=2 | 10 | 24 | 12539.8 |

Figure 10

More typical depths for the CIU design are 32, 64, and 128. Many of the properties and sequences in the design were able to proven at the larger sizes; however, powerful end-to-end properties like the one shown in the example above commonly needed design scalability to become tractable for formal analysis. Figure 11 shows how some of the other properties and sequences were easily proven even at more typical parameter settings.

| FIFO Parameters | Inputs | FFs |
|---|---|---|
| D= 32, W=80, RD_P=2, WR_P=2 | 166 | 2572 |
| **Names** | **CPU Time** | |
| assert_fifo_empties | 0.13 | |
| assert_good_empty | 0.13 | |
| assert_good_full | 0.06 | |
| cover_all_rd | 0.04 | |
| cover_all_wr | 0.03 | |
| cover_empty | 0.03 | |
| cover_empty_full_emtpy | 42.45 | |
| cover_full | 2.34 | |

Figure 11

Not only were stand-alone functions attacked with formal analysis, but also entire unit-level testbenches. Many of the properties that were assumptions in the module-level testbenches either found bugs quickly or were easily proven. These properties had a very large return on investment; they were used to drive the module-level formal testbench and became checkers in the higher level unit testbenches. The unit level end-to-end properties were not easily proven even with a scaled design but effectively found bugs.

Scaled formal and simulation testbenches were able to hit most coverage sequences faster than when the testbenches used the actual design parameter sizes. However, scaling the design also caused some coverage to not be hit. These coverage failures highlighted the importance of applying a verification strategy that pairs coverage sequences together with properties in the verification plan. Figure 12 shows the IFV tool reporting that the coverage sequence detecting that all ports are writing at the same time cannot be reached when the depth parameter is set smaller than the number of ports. In this example D was set to 2 and WR_P was set to 3.

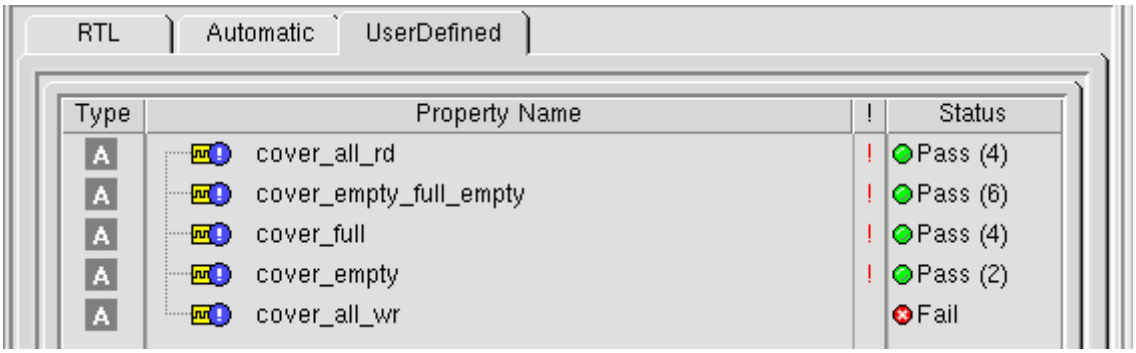


Figure 12

Project milestones, such as early unit-level module integration, were negatively affected by the early application of formal analysis to perform deep verification at the module level. Designers faced the

difficulties of applying a new implementation style to make their designs scalable and also the challenge of becoming proficient in applying formal analysis and using a formal tool. The general view was that formal technologies were very valuable to verification, but where, when, and how the technology is applied is critical to ensuring success.

Tool bugs emerged as another common theme when utilizing many of the language constructs important to a scalable design. Bugs were found in linters, compilers, decompilers, simulators, formal tools, synthesis tools, and equivalency checking tools, from several different companies. Some of the bugs still exist, most have been fixed, and more will likely be found.

# 7. Conclusions

Introducing scalability into a design's implementation and verification strategy has many potential advantages today and in the future. The past, current, and future designs discussed in this paper apply a combined implementation and verification strategy. Combining the strategies promotes collaboration between the implementers of the design and the verifiers. When introducing scalability, a combined strategy is necessary to keep all parties informed about the benefits sought and the difficulties encountered in the process. Like anything new, there exists a learning period where extended effort is required to overcome a slowdown in progress. However, once overcome, the full benefits of a scalable design can be realized. Designers will have many options to reduce the design size to better allow their testbenches and tools to exercise the most interesting functions of the design. The robustness required to adapt to fluctuating physical and architectural design requirements will become part of the design's infrastructure. Advanced verification techniques, including formal analysis, can have more widespread adoption enabling more confidence to be gained in the completeness of verification.

To compliment the advancing skill set of designers and the more complicated yet flexible designs they create, equally impressive advancements in tools, training, and methodologies must be developed. Extensive hands-on training demonstrating how to effectively use the language constructs important to scalability in common design structures is needed. Better training will enable designers to get past many of the early hardships incurred due to lack of experience. Tools or methods to improve the debugability of scalable designs are needed. Particularly, visibility into intermediate values of procedural loops would be helpful.

Design scalability is an enabler for widespread formal analysis. When the implementation and verification strategy calls for quick unit-level integration of a design's sub-modules to take advantage of all the significant advantages discussed, formal analysis has to be applied carefully. It is a contradiction to ask designers to quickly develop their modules and at the same time ask that they implement a scalable design, develop checker properties, enter coverage sequences, and fully verify their designs with formal analysis. However, if the formal analysis enablers (scalability, checker properties, and coverage sequences) are focused on initially, then the technology can be applied at any time.

Applying formal analysis later in a development cycle has some advantages. Not only can an initial unit be developed faster, but also simple, early design problems can be found in a simulator which is often better suited for finding such problems. One typical early design problem is undriven nets. Formal tools will drive undriven nets will all values whereas a simulator can propagate "x"s whose source can more easily be found. Additionally, designers can usually get a good initial feel for the shape of their logic by simply looking at a waveform of the design out of reset. Once the unit design is off the ground, i.e. the basic module-to-module communications problems worked out, simple typographical errors fixed, undriven nets resolved, etc., then more deep formal analysis can be applied. These facts indicate another

need; the ability to more easily transition back and forth between simulation and formal technologies. The ideal armament for a designer would be a tool that could apply simulation vectors to a design when dealing with early start-up problems. Then, without changes to the testbench, switch over to a formal engine when feeling more confident in the design, and finally, again perhaps switch back if the formal engines run out of steam.

A common project goal is to deliver a full featured product with minimal defects in the least amount of time. Any new strategies should attempt to help any or all of these goals. Adding scalability to a design's implementation and verification strategy has the potential to help these goals. However, without proper training, upfront thought, and tool support, it could also have the opposite affect. A design team needs to consider the challenges they are facing, the experience of their team, and potential advantages of scalability before attempting to employ the strategy into their design's development.

# Appendix A

```
 1  module fifo
 2  #(
 3  parameter
 4  WR_P=2,   // Number of write ports
 5  RD_P=2,   // Number of read ports
 6  D=3,      // Depth of FIFO
 7  W=1       // Width of FIFO data
 8  )
 9  (
10  input                     clk,      // Clock
11  input                     rst_l,    // Low-active reset
12  input [WR_P-1:0]          wr,       // Write indication for each port
13  input [(WR_P*W)-1:0]      wr_data,  // Write data for each port
14  input [RD_P-1:0]          rd,       // Read indication for each port
15  output reg [(RD_P*W)-1:0] rd_data,  // Read data for each port
16  output reg [WR_P-1:0]     full,     // Full indication for each write port
17  output reg [RD_P-1:0]     empty     // Empty indication for each read port
18  );
19  // ************************************************************************//
20  // A verilog shortcoming is the lack of multidimensional I/O ports.         //
21  // This code converts all flat input/output ports to/from multidimensioned  //
22  // nets to be used internally by the rest of the code.                      //
23  // ************************************************************************//
24  reg [W-1:0] wr_data_md [WR_P-1:0];
25  reg [W-1:0] rd_data_md [RD_P-1:0];
26  integer mdi;
27  always @* begin
28    for(mdi=0;mdi<WR_P;mdi=mdi+1) begin
29      // Index part-select references each data input
30      wr_data_md[mdi]=wr_data[(mdi*W)+:W];
31    end
32
33    for(mdi=0;mdi<RD_P;mdi=mdi+1) begin
34      rd_data[(mdi*W)+:W]=rd_data_md[mdi];
35    end
36  end
37  // *************************//
38  // End multi-dim conversion  //
39  // *************************//
40
41  localparam
42  ADDR_W=clogb2(D);  // Constant function used to calculate base 2 log.
43  reg [ADDR_W-1:0] mem_wr_addr,mem_rd_addr;
44  reg [W-1:0] mem_q [D-1:0];
45  reg [W-1:0] mem [D-1:0];
46  // Extra bit needed in these addresses to help distinguish
47  // full from empty.
48  reg [ADDR_W:0] wr_addr_q,wr_addr,rd_addr_q,rd_addr;
49
50  reg signed [ADDR_W:0] twos_diff;
51  reg [ADDR_W:0] mask,diff;
52
53  // Write ports
54  integer wri;
55  always @* begin
56    // ** Assign initial values
57    // mem and wr_addr are updated by the always block.  Their
58    // initial values are the previous state.
59    for(wri=0;wri<D;wri=wri+1) begin
60      mem[wri]= mem_q[wri];
61    end
62    wr_addr=wr_addr_q;
63    // ** End initial assignments
64
65    // ** Main loop
66    for(wri=0;wri<WR_P;wri=wri+1) begin
67      // Full based on current wr_addr and clocked rd_addr
68      // Absolute value of difference
69      twos_diff=(wr_addr-rd_addr_q);
70      mask= twos_diff >>> ADDR_W;
71      diff=(twos_diff ^ mask) - mask;
72      full[wri] = (diff==D);
```

```
73
74        // Mem addr can only be as large as the depth of the FIFO
75        mem_wr_addr=wr_addr % D;
76        mem[mem_wr_addr]= wr[wri] ? wr_data_md[wri] : mem[mem_wr_addr];
77
78        // Wrap wr_addr when it reaches 2x the depth.
79        wr_addr=(wr_addr+wr[wri]) % (2*D);
80      end
81      // ** End of main loop
82    end
83    // wr_addr state assignment
84    always @(posedge clk) begin
85      wr_addr_q <= ~rst_l ? 0 : wr_addr;
86    end
87    // mem state assignment
88    integer memi;
89    always @(posedge clk) begin
90      for(memi=0;memi<D;memi=memi+1) begin
91        mem_q[memi] <= mem[memi];
92      end
93    end
94
95    // Read ports
96    integer rdi;
97    always @* begin
98      // Assign initial values
99      rd_addr=rd_addr_q;
100
101     // Main loop
102     for(rdi=0;rdi<RD_P;rdi=rdi+1) begin
103       // Empty based on current rd_addr and clocked wr_addr
104       empty[rdi] = (rd_addr==wr_addr_q);
105
106       // Mem addr can only be as large as the depth of the FIFO
107       mem_rd_addr=rd_addr % D;
108       rd_data_md[rdi]=mem_q[mem_rd_addr];
109
110       // Wrap rd_addr when it reaches 2x the depth.
111       rd_addr=(rd_addr+rd[rdi]) % (2*D);
112     end
113   end
114   // rd_addr state assignment
115   always @(posedge clk) begin
116     rd_addr_q <= ~rst_l ? 0 : rd_addr;
117   end
118
119   // Constant function to calculate base 2 log.
120   function integer clogb2;
121   input [31:0] depth;
122   integer i;
123   begin
124     clogb2=1;
125     for (i=0;2**i<depth;i=i+1)
126       clogb2=i+1;
127   end
128   endfunction
129
130   // Verification code
131   `ifdef ABV_ON
132   // Interface assumptions.
133   // Will be used as assertions when module is instantiated.
134   // Constrains design for stand-alone verification with formal.
135
136   // Make sure only good reads and writes occur.
137   //psl assume_good_rd: assume never(rd & empty)@(posedge clk);
138   //psl assume_good_wr: assume never(wr & full)@(posedge clk);
139
140   // If the FIFO is not emtpy, a read should eventually occur.
141   //psl assume_ev_rd: assume always(
142   // {~(&empty)} |-> {[*];(|rd)}!
143   //)@(posedge clk);
144
145   `ifdef FA_ONLY_CHECKS
146   // The test_data net is undriven and therefore formal will try all
147   // legal values and sequences of values.  The stable_data constraint
148   // however limits the proof to only try all legal values because
```

```
149   // the data is constrained to be stable for any particular sequence.
150   wire [W-1:0] test_data;
151   //psl assume_stable_data: assume always(stable(test_data));
152
153   // Extra verification logic to detect when a write or read occurs of
154   // the test_data occurs.  Used in properties to follow.
155   reg wr_test_data,rd_test_data;
156   integer testi;
157   always @* begin
158     wr_test_data=0;
159     for(testi=0;testi<WR_P;testi=testi+1) begin
160       wr_test_data=wr_test_data | (wr[testi] & (wr_data_md[testi]==test_data));
161     end
162
163     rd_test_data=0;
164     for(testi=0;testi<RD_P;testi=testi+1) begin
165       rd_test_data=rd_test_data | (rd[testi] & (rd_data_md[testi]==test_data));
166     end
167   end
168
169   // Data that is written should eventually be read.
170   //psl assert_data_in_data_out: assert always(
171   //  {wr_test_data} |=> {[*];rd_test_data}!
172   //)@(posedge clk);
173
174   // Data should only be read if it has previously been written.
175   //psl endpoint wr_rd_test_data = {wr_test_data;[*];rd_test_data};
176   //psl assert_good_data_out: assert always(
177   // {rd_test_data} |-> {wr_rd_test_data}
178   //)@(posedge clk);
179
180   `endif
181   // If writes stop occuring, the FIFO should empty.
182   //psl assert_fifo_empties: assert always(
183   // {[*];(&empty)}! abort (|wr)
184   //)@(posedge clk);
185
186   // Since the full signal is based on the clocked rd address, if
187   // one port is considered full all the following ports should indicate
188   // full as well.
189   //psl assert_good_full: assert
190   // forall i in {0:WR_P-1}: forall j in {0:WR_P-1}: always(
191   //  {i<j & full[i]} |-> {full[j]}
192   //)@(posedge clk);
193
194   // Same as full
195   //psl assert_good_empty: assert
196   // forall i in {0:RD_P-1}: forall j in {0:RD_P-1}: always(
197   //  {i<j & empty[i]} |-> {empty[j]}
198   //)@(posedge clk);
199
200   // Coverage sequences to detect interesting functional conditions.
201   //psl cover_all_wr: cover {&wr} @(posedge clk);
202   //psl cover_all_rd: cover {&rd} @(posedge clk);
203   //psl cover_full: cover {&full} @(posedge clk);
204   //psl cover_empty: cover {&empty} @(posedge clk);
205   //psl cover_empty_full_empty: cover {
206   // (&empty);[*];(&full);[*];(&empty)
207   //} @(posedge clk);
208
209   `endif
210   endmodule
```

# Appendix B

```
$ ifv fifo.v +define+ABV_ON +define+FA_ONLY_CHECKS +tcl+run.tcl

run.tcl:

force rst_l 0
run 100
init -load -current
constraint -add -pin rst_l = 1
define effort 2 hr
prove
```