University of California
Santa Barbara

Advancing Synthesizable Verilog/SystemVerilog Education with Open-Source Tools and Autograders

A Thesis submitted in partial satisfaction of the requirements for the degree Master of Science
in
Electrical and Computer Engineering

by

Ethan Sifferman

December 2023

The Thesis of Ethan Sifferman is approved.

Professor Dmitri Strukov

Dr. Yogananda Isukapalli, Lecturer

Professor Jonathan Balkind, Committee Chair

September 2023

Advancing Synthesizable Verilog/SystemVerilog Education with Open-Source Tools and Autograders

Curriculum Vitæ

Ethan Sifferman

Education

Sep 23 *(expected)*	M.S. in Computer Engineering, University of California, Santa Barbara
June 22	B.S. in Computer Engineering, University of California, Santa Barbara

Professional Experience

June 22 – Dec 22	Intel Corporation SoC Design Engineer Graduate Intern

Teaching Experience

Sep 22 – June 23	UCSB Electrical and Computer Engineering Department Teaching Assistant
Sep 20 – June 22	UCSB Computer Science Department Learning Assistant

HDL Courses Assisted

Spring 22	UCSB CS 154 — Computer Architecture
Fall 22	UCSB ECE 154A — Introduction to Computer Architecture
Winter 23	UCSB ECE 154B — Advanced Computer Architecture
Spring 23	UCSB ECE 152A — Digital Design Principles

HDL Courses Instructed

Spring 22	UCSB IEEE "FPGA Screensaver Project"

Miscellaneous

May 23	UCSB College of Engineering Outstanding ECE TA Award, $2,600
Mar 23	FOSSi Foundation Latch-Up Presentation, "Using CVA6 in Architecture Education"

Abstract

Advancing Synthesizable Verilog/SystemVerilog Education with Open-Source Tools and Autograders

by

Ethan Sifferman

In the rapidly expanding semiconductor industry, there is an increasing demand for skilled chip developers. Yet, the steep learning curve associated with Hardware Description Languages (HDLs) often acts as a significant barrier for students hoping to pursue a career in digital design. Drawing upon my experience as a HDL educator, which includes teaching Verilog to UCSB's IEEE student chapter and serving as a Teaching Assistant for UCSB's Verilog courses, I have meticulously developed and refined a comprehensive set of methods and resources for Verilog education. My objective encompassed two key facets: equipping students with quality industry-preparation and kindling passion for exploring hardware design. Through a strategic blend of approaches consisting of the integration of accessible open-source tools, the enforcement of popular coding style guides, the implementation of autograders for personalized feedback, and the incorporation of open-source IP blocks into lessons, students can attain proficiency in designing RTL (Register Transfer Level) for rigorously verified hardware systems. These strategies help reduce Verilog's steep learning curve while also expediting the introduction of more advanced topics in digital design and computer architecture. The methods and resources detailed in this thesis will prepare students for the expectations of the semiconductor industry, enhance their coding skills, and promote an accessible and engaging learning environment, ultimately meeting the growing demand for chip developers.

Table of Contents

List of Figures

Chapter 1

Introduction

At the end of Dennard scaling, it may be tempting to think that VLSI and digital design are becoming less-important engineering fields. However, the United States government wouldn't agree, considering they are investing $280 billion over the next ten years into the CHIPS Act [1]. Similarly, TSMC, the world's leading semiconductor manufacturer, is expecting to open three new advanced-node fabs in 2024 [2]. In addition, in 2020, Google started a partnership with GlobalFoundries, SkyWater Technology, and Efabless to provide fully open-source Process Design Kits (PDKs) and tool-chains to lower the barrier of entry for new Silicon engineers [3, 4]. Investments like these have created a never-ending demand for chip developers, and Universities should be working to meet this demand.

In the semiconductor industry, the process of designing an application-specific integrated circuit (ASIC) follows a well-defined sequence, where each step requires extensive training and practice [6, 7, 5]. The following presents an abbreviated flow tailored for students and open-source tool usage, as illustrated in Figure 1.1. This process begins with the design phase, where engineers specify the functionality and requirements of the finished integrated circuit (IC). Then, they use hardware description languages (HDLs) like SystemVerilog to detail the circuit's operations and verify its correctness through simulation. After the HDL implementation passes a series of behavioral simulations, it is

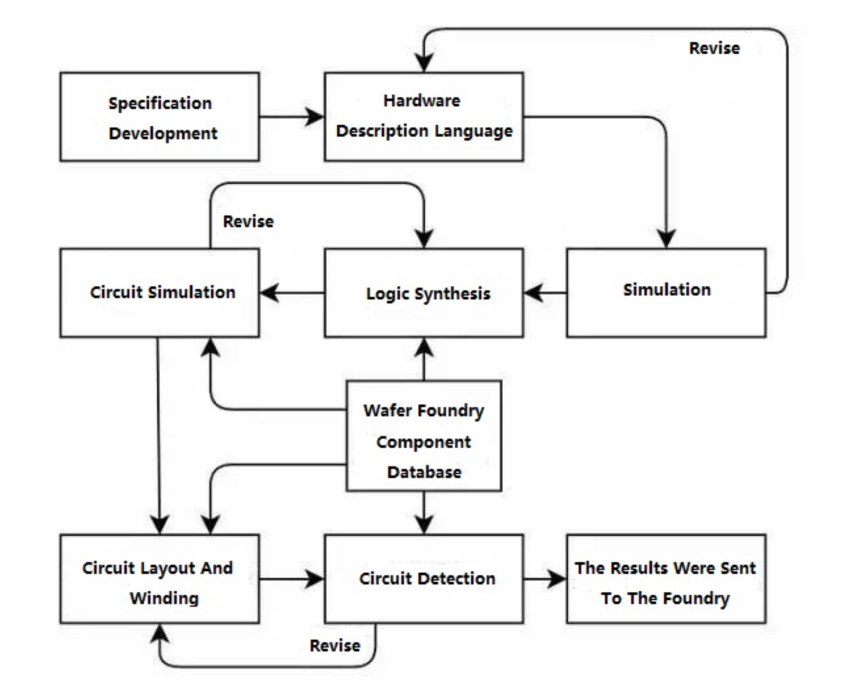

Specification Development
Hardware Description Language
Revise
Revise
Circuit Simulation
Logic Synthesis
Simulation
Wafer Foundry Component Database
Circuit Layout And Winding
Circuit Detection
The Results Were Sent To The Foundry
Revise

Figure 1.1: Abbreviated diagram of ASIC design flow published by Kynix [5].

synthesized for the target's standard cell library and paired with additional digital and mixed-signal IP blocks. Further testing can include running logical equivalence checks (LECs) to ensure the synthesized netlist is correct, rerunning the behavioral simulations on the synthesized netlist to check for reset behavior, or by running intensive simulations on a Field Programmable Gate Array (FPGA). Once these circuit-level tests pass, layout is completed according to timing constraints and design rule checks (DRC). If timing cannot be met, the layout or HDL implementation may need to be adjusted. Finally, after the layout and simulations checks pass, the design is converted into a Graphic Data Stream (GDS) file which is sent to a semiconductor foundry for mass-production.

Because of the multitude of skills are required to create an ASIC design, it is crucial for Universities to offer a strong foundation in writing and working with HDLs to design hardware. Nevertheless, HDLs come with a formidable learning curve, partly due to the difficulties of distinguishing between what code is synthesizable (able to be converted into hardware) and what should be used solely for verification purposes. Additionally, the prevalence of bugs in common HDL tools, the extraordinary inaccessibility of proprietary tools, and the lack of reliable online educational resources can be a major deterrent for students and hobbyists who wish to experiment with digital design on their own. Another factor contributing to the complexity is the interdisciplinary nature of ASIC design. For many students, especially those with a software background, this may be their first experience with hardware design, while hardware students may also be transitioning into a more software-centric environment. Bridging the gap between hardware and software is vital in today's world due to the prevalence of computer-aided design (CAD) software. Universities must recognize the formidable learning curve associated with HDLs and provide comprehensive educational resources, practical hands-on experiences, and interdisciplinary exposure to prepare students for the intricate realm of ASIC design.

To aid educators in combating the intrinsic difficulty in teaching digital design, this thesis presents a collection of methods and resources for Verilog education that will adequately prepare students for writing synthesizable Verilog in industry.

- Chapter 2 delves into the accessibility of open-source tools, highlighting how they provide a cost-effective alternative for students, unlike proprietary tools burdened with high licensing fees. It also discusses the ease of installation and the unique features offered by open-source tools, emphasizing their ability to foster a more exciting and accessible learning environment.

- Chapter 3 shifts the focus to teaching SystemVerilog from a synthesis perspective, and how SystemVerilog can be effectively used for digital design education. It argues that a synthesis-oriented approach to teaching SystemVerilog can bridge the gap between abstract syntax and tangible hardware implementations, providing students with a deeper understanding of digital circuits.

- Chapter 4 explores the best resources for learning synthesizable SystemVerilog. It argues the significance of SystemVerilog style guides in ensuring code quality and adherence to industry standards while warning against relying solely on popular yet misleading Verilog tutorial websites.

- Chapter 5 emphasizes the importance of teaching students how to efficiently work on large-scale Verilog projects, discussing features in SystemVerilog that aid in code organization, and the significance of version control and in-module verification.

- Chapter 6 argues for the incorporation of autograders in Verilog/SystemVerilog education, demonstrating their value in providing instant, high-quality feedback to students. It also discusses how autograders can facilitate remote testing of Verilog designs and foster a sense of community and collaboration among students.

- Chapter 7 culminates the thesis by showcasing a series of assignments I created that provide hands-on experience with advanced computer architecture concepts using the open-source CVA6 RISC-V core.

- Finally, Chapter 8 explores how ideas presented in this thesis could enhance additional advanced Verilog courses such as verification with Universal Verification Methodology (UVM) and SystemVerilog Assertions (SVA) for verification, embedded system and System-on-Chip (SoC) design, and ASIC and VLSI projects.

Chapter 2

Advantages of Open-Source Tools in Education

Although proprietary Verilog tools are predominant in industry and often boast a wide assortment of features, I argue that they can be one of the biggest deterrents for those first learning digital design. Whether students become aghast at the preposterous licensing fees or become hung up on the steep learning curve of the interfaces, the most popular Verilog tools often have trouble exciting students into furthering their Verilog education. Fortunately, there are several open-source Verilog tools that are praised for their accessibility and ease-of-use. A valuable resource for open-source hardware tools, generators, and reusable designs can be found in the curated list provided here: `https://github.com/aolofsson/awesome-opensource-hardware` [8]. For example, a common collection of tools include the simulators Icarus [9] and Verilator [10], the waveform viewer GTKWave [11], the synthesis tool Yosys [12], the FPGA place and route tools Nextpnr [13] and VTR [14], the ASIC flow OpenLane [15], and the hardware package manager and build system FuseSoC [16, 17]. In this chapter, I explain why open-source tools excel over proprietary tools in introducing Verilog and digital design to students.

2.1 Proprietary tool prices deter students.

Many students choose to pursue a degree in computer engineering due to the plethora of creative outlets that it introduces them to. Consider the hands-on process of purchasing affordable circuit components for breadboards. Likewise, platforms like Arduinos and Raspberry Pis are often explored alongside the utilization of programming languages such as C++, Python, and JavaScript. The accessibility and low cost of these mediums often foster self-directed learning. Similarly, introductory Verilog courses can serve as yet another avenue for creative expression, particularly when orchestrated using free and open-source tools. However, a significant obstruction emerges when proprietary alternatives such as Questa, VCS, and Xcelium, coupled with licensing fees over $5,000 [18, 19], become the focal point of a student's introduction to Verilog. Such financial barriers can easily deter enthusiasm for self-guided learning, particularly when students anticipate losing access to the software upon graduation. This is likely the reason why several students in UC Santa Barbara's Verilog courses choose to disobey the requirement of using ModelSim, and instead use Icarus and GTKWave. Students do not want to feel like their time is wasted learning a tool if they lose access to it upon graduation.

2.2 Open-source tools are easy to install.

Even if students do not want to expand their arsenal of tools for expressing their creativity, it is still important to keep Verilog tools accessible by improving students' first-time user experience. For example, Vivado is infamous for its tedious and complex installation process while taking up over 16-60GB when installed. This is contrasted with the overall positive first-time user experience that open-source tools offer. For example, with the "OSS CAD Suite" project [20], you can download all the latest binaries of the most popular open-source Verilog tools in under a minute. This beats Vivado's complicated installation process by a mile and is more-likely to remain on student's computers after the course is over since all the tools only take up half a gigabyte, as opposed to the 16-60GB that Vivado requires.

2.3 Unique benefits and features of open-source tools.

Undoubtedly, proprietary tools offer a multitude of functionalities for advanced users that open-source tools cannot offer. However, open-source tools offer many unique features that are often better for beginners. For example, Icarus runs short simulations much faster than proprietary simulators, making it perfect for receiving instant feedback as students are still learning the language syntax. Similarly, Yosys and Nextpnr perform synthesis and layout significantly faster than tools such as Vivado and Design Compiler, allowing for more rapid prototyping. Also, while ModelSim may happily parse and simulate unsynthesizable code, Verilator will give much more strict warnings, helping to demonstrate the syntax and features that should be allowed in synthesizable designs. (This is further discussed in Section 4.2).

But possibly the most important attribute of open-source tools is that they get updates every day, and offer full transparency when reporting bugs and requesting new features. Depending on the difficulty of the request, the tool maintainers may complete the request within a few weeks. Instructors may even decide they want to do a pull-request themselves; I personally have made several contributions to multiple tools (see Appendix A). This is contrasted with the fact that many universities are not always running the most up-to-date proprietary software. For example, as of 9/10/23, UC Santa Barbara's Engineering Computing Infrastructure's latest version of ModelSim is 10.7d from April 2019, which does not support width-casting from parameters. This negatively affected one of the UCSB Spring 2023 ECE 152A labs. Bugs in tools will undoubtedly happen, but the only solution Siemens offers is to pay for an updated version with the bug fixed. Contrast this with submitting a GitHub issue with Verilator, and having the bug fixed by the next time the course is offered. Open-source tools are often simply the better choice for instructors and students alike.

2.4 Avoid graphical user interfaces.

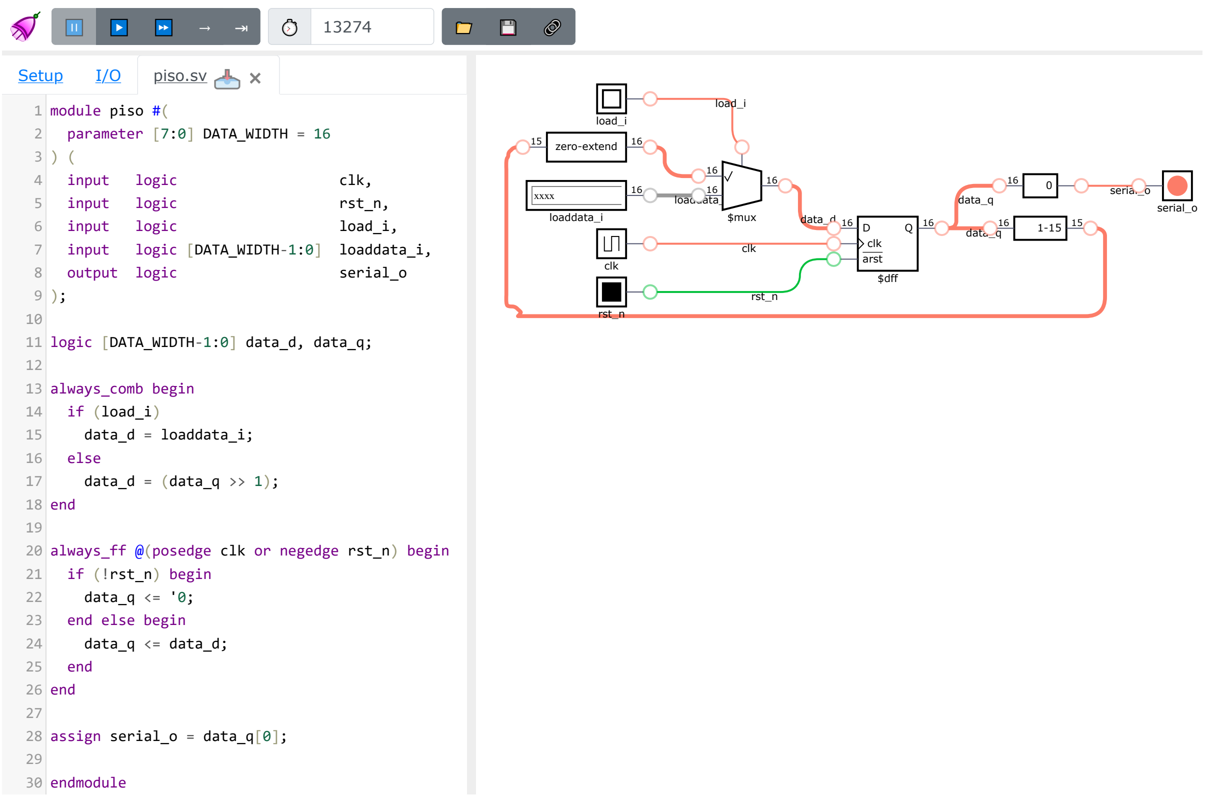
One of the most important ways to make Verilog more accessible is by guiding students to use command line interfaces (CLIs). Nearly all large-scale Verilog designs are paired with user-friendly build scripts and Makefiles for running simulation and synthesis. This is contrasted with complex graphical user interfaces in tools like Vivado, ModelSim, VCS, and Quartus, which can feel daunting for beginners due to the number of options that users have direct access to. UC Santa Cruz Professor Dustin Richmond compares learning Vivado to being stuck in a "point-and-click adventure game" in his talk "So, you want to be an open sourcerer?" [21]. By using CLIs, you can abstract away and decouple tool-specifics from digital design concepts.

To aid in the usually lengthy crafting of TCL scripts and Makefiles, an extremely powerful pair of open-source tools named FuseSoC and Edalize [16, 17] attempt to standardize build files for all Verilog tools with a central `.core` file. The `.core` file, interpretable by FuseSoC, contains the "run" specifications, which are then seamlessly communicated to Edalize for automated generation and execution of build scripts. FuseSoC is compatible with nearly every Verilog tool, so switching simulators is often as easy as changing one line of code in the `.core` file. While TAing for UCSB's ECE 152A, 154A, and 154B, I capitalized on the utility of FuseSoC and Edalize to substantially accelerate assignment setup time. I could provide students with the Makefile automatically generated by Edalize, or I could provide the `.core` file for students to run themselves with FuseSoC. Either option was effective in simplifying the build process so students could better focus on the learning-goals of the assignments.

Chapter 3

Using SystemVerilog for Digital Design Education

Navigating the realm of Verilog/SystemVerilog education presents a distinctive challenge: effectively bridging the gap between abstract Verilog syntax and tangible hardware implementations. Teaching this intuition is of paramount importance, as it can equip aspiring electrical engineers with a deep comprehension of digital circuits, while giving them the hard-skills in computer-aided circuit creation. While it may feel there is a trade-off between teaching Verilog design strategies and teaching circuit design strategies, I argue that teaching inference and synthesis of Verilog will actually augment student understanding of more advanced digital circuit concepts. I would compare this to a programming course using C++ to teach algorithms. For many computer science students, algorithms are synonymous with code, not with logic proofs. For computer engineering students, as long as Verilog is taught with a synthesis-oriented approach, the connection between theoretical circuit concepts and tangible hardware construction becomes seamless, facilitating a more rapid and all-encompassing digital design education.



```
module piso #(
  parameter [7:0] DATA_WIDTH = 16
) (
  input   logic                    clk,
  input   logic                    rst_n,
  input   logic                    load_i,
  input   logic [DATA_WIDTH-1:0]   loaddata_i,
  output  logic                    serial_o
);

logic [DATA_WIDTH-1:0] data_d, data_q;

always_comb begin
  if (load_i)
    data_d = loaddata_i;
  else
    data_d = (data_q >> 1);
end

always_ff @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    data_q <= '0;
  end else begin
    data_q <= data_d;
  end
end

assign serial_o = data_q[0];

endmodule
```

Figure 3.1: Schematic for a Parallel-in Serial-out shift register generated by the netlist graph viewer DigitalJS Online [22].

3.1 Netlist graph viewers teach Verilog inference intuition.

By using netlist graph viewers to provide a visual representation of the synthesis process, students can gain a deeper insight into how their high-level descriptions are transformed into hardware components. DigitalJS Online [22] stands as a notable example of such netlist graph viewers (see Figure 3.1). Through its zero-setup, interactive web interface, students can witness instant translation of their Verilog code into synthesized hardware, which encourages experimentation and rapid prototyping. Additionally, its text editor runs automatic linting with Verilator, which gives incredibly helpful feedback if a syntax-related bad-practice is detected. During a volunteer lecture for UCSB's IEEE student chapter, I taught Verilog concepts from DigitalJS Online's text editor, which seamlessly visualized the logic I was describing in my examples. Also, as a TA for ECE 152A and 154B, I curated several assignments that challenged students to use DigitalJS Online to transform `for` loops and `if` statements into comprehensive, hand-drawn circuit diagrams. Similarly, UC Santa Cruz Professor Dustin Richmond uses netlist graph viewers to teach best practices concerning `case` and `if` statements [21]. Plus, after enough exploration with netlist graph viewers, students can gain the ability to convert Verilog to schematics by hand. By assigning homework and in-lecture exercises that prompt students to deduce hardware constructs from abstract Verilog syntax, their aptitude to understand both Verilog and synthesis will be significantly enhanced.

3.2 Enabling optimizations in netlist graph viewers creates complexity.

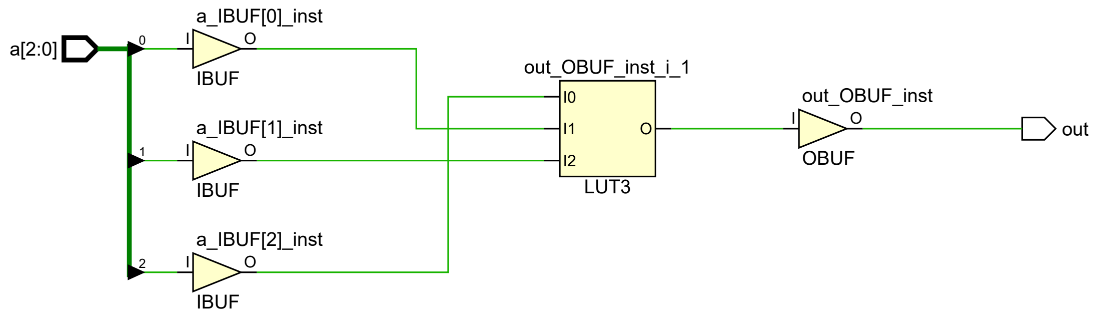
While synthesis tools may run their own specific optimizations [23], learning these intricacies are not critical, given the overall proficiency of available tools and the limited need for target-specific code optimization. Instead, the primary focus should be on teaching students to write clear and transferable code, adhering to best practices covered in the class. (This focus is also described in Section 6.3). While it is acceptable to encourage students to explore various tool and language features as illustrated in Figure 3.2, it is crucial to maintain a balance. Experimentation can stimulate curiosity and self-directed learning, but there may be instances where netlist graph viewers create confusion rather than facilitate understanding. For example, as students start working with larger designs, the chances are increased that a quietly-applied, tool-specific synthesis optimization will result in a netlist that, while valid, would take too much time to decipher and understand. This may turn instructors away entirely from using netlist graph viewers due to the additional complexity that they cause. However, I argue that they are still an essential resource for introducing Verilog, helping students transition from gate schematics to

HDLs. These tools serve as a foundation for students to build their intuition for synthesis, ultimately empowering them to undertake the more advanced design challenges. Even if netlist graph viewers lose their effectiveness as designs get complex, they illustrate to students the vital connection between digital design concepts and Verilog concepts.

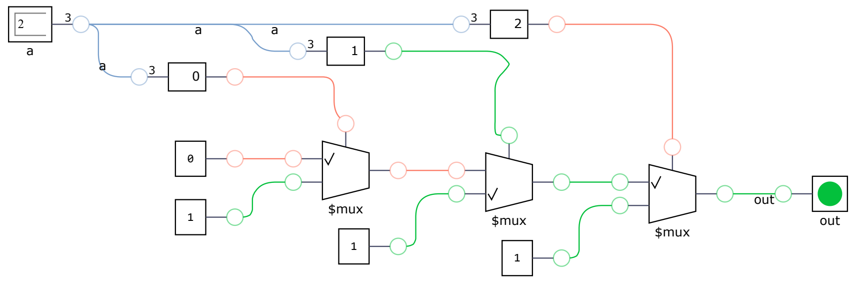
A similar example is providing simplified schematics of transistor implementations of digital gates to relate electrical engineering students to their prior knowledge of analog design. Because transistor implementation specifics are largely unimportant due to the low demand for PDK designers, it is fine to simply introduce basic technologies such as pass-transistor logic instead of analyzing modern multi-finger FinFET CMOS designs. But only after receiving *some* connection to their prior experience with transistors will electrical engineering students feel comfortable working with gates. Similarly, when introducing Verilog, netlist graph viewers can connect prior knowledge of digital elements to code syntax. Much like electrical engineers need some familiarity with transistor-level gate implementations prior to diving into digital design, Verilog students greatly benefit from a basic understanding of the behavior of synthesis tools.

```
wire [2:0] a;
always_comb begin
  out = 0;
  for (integer i = 0; i < 3; i++)
    if (a[i])
      out = 1;
end
```

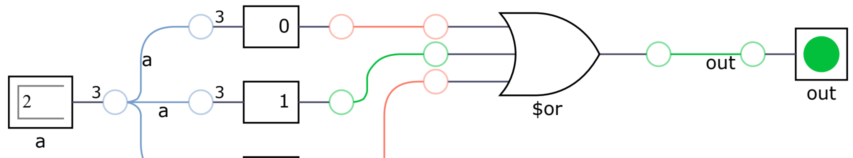
(a) Superfluous implementation of `assign out = |a;`



(b) Vivado correctly infers the code into one parallel LUT.



(c) Yosys without optimizations inefficiently infers the code as a series of 2:1 MUXes.



(d) Yosys with optimizations corrects the inference into one parallel OR gate.

Figure 3.2: Comparison of synthesis optimizations.

3.3 Teaching features that rely on inference is difficult but important.

To promote uniformity among tools, IEEE standardized synthesis of Verilog 1364 features under the label "1364.1". However, there has been no official "1800.1" SystemVerilog synthesis standard to discuss the many new features that were added with SystemVerilog. Many SystemVerilog IEEE 1800 features are not consistently synthesizable by popular synthesis tools, such as classes, interfaces, and dynamic arrays [25, 24]. This may result from a SystemVerilog feature being subjectively similar to a prohibited feature in the IEEE 1364.1 standard, or may be due to insufficient tool development time. For these reasons, support for many features is inconsistent across different open-source tools [26], and Figure 3.3 shows Synopsys' own tools have inconsistencies across each-other. Since there is no official synthesis standard, style guides and linters have filled the role of unofficial documentation of SystemVerilog's synthesizable features to help engineers navigate the inconsistencies across different tools. This is further discussed in Section 4.2 and Chapter 6.

A reaction to the inconsistency and ambiguity in SystemVerilog synthesis may be to teach only obviously-synthesizable constructs such as continuous assignment (`assign`) and standard cell initialization, but that would neglect important language features that have become popular in industry designs. Modern-day RTL engineers regularly use constructs such as procedural blocks, `for` loops, and `if` statements from Verilog; and `struct`, `union`, and `enum` constructs from SystemVerilog. Similarly, in computer programming courses, once students understand the underlying mechanisms, it is common to allow use of standard library functions and data structures. This philosophy should extend to the realm of SystemVerilog. As long as the code adheres to linters and well-verified style guides, and students understand the resulting synthesis, higher-level syntax should be prioritized when it improves code clarity and structure, such as in Figure 3.4.

SystemVerilog Language Construct	Design Compiler	Synplify-Pro
`` `begin_keyword ``, **`` `end_keyword ``** compatibility directives	yes	no
Package import before module port list	yes	no
Parameterized tasks and functions, using parameterized static classes	yes	no
Enumerated type methods (**`.next`**, **`.prev`**, etc.)	yes	no
`` `__FILE__ `` and **`` `__LINE__ ``** debug macros	yes	no
`priority` and **`unique`** modifier to **`if`**...**`else`**	yes	ignored
Cross module references (XMRs)[1]	no	yes
`real` data type	no	yes
Increment or decrement operator on right-hand side of assignment statement	no	yes
Nets declared from **`typedef struct`** definitions	no	yes
Extern module declarations	no	yes
`$onehot`, **`$onehot0`**, **`$countones`**	no	yes
Interface modport expressions	no	yes
Immediate assertions	ignored	yes
`let` macros	ignored	yes
Checkers	no	ignored
`expect` statements	no	ignored

[1] The HDL Compiler (DC) reference manual says that cross-module references are supported "if the hierarchical name remains inside the module that contains the name, and each item on the hierarchical path is part of the module containing the reference." This restriction means that references to interface port contents are legal, but references to the contents of some other module are illegal.

Figure 3.3: Differences in SystemVerilog Support in Synopsys Design Compiler vs. Synopsys Synplify-Pro from "Synthesizing SystemVerilog: Busting the Myth that SystemVerilog is only for Verification" [24].

```
assign out =
    in[31] ? 31 : in[30] ? 30 :
    in[29] ? 29 : in[28] ? 28 :
    in[27] ? 27 : in[26] ? 26 :
    in[25] ? 25 : in[24] ? 24 :
    in[23] ? 23 : in[22] ? 22 :
    in[21] ? 21 : in[20] ? 20 :
    in[19] ? 19 : in[18] ? 18 :
    in[17] ? 17 : in[16] ? 16 :
    in[15] ? 15 : in[14] ? 14 :
    in[13] ? 13 : in[12] ? 12 :
    in[11] ? 11 : in[10] ? 10 :
    in[ 9] ?  9 : in[ 8] ?  8 :
    in[ 7] ?  7 : in[ 6] ?  6 :
    in[ 5] ?  5 : in[ 4] ?  4 :
    in[ 3] ?  3 : in[ 2] ?  2 :
    in[ 1] ?  1 : 0;
```

(a) Using purely structural constructs to create MUXes can provide overly verbose code.

```
function automatic logic [4:0] find_first_set32(logic [31:0] in);
    logic [4:0] out = 0;
    for (integer i = 1; i < 32; i++)
        if (in[i])
            out = i;
    return out;
endfunction

assign out = find_first_set32(in);
```

(b) Using C-like constructs such as `function`, `if`, and `for` can provide much cleaner code.

Figure 3.4: Provided is an example of when C-like constructs can be used to write cleaner code compared to purely structural constructs. Sub-figures 3.4a and 3.4b both implement the Find First Set operation, but 3.4b is better.

3.4 HDLs can be abstractions for complex hardware concepts.

With the full set of synthesizable features being utilized, Verilog can be a useful abstraction layer to better explain complex design concepts such as state machines, pipelining, and handshakes. This parallels the use of abstraction in programming courses, where students often draft pseudocode to conceptualize algorithms before delving into detailed implementation. Transferring this approach to digital design can promote a more rapid and comprehensive learning experience. As long as students demonstrate a strong understanding of how Verilog can be synthesized, they will also have an understanding of the circuits needed to implement the complex design concepts. An example of this in practice was when UCSB ECE 154B students were assigned my "Labs with CVA6" Cache Lab, where students were expected to implement a doubly-linked-list to execute a least-recently-used (LRU) replacement policy. With the helpful abstraction layer of structs, `for` loops, and `if` statements, (as seen in Figure 7.3), students were able to demonstrate a high-level understanding of the LRU algorithm while also understanding the low-level hardware that was generated.

Chapter 4

Best Resources for Learning Synthesizable SystemVerilog

When writing synthesizable SystemVerilog, not all features present in the IEEE 1800 specification can be utilized, as synthesis tools support only a subset of these features. Unfortunately, many educational resources for Verilog and SystemVerilog fail to document which features are synthesizable and which are for verification only. To combat this ambiguity, provided to students should be a curated set of resources dedicated to synthesis as additional materials for outside the classroom.

4.1 Stuart Sutherland's synthesis guide is most valuable.

"Synthesizing SystemVerilog: Busting the Myth that SystemVerilog is only for Verification" by Stuart Sutherland and Don Mills acts as a comprehensive list of synthesizable SystemVerilog features. Despite the absence of an official SystemVerilog synthesis standard, this paper gives valuable insight into synthesizable language features, emphasizing their practical application into modern hardware designs. Sutherland and Mills surveyed the Synopsys tools, Design Compiler and Synplify-Pro, to trace the evolution of Verilog-1984 though SystemVerilog-2009 as a comprehensive hardware design and verification language. To assist those working on "Labs with CVA6", I composed and included a summary of Sutherland's synthesis guide [27]. Since then, I have shared this summary with dozens of students looking to improve their understanding of synthesizable Verilog. Sutherland's guide (or my summary) should be provided to students to ensure a strong introduction to synthesizable Verilog syntax and best practices.

4.2 Style guides and linters record synthesizable features and best practices.

Even while avoiding commonly unsynthesizable SystemVerilog features, design tools are infamous for misinterpreting syntax and often providing little or misleading information on errors. Therefore, using linters and well-verified style guides is crucial in ensuring that an RTL implementation will work on an assortment of tools. As mentioned in Section 3.3, style guides and linters help direct engineers away from ambiguous or poorly-supported language features, and towards syntax and features that are verified to tape-out chips successfully. By introducing Verilog alongside an exhaustive style guide, and providing test flows with linting, students can feel much more confident exploring new language features.

The lowRISC Style Guide discusses many best practices of language features such as the `alias` statement, `automatic` scopes, `package` imports, and floating `begin`-`end` blocks [28]. The Bespoke Silicon Group Style Guide is also strong due to its discussion of structures, enumerations, and memories [29]. There are also style guides published by tool manufacturers that show how what syntax works best using their flows [30, 31, 32]. There are many style guides available; see Appendix B for a(n incomplete) list. Personally, I teach the lowRISC style guide because of its thorough explanations, because of the clarity in `_d` and `_q` as suffixes for register inputs and outputs, and to match the "Labs with CVA6" project (as described in Chapter 7).

Linters such as Verible [33], Verilator [10], and svlint [34] are all popular in design flows. Each are configurable to warn on or forbid specific language features, and many open-source projects choose to lint with more than one of these tools. For example, Verible is known for its auto-formatting capabilities; Verilator is powerful enough to warn on multi-driven signals and accidental latches in `always_comb` blocks; and Svlint is unique for its ability to verify complex whitespace layouts and enforce custom net naming styles. Each has their purpose and should be used on student submissions to ensure best practices are enforced (see Chapter 6).

4.3 Verilog tutorial websites should be treated cautiously.

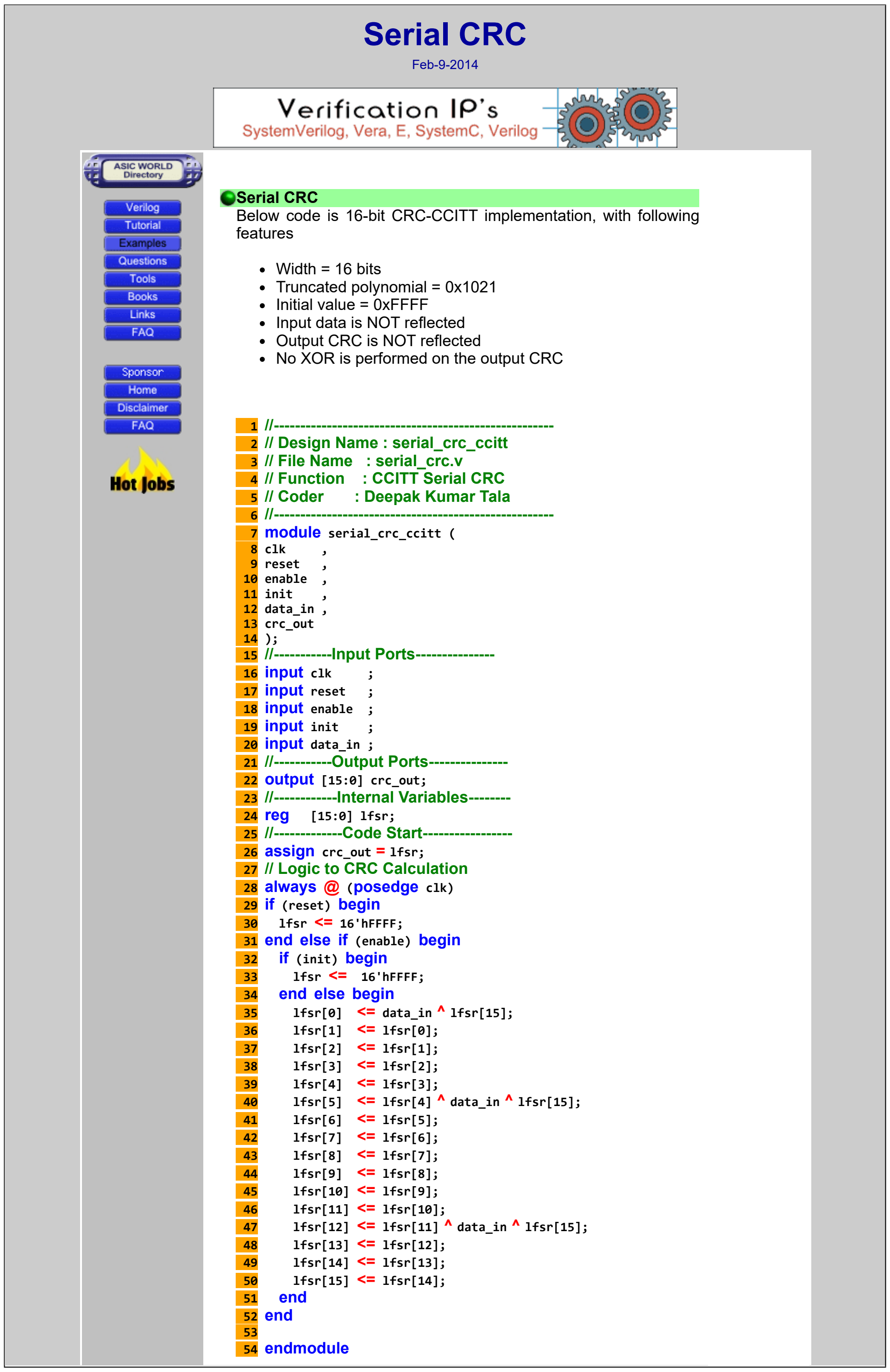
It is important to stress to students the importance of following the provided style guides and linters for Verilog syntax over some of the most popular Verilog tutorial websites such as ASIC World, Chipverify, and Nandland. Despite the user-friendly appearance adopted by these websites, which mirror renowned programming tutorial platforms such as GeekforGeeks, many Verilog tutorial websites often propagate misguided advice for novice hardware developers. While style-guides and linters can act as a reference to well-verified practices for beginners and professionals alike, tutorial websites do not always teach current-day, synthesizable design syntax that is compatible with a multitude of tools. Only if students maintain adherence to the instructor-specified style-guides and the course's subset of synthesizable features, then tutorial websites can be used as resources.

For example, while a TA for ECE 152A, 154A, and 154B, the most prevalent misinformation I saw them encourage in students was to put combinational logic inside of `always_ff` blocks (see Figure 4.2). The lowRISC Style Guide, the BSG SystemVerilog Coding Standards, and the IEEE 1364.1-2005 Verilog Synthesis Standard all recommend only putting reset, set, and enable logic in `always_ff` blocks [28, 29, 23]. Unnecessarily large `always_ff` blocks are prone to bugs because `always_ff` blocks do not offer warn-

```
// not cumulative
always_ff @(posedge clk) begin
    data1_q <= data1_q + 1;
    data1_q <= data1_q + 1;
    data1_q <= data1_q + 1;
    data1_q <= data1_q + 1;
end

// doesn't warn that there is no default case
always_ff @(posedge clk) begin
    if (rst)
        data2_q <= '0;
end

// unclear whether this is a DFF or DFFE
always_ff @(posedge clk) begin
    if ( pkg::func(data3_i) )
        data3_q <= data3_d;
end
```

Figure 4.1: Potentially confusing behaviors of `always_ff` blocks.

ings on unhandled code paths, blocking and non-blocking assignment mismatches can lead to undefined behavior, and synthesis tools may incorrectly infer the incorrect type of flip-flop (see Figure 4.1). In my experience teaching SystemVerilog, whenever a student asked for help solving a bug, but followed this design practice, I immediately asked them to separate the block into an `always_comb` and `always_ff`. Over half the time, that simple refactor incidentally fixed the student's bug.



Serial CRC

Feb-9-2014

Serial CRC

Below code is 16-bit CRC-CCITT implementation, with following features

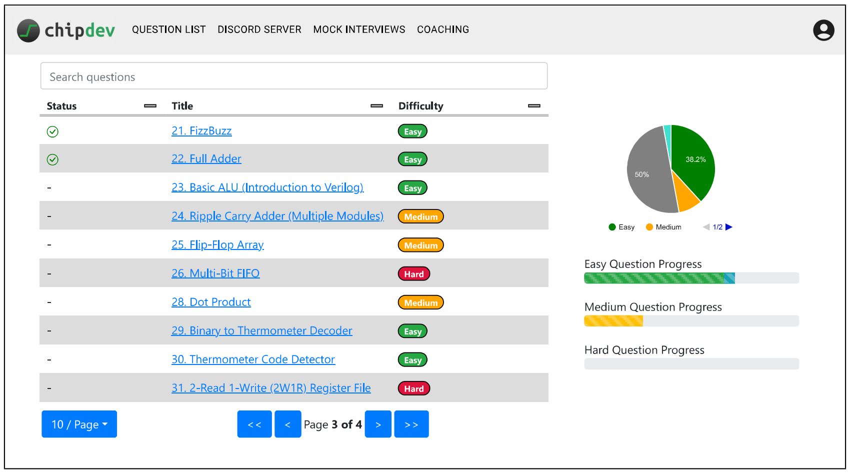
- Width = 16 bits
- Truncated polynomial = 0x1021
- Initial value = 0xFFFF
- Input data is NOT reflected
- Output CRC is NOT reflected
- No XOR is performed on the output CRC

```
//-----------------------------------------------------
// Design Name : serial_crc_ccitt
// File Name   : serial_crc.v
// Function    : CCITT Serial CRC
// Coder       : Deepak Kumar Tala
//-----------------------------------------------------
module serial_crc_ccitt (
clk     ,
reset   ,
enable  ,
init    ,
data_in ,
crc_out
);
//-----------Input Ports---------------
input clk     ;
input reset   ;
input enable  ;
input init    ;
input data_in ;
//-----------Output Ports---------------
output [15:0] crc_out;
//------------Internal Variables--------
reg   [15:0] lfsr;
//-------------Code Start-----------------
assign crc_out = lfsr;
// Logic to CRC Calculation
always @ (posedge clk)
if (reset) begin
  lfsr <= 16'hFFFF;
end else if (enable) begin
  if (init) begin
    lfsr <=  16'hFFFF;
  end else begin
    lfsr[0]  <= data_in ^ lfsr[15];
    lfsr[1]  <= lfsr[0];
    lfsr[2]  <= lfsr[1];
    lfsr[3]  <= lfsr[2];
    lfsr[4]  <= lfsr[3];
    lfsr[5]  <= lfsr[4] ^ data_in ^ lfsr[15];
    lfsr[6]  <= lfsr[5];
    lfsr[7]  <= lfsr[6];
    lfsr[8]  <= lfsr[7];
    lfsr[9]  <= lfsr[8];
    lfsr[10] <= lfsr[9];
    lfsr[11] <= lfsr[10];
    lfsr[12] <= lfsr[11] ^ data_in ^ lfsr[15];
    lfsr[13] <= lfsr[12];
    lfsr[14] <= lfsr[13];
    lfsr[15] <= lfsr[14];
  end
end

endmodule
```

Figure 4.2: This is an example provided by ASIC World that encourages putting combinational logic inside `always_ff` blocks. [35] I explain why this is a bad design practice in Section 4.3.

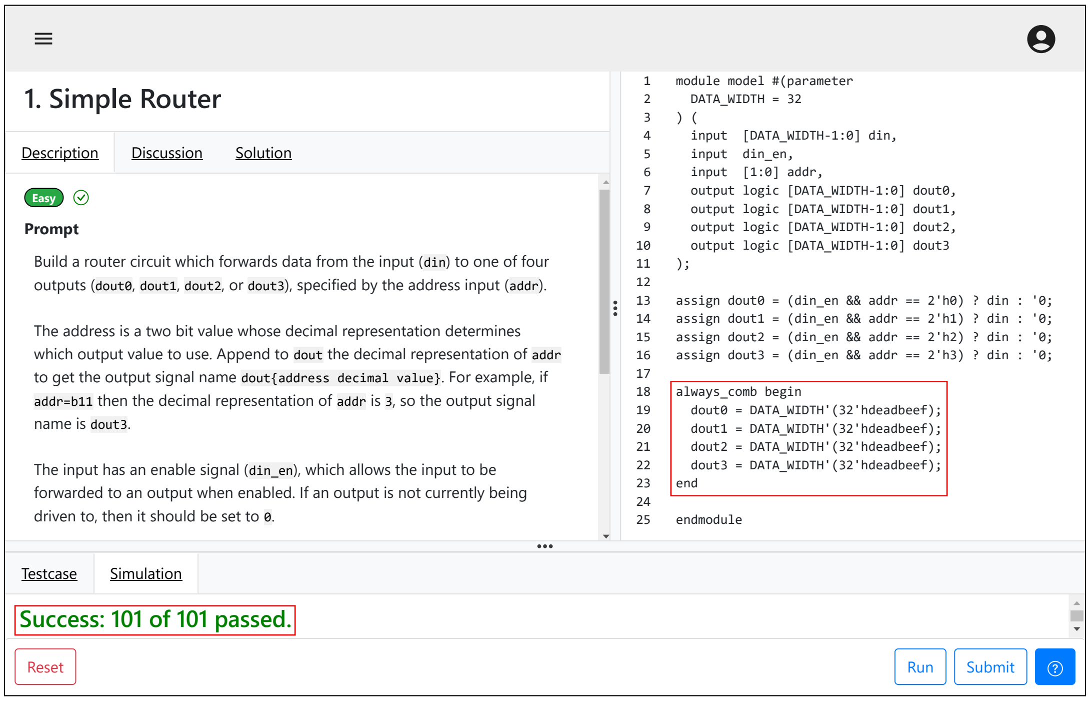
4.4 ChipDev.io can be used to practice Verilog (if used effectively).

The final resource I like to share with students is ChipDev.io, which offers an online collection of popular Verilog questions, paired with an online IDE and testbench. The 30+ questions range from implementing a shift register to designing an ALU; (see Figure 4.3). If students are looking for lots of practice questions as job interview preparation or for general practice, I always recommend ChipDev. However, ChipDev does not run gate-level simulation or logical equivalence checks, so bad submissions may be incorrectly rewarded (see Figure 4.4). Plus, after speaking with the ChipDev team, they notified me that synthesis was not on their priority list. Therefore, I strongly urge students to verify their answers with DigitalJS Online or other synthesis tools before feeling they have a mastery over any question.



Status	Title	Difficulty
✓	21. FizzBuzz	Easy
✓	22. Full Adder	Easy
-	23. Basic ALU (Introduction to Verilog)	Easy
-	24. Ripple Carry Adder (Multiple Modules)	Medium
-	25. Flip-Flop Array	Medium
-	26. Multi-Bit FIFO	Hard
-	28. Dot Product	Medium
-	29. Binary to Thermometer Decoder	Easy
-	30. Thermometer Code Detector	Easy
-	31. 2-Read 1-Write (2W1R) Register File	Hard

Figure 4.3: An example of questions that ChipDev offers [36].



1. Simple Router

Prompt

Build a router circuit which forwards data from the input (`din`) to one of four outputs (`dout0`, `dout1`, `dout2`, or `dout3`), specified by the address input (`addr`).

The address is a two bit value whose decimal representation determines which output value to use. Append to `dout` the decimal representation of `addr` to get the output signal name `dout{address decimal value}`. For example, if `addr=b11` then the decimal representation of `addr` is `3`, so the output signal name is `dout3`.

The input has an enable signal (`din_en`), which allows the input to be forwarded to an output when enabled. If an output is not currently being driven to, then it should be set to `0`.

```
module model #(parameter
  DATA_WIDTH = 32
) (
  input  [DATA_WIDTH-1:0] din,
  input  din_en,
  input  [1:0] addr,
  output logic [DATA_WIDTH-1:0] dout0,
  output logic [DATA_WIDTH-1:0] dout1,
  output logic [DATA_WIDTH-1:0] dout2,
  output logic [DATA_WIDTH-1:0] dout3
);

assign dout0 = (din_en && addr == 2'h0) ? din : '0;
assign dout1 = (din_en && addr == 2'h1) ? din : '0;
assign dout2 = (din_en && addr == 2'h2) ? din : '0;
assign dout3 = (din_en && addr == 2'h3) ? din : '0;

always_comb begin
  dout0 = DATA_WIDTH'(32'hdeadbeef);
  dout1 = DATA_WIDTH'(32'hdeadbeef);
  dout2 = DATA_WIDTH'(32'hdeadbeef);
  dout3 = DATA_WIDTH'(32'hdeadbeef);
end

endmodule
```

Success: 101 of 101 passed.

Figure 4.4: This example shows ChipDev [36] incorrectly accepting this submission despite a potential mismatch between simulation and synthesis. For example, Verilator will override the `always_comb` with the `assign`, but Yosys will override the `assign` with the `always_comb`. This could be corrected if ChipDev chooses in the future to incorporate a similar verification flow to what is outlined in Section 6.2.

Chapter 5

Teaching Code Scalability and Development Practices

Aside from ensuring that student code follows best practices and correctly synthesizes, an equally important skill to teach students is how to efficiently work on large-scale projects. As the number of transistors on an integrated circuit has increased, the scale of Verilog designs has also drastically increased. At Intel, the SoC that my team was verifying had over 500 Verilog source files in the design. Similarly, one of the most popular RISC-V cores, CVA6, is written in nearly 17,000 lines of code [37]. Ensuring seamless development and limiting the number of bugs within these colossal codebases requires strong project management skills. This is achieved by automatic regression, consistent coding styles, and employing version control. For advanced Verilog courses such as Computer Architecture or SoC design, I argue that teaching code scalability is often just as important as teaching microarchitecture implementation methods.

5.1 SystemVerilog offers many features to aid in code organization.

Features such as packages, structs, and parameters are incredibly popular in large-scale SystemVerilog projects. And although IEEE 1364 Verilog does not support packages and structs, many RTL designers have found workarounds with `` `include `` files and func-

```
/*
 * Copyright (c) 2023, University of California; Santa Barbara
 * Distribution prohibited. All rights reserved.
 *
 * File: taillights_pkg.sv
 * Description: Taillights SystemVerilog package.
 *  Includes the enum for the FSM module
 */

package taillights_pkg;

typedef enum logic [2:0] {
    S000_000,
    S000_100,
    S000_110,
    S000_111,
    S001_000,
    S011_000,
    S111_000,
    S111_111
} state_t;

endpackage
```

Figure 5.1: This is a SystemVerilog package that was provided to ECE 152A students to aid in their implementation of a 1965 Ford Thunderbird taillight state-machine.

tions. [28, 38] These features may not be *required* to implement hardware algorithms such as instruction decoding and serial interfaces, but are still extremely prevalent in well-organized, large-scale Verilog designs. In ECE 152A, ECE 154A, and ECE 154B, we were sure to teach students to apply these code organization strategies (see Figure 5.1).

5.2 Version control should be used in Verilog designs.

Aside from code structure, a cornerstone of modern software development is version control. Intel, numerous other companies, and most RISC-V projects extensively rely on Git and GitHub for version management (see Appendix A and Appendix B). Moreover, Git submodules and subtrees provide an elegant and popular solution for integrating IP blocks into designs seamlessly, enhancing reusability and collaboration. Plus, allowing students to post code they have written themselves to GitHub is a great way to aid them in creating an online portfolio for themselves. In ECE 154B, students practiced using Git and GitHub to explore open-source projects, collaborate with peers, add open-source cores as submodules, and more. While software might not be the core focus for some students, being able to work with it efficiently and professionally is still extremely valuable. Because of the invaluable aid Git offers in code quality, and its extreme prevalence in all software development, it is an essential hard-skill for all engineers.

5.3 SystemVerilog assertions and in-module verification are important.

The final design strategy for promoting code scalability is to promote in-module verification. Waveform viewers are incredibly powerful and useful tools, but work best when supplemented with `$display` statements that have already identified where and when a simulation error occurred. Most SystemVerilog in industry designs is full of self-verifying modules by use of SystemVerilog assertions (SVA) and Universal Verification Methodology (UVM). Note that as of 9/10/23, since there is poor SVA and UVM support in open-source tools, projects may need to use `` `ifdef `` macros to disable UVM and SVA calls on a per-tool basis, may need to limit themselves to the subset of supported features, or may need to resort to a basic `always` blocks instead. Verilator has limited support for UVM and SVA, but is getting closer to full support every day [39, 40, 41]. But no matter the specific implementation, in-module verification is a valuable design practice to teach students. In ECE 152A, ECE 154A, and ECE 154B, students were often required to design modules that incorporated simulation-only logic to test basic functionality. By adopting these universal standards, Verilog education becomes better aligned with real-world methodologies for enhanced scalability and proficiency.

Chapter 6

Autograders

The incorporation of autograders within Verilog/SystemVerilog education is arguably the most valuable aspect of RTL education. These tools, particularly exemplified by platforms like Gradescope, can introduce a dynamic and interactive dimension to the learning process, revolutionizing the way students engage with Verilog concepts. Leveraging custom docker containers and custom Bash scripts, Gradescope's autograders easily facilitate Verilog testbench simulations, strict linting, synthesis, gate-level simulation, and more, yielding insights and feedback on various aspects of student submissions. However, managing software licenses on autograder servers can be a hassle, so all these functionalities are often best deployed with open-source tools. In the context of UCSB's ECE 152A, 154A, and 154B courses, students responded extremely positively to autograders, visibly enhancing their mastery over synthesizable Verilog.

6.1 Autograders offer instant, high-quality feedback.

Students are empowered to submit their code multiple times, enabling them to refine their solutions and learn from their mistakes in real time. This back-and-forth approach ensures that students can practice a Verilog concept and receive as much help as they need until they pass all the instructor-defined tests. In the autograders that I set up, it is worth noting that a significant majority of students eventually achieve a 100% score by the assignment deadline. Therefore, autograders fall under the educational approach known as “Ungrading,” where the emphasis shifts strongly toward providing valuable feedback over assigning traditional grades. This phenomenon essentially transforms the grading system into a confidence-building mechanism rather than a competitive ranking system. Ungrading has been shown to help students by reducing stress, inspiring creativity, and encouraging healthy risk taking. [42, 43] However, arguably Ungrading’s largest downside is that the instructor may not have time to provide personalized feedback to all students. Fortunately, an intrinsic attribute of software, (such as HDL implementations), is that code quality and correctness can be run with automatic, subjective computer algorithms. Therefore, by implementing autograders, Verilog educators can easily tap into this pedagogical insight, offering students a more effective way to grasp digital design principles.

6.2 Autograders can run remotely without complex local-setup.

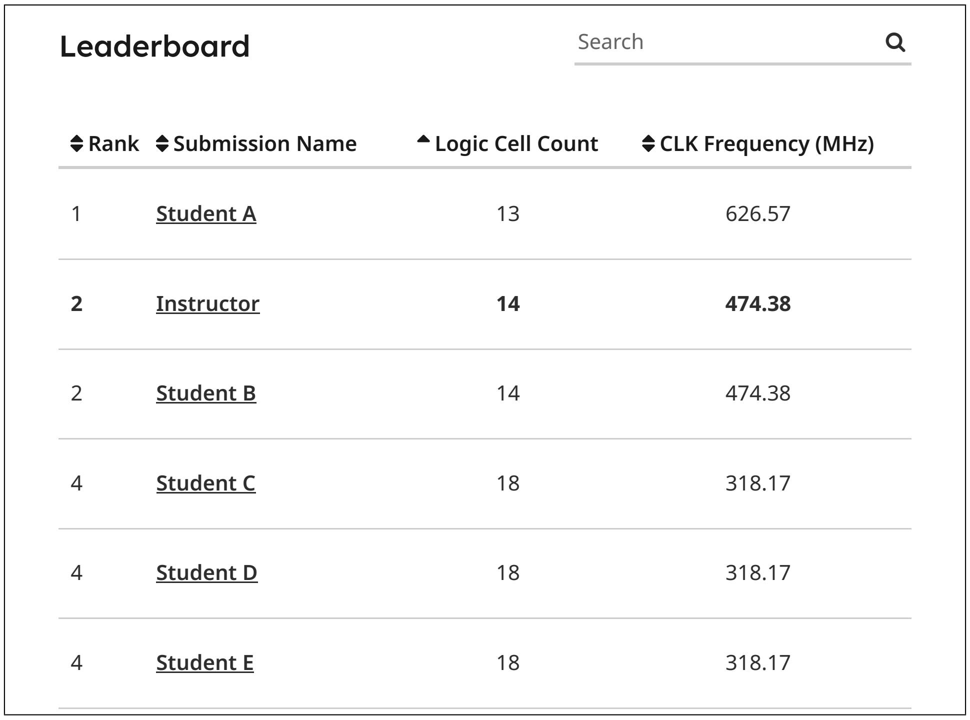
When instructing students on crafting Verilog code that maintains accurate synthesizability across various platforms, it is essential to follow the industry standard of verifying a design with a wide selection of tools. Autograders streamline this process, making it accessible and efficient for students to perform comprehensive testing without the need for local installation. For example, the autograders that I created for ECE 152A, 154A, and 154B would consistently use anywhere from 6 to 10 different tools, sometimes requiring complex installation and setup procedures. Expecting students to complete these setup procedures is often tedious and counterproductive. Therefore, simply giving students access to a fully prepared autograder can remove the setup barrier completely.

As mentioned, an autograder test suite that closely mirrors industry quality should follow all the verification steps demonstrated in Figure 1.1. First, it is important to run behavioral simulations with multiple tools such as Icarus, which supports propagation of unknown (`x`) values, and simulation with Verilator, which has stronger restrictions on bad syntax. Only by passing simulations with both tools should the autograder grant full points. Furthermore, code linters such as Verilator and Verible can ensure adherence to essential coding standards and practices, checking for issues like latches in `always_comb` blocks, correct use of blocking and non-blocking assignments, net-width discrepancies, and more. Considering that the frontends of tools do not always offer helpful warnings, this detailed syntax checking from Verilator and Verible is invaluable for students when fixing otherwise cryptic issues. Then, to deploy SystemVerilog synthesis with open-source tools, Yosys and Nextpnr must be paired with a frontend such as Surelog or zachjs/sv2v. The Yosys synthesis and Nextpnr layout process can verify if students are using too many logic cells, if their design is too slow, or if their design infers prohibited logic cells. As a final post-synthesis step, Icarus can be run one final time on the Yosys output to initiate a gate-level simulation (GLS) with unknown value propagation, and Yosys EQY can be run to perform logical equivalence checking (LEC). All of these features[1] have been successfully implemented in autograders for ECE 152A, 154A, and 154B.

6.3 "For-fun" leaderboards can excite and inspire students.

Aside from a grade assigned by the autograder, another form of feedback can be provided through a class leaderboard. Students can see how their submission compares to the rest of the class on statistics such as logic cell usage, max clock frequency, or branch predictor hit percentage (see Figure 6.1). Since the leaderboards in ECE 152A and 154B did not count towards any points, students really enjoyed seeing how each of

[1]Inclusion of LEC with EQY into an autograder remains outstanding due to time constraints during the project's development.


Leaderboard

Rank	Submission Name	Logic Cell Count	CLK Frequency (MHz)
1	Student A	13	626.57
2	**Instructor**	**14**	**474.38**
2	Student B	14	474.38
4	Student C	18	318.17
4	Student D	18	318.17
4	Student E	18	318.17

Figure 6.1: Example of a Gradescope leaderboard for a counter lab (same lab as in Figure 6.2). Student designs were ranked on cell usage and maximum frequency as calculated by Yosys and Nextpnr-ice40. (Student names have been obfuscated for privacy reasons).

their designs compared to their peers' designs, then would try to beat their friends for bragging rights. Because of this, I added leaderboards to every assignment that I could. However, it is important to clarify to students that code readability should be prioritized over moving up in the leaderboard by saving 1-2 logic cells. However, because autograders running purely open-source tools must only rely on Yosys and ABC for synthesis, students may be incorrectly rewarded for submissions that are not well-optimized for other more prevalent synthesis tools such as Design Compiler or Vivado (similar to Section 3.2). This was a rare edge case that only visibly affected 1 submission (Figure 6.2) across the 600+ leaderboard submissions I saw, but it is still important to monitor in students. Overall, creating assignment leaderboards was a great way to increase student enthusiasm without bringing additional stress or responsibilities.

```
// Assignment: Create a counter with a direction flag

// student submission (13 logic cells)
always_comb begin
    for (int i = 0; i<WIDTH; i++) begin
        // logic here is to toggle if counting up and lowerbits = max
        // or counting down and lowerbits = 0
        lower_mask = (1<<i) - 1;
        count_d[i] =
        count_q[i] ^ (~dir_i & (count_q & lower_mask) == lower_mask |
                      dir_i & (count_q & lower_mask) == 0);
    end
end

// teacher solution (14 logic cells)
always_comb begin
    count_d = count_q + ((dir_i) ? -1 : 1);
end
```

Figure 6.2: A ECE 152A student drastically reduced their code's readability and transferability in order to save 1 logic cell over the teacher solution (same assignment as in Figure 6.1).

6.4 Autograders can foster community and collaboration.

In ECE 152A, students discussed their challenges, insights, and strategies with peers, creating a collective learning environment. Since each student had a clear goal of "passing all the tests", we saw students combine strategies and knowledge with confidence. Plus, ungraded leaderboards offered a fun but optional way for students to collaborate in friendly competitions. Communal engagement not only strengthens individual understanding, but also enriches the overall learning ecosystem. By integrating autograders, students experience a much more positive learning environment while still being provided critical skills and insights for their future engineering endeavors.

Chapter 7

Labs with CVA6 Project

Standing as a culmination of all the Verilog education philosophies described in this thesis is a set of assignments I designed under the project label “Labs with CVA6”. By drawing on my experiences from my time at Intel, my contributions to open-source projects, and interactions with industry contacts, I wrote four labs that aimed to give students practical experience with advanced computer architecture concepts. Each lab is centered around the fully featured RISC-V core CVA6 [37], which is perfect for advanced architecture education due to its 6-stage pipeline, dynamic branch predictor, L1 cache, scoreboard unit, and virtual memory support (see Figure 7.1). The labs presented students with a unique chance to engage with a fully-featured and transparent RISC-V core. This hands-on interaction facilitated a deeper comprehension of important architectural principles while also enhancing their ability to work with large and complex SystemVerilog designs. These labs are available for free under the BSD-3-Clause license [27].

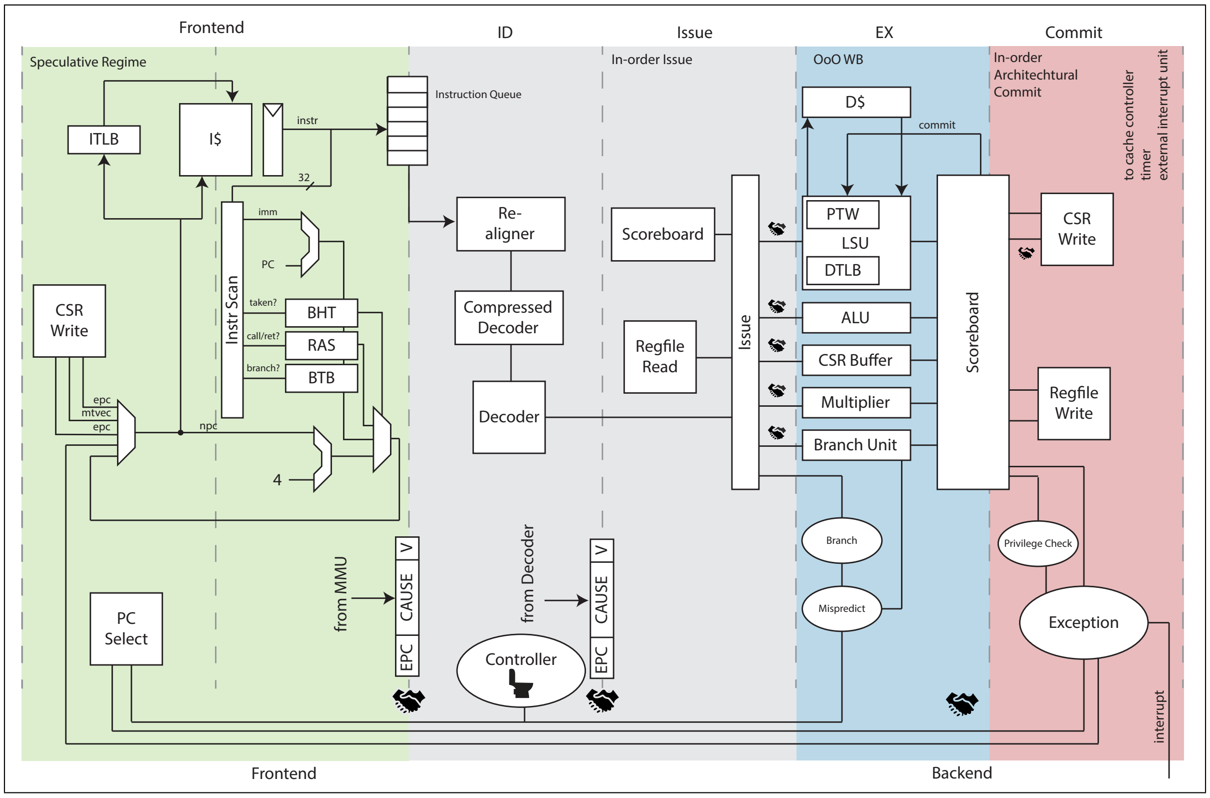


Figure 7.1: CVA6 Block Diagram provided by the core documentation [37]. Students spend the Branch Prediction and Out-of-Order labs studying each of the 6 stages: "PC Generation", "Instruction Fetch", "Instruction Decode", "Issue", "Execute", "Commit."

7.1 These labs provide hands-on exploration of architectural concepts.

"Labs with CVA6" consists of four labs covering branch prediction, caching, out-of-order, and virtual memory. The Branch Prediction Lab guides students in modifying the CVA6 testbench to display hit rate results and in adapting the CVA6 branch prediction unit RTL into a global predictor (see Figure 7.2). Similarly, the Caching Lab asks students to write RTL for a victim-cache module, then requests the creation of assembly scripts that demonstrate CVA6's cache hierarchies and memory management (see Figure 7.3). Then moving away from custom RTL, in the Out-of-Order Lab, students further practice writing assembly to test their comprehension of out-of-order execution

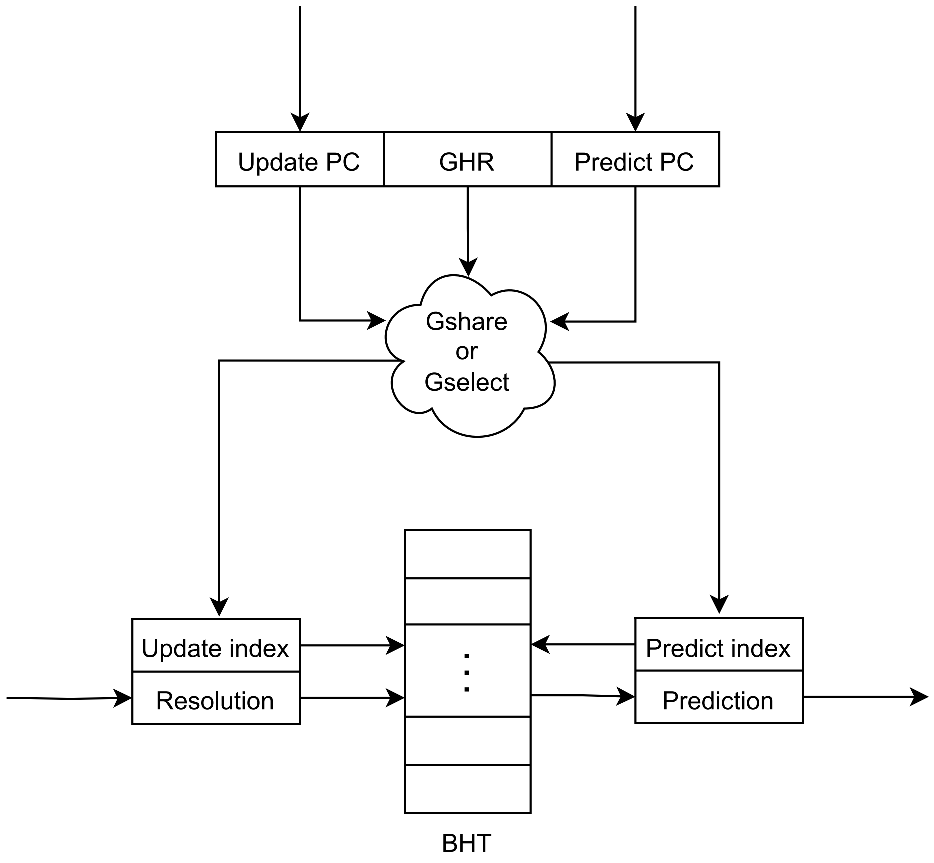


Figure 7.2: Block diagram of the Global Two-Level Branch Predictor Design featured in “Labs with CVA6”. Participants are tasked with transforming CVA6’s branch predictor into a more-sophisticated global predictor, enhancing the processor’s performance by improving branch hit-rate for specifically designed benchmarks [27].

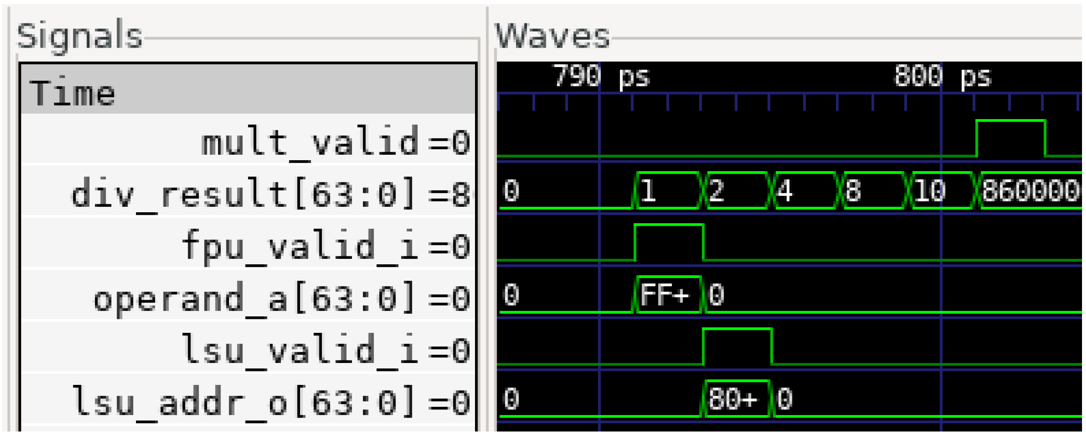
(see Figure 7.4). Finally, the Virtual Memory Lab enables students to configure privilege modes and add page-table entries by modifying a provided bootloader and OS (see Figure 7.5). Based on ECE 154B student responses and post-lab discussions, it was evident that the practical insights offered by these labs substantially deepen understanding of the concepts. The openness of CVA6’s source code was pivotal to the labs’ success, as it granted students the opportunity to interact with every implementation detail and feature of the core. By studying open-source hardware designs, students can gain valuable and distinctive insights that traditional textbooks simply cannot provide.

```
//                    DLL Structure                    //
// MRU - ... - way.mru - way - way.lru - ... -  LRU //

typedef logic [$clog2(NR_ENTRIES)-1:0] way_index_t;

struct packed {
    logic [TAG_SIZE-1:0] tag;
    way_index_t lru; // less recently used
    way_index_t mru; // more recently used
    logic valid;
} dll_d[NR_ENTRIES], dll_q[NR_ENTRIES];

// lru register
way_index_t lru_d, lru_q, mru_d, mru_q;

// index to bump
way_index_t read_index, write_index;


// separate the data from the dll help with optimization
logic [LINE_WIDTH-1:0] data_d[NR_ENTRIES], data_q[NR_ENTRIES];
```

Figure 7.3: Snippet of "Labs with CVA6"'s Cache Lab starter code [27]. This lab focuses on implementing a victim cache with LRU replacement policy to improve hit rate when CVA6 is configured with a direct-mapped cache. Participants are expected to use modern SystemVerilog `typedef` and `struct` constructs.

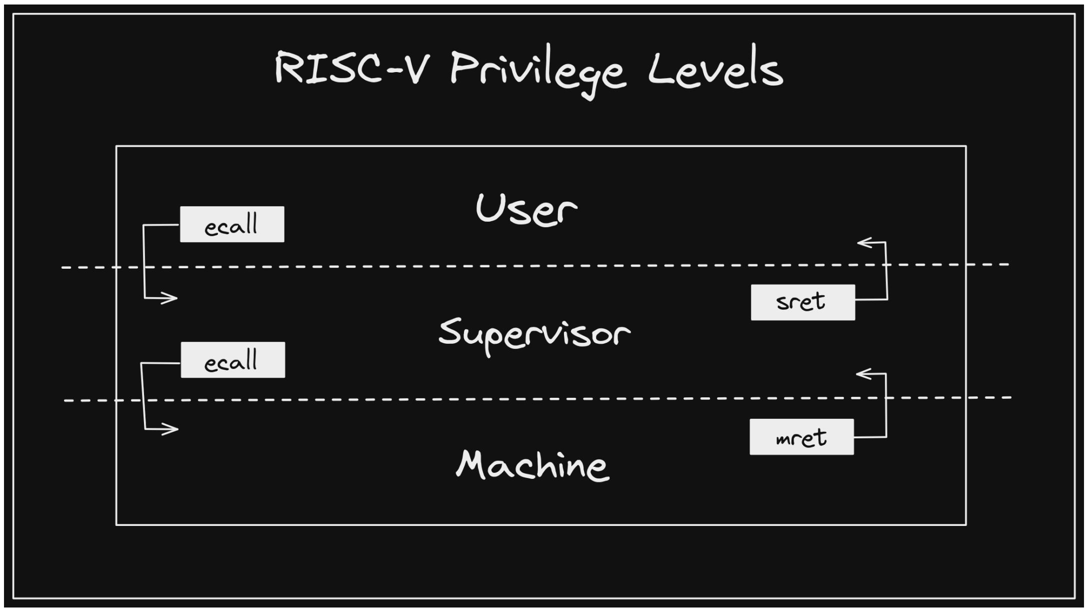
```
# No data hazards
div      t2, t0, t1;      # Takes 25 cycles
fsqrt.s f1, f1;           # Takes 1 cycle
lw      t5, 0(s1);        # Takes 1 cycle
```

(a) RISC-V assembly that demonstrates out-of-order execution.

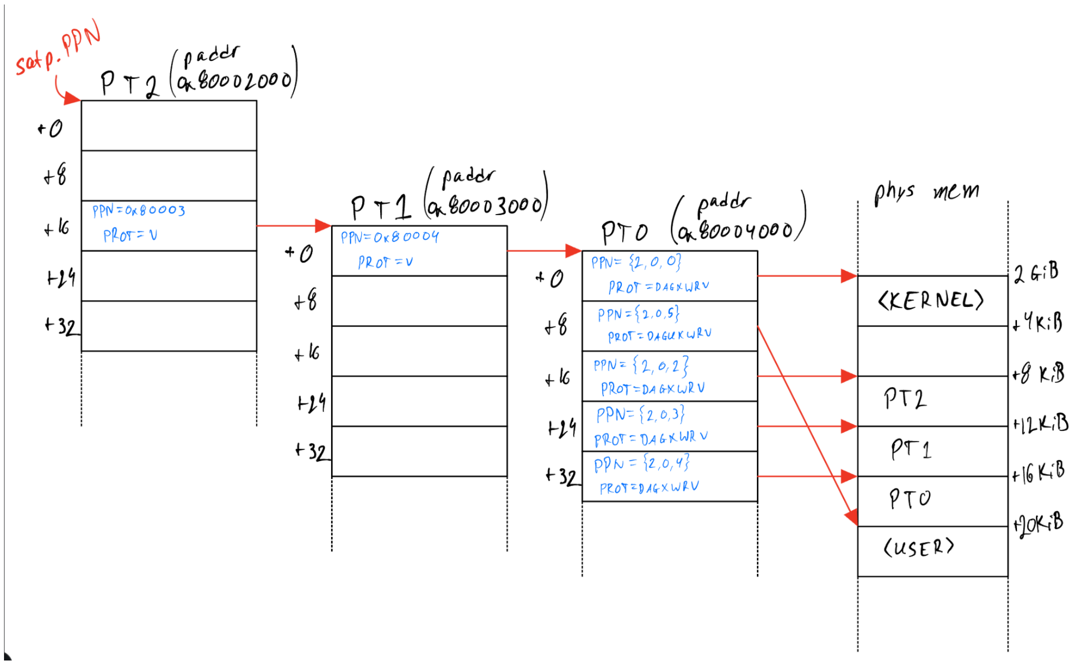


(b) Screenshot of WaveForm. *(Note: this figure omits redundant cycles to improve readability)*.

Figure 7.4: Out-of-Order Demonstration with CVA6: FPU and LSU finish before MULT [44]. In the Out-of-Order lab, participants are asked to write an assembly program to demonstrate code with and without different data-hazards.



(a) Diagram provided to students to illustrate transitioning between privilege modes (from “RISC-V Bytes: Privilege Levels” by Daniel Mangum [45])



(b) Page table diagram of provided OS *(student submission)*

Figure 7.5: The Virtual Memory Lab aids students in understanding concepts such as physical vs. virtual memory, page tables, privilege levels, and trap handling in RISC-V architecture.

7.2 There is high demand for hands-on learning experiences.

Because “Labs with CVA6” is available under the BSD-3-Clause license, it has attracted attention from many non-ECE 154B audiences including instructors seeking to enrich their own architecture courses, researchers aiming to familiarize themselves with the specifics of CVA6, and SystemVerilog beginners eager to learn best practices. In addition, I gave a well-appreciated talk about “Labs with CVA6” at “Latch-Up”, a conference hosted by The Free and Open Source Silicon Foundation [44]. During my presentation, I expressed the practical and unique skills that students acquire through studying the code of well-verified, open-source designs. This resonated deeply with several attendees, notably Rick O’Connor, the President and CEO at OpenHW Group, who notified me of the new OpenHW Group RISC-V core, CV-Wally [46], that is designed as a supplemental codebase for the upcoming textbook, “RISC-V System-on-Chip Design”. The popularity that “Labs with CVA6” has seen, and the recent creation of CV-Wally shows that there is strong demand for curriculums that offer transparency on implementation methods of real-world designs.

Chapter 8

Potential Applications in Other Classes

So far, this thesis has described approaches that could significantly benefit courses in introductory Verilog, digital design, and computer architecture. However, Verilog education stands to gain much more from open-source and publicly available resources. In this short section, I will summarize potential ways that open-source resources could be used in other advanced courses.

8.1 "Verification with UVM and SVA"

Verification is an enormous aspect of chip design, so teaching students the principles of Universal Verification Methodology (UVM) and SystemVerilog Assertions (SVA) can be of paramount value in industry preparation. Courses like North Carolina State University's ECE 748: "Advanced Verification with UVM" have seen large popularity as companies are in constant demand for well-trained verification engineers. As of 9/17/23, Verilator has limited compatibility with UVM and SVA, but is getting closer to full support every day [39, 40, 41]. If full UVM and SVA functionality is required, the open-source build-manager FuseSoC can provide an accessible CLI for proprietary tools, lowering the learning curve. Additionally, adopting a comprehensive Design Verification (DV) style guide and testbench linter can ensure that students continue following best practices when working on verification tasks. Notably, lowRISC has a popular and thorough UVM and SystemVerilog DV feature style-guide [28], and PySlint was advertised as a testbench linter at ORConf 2023 [47].

8.2 "Embedded Systems and SoC Design"

Embedded systems and SoC design courses can leverage a plethora of open-source, high-speed IP blocks that are commonly used in FPGA designs. There are many popular open-source designs for HDMI [48, 49, 50], Ethernet [51], PCIe [52, 53], AXI [54, 55], and more. An educator could provide a similar experience to my "Labs with CVA6" project by teaching students the inner-workings of advanced serial communication modules. Proficiency in high-speed interfaces is highly sought after in industry positions, so a course of this style could be highly beneficial for students.

8.3 "ASIC and VLSI Projects"

For courses focusing on Application-Specific Integrated Circuits (ASICs), open-source resources become critical. Licensing and signing Non-Disclosure Agreements for proprietary PDKs are often impractical or time-consuming for instructors, limiting course opportunities. Fortunately, initiatives like the OpenROAD Project and the SkyWater PDK (SKY130) offer students access to fully-featured ASIC flows. UC Berkeley's EE 194: "The Tapeout Class" utilized Hammer and OpenROAD to offer students the opportunity to tape out an SoC in a semester [56]. Moreover, affordable SKY130 fabrication opportunities like Tinytapeout (160 µm × 100 µm for $100) [57] and Google MPW lottery (2920 µm × 3520 µm for free) [58] enable students to take their designs from simulation to real-world fabrication, providing a hands-on experience of the ASIC design process.

Chapter 9

Conclusion

Universities should be working to lower the barrier of entry into SystemVerilog design. Throughout this thesis, several critical challenges in Verilog/SystemVerilog education have been addressed and resolved, contributing to the enhancement of the learning experience for students. These issues encompassed a range of areas, and the solutions put forward have had a significant impact.

- **Synthesizable vs. Verification Features:** A critical issue in Verilog education lies in distinguishing between synthesizable and verification features. This thesis has addressed this concern through multiple avenues. Using netlist graph viewers like DigitalJS Online, which enables students to visualize the synthesis process, helps visually demonstrate what constructs lead to valid netlists. Additionally, style guides like from the lowRISC Organization and linters such as Verilator document best practices for writing synthesizable code. Finally, autograders can provide immediate, personalized feedback to students on whether their code is both behaviorally correct and synthesizable.

- **Prevalence of Bugs in Common HDL Tools:** Common Verilog and SystemVerilog tools often suffer from bugs when using lesser-used features. To mitigate this, style guides and linters should be used to teach students the best syntax and strategies for avoiding common pitfalls. Autograders can also play a pivotal role by seamlessly testing student code across a multitude of tools. If a submission passes tests for several tools, it is much more likely that it will work properly for all tools.

- **Inaccessibility of Proprietary Tools:** The reliance on proprietary tools in Verilog education has been a barrier to accessibility for many beginners. Open-source tools highlighted in this work present a solution by being cost-free, easier to install, and more user-friendly. This shift towards open-source tools enhances equity in education and industry and enables a broader spectrum of engineers to engage effectively with Verilog.

- **The Lack of Reliable Educational Resources:** There is a shortage of reliable educational resources in Verilog/SystemVerilog. In this thesis, a comprehensive list of resources has been provided in Chapter 4, serving as a valuable reference for both instructors and students. These resources cover a range of use-cases and are designed to support a deeper understanding of synthesizable Verilog.

- **Interdisciplinary Nature of Chip Design:** Verilog and SystemVerilog education must acknowledge the interdisciplinary nature of chip design, involving both software and hardware components. This thesis emphasizes the importance of dedicating time to teaching software essential skills like Git version control and code scalability. Furthermore, I have described that DigitalJS Online should be used to create a stronger connection between Verilog code and hardware circuits, helping bridge the gap between hardware and software.

In light of the rapidly evolving landscape of open-source hardware, there is a pressing need for more universities and educators to modernize their digital design and computer architecture curriculums. Students deserve a curriculum that not only imparts technical knowledge but also inspires them to explore the field further. By incorporating elements like open-source tools, hands-on projects, and interactive platforms, educators can make Verilog education more engaging and appealing to a broader range of students. Furthermore, as demonstrated by the inclusion of "for-fun" leaderboards and "Ungrading" with autograders, fostering a collaborative and vibrant environment can help create a more valuable and enthusiastic learning experience. Institutions should be urged to reevaluate and revamp their Verilog courses to provide students with a more exciting and fulfilling educational experience, ultimately preparing them for the challenges and opportunities in the ever-evolving world of digital design and computer architecture.

Appendix A

Open-source Contributions

This appendix identifies several open-source issues and contributions I have made. Most of the contributions in this appendix are related to my efforts as an HDL educator.

A.1 Open-source issues and contributions created by me

- olofk/fusesoc#645: Improved inheritance elaboration
- verilator/verilator#4409: Check for conflicting options e.g. `--binary` and `--lint-only`
- olofk/edalize#389: Added support for additional Verilator modes
- Rain92/FPGA-Mandelbrot#1: Fixed build issues
- lowRISC/style-guides#66: Prohibit functions from using non-local references
- steveicarus/iverilog#980: Argumentless functions fix
- verilator/verilator#4172: Added NEWERSTD warning
- openhwgroup/cva6#1142: Improved Acronym List in Glossary

A.2 Open-source issues created by my students for my classes

- YosysHQ/oss-cad-suite-build#28: nextpnr-gowin support requested

Appendix B

Style Guide Survey

This appendix identifies several prominent Verilog and SystemVerilog style guides.

B.1 lowRISC Verilog Coding Style Guide

- `https://github.com/lowRISC/style-guides` [28]
- Projects that use lowRISC Verilog Coding Style Guide:
  - lowRISC: Ibex RISC-V Core [59]
  - lowRISC: OpenTitan RISC-V Core [60]
  - OpenHW Group: CV32E40P RISC-V Core [61]
  - OpenHW Group: CVA6 RISC-V Core [37]
  - OpenHW Group: FPnew Floating-Point Unit [62]
  - PULP Platform: Ara Vector Unit [63]
  - PULP Platform: MemPool Many-Core System [64]

B.2 BSG System Verilog Coding Standards

- `https://docs.google.com/document/d/1xA5XUzBtz_D6aSyIBQUwFk_kSUdckrfxa2uzGjMgmCU` [29]
- Projects that use BSG System Verilog Coding Standards:
  - Bespoke Silicon Group: BlackParrot RISC-V Core [65]
  - Bespoke Silicon Group: BaseJump Standard Template Library [66]

B.3 Company-Provided Style Guides

- Xilinx – HDL Coding Techniques [30]
- Intel – Recommended HDL Coding Styles [31]
- Lattice – HDL Coding Guidelines [32]
- Freescale – Verilog HDL Coding [67]

B.4 Miscellaneous Style Guides

- RSD RISC-V Core – Coding conventions [68]
- VeriGPU – Coding guidelines [69]

References

[1] J. Badlam, S. Clark, S. Gajendragadkar, A. Kumar, S. O'Rourke, and D. Swartz, "The CHIPS and Science Act: Here's what's in it." https://www.mckinsey.com/industries/public-sector/our-insights/the-chips-and-science-act-heres-whats-in-it, October, 2022. [Accessed 18-09-2023].

[2] L. Wang, "TSMC says three fabs to start production in 2024." https://www.taipeitimes.com/News/biz/archives/2022/08/31/2003784445, August, 2022. [Accessed 18-09-2023].

[3] rocket55, "Google Partners with SkyWater and Efabless to Enable Open Source Manufacturing of Custom ASICs." https://www.skywatertechnology.com/google-partners-with-skywater-and-efabless-to-enable-open-source-manufacturing-of-custom-asics/, November, 2020. [Accessed 18-09-2023].

[4] Google for Developers, "Build your own silicon." https://developers.google.com/silicon. [Accessed 18-09-2023].

[5] Kynix, "Detailed Explanation of Chip Design Flow." https://www.kynix.com/Blog/Detailed-Explanation-of-Chip-Design-Flow.html, December, 2017. [Accessed 18-09-2023].

[6] Intel, "Intel® Stratix® 10 Device Design Guidelines." https://www.intel.com/content/www/us/en/docs/programmable/683738/current/design-flow.html, August, 2022. [Accessed 18-09-2023].

[7] Anysilicon, "ASIC Design Flow – The Ultimate Guide." https://anysilicon.com/asic-design-flow-ultimate-guide/. [Accessed 18-09-2023].

[8] A. Olofsson, "Awesome Open-Source Hardware: List of awesome open source hardware tools, generators, and reusable designs." https://github.com/aolofsson/awesome-opensource-hardware. [Accessed 18-09-2023].

[9] S. Williams, "Icarus Verilog." `https://github.com/steveicarus/iverilog`. [Accessed 18-09-2023].

[10] W. Snyder, "Verilator: open-source SystemVerilog simulator and lint system." `https://github.com/verilator/verilator`. [Accessed 18-09-2023].

[11] R. Fuest, "GTKWave is a fully featured GTK+ based wave viewer for Unix and Win32 which reads LXT, LXT2, VZT, FST, and GHW files as well as standard Verilog VCD/EVCD files." `https://github.com/gtkwave/gtkwave`. [Accessed 18-09-2023].

[12] YosysHQ, "Yosys Open SYnthesis Suite." `https://github.com/YosysHQ/yosys`. [Accessed 18-09-2023].

[13] YosysHQ, "Nextpnr: portable FPGA place and route tool." `https://github.com/YosysHQ/nextpnr`. [Accessed 18-09-2023].

[14] "Verilog to Routing – Open Source CAD Flow for FPGA Research." `https://github.com/verilog-to-routing/vtr-verilog-to-routing`. [Accessed 18-09-2023].

[15] M. Shalan and T. Edwards, *Building OpenLANE: A 130nm OpenROAD-based Tapeout- Proven flow : Invited paper*, in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–6, 2020. `https://github.com/The-OpenROAD-Project/OpenLane`.

[16] O. Kindgren, "FuseSoC: Package manager and build abstraction tool for FPGA/ASIC development." `https://github.com/olofk/fusesoc`. [Accessed 18-09-2023].

[17] O. Kindgren, "Edalize: An abstraction library for interfacing EDA tools." `https://github.com/olofk/edalize`. [Accessed 18-09-2023].

[18] A. Olofsson, "Goodbye Make, Hello SiliconCompiler!." `https://youtu.be/GM9PKAfTlmQ`, 2023. Latch-Up.

[19] u/[deleted], "Modelsim and Questa license price: Too expensive?." `https://www.reddit.com/r/FPGA/comments/c8z1x9/modelsim_and_questa_license_price_too_expensive/`, July, 2019. [Accessed 18-09-2023].

[20] YosysHQ, "OSS CAD Suite: Multi-platform nightly builds of open source digital design and verification tools." `https://github.com/YosysHQ/oss-cad-suite-build`. [Accessed 18-09-2023].

[21] D. Richmond, "So, you want to be an open sourcerer?." `https://youtu.be/-rXgQxWRKIg`, 2023. Latch-Up.

[22] M. Materzok, "DigitalJS Online." `https://digitaljs.tilk.eu/`. [Accessed 18-09-2023].

[23] IEEE, *IEC/IEEE International Standard - Verilog(R) Register Transfer Level Synthesis, IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1* (2005). doi:10.1109/IEEESTD.2005.339572.

[24] S. Sutherland and D. Mills, *Synthesizing SystemVerilog: Busting the Myth that SystemVerilog is only for Verification*, March, 2013. Synopsys Users Group (SNUG) Silicon Valley conference, Santa Clara, California.

[25] IEEE, *IEEE Standard for Systemverilog–Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018). doi:10.1109/IEEESTD.2018.8299595.

[26] CHIPS Alliance, "SystemVerilog Test Report." `https://chipsalliance.github.io/sv-tests-results/`. [Accessed 19-09-2023].

[27] E. Sifferman, "Labs with CVA6." `https://github.com/sifferman/labs-with-cva6`. [Accessed 18-09-2023].

[28] lowRISC, "lowRISC Style Guides." `https://github.com/lowRISC/style-guides/tree/master`. [Accessed 19-09-2023].

[29] M. Taylor and The Bespoke Silicon Group, "BSG SystemVerilog Coding Standards." `https://docs.google.com/document/d/1xA5XUzBtz_D6aSyIBQUwFk_kSUdckrfxa2uzGjMgmCU/`. [Accessed 19-09-2023].

[30] Xilinx, "HDL Coding Techniques." `https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis/HDL-Coding-Techniques`. [Accessed 19-09-2023].

[31] Intel, "Intel® Quartus® Prime Pro Edition User Guide: Design Recommendations." `https://www.intel.com/content/www/us/en/docs/programmable/683082/23-1/recommended-hdl-coding-styles.html`. [Accessed 19-09-2023].

[32] Lattice, "HDL Coding Guidelines." `https://www.latticesemi.com/~/media/LatticeSemi/Documents/UserManuals/EI/HDLcodingguidelines.PDF?document_id=48203`. [Accessed 19-09-2023].

[33] CHIPS Alliance, "Verible: SystemVerilog parser, style-linter, formatter and language server." `https://github.com/chipsalliance/verible`. [Accessed 18-09-2023].

[34] dalance, "dalance/svlint: SystemVerilog linter."
https://github.com/dalance/svlint. [Accessed 18-09-2023].

[35] D. K. Tala and ASIC World, "Verilog Examples."
https://www.asic-world.com/examples/verilog/, 2014. [Accessed 19-09-2023].

[36] ChipDev, "ChipDev." https://chipdev.io/. [Accessed 19-09-2023].

[37] OpenHW Group, "CVA6 is an Application class 6-stage RISC-V CPU capable of booting Linux." https://github.com/openhwgroup/cva6. [Accessed 19-09-2023].

[38] Z. Snow, "zachjs/sv2v: SystemVerilog to Verilog conversion."
https://github.com/zachjs/sv2v. [Accessed 19-09-2023].

[39] W. Snyder, "Support full UVM -cc code generation – Verilator Issue #1538."
https://github.com/verilator/verilator/issues/1538. [Accessed 19-09-2023].

[40] K. Bieganski, "Open source design testing and verification with UVM and Verilator." https://youtu.be/2zOmpArtdH4, 2023. ORConf.

[41] Divya2030 and W. Snyder, "Limited Support for SystemVerilog Assertions (SVA) – Verilator Issue #4425."
https://github.com/verilator/verilator/issues/4425. [Accessed 19-09-2023].

[42] A. Kohn and S. D. Blum, *Ungrading: Why Rating Students Undermines Learning (and What to Do Instead) (Teaching and Learning in Higher Education)*. West Virginia University Press, 2020.

[43] S. Blum, "The significant learning benefits of getting rid of grades."
https://www.insidehighered.com/advice/2017/11/14/significant-learning-benefits-getting-rid-grades-essay, November, 2017.

[44] E. Sifferman, "Using CVA6 in Architecture Education."
https://youtu.be/6cvJfdh5msQ, 2023. Latch-Up.

[45] D. Mangum, "RISC-V Bytes: Privilege Levels."
https://danielmangum.com/posts/risc-v-bytes-privilege-levels/, December, 2021. [Accessed 20-09-2023].

[46] OpenHW Group, "CORE-V Wally is a configurable RISC-V Processor associated with RISC-V System-on-Chip Design textbook.."
https://github.com/openhwgroup/cvw. [Accessed 18-09-2023].

[47] S. S. Paul, "SystemVerilog Testbench linting with open-source."
https://youtu.be/ypOh9rCRypI, 2023. ORConf.

[48] S. Puri, "Send video/audio over HDMI on an FPGA." https://github.com/hdl-util/hdmi. [Accessed 19-09-2023].

[49] W. Green, "FPGA display controller with support for VGA, DVI, and HDMI.." https://github.com/projf/display_controller. [Accessed 19-09-2023].

[50] C. Wolf, "SimpleVOut: A Simple FPGA Core for Creating VGA/DVI/HDMI/OpenLDI Signals." https://github.com/cliffordwolf/SimpleVOut. [Accessed 19-09-2023].

[51] A. Forencich, "Verilog Ethernet components for FPGA implementation." https://github.com/alexforencich/verilog-ethernet. [Accessed 19-09-2023].

[52] A. Forencich, "Verilog PCI express components." https://github.com/alexforencich/verilog-pcie. [Accessed 19-09-2023].

[53] enjoy-digital, "Small footprint and configurable PCIe core." https://github.com/enjoy-digital/litepcie. [Accessed 19-09-2023].

[54] PULP Platform, "AXI SystemVerilog synthesizable IP modules and verification infrastructure for high-performance on-chip communication." https://github.com/pulp-platform/axi. [Accessed 19-09-2023].

[55] A. Forencich, "Verilog AXI components for FPGA implementation." https://github.com/alexforencich/verilog-axi. [Accessed 19-09-2023].

[56] J. Zhao, "Tapeout-in-a-Semester: The Organization of Berkeley's Tapeout Course." https://youtu.be/slIVkBrkgaM, 2023. Latch-Up.

[57] M. Venn, "Tiny Tapeout." https://tinytapeout.com/. [Accessed 19-09-2023].

[58] Efabless, "Caravel User Project." https://platform.efabless.com/design_catalog/asic_platform/174. [Accessed 19-09-2023].

[59] lowRISC, "Ibex." https://github.com/lowRISC/ibex. [Accessed 18-09-2023].

[60] lowRISC, "OpenTitan." https://github.com/lowRISC/opentitan. [Accessed 18-09-2023].

[61] OpenHW Group, "CV32E40P." https://github.com/openhwgroup/cv32e40p. [Accessed 18-09-2023].

[62] OpenHW Group, "FPnew Floating-Point Unit." https://github.com/openhwgroup/cvfpu. [Accessed 18-09-2023].

[63] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, *Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **28** (2020), no. 2 530–543.

[64] M. Cavalcante, S. Riedel, A. Pullini, and L. Benini, *MemPool: A shared-L1 memory many-core cluster with a low-latency interconnect*, in *2021 Design, Automation, and Test in Europe Conference and Exhibition (DATE)*, (Grenoble, FR), pp. 701–706, March, 2021. doi:10.23919/DATE51398.2021.9474087.

[65] D. Petrisko, F. Gilani, M. Wyse, C. D. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, J. A. Joshi, M. Oskin, and M. B. Taylor, "BlackParrot: An Agile Open Source RISC-V Multicore for Accelerator SoCs." doi:10.1109/MM.2020.2996145. IEEE Micro Special Issue on Agile and Open-Source Hardware, July/August.

[66] Bespoke Silicon Group, "BaseJump Standard Template Library." `https://github.com/bespoke-silicon-group/basejump_stl`. [Accessed 18-09-2023].

[67] Freescale Semiconductor, "Freescale Verilog HDL Coding." `https://michaeltaylor.org/edu/papers/FreescaleVerilog.pdf`. [Accessed 20-09-2023].

[68] RSD, "RSD Coding conventions." `https://github.com/rsd-devel/rsd/wiki/en-devel-coding-convention`. [Accessed 20-09-2023].

[69] H. Perkins, "VeriGPU Coding guidelines." `https://github.com/hughperkins/VeriGPU/blob/main/docs/coding_guidelines.md`. [Accessed 20-09-2023].