

Accelerating functional verification of multi-core processors using a VLIW application-specific instruction-set processor for virtual simulation

Benjamin Gerdemann

September 18, 2008

Abstract

There is a wide consensus that verification times for modern microprocessor designs have reached an unacceptable level. Two industry case studies from Intel and Motorola both cite time spent in verification as a major bottleneck to tape-out and estimates are that between 50% to 70% of engineering effort on a design is spent in verification, the majority of it in functional simulations. [2] [13] [5] We introduce a method to improve the functional simulation performance of large multi-core computer processor designs, by using a VLIW application-specific instruction-set processor (ASIP). The architecture is based on a large number of functional units computing simulation results in a distributed register file architecture allowing a large number of parallel operations. We show substantial performance improvements over software simulation giving designers more flexibility to prototype and test portions of their design rapidly.

1 Introduction

Verification is the process of determining that a design is correct. Like any complex system, a hardware design must be tested for correctness before being released. The general flow of hardware design is to move from abstract high-level models to a concrete transistor-level implementation while ensuring that each lower-level model is a correct implementation of the higher one. This method of development demands a high degree of confidence, because an error found in a higher-level model could force a dramatic restructuring of the lower-level implementations. In the worst case, a bug found after the tape out of the design could require an extremely expensive re-manufacturing of the hardware. “It is a well-known fact that the cost associated with debugging increases exponentially if the bugs are found at the later design stage. This means that a trivial design bug at the functional level design stage leads to critical defects in the market.” [9] Industry studies have shown that more man-hours are spent on verification of hardware designs than in

creating the design itself and that verification is typically in the critical path to tape out. Given the high cost of failure and the necessity of validating multiple models to each other, a large amount of time is spent in verification and this is a critical task in any hardware design.

There are two major techniques used to verify hardware designs: simulation and formal verification. In formal verification, the design is checked against a series of formal mathematical models. The design can then be proved to be formally correct against these models. In simulation, stimuli are applied to the design and the results are measured against the expected behavior. Additional assertion checks may be added to ensure that not only does the design achieve the expected result, but also that it doesn't violate any other properties during execution such as cache-coherency, bus contention, etc.

Both of these techniques have advantages and disadvantages. The clear advantage of formal verification, is that it is mathematically rigorous and when a design passes, it has been proven correct to the given constraints. Simulation only insures that for the series of tests run the design behaved as expected; it doesn't guarantee that in a case not tested, the design will be correct. Unfortunately, despite the clear advantage of mathematical rigor, formal verification fails to be a successful methodology in most practical cases. First, it is often difficult for a designer to define a complete list of constraints for any complex design. For example, in the functional verification methodology of the PowerPC 604 microprocessor Monaco et. al note "The architectural specification for most microprocessors is typi-

cally semi-formal – that is, it is not mathematically rigorous." rather "the model implementation itself comes closest to capturing mathematical rigor" and "a lack of mathematical rigor in the specification process combined with the cultural acceptance of the development method...makes simulation and emulation the de facto options for demonstrating processor model correctness." [11]

Finally, proving formal correctness is a binary operation. A design that passes 99.9% of its functional simulations is close to correct, but formal verification will still fail. This is important for project planners to understand how close a design is to completion. Given the restrictions of formal verification, it is typically used only on small, but complex parts of a design where mathematical constraints can easily be expressed, for example arbitration mechanisms, protocol-based state machines, or coherency of cache protocols. [4]

Industry case studies show that in practice the bulk of verification effort is spent in functional simulation. It is "common knowledge" [5] that at least half of engineering resources are spent in verification, [4] the bulk of it in functional simulation "since simulation is the only practical means which can handle the verification for all levels of a design, it is still the mainstream methodology used in industry by high-end microprocessor design projects." [13] For example, to verify the Pentium 4 design, Intel used a cluster of "several thousand machines" because the full-chip simulation performance was 3-5 simulation cycles per second running on Pentium III based machines. By tape out, they had verified 200 billion cycles in simulation which

was equivalent to only two minutes of real CPU time on a 1GHz CPU. [2]

There are a variety of methods for functionally verifying a design and the most common is software simulation. In software simulation, the hardware “code” is compiled to a software executable, much like a C compiler compiles software code into an executable. This is the most common and flexible solution. Any type of design regardless of size, complexity, timing or synthesizability can be simulated without modification and debugging is simple and straightforward. The problem with software simulation as noted above is that performance is poor.

In hardware emulation, a portion of the design is emulated on a hardware device instead of being simulated in software. In a typical example, the hardware design written in an HDL language, would be synthesized and placed onto an FPGA device. The test bench written either in software (typically C or C++) or behavioral HDL would execute on the software simulator and an interface would be constructed for the two to exchange data.

The major advantage of emulation is the increase in performance, which can be hundreds or thousands of times faster than software simulation. Because of this increased speed, it is then feasible to run longer more intensive tests than in software simulation, for example executing real software code in emulation or even booting an operating system.

Unfortunately, emulation places many restrictions on the design. First, the design must be synthesizable to be implemented in hardware. This requirements restricts the timing of the design being emulated and

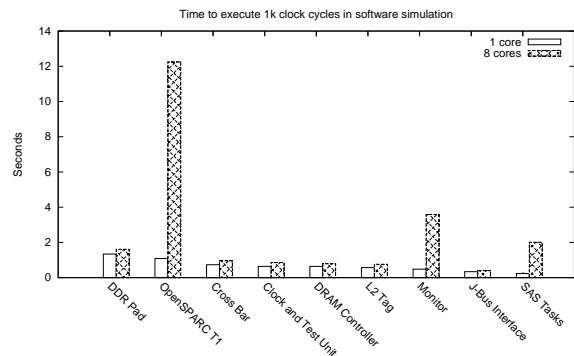


Figure 1: Software simulation profile

may require complicated partitioning if the design is too large to fit on a single hardware device. Additionally, portions of the design may need to be remodeled, particularly memories and caches, in order to be functional on an emulation platform. Finally, debugging an emulated design is tricky and the turn-around time to test corrections is long. Because of these problems, emulation environments are typically only deployed at the end of a project when the design is most static and the most possible to test.

Multi-core chips present designers with special challenges especially in the area of verification. For example, a profile of the software-only simulation of the OpenSPARC T1 processor from Sun Microsystems (figure 1 shows that as that as the number of cores in the design increases, the time spent simulating them in software increases in a roughly linear manner.

Unlike traditional single-core processors, increasing performance is directly correlated to increasing complexity. As the number of cores is increased, software cannot adequately simulate designs through the millions of cycles needed for verification. [8]

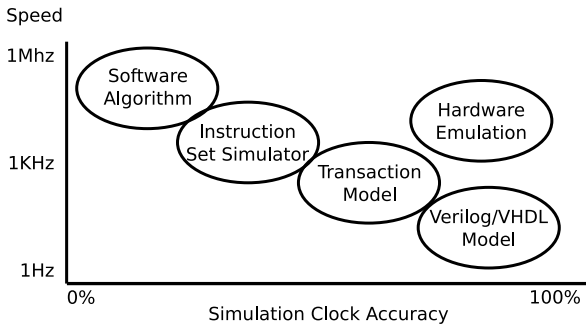


Figure 2: Trade-off between accuracy and simulation speed

It is clear that current methodologies and verification tools will not be able to cope with plans for one-thousand core designs. [3]

2 State of the Art

Every verification methodology has advantages and disadvantages. Some of the principal concerns are, the accuracy of the simulation, the visibility into the design during simulation and ease of debugging, the performance, and re-modeling effort. We will examine the state of the art in each of these areas.

Perhaps the the most important choice a verification designer has to make is the accuracy of the simulation model. As figure 2 shows there is an accuracy/speed trade off between various abstraction levels. The higher the level of abstraction, the faster the simulation, but the less accurate the model. As the figure shows, emulation methodologies have the potential to provide accurate simulations at high speeds. [12] A software algorithm model of a design can run at speeds into the megahertz, but provides no timing information

making it useful for verifying an algorithm itself, but not its hardware implementation. An instruction set simulator (ISS) mimics the behavior a of processor, by executing instructions and maintaining internal variables that match the registers of the processor. A transaction model simulates transactions at the bus-level interface. Timing information for the ISS and transaction models may be approximated by estimating the expected length of each instruction in the processor, but is not cycle accurate. Finally, an HDL simulation is completely cycle accurate, but also has the slowest performance.

A unique example of an ISS is described by Kim et al. [9]. In their proposed “virtual chip” methodology a software functional model is connected to a “pin signal generator” which is synthesized onto an FPGA placed into a target system. The target system is composed of actual hardware devices which allows the functional simulator to be tested in a real hardware environment instead of the typical software “system models.” This allows verification to proceed in a real hardware environment before gate-level implementations are even built. The “pin signal generator” acts as a transactor between the functional model and target system interfacing the two. This solution is an interesting way to get an early look at an architecture model in a real hardware environment, but are not cycle accurate and is concerned with increasing verification coverage rather than speeding up simulations.

Several transaction-based architectures have been proposed for functional simulation. [10] [7] Instead of modeling the bus interface directly, “transactions” are passed

between the test-bench and hardware being tested. The transactions encapsulate both data and synchronization information. Using transactions instead of a direct bus-interface reduces communication overhead, allows for the designer to use a flexible abstract interface and reduces the number of synchronization points. Typically transactional interfaces have been developed between a test-bench written in a higher-level language such as C or C++ and interfaced to the DUT, but recently SystemC has introduced the concept of transactional models allowing them to be embedded inside an HDL design.

Advantages of transactional models are their ease of portability between different environments including simulation and emulation, their ability to interface vastly different languages such as C/C++ and HDL and the elimination of performance bottlenecks by reducing communication overhead and synchronization points. Their disadvantages are that they typically require the designer to develop a “transactor” model to interface between the test-bench and DUT. This transactor may be non-trivial to build and it is possible that the transactor will alter the simulation itself distorting results and possibly reducing verification coverage or introducing its own bugs. Additionally, transaction level models are not cycle-for-cycle accurate and require labor-intensive manual partitioning.

In [7] a method of synthesizing the behavioral test bench is developed by implementing synthesizable transaction units into the hardware to run in parallel with the DUT. Each transaction unit has its own ROM and RAM which is used to store the protocol instructions to be executed as well as

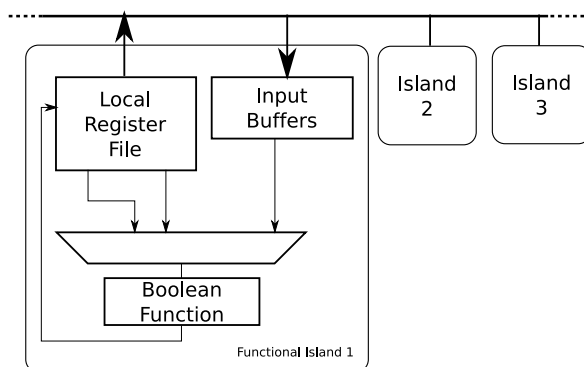


Figure 3: Co-emulation architecture

the results. A synchronization block is built to control the execution of each transaction unit and an automated method is developed to generate each test bench. By synthesizing the DUT and the test bench, the authors were able to show speedups from 625 to 985,000. The disadvantage of this approach is the re-modeling required to map the test bench onto hardware and the restrictions imposed on the test scenarios themselves so that they can be executed in hardware.

3 Methodology

Our proposed solution to the simulation performance problem compiles the hardware design into very long instruction words (VLIW)s to be executed on an application specific instruction set processor (ASIP). The architecture is based on a distributed register file system as shown in figure 3. The typical data-flow graph for a hardware design is massively parallel which is ideal for acceleration using an architecture with many independent functional units. Each functional unit can calculate any arbitrary

boolean operation and has its own register file to read and store results. A cross-bar or bus interface is also connected between the register files to forward data between the functional units.

The micro-architecture of the ASIP can be adjusted by various parameters. These include the number of functional units available (W), the depth of the individual register files (R), the width of bus connecting the register files (I) and the bit-size of the boolean operations (L). Once a design's parameters have been fixed, the ASIP can be synthesized and placed into hardware. A compiler then converts the gate-level hardware design into VLIW instructions for execution on the ASIP given the parameters of the architecture. Examples below show the performance effects of each of these parameters

- *Number of functional units (W)* More functional units allow more boolean operations to be computed in parallel
- *Depth of the register file (R)* Larger register files can store more results and increase the utilization of the functional units by reducing stalls in the pipeline
- *Width of the bus between register files (I)* A larger bus allows more data to be passed between register files, reducing the number of stalls
- *Bit-size of the boolean operation (L)* The more complex the boolean operations each functional unit can calculate, the less computations need to be done

Since the architecture is completely deterministic with no data-dependent branches, cache-misses, interrupts, etc. the compiler can return the exact number of cycles and memory required for a given micro-architecture. This allows the designer to rapidly experiment with different parameters to find values that give the maximum performance while still allowing the ASIP to fit into the hardware and meet timing requirements. More research needs to be done in this area, but in our experiments a ratio of 1:2:2 between W , I and R leads to a reasonable trade off between area and performance. For example, 32 functional units each with a 64-deep register file and a 64-bit bus connecting the islands.

The advantage of this methodology over an emulation-style architecture, is that once the architecture is fixed, it need only be synthesized and placed into the hardware once. Further revisions of the design are compiled in software reducing the simulate-debug-simulate turn-around time significantly. Our un-optimized compiler compiles around 15,000 gates per second. Additionally, this means that the size of the hardware for simulation is not dictated by the size or synthesizability of the design; unlike in emulation, there is no requirement the synthesized design fit completely in hardware. In fact very large multi-core designs that could require many hardware chips to emulate, could be simulated on a single FPGA. With our technique, increasing the size or number of hardware chips for simulation has a linear relationship on performance.

This methodology also permits both a traditional vector-style test bench or a more

complex dynamic software-controlled environment. In the vector-style approach, the test vectors are fixed and pre-computed ahead of time. To execute in this mode, the test vectors can be loaded into the hardware memory and fetched directly by the functional units with no further interaction from the software. The intermediate results can be snooped on as they are passed between functional units on the bus. If a more dynamic test bench is desired, rather than merely snooping on the bus, the software can inject modified data into the simulation based on the intermediate results.

4 Results

In table 1 we compare the expected speed of an ASIP synthesized onto an FPGA running at 100MHz versus the measured speed of the same net list simulated in software with a commercial simulator.

Table 1: Time in seconds to simulate 100k cycles of a design in an ASIP versus software

Design	ASIP32	Software	Speedup
s38417	0.33	50.3	316
DSP	1.46	277	379
leon	29.36	7,140	487

Comparing the performance of two differently sized ASIPs, one with 32 functional units and the second with 64 functional units in table 2 we can see that the performance scales linearly as we double the size of the processor.

The designs benchmarked are from the 2005 IWLS Benchmark suite from the Cadence Berkeley laboratories.

Table 2: Time in seconds to simulate 100k cycles of a design in two differently sized ASIPs

Design	ASIP32	ASIP64	Speedup
s38417	0.33	0.16	2.06
DSP	1.46	0.73	2.00
leon	29.36	14.68	2.00

5 Conclusion and Future Work

Our ASIP based methodology for simulation has the potential to provide substantial speedups versus a software simulation while avoiding most of the complexities involved in a typical emulation platform. The flexible micro-architecture of the ASIP allows it to be implemented on a wide variety of hardware giving designers the flexibility to select an architecture optimal for their design. Additionally, unlike emulation, there is no minimum hardware size required to simulate a design, instead performance scales linearly with the size the hardware. In a typical emulation system, the performance is fixed by the frequency of the emulating hardware.

In the future, we plan to analyze the micro-architecture of the ASIP in more detail to determine which configurations provide the most performance improvement while maintaining a balance with area and timing requirements. As part of this work, we will build an instruction set simulator for the ASIP that will allow for a more detailed performance analysis of the architecture. Finally a complete hardware implementation will be created as a proof-of-concept and to study in more detail the fea-

sibility of the methodology in an actual design.

References

- [1] Ghazanfar Asadi, Seyed Ghassem Miremadi, and Alireza Ejlali. Fast co-verification of hdl models. *Microelectron. Eng.*, 84(2):218–228, 2007.
- [2] Bob Bentley. Validating the intel pentium 4 microprocessor. *dsn*, 00:0493, 2001.
- [3] Shekhar Borkar. Thousand core chips: a technology perspective. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 746–749, New York, NY, USA, 2007. ACM.
- [4] Francoise Casaubieilh, Anthony McIsaac, Mike Benjamin, Mike Bartley, Francois Pogodalla, Frederic Rocheteau, Mohamed Belhadj, Jeremy Eggleton, Gerard Mas, Geoff Barrett, and Christian Berthet. Functional verification methodology of chameleon processor. *dac*, 00:421–426, 1996.
- [5] Joanne DeGroat, Duane Marhefka, Jennifer Stofer, Lyle Hanrahan, Bruce Wile, and Fusun Ozguner. Teaching future verification engineers: the forgotten side of logic design. *dac*, 00:253–255, 2001.
- [6] James Gateley, Miriam Blatt, Dennis Chen, Scott Cooke, Piyush Desai, Manjunath Doreswamy, Mark Elgood, Gary Feierbach, Tim Goldsbury, and Dale Greenley. Ultrasparc-i. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 13–18, New York, NY, USA, 1995. ACM.
- [7] Renate Henftling, Andreas Zinn, Matthias Bauer, Martin Zambaldi, and Wolfgang Ecker. Re-use-centric architecture for a fully accelerated testbench environment. *dac*, 00:372, 2003.
- [8] Ian Kaplan, Michael Bershteyn, Paul Vyedin, and Jerry Bauer. A reconfigurable logic machine for fast event-driven simulation. *dac*, 00:668–671, 1998.
- [9] Namseung Kim, Hoon Choi, Seungjong Lee, Seungwang Lee, In-Cheolo Park, and Chong-Min Kyung. Virtual chip: making functional models work on real target systems. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 170–173, New York, NY, USA, 1998. ACM.
- [10] M. Kudlugi, S. Hassoun, C. Selvidge, and D. Pryor. A transaction-based unified simulation/emulation architecture for functional verification. *Design Automation Conference, 2001. Proceedings*, pages 623–628, 2001.
- [11] James Monaco, David Holloway, and Rajesh Raina. Functional verification methodology for the powerpc 604 microprocessor. In *DAC '96: Proceedings of the 33rd annual conference on Design automation*, pages 319–324, New York, NY, USA, 1996. ACM.

- [12] Yuichi Nakamura, Kouhei Hosokawa, Ichiro Kuroda, Ko Yoshikawa, and Takeshi Yoshimura. A fast hardware/software co-verification method for system-on-a-chip by using a c/c++ simulator and fpga emulator with shared register communication. *dac*, 00:299–304, 2004.
- [13] Qichao Richard Yin and Jen-Tien Yen. Multiprocessing design verification methodology for motorola mpc74xx powerpc microprocessor. *dac*, 00:718–723, 2000.