SAME 2000

Session: SoC

DESIGN AND VERIFICATION METHODOLOGY OF MODERN HIGH-SPEED SWITCHES

F. Abel
IBM Research, Zurich Research Laboratory
8803 Rüschlikon, Switzerland

Abstract:

Focussed research and continuous improvements in silicon technology have enabled high-performance switches and routers to keep pace with the growing rate of traffic on the Internet. A consequence of these improvements is that tremendous pressure is put on the design and validation of such communication systems. In addition to being cost-effective, scalable, and highly available, a new networking system also has to be on the market at the right time. This paper describes a multi-gigabit switch design methodology that has achieved manufacturing success from the first pass.

I. Introduction

It is apparent that traffic on the Internet grows, and shows no sign of slowing down. This constant and substantial increase in bandwidth demand has pushed high-speed networks to multi-gigabit data rates. Moreover, the transformation of the Internet into a basic and ubiquitous commercial infrastructure has not only created rising bandwidth demand, but also significantly changed consumer expectations in terms of reliability, security, and services [1]. This dual trend of rising demand on bandwidth and differentiated services has increased the complexity of Internet nodes where cells and packets are processed.

The design of VLSI technologies for broadband telecommunications has to accommodate these Internet changes. Integrated devices are growing in complexity, whereas the design cycle is shrinking. But not only system performance has to double every 18 months (Moore's Law), also products needed by the market have to be available on time. Given the average manufacturing time of 10 to 12 weeks for a high-speed ASIC (Application-Specific Integrated Circuit), there is little chance to be on the market just in time if the design does not achieve a first-pass manufacturing success.

This paper presents the VLSI design and validation methodology that we used to produce two releases of high-speed packet-routing switches based on the PRIZMA[1] architecture. In both cases, chips were delivered on time to manufacturing. Moreover we were able to complete both developments in almost the same amount of time, although the second switch chip provides twice the effective throughput, twice the number of ports, and has twice the complexity[2].

The remainder of this paper is organized as follows: Section II gives a brief overview of the Prizma switch [2,3] and its environment. Section III presents our design and optimization methodology based on a C++ behavior model. In Section IV the principles of validation are described. A so-called "gray box" verification approach is presented, including some combined efforts between the design, verification, and bring-up teams. Section V presents the simulation results and discusses directions for future research. Conclusions are drawn in Section VI.

II. Architecture Overview of Prizma-E and Prizma-EP

To understand the design and validation decisions of a high-speed packet switch, it is necessary to be familiar with a few details regarding the architecture and the features of such chips.

Prizma-E[3] and Prizma-EP[4] are second-generation devices of IBM packet-routing switches, and

[1] The name of PRIZMA (Packetized Routing in Zurich's Modular Architecture) refers to a technology developed at IBM's Zurich Research Laboratory in the past decade, and it is not to be considered an official product name, which is PRS (Packet Routing Switch).

[2] Because of added support for best-effort traffic policies and enhanced quality of service.

[3] We refer to Prizma-E for the IBM Packet Routing Switch PRS28.4G.

[4] We refer to Prizma-EP for the IBM Packet Routing Switch PRS64G.

were developed using the methodology described here. Both devices are built on Prizma's fixed-length packet, non-blocking switch technology, and follow the architecture of their lower-speed earlier version [2] to a large degree. Prizma-E and Prizma-EP are single-chip switch elements that exploit the performance advantage of the shared-output queuing structure [3], and from which larger, self-routing single-stage or multistage switch fabrics can be constructed in a modular way.

The Prizma-EP chip provides a switch fabric with 32 input and 32 output ports, each running at a throughput of 2 Gb/s. It is a self-routing module that integrates data buffering and control to achieve a 64-Gb/s aggregate throughput[5]. Two identical chips can operate in parallel (speed expansion) to increase the aggregate throughput to 128 Gb/s. The Prizma-EP switch is backward compatible with its predecessor, the Prizma-E switch, which provides only half the effective throughput and half the number of ports. We will henceforth only refer to the Prizma-EP design, which has all the features of the Prizma-E design plus some additional complexity as summarized in Table 1.

Table 1. Summary of switching chip features.

	Prizma-E	Prizma-EP
Effective Throughput (Gb/s)	28.4	64
Number of Ports	16	32
Logical Port Speed (Gb/s)	1.77/3.54	2/4/8/16/32
Clock Frequency (MHz)	111.1	250
Millions of Transistors	3.8	37
L_{eff} CMOS Process (µm)	0.25	0.18
Package	624 CCGA	1088 CCGA

II.A. The Packet Format

The packet length is configurable and can be 32 to 40 bytes (increment of 2), 64 to 80 bytes (increment of 4), 128 to 160 bytes (increment of 8) or 256 to 320 bytes (increment of 16). Each packet carries a configurable header of 2 to 5 bytes. The first byte of the header is the packet qualifier containing information on packet identification (data, idle), quality of service (priority, best effort), switch redundancy support (filter color) and parity of the header. The address of the packet destination is provided by a 8- to 32-bit bit map, carried by bytes 2 to 5 of the header. Each bit of the bit map is associated with a logical output port. The bit map field can point to multiple output ports (multicast).

[5]Some companies refer to aggregate bandwidth by summing bandwidth in and out of the switch. In that case, Prizma-EP is a 128-Gb/s switch.

II.B. The Switch Core Element

The architecture of a $N \times N$ Prizma element is characterized by the separation of the data section from the control section as shown in Fig. 1. Incoming packets are buffered in a shared memory, while the storage and retrieve addresses are being processed by the control section. The switch receives packets on N input ports and routes them to one or more of N output ports based on the bit-map information carried in the packet header. Quality of service support is provided through four levels of packet priority. The architecture supports flow control based on a grant mechanism at the input and output side.

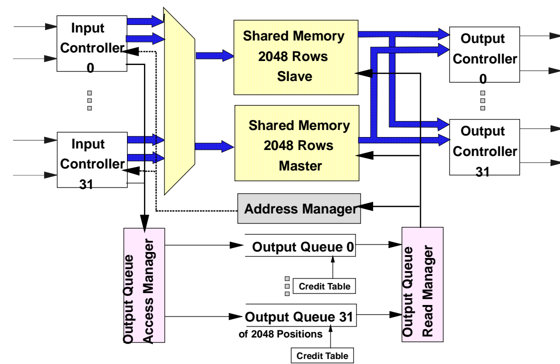

Figure 1. Prizma-EP block diagram.

II.C. The I/O Technology

To minimize the number of I/O pins, Prizma-EP uses 500-Mb/s, unencoded serial differential interfaces. Each switch port uses a hard macro composed of four receive and four transmit interfaces that provide full duplex data communication of 2 Gb/s.

On the receive side of the serial interface, each receive macro has to perform bit-phase, word and packet alignment. This process is handled by a central picoprocessor that shares its control algorithm among all the receive macros, a concept that greatly reduces the amount of silicon area otherwise required by dedicated phase-alignment macros.

To obtain phase alignment, a training sequence is required during the initialization phase. Once this is achieved, the control algorithm continues to monitor the serial interface to dynamically compensate for slow variations due to temperature changes. The dynamic adjustment directly operates on user data and does not require additional bandwidth.

Various protocol engines can be attached to the Prizma-EP switch. We will use the generic term of "*adapter*" when referring to attached devices based on protocols such as Packet Over Sonet (POS), Gigabit Ethernet, ATM or IP.

II.D. The Expansion Modes

Prizma-EP provides highly flexible cost and performance scalability by implementing a set of built-in expansions modes.

- Internal speed expansion combines two physical ports into one logical port at twice the speed. This halves the number of ports, and is an economical solution for building small switches with faster ports.
- External speed expansion can be used to operate two chips in parallel (in a master/slave mode), thus doubling the port speed without reducing the number of ports.
- Internal and external speed expansion can be combined to quadruple the port speed.
- Port expansion allows multiple devices to be interconnected in parallel, in a single-stage, so as to increase the number of physical ports, while keeping port speed constant.
- Link paralleling[6] can be used to group four physical ports into a logical one. This grouping is supported at the input side of the switch, the output side, or both. Unlike the other expansion modes, not all switch ports need to operate in link-paralleling mode at the same time (configurable by register). This feature is used to switch various Optical Carrier (OC) speeds within the same device.

Up to 512-Gb/s single-stage switches can be built by concurrently combining all of these expansions modes in any fashion.

II.E. The Serial Host Interface

Prizma-EP provides a serial host interface to a general-purpose microprocessor used to control and set up the switch configuration. About 200 status and configuration fields can be accessed through 63 registers of 32 bits.

III. Design Methodology

Complexity of this multi-gigabit switch design can be characterized by a short clock cycle in the range of 4 to 10 ns and by a high degree of parallelism between *N* inputs trying to access *N* outputs concurrently. From an implementation point of view, this parallelism translates into a very high coupling between many parts (chiplets) of the switch. In our case, this coupling is particularly high because of all the expansion modes supported. One negative effect of this coupling is that any minor change in a single chiplet may impact the behavior and performance of the entire design. The short clock cycle[7] increases complexity in the sense that only a small number of combinatorial operations can be performed between two latches. It also adds a sizable bottom-up constraint the design methodology has to cope with.

The starting point of the design process is a custom-built C++ performance model. The performance model is a high-level abstraction of the switch. It is intended to verify the concepts and validate the algorithms at a system level. We used it to investigate the performance of multistage and Combined Input- and Output-Queued (CIOQ) architectures using Virtual Output Queueing (VOQ) [4], and to define the main characteristics of our output queuing switch, such as number of ports, size of the shared memory, number of priorities, flow control mechanisms, and scheduling algorithms.

Considering the size and the complexity of such a project, system design usually starts with a high-level behavioral model implemented in a traditional Hardware Description Language (HDL). Although these languages are highly geared for hardware modeling of digital systems at many levels of abstraction, ranging from the algorithmic to the gate level, they are often verbose and very slow to simulate. On the other hand, high design productivity and fast simulation—which are crucial when designers are particularly concerned with complexity issues—can be achieved by using the standard language constructs of a software-programming language to model hardware [5]–[7]. Architecture and design exploration may lead to many different implementations and solutions that have to be rapidly and intensively simulated at all conception steps, particularly after any modification that has to be validated. Our design methodology uses this approach by accurately modeling the hardware behavior of our switch in C++.

There are pitfalls, however, in following a pure software-programming-language description to model hardware. Once the concepts and algorithms are validated on the C++ model, the design has to be translated manually into a HDL for actual hardware implementation. The same tedious and error-prone process also applies to the testbenches created for the C++ model validation as they typically cannot be run against the HDL model without conversion. After the model has been converted to HDL, the HDL model becomes the focus of the development. The C++ model quickly becomes outdated as changes are made, and changes are typically made only in the HDL model, which slowly disconnects from the system model.

To circumvent the design drawbacks of such a software-based methodology while avoiding these traps, the following options were taken:

[6] This expansion mode is only supported by Prizma-EP.

[7] 4 ns for Prizma-EP.

- To model and describe a hardware abstraction of the switch, the behavior model builds on a subset of C++ and implements a particular set of constructs together with a specific semantic close to a HDL. The aim of this similarity is to avoid manual translation into a VHDL behavioral description by using an automated translation process.
- To avoid multiple system validations, we set up a unified environment (as described in Section IV) that was used at all steps of the design process, from system-level validation down to gate level.
- To avoid overhead in maintaining multiple models, the C++ model was refined to a clock-accurate level of detail and was used as reference model until very late in the design process. Simulation of the VHDL model only started at the chip-finishing phase when clock trees were generated and the JTAG inserted.

III.A. C++ Behavior Model

The behavior model implements all abstraction mechanisms needed for building a large system out of smaller modeled components. Design is described in a netlist of blocks, in which component instances are bound by shared variables. To represent an external view of a component that is independent of its implementation, interface specification is separated from the component's body specification. Such a feature is useful for iteratively refining the hardware model via successive implementations, without altering the external view of the component [5]. Figure 2 shows a declaration and definition example of part of an input controller. Figure 2a shows the declaration of the input controller entity with the list of interface ports through which the entity communicates. The port types can be a predefined data type of the C++ language or any new data abstraction defined by the designer. Figure 2b shows the body of the input controller. The specification of the architecture is always based on a VHDL behavioral style of modeling. The `ClockTick()` function is called every clock cycle and can be compared to a VHDL process declaration that has no sensitivity list but an explicit "wait" statement in its process statement (`wait until clock'event and clock='1'`). Reading and writing of signals are performed by the **`sread()`** and the **`swrite()`** functions. Note that the code looks like a VHDL program but is actually entirely C++ code. The entire C++ semantic is hidden by inline functions and macros (shown in bold) that render the program easy to read and use by a VHDL designer.

Although the C++ behavior model implemented all the semantic and abstraction mechanisms to perform behavioral synthesis, this approach was ultimately abandoned. One of the main reasons was that we focus on the design of a multi-gigabit switch rather than on an Electronic Design Automation (EDA) tool. Moreover the semantic and structural design of the C++ behavior was so close to a VHDL description that a reliable and direct translation could be manually performed in a short time.

(a)

```
entity(InpCtrl)
  // Itf with the Application Registers (ARG)
  input      ( boolean, ARG_StandbyReset );
  input      ( boolean, ARG_MasterPort );
  input      ( boolean, ARG_ControllerEnable );
  inputArray ( boolean, ARG_OQueueEnable, 32 );
  inputArray ( boolean, ARG_BMFilterMask, 32 );
  ...
   // Itf with Input Frame Processing (IFRAM)
  input      ( boolean, IFRAM_DiscardThisCell );
  input      ( int,     IFRAM_RXCountIn );
  input      ( int,     IFRAM_RowCountIn );
  ...
  // Itf with the On Chip Monitor Registers (ORG)
  output     ( boolean, ICTRL_NoAddrError );
  ...
  // Itf with Packet Memory (PktMem)
  output     ( boolean, ICTRL_InsAddr );
  ...
  // Itf with On Chip Monitor (OCM)
  //   For beh model only. Used to gather some stat.
  input      ( p_oGlobalChipData,  ARG_GlobData );
};
```

(b)

```
architecture(default, InpCtrl)
  ...
  boolean      Receiving;
  boolean      IdleCell;
  boolean      DataCell;
  boolean      ControlCell;
  byte         Qualifier;
  boolean      ASAInvalid;
  uLong        BMHeader;
  ...
  void ClockTick(int itfnr) {
    ...
    int   ByteCounter;
    int   RowCounter;
    ...
    if (sread(ARG_StandbyReset) == TRUE) {
      // Reset Sequence
      ...
      return;
    }
    ...
    ByteCounter = sread(IFRAM_RXCountIn);
    RowCounter  = sread(IFRAM_RowCountIn);
    ...
    if ((RowCounter == 0) && sread(ARG_MasterPort)) {
      ...
      if (ByteCounter == 7) {
        Receiving = FALSE;
        if (sread(IFRAM_DiscardThisCell) == TRUE) {
          Trace("Received cell is discarded due to header\
              parity error (ByteCounter=%d)", ByteCounter);
          IdleCell    = FALSE;
          ControlCell = FALSE;
          DataCell    = FALSE;
        }
        else if (Qualifier != 0xCC) {
          if (ASAInvalid == TRUE) {
            if (DataCell == TRUE) {
              swrite(ICTRL_NoAddrError) = TRUE;
              Error("No ASA Address available!");
            }
          }
          else if ((DataCell && (BMHeader != 0)) ||
                     (ControlCell == TRUE)) {
            if (sread(ARG_MasterPort) && ControlCell)
              swrite(ICTRL_InsAddr) = TRUE;
            Receiving =  sread(ARG_ControllerEnable);
          }
        }
      }
      ...
      // Behavior model only: perform some statistics
      //---------------------------------------------
      if ((ByteCounter == 7) && (Receiving == TRUE)) {
        sread(ARG_GlobData)->TotNrOfPacketsReceived++;
        sread(ARG_GlobData)->TimeStamps[ASA|RowCounter]
          g_SEClockCycle - ByteCounter;
       }
      ...
    }
    ...
  }
end_architecture(default, InpCtrl);
select_architecture(default, InpCtrl);
```

Figure 2. Input controller process: (a) entity declaration and (b) body of the process.

III.B. Logical Design Flow

As gigabit switch design is particularly constrained by timing requirements, it cannot be handled at a C++ level of description without ensuring that the timing specifications can be met. Our early timing analysis approach was to perform synthesis at VHDL chiplet[8] level only. Because of the chip size, we broke up the C++ design into a series of VHDL-like chiplets corresponding to the main functional units of the design. This split and the interfaces between these units were defined by the VHDL designers, and it was agreed that all chiplet outputs should be latched. The well-defined interfaces and large independence of the chiplets were desirable because different chiplets were assigned to different VHDL designers.

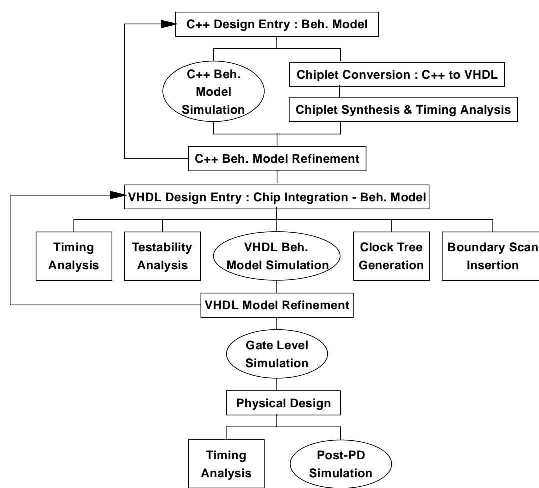

Figure 3. Design flow.

Figure 3 shows a schematic view of this part of the methodology. The design steps are shown in the sequence in which they are applied with mutually independent steps displayed in parallel. Note that no simulation was performed at the VHDL chiplet level because of the one-to-one translation from the C++ chiplet models to the VHDL chiplet models. Inserting the VHDL model into the validation environment only occurred after all chiplets have been connected into a top-level structure, i.e., near the end of the design phase. Unsatisfactory results in VHDL chiplet synthesis could lead to architectural changes which were fed back into the C++ behavior model for refinement. Efficiency and accuracy of the C++ to VHDL translation is a crucial point in this methodology.

IV. Validation Principles

In Section III we focussed on the design flow used to translate a design specification into design solutions and implementations. Here we discuss the validation principles used to evaluate and verify that the implementation achieves the required functions.

Our design verification methodology is based on simulation. All output signals of the Prizma chip model are checked according to various sets of stimuli applied to all input signals. Output signal in this context are the Prizma cells from the transmit part (including in-band flow control), all outgoing signals on corresponding control lines, and all values that can be looked up by the general-purpose microprocessor controlling the chip. Input signals in this context are all Prizma cells of the receive part, all incoming signals on corresponding control lines, and all values that can be written into configuration registers.

Simulation is not a completely reliable method of demonstrating correctness[9], but has the advantage of providing accurate information about the actual behavior of a design, with a reasonable effort. To get close to the mathematical level of accuracy and validity that proofs[10] or chip emulation deliver, a simulation approach has to provide a very large number of high-quality checks.

With C++ being our reference model used to validate the design, replication in writing or converting testcases[11] from one environment to another has to be avoided. Therefore we set as a prerequisite that any testcase written for the behavior model must be reused at all levels of the circuit hierarchy (see Fig. 4 and [8]).

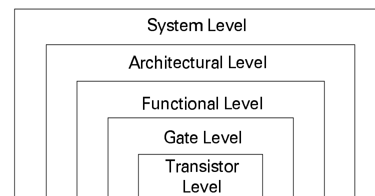

Figure 4. Levels of circuit hierarchy.

Another aspect usually overlooked in the design phase is the production and bring-up tests, which are used to verify whether the manufactured copies of a circuit conform to the design. To minimize the overall verification effort we made sure that a large percentage of the validation programs could also be reused during hardware bring-up. This feature is further discussed in Subsection IV.C.

[8] A chiplet is a black box bound by latches.

[9] Simulation can be used to show the presence of bugs, but never to demonstrate their absence.

[10] Formal verification approaches were not considered because of complexity issues and resource requirements. These methods are mostly restricted to small or mid-sized designs.

[11] Often called testbench.

IV.A. Testcases

To detect errors, testcases are written in C++ and in a subset of commands especially required for creating packets as well as other in-going signals sent to the switch. There are several categories into which testcases can be classified. Major types within this classification include [9]:

- Requirement-based testcases: Derived from the functional specification of the packet switch. They are independent of the design logic and are based on a "black box" formulation. Design goals at the input of the black box are transformed into design results at the output side.
- Design-based testcases: Derived from the logical packet-switching system structure, they depend on the design logic (white box).
- Randomized testcases: Derived from a randomizing technique, they are intended to catch errors that are not caught with the above testcases.

To cover all features in a packet-switching system design as well as all functions defined in the specification, a combination of these three testcase-type strategies is needed. In this way we created a so-called "grey box" simulation approach that allows the requirements as well as the system design to be validated.

Before we further discuss this testcase partitioning between white- and black-box simulation, we first have to introduce the validation environment.

IV.B. Validation Environment

Because system verification is one of the most important design activities (more than 70% of development time [10]), the validation environment was designed with the objective of simulating at all circuit hierarchy levels. The main benefit is that only one validation environment needs to be developed, maintained and debugged. In our particular case, 90% of the testcases can be run at all the chip levels.

Most of the validation is based on sending entire packets through the switch. As shown in Fig. 5, activities of the validation environment are split into three parts, which are executed in a sequential order: **Generation → Switching → Checking**.

The advantage of this architecture is to keep the complexity of the environment low in comparison with that of reflecting architectures where generation can react to the switching phase. The disadvantage is that special occurrences during switching or checking can no longer influence the generation activity. All intentions have to be covered before going to the switching part.

Generation takes a testcase program as input and translates it into input stimulus to the switch. The

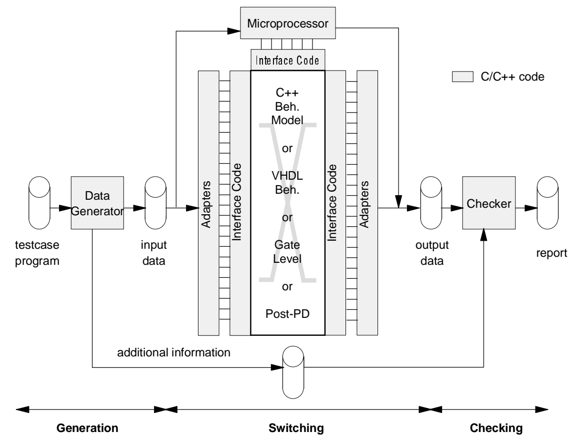

Figure 5. Validation environment.

testcase specifies the switch configuration, the types and times of packets to be sent from input *X* to output *Y*, the setting of specific input control signals such as transmission grants, and asks for some specific extended checks to be performed (see checking).

Switching is the phase where simulation of the switch takes place. Input files are read by the input adapters and translated into input stimuli to the switch. Next, packets are switched and time stamped prior to being written to files by the output adapters. From a validation-environment point of view, the model used (C++, VHDL/RTL, Gate Level, Post-PD) is fully transparent. An interface code is used to adapt between the C/C++ and the VHDL simulator.

Checking is the last step performed. The checker has some built-in fixed rules that it combines with the output data and additional information provided by the data generator to automatically find errors in the switching phase. Two kind of checks are performed:

- Basic checks are fundamental rules that must always hold when a packet comes out. Parameters checked are corruption (packet length and content), routing (did the packet arrive at the right output, packet sequence, duplication and loss, allowed packet type) and some basic timing between the outputs (sequencing, speed expansion, link paralleling).
- Extended checks are performed on demand and must be explicitly specified in the testcase. These checks are used to verify that a simulation result occurred as expected by the testcase scenario. Extended checks derive from formal verification methods and are used to verify a variety of temporal assertions, such as response time violation, fairness and lifeness violations.

IV.C. White-Box Simulation

The aim of white-box simulation is to check the individual functions and features defined in the specification. It takes the details of the switch implementation into account and attempts to validate each part of the switch separately. Effectiveness and error coverage are the objectives pursued by this kind of validation. Therefore one or more short testcase programs are developed for each functionality to be validated. Small testcases are faster to simulate, and errors are easier to locate. Dedicating one or more testcases to a particular item also reduces program complexity compared to "bigger[12]" testcases, which consider more scenarios and which might overlook some aspects more easily. Another reason for splitting testcases into several parts is compatibility with the bring-up tests. The white-box testcases are executed on a simplified validation environment that is the exact replica (model) of the bring-up environment used to check the manufactured chips. Therefore, all testcases written for the white-box approach can all be reused (at no design cost) to check all individual functions of the real silicon.

Figure 6 shows the simplified validation environment used for white-box simulation. Outputs

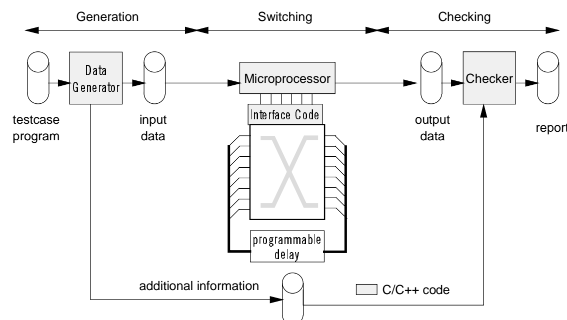

Figure 6. Loop-back environment.

of the switch are directly connected to the inputs in a loopback mode, whereas all traffic is generated and collected by the general-purpose processor by means of control packets.

IV.D. Black-Box Simulation

Black-box simulation takes an opposite approach to white-box simulation. The objective is to avoid considering any internal details of the switch as much as possible[13]. This approach leads to testcase implementations that are highly independent of the internal details of the switch and hence also much more stable against changes of these details. Because black-box simulation performs design validation at a system level of abstraction, it is particularly suitable for the design exploration phase and algorithm validation. The disadvantage of validating at this level of abstraction is that it is difficult to locate the cause of an error. However, to verify the behavior of a switching system with black-box simulation, fault detection has a higher priority than fault localization.

Another object of black-box simulation is to catch the corner-case errors. Experience shows that 90% of the remaining bugs are due to corner cases that escaped detection during simulation. To catch these corner cases, black-box simulation is also used to simulate the switch behavior while combining multiple basic functions. This mode of validation only starts when the design stabilizes and is reasonably covered by white-box simulation. To further increase the test coverage of corner cases, black-box simulation relies on randomized traffic patterns sent through the switch. In fact, each testcase models a random traffic generator specifically shaped for a scenario to be validated.

V. Results

The Prizma-E and Prizma-EP designs have been validated using the environments depicted in Figs. 5 and 6. As a result of extensive simulation throughout the design cycle, the Prizma-E ASIC was correct the first time. At the time of writing, the Prizma-EP is still under test in the lab, and we anticipate a first-pass success.

The entire validation environment consists of 165,000 lines of C/C++ code and of 5,000 lines of Korn Shell scripts. On average, in the course of each project we had only one architect and one software engineer dedicated to its development. The architect was in charge of the behavior models of switch, adapters and microprocessor, whereas the software engineer took care of the packet generator and the checker. Testcases were handed to several team members so as to obtain various interpretations of the specifications.

The simulation is based on a master set of about 100 testcases: 40 white-box and 60 black-box testcases. Each testcase is configurable and is executed several times using different parameters specified by a configuration file. A full regression to check all features and configurations of the switch consist of a total of 5,000 testcase runs.

To increase the test coverage by exercising the random behavior of each testcase, a regression needs to be run multiple times. We used

[12]Note that the term "big" testcase here does not reflect to the number of lines in the testcase but rather that several items to be tested are combined in one testcase.

[13]A complete black-box approach is not always possible, and some testcases still may have to deal with some timing dependencies.

Table 2. Simulation time (in seconds) and speed (in clock cycles/second)

		C++ Beh. Model	VHDL/RTL	Gate Level
tc_OnePck	time	10	181	22,007
	speed	371	20.5	0.17
tc_connect	time	43	773	35,747
	speed	277	15.5	0.33

Table 3. Size of the simulation process (in MB)

	C++ Beh. Model	VHDL/RTL	Gate Level
Process Size without Speed Expansion	6.8	150	650
Process Size with Speed Expansion	10	200	913

"Loadleveler[14]" [11], a job management system to distribute these 5,000 testcases over a network of RISC System 6000 workstations. While running against the C++ behavior model and using a pool of about 50 machines, a full regression could be run every night (in about 12 hours). The results in Table 2 compare the simulation time at three different levels of the circuit hierarchy. Simulation time is provided for two types of testcases executed in the system depicted in Fig. 5 and without speed expansion.

- "tc_OnePck" is a very basic testcase that sends a single packet from a randomly selected input to a randomly selected output. About 80% of the simulation time is spent for initialization of the switch and the serial interfaces.
- "tc_connect" checks connectivity between input and output ports by sending a packet of each priority from all inputs to all outputs. For a 32×32 switch, a total of 4096 packets are generated by this testcase.

The simulation has been performed on a RS/6000 Model 44P-270 with one 375-MHz processor and 2 GB of system memory.

Table 3 compares the size of the Unix process for a behavioral C++, a VHDL/RTL and a gate level simulation. Numbers are provided for the instantiation of one switch system (no speed expansion) and two switch systems (with speed expansion). We used the ModelSim EE/PLUS simulator from Modeltech Technology Incorporated [12].

A C++ behavior model not only provides significant time improvement over a VHDL simulation, approx. 18 times faster in our case, it also runs on smaller workstations. Also, because of the greater memory space required when running a VHDL simulation, a smaller number of machines is available. Depending on the number of high-performance workstations available, the overall simulation time for a full regression might be another order of magnitude longer.

Because of the high quality of the C++ to VHDL translation, very few errors were found as design progressed down to gate level. Therefore the VHDL simulation mainly focussed on testcases with highly randomized traffic generators.

The plug-in of the VHDL behavior model into the validation environment was fast and painless. As the microprocessor and all the testcases and adapters were fully debugged using the C++ behavior model, VHDL integration was reduced to a connection process together with few timing check adaptations.

Another successful strategy was to perform pin-out ASIC simulation with randomized testcases. Pin-out simulation means that the switch model was combined with the behavior models of adapters and microprocessors to create highly realistic stimuli similar to the final board-level connecting circuitry. Both techniques enabled unexpected behavior to be detected and corrected very early in the design.

The next switch generation targets to implement scalability over multi-terabits of aggregate band-width. As clock speed cannot be increased by the factor of ten needed, new designs will have to take advantage of parallel architectures and wider data paths. Investigation of such new architectures can no longer be done without integrating power dissipation and die-size utilization, which become extremely constraining at the speeds considered. As a consequence we will continue to use this success-ful methodology, but we will make it more effective to explore new architectures by including these physical parameters in the design phase. With C++ remaining our language of choice for architectural specification, future research will have to deliver hardware synthesis directly from C++ . Providing a fast and automated path into hardware implementation from C++ will enable bottom-up

[14] Loadleveler is a job management system that allows users to run more jobs in less time by matching their processing needs to available resources. As part of its job management system, Loadleveler serves as a job scheduler and provides a facility for building, submitting, and processing jobs quickly and efficiently in a dynamic environment.

feedbacks to be incorporated very early in the architectural design phase.

A similar design methodology was launched by the Open System C Initiative (OSCI) during the end phase of our project. SystemC [13] is a standard modeling platform consisting of C++ class libraries for design at a system-behavioral level, and can interoperate with EDA tools for synthesis into a gate-level netlist or into a synthesizable VHDL or Verilog RTL. We are currently investigating SystemC, and consider using it for our next design.

VI. Conclusions

We have presented our environment for hardware design and verification in C++. A design flow based on this approach offers several advantages.

The C++ language can be very efficiently compiled, debugged and executed on today's work-stations, enabling the development of very fast models for functional verification. Saving time during functional verification provides significant reduction of the design cycle, and leaves the designer with more time for architectural investigations and optimizations.

Our environment uses the same language and modeling paradigm for both simulation and creation of models. Therefore, the interaction between the two parts is greatly simplified, and the validation environment becomes faster to develop and execute. Moreover, at the time the hardware implementation starts, this environment is ready and has been intensively debugged. The designer can then concentrate on synthesis and real error corrections.

Finally, while combined with OO techniques, C++ allows one to design at a high level of abstraction. This high level of abstraction results in a code that can easily be changed, maintained and reused from one generation of switch to the next.

Acknowledgments

The following people have been instrumental in the success of the project, therefore the author extends a special thanks to O. Bocquillon, B. Brezzo, A. Bussani, M. Colmant, F. Gramsamer, M. Gusat, M. Heddes, R. Luijten, C. Minkenberg, M. Sing, and D. Wind.

References

[1] V.P Kumar, T.V. Lakshman, and D. Stiliadis, "Beyond best effort: router architectures for the differentiated services of tomorrow's Internet," *IEEE Commun. Mag*., 36(5), 152-164, May, 1998.

[2] H. Ahmadi, W.E. Denzel, C.A. Murphy, and E. Port, "A high-performance switch fabric for integrated circuit and packet switching," *Int'l J. Digital & Analog Cabled Systems*, 2(4), 227-287 (1989).

[3] W.E. Denzel, A.P.J. Engbersen, I. Iliadis, and G. Karlson, "A Highly Modular Packet Switch for Gb/s Rates," *Proc. 14th Int'l Switching Symp. "ISS '92,"* Yokohama, Japan, October 1992, pp. 237-240.

[4] C. Minkenberg, T. Engbersen, and M. Colmant, "A robust switch architecture for bursty traffic," *Proc. 2000 Int'l Zurich Seminar on Broadband Communications*, (IEEE, Piscataway, 2000) pp. 207-214.

[5] R.K. Gupta and S.Y. Liao, "Using a programming language for digital system design," *IEEE Design & Test of Computers*, 14(2), 72-80 (April-June 1997).

[6] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int'l J. Computer Simulation*, special issue on Simulation Software Development, 4(155) 155-182 (April 1994).

[7] J.S. Young, J. MacDonald, M. Shilman, A. Tabbara, P.Hilfinger, and A.R. Newton, "Design and specification of embedded systems in Java using successive, formal refinement", Dept. of EECS, Univ. Calif. At Berkeley, Design Automation Conference, 1998.

[8] S. Davidson, "Software tools for hardware tests," *IEEE Computer*, 22(4), 12-14 (1989).

[9] W.C. Hetzel, *The Complete Guide To Software Testing*, 2nd Ed. (QED Information Sciences, Inc., 1988).

[10] T. Kropf, *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems* (Springer, 1999).

[11] Loadleveler - http://www.rs6000.ibm.com/software/sp_products/loadlev.html

[12] ModelSim EE/PLUS - http://www.model.com/products/msvhdl.html

[13] SystemC - http://www.systemc.org