ParC – Extending C++ for Complete System Design

Author: Kevin Cameron

Introduction

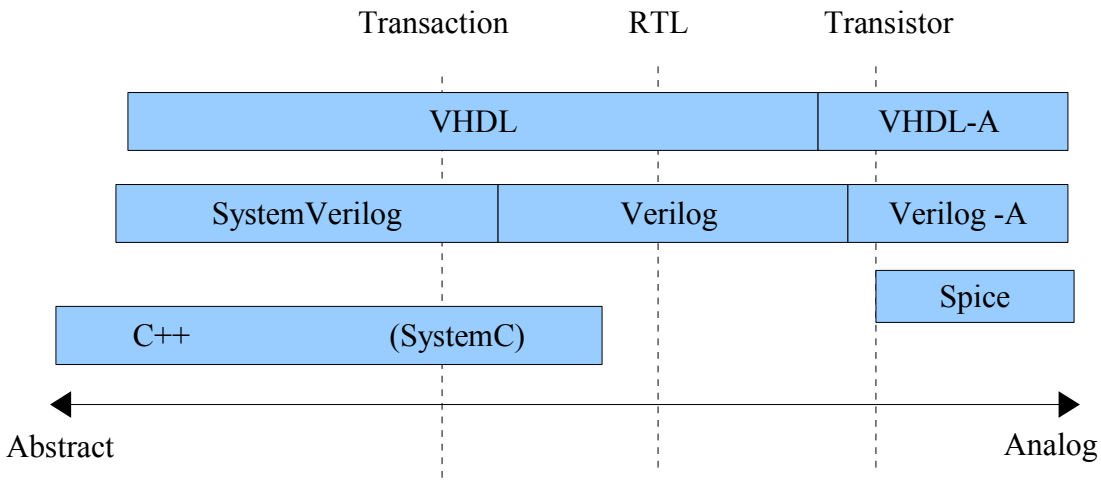
Over the last couple of decades electronic systems have become very complex, and all indications are that the complexity will continue to increase. Particular trends that are problematic for designers are the requirement for integrating multiple processor cores along with peritheral interfaces and memory as well as analog and RF circuitry for SoC (System-on-Chip). In addition to the increasing hardware complexity, these systems also have increasingly complex software, often targeting different processors for different functions like DSP and graphics. The combined complexity of the hardware and the software makes each generation of systems exponentially harder to design and verify.

A secondary trend in Silicon is that although devices get faster every time process geometries shrink, wiring resistance scales adversely, and it is harder to improve single processor throughput, so computing is moving to using multi-core chips and parallel processing to improve performance. In a decade the number of cores in the average computing platform will probably be in the hundreds.

HDLs and Programming Languages

Because all the tasks involved in designing complex systems were originally handled separately each tends to have it's own toolset e.g.: Spice for analog, Verilog/VHDL for logic design and C/C++ for software. Very few of the existing CAD tools are capable of parallel processing so in general they are not keeping pace with the complexity of new designs, and they are not well integrated with each other either. Most existing CAD tools are highly optimized single threaded applications that cannot easily be converted to take advantage of parallel processing hardware. Those that have been converted to run on SMP will will probably need a rewrite when SMP ceases to scale due to the cost of maintaining coherent memory, physical limitations and Amdahl's law– maybe at the tens of cores mark.

In order to be able to design systems when the relative performance of tools for simulation etc. continually declines system designers push up the level of abstraction at which they work to reduce the relative complexity and get the job done in reasonable time. This has lead to the use of C++ for modelling hardware (as SystemC) at system level, but that leaves a gap between the design level and the implementation level. A rough map of which levels of simulation abstraction are handled by which languages is shown below.



C++ is not shown as reaching down to RTL and below is because it becomes awkward for users to describe hardware at that level due to the lack of direct support for fine grained multi-threading. VHDL doesn't have the same expressive power as C++ and has some dysfunctional semantics where it crosses into the analog domain: it is not possible to bridge domains transparently or bidirectionally without timing errors as it is with Verilog-AMS (but Verilog-AMS has no user-defined types). SystemVerilog added a lot of the expressive power of C++ to Verilog but uses different syntax and semantics and most of the object-oriented features are aimed at verification rather than hardware description.

The main difference between C++ and Verilog or VHDL is that the HDLs support light-weight multi-threading within the language whereas C++ only supports multi-threading through the use of external library code which results in a much higher per-thread overhead: for a large design containing millions of gates an HDL can easily handle have a thread per gate whereas C++ would probably run out of memory or spend all it's run time in the mechanics of switching thread context.

Despite the fact that HDLs describe systems in terms of threads that logically run concurrently there are few successful parallel processing simulators, and similarly although C++ (as SystemC) has a fairly high cost in terms of the user effort required to describe a multi-threaded system there does not appear to have been much effort put into making it run parallel.

C and C++ are the primary languages for software design and are available from multiple vendors as well as from the open source community for free[1]. Although HDLs have acquired many of the capabilities of C and C++ few people would consider using an HDL for developing a regular software application. That means when considering a whole system there is usually a fairly well defined boundary between the hardware and software, possibly only crossed by the odd tool that does high level synthesis from C/C++ to RTL (Verilog/VHDL for hardware synthesis), and the tools used on the software side are very

1 Part of the motivation behind SystemC

likely to be completely different from those used on the hardware side.

Similar to the split between hardware and software are the internal splits between analog, digital and RF within the hardware design. This has partly been perpetuated by poor progress at organizations like Accellera and the IEEE at developing and combining the standards, e.g. Verilog-AMS was developed at Accellera but was neither transferred to the IEEE for integration with the IEEE Verilog standard or integrated with the SystemVerilog standard as it worked it's way through to the IEEE. Very little work has been done on integrating RF with Verilog (System or AMS), and very little has gone into providing back-annotation for SystemVerilog or Verilog-AMS and other issues related to power management - Verilog and VHDL date back to when digital circuits usually had only one set of power rails.

It is unlikely given the slowness of the language development process at the IEEE and Accellera, and the EDA companies dependence on the existing plethora of tools that VHDL or SystemVerilog will improve greatly in the forseeable future.

Methodology

Traditional digital design methodology is synchronous, i.e. the description of the desired behavior (used by synthesis or verification) includes the clocking schemes. This causes a couple of problems for the tools: it is difficult to extract the actual required function and it is therefore difficult to implement the design with a different clocking scheme (with clock gating) or asynchronous. Ideally, functionality and timing/power constraints need to be defined separately so that the functionality part is reusable, and constraints can be tweaked for different taret platforms.

Most SoC designs use a lot of predesigned blocks (IP), so most of the effort in completing a design goes into creating the interconnect between the blocks. Since the "reach" across a chip - how many devices you can communicate with in a clock cycle for a given frequency - shrinks as device dimensions shrink due to wire resistance scaling adversely, maybe multiple stages of data buffering are required between initiators and targets or peers. Describing how all this communication works is done most easily at the system level with asynchronous point-to-point software communication mechanisms that can be mapped into hardware communication mechanisms by communication synthesis tools.

None of the HDLs mentioned above or SystemC have primitive mechanisms for representing the point-to-point communication required for supporting asynchronous descriptions or communication modelling. Transaction-Level-Modeling in SystemC is the nearest thing, but is not very abstract (so not for programmers).

Parallel Processing

The main reason that parallel processing has not gained much traction in simulation is that the use of a synchronous/RTL design style leads to intrinsically unstable simulations that are hard to debug. Parallel processing simulation works better for descriptions with full timing information so that event ordering is (nearly) the same for the parallel and single threaded cases, however including that extra detail slows down simulation, which probably relegates parallel processing simulation to the backend of the design process for standard HDLs. A number of attempts have been made by people to build parallel simulators, but their general lack of (commercial) success has discouraged the larger EDA companies from developing such products, and as mentioned above the existing products are highly optimized as single-threaded applications that cannot be easily converted to do parallel processing. The efficiency of parallel processing solutions depends on the ratio of computation to communication, i.e. when a task is split between processors there is a communication and synchronization overhead and if the overhead is too high there is no gain in throughput, for that reason parallel analog simulation has received more attention since the matrix solvers in Spice simulators are very compute intensive and so give a good c/c ratio if the circuit can be partitioned (a non-trivial exercise).

Software tools for debugging multi-threaded software are not generally as good as HDL debuggers, although some SystemC debug environments are solving the problems. The major problem with parallel processing C/C++ is that the mechanisms for controlling cross-thread data access tend to be quite heavy-handed because the compilers do not understand multi-threading and all operations are performed by library functions[2] so developers tend to avoid using them and subsequent bugs due to indeterminate ordering (and overlapping) of data access can be extremely difficult to find when threads actually run concurrently. Multi-threaded applications that run with a single thread of control (no concurrency) are easier to debug but don't provide any performance gain for the increase in complexity.

2 Posix Threads - http://www.llnl.gov/computing/tutorials/pthreads/

CSP

Communicating sequential processes (CSP[3]) is a software design paradigm for parallel processing proposed by Tony Hoare[4] - this and other major research work on verifiable computing earned him a Knighthood. It is basically a methodology where the code that does the work is described in fairly small chunks of procedural code (single threaded processes) that communicate with each other by passing messages over point-to-point channels. CSP was partly conceived as a way to develop parallel processing software that would be formally verifiable since each process behaves much like an isolated state-machine. Symmetric parallel processing based methods (i.e. shared memory and mutexes) is a more popular paradigm than CSP since it mirrors how current hardware works (multi-core unified memory), and MPI[5] has gained some traction for applications needing message passing (for distributed computing), however neither of those approaches map very well into hardware/system modelling.

Message passing paradigms like CSP map well into an asynchronous hardware description methodology since the individual processes do not need to consider time the way that (say) processes in a Verilog simulation do, e.g. where a Verilog process might be sensitive to a clock and reads data synchronously from a signal, a CSP process is just sensitive to data arriving on a channel. It also has the advantage that any one process is essentially unaware of what kind of processes are on the other end of the channels it communicates over so it is easy to mix hardware and software processes which simplifies design refinement, and describing heterogeneous processor systems.

CSP channels also map well to serial busses like PCI Express, USB, Firewire etc. for modeling board level systems or ethernet etc. for networked systems.

MPI doesn't lend itself to hardware modelling because the channels are not declared, i.e. any process can send messages to any other process without having to declare the channel first, so it is hard to extract the communication requirements for synthesis.

Rationalizing the Design Flow

As described above, the current set of systems design tools are a poorly integrated hodgepodge, and are not improving in performance at the same rate as their supporting hardware. To fix this problem requires improving the performance and breadth of some (or one) of the tools so that others fall by the wayside. What the new/improved tools have to do is handle design from software to transistors in a unified environment using parallel processing and supporting asynchronous design methodologies.

Because the user base of C/C++ is considerably larger than that of HDLs and it seems extremely unlikely that software engineers will suddenly start using SystemVerilog or

3 http://www.usingcsp.com/

4 Sir Charles Antony Richard Hoare - http://en.wikipedia.org/wiki/C._A._R._Hoare

5 http://www.mpi-forum.org/

VHDL for programming, extending C++ to handle areas it doesn't currently support has the most appeal. In particular extending C++ to handle multi-threading (and parallel processing) within the language itself (rather than the current approach using libraries) would be a huge boon without considering the benefits in hardware modelling. Since C++ is otherwise a functional superset of most other languages it is not hard to translate other languages (Verilog and VHDL) into the extended C++ for backward compatibility.

ParC was developed for this purpose, initially as an ESL language to replace SystemC, latterly as a programmer friendly approach to parallel processing for multiple platforms (e.g. FPGA and GP-GPU). It is superior to other approaches like OpenCL, TBB and CUDA because it works at a higher level of abstraction and is not platform dependent.

ParC Extensions

The extensions needed for C++ to handle hardware modelling down to transistors are fairly straight forward and include the addition of the primitive elements seen in HDLs. These basic elements include:

- Signals
- Processes
- Modules

The additional functionality required to support asynchronous design and CSP style parallel processing is a simple fifo/pipe object. The use of inheritance and templates in C++ make it possible to produce a broad range of functionality without having to define many constructs - the language reference manuals for HDLs tend to be large volumes due to poor language design, and not giving the designers the capability to do things themselves.

The extensions above require modifying the C++ compiler frontend (which does the parsing), and also providing a runtime kernel that supports the multi-threading and coordinates the interaction of the processes with signals and channels. The runtime kernel is also responsible for managing parallel processing.

Other functionality that would be handled by the kernel (optionally) would be back-annotation (maybe initially from SDF).

ParC Additional Functionality

ParC is intended to be a superset of existing HDLs and C++ so that HDL (and SystemC) descriptions can be converted to ParC, or at least those parts used for hardware modeling. There are a number of features missing from HDLs and badly designed features that can be improved upon by moving to ParC, these include:

- Dynamic hardware reconfiguration
- Better assertion timing
- Better mixed signal support (multi-type signal resolution, inc. analog and RF)

The dynamic reconfigurability capability is useful for modelling hot-plugging or systems where components may change in function e.g. FPGAs, or for modeling component failure. Verilog and VHDL are static descriptions, which may be satisfactory for chip design, but really is inadequate for complete systems.

The assertion timing issue comes from some unnecessarily complex clock definition syntax and semantics in SystemVerilog and some semantic deficiencies in VHDL.

The improved mixed signal support comes from using different semantics for signal resolution[6] than VHDL (whose the hierarchical resolution and port-bound type conversion schemes make describing bidirectional signal flow near impossible), and supporting multi-type resolution which is not supported by SystemVerilog. Most analog simulators like Spice have internal models written in C or C++ so it's not much of a leap to use ParC instead of (say) Verilog-A, most of the hard work is done in the solver (part of the kernel), making the analog talk to the digital (cross domain resolution) is a problem already solved by Verilog-AMS but the ParC implementation can fix some of the issues with how it works with respect to power distribution not addressed by Verilog-AMS. Integrating RF (as spectral domain) requires being able to support complex arrays (for Fourrier Transforms) which is possible in VHDL and SystemVerilog as well as C++, but neither VHDL or SystemVerilog has the mechanisms for handling that kind of cross domain communication automatically. The job of locating the resolution functions and modules required to make the various forms of resolution work is that of the runtime kernel, and will be relatively transparent to users.

6 How the value of a wire connected to different sources is evaluated.

ParC Backend Support

ParC can be implemented by using a preprocessor or modified parser and a simple single-threaded kernel. To gain the full benefit of the multi-threading and parallel processing requires modifying the C++ debuggers so that they comprehend the threading model and the correspondence between code and thread instances - C/C++ debuggers generally don't understand lightweight threading other than Posix threads, making debugging somewhat awkward (though not impossible). SystemC debuggers already handle the multi-threading multi-instance case but are unlikely to handle the parallel processing or CSP aspects of ParC because SystemC does not directly support CSP style multi-threading (so the debugger would have to understand the intent of the users' code). The sc_fifo and TLM components in SystemC can be implemented over channels in ParC to enable parallel processing.

For systems using CSP style parallel processing the kernel can provide the hooks for analyzing traffic in the channels, but it is useful if the debugger understands them too so that it is easy to traverse from one process to another.

Debugging for analog depends on the capabilities of the analog solver (kernel), noting that there is likely to be multiple analog solvers running in parallel for a full system.

If ParC is compiled to a standard kernel API the code can be shared while protecting the designers' IP.

Conclusion

The current tools and methodology for system design are rooted in old approaches to design and are not keeping pace with the evolution of Silicon technology. Many problems in software design have been solved with C++ and it's associated tools, however those improvements have not been migrated effectively into the realm of hardware design, and C++ itself fails to exploit the parallel processing capabilities of modern computing hardware. ParC solves this problem by extending C++ to handle low-level hardware descriptions and fine-grained parallel processing, and simplifies the job of the designer by reducing the number of languages they need to learn to get their job done, and allows the use of a single development environment for software and hardware from ESL to RTL and transistors. ParC also unifies analog and digital design which is important for single chip SoC solutions and handling the growing analog issues with shrinking devices dimensions. The unified environment of ParC also makes it easier to reuse verification testbenches as the design is refined from the ESL to RTL levels.

ParC provides the basic building blocks for new asynchronous design methodologies and tools, in particular a CSP approach to design that is more easily formally verifiable, and supports high-level architectural analysis.

In addition ParC provides a platform for developing software applications that can take

advantage of new parallel and heterogenous hardware, like multi-processor game machines, or the latest generation of multi-core CPU, as well as FPGA and GP-GPU. ParC should ultimately replace existing tools like VHDL and Verilog because it will support much of the same functionality and will span a wider user base.