Functional Verification Methodology of Chameleon Processor

Françoise Casaubieilh, Anthony McIsaac, Mike Benjamin, Mike Bartley, François Pogodalla,
Frédéric Rocheteau, Mohamed Belhadj, Jeremy Eggleton, Gérard Mas, Geoff Barrett, Christian Berthet

Chameleon Programme, SGS-THOMSON Microelectronics

Abstract - Functional verification of the new generation microprocessor developed by SGS-THOMSON Microelectronics makes extensive use of advanced technologies. This paper presents a global overview of the methodology and focuses on three main aspects :

- Use of acceleration and emulation technologies for the verification of the VHDL specification in the early stages of the design.

- Development and use of sequential verification methods built upon a commercially available formal proof tool.

- Extensive use of combinational proof for circuit-level verification, in conjunction with transistor abstraction.

1 Introduction

Chameleon is a programme of next generation microprocessors developed by SGS-THOMSON Microelectronics. It is based on a modular, core-based 64-bit superscalar architecture. The first microprocessor generation is targeted at the consumer computing market. It implements multi-media features, as well as common microprocessor capabilities.

For such highly complex microprocessor developments, functional verification is estimated to take between 30 and 50% of the design resources. Imperfections in the verification process not only affect time to market but also lead to costly mask revisions.

The goal of silicon design functional verification is to gain a high level of confidence that the silicon implementation satisfies the specification of behaviour. To achieve the verification of such a highly complex chip, and to ensure quality improvement throughout the design process, different technologies are used: simulation, acceleration, emulation, test generation, formal verification and ASIC prototyping.

Priority is given to the objective of reaching a high level of confidence in the first stages of the design. In fact when the physical design starts, the RTL (Register Transfer Level) specification has already been verified by running billions of machine cycles and making use as much as possible of formal verification techniques. Each step of the physical design is checked versus the RTL specification.

This paper explains the functional verification methodology used for the design of Chameleon processors. It consists of 2 major points:

- verify that the VHDL specification is conformant to architecture and microarchitecture specifications.

- verify that actual layout is conformant to the VHDL specification.

The first issue is addressed in Sections 2 (Description levels), 3 (Simulation-based verification) and 4 (Sequential verification). Verification of the VHDL reference specification uses both simulation-based (including acceleration and emulation) and formal verification-based techniques. The second issue is addressed in Section 5 (Circuit verification): transistor abstraction and combinational formal proof are the primary mechanisms used for circuit-level verification.

2 Description levels

The different levels of specification developed for Chameleon design and relevant in terms of functional verification are the following:

- Level 0: Instruction Set C simulator.

- Level 1: Behavioural VHDL RTL model.

- Level 2: Structural VHDL RTL model.

- Level 3: Circuit transistor level model.

The instruction set C simulator is the golden reference model; it accurately models the function of the processor instructions. This simulator is used by the software group, application developers and customers.

The level 1 model is a behavioural VHDL model with two main characteristics: first, it is accurately faithful with respect to the micro-architecture specification, and second it is synthesizable.

Compliance w.r.t. the micro-architecture specification means that it is cycle-accurate at component boundaries. It describes precisely the interactions between the components of the chip at each clock cycle. Within each component, resource allocation and exact scheduling of operations does not necessarily reflect the silicon implementation. Level 1 is *sequentially* equivalent to the final implementation with respect to the observable behavior at component boundaries.

This level 1 model is also designed to be synthesizable in order to benefit from acceleration, emulation and sequential proof as explained in the following sections. The objective of producing this model is to have, as soon as possible during the design process, a sensible model of the chip which can serve as a reference for the rest of the logical and physical design.

The level 2 model is a structural and state accurate VHDL model obtained by successive refinements of the level 1 model. It is *combinationally* equivalent to the final implementation, i.e. each VHDL state corresponds exactly to a memory element in the actual silicon. Moreover, the block decomposition exactly matches the physical block decomposition.

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

Finally, the level 3 model is the final implementation in the silicon directly extracted from the chip layout databases.

The VHDL model is used as the reference design specification throughout the entire design cycle. It is validated, both at level 1 and level2, by comparison with the instruction set C simulator by running billions of cycles. The size of the VHDL code is typically of the order of several hundred thousand lines.

3 Simulation-based verification

3.1 Global overview

The important point about the implemented methodology is that the functional verification is done at chip-level. This requires some validation to be done by the designers on their blocks before going to integration in the whole model, using an appropriate testbench for each block. Then, after integration of the different components into the whole model, the functional validation of the VHDL specification is done via chip-level verification.

Verification patterns are applied to the whole chip rather than component by component. This is realised by putting the chip in either a VHDL model of its environment (VHDL, acceleration) or in a hardware environment (emulation). Chameleon binaries (assembler programs) are effectively executed by the model. Executed means that the design fetches itself its patterns as they are actual programs. Patterns should not be considered as classical bit-vector patterns, but rather as design-environment communication in the execution of a program.

A regression test procedure serves as the cornerstone of the verification at chip-level. This procedure is composed of Chameleon assembly code programs. Any version of the design process must successfully pass this regression test before being officially released.

The design of Chameleon, as any other microprocessor design, requires the simulation of many billions of machine cycles in order to assess the global coherency of the circuit design with respect to its operating environment. Therefore, regression testing is extended to include large programs such as OS, C programs and applications.

The size of test suites is traditionally a limiting factor in this validation phase. This is overcome by faster implementations of the models (hardware acceleration and emulation) and by using test suites which give a high degree of coverage within a small number of cycles.

3.2 General Simulation Flow

The following figure shows the global flow for validation of the chip, from test generation to the comparison between the software simulator and the hardware models.

3.3 Test Case Generation

Test generation is driven by a test-plan with 2 parts: architecture verification and implementation verification. The first part is dedicated to testing the compliance of the chip to its architectural specification. Architecture verification tests aim at verifying access/ update of architectural resources as defined in the architecture manual, without regard to implementation issues. They are targeted at stressing the architecturally visible resources just as a user of the chip could do by using the microprocessor in a system and running code on it.

The second part of the test plan validates the conformance of the chip to its micro-architectural specification. It is intended to verify the micro-architectural features like protocols between blocks, state machines, hardware resources (circular tables, counters, ...).

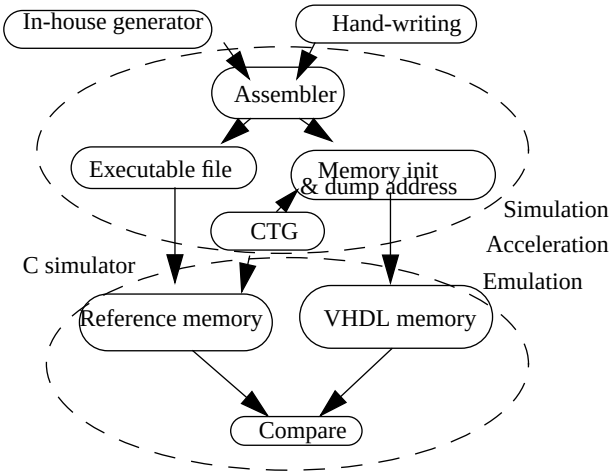

Fig.1. Simulation flow

Both parts result in the production of tests, so-called Architecture Verification Patterns (AVP) and Implementation Verification Patterns (IVP) [1]. These tests are actually assembler programs that are either compiled with the Chameleon software toolchain (assembler-linker) to produce Chameleon microprocessor binary code or directly generated as binaries to bypass the constraints and optimisations done by the assembler-linker. Assembler programs may be hand-written or generated with an automatic generation tool.

Chameleon uses both an internal tool and Chameleon Test Generator (CTG), built upon a model-based technology developed by IBM research laboratories at Haifa [1].

The internal tool generated tests for the verification of a Chameleon ASIC prototype that has been developed within the project. It produced a test database of 17 million clock cycles, all being AVPs and single instruction based (that is, dependencies between instructions are not taken into account for example). Hence, the fact that the needs in terms of test generation were bigger for the verification of the Chameleon led the project to go for CTG.

The motives for using such a tool are the following :

- changeability of architecture details, reusability of generator core : during the design process, the way the architecture evolves should not impact the generator core. In addition, it is important to be able to reuse this core for different designs of the architecure, avoiding any redevelopement, and allowing immediate focus on test specification issues

- visibility, changeability and reusability of the testing knowledge : the improvement of the quality of tests requires the building of a continuously upgraded knowledge database. Moreover, this knowledge database has to be readable and reusable for future designs based on the architecture

- capability of biasing the tests : it is essential to be able to specify test patterns that have to be exercised (hit corner and boundaries of test-space, stress shared architectural or micro-architectural resources)

- dynamic generation : the generation is interleaved with the computation of the expected results; thus each test instruction generated can be chosen in a way that depends on the current state of the machine. Test scenarios that involve numerous instructions can be described.

The test-cases binaries are run on both the reference behavioural C simulator and the VHDL model.

All the tests are built according to the scheme:

- resource initialisation (registers,...)
- instructions under test execution
- write results in memory

The validation consists in comparing the resulting memory zones against the reference dumps that are produced by the reference simulator. This technique has several advantages: it stresses the design with actual instructions; it provides a coverage that is dependent on the quality of the test specification; and it is usable on all the validation platforms (simulation, acceleration, emulation), as well as on a real hardware platform, providing the capability to run the tests that have been used during the verification process on the actual silicon. However, some tests, targeted at exercising specific microarchitecural resources (IVP), require the forcing of internal states of the machine, which is not easily portable across platforms. But effort is put into using that kind of techniques only when biasing of chip-level tests fails to provide precise control over these resources.

3.4 Simulation Tools

VHDL models and testbenches are developed by Chameleon designers using a VHDL environment. VHDL is used for rather high-level descriptions, but these are still RTL and not fully behavioural.

As validation of a complex microprocessor requires a large number of cycles, RTL simulators are not performant enough (less than 10 clock-cycles per second) and faster implementations must be used to achieve the goal of running several billions of machine cycles before taping out. The techniques used on Chameleon are hardware acceleration and hardware emulation.

An HW accelerator is a dedicated machine that does event-driven simulations of a gate-level netlist. The performance obtained with such a machine is about 200 times faster than a software simulator, which is about 2 kHz.

HW emulation consists in mapping a gate-level netlist onto an FPGA network, which results in an actual programmable hardware implementation of the netlist. Here the performance is around 350 times the performance of an accelerator, ie around 700 kHz.

Both of them require a gate-level netlist as input. It is obtained from RTL VHDL code via synthesis. Although gates and not RTL statements are simulated, the objective remains the same: to run programs on the chip and to verify the correctness of the results.

In this context, the model of the chip is not sufficient for the verification, and some environment is also required : memory for handling the code and the results, interrupt generators, ... This environment is connected to the chip to run the tests. The model of the environment is driven by the same synthesis constraints as the actual design, as it has to be accelerated and emulated as well.

3.5 Tool-specific flows

Although the global methodology is the same whatever the simulation platform, there are some specificities depending on whether the target is a VHDL simulator, an accelerator or an emulator. The following figure details the possible paths.

3.5.1 Synthesis

The synthesis is targeted at a generic functional library. The same netlist can be used as input for both acceleration and emulation platforms. When entering the part of the flow that is specific to acceleration or emulation, each library cell is described using the vendor's library.

This verification methodology implies that one can obtain, from some kind of high-level description, a netlist compatible with acceleration and emulation. This means that the synthesis tool must be capable of synthesising VHDL code that is not really optimised for real circuit design, code with generics, redundant operators, complex procedure calls, etc..

With such a huge design, synthesis time is not negligible. In order to have the highest possible turn-around time, synthesis is performed on each block, independent of hierarchy, which saves synthesis time for unmodified blocks from one VHDL release to another, but entails very strict data management. It is also compatible with the parallelisation of synthesis jobs on several machines.

Synthesis is performed in batch mode with a set of script files to automatically generate the full model netlist. The synthesis is not focused on area or any other strong constraint, in order to keep synthesis time reasonable. The purpose of this synthesis being functional verification, there is no constraint on timings, and the use of a functional library (few cells, no timing informations) yields gains in synthesis time.

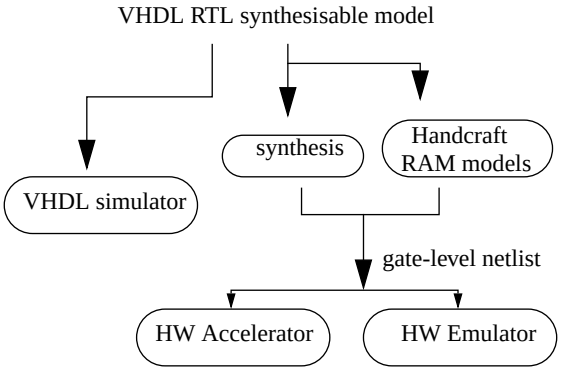

Fig.2. Accelerator - Emulator mapping

3.5.2 Hardware platforms

Acceleration and emulation require some extra work to be done on the netlist, particularly with RAMs. All the memories described in the VHDL specification, if synthesised as flip-flops, require a lot of resources (either generate events in acceleration or use emulation gates). Some structures would require such a long CPU time to synthesise that it would be prohibitive. It is worth extracting these RAMs and instantiating specific memory components provided by the acceleration/emulation vendors. These components rely on real memories embedded in the platforms.

In this context, it is sometimes an issue to map a memory in the design onto a different memory : as an example if a platform provides dual R/W port memories, and if a design needs a n-read/m-writes, n>2 and m>2, this becomes an issue. A simple way of multiplying read ports is to duplicate dual ported memories. But there is no simple way of increasing the number of write ports : it requires a division of the clock period into several sequential write operations as shown in the next figure.

Emulation brings specific difficulties : as it is a real hardware machine that basically connects gates, and as resources in terms of gate capacity and routing are limited, some care must be taken when mapping a netlist : as in silicon, routing such a complex design is not straightforward, and this operation requires the user to bear in mind the available resources and the needs of the design. As an example, let us consider an FPGA containing n truth tables and m IO pins. If one gate of the design has a fanin + fanout equal to m,

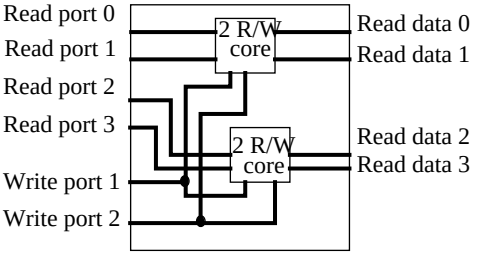

Fig. 3a. Multi read-port memory mapping

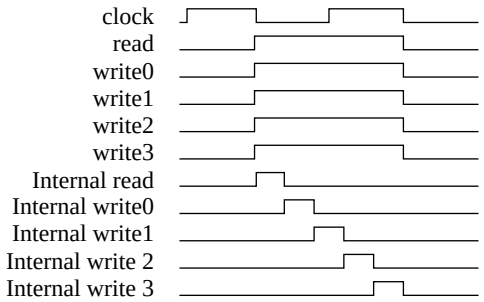

Fig. 3b. Multi write-port memory mapping: accesses serialisation

then only this gate can be mapped in the FPGA ! If it has fanin + fanout greater than m, then it has to be duplicated into several FPGAs until the right IO resource is obtained, leading to a very bad filling ratio of the machine. Tri-states are not emulated as tri-states : they are resolved with OR-gates, increasing the number of nets to be routed.

More generally, emulation requires the user to know the machine quite well, as it happens that the tools themselves do not always find a routable partitioning of the netlist. This leads to modification of the netlist logic hierarchy, modification which is driven by placement and routing issues to help the emulation compilation tools.

Due to these limitations, emulation is introduced into the verification process only when a good level of maturity and confidence is reached with acceleration.

3.6 Code Coverage

A code coverage analysis is done using a commercially available tool in order to make sure that any VHDL statement of the specification is exercised at least once during the regression test simulation. Using this code coverage facility, it is possible to have a very precise idea of the regions of code that receive little or no coverage. Additional tests can then be generated to increase this coverage. The designers and verification engineers collaborate to derive these ad hoc tests from the analysis of the coverage results, on a case by case basis.

4 Sequential verification

SGS-THOMSON Microelectronics has long recognised the value of formal verification within the silicon design process. Sequential proof tools can automatically prove sequential properties of complex control logic. A particularly fruitful area for the application of sequential property checking is in the verification of arbitration mechanisms and other protocol-based state machines. For instance, it is possible to prove absence of starvation in bus protocols and the correct implementation of coherency in cache protocols [7].

Temporal properties can be used to “simulate” a (possibly infinite) set of test programs. For instance, suppose we wish to verify that once a request line is asserted, then eventually the response line will be asserted. It potentially requires an infinite number of test sequences to verify this property exhaustively. However, using formal mathematical techniques it is possible to verify such a property for all possible test sequences, often within seconds.

Current property checking methods do not scale to the full size of modern designs. It is only possible to apply property checking to large designs if details of the design which are not relevant to the property can be abstracted.

4.1 Sequential verification tools

With the new generation of formal verification tools, it is possible to fully integrate formal verification within a design flow. Sequential proof techniques require a powerful Boolean engine and a robust interface to VHDL. Chameleon uses two sets of tools:

a. Vformal: a commercially available suite of tools developed by Bull and marketed by Compass. Two of the most important tools are fsmc, which compiles VHDL designs to a finite state machine format, and vprover, which proves that two VHDL designs are combinationally equivalent. For sequential verification, Chameleon uses fsmc; vprover is used in the circuit verification (Section 5).

b. Shadow: a tool developed within SGS-THOMSON [3]. This works as a back end tool with fsmc: it manages the manipulation of the FSM representations produced by fsmc. It performs two main functions:

- Abstraction. The tool is able to reduce the complexity of a design in such a way that parts of the design which are not relevant to the property can be removed. Given a property and a finite state machine, Shadow automatically removes those variables that do not influence any of the variables appearing in the property, and constructs the projection of the FSM onto the remaining variables. Then the property holds for the abstracted machine if and only if it holds for the original system.

- Sequential proof. The proof engine of Shadow verifies either that two FSMs are equivalent in terms of input-output behaviour, or that a property holds for a given FSM. A property can be expressed as an assertion in the original VHDL; for sequential properties that express the relationship of variables over a number of cycles, one can construct an automaton to represent the property. The proof engine is based on the Bull BDD library TDGLib. It uses both (i) algorithms based on an explicit construction of the transition relation, as in the SMV tool developed at Carnegie Mellon University [9], and (ii) state traversal algorithms based on a representation of an FSM as a vector of BDDs, one for the transition function of each state variable [6]. Algorithms of the latter type can achieve results when the transition relation is too large to be constructed.

If there are assertions in the VHDL, the output of fsmc includes BDDs representing the sets of states in which they are violated. The abstraction process also constructs BDDs representing the sets of states in the abstract FSM where the assertions are violated. If there are assertions to be checked, the proof engine determines whether any of the states in which the assertions are violated are reachable from the initial states of the system.

Recently, model-checking for the temporal logic CTL [5] has been implemented in Shadow, making it possible to check a wider range of liveness properties. However, in the work reported here, such properties have been checked using the SMV tool, after translating the FSMs into the SMV format.

4.2 Sequential verification flow

Sequential verification has been used in the validation of the VHDL specification. The areas to which this technique is applied have been chosen so as to reinforce simulation-based verification in cases where incorrect behaviour might only be shown up by particular sequences of signal values over many cycles. A number of crucial temporal properties have been identified, expressed either as VHDL assertions in the original VHDL, or as assertions in VHDL automata specifically representing the property, or as CTL formulae. These properties are checked at component level.

The VHDL for the component is compiled using fsmc. The resulting FSM, and the property to be checked, are then read into Shadow, which constructs a simpler abstract FSM that is adequate for the verification of the property. The property is then checked for the abstract FSM.

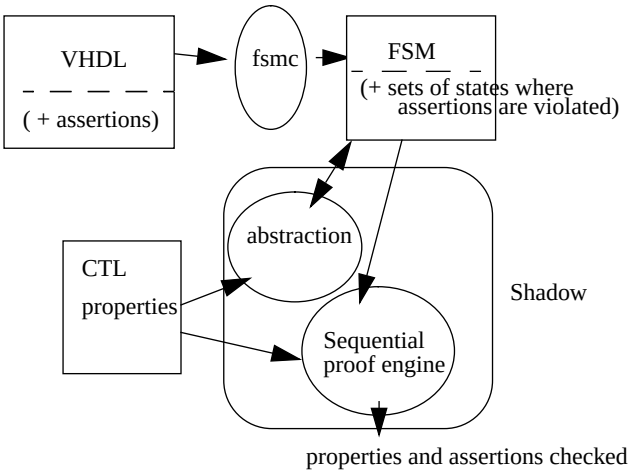

Fig. 4. Sequential verification flow

4.3 Tool performance

The most substantial example to which the techniques have been applied is a subcomponent of the unit that handles memory instructions. There are complex interactions betweeen the various subcomponents in this unit; the property to be checked was that the subcomponent in question cannot permanently send a stall signal to the buffer from which it receives its data.

The input to fsmc was a release of the Level 1 VHDL for the subcomponent, exactly as written by the designer. The construction of the FSM representation took 24 hours on a Sparc 10 processor. Memory requirements were 40MB. The FSM had 696 state variables (not including inputs and outputs). The abstraction process took 20 minutes; the abstract FSM contructed by Shadow had only 37 state variables. The construction of the transition relation and checking of the property in SMV took 25 minutes, using 8 MB of memory.

The property turned out to be violated. There were circumstances under which the stall signal can be permanently asserted; these depended on a particular pattern of the way that resources became available to the subcomponent. At the stage this work was done, these circumstances had not arisen in simulation, although the model had been in use for several weeks.

The property was checked on a new release and found to hold. The figures were similar, except that the abstract FSM had 55 state variables. The time for SMV increased to 28 hours, using 140 MB of memory.

Since the verification was carried out at component level, the validity of the property depended on assumptions about the environment of the component. The hardest part of the work of the verification engineer was the specification of an adequate and tractable model of the environment.

These results reflect the relative demands of the various stages, but other work on smaller components has been less demanding overall. For a set of 6 properties fully defining a preliminary version of a bus arbiter, typical times were 20 minutes for fsmc, less than 1 minute for abstraction (14 state variables reduced to 10), and 5 minutes for SMV. The reduction of the number of state variables from 696 to 37 in the memory unit example was one of the most striking results, but not exceptional: reductions of between 75% and 95% for large components are typical.

Using vectors of transition functions in Shadow, it has been possible to check assertions for FSMs with more state variables. Typically, these have been assertions that a component cannot get into some bad state. Some results are: 3 assertions for an FSM with 30 state variables checked in 3 seconds; 1 assertion for 357 state variables in 3 seconds; and 5 assertions for 507 state variables in 25 seconds. In the last two cases, it would have been impossible to build a BDD representation of the transition relation on our largest machines, although the properties were fairly simple, and the proofs did not involve large intermediate BDDs.

5 Verification of the circuit

Functional verification of the circuit uses combinational formal proof, providing an exhaustive comparison between the behaviour of the circuit and that of the specification.

Application of formal proof techniques to large industrial designs has been used for many years and has now reached a very high level of maturity [2], [8]. This is mainly due to, first, the general acceptance of this technology within the design community and second, the existence of robust and reliable commercial tools, such as VFormal, integrated with standard Hardware Description Languages.

In Chameleon methodology, formal proof is performed at block level between the level 2 VHDL specification of a block and a VHDL gate-level description automatically extracted from the layout of the block.

5.1 Transistor abstraction

A VHDL gate-level description is extracted from the transistor-level view of the block using a transistor abstraction tool internally developed by SGS-THOMSON Microelectronics. This tool, called Laybool, generates a gate-level description from the layout view. More precisely, it transforms a SPICE-like netlist of transistors (possibly hierarchical) into a VHDL dataflow description (keeping the same hierarchy).

The VHDL dataflow description contains the same nets as the transistor netlist; each channel-connected group is transformed into one or several VHDL dataflow statements. This description can be used as input to either a VHDL simulator or a formal proof tool.

The tool is completely generic in the standard cells/custom approaches used in the circuit design. It is capable of handling transistor strengths, pass transistors, precharge logic and other standard circuit design techniques. It does not use any sort of library structural models and is based upon functional boolean computations built upon Bull BDD library TDGLib.

The tool uses techniques similar to those described in [4], plus additional capabilities. Memory elements (latches) are recognized

as stable fix-points of feed-back loops. Oscillations are flagged and clock and reset conditions are automatically extracted. Tri-state drivers are recognized.

5.2 Block-level combinational formal verification

The formal proof tool used by Chameleon is VFormal, a commercial combinational prover from Compass. It is applied for block verification by comparison of the VHDL specification w.r.t to the extracted functionality (extracted from the transistor level using the transistor abstraction tool).

The following figure describes the complete process.

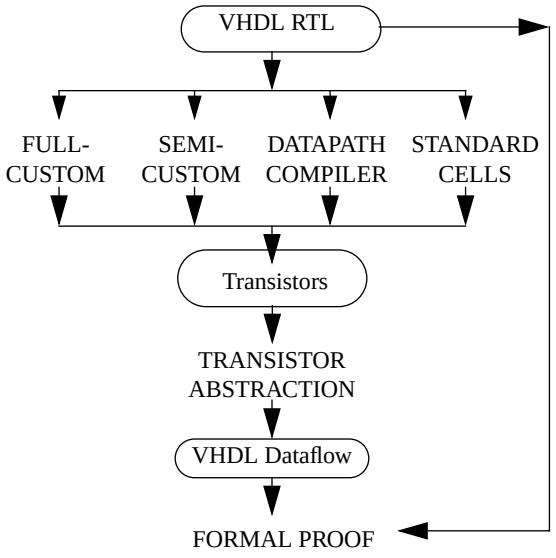

Fig. 5. Circuit verification flow

Application of this technique to a 64-bit adder datapath design gives the following figures: transistor abstraction is performed in a few seconds and formal proof of equivalence to the VHDL specification in a few minutes.

Application of formal proof requires that each block be decomposed into a suitable hierarchy. A typical block is a complex shifter made of 30 thousand transistors and decomposed into 13 sub-blocks; this block is formally proved equivalent to the VHDL specification in less than one hour.

Formal proof is applied on the vast majority of Chameleon blocks. Unfortunately, not all designs can be compiled using VFormal as the underlying BDD representation can become too large to manage, even on today's largest computers (the representation of a 16-bit multiplier requires more than a gigabyte of physical memory and more than a week of CPU time).

When formal proof fails, comparison of the extracted gate-level with its VHDL specification is done using simulation techniques. This verification is either partial or exhaustive, as it was done for the exhaustive co-emulation of a 16 bits multiplier, by running 4 billion vectors in less than one hour at 1 MHz frequency.

VFormal is also applied to prove the equivalence of the two hierarchical netlists, the VHDL specification on the one hand and the netlist extracted from the layout on the other hand.

6 Conclusion

As the size and complexity of the design increase, it becomes a strong requirement to put in place powerful verification methods. Within the SGS-THOMSON Microelectronics Chamelon project, several state-of-the-art techniques are used to address this complexity.

The starting point is an RTL VHDL description, on which the functional verification is fulfilled by running billions of machine cycles as tests. Acceleration and emulation technologies dramatically increase the power in terms of cycles simulated per second, and provide an acceptable turn-around time. Formal verification techniques are used to complement the verification of the specification wherever it is possible to use them. Combinational proof is in a state of maturity that enables one to rely almost entirely on it for circuit-level verification.

The main point of this methodology is to use advanced technologies such as acceleration, emulation and formal verification in the very first phases of the design rather than after physical design. The design implementation starts with a functional specification that can be used at each step as a reference, minimising the risk of facing huge functional issues just before tape out.

References

[1] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho & G. Shurek, "Test Program Generation for Functional Verification of PowerPC Processors in IBM", Proceedings of the Design Automation Conference, pp.279-285, 1995.

[2] D. P. Appenzeller and A. Kuehlman, "Formal Verification of a PowerPC Microprocessor", Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors, pp 79-84, IEEE, October 1995.

[3] G.Barrett, M.Belhadj, C.Berthet, A.McIsaac and F.Rocheteau, "The Application of Design Abstraction and Transistor Abstraction in an Industrial Design Flow", submitted to FMCAD, 1996.

[4] R.E. Bryant, "Extraction of Gate Level Models from Transistor Circuits by Four-Valued Symbolic Analysis. In Proceedings of the International Conference on Computer-Aided Design, pages 350-353, 1991.

[5] E.M.Clarke, E.A.Emerson and A.P.Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", in ACM Trans. on Programmimg Languages and Systems, 8(2), 1986.

[6] O.Coudert, C.Berthet and J.C.Madre, "Verification of Sequential Machines using Boolean Functional Vectors", Proc. of the Workshop on Applied Formal Methods for Correct VLSI Design, Houthalen, Belgium, November 1989, in Formal VLSI Correctness Verification, vol. II, North-Holland, 1990.

[7] A.Th.Eiriksson and K.L.McMillan, "Using Formal Verfication/ Analysis Methods on the Critical Path in System Design: A Case Study", in P.Wolper (ed.), CAV'95, Lecture Notes in Computer Science 939, Springer Verlag, 1995.

[8] C. Malley & M. Dieudonne, "Logic Verification Methodology for PowerPC Microprocessors", In Proceedings of the Design Automation Conference, pp. 234-240, 1995.

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