The A to Z of SoCs

Reinaldo A. Bergamaschi
IBM T. J. Watson Research Center, Yorktown Heights, NY, USA
John Cohn
IBM Microelectronics, Burlington, VT, USA

Abstract— **The exploding complexity of new chips and the ever decreasing time-to-market window are forcing fundamental changes in the way systems are designed. The advent of Systems-on-Chip (SoC) based on pre-designed intellectual-property (IP) cores has become an absolute necessity for embedded systems companies to remain competitive. Designing an SoC, however, is extremely complex, as it encompasses a range of difficult problems in hardware and software design. This paper explains a wide range of SoC issues including market drivers and trends, technology and integration aspects, early architecture definition, methodology, hardware and software design and verification techniques.**

I. Introduction

In the last 20 years embedded systems have become the most widespread carriers of advanced hardware and software technologies. In this period the technologies implementing embedded systems evolved from microcontrollers and discrete components to fully integrated systems-on-chip (SoC). SoCs and related technologies are the engines of embedded systems today and in the foreseeable future.

New and very demanding applications (in terms of processing power) have appeared in the last 20 years, fueled by the boom of the PC, the Internet, and the home, office and wireless environments. These applications demanded processing power (for ever expanding software) and levels of hardware integration much greater than could be offered by integrating discrete components on a printed circuit board. Product cycles continue to shrink as evidenced by the adoption rate of consumer electronics items. While color TV took over 10 years to sell 1 million units, DVD players took just over 1 year [1]. This puts strong requirements on productivity increases and cost reduction.

CMOS technologies have evolved significantly in the last several years, allowing chip capacity to follow Moore's law and produce chips with over 70 million gates and gate delays of around 21ps [2]. Large gate counts and high operating frequencies allied with new chip architectures led to considerable increases in processing power. At the same time, new design tools and methodologies were developed, which were able to take advantage of complex process technologies and deliver highly complex chips.

The concept of systems-on-chip is unique in leveraging the new technologies and architectures in order to provide the processing power and productivity increases needed by the new applications.

Systems-on-chip started with the idea of integrating all components in a board into a single chip. To increase the productivity for future designs, the approach of reusable components was adopted. However, in the early days of SoCs, components were not really designed for reuse. The lack of standard deliverables as well as multiple interface protocols made it difficult for components to be taken from one design and reused in the next without modifications. Gradually component design evolved to include more parameterization, deliverables such as synthesis scripts and test vectors, and standard interfaces. This evolution led to a new industry devoted solely to the development of intellectual property cores (IPs), as the reusable building blocks of systems-on-chip [3].

IP components or cores are usually available in three forms: hard, firm and soft cores. Hard cores are provided as black boxes, usually in layout form and with an encrypted simulation model. Due to their high performance and/or design complexity, these cores need to be provided as an optimized layout in a given technology, with known performance. Examples of hard cores are microprocessors, phase-locked loops, and mixed signal blocks. Firm cores are provided as a synthesized netlist, that is, after logic synthesis and technology mapping, but without layout information. These are cores which users do not have to re-synthesize, but are given the netlist in a hardware-description language (HDL), which can be simulated and changed it if necessary. Soft cores are given as register-transfer level HDLs, and the user is responsible for its synthesis and layout. Usually along with the soft cores, the IP providers also supply synthesis and layout scripts and timing assertions.

The majority of embedded applications can be implemented on architectures built using common architectural

0-7803-7607-2/02/$17.00

blocks and customized with application specific components. Hence it was logical for SoC providers to develop these common architectural blocks formed by a CPU communicating with peripherals and memory over one or more shared busses. The common architectures typically include a CPU, memory and peripherals communicating over a fast bus (for the CPU and memory) and a slow bus (for the peripherals). This initial configuration can be extended with an MPEG decoder for video applications, or with an Ethernet controller for communications applications. These common architectures and the supporting technologies (IP libraries and tools) are called platforms and platform-based designs [4], [5].

In practice, a platform consists of the following: list of cores and their descriptions (in HDL for soft and firm cores, and GDSII for hard cores), the top-level RTL HDL specifying the interconnections amongst the cores based on the chosen reference architecture, synthesis scripts for running logic synthesis on the cores, timing assertions, device drivers for peripheral components and software tests for testing the complete chip.

The factors above led to the current state-of-the-art in SoCs as *IP-based, Platform-based systems-on-chip*. These SoCs are systems that contain IP blocks such as embedded CPUs, embedded memory, real world interfaces (e.g., PCI, USB, Ethernet), mixed-signal blocks, and software components, such as device drivers, real-time operating systems and application code.

This paper presents an overview of the main issues and steps in designing an SoC, from the technology challenges, to initial concept and feasibility phases to physical design and software issues.

II. Technology Aspects

IP-based, Platform-based SoCs have firmly established themselves as the engines of high-performance embedded systems. Although the SoC concept is about rising the level of design abstraction, it is getting increasingly difficult to mask the complexities of deep sub-micron technology scaling. This section outlines the principle challenges of performance scaling, power, signal integrity, technology integration and cost associated with SoC design.

Performance scaling

One of the largest obstacles to increasing SoC performance is the divergence of device and interconnect performance scaling. Global interconnect delays are increasing relative to device performance due to increases in wire resistance and lateral capacitance. Where once it was possible to assume that all IP on a chip was within a single clock cycle, it may now be necessary to accommodate interconnect delays at an architectural level using asynchronous signaling or pipelining. In a bus-based SoC this may require the splitting of a single logical bus into multiple local regions or the addition of pipelining into emerging bus specifications.

Power

Power dissipation has emerged as one of the fundamental limits to SoC complexity. Despite gradual lowering of supply voltages, total chip active power is increasing due to increased transistor density coupled with increased switching speed. Smaller device geometries, thinner gate oxides and lower device thresholds also contribute to increases in leakage or stand-by power. Moreover, these power increases drive up system cost significantly by complicating chip packaging, system cooling and power supply or battery life.

Modern SoCs use a number of techniques to manage both active and leakage power. Significant power reductions can be attained through *clock gating* and *voltage islands*. Parts of the chip (islands) which must run fastest are powered by higher voltages, while those that can run slower are powered by lower voltages. In addition, voltage and frequency scaling can be used to save power while adjusting for changing performance requirements [6]. New methods and tools must be developed to take advantage of these new power reduction techniques.

Signal integrity

In addition to performance and power, SoC designers are faced with a growing list of signal integrity or noise concerns. The dominant noise concerns in a modern SoC are coupling noise, power supply noise and substrate noise. Coupled noise is signal injected through the mutual capacitance of adjacent wires. Power noise is supply voltage variation caused by large current transients passing through supply resistance and inductance, and substrate noise is signal injected into the substrate by switching transistors. Each of these noise sources can cause unintended variation in system timing and/or function. The magnitude of each of these effects is increasing with process scaling [7].

Signal integrity concerns affect both IP developers and SoC chip integrators. IP developers must analyze and characterize their IP to describe its noise tolerance. Suppliers of mixed signal and DRAM IP must take particular care to manage noise.

Often mixed signal IP is designed with separate power bussing and decoupling to minimize power noise and substrate diffusion guard rings to minimize injection of substrate noise [8].

SoC integrators are responsible for analyzing the signal integrity of the global routing and mitigating problems by separating or shielding noise sensitive signals. Global analog signals may require extra care in the form of differential routing and/or custom wire width tuning. In some cases it may become necessary to protect against noise at an architectural level by adding parity or other error mitigation techniques to SoC bus protocols. Similar care must be taken to ensure sufficient power bus width and decoupling capacitance to provide adequate power noise immunity at the chip and package level.

Technology integration

One of the main drivers of SoC has been the integration of several chips onto a single silicon die. Integration can reduce design cost by lowering packaging cost, improving performance and reducing physical size. However, technology integration has its own set of great challenges.

Integrating process features from two or more separate technologies often requires significant trade-offs in technology capabilities. For example the substrate temperatures found on fast logic chips is not optimum for DRAM process. As a result, embedded DRAM generally needs more frequent refresh than stand-alone DRAM [9]. Similarly, process optimizations and supply voltage limits for high performance logic are often at odds with those which would optimally be used for high precision analog design [10]. Mixing IP can also raise issues of electrical compatibility. Signal integrity becomes a significant concern when placing sensitive analog or DRAM elements onto the same chip as fast logic.

Chip cost is also an issue when integrating mixed technologies. Because all elements share a common silicon substrate, certain processing steps required for one piece of IP (e.g., silicon-germanium RF transistors or trench isolation steps for embedded DRAM) must be applied to the entire chip. These extra steps can dramatically increase the cost per areas of logic on the chip making the SoC less cost competitive [11].

System-on-Package

One increasingly attractive alternative to system-on-chip is system-on-package (SoP). Through advances in packaging technology, the traditional electrical overhead of chip to chip interconnection has been reduced. It is now possible to integrate multiple dies together using interconnections that approach the density and delay characteristics of on chip interconnections. Advances in silicon package [12] and chip stack [13] technology have made it possible to vertically stack multiple dies using specially modified flip chip or wire-bond techniques.

These SoP techniques allow individual system components to be in the best available technology. The cost savings associated with being able to separately yield theses smaller *chiplets* can in some cases offset the increase in complexity of multi-chip packages [14]. This is particularly true if one of more IP blocks may require multiple design passes to ensure functionality, as is often the case in analog IP design.

Design cost

While cost has always been an issue in SoC design, process complexity has reached a point where design cost may be a bigger limiter than physics in SoC performance and function. With mask costs increasing to more than a million dollars per design pass [15], designers may soon find it difficult to justify custom SoCs for all but the highest volume applications.

For this reason there is a growing need for the development of reusable platform-based designs based on embedded FPGAs [16], gate arrays or programmable fabrics[17] which will allow for low cost customization.

III. How to design an SoC

This section describes the various steps in digital SoC design as currently practiced in industry. The IP-based, platform-based SoC design business environment is typically divided into companies who own the applications and the specifications (such as telecom companies) and silicon providers who own the IP libraries, the platforms, and the design services. In addition, more vertically organized companies also exist which develop both the applications and the silicon implementations.

Figure 1 illustrates the methodology flow and major steps in SOC design. It normally starts with an initial contact between the party controlling the application and the SoC/Silicon provider. Together they decide on the major functions, constraints and requirements for the SoC. This initial step is followed by early design definitions and estimations, then detailed architecture design, mapping to platform, chip-level (hardware) design and release to manufacturing.

The embedded software development starts even before a simulatable model of the hardware is available and proceeds

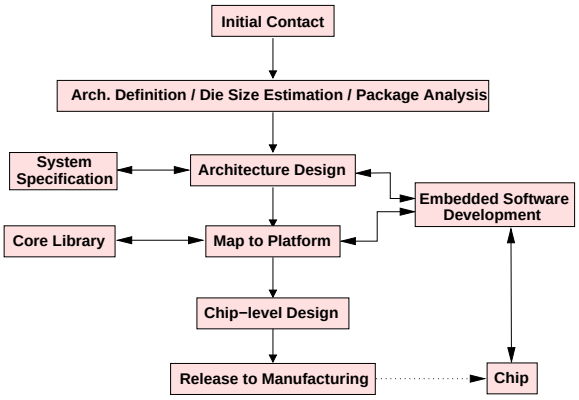

Fig. 1. SoC Design Methodology Flow

in parallel with the hardware development. Details of each step are given in the following sections.

A. Early decisions: architecture definition, die size estimation, packaging selection

In this stage the early decisions on the architecture are made. This includes creating the block diagram of the chip and making sure that the major hardware functions needed in the application are supported in the platform. Die size, package selection and power analysis are required in order to estimate the final manufacturing cost.

Die size estimation is based on the overall cell count and I/O requirements. Overall cell count is based on the estimated total number of gates, corrected for wireability and an empirically-derived correction factor [18]. A given package type can comport a range die sizes and can support a given maximum number of IOs.

Early Power Analysis is also considered when choosing packaging and estimating costs. At this early stage, power is analyzed in the context of package type, substrate, number of power planes, switching frequency of off chip drivers, available chip power and ground pins [19].

These estimations use approximate algorithms, designer experience and empirical data based on previous designs. They are needed in order to reach a final agreement between the client and the silicon provider on the schedule and cost (and for the contract to be signed).

B. Architecture Design

Once the early decisions are made, the designer can then design the architecture of the SoC in much greater detail. This step involves the following actions.

The functionality and requirements may come in different forms: (1) a high-level description of the system, (2) A legacy SoC with new requirements, or (3) a textual specification. In the case where the client and the silicon provider are different parties, it is unlikely that a full high-level description of the system is provided. This happens because either such complete description (including software and hardware functions) does not exist, or because the software (application) part is proprietary and the client may not let the silicon provider have access to it. In this case, approximations of the work load and instruction traces may be used.

One of the early decisions at this stage is the partitioning between hardware and software. In practice this is done manually and to a large extent dictated *a priory* by the specification. For example, if there are software applications which were written for a previous system that need to run in the new system as well, or if there are legacy requirements on the hardware, that effectively dictates the hardware/software partitioning.

Once a platform is chosen, the designer has to verify that functionality, performance and customer requirements satisfied by the platform. If there are required functions which are not part of the platform, they need to be added either by inserting new IP cores from the library or by creating user-defined logic blocks. Alternatively, if the required IP core is not available in the silicon provider's library, the client has the option of licensing it from a third party IP vendor.

The designer starts with a platform specification and a target architecture and explores the space of architecture modifications that are possible using the platform and the available cores. For example, in the case of video applications, should the design use a single high-speed bus for both the CPU and the MPEG subsystem to communicate to memory, or should it use two busses for higher throughput by allowing the CPU and the MPEG decoder to access memories simultaneously. This design space exploration involves detailed performance analysis and simulation in order to be validated [20]. Different modeling approaches above the RT level can be used here, namely, bus-functional models, instruction-set simulators, memory models, co-simulation tools.

C. Mapping Architecture to Platform

Once the architecture is defined, it can then be mapped onto the chosen platform. This will result in the top level RTL HDL description for the SoC, plus descriptions for all user-defined logic blocks. The IP descriptions are taken from the library.

Although the mapping uses an existing platform as a ba-

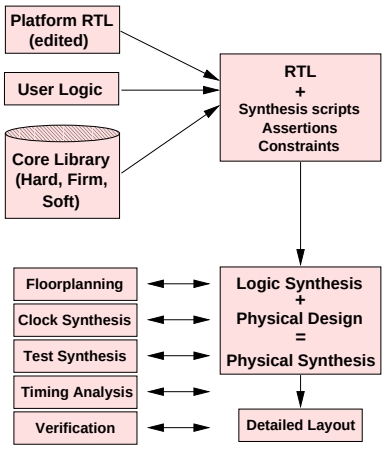

Fig. 2. Chip design methodology flow

sis, it is very common that the RTL for this platform needs to be edited to conform with design specifications from the architecture definition phase. This involves editing the RTL description manually (text or schematic diagram) which can be quite complex since it requires designers to understand the functionality of tens to hundreds of pins in several cores [21].

In this mapping process, the designer has to enforce all architectural decisions in the netlist, including:

- Interrupt subsystem: define priorities of all interrupt requesters and interconnect them correctly.
- DMA subsystem: choose DMA channel assignments and channel sharing (multiple devices connected to one channel) and interconnect them accordingly.
- Address maps: pass the peripheral devices address map values as parameters to each core.
- Clock domains: define all valid domains and clock generation scheme.
- Inputs and Outputs: define all chip I/Os, I/O logic, test control logic.
- Document the system: pass address maps, interrupt priorities, DMA channels, I/Os, to software developers.

D. Chip-Level Design

The chip design methodology overview flow is illustrated in Figure 2. It starts with the complete RTL description, formed by three parts, namely: the platform RTL properly edited and parameterized, the user-defined logic HDL and the HDL for the cores. Firm cores are read in but usually not touched by logic synthesis unless there is a timing problem. Soft-cores are fully synthesized. Hard cores are treated as black boxes by all synthesis and verification stages, being finally integrated during the detailed layout phase.

Soft and firm cores come with specific synthesis scripts and assertions. Synthesis is first run on each core separately and then run on the whole design (this time not going inside the cores), but just handling the glue logic in between cores.

Floorplanning is performed throughout the design process, but most specifically in the beginning, and updated as necessary, as the synthesis progresses. Timing Analysis is also performed at various stages. Clock trees and scan chains also need to be taken into account in the initial floorplan.

Verification includes both functional and equivalence checking. Formal methods are used in both cases [22], [23].

Floorplanning

Floorplanning deserves special attention in SoC design because of the many placement requirements imposed by diverse IP blocks such as hard cores and tight timing constraints. At an early stage, floorplanning derives relative/fixed positions for major design blocks and pass information forward to the design team on major design constraints and critical wires, to help synthesis and physical design achieve wiring and timing closure.

It is an iterative process involving the following steps: get physical data for all design cores (for soft cores, this has to be an estimation), understand the system architecture with respect to busses and clock domains, understand the chip image and package limitations, plan and assign I/Os, place large cores manually (complete automatically if supported), account for clock trees, test logic, power, analog blocks, switching noise and blockage constraints.

Some manual placement is needed to satisfy known constraints. For example, when cores (hard or soft) connect directly to the I/Os, then it is often beneficial to place them right at the edge of the die, close to the corresponding I/O pins. Similarly, when cores with tight timing constraints connect directly to each other, they should also be placed together. The designer will manually enforce these requirements, and then let an automated floorplanning tool complete the task [24].

Clock Distribution

Clock distribution in SoCs is particularly difficult because of the large die size, high frequency and blockages due to hard cores. For high frequency designs (>200MHz) it is important to minimize clock skew and clock distribution wire length. In IBM's SoC methodology, it was found that the use

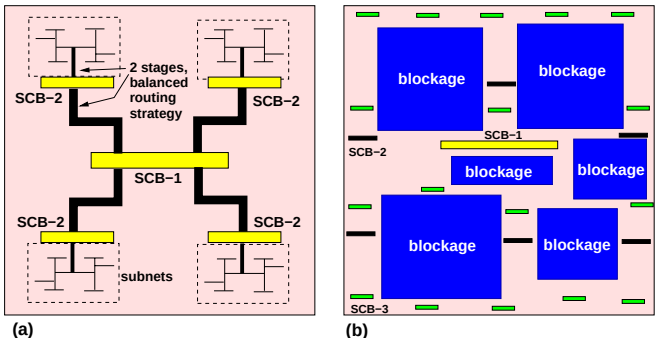

Fig. 3. (a) Structured Clock Buffer Tree, (b) SCB distribution in the presence of blockages

of *Structured Clock Buffer* (SCB) trees resulted in lower skew and wire lengths for a variety of chip designs and styles, when compared to traditional balanced clock distribution networks [25].

The SCB methodology uses a clock tree from the center of the chip to its quadrants using large buffers and few levels of buffering, as shown in Figure 3a. The routing is balanced to produce the same delay down to the end buffers. From the end buffers (reaching quadrants or regions on the chip), *subnet* trees are created to take the clock signal to the individual registers. These trees are created using balance clock routing which produce approximately equal capacitance. The automated clock tool [25] can layout the SCB trees automatically given user input on the number of levels, maximum latency and skew.

In case of SoCs with large hard cores creating blockages, the SCB approach has to work around the blockages by changing the number of levels and position of the large buffers. Figure 3b shows an example, where due to blockages in the center and quadrants of the chip, the simple 1-to-2-to-4 SCB tree was not possible, and the solution was to build a tree with more stages and smaller buffers, such as a 1-to-6-to-17 SCB tree.

Physical Synthesis

Once an initial floorplan and clock trees are defined, one can proceed with detailed logic synthesis and physical design. Typically, the steps of synthesis and layout were performed in sequence, followed by capacitance extraction from the layout, back-annotation and re-synthesis. This loop can take a long run time and several iterations to achieve timing closure. The main problem is that traditional logic synthesis uses average wire-load estimations based on the fanout of each gate, and is not aware of placement locations, resulting in poor quality electrical optimizations. Conversely, physical design algorithms usually optimize the placement for minimum total wire length, and is not aware of local timing and wiring constraints.

Physical synthesis is the name given to the relatively new area of combined logic synthesis and physical design [26]. The approach consists of interleaving logic synthesis and placement transformations, such that when synthesis considers a part of a network, it has more precise knowledge about the placement and individual wire lengths (estimated using Steiner trees) and can thus optimize it more efficiently.

In an SoC context the problem of synthesis and physical design is akin that of a hierarchical design. Hard cores are treated as black boxes with fixed pin capacitances and assertions. Firm cores are pre-synthesized but may be subject to partial changes during physical synthesis if placement decisions require partial re-synthesis. Soft cores are fully synthesized using physical synthesis plus pin capacitance limits and assertions derived from the environment (during the early floorplanning phase).

Once the individual cores are done, the top-level design is then run through physical synthesis, using information spanning the hierarchy. For example, for computing the Steiner tree of a net originating inside a core, it needs to be able to look inside the hierarchy, obtain the placement of the source gate and compute the Steiner tree based on the placement of the sink gates of the net (which may lie at the top-level or inside other components), and use that to compute the net capacitance and timing.

E. Embedded Software Design

With the growing complexity of embedded applications, software has become at least as complex to design and as costly as the hardware. The domain of software in embedded systems is illustrated in Figure 4. The user interfaces with an embedded system through the applications software, which calls functions available in the real-time operating system (RTOS). The RTOS contains both high-level functions to be used by the application code as well as low-level functions, called device drivers, to communicate with the hardware.

Figure 4 also shows the path of a simple example of a *printf* function when called from the application code. This *printf* is decomposed by the compiler using functions provided by the RTOS, such as *printchar* to print one character at a time. This *printchar* calls a function *putchar* which is part of the device driver for the serial interface peripheral core (e.g.,

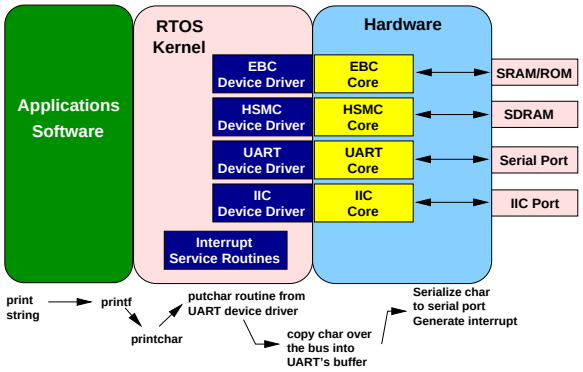

Fig. 4. Embedded software domain

Universal Asynchronous Receiver Transmitter - UART). The *putchar* routine writes the character into a specific register inside the UART core. The UART hardware then is responsible for serializing the character bits into a serial line.

The hardware parameters are passed to software via header files. The real hardware/software interface is a combination of setting hardware parameters, communicating them to the software developers and using low level software to control the hardware (change parameters, get status, set registers). In embedded system design this hardware/software interface assumes the faces of: memory maps, initialization/reset/boot code, device drivers, interrupt service routines, IO pin sharing.

Software development starts as early as possible. Before a hardware description is available, programmers can use a C/C++ model of the hardware in order to start the application development and test interactions with the RTOS. Later on, as the hardware description becomes available, a co-simulation environment can be used, where the hardware is simulated in a hardware simulation environment, and the software in the host system. This allows for testing the hardware/software interfaces and device drivers.

Device Drivers

Device drivers represent the interface between the operating system (OS) and the I/O devices in an embedded system. The purpose is to hide the hardware from the upper software layers, by providing a set of functions to control the operation of the peripheral devices and their interactions with the OS. Such functions include: initialization services (e.g., setup baud rates, timer periods); configuration services (e.g., set values into registers); I/O interface services (e.g., send/receive characters, start/stop timers, control DMA transfers); interrupt service routines (to respond to hardware initiated interrupts). The device driver code is part of the OS and linked with the OS kernel [27], [28].

F. Co-Verification Approaches

A differentiating aspect between SoCs and application-specific integrated circuits (ASICs) is the need in the former for co-verification of hardware and software. Hardware and software may start development approximately at the same time, and may proceed reasonably independent for some time until the point where hardware and software interactions need to be tested.

On the hardware side, testing is done mostly at the RT level. RTL simulation is cycle accurate and can test actual device operation. In order to use the actual application software to drive the RTL simulation, one needs to cross compile it for the processor modeled as RTL, and load the instructions in the HDL memory model (which is part of the RTL simulation). This allows some software testing but it is extremely slow. RTL simulation is appropriate for hardware testing but far too slow for any serious software testing.

The other extreme is to test the system completely in software. In this case, the interfaces to the hardware are modeled as C functions, which are called by the application software, compiled and run on the host system. This scheme is appropriate for testing the application software independent of the hardware. Only limited testing of the interactions with hardware is possible.

In between these extremes lie the co-verification approaches. They are simulation environments which allow the designer to split the system into parts which should be simulated as hardware in HDLs, and parts simulated as software as code running in an instruction set simulator (ISS) or host system. In the case of an ISS modeling the CPU, full details on the processor internals are not modeled, but it may be possible to account for pipelining stages, superscalar architectures and cache misses. In the case of software running on a host system, no modeling of the real CPU details is possible.

The co-verification tools are responsible for the communication between the diverse simulation environments. The co-simulation kernel handles the synchronization and data transfers between the HW and SW environments, that is, between HDL simulation and the CPU execution (ISS or host system). The CPU execution environment provides a trapping mechanism that allows C function calls to be made when certain addresses are accessed by the executing soft-

ware. These trapping mechanisms direct memory-mapped accesses from the HDL or the CPU to the program memory or the HDL memory (or vice versa) depending on the address maps. Both CPU and HDL issue memory accesses. In advanced co-simulation environments, the user can control which simulation gets access to which memory addresses.

Bus activity related to load/store is usually handled between the CPU and the program memory, which is very fast. Bus activity related to non-CPU memory-mapped devices and ports is passed to the HDL simulator. Total system simulation throughput is gated by the frequency of call outs to the HDL simulation [29].

IV. Conclusions

This paper presented an overview of the most important issues involved with SoC design, including challenges associated with deep sub-micron technology scaling, and details on the SoC design methodology and tools. One important aspect of an SoC is that it is significantly more complex to design than a traditional ASIC. Although both SoCs and ASICs can be very large, ASICs are simpler to design because its optimization metrics are usually area and delay, whereas SoC design involves a multi-domain optimization problem. For example, even if the clock period target of the SoC is met, it may not work properly because of wrong architectural decisions, inefficient software or signal integrity problems due to mixed signal IP. In SoC design the high-level design decisions (e.g., architecture, software) are much more interdependent on the low level design decisions (logic synthesis, layout, technology), than in ASIC design. This cross-domain optimization problem makes the SoC design much more complex which requires specialized tools and methodologies.

References

[1] R. Wawrzyniak, "Systems-on-a-chip: A brave new world." Semico Research Corporation Report SC101-1-99, September 1999.

[2] "IBM ASIC CU-08 technology notes," 2002. Available for downlod from http://www-3.ibm.com/chips/techlib/techlib.nsf/ /products/ASIC_Cu-08.

[3] A. Rincon, W. Lee, and M. Slatery, "The changing landscape of system-on-a-chip design," *IBM MicroNews*, vol. 5, 3rd Quarter 1999.

[4] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design & Test of Computers*, vol. 18, pp. 23–33, November/December 2001.

[5] D. Senzig, "The IBM PowerPC 44GP system-on-chip," *IBM MicroNews*, vol. 6, 4th Quarter 2000.

[6] K. Nowka, G. Carpenter, E. MacDonald, H. Ngo, B. Brock, K. Ishii, T. Nguyen, and J. Burns, "A 0.9V to 1.95V dynamic voltage-scalable and frequency-scalable 32b PowerPC processor," in *Proceedings of the International Solid State Circuits Conference*, IEEE, February 2002.

[7] K. L. Shepard and V. Narayanan, "Conquering noise in deep sub-micron digital ICs," *IEEE Design & Test of Computers*, vol. 15, pp. 51–62, January/March 1998.

[8] B. R. Stanisic, N. K. Verghese, R. A. Rutenbar, L. R. Carley, and D. J. Allstot, "Addressing substrate coupling in mixed-mode ICs: Simulation and power distribution synthesis," *IEEE Journal of Solid-State Circuits*, vol. 29, pp. 226–237, March 1994.

[9] S. C. et. al., "Integration of trench DRAM into a high-performance 0.18mm logic technology with copper BEOL," in *Technical Digest of the International Electron Devices Meeting*, 1998.

[10] M. H. et. al., "Design considerations for gigabit ethernet 1000Base-T twisted-pair transceivers," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, IEEE, May 1998.

[11] "Toshiba shows 100-nm SoC that supports eDRAM," 2001. Silicon Strategies, June 13.

[12] H. B. Pogge, "The next chip challenge: Effective methods for viable mixed technology SoCs," in *Proceedings of the 39th ACM/IEEE Design Automation Conference*, pp. 84–87, ACM/IEEE, June 2002.

[13] J. Dufresne, S. Ouimet, and T. R. Homa, "Hybrid assembly technology for flip-chip-on-chip (FCOC) using PBGA laminate assembly," in *Proceedings of Electronic Components and Technology Conference (ECTC'00)*, 2000.

[14] "IBM researcher stumps for system on package designs," 1999. EE Times, http://www.eetimes.com/semi/news/OEG19991208S0022, Dec 8.

[15] "Chip industry tackles escalating mask costs," 2002. EE Times, http://www.eetimes.com/semi/news/OEG20020617S0050, Jun 17.

[16] P. S. Zuchowski, C. Reynolds, R. Grupp, S. Davis, B. Creman, and B. Troxel, "A hybrid asic and fpga architecture," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, IEEE, November 2002.

[17] "Gigascale silicon research center publications." Available for download from http://www.gigascale.org/pubs/.

[18] "Die size estimation," 2000. IBM Microelectronics Application Note. Restricted access through http://www.edge.ibm.com.

[19] "Power estimation in ASICs," 2001. IBM Microelectronics Application Note. Restricted access through http://www.edge.ibm.com.

[20] J. Darringer, R. Bergamaschi, S. Bhattacharya, D. Brand, A. Herkersdorf, J. Morrell, I. Nair, P. Sagmeister, and Y. Shin, "Early analysis tools for systems-on-a-chip designs," *IBM Journal of Research and Development*, vol. 46, November/December 2002. To appear.

[21] R. A. Bergamaschi and W. R. Lee, "Designing systems-on-chip using cores," in *Proceedings of the 37th ACM/IEEE Design Automation Conference*, (Los Angeles), pp. 420–425, ACM/IEEE, June 2000.

[22] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proceedings of the 34th ACM/IEEE Design Automation Conference*, pp. 263–268, ACM/IEEE, June 1997.

[23] I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal, "Rulebase: Model checking at IBM," in *Proceedings of the International Conference on Computer-Aided Verification*, pp. 480–483, 1997.

[24] J. Y. Sayah, R. Gupta, D. D. Sherlekar, P. S. Honsinger, J. M. Apte, S. W. Bollinger, H. H. Chen, S. DasGupta, E. P. Hsieh, A. D. Huber, E. J. Hughes, Z. M. Kurzum, V. B. Rao, T. Tabtieng, V. Valijan, and D. Y. Yang, "Design planning for high-performance asics," *IBM Journal of Research and Development*, vol. 40, July 1996.

[25] K. M. Carrig, N. T. Gargiulo, R. P. Gregor, D. R. Menard, and H. E. Reindel, "A new direction in ASIC high-performance clock methodology," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 593–596, IEEE, May 1998.

[26] S. Hojat and P. Villarrubia, "Integrated placement and synthesis approach for timing closure of PowerPC microprocessors," in *Proceedings of the IEEE International Conference on Computer Design*, IEEE, October 1997.

[27] G. Pajari, *Writing UNIX Device Drivers*. USA: Addison-Wesley, 1991.

[28] S. Gal-Oz and A. Cohen, "The hazards of device driver design," *Embedded Systems Programming*, pp. 34–46, May 1997.

[29] B. Morasse, "Effective use of various levels of system abstractions within a hardware/software co-verification environment," in *Proceedings of the Embedded Systems Conference*, CMP Media Inc., Fall 1999. Class 545, available online at www.esconline.com.