Hierarchical Random Simulation Approach for the Verification of S/390 CMOS Multiprocessors

Jörg Walter, Jens Leenstra, Gerhard Döttling, Bernd Leppla, Hans-Jürgen Münster
IBM Deutschland Entwicklung GmbH
D-71032 Böblingen
Germany

Kevin Kark, Bruce Wile
IBM Corp.
Poughkeepsie, NY
U.S.A

Abstract --- In this paper an approach is presented for the hierarchical verification of the memory control units, I/O adapters and processor interconnect units as found in multiprocessor computer systems. It is shown how such units could be verified better and faster by the introduction of random executable timing diagrams and associated CAD tool support. Furthermore, it is shown how the timing diagrams for the unit network verification are easily derived from the timing diagrams specified for the units. The multiprocessor hardware test showed the effectiveness of the proposed verification approach.

1. Introduction

To keep pace with the processing performance growth of 75% per year, not only the complexity of the processor unit (PU) with its caches will increase, but also that of the supporting units, like processor interconnect units, memory control units and I/O adaptors. To reduce the time the PU waits for data, many complex functions must be performed by such units. These chips for example have to: (1) buffer and arbitrate incoming requests, (2) handle multiple outstanding (for example load/store) operations, (3) maintain the consistency and coherency of the multi-level caches, (4) handle the complex high speed bus protocols and so on. In the current multiprocessor systems so many tasks have been added to the supporting units that it becomes a very challenging and time-consuming task to verify the overall logic behavior of these units. While faults in the PU can often be circumvented by microcode, such is generally not possible for these supporting units. Therefore, it is vital for the time-to-market and development cost of the complete project that such chips are fully operational from first silicon.

A number of verification approaches have been proposed for the verification of multiprocessor systems. Widespread is the use of the traditional late stage functional verification of the system using test programs executed by the PU. These test programs are often generated by advanced code generators using pseudo-random techniques[1, 2, 3]. The use of this approach alone is no longer sufficient when the supporting chips become more and more complex. Formal approaches have been proposed for verifying the supporting chips. For example, they are used to prove the correctness of the cache updates and so on [4, 5]. However, the size of the circuits which can be handled by such approaches is still too limited.

This paper is organized as follows: Section 2 reviews the problem of verifying the bus controller ASICs for our multiprocessor system. Section 3, presents the incremental verification approach used to verify our ASICs. In Section 4, the CAD tools will be introduced as used for hierarchically specifying the timing diagrams followed by the execution of the timing diagrams, such that the design parts are verified by a “controlled” random simulation. In Section 5, the results are presented. Finally, in Section 6 our conclusions are given.

2. Verification of our Multiprocessor Systems

Figure 1 shows the basic structure of the processor module of the third generation IBM S/390 CMOS processors. These modules form the processor core of the “IBM S/390 Parallel Enterprise Server - Generation 3” and the “IBM S/390 Multiprise 2000” systems. These systems are the successors to our previous water-cooled high-end mainframes[6].

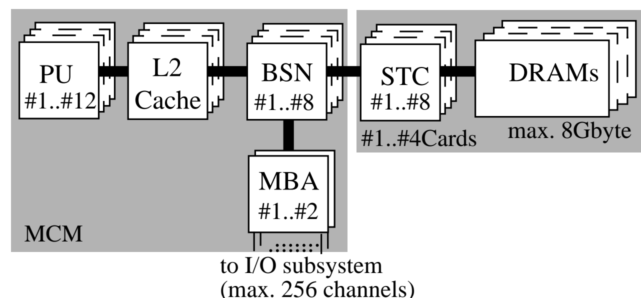

Fig. 1. S/390 CMOS Multiprocessor System Structure

“Permission to make digital/hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.”

DAC 97, Anaheim, California
(c) 1997 ACM 0-89791-920-3/97/06 ..$3.50

IBM, S/390, IBM S/390 Parallel Enterprise Server - Generation 3, IBM S/390 Multiprise 2000 and RS/6000 are trademarks of International Business Machines Corporation.

Our multiprocessor systems are built out of 2..12 PUs, associated secondary caches (L2), processor interconnect units (Bus Switching Networks, BSN), I/O adaptor units (Memory Bus Adapter, MBA) and memory cards (Storage Controller, STC).

In our previous multiprocessor systems we used the traditional late stage functional verification approach which was preceeded by “unit simulation”, verifying more complex units separately. The late stage verification tested the communication between the units and the overall logic behavior of the units by executing test programs (called “test cases”) in the PUs. Such a verification approach was possible since the BSN was a relatively simple switching network and the design changes in the STC and MBA were minimized through re-use of large parts from our first generation CMOS designs.

Such a re-use did not lead to twice the performance in the case of our third generation systems. This is because the memories become larger and do not decrease access time in the same ratio as the logic chips. Therefore, all supporting units had to be completely redesigned. A shared L3 Cache was added to the BSN and the protocol had to be re-specified to handle the consistency and coherency of the multi-level caches. Furthermore, many sophisticated bus/cache line interleaving schemes were added to reach the required data transfer bandwidth on each interface. How the complexity of the supporting units grew, compared to the PU, is indicated in Table 1. The “ratio” numbers give the factor of increase between the second and third generation systems. The “Random Logic Complex Gates”, in particular, gives a good indication of the complexity increase of the supporting chips in relation to the PU.

	#Transistors	Chip Size (mm)	Pins	Pin Incr. Ratio	Random Logic Complex Gates	Gate. Incr. Ratio
PU	7.185.512	14.6	744	1.7	164.056	1.8
BSN	16.617.686	14.6	758	1.8	68.176	**8.9**
STC	995.690	12.7	737	1.6	58.847	**1.9**
MBA	3.607.715	15.5	770	1.6	206.037	**6.0**

Table 1. Chip Characteristics

The new features of tightly coupled complex interfaces between units make it no longer feasible to sufficiently verify the overall behavior of the units by simply executing test cases in the PU. The added functions in the support units and the higher speed protocols resulted in a much larger number of overall multiprocessor machine states.

The basic difficulty of verifying the supporting units is, that a limited number of commands which store/fetch data from the memory and update the caches control/data bits can generate a huge number of different execution sequences and cache/memory update scenarios. Due to the required bandwidth between PU, memory and I/O, all or multiple chip ports may be active at the same time and all ports may execute one or more (interleaving) different commands. Writing test cases for all these different scenarios (to verify the behavior by simulation) is no longer feasible. Note further that the execution of all these test cases in the system model would be very slow due to the increased size of the system model. Therefore, a new verification method was needed which could provides us with a high quality and high-speed verification of overall functionality of the supporting units, without the overhead of generating the verification stimuli via the PU/L2 units.

3. The Verification Approach

As previously mentioned, the basic problem of verifying the supporting units is, that “a limited number of commands which store/fetch data from the memory and update the caches control/data bits can generate a huge number of different execution scenarios and cache/memory update scenarios”. The following approach can provide a solution when writing a test case for each such possible scenario is no longer feasible:

1) *Write a test case for each basic command:* During this step the input and output patterns are given for each basic command. In our case the basic commands are “fetch data from memory”, “store data to memory”, “store cache line & fetch memory line into cache”, “fetch data from cache via cast-out” etc.
2) *Select randomly when each test case is started during the simulation:* This step creates a random test program for the simulation. By permitting that several of the basic commands may be active at the same time, the occurrence of very complex traffic situations on each chip interface is enabled.

For example, in Figure 2 test case 1 describes a “fetch data from memory” and test case 2 describes a “store data to memory”.

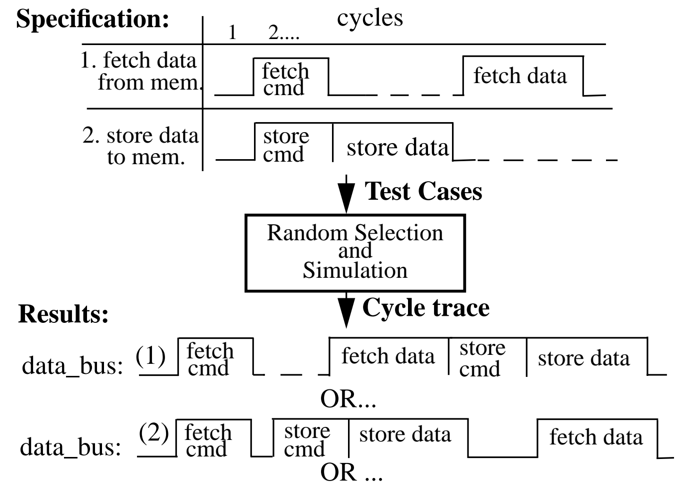

Fig. 2. Creating complex traffic from simple test cases.

The random selection process decides how both test cases are started in relation to each other. This leads to an outcome that (1) both commands are executed one after another or that (2) the store command is executed between the start of the fetch command and the return of the fetch data from memory. Note that the number of cycles between the start of the fetch and the start of the store command has to be varied before it can be concluded that the protocol is implemented correctly, since it must be verified that fetch data is not returned during

the store data transfer. So with just two simple commands several scenarios exist. The number of scenarios even increases, when the number of cycles between a fetch command and the return of fetch data is not fixed. This may be caused by a DRAM refresh, or by 'shared' or 'exclusive' states of data in the L3 cache. Furthermore, faults like parity errors, uncorrectable memory errors and protection errors may be reported for each command further, increasing the number of scenarios.

When using such a random approach it is never guaranteed that all complex traffic scenarios of interest will really be covered. However, with the help of fault coverage analysis tools and control parameters for the random selection process the probability can be increased to a very high level.

During the development of the second generation S/390 CMOS processor chips we used such a random approach in the unit simulation for the MBA. Thereby, the test cases, as well as the randomizer, were implemented with help of a C like behavioral modelling language. However, it was found that such a behavioral model (coded by C or some other programming language) has the disadvantage that the behavioral model code is hard to re-use for testing the communication between the units and that the randomizer is written newly in each of the behavioral models due to the restrictions when commands may start. Furthermore, such a model is quite complex and hard to debug. Therefore, another method will be proposed for entering the test cases and for controlling the randomizing process.

The design of the protocols is usually started by specifying them in the form of timing diagrams. On the system level, timing diagrams specify in an easy to comprehend manner the timing of the commands on the interfaces of each unit. Thus, our preferred manner for specifying test cases was the use of timing diagrams. An example of how a test case for the BSN unit can be specified as a timing diagram is illustrated in Figure 3 by the timing diagram of a "fetch data from memory".

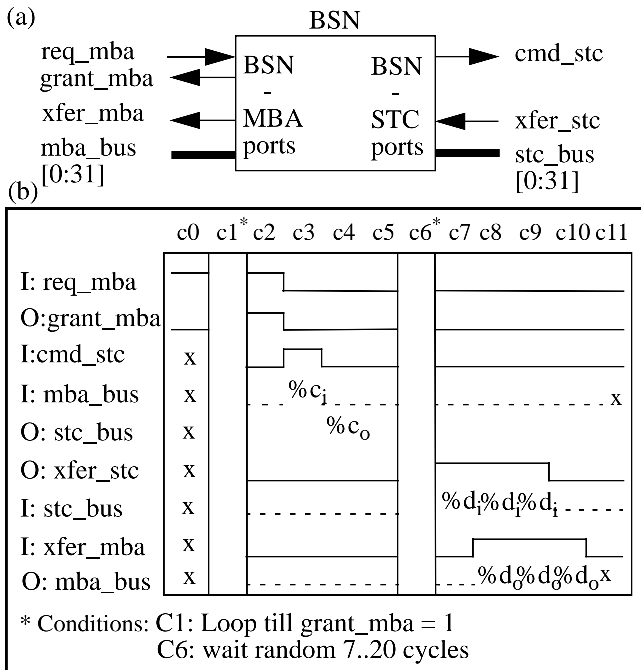

Fig. 3. (a) Part of BSN and (b) timing diagram for "fetch data".

The BSN inputs and outputs which are needed to execute this command are given in Figure 3(a). In Figure 3(b) the signal names preceeded by "I:" specify how the input and bidirectional port have to be set at the beginning of each cycle, the signal names preceeded by "O:" specify the state of the output signals at the end of the clock cycle. For a bidirectional bus ("mba_bus" and "stc_bus") the input row "I:" specifies when the bus has to be driven and the output row "O:" when the data on the bus has to be verified. In the timing diagram of Figure 3 there are two special cycles "C1*" and "C6*" expressing a "recurring cycle" with associated waiting conditions. For example, to start the execution of a "fetch data" command first the "req_mba" has to be raised. The state of all other signals except the "grant_mba" is "don't care" since another operation may be going on. Next, it will take an unknown number of cycles before the BSN unit will allow the start of the command by a "grant_mba" pulse. Therefore, the timing diagram state will remain in "C1" until the condition "grant_mba = 1" has been fulfilled.

After the request has been granted, the command data has to be driven onto the bidirectional "mba_bus". This is done by calling a small C program as given by "%c_i". This C program puts the data onto the "mba_bus" and later on the call "%c_o" verifies that the correct command data has arrived at the "stc_bus".

In the same manner, fetch data is put on the "stc_bus" by calling "%d_i" and the arrival of this data at the "mba_bus" is verified by the call "%d_o". When the data is put on the "bus_stc", it is accompanied by an active "xfer_stc" to tell that valid data is on the "stc_bus". Finally, by the predefined timing, for "xfer_mba" it is verified that the "xfer_mba" follows the "xfer_stc" signal with a one cycle delay.

Once we have specified the timing of a command for the simulation of an unit, it is basically very easy to convert such a test case into a test case for the simulation of a connected unit. For example, once we have the "fetch data" test case for the BSN as shown in Figure 3(b), we can construct the test case for the STC simulation by:

1) deleting the "mba_* and "*_mba" signals and some columns
2) replacing "I:" by "O:" and "O:" by "I:" and
3) replacing the calls (or a parameter which defines the bus on which the data has to be written or verified) and modifying the conditions.

In Figure 4 the resulting timing diagram for the STC simulation is given. The "%e_i" call now puts data into a memory line and is called by introducing a "dummy" signal (i.e. signal not present in the circuit). Furthermore, another parameter is associated with the "%d_o" call (not shown in the Figure) such that the called C program knows that it has to check the data on the "stc_bus" instead of the "mba_bus".

After the timing diagrams for the units have been constructed, they are used to verify each unit during simulations. In our bottom-up hierarchical verification approach, the verification will next continue by verifying the network of connected units. Thereby, re-use is made of the timing diagrams

of the units. In other words, the timing diagrams for the larger network are defined making simple changes to the timing diagrams for the units. Note that once we have written and debugged the C routine for the calls in the unit simulation the C routines for the overall simulation are the same except for the signal names on which the data occurs/is checked. In our approach the signal names are therefore simply made known to the C program by calling the routine with a parameter which defines the bus.

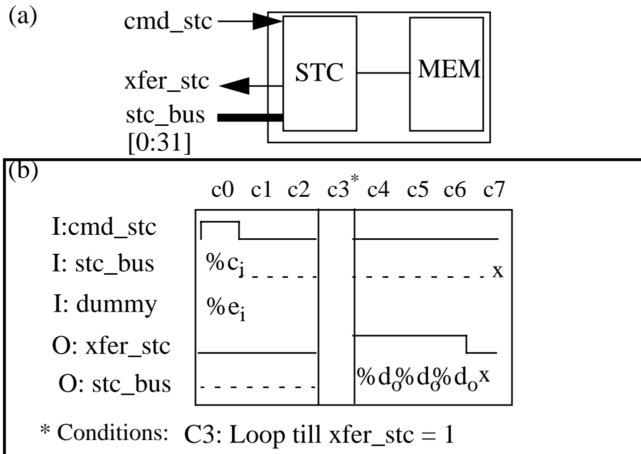

Fig. 4. (a) STC/MEM and (b) timing diagram for “fetch data”.

So far, we only discussed how we specify each command by the use (and re-use) of timing diagrams. As discussed before, to actually verify each unit and the network of units the timing diagrams have to be randomly started such that complex traffic situations are created from the simple timing diagrams we just defined. How this is done will be discussed in the next section by introducing the CAD tools for entering the timing diagrams and for executing them randomly.

4. CAD Tools

We used an IBM CAD tool called TIMEDIAG which provides the possibility to enter simple test cases in the form of timing diagrams. Furthermore, a tool called GENRAND was used which enables the random execution of timing diagrams specified using TIMEDIAG. These tools were basically developed to ease the verification of interface logic. However we used them for the verification of our large chips. The problem of using timing diagrams without supporting C programs is, that the number of signals which have to be specified becomes large (especially for data busses) and it becomes very time-consuming to construct the timing diagrams. However, with the introduction of calls to C routines, implementing data generation and data verification on the complex busses, we found that we could use the same tools for the verification of very large chips and even subsystems.

4.1 TIMEDIAG

TIMEDIAG is a graphical editor to create or modify timing diagrams. The timing diagrams have the structure of a spreadsheet as shown in Figure 3(b). Signals and variables are specified in rows and cycles in columns. Variables are global variables in the sense of a programming language like C. The value assigned to a variable is known in all timing diagrams and can for example be used to control the duration of a recurring cycle such that a timing diagram waits until another timing diagram has entered a specific cycle.

The content of the cell on the intersection of a row and a column determines the value of the signal or variable in that cycle. In this cell, a value, an equation, a call to a C routine, or a “don’t care” condition can be specified.

In TIMEDIAG a so-called “limiter”, is used to control the start of each timing diagram during the simulation. Using the “limiter” it can be (and has to be) prevented that two timing diagrams require to set the same input signal to opposite values. For example, if the “fetch data” timing diagram and “store data” timing diagram use the same signals then the limiter must prevent that both timing diagrams set the same signal at the same time.

Each limiter consists of a boolean equation which must evaluate to “true” before the timing diagram can be started. Furthermore, the limiter contains a variable field for the specification of a global variable name. This variable is incremented when GENRAND selects the timing diagram to start and it is decremented when the timing diagram execution is ended. The simultaneous execution of the “store data and “fetch data” timing diagrams can now be prevented as follows. In both timing diagrams the use of the same variable for increment/decrement is specified. For example, lets us assume that the variable is called “port0_busy”. This variable is set to “1” as soon as the fetch or store timing diagram is selected to be started. If the limiter equation of each timing diagram is set to “port0_busy < 1” then no other timing diagram can be started before the active fetch or store timing diagram has ended.

4.2 The GENRAND Simulation Environment

The structure of the simulation environment is shown in Figure 5.

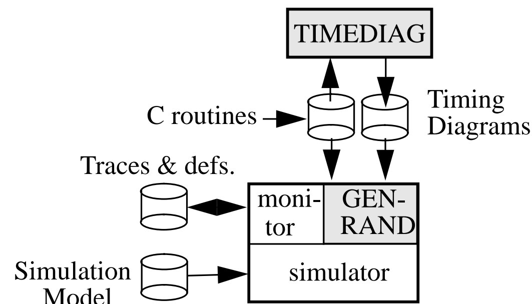

Fig. 5. GENRAND simulation environment.

The timing diagrams created with TIMEDIAG are stored in a database. Together with the compiled C routines for the calls in the timing diagrams this database is the input for GENRAND. It provides a graphical interface for the selection of:

- *the set of timing diagrams to be executed.*
- *a timing diagram start probability;* GENRAND tries to start the timing diagram every cycle when setting the probability to 100%.
- *a seed number;* the seed number is generated automatically and it forms the starting point for the randomizer. The seed

number will be entered manually if a simulation run has to be recreated, for example, to verify that the logic fault found by a particular simulation has been corrected in the design.

- *run time control;* for the selection of the number of cycles which have to be simulated and so on.

GENRAND acts as a simulation driver for our simulator, randomly starts timing diagrams and checks the outcome of the execution. In case GENRAND detects an error, control is passed to the monitor, which controls the output of trace files and enables viewing of the state of all signals in the simulation model. Furthermore, GENRAND writes into a result file the characteristics of each simulation such as the number of times a timing diagram was started, which timing diagrams were concurrently active and so on.

4.3 C Routines

In addition to the routines implementing the calls in the timing diagrams, a number of other small programs were written for trace file analysis and for copying and (hierarchically) modifying timing diagrams. For example, a small conversion program was written which translates the timing diagram defined for a specific PU/L2 port of the BSN unit into the timing diagram for all other PU/L2 ports.

5. Results

5.1 Creating the Timing Diagrams and C routines

As indicated in previous sections, we started by creating timing diagrams for the MBA, BSN and the memory card (STC unit and behavioral model for memory bank) such that each unit could be simulated separately by unit simulation. The MBA was only partially included since many of its parts could be covered by the models developed for the random MBA unit simulation in our second generation system. To perform the unit simulation all C routines which were called by the timing diagrams were written. There are 4 different classes of these C routines:

1) *Procedures to drive data and check data on the busses;* These routines were written such that the names of the buses were defined globally and that a parameter specifies, on which bus the data has to be driven or verified. The introduction of such a parameter circumvents the repeated use of signal names for busses and makes the routines modular. The use of the same routine for the simulation of another unit only requires the modification of the parameter.
2) *Procedures to set and check the L3 Cache contained in the BSN;* With these routines, the L3 Cache data and the cache control bits can be set. Furthermore, these routines are used to check if L3 Cache lines are loaded and the control bits set correctly.
3) *Procedures to access the behavior model for the DRAMs;* Data in main memory and in the keystore array can be set or checked by using these functions. Furthermore, these routines generate the error conditions which can be reported by the memory, like “correctable memory errors”, “uncorrectable memory errors”, “key protection error” and so on.
4) *Misc. procedures;* These functions control the behavior of the memory model with respect to access time, refresh rate and so on.

For our unit simulation a total of 24 different C routines have been implemented.

For the MBA, eight basic timing diagrams needed to be created. After these basic timing diagrams were running, the timing diagrams were copied and error handling (parity errors, uncorrectable memory errors and so on) was added. A total of 29 timing diagrams resulted. These 29 timing diagrams were all translated into timing diagrams for the BSN, MBA and STC subsystem model. For the BSN, only the command started by the PU/L2 units was modelled, since timing diagrams of the MBA ports could easily be converted from timing diagrams discussed previously for MBA unit simulation.

For the BSN, twenty basic timing diagrams needed to be created in order to cover the 41 commands in our system. Note that many different operation codes have the same basic timing and therefore we can simply randomly select the operation code when the command is put onto the bus by calling a C routine. Inserting “error handling” and generating the timing diagrams for other PU/L2 ports, resulted in a total of 61 timing diagrams. All timings check the predicted data, L3 cache states and the compliance with the expected protocol.

The basic timing diagrams for the STC were composed by extracting them from the BSN/MBA timing diagrams. 22 timing diagrams were needed.

The overall time which was needed for the creation and the debug of these timing diagrams, as well as the units, was about three months. This is very fast in comparison to the random approach used for the MBA in our second generation system. In the latter case, it took 6 months before our MBA unit simulation was running stable.

5.2 Running the Simulation

The simulation was run on RS/6000 workstations. At the start of the verification, each unit simulation was done on a separate workstation. During the final regression runs, we used 6 workstations to obtain the number of simulation cycles which gave us confidence, that the design contained no logic faults in all logic essential for the hardware test of our multiprocessor system. The unit simulation reached a performance of 10 cycles/second. This was a factor of 5 faster than when simulating the units connected together which ran at 2 cycles/second. Note that this is still significantly faster as our traditional late stage simulation where the performance was limited to 0.25 cycle/second.

Furthermore, when comparing the traces of our simulation with the traditional simulation, it was found that the number of commands which was executed in the same number of cycles was about a factor of 10 more. This results in a much higher fault coverage, since the supporting units are “stressed” in a way which was seldom achieved by the generated verification code as executed by the PUs. Hence, to reach the same fault coverage with the traditional functional verification approach we would have needed more than a hundred workstations.

5.3 Verification Quality

The effectiveness of the presented simulation approach was proven during the real hardware multi-processor system verification. Although the complexity of the BSN grew by a factor of 8.9, the number of problems found in the hardware on the test floor was the same as in the predecessor machine. Only 3 minor problems showed up in first silicon, which could have been found by more unit simulation. For the part of the MBA logic which was included into our model MBA, only a single problem was found. This problem was easily circumvented by setting the counter for a specific window to a particular value. Finally, 3 STC problems were discovered. Two of these were caused by an incomplete modelling of the DRAMs and therefore could not be discovered by our simulation. This is a reduction of 60 percent as compared to the previous system.

These problems did not seriously hamper system test. All problems were fixed by using spare gates on the chip and wiring to existing gates (metallization change only). Hence, no new silicon had to be produced and only masks for the metal layers had to be changed. The overall result was a 6 month period between first-silicon and first customer shipment.

5.4 Advantages of the Approach

We found that the timing diagram based approach has many advantages compared to our previous approach, were the test cases and the randomizer were implemented by writing models (in a C like programming language) for each interface:

- *Timing diagrams result in a precise and easy to read specification;* The use of timing diagrams in contrast to implementing the same behavior by using a programming language like C is that the timing diagrams are a much easier to comprehend specification. Also in previous projects timing diagrams were used to specify the protocol for the command execution of the supporting units. So the effort to convert the timing diagram specification into models which randomly stimulate each interface is no longer needed.
- *Specification can be executed and remains up-to-date;* By entering the timing diagrams with the help of TIMEDIAG each timing diagram can be directly used for verification. Using the specification for verification of the logic behavior keeps the specification up-to-date during the whole project.
- *The randomizer is now a part of the GENRAND tool;* In the past each behavior model had its own code for generating random patterns for the simulation. This is no longer needed in the approach presented.
- *Several persons can create and debug timing diagrams for the same interface;* The writing of programs for stimulating an interface often has to be done by a single person, since he is the only one who understands the main structure and partitioning of his program. Especially in the case of faults, the programmer is the only one who can correct his program. In the new approach, everyone in our team could understand the timing diagrams and could debug them without assistance.
- *Possibility to start very early with simulation;* As soon as a timing diagram has been created we could immediately start the verification. In the old approach that was much more complicated since many routines had to be ready before the verification could start.
- *Better verification of protocols as the state of a signal has to be specified cycle by cycle;* In a TIMEDIAG spreadsheet, the state of each signal has to be specified for each cycle. In the past, the behavioral model did not completely test for the state of the output signals, due to the amount of extra code needed to implement such features.
- *Re-use of the timing diagram;* Once we had the timing diagrams for a particular model, the timing diagrams for another simulation model were created by modifying the existing timing diagrams. Such a re-use was not possible in previous projects, because a program written for a specific interface requires an overall modification of the program before it is suited to drive another interface.
- *Hierarchical and incremental;* The use of the approach presented here enabled us to start the verification for small units. It enabled a fast debug of the logic, the timing diagrams and the C routines which were called. In previous approaches we directly started to verify relative large circuit parts since it was too time consuming to write a (not to re-use) program stimulating each interface.

6. Conclusions

In this paper, we presented a new approach which uses randomly executable timing diagrams for the hierarchical verification of supporting units between processor, memory and I/O channels. The use of timing diagrams has the advantage that they are a precise and an easy to read specification. The presented approach supports (and is restricted to) the verification of units in which the verification complexity is caused by the concurrent execution of many relatively simple commands (or timing diagrams). We found that timing diagrams constructed for a unit could easily be re-used for the verification of connected units, as well as the overall network, when calls to C routines were introduced for driving and verifying data busses. With this approach, only three months were needed for verification and a much better fault coverage was reached than for previous systems. As a result of the better and faster verification approach, no new silicon had to be produced and the system could already be shipped six months after first-silicon.

References

[1] A. Aharon et. al., *Test Program Generation for Functional Verification of PowerPC Processors in IBM*, Proceedings 32th ACM/IEEE Design Automation Conference, 1995.

[2] A. Hosseini, et. al., *Code Generation and Analysis for the Functional Verification of Microprocessors*, Proceedings 33th ACM/IEEE Design Automation Conference, 1996.

[3] A. Chandra et. al., *AVPGEN-A Test Generator for Architecture Verification*, IEEE Trans. on Very Large Scale Integration (VLSI) Systems, June 1995, pp. 188-200.

[4] K.L. McMillan, *Symbolic Model Checking*, Kluwer, 1993.

[5] I. Beer, et. al., *Rule Base - an Industry Oriented Formal Verification Tool*, Proceedings 33th ACM/IEEE Design Automation Conference, 1996.

[6] J. Young, *Exploring IBM's New Age Mainframes*, Maximum Press, 1995, 481 pages.