EE290C HW 1

Vighnesh Iyer

August 29, 2018

1 Chisel Simulations

1. Which backend would you use to run unit tests for a small block? Which backend would you use to run integration tests for a whole RISC-V core? Explain your answer.

   Use the FIRRTL interpreter to test small blocks for small amounts of time, since it becomes much slower than Verilator for more than 1k cycles or even a moderately complex block. Use Verilator for testing large blocks (such as a whole RISC-V core) since its simulation speedup is much greater than its startup penalty.

2. You can compute a rough estimate of the "frequency" of a simulation by dividing the number of simulation cycles by simulation time. You shouldn't include compilation and other start-up costs in this compilation time, so don't forget to subtract that out from the total simulation time. Use the 10 cycle simulations as a rough estimate of compilation time and other startup costs. Roughly, what frequency does the interpreter and Verilator achieve for each design complexity?

   For a 10 cycle simulation with a complexity of 51 taps, the startup time for the interpreter is 275 ms and for Verilator is 1426 ms.

   For the 10000 cycle simulation of same complexity:

$$\begin{aligned}\text{Interpreter Speed} &= \frac{10000}{(19510 - 275) \cdot 1\text{e}{-3}} = 520 \text{ Hz} \\ \text{Verilator Speed} &= \frac{10000}{(1625 - 1426) \cdot 1\text{e}{-3}} = 50 \text{ kHz}\end{aligned}$$

3. FPGAs can be a useful way to accelerate simulation. Let's say that building and deploying a design for FPGA takes at least 15 minutes and will run at 10 MHz. Approximately how many cycles do you need to simulate for FPGA emulation to be worthwhile for this design?

   Assuming the Verilator compile time is insignificant compared to the simulation time; let $V(c)$ be the time it takes to simulate c cycles using Verilator, and $F(c)$ for FPGA.

$$\begin{aligned}V(c) &= \frac{c}{50\text{e}3} \\ F(c) &= 15 \cdot 60 + \frac{c}{10\text{e}6} \\ V(c) = F(c) &\to c = 45\text{M cycles}\end{aligned}$$

2 Chisel Generator Bootcamp

The IPython notebooks and HTML are attached for bootcamp sections 1, 2.1, 2.2

3 Reading Notes

Summary with my commentary in blue.

3.a Bachrach - DAC 2012 - Chisel: Constructing Hardware in a Scala Embedded Language

1. Identifies issues with using Verilog/VHDL to write RTL descriptions of digital circuits
   (a) Originally designed as simulation languages so a strict synthesizable subset needs to be understood by the designer and tools. This is an issue with any HDL as it evolves; there are useful non-syntheizable constructs that must exist along with the design to simplify simulation (magic memory loading/backdoors), formal verif (assert/assume), design assertions, print-based debugging (as used in DESSERT), register initialization (for GLS and maybe FPGAs), linting waivers, behavioral model stubbing, etc.
   (b) Poor abstraction facilities compared to modern languages which leads to low reuse. Modern SystemVerilog provides facilities for OOP, structs, interfaces, and many other niceties, but I agree Scala is more powerful even still. Whether reuse is limited by abstraction limitations is arguable.
   (c) Verilog/VHDL aren't suited to creating generators to support design space exploration
2. To work around limitations of Verilog/VHDL metaprogramming layers and macros are often used. Definetely true and a big problem.
3. Prior attempts at constructing a DSL to generate hardware (Esterel/Bluespec) are too domain specific (stream processing, guarded atomic actions) and don't map well to other hardware types. I'm sure the Bluespec people would disagree. RTL isn't always the best abstraction either.
4. What is Chisel
   (a) Simple, integrated into a modern language
   (b) Low-level hardware construction that can capture high-level design patterns
   (c) Generates code for multiple targets. I think the C++ simulator has been deprecated since

The rest of the paper is an overview of what Chisel supports at the time. One thing I found interesting was the functional style used to describe various forms of hardware generation (foldR and map in FIR filter generation) and the module instantiation recursion used to generate a sorting circuit. These are both not possible in Verilog/VHDL and represent an elegant way to create those circuits. Although Bluespec/Cλash both support very similar, if not even more elegant ways to do this. Both are based on Haskell.

The paper makes a special point to mention the fast C++ simulator, which is now gone. I think this was a useful feature, although I'm not sure of how it compared with Verilator, and the new Treadle FIRRTL execution engine in terms of performance and usability. From the VCS result presented in the paper, the C++ simulation is much faster than I would expect Verilator to be; this is probably from the limited simulation support in Chisel vs Verilog's non-synthesizable constructs, since I expect VCS to have finely-tuned ILP optimizations too.

It is unsurprising that Chisel compares similarly with Verilog in VLSI QoR since it operates at the same level of abstraction. A comparison with other DSL based approaches would be more interesting.

3.b Izraelevitz - ICCAD 2017 - Reusability is FIRRTL Ground...

The idea here makes a lot of sense, which is to formally define a hardware IR which can be used to customize designs to specific targets and encourage reusability. The hardware IR is especially useful to write transforms for DFT/DFM, FPGA implementation (the double-pumped FPGA BRAM transform for many-ported memories is very neat), and constraint generation.

The major issue here is what belongs in the hardware IR. Do assertions, print statements, memory backdoors, etc belong there even though they aren't synthesizable? What about design or timing constraints (like for an async FIFO) that should be propagated to the synthesis tool? Do interfaces (bundles) belong in the IR as opposed to just separating the interface into wires?

The other major issue is adoption. For FIRRTL to be "LLVM for hardware" it has to become commonly used, which requires an easy on-ramp for already written Verilog/VHDL. If design intent is lost or non-syntheizable constructs are stripped, this becomes more difficult.

This paper is great at presenting the motivation and some of the underlying details of FIRRTL, as well as showing a few evaluation results (although a few are unfair or ovbious comparisons). But I would like to see the question of what belongs in an IR addressed more thoroughly, and have this motivate what differentiates FIRRTL from Yosys's IR or PyVerilog's AST.

3.c Wang - DAC 2018 - ACED: A Hardware Library for Generating DSP Systems

The case is made that design of systems should be integrated than split across several tools, algorithms, and abstraction boundaries. The paper covers how the ACED library helps perform all the system design in Scala and can easily share info from high-level architecture evaluation to RTL generation. The use of the Scala type system to cleanly parameterize DSP modules is a clear advantage over using other HDLs. I really like the diagrams in this paper.

The bitwidth optimization routines using interval types represent something that can only be done with a high-level language and transformation passes (impossible in Verilog without an external tool). The automatic bitwidth reduction passes are effective in trimming registers when the precision isn't necessary.

What struck me at the end was the generator result comparison between MATLAB, HLS, and ACED. If only looking at the area, HLS produced the best QoR. I would like to have seen timing slack comparisons between the generators. It is unclear if Vivado HLS performs any register trimming in the same way as ACED, or if it is just better at targetting FPGAs. In any event, I think the result is unsurprising; from what I've seen HLS is becoming very effective, especially in the DSP domain. It is mentioned that the result from HLS required specific pragmas and refactoring, but that may be OK if the total code size is much smaller, and if the problem is really the HLS tool generating sub-optimal schedules and resource allocation (that can be improved).