A Methodology For the Verification of a “System on Chip”

Daniel Geist, Giora Biran, Tamara Arons, Michael Slavkin, Yvgeny Nustov, Monica Farkas, Karen Holtz
IBM Haifa Research Lab
MATAM Advanced Technology Center
Haifa, Israel
+972 4 8296286
dannyg@vnet.ibm.com

Andy Long, Dave King, Steve Barret
IBM Field Design Center
Essex Junction, VT
U.S.A.

1. ABSTRACT

This paper summarizes the verification effort of a complex ASIC designated to be an “all in one” ISDN network router. This ASIC is unique because it actually consists of many independent components, called “cores” (including the processor). The integration of these components onto one chip results in an ISOC (Integrated System On a Chip). The complexity of verifying an ISOC is virtually impossible without a proper methodology. This paper presents the methodology developed for verifying the router. In particular, the verification method as well as the tools that were built to execute this method are presented. Finally, a summary of the verification results is given.

1.1 Keywords

Systems on chip,verification, test and debugging.

2. INTRODUCTION

The rapid progress of chip fabrication technology has given rise to new hardware solutions consisting of chips that contain whole systems. On the one hand, the fabrication technology makes it now possible to fit several chips, that were previously fabricated stand-alone, onto a single silicon chip. On the other hand, the design of complicated chips has not advanced as rapidly. That is, efficient utilization of such fabrication ability can only be truly exploited in very high performance projects such as state of the art central processor units. However, even without a large design team, the simplest way to exploit the ability to fabricate a bigger chip is to combine many existing chip designs into a single ISOC (Integrated System On a Chip). The design effort is not much greater than the combined effort required for a set of stand-alone chips. The designs are now called “cores” instead of chips - meaning that they do not constitute an end product by themselves. Thus, the integration of the system units which was previously done externally on several fabricated units is now performed inside the ISOC. The payoff is immediate, the fabricated ISOC will be much smaller, faster and cheaper than an equivalent product built from several components.

The design of an ISOC is not a large step from the design of the components that compose it. However, the impact on verification is significantly large. ISOC type designs are not common yet. Our particular ISOC verification problem was considered challenging because of the lack of prior experience and existing solutions. There was only one previous attempt to build an ISOC in IBM that we knew of and it was not finished yet. We did have some experience with the verification of systems but they were all built from different components. Our approach was to rely on this experience. The end result, based on the invested effort and the verification quality, was a good reasonable solution. The first thing to be understood was the difference of this chip from past ones.

2.1 Why is an ISOC Different From an ASIC?

An ISOC is first and for all a single chip but unlike an ASIC (Application Specific Integrated Circuit), its specification is not confined to its external behavior, but also to internal communication protocols among the chip blocks. The internal communication protocols are methodically defined and these definitions are rigid. The designers of these blocks have to conform to the protocols and they can not expect a change in them to make their designs easier (in ASICs, when a problem arises with internal interfaces - you can modify them). In fact, having such well defined boundaries inside the chip makes the overall design effort much simpler (e.g. you can not create complex logic that spans over many blocks). Defined interfaces also allow the easy migration of cores from one ISOC to another.

2.2 Why is an ISOC Different From a System?

The difference between an ISOC and a regular system is clearly stated in its mnemonic: it is integrated. This means that problems in the end system require a fix to the whole product. Problems between components cannot be patched by adding glue logic as done in externally integrated systems. If a design problem is discovered in an ISOC component or in its integration, the only solution is to re-manufacture the whole chip. This increases the importance of the system verification effort and the cost of fixing a bug substantially rises. Therefore, the complexity impact of an ISOC is first and foremost on the verification task. The advantages of ISOCs (as system solutions) are in other areas. For example the synthesis and pin distribution is done for a single chip, instead of for several components.

This paper describes the verification effort of an ISOC consisting of a PowerPC [6] processor based system, consisting of a CPU, two busses (a CPU bus and an IO bus), a memory controller, a sophisticated DMA controller, several communication controllers on the IO bus, and several other small components such as an interrupt controller, bus arbiters, etc., all fabricated on one chip.

Initially there was no solution for the verification problem. The methodology and tools to support it were formed hand in hand with the chip itself. Our experience with this methodology has shown this approach to be of high value in the verification of ISOCs. We were able to cover a good portion of the system functionality with limited resources and time. The approach taken here was to examine how systems are verified and to adapt an existing system verification scheme to the verification of the ISOC. In most part we found that system verification and ISOC verification are similar but that the relative effort invested in system verification compared to the verification of the units is much greater for an ISOC than for a non-integrated system. There were also a few more requirements (such as verification of different system configurations).

The rest of this paper is organized as follows. Section 3describes the chip. In Section 4, we describe the methodology. Section 5 describes the various tools that were developed to support this

DAC 99, New Orleans, Louisiana

methodology. Section 6 contains results from our experience. A summary and conclusions are presented in Section 7.

3. THE NETWORK ROUTER

Monkton, the network router is depicted in Figure 1. Monkton contains an embedded 32-bit PowerPC 401 processor. The Processor Local Bus (PLB) [9] is connected to a memory controller and a programmable packet DMA controller (MAL) for transferring data back and forth from various communication channels that are connected to the I/O bus. The PLB comprises separate 32-bit read and write data buses, and is architected for pipelined, burst and bus-locked transfers to maximize bus utilization. On the PLB is a bridge to the on-chip peripheral bus (OPB) [8], which implements many of the same features as the PLB, but employs a single 32-bit read/write data bus to handle the lower bandwidth requirements of peripheral devices such as serial and parallel interfaces: CODECs and networking communication macros (ComMacs). The bridge gives the processor access to the OPB and is used for programming the channel ComMacs, etc. Each ComMac handles a different communication Protocol: the EMAC [7] core connects to an Ethernet line, the HDLC core connects to an ISDN line and can communicate over 4 different channels in either SCC or TDM mode, the UART cores connect to two RS-232 ports. The SCP is a simple 1 byte transfer protocol for accessing and programming local peripheral hardware.

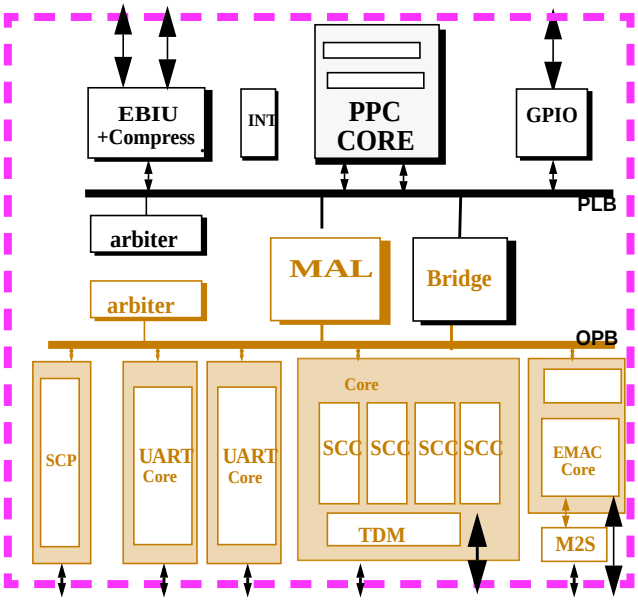

Figure 1. The Router

Perhaps the single most important architectural feature of this chip is the DMA controller or Memory Access Layer (MAL) [3], which transfers data between system memory (connected to the PLB via the EBIU) and the ComMacs. Since the OPB Bridge is a slave on the PLB, OPB devices must use an interrupt mechanism to gain access to system memory. In order to balance the interrupt latency with the amount of FIFO buffering required at each communications port, the Memory Access Layer (MAL) combines the function of interrupt service with a DMA controller. The MAL maintains buffer descriptor structures in system memory for each communication transmit and receive channel. When required, the MAL requests access to the PLB to effect DMA transfer of data from a protocol handler FIFO to system memory via the OPB Bridge and EBIU. The ISOC contains other small blocks: an interrupt controller, general purpose I/O ports and some glue logic. All these reside on a single chip.

The specification of this ISOC was not limited to the external pin behavior: the PowerPC architecture was also specified, both internal busses were specified, all the internal controllers (memory, DMA, interrupt) were specified, the DMA programming model was specified as well as the DMA communication protocol with the ComMacs on sideband signals.

For verification, Monkton served as quite a challenge. It was not a single unit, but rather a complex system of units connected to each other via different bus or channel protocols. It required the development of a different verification methodology, detailed in the following section.

4. METHODOLOGY

Our solution to the ISOC verification problem was to use a random biased test generator - SysGen (Section 5.2). IBM has been using this technology for several years and in particular, SysGen has been used to verify a variety of IBM server systems. The previous ASIC verification methodology was to build an extensive reference model of the verified ASIC that would check on-line for possible errors during simulation. This solution was recognized as too expensive in the case of an ISOC. Since there were similarities between current systems verified with SysGen and the ISOC, the verification team decided on converting the methodology used in IBM server systems verification.

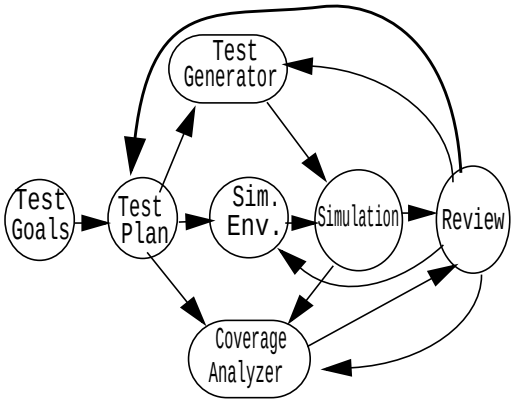

Figure 2. Verification Methodology

The solution was based on the above principles:

- Covering mainly system-wide functionality and directing testing only to what is likely to be used in actual software.
- A test generator that also generated “expected results” per test. (See Section 5.1.)
- Embedding specific ISOC testing knowledge in the generator to achieve quality coverage with random generation.
- Basing most test writing on biased random generation with automatic testing, and writing a minimal number of tests manually.
- Coverage analysis. Recording all test execution paths and analyzing them for missing untested features.

The verification methodology depicted in Figure 2 was formed gradually as the project evolved. Not all of the project was executed according to it, rather the later stages were. Therefore the methodology presented is what we would have done if we had to do it over again.

4.1 Test Goals

The first phase of the methodology was to decide on the focus of testing since it was apparent that covering the functionality of the entire chip was not possible given the allocated time and resources. The ISOC was viewed as a system and the verification team primarily looked at the task as system verification, not unit verification. However there were differences in the overall strategy. In other systems verified, the effort of verification is much higher on the units than it is on the system. System bugs can usually be

easily fixed with software patches. Also, unlike for this ISOC, the system software configuration is well known in advance (being the end product), while this ISOC was intended to support several configurations. This required a substantial portion of the testing to exercise different system configurations which is done in a very limited manner (or not at all) in regular system testing. On the other hand, integration of the cores inside the ISOC limited their functionality which made unit testing simpler: only the functionalities available after chip integration were required to be tested for the ISOC.

1. Testing was mainly on end to end data transfers and MAL interrupts (in different configurations). The tests tried to cause contentions in time (requirement of busses and MAL by two or more operations). Contentions in space were also tested but with specific scenarios (described in Section 5.2), e.g. the shared memory locations between MAL and the CPU.
2. Chip and block configurations tested were mainly the ones used in the software applications. Information from the device drivers and initialization software was used to guide the testing. Note: this did not imply a single configuration, only how and when configurations can be programmed.
3. The PowerPC 401 was assumed tested. No testing was targeted towards the PowerPC 401 specifically.
4. Bad machine path (error handling and recovery e.g., timeout) were tested partially. The responsibility on testing bad machine path comprehensively was on unit simulation unless agreed otherwise for specific error conditions.
5. System testing only checked unit registers and configurations partially and randomly. Comprehensive unit register and configuration testing was done in unit simulation. Writes to registers were performed randomly in such a way that they were not expected to change the existing configuration (e.g. writing to a register its current existing value). This only tested the configuration write mechanism intermixed with regular writes (and not individual registers).

4.2 The Test Plan

The test plan, derived from the specification and test goals, was targeted for a random biased test generator. Instead of writing specific tests to cover, we wrote event tables that described combinations of events that should be covered during simulation. The test plan would then direct three activities:

1. The encoding of testing knowledge to the test generator in terms of test variants (what possible inputs can vary in the test and in what ranges). Different events meant different features to be generated randomly. In addition, biasing on events was specified and encoded into the test generator. In a sense instead of writing tests, we wrote test templates to be exercised randomly. The random test generator also gave us control over the frequency of events as described in Section 5.2.
2. The test plan directed writing behavioral models (behaviorals) for the simulation environments. Events described in the test plan had to be supported by the simulation environment input (i.e. environment behaviorals).
3. The test plan also directed coverage analysis. Since most of the tests were generated automatically we needed feedback to make sure that we were indeed generating what we intended. We used a coverage tool (see Section 5.3) to analyze traces from simulation and to report on coverage according to coverage models we coded in from the test plan.

After simulating the tests, the results were analyzed for failures, coverage, bug rate, etc. As a result of feedback from the review, modifications could be made to the test plan, coverage simulation environment, test generator and test coverage generator, in order to fix problems by adding more testing, correcting wrong assumptions, etc.

5. TOOLS

In addition to enhancements to SysGen, we built a few simulation environment tools to support the methodology:

5.1 The Simulation Environment

Figure 3 shows a high level view of the Barbados system verification environment. The SysGen system test generator creates tests that include the following components:

1. The instruction stream for the PowerPC 401.
2. Commands for ComMac bus behaviorals.
3. Initial memory state.
4. Expected memory state at the end of test.
5. Expected packet transfers for each ComMac during the test.

This test is then read by Simulation Control (built from a few modules) which performs the following tasks:

1. Parses the test.
2. Initializes the memory behavioral locations to be set up for the test. This includes the CPU instructions, descriptors for buffer packet transfers, data, etc.
3. Initializes the PowerPC 401 registers.
4. Submits the test for simulation.
5. Controls the ComMac bus behaviorals during simulation time. The behavioral commands parsed from the testcase are sent to the relevant ComMac bus behaviorals when required.
6. Stops the test on ending condition.

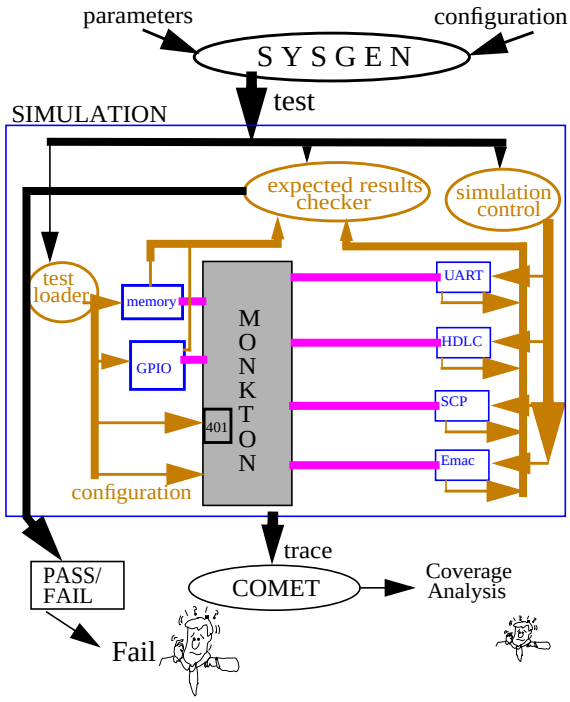

Figure 3. A detailed view of the environment

The results of the test are collected from the memory behavioral and ComMac behaviorals and compared with the expected state predicted in the test. Additionally a trace is produced from the test and submitted to Comet, the coverage tool, which accumulated coverage statistics on the chip simulation in order to measure the overall quality of the testcases.

The above process is almost entirely automatic, enabling a massive amount of test cases to be submitted and checked. The verification staff have to manually intervene only when a test fails (for debugging) and to review the coverage analysis.

5.1.1 ComMac bus behaviorals

All the ComMac bus behaviorals were tailored to have a common interface in order to make simulation control as simple as possible. The behaviorals' responsibility was to accept commands from simulation control in order to drive data into the network router and to dump all packets transmitted from the router to a file. The behaviorals requested all commands from simulation control. Simulation control supplied a command according to the test. When a behavioral finished executing a command, it requested another one until all the commands for it were executed. Additionally, all behaviorals produced a common format dump file containing all the packets transmitted to them throughout the test. This made both behavioral control and checking generic.

5.2 The Test Generator

The test generator, SysGen, was the heart of the verification solution. SysGen is a tool dedicated to system verification based on a generic system approach. SysGen builds a reference model according to a configuration file in which the user describes the contents and capabilities of the system under test. The configuration file includes the address spaces that exist in the system, the units that are active and which address spaces they can access, the system working modes, etc.

SysGen generates tests consisting of system transactions i.e., data transfers, interrupts, configuration transactions or scenarios. A scenario is a specific set of transactions (e.g. data transfers) used to generate a more complex transaction (e.g. a communication protocol transfer which may require multiple processor writes), or a specific set of transactions that exercises a specific system situation (e.g. back to back transactions of two units on the same bus).

SysGen's generation driver chooses transactions to run on the reference model guided by a user parameter file. The user has the ability to specify the test size and very explicit requirements for each transaction generated, such as the data transfer length, the target address space, and more. If the user does not specify choices, SysGen attempts to fulfill required choices randomly. The random choices are usually controlled by weights that bias the probability of choosing some possibilities over others. SysGen may also bias its choices on previous choices during the history of the test generation (e.g reuse of cache lines) and on given biasing rules. The user may control biasing by turning rules on/off or by assigning weights to them. SysGen can also choose to randomly insert scenarios in the test. Scenarios are usually not completely specified. While they ensure their purpose will be fulfilled, there is still ability to partially control them by the user or if not random biased choices are made.

SysGen views a system as composed of agents (active units in the case of Monkton, the CPU and ComMac behaviorals), targets (usually address spaces) and the relations among them defined by accessors. We call an accessor the set of information describing a transaction type. This information can be classified into information needed to run a transaction on the reference model, and information needed to build the directives for the simulation environment in order to run the same transaction on the simulated model. Sysgen uses the information contained by the accessor of each transaction type to update an internal simplified reference model of the system that it maintains. SysGen uses the test simulation directives contained in the accessor to add the corresponding transaction to the test itself.

Biasing is an important component of testcase generation since the level of novelty that each random test brings can decrease such that newly generated tests and the verification soon becomes inefficient. A method for increasing the quality of system tests is to force interesting sequences of events. This mechanism is used to focus system testing upon areas in which it is highly probable that system bugs occur, for example, system resource contentions.

Scenarios are added to SysGen's generation base, in cases where exercising system functions requires more than one basic transaction or in cases where the probability of hitting specific system conditions is extremely low (even with biased generation). In scenarios, the transactions are partially ordered and interdependent. Scenarios include aborting packets, reconfiguring the TDM in the middle of a test and resetting a channel while a buffer is being transferred. For example, the channel reset scenario was as follows:

1. A packet is placed on the chosen channel.
2. Both the transmit and receive channels of a ComMac are disabled. This causes the packet transfer to be corrupt.
3. After a delay the channels are re-activated and the ComMac reconfigured.
4. Another packet is sent to check that the channel operation has been restored properly.

The expected results are modified to reflect the fact that part of the corrupt packet may have been lost but the packet sent after the restore should be correct.

5.2.1 Adaptation for the network router

SysGen as a tool is adapted by the developers to the particular system it is applied on. The following work was performed in order to adapt Sysgen to Monkton.

1. **Preparing packets for MAL**: One of the biggest areas of concern was the functionality of MAL. Without its correct functionality the chip would not work. The challenge in verifying MAL was due to the many possibilities of data transfer descriptors. Packets are divided into buffers with each buffer having a buffer descriptor. The buffer descriptor stores data about the buffer, such as the address of the buffer, the length of the buffer, whether this is the last buffer in the packet, whether the buffer is "ready" etc. Some interesting variants to check were:
   - Differing packet formats: Packets which are transferred must have a certain format, depending on the ComMac and its configuration. Thus, for example, Emac packets always have a header and depending on the Emac configuration, the CRC may or may not be transferred into memory.
   - Single and multiple packet modes: ComMacs can transmit packets in multiple packet mode, in which case the ComMac, once activated, transmits all ready buffers until a buffer which is not ready is encountered. Alternatively, in single packet mode the ComMac transmits only one packet at a time and must be reactivated for each transmission. In single packet mode, the CPU checked that the previous packet had completed its transmission before reactivating the ComMac. In multiple packet mode, the CPU wrote to the relevant buffer descriptor changing its status to "ready" before activating the ComMac.
   - Buffer and packet lengths: The lengths of buffers and packets were randomized so as to ensure the widest range of testing of buffers. Thus, for example we allowed buffers and packets of zero length, buffers which were flush together, and buffers which were placed at different alignments in memory. The packet lengths varied for different ComMacs so as to allow optimal throughput in the simulated environment where some behaviors are significantly faster than others.

   During generation the descriptors were updated with information about the packet's transfer, for example whether the packet was terminated by a special character or by time out, whether a collision occurred during the packet transfer. SysGen prepared the initial descriptor buffers and predicted what their final values would be.
2. **Random biased configuration:** The system can work in a number of modes. There are global configurations (for example, which agents are in the TDM, whether interrupts are enabled) and configurations local to an agent (different techniques for starting packet transmission, using request-grant

mode, adding a CRC). As the project progressed, more configuration possibilities were added with the biasing prescribed by the user. Many configuration options could also be decided by the user for each test. For example, consider the configuration of a SCC. If not specified by the user, a number of configuration options were chosen randomly (subject to system constraints and interdependencies between the options), including: Single vs. Multiple packet mode, Transparent vs. HDLC mode, SCC or TDM mode, etc. The simulation environment behaviorals were also configured in accordance with the chip configuration. The descriptor buffers generated and expected were also in accordance with the configuration chosen.

3. **Error generation**: SysGen also allowed certain errors to be created, checking that they were correctly identified in the buffer descriptors. Such errors included CRC mismatches, loss of carrier sense during packet transmission, length in header not matching length of packet. This, however, was limited due to the requirement to produce expected results with the test. In tests wiith errors, it is difficult to predict results.

5.3 The Coverage Analyzer

Coverage played an important part in the methodology. Since most test generation was done automatically, it became necessary to monitor test progress automatically. The tool used was Comet[5]. Comet (coverage measurement tool) is a generic coverage measurement tool which receives as input many samples of static and dynamic event combinations. It returns statistics on them, such as how many times a combination was covered, what percent of the combinations were covered, etc. It can also be easily controlled to return statistics on a defined partial set of the combinations (e.g. all combination where the first entry value is 1).

In order to connect the simulation environment to Comet, a simple trace format was defined and a small utility was written to generate traces in this form after each simulation was run. This trace was later submitted to Comet as raw data.

Coverage measurement was defined in the test plan. About 30 small coverage models were defined to measure combinations on various functional areas of the chip. These included bus interactions, DMA interactions and concurrency of activities. For example, Table 1 below depicts such a model. This model measured MAL simultaneous arbitration requests from two ComMacs (including channels on the same ComMac), incorporating checks that each ComMac was configured to a different arbitration group priority, that each ComMac raised its priority level and that all combinations of ComMac pairs occurred at the same time.

Bus Id for ComMac 1	arb group ComMac 1	arb group ComMac 2	arb req level for ComMac 1	arb req level for ComMac 2
Emac, HDLC, UART1, UART2	1,2	1,2	low, high, urgent	low, high, urgent

Table 1: MAL arbitration coverage model

The most important coverage models of this category were models that measured back-to-back transactions on the PLB and especially OPB busses. These models gave us a good measure that we were exercising the system well with our tests.

Different types of models were written for measuring the coverage of different chip configurations. For a given configuration register, a configuration model was written to simply measure coverage of the different values that the register was initialized to in all the different tests. Their purpose was to measure that we were randomizing the configurations well enough.

Once the chip was mature enough, we started a process of review every few days: the coverage results were discussed and when coverage was missing, action items were taken. Sometimes the test generator was enhanced, sometimes manual tests were added and sometimes the missing coverage was check marked as OK, e.g., when we knew that we didn't have to cover a feature.

6. EXPERIENCE

The system verification work lasted for about 6 months. The verification work started earlier than what would be considered efficient because of tight project schedules. As a result the verification environment bring-up was harder than it should be. Nevertheless, since we based our checks on expected results, we could very quickly do basic testing and checking. In fact the expected results checking, once in place, required negligible maintenance (but there were also disadvantages, see Section 6.2).

The verification tasks that were covered were:

1. Memory to Channel data packet transfers.
2. End of Packet interrupts.
3. MAL programming options.
4. Specifically chosen configurations of operation (we could not and did not want to check all).

Figure 4 depicts the rate of hardware bugs each week. The shaded falling curve and bars are the number of bugs per week found, while the dark rising curve and bars are the number of packets simulated. As can be seen, we were seeing many bugs at the beginning due to the premature entry into the verification process (in fact this curve does not include some bugs which we attributed to this). Later the bug rate steadily dropped. Hand in hand, the number of simulation cycles increased, this had to do with the stabilization of the system. Since we were generating tests automatically there was no limit to the number of tests we could run. However, in the beginning most tests were failing. The packet drop in week 10 was due to a bug in the simulation environment when hardly any simulation was done.

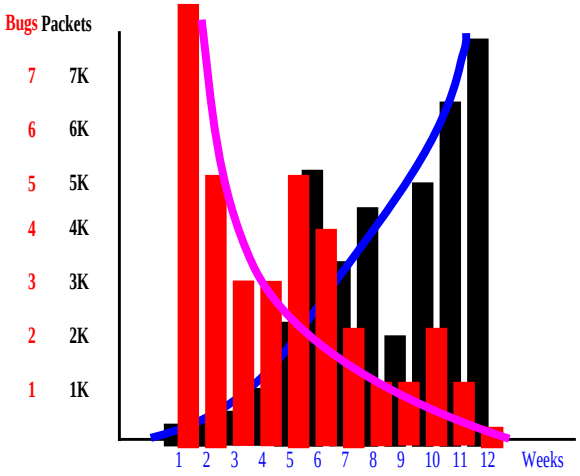

Figure 4. Simulation results

The bottleneck therefore became debugging and fixing bugs. Since we decided to perform most of our checks at the end of the simulation, we received very little information as to the failure cause. Debugging problems could take days due to the complexity of the chips. Eventually an expertise was achieved by the verification team and we reduced debugging time dramatically. There was a non-trivial resource investment here, but as time progressed this cost dropped sharply (to less than 1/2 hour). This was due to two reasons: firstly, the debugging expertise developed (it was quicker to pinpoint which unit or transaction were at fault); and secondly, due to the drop in failure rate (we had a lot of

failures which were not hardware, but rather, simulation environment bugs).

6.1 Tool Experience

The generator proved to be a very effective method for creating tests. It was impossible to manually perform the testing that we obtained with the generator. We also found very interesting bugs in corner cases:

Sample Bug #1: The CPU did a configuration read from EMAC: When the EMAC macro got two reads back-to-back by the MAL and bridge, it supplied the same data to both incorrectly.

Sample Bug #2: When MAL processed a packet with varying buffer lengths, it sometimes lost data.

In fact, it was a relatively low cost task. We had one programmer full time adding testing knowledge. We did not really have any test case writer per say (except for a small number of manual tests). There were however disadvantages.

The distribution of Comet models is depicted the table below. The first column is the domain. The REG domain depicts the configuration coverage models. The number of rows per trace specifies the amount of information the models had to process per simulation trace. There were about 60 new traces per day and a total of about 3000 traces analyzed.

Domain	Num coverage models	Rows per trace
PLB	5	~15K
OPB	5	~1.5K
MAL	8	~1K
REG	31	~50

Table 2: Coverage model distribution

Comet result reviews led us to three missing features in the hardware. However, hardware for those was not fixed because of time pressure. The reviews however gave us confidence in the quality of our tests and that the generator was doing its job. The test plan and SysGen testing knowledge were updated accordingly.

The back to back coverage models and the concurrent MAL request for service model gave us a good level of confidence that we were exercising the system well. Also the fact that coverage was constantly increasing on all the various modes of operations (we could only cover a fraction) showed us that the randomization was working well.

Comet as a tool cost more resources than we expected, but this was largely due to the fact that it was a first time experience. A lot of infrastructure was missing, but our continuing projects can now benefit from the work on Monkton.

6.2 Drawbacks

Using expected results for verification resulted in high debugging time and generation of tests limited to those where we could unambiguously determine the expected results. In some cases such as channel reset, we “patched” the expected results by marking memory locations as “undefined”.

Using expected results as a means for checking the system made generation easy when testing Good Machine Path (GMP), i.e., execution paths of normal operations. However, this method made it difficult to test events whose occurrence in time is difficult to predict, such as exceptions and interrupts. In most cases we refrained from testing Bad Machine Path (BMP). Our strategy was that these would be tested in the unit simulation environment. In the case of MAL interrupts we decided that it had to be verified in the system platform since it a very basic feature of normal system operation. The work on this feature within SysGen was substantially bigger than work on other features.

This was not a big difficulty in our case due to the fact that we did not verify the entire ISOC functionality. We prioritized the BMP features and invested resources in adding testing knowledge for those BMP features we were interested in.

6.3 Bugs Missed

A few hardware problems were found in the lab after the chip was fabricated. In most cases except one, the problems could be circumvented and software for the chip could proceed in development. A quick analysis of those bugs determined that most of them could not be found in system simulation. However, one or two could have been found, and the main reason we did not find them was due to the limits of our simulation resources. A typical simulation could take between 1-2 hours. The bugs missed were in the slower components (SCC). It was difficult to simulate long packet transfers for them (in the system simulation). Only a few modes were tried, since such simulations could take between 6-8 hours. These bugs could also be found in unit simulation where the simulation overhead was much smaller. The conclusion was that if the cost in system simulation of specific features is high, the system test plan should forward specific tests to the simulation environment of the units.

7. CONCLUSION

This paper describes a methodology for the verification of an ISOC, as well as our experience in one such project. ISOCs are just starting to emerge in the market and it is still an open question what is a good methodology for their verification. We believe the experience we depicted in this paper presents a reasonable solution to this question. The solution depends primarily on automatic generation of testcases and automatic checking of simulation results, similar to methods used in processor and system verification[1, 2, 4].

We found that for the most part, ISOC verification is a variant of system verification, but that the verification focus in an ISOC shifted from unit to system. The assumption we took in the verification plan is that the units (cores) have all been verified separately. Therefore, we can confine ourselves to verification of functionality that involves the interaction of several units such as date transfers. The verification of this functionality is largely covered by this methodology. On the other hand there are some drawbacks to this methodology such as debugging time for failed tests and covering BMP functionality. Additionally, some testing needs to be forwarded back to unit verification when the simulation resources required in system verification are too great.

8. REFERENCES

[1] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test program generation for functional verification of powerpc processors in ibm. DAC, 1995.

[2] A.Aharon, A. Bar-David, B. Dorfrman, E. Gofman, M. Leibowitz, and V. Schwartzburd. Verification of the IBM RISC System/6000 by a dynamic biased pseudo-random test program generator. IBM Systems Journal, 30(4), April 1991.

[3] G. Biran. MAL Functional Spec.. HDG, Haifa, ISRAEL, 1997.

[4] A. Chandra, V. Iyengar, D. Jameson, R. Jawalkelar, I. Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal. AVPGEN - A Test Case Generator for Architecture Verification. IEEE Transactions on VLSI Systems, 6(6), June 1995.

[5] R. Grinwald, Harel E., M. Orgad, S. Ur, A. Ziv. User defined coverage-a tool supported methodology for design verification. DAC 1998.

[6] C. May, E. Silha, R. Simpson, and H. Warren, editors. The PowerPC Architecture. Morgan Kaufmann, 1994.

[7] A.Mesh, EmacII Functional Spec., HDG, Haifa, ISRAEL, 1997.

[8] M. Schaffer and E. Green. On-Chip Peripheral Bus Specification. PowerPC Embedded Proceesor Solutions, RTP, NC, Mar. 1996.

[9] M. Schaffer and J. Revilla. PowerPC 4XX Local Bus Specification. PowerPC Embedded Proceesor Solutions, RTP, NC, Oct. 1996.