System-Level Modeling of DSP and Embedded Processors

Vojin Živojnović
AXYS Design Automation, Inc.
Irvine, CA 92606, USA
vz@axys.de

Chris Schläger, Joachim Fitzner
AXYS GmbH
Herzogenrath, D-52134, Germany
cs[jf]@axys.de

Abstract

High complexity and development costs of processor-based DSP and embedded designs permanently force the hardware and software designers to develop and intensively use processor abstractions in form of abstract processor models for specification, design, and verification of processor hardware and software. Based on the SuperSim processor modeling technology we have developed a new powerful methodology that can be used in an broad spectrum of applications ranging from DSP C compiler design over assembly code development to HW/SW co-simulation with HDL simulators.

1 Introduction

The trend of processing power hungry DSP and embedded applications with their complex implementations under shrinking time-to-market conditions continues. A typical current-day electronic system implementation includes a number of microcontroller and DSP cores fully loaded with code, memories and dedicated logic, and interconnected through shared memories and busses. At the same time, advances in VLSI technology offer single dies with hundreds of millions of transistors allowing complete systems-on-chip to be easily manufactured. The arguments of lower power consumption, size and costs prevailed and resulted in an industry-wide undisputed acceptance of this new opportunity introducing the system-on-chip.

However, significant concerns related to the system-on-chip design methodology are emerging at both antipodes of the abstraction spectrum. Whereas the issue of high integration at the submicron level received significant attention, the impact of system-on-chip technology revolution on the embedded software design and HW/SW co-design and verification is still insufficiently explored. Especially the design verification is a problem in real system-on-chip designs. A system-on-chip is much more than a functionally-equivalent and downscaled version of a standard printed circuit board (PCB) with discrete components. Integration on a single chip reduces our ability to observe and control events in the implementation and makes the divide and conquer approach impossible. It is increasingly hard to provide instruments giving us the full observation of intra-chip signals and full control over its resources. In a single chip solution the probes of the logic analyzer cannot be connected to the signals interfacing the components, neither in-circuit emulators can be inserted instead of the processors. The costs of providing observation and control through implementation of testing pins and dedicated interfaces tend to grow with the number of processors on chip.

One common alternative way is to develop a prototype realized as a PCB with the same components as the system-on-chip. Our experience has shown that this approach has significant drawbacks. First, a handful of chips has to be manufactured and tested with the only purpose of prototyping. Even with the chip fabrication facility in the backyard this process is time consuming and costly. Second, the PCB has to resemble the final system-on-chip which requires the architecture to be fixed quite early. Experience shows that with the constantly changing requirements and priorities of the embedded system market this task becomes very hard, almost unsolvable. Actually, the design team ends up in developing two or more complete systems instead of one.

The second alternative is to apply emulation with the help of powerful FPGA-based emulation systems. Emulation-based prototyping has shown to be useful for the development of general purpose processors. In the case of embedded systems, there are again two evident drawbacks. First, to achieve an optimal mapping, significant skills have to be acquired within a short period of time. With the high dynamic of new processor designs produced each year it is often impossible to invest in lengthy preparation of the verification process. Second, the costs for a complex system-on-chip with a large number of gates are too high for the embedded industry to justify the investment. The arguments provided above favor the simulation approach where the model of the whole system is developed in software and run on the workstation. The advantages of simulation for

system-on-chip verification are shorter model design time, high flexibility for model changes, full observability and controllability of the design, and more freedom in selecting the model precision. This allows the model-before-silicon approach to be followed allowing tool and software designs to start long before the actual chip is produced.

The main drawbacks of the simulation approach are speed and model reliability. The execution speed of standard cycle-based ISA simulators is in the range of tens thousands of instructions per second on 300 MIPS hosts. This is much lower than the typical speed of emulators or prototypes. Simulation of complete systems-on-chip introduces an additional slowdown. The ways to remedy the speed problem are compiled simulation and cycle-based ISA modeling in standard programming languages, like C or C++.

A device model is reliable if under a certain precision provides a faithful description of the underlying object. In order to produce reliable system-on-chip models it is important that the hardware designers keep updating the higher level models as they change lower level models. For example, if the gate- level model of the design is modified without properly updating the corresponding RTL model, the latter becomes definitely unreliable to serve as a reference for the higher level HW/SW co- verification, or ISA model. Although the same issue influences also FPGA-based emulation, it hurts especially the simulation. The solution to this problem is to adopt a methodology of processor hardware development which will guarantee proper back annotation of higher level model after lower level model updates. Despite of both drawbacks we believe that driven by the time-to-market issues the demand for simulation-based verification tools and processor models will grow in the future.

This paper provides a review of the previous work on system-level DSP and embedded processor models and simulation, introduces a new application driven taxonomy and discusses the use of compiled simulation technology to bridge the gap between HW- and SW-oriented processor models.

2 Previous Work on Processor Modeling and Simulation

Processor simulators such as instruction set simulators are standardly supplied with off-the-shelf or in-house DSP processor. They enable comfortable debugging through controlled program execution and provide visibility of processor resources necessary for code development. All commercially available instruction set simulators use the interpretive simulation technique. Their main disadvantages are the low simulation speed (2K-20K insns/s according to [1]) and the inability to be extended by the user or retargeted.

Processor models are also inevitable components of HW/SW co-design environments. The Ptolemy HW/SW co-design environment [2] uses the RTL circuit simulator Thor [3] for simulation of programmable digital signal processors. A stand alone behavioral simulator for the DSP56000 processor is interfaced to the Thor environment by socket connections. The interpretive DSP56000 simulator is able to deliver pin-exact simulation while executing the code.

The HW/SW co-simulation environment reported in [4] permits instruction- and phase-accurate HW/SW co-simulation of the Mitsubishi M16 microprocessor. The simulation speed varies between 4.000 and 50.000 insns/s on a 87 MIPS machine (Sparc 10) depending on the selection between phase- and instruction-accurate simulation.

Kra [5] attacks the problem of slow and uncomfortable software debugging in hardware design verification environments. His approach is based on tracking of hardware-software interactions along the simulation, and later reproduction within the software debugger.

Other cosimulation approaches have been proposed in [6], [7], [8], [9] and [10].

The state-of-the-art commercial HW/SW co-verification tools, like [11] and [12] use a combination of interpretive instruction-set models and bus functional wrappers to integrate the models in HDL environments and provide full software debugging. At the same time, system-level simulation tools, like [13] directly interface the instruction-set processor models supporting full software debugging.

3 Application-Oriented Processor Modeling Taxonomy

Although the classification of models according to the temporal (instruction, cycle, phase) and spatial (gate, RTL, instruction, statement) accuracy, like those suggested by the RASSP Taxonomy Working Group [14] characterizes the processor models well, we believe that the characterization according to the application domain, i.e. design-step, is closer to the model users. The user is primarily interested to know whether the proposed model is suited for the application at hand, i.e. the specific design-step which will be taken.

Figure 1 shows the various processor models from the perspective of different design steps.

- Algorithm Development (Computation Model)

  During algorithm development the designer starts mostly with a statement-level spatial accuracy and without detailed timing information. Although the influence of the underlying processor is mostly small, already at this level the designer can make some coarse predictions about the size and execution performance of the program.

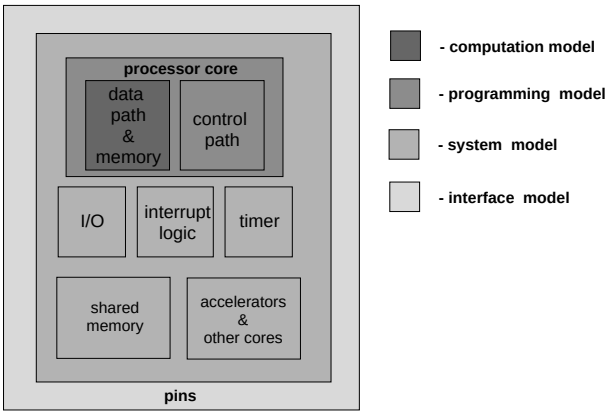

Figure 1. Processor Models According to Application Domains.

The next step is to explore the effects of the processor arithmetic used for computation, like finite word-length effects, and tune the embedded program appropriately. The designer has to switch to the next lower level of spatial accuracy in order to see details of the data registers and memory at the bit level. Of interest are the saturation effects, overflows and effects of register spill-outs.

- Code Optimization and Compilation (Programming Model)

  The process of code optimization has three phases: instruction selection, register allocation, and instruction scheduling. This design step can be done manually by the programmer or automated by the compiler. In both cases successful code optimization requires a detailed timing model, and an pure instruction-accurate model has to be augmented by the cycle counting logic. For pipelined processors even the clock-accurate model has to be used. Also, the information about some specific architectural details is necessary.

- System Design (System Model)

  The system designer needs to extend the model by including the peripherals (I/O, timer, etc.) and the interrupt logic. Additionally, the processing accelerator, other cores on the chip and their interfacing have to be modeled. The spatial and temporal accuracy have to be refined in order to explore the interfacing between the components of the system-on-chip and the asynchronous events they cause. For that task the designer needs cycle-accurate models of the processor core and of all attached units.

- Processor Interfacing (Interface Model)

  Using a complete RTL model of the processor for the design and verification of the interfacing hardware and software is an obvious overkill. It is definitely advantageous to extract and model only those parts of the processor which are relevant for the interface design. Two models serving this purpose can be distinguished. One is the standard bus-functional model. It is characterized by the complete modeling of all pins for a standard access sequence, like a write cycle to an external memory, and a full-timing temporal accuracy. The alternative model provides additionally programming-related details, like registers and memory, and is based on a phase-accurate timing.

4 Compiled Processor Models

We believe that processor models written in C/C++, especially compiled processor models, are highly beneficial for HW/SW co-verification. They provide the flexibility needed in HW and SW environments and also the performance necessary for today's embedded systems.

Our compiled processor models are based on the SuperSim [15] compiled simulation technology. A compiled simulation is generated by a Simulation Compiler (SimCom). The SimCom reads in the object of the program, extracts the static behavior and generates a C program that is functionally equivalent to the read–in program. The C file is compiled and can be run as a batch job or interactively controlled by a standard C Debugger, like the Sun SparcWorks Debugger or the public domain DDD [16]. With a few user–defined commands these debuggers provide the same debugging comfort as traditional simulators. This debugging method is non–intrusive, that is, the state of the processor is not changed during the debug process. Very often simulators use techniques derived from emulation devices to facilitate single–stepping, data inspection and breakpoints. This is usually done by injecting instructions into the pipeline. For a stand–alone instruction set simulator this can be tolerated but with the simulated processor a part of a system, the side effects can corrupt the simulation results.

4.1 Integration with HDL Simulators

An HDL block that is activated on the clock signal needs to be created to integrate a cycle based C simulator into an event–driven HDL simulation . Every time the block is activated the C simulator simulates the processors operation during that clock cycle. The integration of the C model into the HDL simulation can be done using the support for non-HDL code, like C, using the standard FLI and PLI interfaces. The processor simulator can be either a separate process that communicates over IPC with the FLI/PLI, or it

can be linked directly into the HDL simulator. In the latter case the processor model needs to be a dynamically–linked library. This approach removes the IPC overhead but is only beneficial if the HDL simulator can simulate faster than 3000 clock cycles per second. Otherwise, the HDL simulator and not the IPC connection will be the bottleneck.

When signals are passed from HDL signals to C variables they are converted into two–state signals. The processor model can use more than 2 states but this makes the simulator more complex and hence slower. To use C++ classes and overloaded operators is the best way to implement this feature. Beside the reduced number of states also the timing information is lost. Signal changes are forced to the clock edge that activates the C simulator. Signals that are passed from the C part to the HDL simulator need to have this timing information attached to it again. This is fairly simple for signals that come from registers or latches but it can be difficult to regenerate the timing for signals that travel through complex logic before leaving the processor. In such cases a worst–case value can be used or a value can be selected from a set of values by another signal that is provided by the C simulator. In any case these values need to be determined using a HDL simulator. In typical situations the interface between the HDL and the C model will be the bus interface of the processor.

When the user debugs such a combined HDL/C system he or she will have two graphical user interfaces on the screen, one for the HDL simulator and one for the instruction set simulator. Both provide functions to advance the simulation. The optimal solution would be that both can control the simulation. When the user is stepping through the code the HDL simulator advances the appropriate amount of time. When the user is advancing the HDL simulator the ISS is stepping forward appropriately. This requires that both simulators can run in master and slave mode. Another alternative would be to have a third instance, a global simulation controller, that keeps the simulators in sync. With this global simulation controller it would be easier to have more than two simulators in the system. So, it would be possible to simulate a system that consists e.g. of a micro controller, a DSP, several ASICs and some glue logic. The most flexible approach would be to use IPC based interfaces between the components. This is much easier to realize than linking all together into one process. The drawback would be again the IPC bottleneck. We have found that such systems cannot exceed simulation speeds of 3000 clock cycles per second. The alternative is to reduce the amount of IPC by loosening the coupling. Instead of running the simulators completely synchronously we can run them in free–run mode and sync them whenever necessary. This approach cannot simulate the system in a clock–cycle accurate fashion and is therefore only suitable for the functional design phase but not for the verification phase. It also requires that a handshake protocol is used in the system for the interfaces between the simulators. This limitation of the design space might restrict the system designer more than necessary.

4.2 Integration with SW Design Tools

For the software designer the SuperSim compiled models provide several new and useful features. The SimCom generates an functionally equivalent C version of the program. It provides several hooks inside the simulation where pieces of C/C++ code to extend the capabilities of the simulator can be attached. Hooks are provided at the beginning and the end of the simulation, the beginning and end of every instruction or clock cycle and at every label. Whenever the simulation reaches such a hook it will execute the attached C code. The user has full access to the state of the processor as well as the symbol information. E.g. reading the data from a file and writing it to certain registers or variables in the program can be easily implemented. The designer can use hooks to profile the code or determine dynamic instruction distributions, a feature that is very useful for compiler designers. A predefined set of functions makes the hooks even for inexperienced C programmers as comfortable to use as the rest of the simulator functionality.

As SuperSim simulators can be run in the slave mode it is also possible to call parts of an embedded program from within a C/C++ program that runs on the host. This is very useful when migrating a C/C++ prototype to a program that runs on the embedded system. The C code can call subroutines on the processor handing over the execution control to the simulator. It returns the control back to the C/C++ program when the subroutine is finished. A set of C functions is provided to transfer data between the C/C++ code and the simulated processor. Full symbol debugging support is still available.

High volume DSP systems like modems or cell phones are very often designed around a fixed-point signal processor. Due to the poor quality of high–level language compilers these processors have to be programmed in assembly language. The DSP code development takes typically one third of the overall design time. Comparing this to the time one would need to type the code into a file it becomes clear that roughly 90% of the time is used for debugging.

For most systems two prototypes, a floating-point and a fixed-point, are developed in C or C++ prior to starting the work on the actual assembly code. We have developed the SuperSim Mixed–Mode methodology that enables the designer to make a seamless transition from the fixed-point C prototype to the final assembly code. The SuperSim Mixed-Mode, turns the C++ prototype into a test bed for any piece of assembly code. The assembly code is executed as part of the embedding C++ program. Now we can check whether

the DSP code computed the right results by comparing the results of the equivalent C operations. In case of a mismatch an error is flagged. As the SuperSim C–Assembly–Mixed—Mode imposes no limitation on the size of the DSP code, the programmer can use any granularity that is helpful. E.g. in case of a tight coupling errors in the DSP code can be found automatically within 10 to 15 instructions.

The SuperSim Mixed-Mode is highly beneficial for assembly code implementation and optimization. The high speed of the SuperSim simulator enables the designer to frequently run huge sets of tests to validate the correctness of the code during the design process. This technique, called *Comparative Debugging Technique* [17] can be automated. Users have reported a 5- to 10-fold speedup in debugging their assembly code by applying the proposed methodology.

As we use standard C/C++ debuggers like SparcWorks or DDD to control the simulation we have full debug support for the C/C++ code available. The user can step through the C code that runs natively on the host and when the processor simulation is called the display is switched to the processor code. Now the user can step through the processor simulation until it hands back the control to the calling C code. Then the display switches back automatically again. With full access to the C data and processor state this is a very powerful debug tool.

The SuperSim Mixed–Mode can also be used in conjuction with tools like COSSAP [13] or SPW [18]. These tools generate their simulation out of C code that was provided for each block in the system. With the SuperSim Simulation Compiler it is possible to convert a program for the embedded processor into a C representation. This can be inserted as one or multiple blocks into these tools. Contrary to using the IPC based ISS interface that these tools provide, this is an extremely efficient way of co–simulating a processor with COSSAP or SPW. Again it is very helpful that the SuperSim simulator can use a standard C debugger as debugging interface, so the user can use a single front–end to debug the C code and the program running on the simulated processor.

5 Conclusion

We have outlined the challenges designers are facing while designing and verifying todays systems–on–a–chip and presented a technique that is a reasonable compromise between simulation accuracy and simulation speed. C/C++ models seem to be the natural bridge between the processor models used by hardware and software designers. The introduction of the compiled simulation of processor models is the key element to provide the flexibility and simulation speed necessary to create this compromise.

References

[1] J. Rowson, "Hardware/Software co-simulation," in *Proc. 31st Design Automation Conference*, 1994.

[2] A. Kalavade and E. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design & Test of Computers*, pp. 16–28, Sept. 1993.

[3] VLSI/CAD Group, Stanford University, Stanford, CA, *Thor Tutorial*, 1986.

[4] A. Ghosh, *et al.*, "A hardware-software co-simulator for embedded system design and debugging," in *Proc. of Asia and South Pacific DAC, Chiba, Japan*, pp. 155–164, Aug. 1995.

[5] P. Kra, "A cross-debugging method for hardware/software co-design environments," in *Proc. 30th Design Automation Conference*, pp. 673–677, 1993.

[6] D. Becker, R. Singh, and S. Tell, "An engineering environment for hardware/software cosimulation," in *Proc. of DAC'92*, 1992.

[7] S. Coumeri and D. Thomas, "A Simulation environment for hardware-software codesign," in *Proc. of ICCD'95*, 1995.

[8] B. Schnaider and E. Yogev, "Software development in a hardware simulation environment," in *Proc. of the DAC – 1996, Las Vegas, USA*, pp. 684–689, June 1996.

[9] Lin, B. et al., "Designing Single Chip Systems," in *Proc. of ASIC'96*, 1996.

[10] R. Earnshaw, L. Smith, and K. Welton, "Challenges in cross-development," *IEEE Micro*, no. July/Aug., pp. 28–36, 1997.

[11] "Mentor Graphics Seamless CVE Tool." Available on http://www.mentorg.com/codesign/main-f/collateral/Seamless-CVE/datasheet.htm.

[12] "Synopsys' Eagle Tools." Available on http://www.synopsys.com/products/hwsw/eagle_ds.html.

[13] "Synopsys' COSSAP Tools." Available on http://www.synopsys.com/products/dsp/dsp.html.

[14] "RASSP VHDL Modeling Terminology and Taxonomy ." Available on http://rassp.scra.org.

[15] V. Živojnović and H. Meyr, "Compiled HW/SW co-simulation," in *Proc. of the DAC 1996 – Las Vegas*, pp. 690–695, June 1996.

[16] "The Data Display Debugger (DDD) ." Available on http://www.cs.tu-bs.de/softech/ddd.

[17] C. Schläger and V. Živojnović, "C/C++ - based techniques for production-quality DSP code development," in *Proc. of ICSPAT'95 - Boston*, Oct. 1995.

[18] "Cadence Signal Processing Workstation." Available on http://www.cadence.com/alta/products/spwhds_dat.html.