Designing a Simple Pipelined RISC-V CPU

Rishiyur S. Nikhil
Bluespec, Inc.

DRAFT: July 18, 2023, 22:00h EST

Contents

Chapter 1

Introduction

This book accompanies a course taught by Bluespec, Inc. on how to design a simple, pipelined, RISC-V CPU:

- ISA (Instruction Set Archictecture):
  - From the Unprivileged ISA spec: RV32IM (basic integer ops, integer multiply/divide)
  - From the Privileged ISA spec: a few items enough to handle traps and interrupts

  Note: This is adequate for embedded systems, but not enough for OSs like Linux, which also need Supervisor Mode and Virtual Memory, and can benefit from other ISA extensions such as Atomics and Floating Point; those can be covered in a follow-on course).
- Simple microarchitecture:
  - 5 stages for basic integer pipeline
  - Additional stages as necessary for other pipelines (memory, ...)
  - Simple branch prediction for control hazards
  - Simple register scoreboarding for register data hazards

Fig. 1.1 is an overview of topics to be covered. We will execute RISC-V programs on our RISC-V design, in three different ways:

1. In a simulator of RISC-V written in C (using the Bluespec Cissr tool).

   Although this does not directly provide any hardware design insights, this is invaluable for understanding the RISC-V ISA and providing a "reference model" against which to measure the correctness of our hardware design. It is also invaluable for RISC-V software development.

2. In a Verilog simulator (using the Verilator tool).

   This will provide an exact, cycle-accurate simulation of the very same design that we'll run on the FPGA. This is invaluable for debugging the hardware design, because the turnaround time to fix a problem and run a new simulation is very short (minutes) compared to creating a new version of the FPGA build (several hours).

   Of course, this will run much more slowly (10,000x or more slower) compared to the FPGA, and so is useful only for smaller programs.

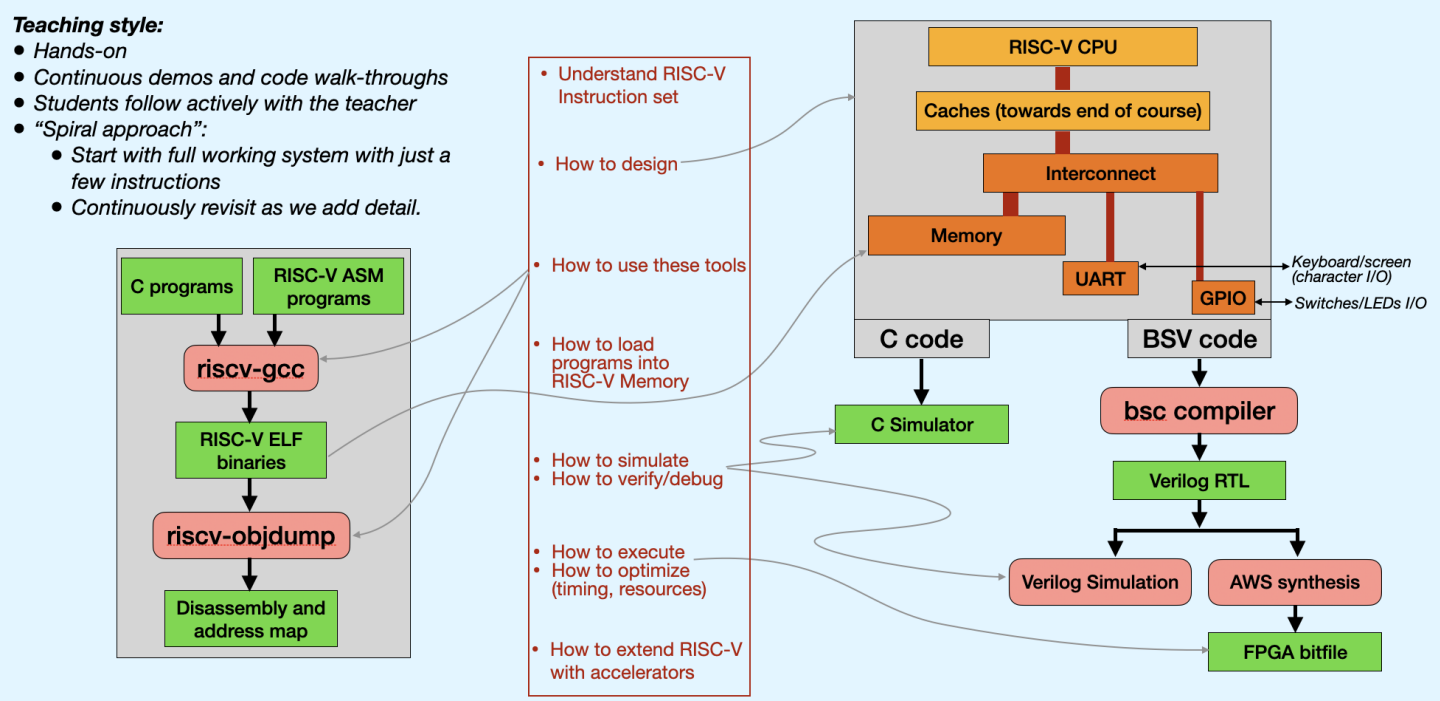

Figure 1.1: Topics covered (in red text in red box)

3. On an Amazon AWS FPGA. We will use the Amazon AWS design flow to create the FPGA bitfile, and run it on an "AWS F1 instance", which has an attached FPGA.

   This is our final hardware design.

Please see Appendix A for a detailed listing of resources (documents and software tools) needed for this course.

Appendix A

Resources: Documents and Tools

This appendix describes all the resources relevant to this course.

A.1 GitHub

We will be using GitHub extensively. Course materials will be provided in a public GitHub repository, and GitHub's "discussion" facilities can be used to for questions and answers, visible to all.

For students who do not already know how to use GitHub, we will teach the basics.

More detailed documentation can be found starting at: https://docs.github.com/en/get-started/quickstart

A.2 RISC-V ISA (Instruction Set Architecture) Specifications

We will refer to the Unpriviledged ISA very frequently, so you may wish to download a copy of the PDF for your laptop, and/or print a copy. The Privileged ISA document is not needed until later.

- "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA".
  Bibliography entry [13] contains a link to `riscv.org` from which to download a PDF.
- "The RISC-V Instruction Set Manual Volume II: Privileged Architecture"
  Bibliography entry [14] contains a link to `riscv.org` from which to download a PDF.

A.3 RISC-V Assembly Language Manuals

We will not do very much assembly language programming, and we will teach whatever notation we need during the course.

There are several RISC-V Assembly Language manuals available online, and some in bookstores; download them only if you prefer a local copy:

- "RISC-V Assembly Programmer's Manual", Palmer Dabbelt, Michael Clark and Alex Bradbury.

  Bibliography entry [2] contains a link to online manual.

- "RISC-V ASSEMBLY LANGUAGE Programmer Manual Part I", Shakti RISC-V Team, Indian Institute of Technology, Madras, India. Please see bibliography entry [12] for link from which to download a PDF.

- "An Introduction to Assembly Programming with RISC-V", Edson Borin.

  Bibliography entry [1] contains a link from which to download a PDF.

- "RISC-V Assembly Language", Anthony J. Dos Reis.

  Bibliography entry [4]. Available in bookstores.

A.4 RISC-V GNU tools, including `riscv-gcc` compiler

We will be using the GNU tool chain, specifically the *gcc* compiler and linker, and the *objdump* tool for disassembling an ELF file.

During the course we will show you how to install and use these tools.

The use of these tools is mostly the same as when targeting any target architecture, including well-known architectures like x86 and ARM; the student can find voluminous tutorial materials available on the GNU tool chain on web and in books.

gcc has some specific options for RISC-V; these are documented here:

- https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Options.html
- https://gcc.gnu.org/onlinedocs/gcc/gcc-command-options/machine-dependent-options/risc-v-options.html

It is also useful to know how to use the GNU debugger tool, *gdb*. Again, the student can find voluminous tutorial materials available on on web and in books.

A.5 BSV

In this course, we design the hardware of our RISC-V pipelined CPU using the High Level Hardware Description Language **BSV**. The reasons for our choice (instead of using Verilog, SystemVerilog or VHDL) are discussed in more detail in Appendix B of this document, as well as in the Introduction of the "BSV by Example" book described below.

No advance knowledge of **BSV** is needed for this course; we will teach all necessary **BSV** concepts during the course as we go along.

However, for those who would like to study **BSV** on their own, or wish to view additional **BSV** materials, the following sections provide some resources.

A.5.1 "BSV By Example" book (free downloadable PDF)

This book takes the student through a series of small, targeted **BSV**: examples:

BSV by Example, by Rishiyur S. Nikhil and Kathy R. Czeck, 2010.

Quoting from the Introduction:

> "This book is intended to be a gentle introduction to BSV."
>
> " This book tries to take you into the BSV language one small step at a time. Each section includes a complete, executable (and synthesizable) BSV program, and tries to focus on just one feature of the language"

A bound copy of the book can be purchased on Amazon, but a PDF copy of the book and a tar file containing all the BSV program examples in the book can be downloaded for free from the GitHub BSVLang repository at:

https://github.com/BSVLang/Main/tree/master/Tutorials/BSV_Training

- Book (PDF):
  repository/`Tutorials/BSV_by_Example_Book/bsv_by_example.pdf`
- Machine-readable version of all examples in the book:
  repository/`Tutorials/BSV_by_Example_Book/bsv_by_example_appendix.tar.gz`

A.5.2 BSV Tutorial

A **BSV** self-paced tutorial is available in the GitHub BSVLang repository:

https://github.com/BSVLang/Main/tree/master/Tutorials/BSV_Training

in the directory *repository*/`Tutorials/BSV_Training/` which looks like this:

```
BSV_Training/
    Build/
    Example_Programs/
        Common
        Eg02a_HelloWorld
        ...
        Eg03a_Bubblesort
        ...
        Eg04a_MicroArchs
        ...
        Eg05a_CRegs_Greater_Concurrency
        ...
        Eg06a_Mergesort
        ...
        Eg09a_AXI4_Stream
    Reference
```

Each of the `Eg*` directories cotains a complete example, along with documentation explaining the example, and instructions on how to compile and Verilog-simulate it. The `Reference` directory contains a collection of lecture slide decks explaining the **BSV** language.

A.5.3 MIT Course Material

Massachusetts Institute of Technology (MIT) periodically teaches courses on using **BSV** for digital hardware design. The following link:

http://csg.csail.mit.edu/6.375/6_375_2013_www/handouts.html

contains downloadable material:

- PDFs of slide decks for 12 lectures
- PDFs of slide decks for 4 tutorials classes
- PDFs and codes for 6 laboratories

A.5.4 University of Cambridge Course Material

Prof. Simon Moore of University of Cambridge, UK, uses **BSV** in his teaching and research. Several of his **BSV** examples can be found here:

https://www.cl.cam.ac.uk/~swm11/examples/bluespec/

These examples are somewhat more advanced than the ones in the previous sections.

A.5.5 *bsc* download and installation; *bsc* and BSV manuals

bsc is free and open-source, and can be downloaded and installed as described in **BSV**'s GitHub web site https://github.com/B-Lang-org/bsc.

On the main page of that repository you will find links to the following documents (same links also given here):

- The "**BSV** Language Reference Guide". This document describes the syntax and semantics of **BSV**.

  PDF: https://github.com/B-Lang-org/bsc/releases/latest/download/BSV_lang_ref_guide.pdf

- The "BSC Libraries Reference Guide". This document describes the extensive set of libraries and IP (Intellectual Property blocks) available to the **BSV** user.

  PDF: https://github.com/B-Lang-org/bsc/releases/latest/download/bsc_libraries_ref_guide.pdf

- The "BSC User Guide". This document describes how to use the *bsc* compiler, which compiles our hardware descriptions written in **BSV** into Verilog (which can then be simulated or synthesizes using standard Verilog tools).

  PDF: https://github.com/B-Lang-org/bsc/releases/latest/download/bsc_user_guide.pdf

We will be using the Language Reference Guide and Librares Reference Guide extensively, so you may wish to download a copy for your laptop.

A.6 Verilator (or other Verilog simulator)

We will be doing Verilog simulations extensively during this course. For low cost (free), and uniformity, we will be using Verilator.

During the course, we will show you how to install Verilator and use it.

The Verilator web site, https://www.veripool.org/verilator/, contains instructions on how to install Verilator, and also links to PDF and HTML manuals for Verilator. Version 5.004, or any more recent version, will be suitable.

You can use other Verilog simulators if you prefer, but you should independently know how to use them because we cannot offer support during the course. Some possibilities:

- Icarus Verilog, also known as "iverilog". This is a very good, free and open-source, easy-to-use Verilog simulator, but is quite slow compared to other Verilog simulators and so may be less useful for large designs.

  https://steveicarus.github.io/iverilog/usage/getting_started.html

- Commercial simulators from Synopsys, Cadence or Siemens/Mentor Graphics), Aldec, and others. Each of these needs a paid license.

A.7 Amazon AWS

All hands-on work in this course will be run on the Amazon AWS cloud. This way, everyone in the course has a common, stable, predictable environment and we do not have to waste any time dealing with the countless variations in environments found on different laptops and servers.

During the course, we will explain all necessary concepts as we go along, including how to set them AWS instances and use them.

The Amazon AWS cloud offers, on the "AWS Marketplace" a vast variety of choices for virtual machines or, to use AWS terminology, *instances*. We expect to use the following kinds of instances:

A: An instance running the latest version of Ubuntu (Linux).
B: A so-called "F1 instance", also running Ubuntu. F1 instances have attached FPGAs.
C: An instance running the so-called "AWS FPGA Developer AMI" available in the AWS Marketplace. This runs CentOS (Linux) and comes pre-installed with Xilinx Vivado tools, which we will use for creating FPGA bitfile images during the course.

In Amazon's pricing, (B) is the most expensive, and so we will use that only when we actually run on FPGA. For general development and simulation activities, we'll use (A) which is much cheaper. We will use (C) whenever we're creating a new FPGA bitfile image.

The standard Amazon documentation is can be found here:

- "Set up to use Amazon EC2"
  https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html
- "Tutorial: Get started with Amazon EC2 Linux instances"
  https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html

A.8 Xilinx Vivado

The FPGAs on Amazon AWS F1 instances are Xilinx Ultrascale FPGAs. Thus, when we build bitfiles on Amazon AWS, we will be using Xilinx Vivado tools (which are provided by AWS for zero incremental cost on AWS FPGA Developer AMI instances, see A.7).

During this course, we will explain all necessary concepts as we go along.

When building a bitfile, it is particularly useful to understand how to interpret the Vivado timing and resource reports. The timing report indicates:

- whether or not our design has successfully met our desired frequency target (MHz), and
- if it did not, which part of our circuit is the likely culprit, which needs to be fixed.

The resource report indicates the "size" or our design (how may LUTs, flip-flops, BRAMs, DSPs, *etc.*).

For more details, Xilinx has extensive documentation for which a good starting point is the "Vivado Design Suite Overview" at

https://docs.xilinx.com/r/en-US/ug910-vivado-getting-started/Vivado-Design-Suite-Overview.

A.9 RISC-V textbooks

This course is self-contained, and it is not necessary to acquire any textbooks.

The following list is provide only as a courtesy and convenience. All these books are written using the RISC-V instruction set as examples, and are available in bookstores.

- "The RISC-V Reader: An Open Architecture Atlas", David Patterson and Andrew Waterman, Strawberry Canyon, 2017. Available in bookstores.

  Bibliography entry [3].

- "Computer Organization and Design RISC-V Edition (2nd Edition): The Hardware Software Interface", David A. Patterson and John L. Hennessy Morgan Kaufman, 2020. Available in bookstores.

  Bibliography entry [10].

- "Computer Architecture: A Quantitative Approach, 6th Edition", John L. Hennessy and David A. Patterson, Morgan Kaufmann, 2017. Available in bookstores.

  Bibliography entry [5]. This is the "classic" textbook on computer architecture, a more advanced textbook.

Appendix B

Why BSV?

The BSV language is a modern, high-level, hardware description language with a strong formal semantic basis. It is *fully synthesizable*, *i.e.,* the *bsc* compiler can also compile your source code into Verilog [7], which we regard as the "assembly language" of hardware design. That Verilog code can then be further compiled by ASIC synthesis tools such as Synopsys' Design Compiler or FPGA synthesis tools such as Xilinx Vivado or Altera Quartus, for implementation in ASICs and FPGAs, respectively.

BSV is very suitable for describing architectures precisely and succinctly, and has all the conveniences of modern advanced programming languages such as expressive user-defined types, strong type checking, polymorphism, object orientation and even higher order functions during static elaboration.

All computation in BSV is expressed using "Rules". For many people, this takes a little acclimatization because it is *very* different from traditional programming models (such as in C++ or Java) which are based on sequential processes. But over time, it becomes *the* natural way to think about hardware computation, which is based on massive, fine-grained, heterogeneous parallelism. Complex and high-speed hardware designs are full of very subtle issues of concurrency and ordering, and BSV's computational model is one of the best vehicles with which to study and understand this.

Modern hardware systems-on-a-chip (SoCs) have so much hardware on a single chip that it is useful to conceptualize them, analyze them and design them as *distributed systems* rather than as globally synchronous systems (the traditional view), *i.e.,* where architectural components are loosely coupled and communicate with messages, instead of attempting instantaneous access to global state. This is because the delay in communicating a signal across a chip is now comparable to the clock periods of individual modules. Again, BSV's computational model is well suited to this style of design.

A key to architecting complex systems and reusable modules, whether in software or in hardware, is powerful *interfaces*. Module interfaces in BSV are object-oriented (based on methods), polymorphic/generic, and capture certain computational protocols. This facilitates creating highly reusable modules, enables quick experimentation with alternatives structures, and allows designs to be changed gracefully over time as the requirements and specifications evolve.

Architectural models written in BSV are fully executable. They can be simulated in the Bluesim^{TM} simulator; they can be synthesized to Verilog and simulated on a Verilog simu-

lator; and they can be further synthesized to run on FPGAs or be etched into ASIC silicon, as illustrated in Fig. 1.1. Even when the final target is an ASIC, the ability to run on FPGAs enables early architectural exploration, early development of the software that will later run on the ASIC, and much more extensive and early verification of the correctness of the design. Students are also very excited to see their designs actually running on real FPGA hardware.

In this book, we teach the use of BSV for the design of complex hardware modules and systems by going in detail through a series of examples, and exploring basic concepts as needed along the way, such as combinational circuits, pipelines, data types, modularity and complex concurrency. At every stage the student is encouraged to run the designs at least in simulation, but preferably also on FPGAs.

By the end of the course we will have seen all the source code for a complete simple, pipelined RISC-V CPU along with a small "SoC" (System-on-a-Chip) including an interconnect, a connection to memory, and a few devices such as a UART.

Who uses BSV?

BSV has been used in teaching and research at major universities, including MIT (Massachusetts Institute of Technology, USA), Universisty of Cambridge (UK), Indian Institute of Technology, Madras (India), Indian Institute of Technology, Mumbai (India), Seoul National University (South Korea), University of Texas at Austin (USA), Carnegie Mellon Univerity (USA), Georgia Institute of Technology (USA), Cornell University (USA) and Technical University of Darmstadt (Germany).

BSV has been used to design major IP components in commercial ASICs from Texas Instruments, ST Microelectronics and Google. It has been used for FPGA-based modeling at IBM, Intel, Qualcomm, Microsoft Research, several DARPA projects, and others. It is being used for commercial RISC-V processors from InCore Semiconductors (Shakti line of RISC-V processors, India) and The C-DAC (Center for Development of Advance Computing) (Vega line of RISC-V processors, India).

B.1 Why BSV instead of some other Hardware Design Language?

The rest of this chapter is intended for those interested in comparing BSV's approach to other approaches (Verilog, SystemVerilog, VHDL, and SystemC), and can be safely skipped by others who just want to get on with learning BSV.

One may be curious why the material in this book could not have been covered using one of the more widely known languages for hardware design: Verilog [7], VHDL [6], SystemVerilog [9], SystemC [8]. There are several reasons, outlined below.

In the following paragraphs, we will refer to all the above languages, or at least their synthesizable subsets, as "RTL" (Register Transfer Level languages).

B.1.1 A better computational model

Paradoxically, the formal definitions of the semantics of traditional hardware design languages (HDLs)— Verilog, SystemVerilog, VHDL and SystemC— are not in terms of hardware concepts, but in terms of *software simulation* on conventional computers. Like traditional software programming languages, they are defined in terms of sequential statement execution, with traditional conditionals, loops, and procedure calls and returns, reading and writing conventional variables. Programs can have multiple concurrent *processes* (e.g., "always blocks" in Verilog), but each of them is defined with traditional sequential programming semantics.

Digital hardware, on the other hand, has a quite different computation model. It consists of hundreds, if not thousands of concurrent "state machines" that transform the current state of the hardware, implemented using registers, memories and FIFOs. By and large, there is no sequencing of these state machines based on program counters or statement sequences. Rather, these state machines are independent and "reactive", *i.e.,* each one performs an action whenever certain conditions hold, e.g., when a register holds a particular value, or a value is available in a FIFO, etc.

To bridge this rather large gap from conventional sequential processes to concurrent reactive state machines requires a major mental shift. One must severely restrict code to only a much smaller so-called *synthesizable subset* of a conventional HDL. Processes are restricted to simple clocked loops: "`always @posedge CLK ...`", also known as an "always-block". Even more draconian is a transposition away from the natural concurrent state machine view to a *state element-centric* view: even though a state element may be read and written by multiple state machines, all updates to that state element must be concentrated in a single always-block, usually in a large conditional construct (if-then-else, case, ...) that describes all the different contributions of different state machines. This transposition, from the natural state machine-centric view to the rather unnatural state element-centric view, is necessary because in the the synthesizable subset there is no synchronization between always-blocks; the programmer has to plan every detail of how to resolve (arbitrate) competing updates to each state element.

In other words, in conventional HDLs, neither the simulation view (sequential processes) nor the synthesizable view (state element-centric always blocks) are a natural way to model hardware behavior.

BSV programs, instead, directly express the natural model of hardware— concurrent state machines. Each "rule" in BSV is a reactive state transition that awaits some condition on the hardware state and then takes an action to transforms the state. Further, each rule is an *atomic transaction*, *i.e.,* the details of how one arbitrates competing accesses from multiple rules to common shared state is left to the compiler. This kind of arbitration logic, which is hand-written in other HDLs, is a major source of bugs.

In BSV, unlike in other HDLs, the semantics are identical whether you execute in simulation or in hardware—there is no mental gear shift necessary, and simulation behavior is always identical to synthesized hardware behavior.

Finally, the Rules computation model uniquely encourages *refinement*, a powerful design methodology. We initially create a high-level, approximate model of a target design, using a few large rules. Both the level of micro-architectural detail and the range of functionality

are approximated (abstracted). Often such a model can be written in less than a day, and it can immediately be executed to verify functional correctness. Then, over time, we incrementally add architectural detail—for example, pipeline registers and state machines with more, smaller steps—and the original rules (large step state transitions) are replaced by more, smaller rules (small step state transitions). The atomic semantics of rules makes this a robust methodology, *i.e.*,, a refinement does not have a large ripple effect. This is in quite dramatic contrast to the difficulty in changing RTL, which is notoriously brittle and unforgiving.

Refinement allows early and continous confidence in functional correctness and completeness, since we execute the code very frequently. Refinement allows mid-course corrections in functionality, after observing execution on real data. Refinement allows separating *functionality* from *performance*, achieving functionality early and holding it constant while we improve performance to meet a performance target (by target performance we mean some desired targets for speed, area, and power).

B.1.2 Modern language features

The field of programming languages has seen tremendous progress since the early days (1950s). Modern high-level languages have advanced type systems (polymorphism, typeclasses and overloading, functional types, and so on). Modern high-level languages have strong mechanisms for encapsulation and abstraction (such as object-orientation) which promote the separation of concerns between externally visible behavior and internal representation choices. Modern high-level languages make frequent use of higher-order functions—functions whose arguments and results can themselves be functions and data structures whose components can be functions.

Unfortunately, practically none of these powerful features are present in the synthesizable subsets of conventional HDLs[1]. BSV, on the other hand, adopts the full power of the Haskell functional programming language [11]: algebraic types, functional types, polymorphic types, typeclasses, higher-order functions, and recursive and monadic static elaboration. This delivers unprecedented expressive power, type safety and type flexibility in a hardware design language.

B.1.3 Comparison with C++-based High Level Synthesis

Recently, some tools have become available under the rubric of "High Level Synthesis" (HLS) that claim to shield you from this mental gear shift from simulation to hardware. Designs are written in a traditional sequential programming language (typically C++), and an HLS tool automatically compiles this into a hardware implementation. While beautiful in concept, there are many serious limitations in practice, which are discussed below.

[1] SystemVerilog and SystemC have object-orientation, polymorphism, and overloading, but these are typically used only in simulation for verification environments of hardware designs, not for actual hardware design itself.

C++ codes need significant rewriting

C++ HLS tools will rarely accept arbitrary, off-the-shelf C++ codes and produce good hardware implementations. C++ codes often requre significant restructuring to achieve good results.

First, the tools only accept a limited subset of C++ syntax. In particular, these tools are very averse to any kind of pointer-based argument passing or data structures, unless all the pointers can be resolved by the compiler (*i.e.,*, the compiler statically knows the addresses represented by the pointers). This is because, while C++ normally executes on machines that provide the abstraction of a single large memory with a single address space (so a pointer is fundamentally an address, and dynamic allocation and relocation are easy), hardware designs typically use hundreds or thousands of individual memory units, from registers to register files to SRAMs, DRAMs, Flash memories, and ROMs, each with its own address space.

Second, most C++ codes written for conventional execution rely deeply on sequential execution. For example, they may re-use a variable (multiple reads and writes in different phases of the code). Many of these programming techniques, often a good idea for higher performance and smaller memory footprints in conventional execution, are exactly the opposite of what is needed for hardware implementation, which is highly parallel.

Overall, for good results, one must develop a keen sense of the hardware implementation impact of various "styles" of writing C++ code. Small changes in style can mean the difference between a terrible implementation and an acceptable one. One vendor insists that any team adopting their tool should not consist solely of C++ experts, but must also include hardware engineers.

Narrow range of applicability due to automatic parallelization

C++ is, by official definition, a completely sequential language. Hardware, on the other hand, relies on massive, fine-grain parallelism. It is the HLS tool that has to pull off this magical transformation.

C++ HLS tools rely on a body of knowledge called CDFG Analysis (Control and Data Flow Graph Analysis). After parsing and typechecking, the C++ program is represented internally in a data structure called the CDFG. This CDFG, initially directly reflecting the sequential nature of the source, is analyzed and transformed into a parallel representation from which, eventually, hardware is generated.

It turns out that this transformation only works well for a narrow range of program structures—cleanly nested `for`-loops with fixed iteration bounds, operating on dense rectangular arrays. Of course, many signal-processing and image-processing applications do have this structure, and C++ HLS tools have found their greatest success in this arena.

But the moment we step outside this sweet spot, towards sparse arrays or programs that are highly control-dominated, these tools fall off a cliff. Most hardware design in fact involves components that don't fall into the C++ HLS sweet spot: CPUs, cache systems, switched interconnects, flash memory and disk controllers, high-speed I/O controllers for Ethernet, PCIe, USB, and so on. For example, we are unaware of any project using C++ HLS for CPU design, whereas there are over a dozen such projects using BSV.

Lack of "Algotecture": Architectural transparency and predictability

Most people with some training in Computer Science are familiar with the idea that Algorithms are Job One—when writing performance-critical software, the first-order concern is to design a good algorithm. Further, creating a good algorithm is a creative act; compilers don't automatically create good algorithms for you[2].

Unfortunately, because most of our codes run on classical von Neumann machines, many people forget that, when the execution platform changes, our old algorithms may no longer be any good—the assumptions about the cost of fundamental operations may no longer valid and in fact may be wildly different, requiring a complete re-think of the algorithm.

This bring up a fundamental difference between software design and hardware design. In software, you are given a particular target architecture (CPU, GPU, cluster, vector machine, ...), and the designer's job is to design a good algorithm for that fixed architecture. In hardware, on the other hand, the designer's job is to design the algorithm and the architecture *jointly*. In other words, for hardware designers, algorithm and architecture are joined at the hip; it is meaningless to separate these activities. We thus use the term *Algotecture* to describe this integrated activity.

Unfortunately, most C++ HLS tools provide very narrow visibility and control into architecture. For example, directives for loop unrolling and loop fusion may allow you to express some variation in iterative *vs.* parallel *vs.* pipelined structures. But, basically, it's the tool that chooses the architecture, and you have some weak knobs to guide its choices. A common syndrome with C++ HLS tools is that one quickly produces an implementation, but it is terrible in area or performance, and this is followed by a *long* tail of activity in which the designer tweaks the knobs every which way in an effor to beat it down into the desired performance envelope.

In contrast, with BSV, architectural choices (like algorithmic choices) are in the hands of the designer, where it should be. There are no surprises with respect to architecture; performance is never a mystery, and the designer can quickly improve it and converge to an acceptable solution.

Summary

In summary, it is our experience that BSV is a much better language for complex hardware design, whether control or data oriented, whether for modeling or architectural exploration or final implementation, or for synthesizable on-FPGA verification transactors. Following the philosophy of DSLs (Domain Specific Languages), BSV is very much an expressive DSL beautifully suited for hardware design, whereas sequential C++ is certainly not (it was never intended to be!).

[2]Of course, there is research in this area, but this starts entering the realm of Artificial Intelligence.

Bibliography

[1] E. Borin. *An Introduction to Assembly Programming with RISC-V*. 2021 (Revised: May 9, 2022). PDF online: https://riscv-programming.org/book.html.

[2] P. Dabbelt, M. Clark, and A. Bradbury. RISC-V Assembly Programmer's Manual, Recent update: Jun 29, 2023. Online: https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md.

[3] P. David and W. Andrew. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017. ISBN-10: 0999249118, ISBN-13: 978-0999249116. Available in bookstores.

[4] A. J. Dos Reis. *RISC-V Assembly Language*. 2019. ISBN-10: 1088462006, ISBN-13: 978-1088462003. Available on Amazon.com.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 6th Edition*. Morgan Kaufmann. The Morgan Kaufmann Series in Computer Architecture and Design. ISBN-10: 0128119055, ISBN-13: 978-0128119051. Available in bookstores.

[6] IEEE. IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993, 2002.

[7] IEEE. IEEE Standard Verilog (R) Hardware Description Language, 2005. IEEE Std 1364-2005.

[8] IEEE. IEEE Standard for Standard SystemC Language Reference Manual, January 9 2012. IEEE Std 1666-2011.

[9] IEEE. IEEE Standard for System Verilog—Unified Hardware Design, Specification and Verification Language, 21 February 2013. IEEE Std 1800-2012.

[10] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition (2nd Edition): The Hardware Software Interface*. Morgan Kaufman, 2020. The Morgan Kaufmann Series in Computer Architecture and Design. ISBN-10: 0128203315, ISBN-13: 978-0128203316. Available in bookstores.

[11] S. Peyton Jones (Editor). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 5 2003. haskell.org.

[12] SHAKTI Development Team @ iitm '20 shakti.org.in (Indian Institute of Technology, Madras, India). RISC-V ASSEMBLY LANGUAGE, Programmer Manual Part I, 2020. PDF online: https://shakti.org.in/docs/risc-v-asm-manual.pdf.

[13] A. Waterman and K. Asanović. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, December 13 2019. Document Version 20191213.. PDF online (and newer versions, if any): https://riscv.org/technical/specifications/.

[14] A. Waterman, K. Asanović, and J. Hauser. The RISC-V Instruction Set Manual Volume II: Privileged Architecture, December 4 2021. Document Version 20211203. PDF online (and newer versions, if any): https://riscv.org/technical/specifications/.