A SoC simulator, the newest component in Open64 Report and Experience in Design and Development of a baseband SoC

Wendong Wang, Tony Tuo, Kevin Lo, Dongchen Ren, Gary Hau, Jun zhang, Dong Huang
{wendong.wang, tony.tu, dongchen.ren, jun.zhang, dong.huang}@simplnano.com
{lokaiman, waternoose}gmail.com
SimpLight Nanoelectronics Ltd., Beijing China 100088

Abstract

First silicon success, total system performance and time to market are three challenges in SoC design. Hardware/software co-design approach with a good simulation environment is usually used to address these challenges. In this paper we report a SoC simulation infrastructure which is used for complex SoC design. We accomplished the three big challenges with the help of a simulation tool set. Experience on design of a baseband processor based on the simulation environment is also presented. We show how we start from scratch, produced a product ready SoC within two years and achieve first silicon success with our production tape out. This system simulator has been open sourced as part of Open64 source tree in the summer of 2008. With this simulator, the Open64 is one step closer in becoming a complete toolchain suitable for new processor and SoC development.

1. Introduction

First silicon success, total system performance and time to market are three challenges in SoC design. Hardware/software co-design approach with a good simulation environment is usually used to address these challenges. With a complete set of simulation tools, designers can run benchmarks and/or targeted applications to experiment with hardware design at very early stage of design cycle. At the same time software programmers can use these tools to develop and test software before the real hardware becomes available [1, 2].

Application specific instruction set processor (ASIP) is a popular choice as a compromise between ASIC and general purpose processors (GPP). ASIP is programmable, their architecture is tailored for a certain application domain, such as communication, audio or video processing. Their performance requirements are usually very rigid (e.g. video codec processor). This requires target flexible instruction set simulator and compiler to assist in ISA design as well as micro-architecture design.

This paper reports our experience in design and implementation of a SoC simulation tool set for SL1, Simplnano's first SoC. The simulation tool set, named SoC SIMulator (SSIM) [3], is designed with the following goals: (1) ISA and SoC architecture design and verification; (2) early SoC system bring up and RTL co-verification; (3) early system software stack bring up and application software development and testing.

SSIM has been used for designing of SL1 which is a baseband SoC processor. SL1 is designed to meet the processing performance requirements for common 2G/3G, such as GSM/GPRS, EDGE and TD-SCDMA mobile handset devices. The SoC architecture adopts a hardware multi-threaded design that is based on a single programmable engine with application specific fix-function accelerators. All instructions of SL1 core are designed to make it run wireless application specific work load efficiently. With the help of SSIM, we delivered the SoC within two years of design-development cycle. And we were able to run MPEG4 decoder on top of a RTOS within hours of the chip came back from a production tape out.

The rest of the paper is organized as follows. Simulator related work is discussed in section 2. Section 3 gives detail description for SSIM and the use for hardware/software coverification. Retargeting and speed of SSIM are also discussed in section 3. In section 4, we concluded our work with projection for future work.

2. Related work

There are lots of simulators used for architecture research or design exploration. SimpleScalar [2], ASIM [5] and Skyeye [7] concentrate on accurately modeling the processor architecture design and they are cycle accurate. And other simulators which designed for architecture verification as well as software development such as FAST [4], are functional accurate. There are also full system simulators capable of running commercial workloads for an unmodified operating system, such as SIMOS [8], Simics [5] and Skyeye [7].

But all of them have one disadvantage, which leads us to develop a new simulation environment. They are not designed for hardware/software coverification in SoC system design process. Here coverification means: (1) same code modules can be shared by SSIM and C-model; (2) SSIM, FPGA and real chip can be verified with the same test cases and targeted application binaries.

3. SSIM

Figure 1 presents the kernel components of SSIM. Both system and user-level programs can run on this full-system simulation environment. There are three main parts for SSIM infrastructure: core instruction set simulator (ISS), SoC simulator and target dependent performance simulator.

Instruction set simulator in SSIM can run standalone or as one part of SoC simulator. Both modes use same codes modules as can be seen from figure 1. The simulator runs as a bit accurate interpreter for applications which are developed for the target system. It fetches instructions from cached memory, decodes it into internal format, executes the instruction, and updates register files and/or memory. Further more, ISS and C-model, which is used for hardware verification, can share some components when this design pattern is used. This makes SSIM effective for hardware/software coverification and it is the motivation that we design SSIM although there are many simulators can be used.

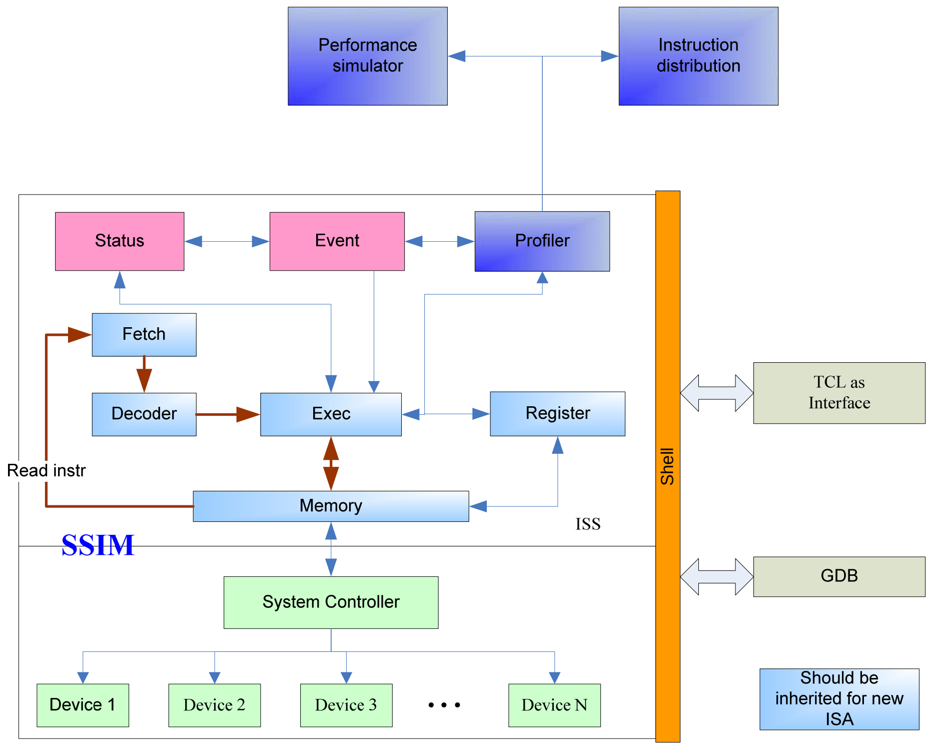

Figure 1. Key components in SSIM

Apart from ISS, SoC simulator includes registered devices/peripherals. Considering simulation speed, we emulate devices using transaction level accuracy, which means the simulator handle each interaction to the device as a unit: the device is presented with a request, computes the reply, and returns it in a single function call. Only registered devices can be accessed in runtime. SoC simulator will throw error message when unregistered devices are accessed by application. This can reduce simulation overhead for some applications which do not use all emulated devices.

Finally, there is a target dependent performance simulator, which models the internals of the target system and it is cycle accuracy. The SoC simulator implements what programmers see, but performance simulator implements what programmers not see and what hardware designers see. The decoupling method makes SSIM fast enough for software development and makes performance simulator accurate enough for hardware design. Note that the performance simulator is not open sourced in the Open64 source tree.

3.1 Coverification

In today's SoC design processes, verification phase has become the major part, not only because it is time consuming but also due to the increasing complexity in both hardware and software. Traditional SoC design-develop cycle is: Architect mapping partitions of SoC design into hardware and software components, and the specifications are handed to the hardware and software teams for implementation. The hardware team implements hardware portion of the design in Verilog or VHDL, using hardware simulators for verification. The software team develops software modules in assembly, C, or C++ languages and uses ISS to test the software. Many problems can arise during the system integration process. The problems are due to such things as misunderstanding of specifications, incomplete interface definitions, and late design changes.

Hardware/software coverification method eliminates these integration errors through moving integration phase early. The coverification occurs in all system design phases: RTL module verification, full-chip simulation, FPGA verification and chip verification.

Our coverification approach is carried out in three steps: (1) Develop SSIM and use test cases to verify it. Meanwhile, run typical applications under SSIM to collect performance data for design choices. (2) Use small test cases to verify RTL modules with the help of SSIM and develop target applications on SSIM. (3) Use same unit test cases and applications to verify the correctness of FPGA and real chip. The approach is illustrated in Figure 2 and further discussed below.

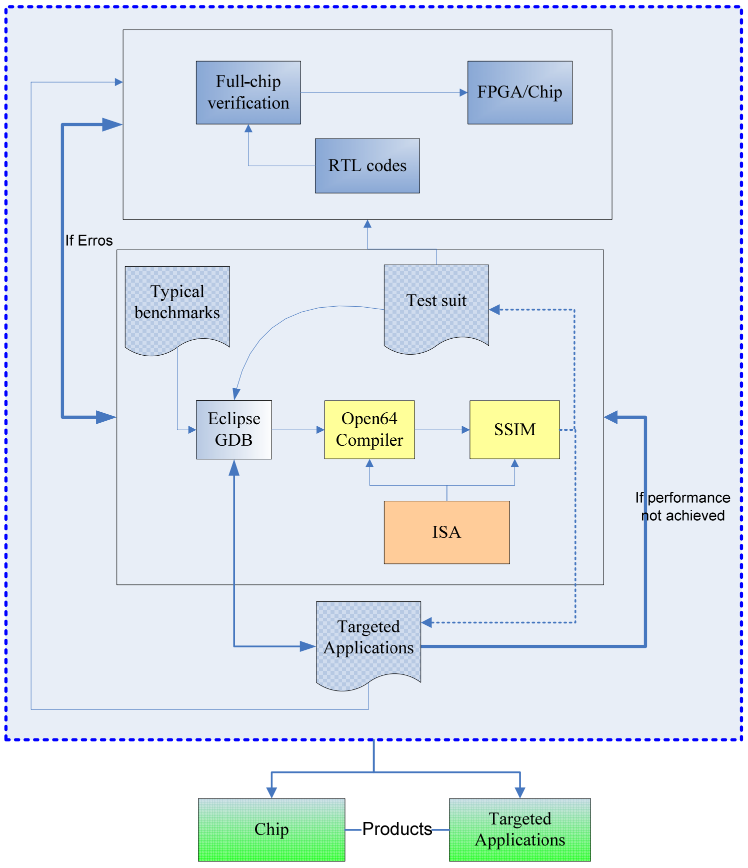

Figure 2: Coverification with the help of SSIM

After SSIM is verified by small test cases, targeted embedded applications are developed with the help of SSIM. At the same time, system designer could modify architecture (include ISA and micro-architecture) of SoC if performance of targeted application does not match the requirement. Eclipse and GDB is supported to reduce difficulty for embedded software development as shown in figure 2. SSIM parses the command line, if needs to work in the debug mode, a thread for the debugging system is forked.

To verify separate RTL modules, small test cases are issued by testbench through PLI interface to SSIM, The result is compared with result of RTL codes. For full chip verification, more unit tests are designed that includes carefully chosen subset of devices for easy triaging and pinpointing of root causes.

After verification at the RTL-level has been performed, the verification process typically continues on a prototype such as FPGA or full-chip RTL simulation. The applications and test suit developed in above stage can be used to test the final prototype system without modifications (except timing modules). If the verification failed, specification need to be checked by software and hardware engineers to make sure it is well understood. In this phase, some hardware design errors can be found and located through the help of SSIM.

Figure 3: A Packaged SL1 SoC

With this approach, same test cases and applications are used for RTL modules, FPGA and real chip verification. So, time for SoC verification can be reduced dramatically. When correctness verification for real chip passed, both chip and target system applications are ready for market. We use this approach to develop a baseband processor named SL1, shown in figure 3.

```
class AtomRegister : public BaseRegister<AtomInstr> {

private:
        WORD _gpr[REG_GPR_SIZE];
        WORD  _SP;
        ...
public:
        AtomRegister(ProcessStatus<AtomInstr>& status, UINT thid);

        WORD getSP(void){
                return _SP;
        }

        WORD getGPR(UINT32 index) { return _gpr[index]; }
        void setGPR(UINT32 index, WORD data) {
                _gpr[index] = data;
        }
        ...
}
```

Figure 4: Register class for Atom platform

3.2 Retarget

SSIM are written in C++. The inheritance of C++ made it easy to add new ISA. Figure 4 give one example which realizes register class for Atom platform in SSIM.

The interfaces in SSIM are well defined. Figure 1 gives internal structure for SSIM and its interfaces, where each component is defined as one class. Common interfaces are defined as virtual functions, and each target needs to inherit from base classes to realize these virtual functions.

An instruction set table (figure 5), which includes assemble name, decode function, execution procedure and disassembler for each instruction is defined in SSIM. New table need to be added if a new ISA is introduced into SSIM.

As we stated in section 1, we use two simulators: function simulator which implements the part which can be seen by programmer, and performance simulator which implements the parts cannot be seen by programmer. This decoupling implementation makes SSIM very flexible to retargeting.

```
{"mvfc", mvfc_id, 0, EIG_alu, &SL1Decoder::decodeCore04_06, &SL1Exec::execMvfc, &SL1Disasm::disasmCore04_06},
{"mvtc", mvtc_id, 0, EIG_alu, &SL1Decoder::decodeCore04_08, &SL1Exec::execMvtc, &SL1Disasm::disasmCore04_08},
{"nor", nor_id, 1, EIG_alu, &SL1Decoder::decodeCore04_02, &SL1Exec::execNor, &SL1Disasm::disasmCore04_02},
{"setlt", setlt_id, 1, EIG_alu, &SL1Decoder::decodeCore04_02, &SL1Exec::execSetlt, &SL1Disasm::disasmCore04_02},
{"setltu", setltu_id, 1, EIG_alu, &SL1Decoder::decodeCore04_02, &SL1Exec::execSetltu, &SL1Disasm::disasmCore04_02},
{"shra", shra_id, 1, EIG_alu, &SL1Decoder::decodeCore04_02, &SL1Exec::execShra, &SL1Disasm::disasmCore04_02},
{"sub", sub_id, 1, EIG_alu, &SL1Decoder::decodeCore04_02, &SL1Exec::execSub, &SL1Disasm::disasmCore04_02},
{"mc.r.eq", mcreq_id, 1, EIG_alu, &SL1Decoder::decodeCore04_04, &SL1Exec::execMcreq, &SL1Disasm::disasmCore04_04}
```

Figure 5: ISA table example for SL platform in SSIM

A good compiler is needed for retargeting and performance evaluation in ASIP. The Open64 compiler was chosen for its machine independent optimization so that once retargeted, the generated code will be of high performance quality. Then accurate performance data which collected can be used for architecture design.

3.3 Simulation speed

In most cases, instruction processing occupies most of the execution time for simulation, and programmer use instruction simulator (or SoC simulator) to develop software before real hardware becomes available. Thus, speed is very important for SSIM.

First, template is used to accelerate SSIM so that most references are solved at compile time.

Second, as described in above section, we emulate instructions in SSIM as it executed in hardware. Each instruction is decoded from the binary program and is passed to an appropriate simulation routine, which interprets the instruction and correspondingly updates the simulator status. So a technique named decoded instruction caching is used to speed up the performance. SSIM allocates one cached page and decodes all instructions belongs the page when the first instruction in a page is accessed. This method reduces a lot of preparations needed to fetch and decode of instructions one at a time. It also eliminates unnecessary decoding overhead when executing loops. SSIM provides one parameter which can be used to control number of decoded cache pages. And page size can also be configured to match page size of host system in order to get maximum performance.

Through these optimizations, SSIM can reach tens of million of instructions per second, which is sufficient to execute large real embedded workloads as can be seen from table 1. The data is obtain on machine with Intel core2 CPU (6600, 2.4G, single thread).

	EDGE Equalization	**AAC decoder**	**MPEG4 decoder**	**Matrix Decomposition**
With instruction caching	17060748	16243558	15879146	16826678
Without instruction caching	9748999	9163032	9283193	9707699

Table 1: Speed of SSIM with/without instruction caching (instructions per second)

4. Summary

We reported some preliminary results and illustrated, through the process of designing a real baseband processor, how SSIM achieve its objectives: efficient for hardware/software coverification, easy to retarget and acceptable simulation speed.

As future work, we plan to add one flexible machine description language (MDL)[9, 10] for SSIM, through which SSIM can be retarget automatically as shown in figure 6. Furthermore, the current SSIM does not simulate bus traffic, hence it is not suitable to study SoC system performance issues such as power usage, bus contention etc. This will be our future goal.

Acknowledgments

The author would like to acknowledge Sun Chan for his wisdom that directs us in SSIM design.

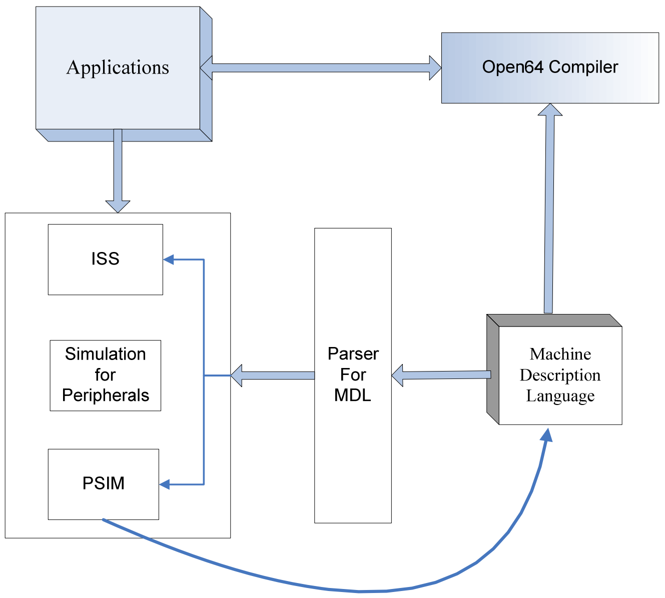

Figure 6: ASIP design flow with MDL

5. Reference

[1] M. Heinrich, D. Ofelt, M. Horowitz, J. Hennessy. “Hardware/Software Codesign of the Stanford FLASH Multiprocessor”. In Proceedings of the IEEE Special Issue on Hardware/Software Co-design, Vol. 85, No. 3, March 1997.

[2] T. Austin, E. Larson, and D. Ernst. “Simple Scalar: An Infrastructure for Computer System Modeling”. IEEE Computer, 35(2): 59–67, Feb. 2002.

[3] http://svn.open64.net/svnroot/open64/sim/fsim

[4] C. Juan, Z. Weirong, H. Ziang, G, GuangR. “FAST: A Functionally Accurate Simulation Toolset for the Cyclops64 Cellular Archi tecture”. Workshop on Modeling, Bench marking, and Simulation (MoBS2005).

[5] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. “Simics: A full system simulation platform”. IEEE Computer, 35(2):50–58, February 2002.

[6] J. Emer, P. Ahuja, E. Borch, A. Klauser, etc. “Asim: a performance model framework”. IEEE Computer, 35(2):68–76, Feb. 2002.

[7] http://www.skyeye.org/index.shtml

[8] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. “Using the SimOS machine simulator to study complex computer systems”. ACM Transactions on Modeling and Computer Simulation, 7(1):78–103, January 1997.

[9] http://www.retarget.com/products/whatisnml.php

[10] Eric Schnarr, Mark Hill, and James Larus, "Facile: A Language and Compiler For High-Performance Processor Simulators," in the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI01), Snowbird, Utah, June 20-22, 2001.