March 12, 2013

HLS versus OpenCL

Xilinx and Altera Square Off on the Future

by Kevin Morris

If you have a visit with Xilinx and Altera these days and ask them about FPGA design methods above and beyond RTL, you'll get very different answers. Xilinx will tell you they're having great success with high-level synthesis (HLS). Altera will tell you that OpenCL is the wave of the future. Both sides make compelling arguments, which sound like they have nothing whatsoever in common. What does it all mean?

We all know that RTL design is tedious, complicated, and inefficient. We've known it for twenty years, in fact. To paraphrase Winston Churchill: RTL is the worst possible way to design electronics - except for all of the other ways that have been tried. (OK, and we know - Churchill was actually paraphrasing someone else. See? IP re-use works, even in politics!)

But, if we know that RTL is so bad, we also know somewhere, deep down, that we will need to do something different someday. But what? and When?

About two decades ago, the hip answer to that question was "Behavioral Synthesis." We had only recently switched from schematic design to language-based design, and we knew we weren't going back across THAT bridge. We had raised our level of abstraction twice already. First - from gate-level design (ANDs, ORs, and NANDs) to MSI-level design (adders, multipliers, registers), and then, with the advent of logic synthesis technology, to language-based design at the register-transfer level (RTL). Each time we raised our level of abstraction, we reduced the amount of detailed design work required. We went from hundreds of pages of gate-level schematics to tens of pages of MSI-level schematics. When that exploded to hundreds of pages, we went to hundreds - and then thousands of lines of RTL.

As our RTL exploded to tens of thousands of lines, and our brains lost the ability to understand the whole design at once, we asked the obvious question "What's next?"

"Behavioral Synthesis," said one camp. You'll still be doing language-based design, but instead of writing tens of thousands of lines of detailed RTL, you'll be writing a few dozen to a few hundred lines of "behavioral" code. Instead of describing the detailed architecture of your system, you'll simply tell the synthesis tool what you want the design to do - it will synthesize the architecture for you. That approach sounded fantastic. However, it required the invention of a new class of EDA tool - one that was sophisticated

enough to replace an engineer's judgment and creativity on making an architecture that would balance the conflicting goals of functionality, power, performance, cost, and reliability.

University students were dispatched, there was an explosion of research, and a number of behavioral synthesis products limped onto the market. Early adopters were brutally punished. Behavioral synthesis was most definitely not ready for prime time. Only the very smallest of designs could be synthesized - a few dozen lines of code. The resulting architecture was... let's say... "limited." We could create a datapath with some muxes to share expensive resources like multipliers. We could create a simple state machine to control the datapath. We could support one clock domain only, and we could do a limited amount of user-directed loop unrolling, pipelining, chaining, and other optimizations.

If you wanted to design an FFT, these tools could do it almost as well as hand-written RTL. The FFT demo was shown far and wide. The problem was - not many people had a desire to design an FFT. They already had one, and "almost as good as RTL" wasn't a great motivator. And, if you wanted to design anything else - well, you were pretty much out of luck.

Mostly at the expense of Synopsys's "Behavioral Compiler," the term "Behavioral Synthesis" became anathema. It was placed on the "banned terms" list for all EDA-eternity - along with "Framework" and other spectacularly failed ideas that had been aggressively pursued by the industry. However, the idea of behavioral synthesis persisted. The prospect of a power tool that could quickly explore architectural tradeoffs - without the necessity to write and re-write thousands of lines of RTL in the process - was just too enticing to ignore. We scraped off the "behavioral synthesis" signs and replaced them with "high-level synthesis" (HLS). There! Think anybody will notice?

High-level synthesis aficionados had learned the lessons of the behavioral synthesis debacle. They were not going to go up in flames spectacularly a second time. They toiled quietly in back rooms - working to fix the myriad issues with their tools. They needed to synthesize more complex and diverse architectures - not just datapaths. They needed to synthesize the IO portion of the design - to be able to make hardware that supported interface standards and protocols. They needed to match the processing power of their synthesized architectures to appropriately-sized pipes getting data in and out. They needed to change source languages from something nobody wrote: behavioral-level VHDL and Verilog - to something everybody wrote: C and C++. And, they needed to find ways to help the designer understand the tradeoffs that were being considered - speed versus area versus power consumption, latency versus throughput, memory versus logic. The design space was vast and varied, and the tools could easily create thousands of unusable solutions in the quest for a good one. The "to-do" list for the tool technology exploded. We didn't hear much from the behavioral synthesis folks for a long time.

Meanwhile, back in the real world, where design teams still had to get products to market, another way of raising the level of design abstraction had caught on. It was rather obvious, really. Clever engineers had started to avoid redesigning the wheel. They took large blocks of RTL that had already been proven to work, bundled them up as reusable IP, and evolved a methodology for plugging them together to create complex systems - with very little requirement for detailed RTL design. Schematic-style design was reborn - only the "components" were now complex subsystems - processor cores, peripherals, busses, memory structures... and each of these was described and implemented in RTL. A typical system could be 80-90% constructed from re-used, previously-designed, already-verified IP blocks. The team could then focus their design efforts on writing RTL for the remaining 10-20% that actually constituted new, original content (and this was usually where the product differentiation was realized.)

This beefing-up of the IP-reuse infrastructure had the effect of marginalizing the high-level (behavioral) synthesis efforts, even as that technology began to mature to usable levels. HLS was scoring some significant victories - particularly among design teams doing complex, high-performance algorithms that needed to be implemented in hardware in order to achieve the required performance. DSP problems that were at or beyond the capabilities of the most sophisticated DSP processors were dropped into HLS tools with fantastic results. HLS build a cult following. But it was no longer viewed as a "system design" methodology where one would describe and compile an entire system at the behavioral level. Instead, HLS was viewed as a way to quickly implement very high-performance IP blocks - without having to write detailed, custom RTL.

The HLS folks were no longer set on world domination and becoming the center of the design process, but they had found a niche that suited them well. Their tools were now designing real parts of real chips.

Of course, as always happens, while the HLS and IP folks were fine-tuning their higher-level-of-abstraction solutions for electronic design, the tectonic plates of technology shifted again. It seems that the 10-20% original hardware content that was implementing the differentiating features of each design had begun to decline. The system-on-chip revolution had given us very fast and efficient processors on our chips, and the lure of software as the primary means of achieving product differentiation was too strong to resist.

Now, system design was being driven by the software domain, and our hardware systems-on-chip were becoming less differentiated and subservient to the overlying software architecture.

If we view an electronic system design as a pure aggregation of desired functionality, we can see that we will often hit portions of that capability that cannot be accomplished in software running on a conventional processor. If our system-on-chip is an FPGA, or an FPGA next to a conventional processor, we have certain resources available to us. The question, then, becomes: "which part of our engineering team is tasked with creating the non-conventional-software portion of our design?" If the answer is: "The software engineers," one might intuitively favor Altera's OpenCL solution. OpenCL was, after all, designed for software engineers who needed to create very high performance algorithms running on massively parallel processor architectures. Those same software engineers should be able to transition fairly easily to targeting that OpenCL code to FPGA hardware. Altera has gone to great lengths to create a flow for OpenCL that minimizes the pain of getting from OpenCL code to optimized FPGA hardware - with little requirement for understanding of the underlying hardware design concepts. And, they've created a framework for easily integrating those new blocks into a processing subsystem as accelerators or co-processing elements.

If, on the other hand, the beyond-processor-capability tasks are left to the hardware engineering team, Xilinx's HLS-based approach might seem superior. Hardware engineers would have the architectural expertise to evaluate the results from an HLS tool that was synthesizing behavioral code into parallelized hardware architectures. A few years ago, Xilinx acquired HLS company AutoESL and has since integrated AutoESL's HLS technology into their own design flow. Xilinx also has invested a great deal in creating a smooth path for designers wanting to adopt this higher level of abstraction in their FPGA-based designs.

The two companies' paths, then, can actually be viewed as complementary. The difference is the target audience. Altera's strategy appears to favor the software engineer as the driver, and Xilinx's strategy appears to favor the hardware engineer. In the short term, it is unlikely that this difference will be the deciding factor in any team's decision to use one vendor or the other for the FPGA portion of their design. In time, it's entirely possible that both companies will add the other strategy to their portfolio. In the long run, though, it will be interesting to see which approach gets more traction in the design world - or if some new alternative comes along that renders both of these methodologies obsolete.

Channels

EDA (/design/eda). FPGA (/design/FPGA). Software (/design/software).

Comments:

Posted on March 12, 2013 at 6:28 PM

We won't be using RTL to design FPGAs forever. What do you plan to do to raise your level of design abstraction?

(/index.php/profile/3

kevin (/index.php/profil

Total Posts: 321

Joined: Apr 2009