Getting High-Performance Silicon from System-Level Design

W. Rhett Davis
Department of Electrical and Computer Engineering
North Carolina State University
rhett_davis@ncsu.edu

Abstract

System-level design techniques promise a way to lessen the productivity gap between fabrication and design. Unfortunately, these techniques have been slow to catch on, in part because they do little to help designers optimize hardware. This paper presents a brief summary of three system-level design techniques, Platform-based design, SystemC, and Chip-in-a-day, in order to propose that more system-level abstraction of physical performance is needed to make these techniques more useful. An analysis of design-productivity for three chips designed with the Chip-in-a-Day flow is also presented.

1. Introduction

Today's silicon technology offers ASIC designers the potential for great speed, density, and energy efficiency. Unfortunately, issues such as interconnect-delay and cross-talk make it very difficult to prototype systems with the latest technologies, forcing many promising ideas to go untested. To make up for increasing design time, companies have no option but to throw more engineers at every project. This problem is evidenced in the "productivity gap" between the number of transistors-per-chip that we can effectively manufacture and the transistors-per-designer-per-year that we can effectively design. The International Technology Roadmap for Semiconductors (ITRS) [1] claims that cost of design is the greatest threat to the continued growth of the semiconductor industry.

System-level design techniques, as described in the ITRS, promise a means to increase designer productivity. These techniques use specifications at a higher level of abstraction than RTL or C code that are intended to allow software and hardware to be optimized simultaneously. Most system-level methodologies aim to increase productivity by making the behavior of a system independent from its architecture. This independence would simplify the mapping of a system's functionality to blocks of existing intellectual property (IP).

The promise of system-level design allows us to envision integrated systems with the speed, area-efficiency, and energy-efficiency of dedicated hardware but with the flexibility of software. Unfortunately, the existing techniques for system-level design are focused much more on the communication between programmable cores (software-software co-design) and offer very little to accelerate the design of dedicated hardware. As a result, we tend to abandon system-level design and use old, unproductive techniques when we want high-performance silicon.

This paper presents a perspective of how modification of our system-level design approach could lead to the promised improvements in productivity without sacrificing circuit performance. The paper begins by discussing a model for analyzing a designer's productivity in the context of a specific design flow. Next, this model is used to analyze two popular system-level design techniques, *SystemC* and *Platform-based design*, as applied to the development of dedicated hardware, to illustrate where circuit performance is lost. Then this model is applied to a lesser known system-level approach called the *Chip-in-a-day* flow, which is focused on accelerating the design of dedicated hardware. It is shown that the distinguishing aspect of this technique is that it attempts to provide system-level abstractions for physical performance. This section concludes by comparing the designer-productivity for three chips made with this flow with a fourth made with a traditional flow.

2. A model for analyzing productivity

When we examine the productivity of designers on large, dedicated hardware projects, we see that most of their time is spent repeating lengthy cycles of a design flow as they explore the design-space. Whether a system-designer is choosing the number of function units, a hardware-designer is writing lines of RTL code, or a physical designer is creating a floorplan, all must make decisions and test their assumptions with a variety of CAD tools to see if they are correct. If we examine the time spent in these cycles more closely, we find that the majority of time is not spent analyzing the data and making new decisions. Instead, the majority of time is spent figuring out how to make each CAD tool run the desired test.

Figure 1 shows a portion of an automated design flow. The boxes represent design data, while the lines represent steps in the flow. Here we use solid lines to represent fast automated steps, while the dashed lines represent steps in which time-consuming user input is required. This figure illustrates a designer creating an initial specification, then using an automated synthesis step to create an implementation. When the automated flow completes, the designer makes a judgment about the performance of the implementation and then may or may not use this information to modify the specification.

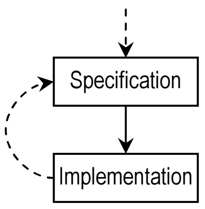


Figure 1: A simple automated design flow

There are four ways to increase a designer's productivity in such a flow:

- Reduce the length of time needed to generate the user-input (dashed lines).
- Reduce the number of iterations needed to converge on the final implementation.
- Accelerate the automated steps.
- Remove steps from the flow.

The remaining sections discuss how various system-design techniques attempt to improve productivity in one of these four ways.

3. SystemC design flow

SystemC^{TM} [2] is a C++ class-library that allows high-level modeling of hardware and software. The most popular use of SystemC^{TM} is to co-simulate hardware and software, encapsulating low-level models of the hardware and bus inside software system-calls, which is called Transaction-Level Modeling [3]. Figure 2 illustrates how we might use SystemC^{TM} to improve productivity when designing the dedicated-hardware portion of a project. A traditional system-level specification, such as a C or C++ simulation, requires manual translation to RTL code before a standard-cell netlist can be synthesized, as illustrated in Figure 2(a). However, by using a SystemC^{TM} specification and a synthesis tool such as the CoCentric^{TM} compiler from Synopsys^{TM} [4], the RTL authoring and optimization steps can be removed, thereby accelerating the flow as shown in Figure 2(b). In these figures, the parts of the flow to optimize the system-level specification have been omitted for simplicity, since they should be nearly the same.

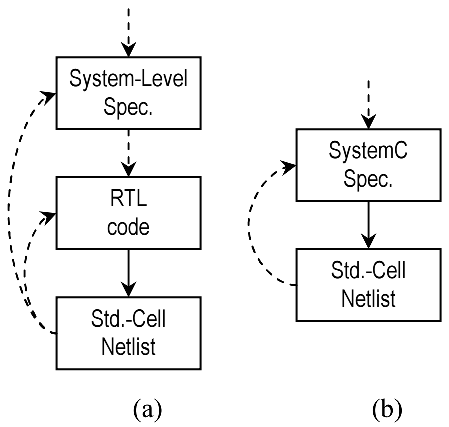
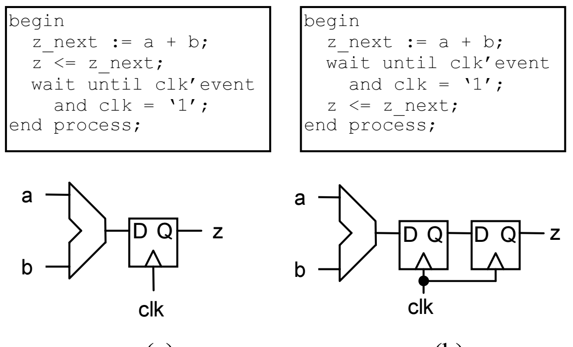


Figure 2: A traditional flow (a) compared with a SystemC flow (b).

The flow in Figure 2(b) assumes that behavioral synthesis is used to generate the netlist. Although SystemC^{TM} and CoCentric^{TM} do allow RTL modeling and synthesis, it is likely that the system-level code would be quite different from the RTL code, requiring manual translation as in Figure 2(a). Many argue that they are less productive using SystemC^{TM} in this way, since it is much more verbose that Verilog for RTL modeling.

Unfortunately, many designers feel that it easier to optimize a system without behavioral synthesis. This is because some changes in behavior that may seem insignificant to a designer can cause the compiler to produce circuits with significant differences in performance. Figure 3 shows two similar VHDL behaviors synthesized with Behavioral Compiler^{TM} from Synopsys^{TM} [4]. We may assume that the use of SystemC^{TM} and CoCentric^{TM} would produce a similar effect. Although the code for the examples is nearly identical, the architecture in Figure 3(a) has less area, less latency, and consumes less power than the architecture in Figure 3(b). An experienced designer would never use the code in Figure 3(b), but for the inexperienced designer, there is nothing in the behaviorally abstracted code that alerts him or her to the fact that a slight change would improve performance. In addition, behavioral synthesis is typically much slower than RTL synthesis, which deters designers from spending the time to gain an intuition of good coding styles. For these reasons, the use of SystemC^{TM} seems most appropriate when we have a behavior and need to build a chip quickly with little regard to its performance. However, if we need to build a chip that outperforms another, we are unlikely to choose this approach.



```
begin
  z_next := a + b;
  z <= z_next;
  wait until clk'event
    and clk = '1';
end process;
```

```
begin
  z_next := a + b;
  wait until clk'event
    and clk = '1';
  z <= z_next;
end process;
```

(a) (b)

Figure 3: Two similar behaviors synthesized with Synopsys^{TM} Behavioral Compiler^{TM}.

4. Platform-based design flow

Platform-based design [5] is an approach to system-level design that encourages extensive, planned design reuse. Tools are beginning to emerge, such as Virtual-Component Co-design (VCC) from Cadence^{TM} [6], that offer a glimpse of an automated platform-based design flow. We may project that, given more widespread availability of IP, that VCC will eventually be able to provide a flow like the one in Figure 4.

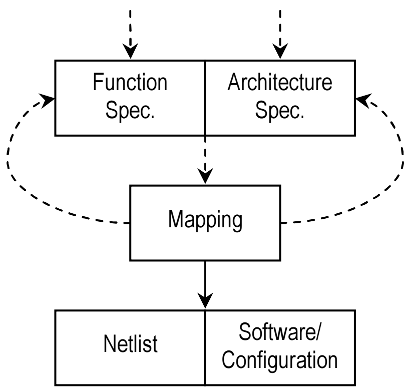


Figure 4: A Platform-based design flow.

As with SystemC^{TM}, a Platform-based design flow attempts to treat behavior and architecture as independent. The flow in Figure 4 begins with the development of function and architecture specifications. In VCC, the function specification is annotated C-code called “White-C”, and the architecture is specified as a high-level collection of IP-blocks (called “virtual components”) with some communication resources connecting them. Once both the function and architecture are specified, the designer creates a mapping of the function onto the architecture and may request analyses of system performance, such as an estimate of processor utilization for functionality mapped into software. The designer may map more functionality to an under-utilized processor or may add another virtual component to accept some of the functionality of an over-utilized processor. This flow improves productivity by reducing the time required to generate user input. The user-input cycles in Figure 4 would proceed much faster than the cycles in Figure 2(a) and (b), because the specifications and mapping have very little detail and are easy to change.

Once the mapping is finalized, we can envision an automated step that synthesizes the interfaces to the virtual components, using tools like CoWare^{TM} [7], or connects them with a special interconnection component, such as the Silicon Backplane^{TM} from Sonics^{TM} [8]. This synthesis step could also generate the software or configuration bits for the various programmable components in the architecture. Note that Figure 4 shows no user-input cycles after the generation of the netlist and software. Assuming that the virtual components are well characterized, there is no need to analyze the performance of the netlist or software in order to change the function or architecture specification.

This flow will generate high-performance hardware, given a fixed library of virtual components. If we want to determine what new virtual components to build, however, then it becomes less clear how this flow helps to achieve high-performance hardware. In order to develop a highly-efficient, flexible architecture with this flow, we would need to be able to map functionality onto non-existent, dedicated hardware and then obtain performance estimates. Unfortunately, obtaining these performance estimates is very difficult. Most of the currently available Platform-based design tools use behaviorally abstracted function-specifications, which make it difficult to converge on efficient behavior-architecture pair. Instead, we need a way to specify behavior that makes it easy to predict the most efficient architecture.

5. Chip-in-a-day flow

The goal of the Chip-in-a-day flow [9] flow is to enable feasibility studies from high-level descriptions by aggressively automating a flow to produce fully functional mask-layout within a day. This flow differs from other system-level design techniques in that its function specification is a discrete-time signal-flow graph rather than behaviorally abstracted code. With this flow, behavior and architecture are not independent and must be optimized together. As such, this flow does not attempt to reduce the number of iterations and instead attempts to improve productivity by removing steps from the flow.

Figure 5 shows an example of how this flow removes design-flow steps. Figure 5(a) shows a flow similar to Figure 2(b) with added layout steps. This flow begins with entry of the system-level spec, which was performed with the signal-flow graph editor Simulink^{TM} from the Math Works^{TM} [10]. This specification is then mapped to an RTL description and synthesized to a standard-cell netlist, using an approach very similar to the one used by the tool System Generator^{TM} from Xilinx^{TM} [11]. This approach removes the RTL-optimization cycles as the SystemC^{TM} flow did in Figure 2. However, this is no guarantee that this architecture will work once wire capacitances are included. To prove the correctness of the architecture, the designer must create a floorplan, which can then be routed automatically to create layout. A second user-input cycle illustrates the need for the designer to modify the floorplan depending on the quality of the layout. If no floorplan can be found to make the architecture work, then the system-level specification must be changed. This flow ends up being very time consuming, because the designer cannot begin to create the floorplan until after the netlist has been created. A time-consuming cycle to perfect the floorplan must be performed every time the designer wishes to see how a change to the system-level specification affects the system performance.

The Chip-in-a-day flow attempted to remove this cycle by creating a system-level spec that combined floorplanning information with the discrete-time signal-flow graph, thereby eliminating all but the outermost user-input cycles, as shown in Figure 5(b). This flow was never completely realized but was instead approximated as shown in Figure 5(c), which merges the last floorplan with the current netlist on each iteration of the flow. The hope was that, if the automated flow were fast enough, it could give the appearance of a floorplan and system-level specification being developed side-by-side. Unfortunately, for large designs, the generation of the standard-cell netlist and merging of the last floorplan would take 30 minutes to an hour to perform. Designer effort was saved only if the system-level specification changed very slightly. Significant changes still required the design-flow cycle to perfect the floorplan.

There are three main drawbacks to this flow compared to the other system-level techniques. First, it requires the creation of a floorplan, which is difficult and time consuming. Users often forgo the generation of layout because it takes too long. Second, the flow is difficult to maintain. It requires extensive scripting of physical design tools that continually change as technologies evolve. Third, it has no approach to deal with software. However, Simulink^{TM} does have software generation capabilities, which could lead to software generation methods in the future.

Even with these drawbacks, the Chip-in-a-day flow is much better suited to the development of highly-efficient, dedicated architectures than the other system-level techniques. This is because the discrete-time signal-flow graph leads to a much more predictable architecture than behavioral code does. This predictability comes from hints or abstractions of the physical performance in the system-level specification.

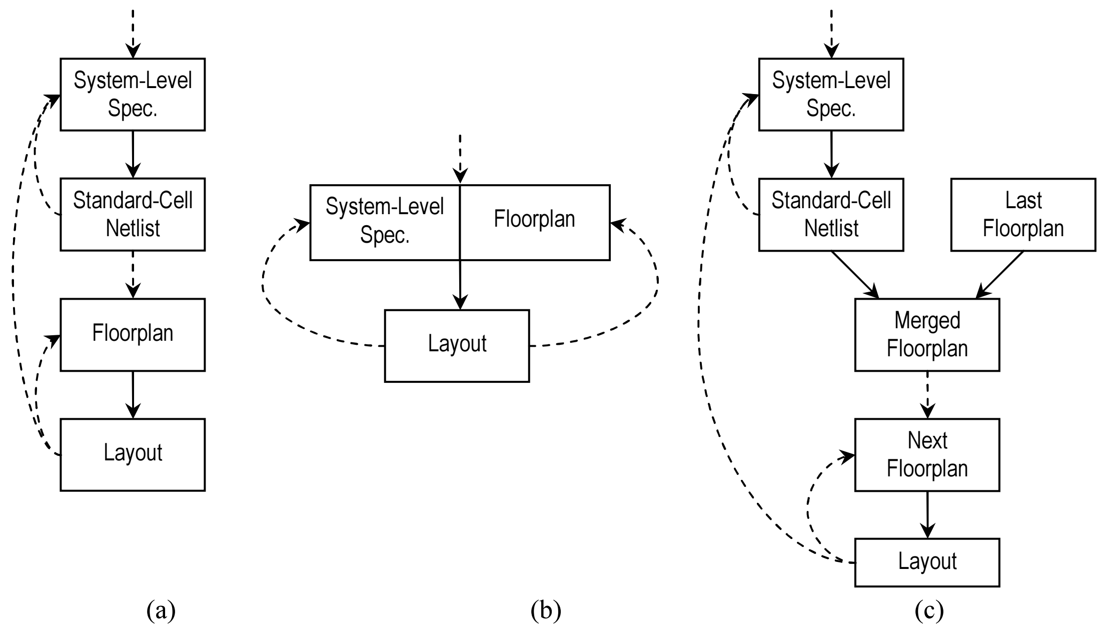


Figure 5: Illustration of removing steps in the chip-in-a-day flow, before (a), ideal (b) and actual (c)

Table 1: Comparison of design effort for chips made with and without the automated flow.

	MUD (block-based flow)		SOVA (automated flow)		TDMA (automated flow)		LZ-DECOMP (automated flow)	
	no. of designers	person-months	no. of designers	person-months	no. of designers	person-months	no. of designers	person-months
Lead	1	13	1	10	1	13	1	10
Sub-block	3	19	1	3	2	6	-	-
Physical	1	7	-	-	-	-	-	-
Total	4	39	2	13	3	19	1	10

Table 2: Comparison of complexity and design-productivity for chips made with and without the automated flow.

	MUD (block-based flow)	SOVA (automated flow)	TDMA (automated flow)	LZ-DECOMP (automated flow)
Design-Productivity	130,000 xstrs./person/year	600,000 xstrs./person/year	580,000 xstrs./person/year	420,000 xstrs./person/year
Transistors	410,000	340,000	630,000	2,800,000
Die Size	4.6 mm x 3.4 mm = 15.5 mm^2	1.9 mm x 1.9 mm = 3.5 mm^2	3.7 mm x 3.7 mm = 13.8 mm^2	5.2 mm x 2.5 mm = 13.2 mm^2
Core Size	3.7 mm x 2.5 mm = 9.1 mm^2	1.0 mm x 1.0 mm = 1.0 mm^2	1.8 mm x 1.3 mm = 2.3 mm^2	4.7 mm x 1.9 mm = 9.0 mm^2
Process	0.25 μm, 6 metal	0.18 μm, 6 metal, low-threshold	0.18 μm, 6 metal, high-threshold	0.18 μm, 6 metal, high-threshold
Supply	1.0 V	1.0 V	1.0 V	1.8 V
Frequency	25 MHz	500 MHz	25 MHz	100 MHz
Power	10 mW (simulated)	800 mW (measured)	15 mW (measured)	560 mW (measured)

Such abstractions include

- Operator depth to denote delay
- Size of the model to denote area
- Operator locality to denote locality in floorplan
- Operator activity to denote power
- Commands to indicate use of particular circuits, such as ripple-carry vs. carry-lookahead adders or flip-flops vs. SRAMs

Even if the designer does not execute the complete flow, he or she can still roughly predict the performance of the architecture based on extrapolations of past executions of the flow. These abstractions lead to an overall increase in productivity when designing dedicated hardware.

Tables 1 and 2 show the evolution of design productivity resulting from use of the Chip-in-a-day flow. All designs were dedicated-logic architectures for DSP applications, and all were designed by full-time students who had never designed chips previously. In all cases, the duration of the project is measured from concept to tape-out, including time spent developing the system-level specification. Time is measured in months during which the chip was the designers primary focus.

The first chip was multi-user detection CDMA chip (MUD) [12] designed with a block-based, semi-custom flow. This project required 4 hardware designers and one physical designer working a total of 39 person-months, as shown in Table 1. Once support for the automated flow was developed, a physical designer was no longer needed. The next three chips implement a Soft-output Viterbi Algorithm (SOVA) [13], a TDMA baseband receiver (TDMA) [14], and a Lempel-Ziv decompressor for maskless lithography (LZ-DECOMP). Table 2 shows design-productivity statistics for each chip in terms of transistors per person per year, along with complexity data for each chip. Because the LZ-DECOMP chip consisted of 8 identical rows that did not communicate, only one row was used to calculate design-productivity. Overall productivity for the LZ-DECOMP chip was lower, because this chip made extensive use of SRAM, which was not well supported by the automated flow.

6. Conclusions

This paper has presented a discussion of three system-level design techniques from the perspective of optimizing dedicated hardware. The popular system-level design techniques SystemC^{TM} and Platform-based design promise to alleviate the design-productivity gap by simplifying the mapping of behavior onto reusable, programmable cores. However, it is important not to sacrifice performance in the pursuit of greater productivity. The Chip-in-a-day flow uses higher levels of abstraction to accelerate the design of high-efficiency hardware. The primary means of this acceleration is the abstraction of circuit performance at the system level. It is possible that system-level design can be made more attractive for high-performance chips by focusing less on behavioral abstraction and more on abstraction of circuit performance.

Acknowledgments

Many thanks to Ben Wild, Josie Ammer, Engling Yeo, and Ning Zhang for their design-productivity and chip-complexity statistics, to Bob Brodersen and Borivoje Nikolić for continuing advice, to ST Microelectronics for fabrication services, to Cadence for design tools, and to DARPA for making this work possible through their support.

References

[1] *International Technology Roadmap for Semiconductors*, 2001, available at http://public.itrs.net
[2] SystemC^{TM}, http://www.systemc.org
[3] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*, Kluwer, Boston, 2002.
[4] CoCentric^{TM} and Behavioral Compiler^{TM}, from Synopsys^{TM}, http://www.synopsys.com
[5] H. Chang, *et al.*, *Surviving the SOC Revolution: A Guide to Platform-Based Design*, Kluwer, Boston, 1999.
[6] VCC^{TM} from Cadence^{TM}, http://www.cadence.com
[7] CoWare^{TM}, http://www.coware.com
[8] Silicon Backplane^{TM} from Sonics^{TM}, http://www.sonicsinc.com
[9] W. Rhett Davis, *et al.*, "A Design Environment for High-Throughput, Low-Power Dedicated Signal Processing Systems", *IEEE Journal of Solid State Circuits*, March 2002, pp. 420-31.
[10] Simulink^{TM} from the Math Works^{TM}, http://www.mathworks.com
[11] System Generator^{TM} from Xilinx^{TM}, http://www.xilinx.com
[12] N. Zhang, C. Teuscher, H. Lee, and B. Brodersen, "Architectural Implementation Issues in a Wideband Receiver Using Multiuser Detection," *Proc. of the Allerton Conf. on Communication, Control, and Computing*, Sept. 1998, pp. 765-71.
[13] E. Yeo, S. Augsburger, W. R. Davis, and B. Nikolic, "500 Mb/s Soft Output Viterbi Decoder," *Proc. IEEE European Solid-State Circuit Conf.*, Sept. 2002, pp. 523-26.
[14] J. L. da Silva, *et al*, "Design methodology for PicoRadio networks," *Proc. of Design, Automation and Test in Europe*, March 2001, pp. 314-23.