Rajesh Gupta: What does ESL mean to you?

Forrest Brewer: ESL, as I understand it, is all the parts of the design that essentially require system management and understanding beyond the conventional structure and the design. In other words, it consists of the tools and paradigms to enable design and design analysis of complex systems, methods to compose software and hardware, and—the part that's often missing—real-world models and artifacts, which are crucial if you're building systems. Our work has been in feedback-control systems recently. We need physical realizations in order to have some notion of a practical structure on which to base the assumptions we're making in the design. A lot of modern architecture work goes out to spec benchmarks and then constructions. By contrast, in controls there are no decent benchmarks for many-state problems. If you're not actually hooked to a system or capable of extracting detailed information about what the spectrum of values really are, then you can be steered way off course in what you think appropriate behaviors and optimizations would be for a control problem. For example, it is hard to structure a real-time system describing reliability of a braking system in a car without having some notion of the physical car system. ESL encompasses tools that allow you to actually pull models from real-time or physical systems, couple them into hardware and software systems, and then let you make decisions about the design that are appropriate within that problem. So, from my point of view, ESL tools enable you to actually build cogent designs and make system design trade-offs, which you would either not be able to do at all or would only be able to do with a great deal of manual effort.

Gupta: In other words, the value to you in ESL is one of design time or design cost.

Brewer: More of design understanding in the sense that I don't necessarily know where the important trade-offs would actually be.

Gupta: Don't you need to understand design whether or not you do ESL?

Brewer: You have to understand design in either case. But, for example, if I built a control circuit for a gyroscope that's microscopic, I obviously have to make it consume low power. However, it's not clear from the get-go that I can afford certain kinds of implementations, but when I look at the execution data, I really can. I'm speaking of tricks that would be stupid for spec benchmarks, or indeed for braking systems in a car, that end up being wonderfully good for certain kinds of MEMS (microelectromechanical systems). I wouldn't know that without analyzing the real-time behavior and statistics of the physical design, coupled with the practical limitations of the trial implementation.

Gerard Berry: I see two different communities talking about two different meanings of ESL. One corresponds...
to what Forrest was saying: ESL is about overall system design, where a system may be a car where you have to do safe design, make trade-offs, do architectural studies, develop new hardware and software components, and so on. This is the meaning of ESL you’ll find at many conferences. But I’ve found a different type of meaning used by my customers, which include big hardware companies like Texas Instruments or STMicroelectronics. First, for them, the notion of a system can be a little bit different: the system is simply the most complex thing they are designing within their industry. So, typically, they will define a system as one telephone platform, not something much bigger. The definition is often relative to the industry. Second, the related notion of ESL there has been quite fuzzy for a long time, but it seems to stabilize to ESL as dealing with any abstraction level above RTL, including kernel software, and ESL synthesis as dealing with RTL synthesis from ESL levels.

These companies build complex embedded platforms. Their three main needs are not all related to classical EDA. The first need is fast system-level simulation. Since classical software simulation tools don’t work anymore because of design size, they must be replaced by much faster simulation tools and must include increasingly more kernel or application software, not just simulate the hardware. Second, there is a complementary need to build hardware simulators that will simulate the hardware before production, while supporting software development as well: one cannot wait any more for the hardware to be finished before developing the software. The third need, which is quite different but also essential, is to raise the design and verification level from the RTL to more behavioral levels, in the same way as the level was previously raised from transistor to gates, from gates to RTL, and so on. Our customers refer to these three things collectively as ESL. This definition is more limited than the academic definition of ESL.

**Gupta:** This is interesting because there are commonalities here. Grant Martin and Gary Smith had classified ESL into five spaces: algorithmic design and implementation, behavioral synthesis, associated constructions and analysis, validation and prototyping, and functional data chip codesign.

**Arvind:** I basically agree with Gerard’s characterization. In generic terms, I see two different communities coming in contact with each other because of the changes that are taking place underneath. It’s a well-accepted fact that chips are getting more complex, and all modern chips require enormous amounts of software that must be developed at the same time the hardware is designed. Nobody’s going to tape out a chip unless they’re pretty sure that the required software will run on it. One needs simulation and testing infrastructure to enable this. There is also a need for a hardware model on which all software, not just what is needed for tapeout, can be developed and tested. For this, the simulation speed is the paramount problem. One often compromises on the accuracy of the hardware model for gains in simulation speed. One of the perpetual tensions is that if you don’t have enough fidelity, then the hardware guys completely ignore the models being used by the software people. This often leads to an extreme divergence of models. Consequently, when the real hardware appears, the software does not run it. Hopefully the ESL community can address these issues so that we can have systems models for the chips that are faithful enough that when the hardware comes along, the actual software will run on it.

**Gupta:** These answers fall into two broad categories: what is ESL and why ESL? Regarding the “why” part, Forrest mentioned the ability to do trade-offs or to gain a better understanding of the design so that you can make those trade-offs. Exactly how you do that and how much progress has been made are debatable. The “what” part—whether three things or five things—is a different issue. Are we missing anything in the “why” part? For example, we clearly want to enable better trade-offs. Because we want to raise the abstraction, does ESL enable new people to do some new things, or does it enable people who are already doing things on trade-offs to do a better job of it? Of course, the answer
could be both: clearly you want to enable architects to do better design trade-offs, but you also want to enable programmers to do software design or even software architecture design even before the hardware is ready. When we made the transition to the RTL from schematics, we enabled logic designers to do code, and coders could do some hardware design. If ESL does succeed, in your mind, who are we enabling?

Brewer: I agree with your point, but you need to have a very sharp notion of design fidelity. You need a clear and unambiguous specification of what the abstraction level is. Moreover, you need executable renditions of how the communication channels work and how the underlying execution model runs. The software running on a design is a crucial part of the system. There seems to be an idea that if you build correct VHDL, you can hook the models up to build systems that work. My experience, though, is that you have a lot of crud at the module level, tremendous software overhead in device drivers, and other band-aids because people build essentially inappropriate or insufficient models at the abstraction level between the software and hardware connections, or indeed firmware and hardware. Firmware construction is typically a nightmare because you have parallelism and synchrony and other things you are trying to conquer, often at the wrong abstraction level. It wasn’t correctly handled in the hardware, so then you had to band-aid it at the software level. If you can solve that problem of fidelity so that people can do it, then you can enable a software system designer to create hardware, or indeed to create systems.

Gupta: Arvind, who are we enabling? Are we enabling the architect? Are we enabling the design?

Arvind: It is hard for me to answer this question. We all agree that chip complexity is increasing and that if we keep using old design methods, we can’t exploit the latest silicon technologies. In some sense, there is no choice in the matter; we have to raise the level of design. As Gerard articulated, that means we have to abstract away lots of things. We do not want people to think in terms of transistors. We do not want them to think in terms of gates. We do not want them to reason about what happens in each clock cycle. The design abstraction must be raised so that the designers can focus on high-level functionality and high-level performance concerns.

Gupta: Why is that? We have had complex designs in the past, and we build billion-transistor chips today.

Arvind: Right, but it’s a question of the development cost, which is the number of people involved multiplied by the time it takes to do the design. Typical microprocessors teams involve hundreds of design engineers. Now we want to do the designs at the same scale or larger with a team of 10 to 20 people. The only way this can happen is through extreme automation and extreme reuse of existing intellectual property.

Gupta: So, Gerard, are your customers complaining to you that the design costs too much?

Berry: Oh, yes, but not only design—both design and verification. Design may become dominated by the cost of bugs. Circuit design used to be bug free, but now the bugs tend to appear more quickly than we can remove them. The only way to reduce the number of bugs is to make design description more compact, which requires raising the abstraction level. Another key issue is that bugs are found much too late, when fixing them is very expensive because one has to redo the whole loop. Finding a bug late is always much more expensive than finding it early. In the ESL approach, verification is done early at the specification level, and one tries to get rid of bugs before generating the actual circuits.

Gupta: What about reuse?

Berry: Reuse is fine in theory, but it is not yet working that well in practice—for example, because many designs do not have good behavioral specs. In the ESL movement, the focus is on building better specs and raising the abstraction level, not only for simulation but also for actual circuit production. ESL technology aims at simplifying design descriptions, automating many more synthesis steps, and making it possible to simulate and to find bugs much earlier. I think that most ESL tools try to do that.

Arvind: Whenever I say “chip design,” I am always including all the low-level software that is needed to make it all work. It is no longer just a question of laying out gates or anything like that. Everything has to work; otherwise, you don’t have a chip.
**Berry:** There are three technical challenges: The first one is to reduce the diversity of models and the inconsistency between models. When developing transaction-level software models together with RTL models, one has to deal with several models of different natures that are not easy to show equivalent. So, a challenge is to make it possible to use a single high-level model or set of models for simulation, synthesis, and verification. Also, we need to make the models high-level enough for the architects to look at them, use them in simulation and what-if analysis, and make trade-offs from them. Reducing the distance from architecture to design is important.

The second challenge, which is a consequence of the previous one and absolutely nontrivial, is to develop the appropriate computation models, design languages, synthesis, simulation, and verification algorithms for this scheme to work. Subsetting existing languages or modifying existing tools will not be enough.

The third challenge is a big issue: understanding how to manage large-scale concurrency. There are two very different kinds of concurrency in circuits. The first is synchronous concurrency, which is deterministic and thus relatively easy to verify, even formally. Then there is asynchronous concurrency—for instance, for the communication between different units or different clock domains in SoCs. Asynchronous concurrency makes verification very hard, and programming models for it are not well-established. For instance, building multiple cores in a microprocessor is useful, but it's very unclear how to efficiently use the architecture from a software-programming point of view.

**Gupta:** Are you advocating a new language? Don’t we have enough languages?

**Berry:** No, we don’t. I’m absolutely sure that we have enough variants of the same language. We have enough variants of C; we have enough variants of HDLs. But neither C nor an HDL is up to the goal: C has really no concurrency model that is at the right level and understandable; managing concurrency with threads in C is a nightmare. HDLs lack a nice sequencing model. And both C and HDLs cannot conveniently handle complex control descriptions that involve deeply nested concurrency and sequencing. For this, one needs real temporal primitives that are simply not available in C or HDLs. We need them to build intrinsically safe models.

**Gupta:** What are the challenges to improving system design or system-level designs?

**Arvind:** Anytime you walk in with a new tool or methodology, the first thing people want to know is how the results compare with the old way of doing things. In the case of Bluespec, this meant showing that synthesized RTL was as good as the hand-written...
RTL in terms of generated area and timing. This always reminds me of a story that John Backus once told me. He said no one was interested in the productivity offered by Fortran. The only thing people wanted to know was if a programmer could beat the compiler by hand. Once they were able to overcome this hurdle, people were amazed that they could program the machine without knowing how many registers it had. Once demonstrated, the efficiency question completely disappeared—so much so that later Fortran compilers were allowed to produce far worse code. Something similar is happening with Bluespec now. The moment people really start using it, it becomes clear to them that it's not worth focusing on the 5% gain that a hand optimization could garner. The added designer productivity completely overshadows any such gains.

In the context of productivity, there is one thing that is extremely relevant but hard to quantify. In current design flows, at some point an RTL design is finalized and handed over to physical-design and verification teams. After that point, any change in the RTL is a big deal because the subsequent steps are not totally automated. A single RTL change can cause major delays. This has two consequences: First, you cannot incorporate obvious solutions that become apparent later—for example, inserting another pipeline stage somewhere to reduce the critical-path delay is likely to cause a verification nightmare. Second, it becomes very difficult to deal with changing specifications. Although we may have perfect specifications for the floating-point multiplier of even an FFT block, when we design a complex system, the specification can change during the design cycle. The design may be underspecified, and its deficiencies may not be apparent until a significant amount of the design has been completed. If you have high-level design tools, it is possible to do fairly late binding of many design decisions. This is a difficult value to convey to designers because they might have not experienced it before.

Take PowerPoint, for example. It is sold as a productivity tool, but it has never increased my productivity: preparing a talk in PowerPoint takes longer than making hand-written slides. However, it is definitely superior in some other senses. Over time, my slides contained fewer bugs, because of the ease of mechanical cutting and pasting. It also became much easier to have many different versions of a talk. As my slide library built up, my productivity went up dramatically—not in terms of the speed it took to prepare a specific slide, but in terms of the whole talk. These are the sorts of productivity gains that are on the horizon. We can play “what-if” games and rapidly try out many designs. This is not possible today in hardware design using traditional tools.

Berry: I am often asked this question: “Can you do exactly the same thing as before but much better?” [laughter] But, unfortunately, that doesn’t work, and I totally agree with Arvind. The benefits are not obvious the first day, and that’s a major problem in convincing people.

Gupta: It is surprising to hear that in system-level design, somehow, the quality-of-results issue went away, because productivity matters so much. Have we crossed that bridge? Have we come to the point where we would tolerate less quality because productivity is so paramount?

Arvind: First, a clarification: I’m certainly not saying that Bluespec designs do worse than hand-coded RTL designs. In fact, Bluespec designs do rather well in this regard. If there were a huge gap in the quality of synthesis, the productivity Bluespec offers would have limited appeal. A point that is often missed is that the quality has to be compared against the work of only 80% of the designers and not against the champion designers. People are smart and can often find a way to improve a specific design, provided the design is small. But we can always raise the stakes: if the champion can handle a hundred thousand gates, give him a design that requires a million gates. Sooner or later, automation will win, just because no designer can deal with such huge things by hand. Bluespec is certainly at a level now where I would not concede...
even 5% more area than hand-coded RTL. In my experience, the issue of quality goes away once people start using the tool. In the long term, the productivity issue dominates because it has direct bearing on the cost.

**Brewer:** The metrics that you use to compare designs are highly variable. One thing ESL hits you in the face with is that what you’re designing for now isn’t the same thing you were doing three weeks ago. You may want a design that, for whatever reason, saves enormous power, and you honestly don’t care how much area it’s using in the technology. Power and area and other physical metrics are relatively understandable, but I wouldn’t say that high-level synthesis failed because the quality of results was bad. The quality was terrible because you couldn’t ask for what you wanted. Typically, there was no way to express, in the specification of the problem, what the design freedoms and constraints were. Well, I wanted this design, but I wanted this version of it that met some complex sequential-timing constraint. This wasn’t possible, and I think that many of the modern tools are much better at expressing timing behaviors or sequence behaviors or, in the case of Esterel, protocol behaviors that can be mapped mathematically into very efficient constructions. On the other hand, there are certainly metrics that are still missing. When people verify correctness of this machine, they verify that it works continuously, and it reminds me of the old Campus 1 box that Sun produced. It was a perfectly valid machine that had a nice floating-point unit in it, which would pass formal verification. Unfortunately, there was a bug, which meant that when you had a floating-point hazard of any sort, every single floating-point operation after that would create a hazard as well, so the actual performance was dozens of times slower in floating-point operations than you would expect. That’s a performance issue you can’t really simulate easily, because the conditions wouldn’t have shown up, and conventional formal verification wouldn’t have found it either. Nevertheless, what metric would, in fact, find that? I’d say that there are a lot of questions that are not easy to capture.

**Arvind:** Well, I don’t know all the reasons why high-level synthesis failed, but I have never subscribed to the idea that you write some sequential code and then magically some wonderful hardware comes out of it. My ingenuity as an architect is in specifying the appropriate microarchitecture for a given algorithm or a processor to meet some performance or other goals. Bluespec is a language for describing the structure and behavior of a microarchitecture using high-level abstractions such as guarded atomic actions. This gives flexibility to the compiler in generating implementations and provides flexibility to the designers in doing modular refinements.

Let me describe my experience in using Bluespec to teach a course at MIT on complex digital systems. Two kinds of students take this class: those who have a fair amount of hardware expertise but not enough exposure to advanced programming languages, and those who have good expertise in programming languages but limited or no knowledge of hardware design. The hardware students complain about types and atomic actions but soon learn these concepts. Students who do not have a hardware background but are proficient programmers sometimes write enormous amounts of Bluespec code very quickly, but this code, even when it synthesizes properly, sometimes makes no sense to me. When I ask them, “What were you thinking?” they respond, “What do you mean what was I thinking?” If a student has no model of hardware in mind, it is highly unlikely that he can do a good design.

By way of analogy, this type of hardware design is like programming without knowing algorithms and data structures. Bluespec and similar tools can dramatically ease the life of a designer without impinging on his creativity. These are tools for people who know what they want to design and who want to quickly explore alternative designs to ultimately produce a good robust design.

**Berry:** Absolutely. People also think that tools make them smart, and that’s not true. If you want to have a
Gupta: What can EDA vendors or the EDA community at large do to achieve ESL-centric goals?

Arvind: Here is something that I see as a positive movement: SystemC was an improvement over C because at least it had a model of concurrency. People could start writing modules with the goal of sharing it with others. This goal of sharing was not really achieved, because the semantics of SystemC are grossly underspecified in some critical sense. Now there is a fairly strong consensus to add transaction-level interfaces to these SystemC modules. This is again a move in the right direction, but it still falls short, because SystemC transactions do not have well-defined semantics. We may get the language we want if we keep stumbling this way, but a better way would be to start with fundamental principles and incorporate the kind of concurrency model and other features we need for ESL design. I think this is one of the challenges that Gerard had posed earlier. Things would be much saner for designers and tool developers if a well-developed model of concurrency were incorporated in ESL design languages. We need to agree on a set of concurrent primitives, combinators, models, or whatever you want to call it.

Berry: One thing ESL could do now—maybe it was too early before—is build solid vocabularies and formalisms. Currently, if you ask somebody, “Can you really define what transaction-level modeling means?” you may get a lot of different vague answers, and that’s still a big problem. I think the hardware community altogether has not been historically strong with semantics. A design used to be considered good if it was well-understood by its designer. But that’s not enough anymore. With higher abstraction levels and tools, you need to be much more precise; you need to know exactly what you say, and all readers or reusers of a design should understand it in the same way. In particular, hardware vocabulary is still quite sloppy in many places. Another important issue is that no strong enough asynchronous concurrency exists, even in research. The main problem ESL is facing in this regard is that there is no easy solution in sight.

About the participants

Rajesh Gupta, our moderator, is the Qualcomm Professor of Computer Science and Engineering at the University of California, San Diego.

Arvind is the Johnson Professor of Computer Science and Engineering at the Massachusetts Institute of Technology.

Gerard Berry is chief scientist at Esterel Technologies in Villeneuve-Loubet, France.

Forrest Brewer is a professor in the Department of Electrical and Computer Engineering at the University of California, Santa Barbara.