2026 ASEE ANNUAL CONFERENCE June 21–24, 2026 | Charlotte Convention Center, NC
Host Sponsor UNC Charlotte
ASEE AMERICAN SOCIETY FOR ENGINEERING EDUCATION
Paper ID #52197

Development containers: a tool for authentic, scalable active learning in digital logic and computer architecture

Bill Siever, Washington University in St. Louis

Bill Siever is a Teaching Professor at Washington University in St. Louis. Bill's background includes both Computer Science (B.S. and M.S.) and Computer Engineering (PhD). He has taught courses in Computer Architecture at both the undergraduate and graduate level and taught introductory courses at five different institutions as well as a wide variety of computer science courses.

James Feher, Washington University in St. Louis
Michael James Hall
Roger D Chamberlain

Development containers: a tool for authentic, scalable active learning in digital logic and computer architecture

Abstract

The bulk of topics in many introductory digital logic / computer architecture courses fall into two major thrusts: a) the concepts that support CPU implementations and b) the process by which programs can be represented in machine language and executed. Both areas can require significant infrastructure to convey, such as simulators/synthesis tools, cross compilers/assemblers, and, often, specialized hardware. Although there are a multitude of infrastructure options available, there are corresponding compromises that impact course scale, active learning, adoption and support costs, authenticity of the experience, and learning outcomes.

As part of a recent redesign of an introductory course in digital logic and computer design, the authors explored the use of development containers (dev containers) and open source resources to provide scalable, low cost, authentic experiences with concepts in both CPU implementation and program representation / execution. Dev containers are, essentially, a blueprint for a full-featured, integrated development environment. They rely on contemporary containerization (e.g., Docker) for portability and consistency. Dev containers can specify both specific versions of underlying tools (e.g., a specific version of a synthesis tool) as well as features of the development environment, like inclusion of a waveform viewer.

The dev container used in this work includes support for Hardware Description Language development (Verilog) and work with RISC-V (assembly and C; simulation and deployment to hardware). The use of dev containers allowed active learning activities that were not feasible previously, such as launching a full-featured development environment on student laptops within the first 5 minutes of class sessions. The work includes features that: 1) distinguish it from other academic uses of development containers, such as seamless simulation of logic gate behavior, visualization of logic signals, and deployment to low-cost FPGA hardware; 2) distinguish it from non-academic tools by providing pedagogical support; 3) distinguish it from open source tools in many comparable courses, like the combination of easy adoption coupled with support for hardware deployment; and 4) are platform agnostic, but develop transferable skills. The approach has undergone four semesters of refinement, has been used by more than 250 students, has significantly reduced the barrier to access course content, and can easily be adopted and adapted by others. The paper outlines the infrastructure used and the types of activities that it facilitated as well as providing guidance for others who may want to adopt the environment discussed or a similar approach.

Introduction

Courses in digital logic and computer architecture often cover basic CPU design choice starting from some level of abstraction. The starting point and depth of CPU design options vary based on their target audience and curricular needs, with two extremes:

1. The Electrical Engineering end of the spectrum, such as in [1] and [2], which often includes significant focus on low-level details, like transistor-level implementations of digital logic gates at the expense of the depth of coverage of computer architecture topics.
2. The Computer Science end of the spectrum, such as in [3] and [4], which often relies on abstraction of everything below the digital logic primitives. Often these courses are streamlined to provide a functional understanding of the low-level details via model CPU implementations.

Courses at the Electrical Engineering end of the spectrum often include hands-on (lab) activities, some of which include significant work with analog signals. Courses on the Computer Science end of the spectrum often focus on a higher level of abstraction and use pure, digital signals. The work described in the remaining sections covers the tools curated for a course that falls in the middle of these two extremes and could be readily extended to support either. A brief introduction to the tools is given in [5]. Section 1 covers the context of the course and the factors that influenced the choices that were made. Section 2 provides an overview of the relevant development container concepts and how they support the course content. Section 3 gives an overview of the features provided along with their alignment with the course topics and assignments. Finally, Section 4 summarizes the experience and identifies areas for future work.

1 Motivation, Context, and Goals

This work was a product of redesign in the introductory course in digital logic and computer design at Washington University in St. Louis, a midsize, private, R1 university in the United States. The course is the primary introduction to computer engineering in a required, three-course sequence designed for the BS in Computer Engineering program. In addition, it provides the required digital logic content for the BS in Electrical Engineering program and is an elective for Computer Science students. Computer Engineers typically take the course in their sophomore year, but enrollment includes all levels of undergraduates. Historically, it has been slightly more on the Electrical Engineering end of the content spectrum. Since it is required for multiple degree programs, it is offered every semester and enrollment fluctuates, with an average of 51 students per semester over the last decade. However, enrollments of the updated version were 69 (fall 2024), 94 (spring 2025), and 87 (fall 2025). The sole prerequisite is CS1 (Introduction to Computer Science, i.e., computer programming).

The catalysts for course redesign were a combination of changes in faculty, largely due to retirements, and the length of time that had elapsed since the course had a substantial review. The redesign was an iterative process and multiple stakeholders were consulted to ensure the course would still meet the needs of each. The process included:

- Review of existing course materials, including syllabi and recent assignments.
- Documentation from the prior instructor suggesting areas that needed revision.
- Discussions about course requirements and potential changes with the prior instructor, the Computer Engineering oversight committee, and Electrical Engineering faculty.
- Consultations with department administration on course articulation concerns.
- Surveys of and interviews with Teaching Assistant candidates, all of who had taken the prior version of the course and some of whom had also already served as Teaching Assistants for it. The surveys and interviews included questions about friction points encountered by students. Interviews also foreshadowed the types of content and delivery changes being considered for feedback.
- Web searches for comparable courses at peer institutions and relevant tools/infrastructure.
- A literature search for recent publications for related courses and content.
- The likely pedagogical preferences for faculty who would be teaching the course in successive semesters.

Several themes emerged from the review process.

First, access to resources needed for coursework was a point of friction and getting worse: Major assignments used hardware, a Field Programmable Gate Array (FPGA) board, that was housed in a single lab. Both the number of units and the times they were available were limited. Moreover, the platform was nearing the end of production and support would be unavailable in the future. A commercial development environment was also used, but many students were unable to install it on their personal machines due to incompatibility or resource requirements. The software was also nearing the end of its support cycle and difficult for IT to maintain in both labs and remote access environments. Not surprisingly, a substantial amount of existing course content was developed specifically for the hardware and development environment being used. As is often the case with powerful professional tools in introductory courses, a significant amount of effort was required to carefully guide students through usage of the environment, which some stakeholders felt was a distraction from course content. Updating to the latest generation of hardware/software would have required substantial review. Despite all of the challenges introduced by the platforms being used, most stakeholders felt that exposure to the hardware concepts was vital. Moreover, prior literature provides some evidence that students may perform better when they have significant access (nearly continuous) to hardware platforms [6].

Second, the three course sequence was well suited to the needs of the Computer Engineering program, but the topics partitioned in the introductory course were not well aligned with comparable courses commonly taken by transfer students. Consequently, there was no satisfactory path for handling students who had taken a somewhat similar course elsewhere: they would either be missing vital content required for the second course and likely struggle, or they would be required to take a course that included a lot of redundant material and was not a good use of their time. In addition, there was no satisfying way to provide bridge materials for self-study, since the missing content was largely experience with the hardware and development environment being used.

Context

The course is a 3-unit (3 hour) course that meets for 2, 80-minute sessions per week for 14 weeks of instruction.

The prior version of the course followed a traditional format, using contact time for lectures. Homework was divided into 6-8 written problem sets, some of which required simulation of systems, and three "labs." The labs had more extensive use of simulation and typically culminated in deployment to the hardware in the instructional laboratory.

Many of the future instructors had experience with an introductory course that is also hardware based, but the majority of contact time is spent in active learning activities ([7], [8]). In their experience, this approach increased student engagement, and, consequently, the instructors were interested in changes that would support such a structure. One goal was to have weekly in-class active learning activities and ensure that the majority of homework assignments included hardware deployment.

Prior content was based on a variety of sources, including a digital logic text that had not been updated for nearly a decade, a variant of the SRC CPU architecture [1], and a novel computation pipeline [9]. Although the prior topics provided a rich and broad set of topics, there was a desire to switch the introductory architecture to RISC-V, which is better aligned with transfer courses and provides a platform for later courses in the Computer Engineering sequence. Students also often express a desire to work with "real" and contemporary platforms rather than educational-use models.

Goals

Many of the overall goals for the redesign of the course impacted the development of the environment as well as its applicability to other courses and activities. Many goals were explicit requirements of the program or various stakeholders, but some were personal pedagogical preferences and practical considerations. The most significant goals:

1. Content delivery should be approximately 50% active learning activities.
2. Students should experience contemporary approaches to digital logic.
3. Students should experience working with hardware.
4. Computer architecture content should provide a foundation for successive courses.
5. Coverage of computer architecture should be increased. To compensate, some content should be moved to other required courses for relevant audiences.
6. Course resource costs should be sustainable for both students and the institution.
7. Instructor effort to create, maintain, deploy, and assess assignments should be reasonable.
8. The structure should be amenable to course scaling, both to large and small enrollments.

1.1 Related Work and Influences

The most influential related work was Steven Bell's work at Tufts University [10], which established three goals for lab assignments that were also selected to be targets for both active-learning and the homework/lab assignments: 1) work should be *tangible* rather than abstract, 2) activities should be *accessible*, and 3) work should represent *professional practice*. More importantly, it was an example of a comparable course using a low-cost, open hardware platform. This provided a proof-of-concept that a low-cost hardware platform could be leveraged to make such courses accessible. However, the work utilized vendor-specific development environment, which did not have a pedagogic focus, required substantial resources to install and, although free for academic use, still required maintaining licenses.

Another significant influence was the EDAcation project from the Computer Architecture for Embedded Systems (CAES) department at the University of Twente [11]. EDAcation integrates a variety of open-source tools for digital logic design as an extension on the Visual Studio Code (VSCode) development environment. EDAcation is in use in comparable courses at the University of Twente and provided a proof-of-concept that open source tools are sufficiently robust for such courses. However, there were some deficiencies that didn't meet course goals. First, it did not support hardware platforms in the target price range. Second, it did not include facilities to deploy to hardware. Finally, it achieved cross-platform support by relying on JavaScript implementations of major components, which performed poorly and could lead to an unsatisfactory experience during in-class, active-learning activities. Since the initial review EDAcation has been updated to support local installation of native tools and performs well, but at the expense of significantly higher installation requirements.

2 Development Container Concepts and Implementation

The software stack for the development environment consists of two major layers: A Docker [12] image and an instance of Visual Studio Code (VSCode) [13]. Docker uses the concept of containers to "contain" resources necessary for an application.

Docker has become one of the leading approaches to virtualization and is widely used for both cloud-based services and to provide portability and operating system isolation for desktop applications. A Docker image is a group of software needed for a particular use-case. Images are typically based on Linux distributions and can leverage a wide variety of open-source tools. An image can run on a Docker engine, which is the interface to a real machine. Docker engines exist both for cloud and desktop.

The user interface for the development environment is provided by Visual Studio Code (VSCode) [13], which is an extendible, customizable, open-source editor developed by Microsoft. VSCode has become one of the dominant editors used by software developers [14]. VSCode supports extensions, which can be used to add features and customize its behavior. It also supports the Development Container Specification [15]. A development container is defined by a text file that provides a blueprint for a combination of features for a full-fledged development environment. This blueprint can specify both a specific Docker image to use and VSCode settings and extensions. The former provides a way to ensure a cohesive set of development software and

libraries, and the latter provides a way to specify the overall development environment behavior and additional tools that are desired.

A Development Container can be run either locally, via a local installation of both VSCode and a Docker engine, or via cloud-based services. The most commonly used cloud-based service is GitHub's Codespaces [16], which continues to be free for academic use at this time (2026). Academic use of Codespaces has been reported to reduce barriers to students, like technical difficulties and installation efforts; and to reduce instructor support effort [17]. Codespaces require a GitHub repository, which includes both assignment-specific files and the configuration for a development container.

The course starts students in Codespaces for simplicity, which eliminated most variance of individual setup and other barriers, like insufficient space to install software. Instructors often use local setup, which can have slightly higher performance and is more responsive when developing and updating the development container itself. Each semester several students have decided to install locally, with few problems reported. The configuration has been tested locally on Windows, macOS, and Linux.

Figure 1 shows the structure of cloud-based hosting. GitHub's cloud hosts and runs the Docker image as well as a VSCode instance. A web browser is used as a front-end to the environment. Figure 2 shows the structure of the code repository used for each project. The repository includes three major components: 1) the configuration files for the development container that define the development environment, 2) common resources used for several assignments/projects, and 3) the assignment specific files. The environment can be adopted/adapted to different assignments (or by different people) by cloning or forking the repository and modifying the assignment-specific files.

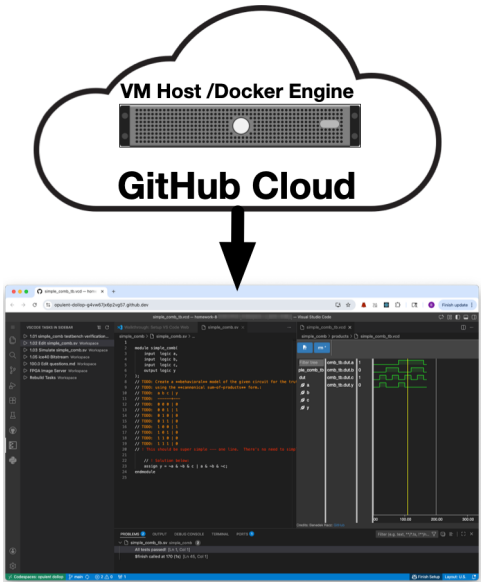

Figure 1: GitHub's Codespaces host Development Containers, allowing students to access development environments entirely from their local web browser without any local installation. However, the same container can be run locally on either Windows, macOS, or Linux via local installations of both VSCode and Docker.

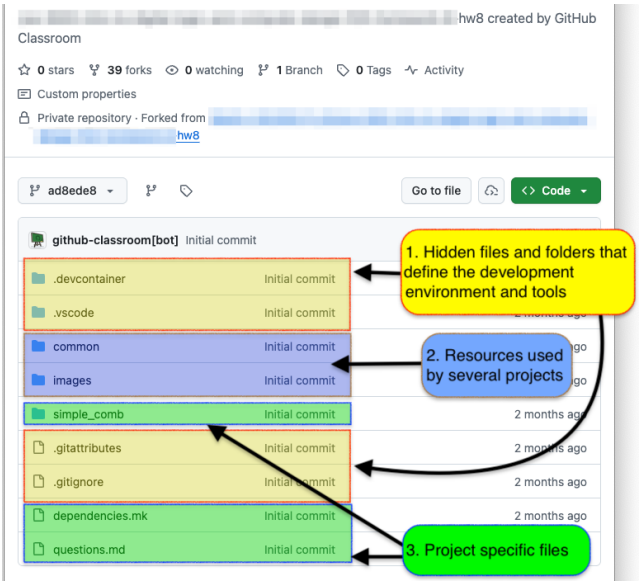

Figure 2: A code repository is used for each project. This example includes: 1. the configuration files for the development container that define the development environment, 2. common resources used for several assignments/projects, and 3. the assignment specific files. 1 and 2 are, essentially, static. 3 is customized for each assignment. The repository facilitates both the cloud-based development environment (a Codespace), version control (via Git), and assignment distribution and collection.

2.1 Peripheral Benefits

The use of GitHub repositories has additional benefits:

- Easy distribution and collection of assignments via GitHub Classroom [18], which can create a repository for each student on each assignment.
- Inherent versioning of student work and cloud-backup.
- Support for multiple approaches to "autograders" [19], [20].
- Instructors can launch the student's environment to assist with asynchronous debugging (without exchanging emails and files!)

3 Features and Assignments Supported

Early course assignments include purely written work and use of a schematic-based digital logic tool ([21]), but the majority of both active-learning assignments and homework/lab assignments utilize the container-based environment. The features supported and open-source projects leveraged are best explained in the context of specific activities and assignments that use them, which correspond to common content in many digital logic and computer architecture courses.

Digital Logic Fundamentals via Hardware Description Languages

Figure 3 shows an example of an in-class, active-learning activity that covers combinational logic design using Verilog. It requires students to implement a simple look-up table. The development container provides easy-access to required tasks, like running a test bench, editing code, and testing their work in an interactive simulator. The activity is included in a set of activities designed to be completed in approximately 80 minutes, allowing students to get hands-on experience with digital logic concepts during class sessions. Figure 4 shows the results of running the instructor's exhaustive test bench. Students can review a general summary of the test bench execution as well as explore the test signals applied and their impact on the Verilog model.

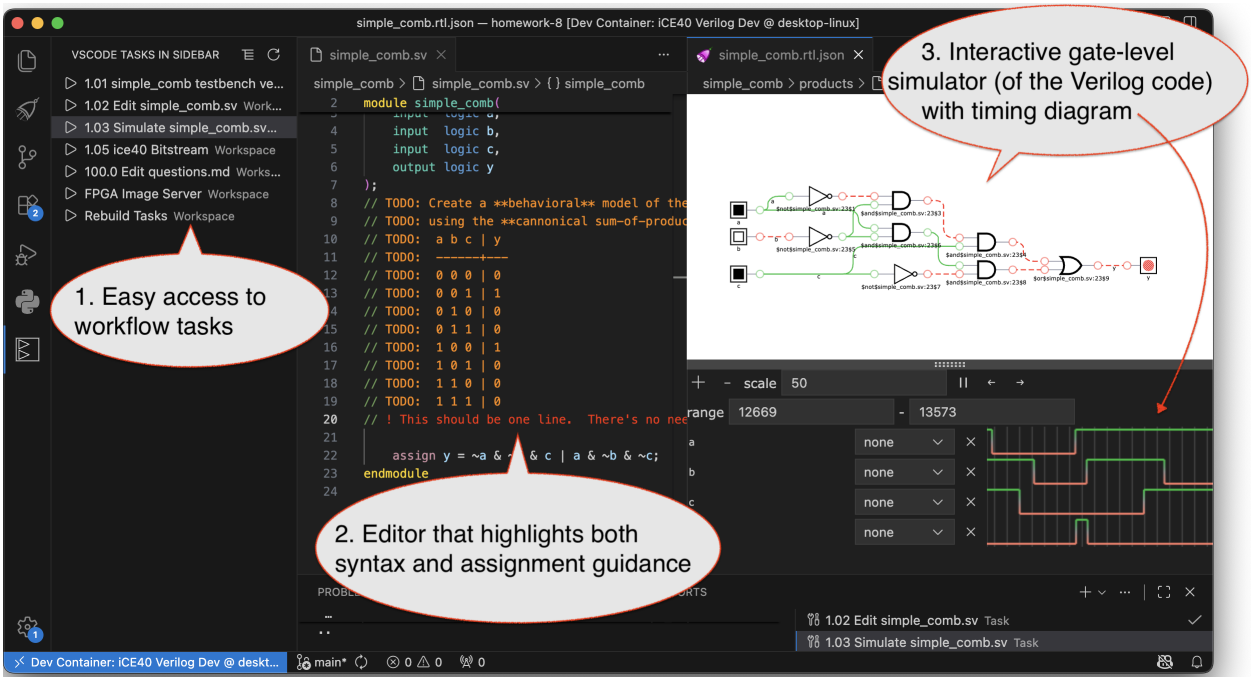

Figure 3: An example of a combinational logic in-class, active-learning activity. Students have easy-access to required tasks (1), like running a test bench, editing code, and testing their work in an interactive simulator (2). Signal waveforms can be used to better understand the overall behavior (3).

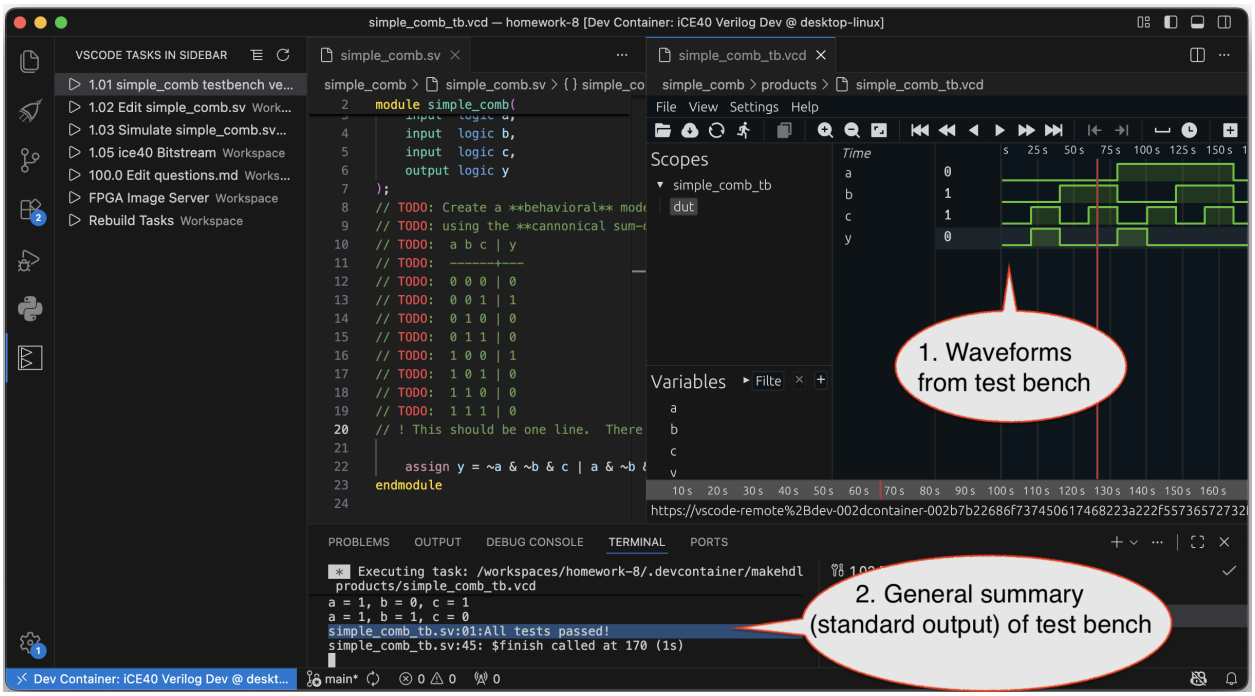

Figure 4: An example of an exhaustive test bench for a simple combinational logic circuit. Students can see both the waveforms of signals involved in the model and the overall status of the test bench.

There are extensive extensions available for VSCode that can be leveraged to provide both development and pedagogical support. Examples used include:

1. *DigitalJS* [22] and the DigitalJS VSCode extension [23] to simulate and visualize digital logic circuits [24].
2. *Surfer* [25] and the Surfer VSCode extension [26] to display waveforms.
3. *Tasks In Sidebar* VSCode extension [27] to manage and run tasks.
4. *HighlightingVerilog-HDL/SystemVerilog/Bluespec SystemVerilog* VSCode extension [28] to provide syntax highlighting for Verilog.
5. *Better Comments* VSCode extension [29] to highlight assignment guidance.

In addition, Icarus Verilog [30] is used to run test benches and Yosys [31] is used for gate-level simulation and synthesis. Test benches are provided for most course work, but some activities require completing a basic test bench.

Although the example above depicts very simple combinational logic, the tools work with the majority of topics used in introductory digital logic: sequential circuits, structural and behavioral designs, and hierarchies of modules.

In addition to being able to work in simulation, it is possible to deploy the code to hardware. The course uses a low-cost, open-hardware platform that has a proven track record in introductory

courses [10], the UPduino [32]. The total kit used in the course and currently supported by the environment consists of:

- The UPduino v3.1, which includes a Lattice iCE40 Ultra Plus 5k with 5280 LUTs, 1024 kbits SPRAM, and 120 kbits block RAM.
- A generic I/O board, typically marketed as an "LED & Key" module, based on the TM1638. They have 8, 7-segement displays, 8 buttons, and 8 LEDs.
- A full-size breadboard.
- A 10cm, 5-pin female-to-male jumper cable.

The UPduino is $36.00 (USD) at the time of this writing and the entire kit can be assembled for an average price under $50 each.

The instructors provide top-level modules and I/O mappings, allowing the students to focus on introductory topics. Figure 5 shows the completed hardware kit being used for the combinational logic circuit example shown in previous figures. Three buttons are used to provide the inputs, a (A), b, and c (C). The current value of the input is shown as a 1 or 0 immediately above the controlling button. The output, y, is also depicted and controlled in real-time by the combinational logic generated from the Verilog model.

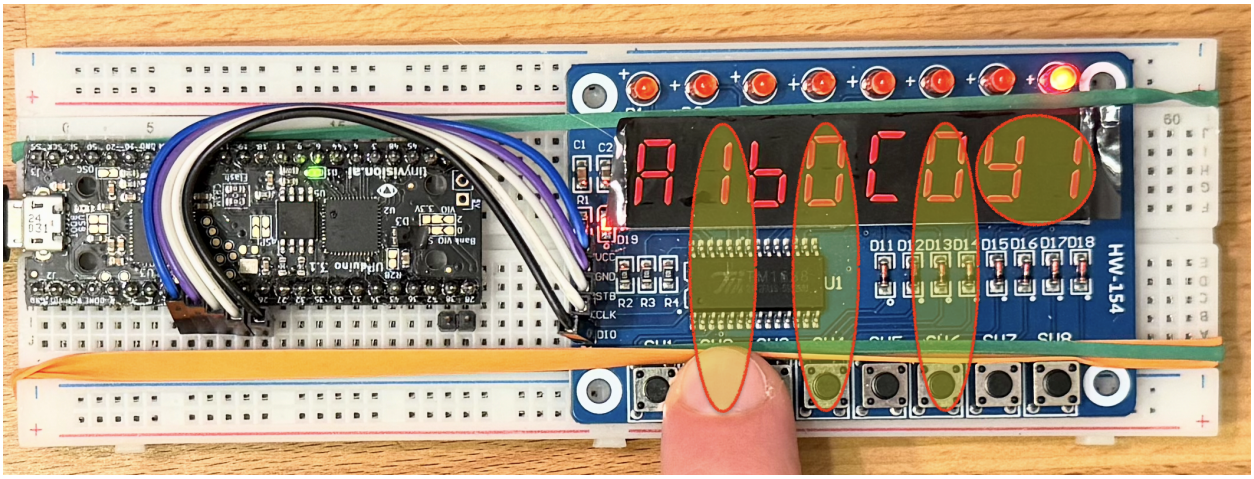

Figure 5: Hardware setup used in class.

Examples of digital logic assignments given in the class include:

- Completion of full-adders and multi-digit adders built (structurally) from full-adders. This exercise includes tests on the hardware and observations of the behavior of carry outs.
- A state machine to control a washing machine. The assignment can be tested in simulation and via a provided test bench. The assignment concludes with deployment to the hardware and a demonstration that the state machine will correctly control the behavior of laundry cycles.

- Elements of the datapath of a basic (RISC-V) CPU. The students complete a basic arithmetic logic unit, which is integrated with a register set (studied in active learning sessions) to create a "calculator". The students themselves serve as a control, dictating which operation code is provided to the ALU, which registers are used as operands, and both where results are written and when they are written. Students have to demonstrate their understanding by "controlling" their calculator to perform basic operations, like computing 3+5 and storing the result in register 2. This activity provides a visceral feel for how instructions are decoded and control the datapath.
- Augmenting a RISC-V model's datapath and control to support additional instructions.

Architecture Fundamentals and Assembly Language

The course introduces students to the RISC-V architecture. As with Verilog, a VSCode extension provides RISC-V syntax highlighting [33]. Execution can be simulated via the VSCode Venus extension [34], which is based on the popular Venus [35] RISC-V simulator. Venus provides tools that are important both for pedagogical reasons and for development, such as the ability to step through code, to use break points, and to observe machine state, like registers and memory. *Open source* extensions provide a unique opportunity to customize behavior for course-specific needs. As shown in Figure 6, the Venus simulator has been customized to include support for the I/O board used in the course. The simulator has been updated with `ecall` (environment call) support to interact with the I/O board. `ecall`s are provided for reading buttons, controlling the seven segment displays, controlling the LEDs, and UART communication for console I/O.

The combination of development environment and hardware provide an opportunity to introduce soft cores and FPGA memory organization. The PicoRV32 [36] soft core is used with custom firmware, which supports the I/O board. Students can deploy the soft core to the FPGA's flash memory, effectively transforming the hardware into a RISC-V computer. The underlying Docker image includes the GNU RISC-V toolchain [37], allowing students to easily port their assembly language code from the simulator to a firmware binary that can be deployed to the soft core. The approach provides an opportunity to talk about the variety of memory types available and how each contributes to the system: the UPduino's flash stores both the bitstream that configures the UPduino as a RISC-V computer and the firmware it will use. The FPGA's block RAM serves as both the data segment and the text segment.

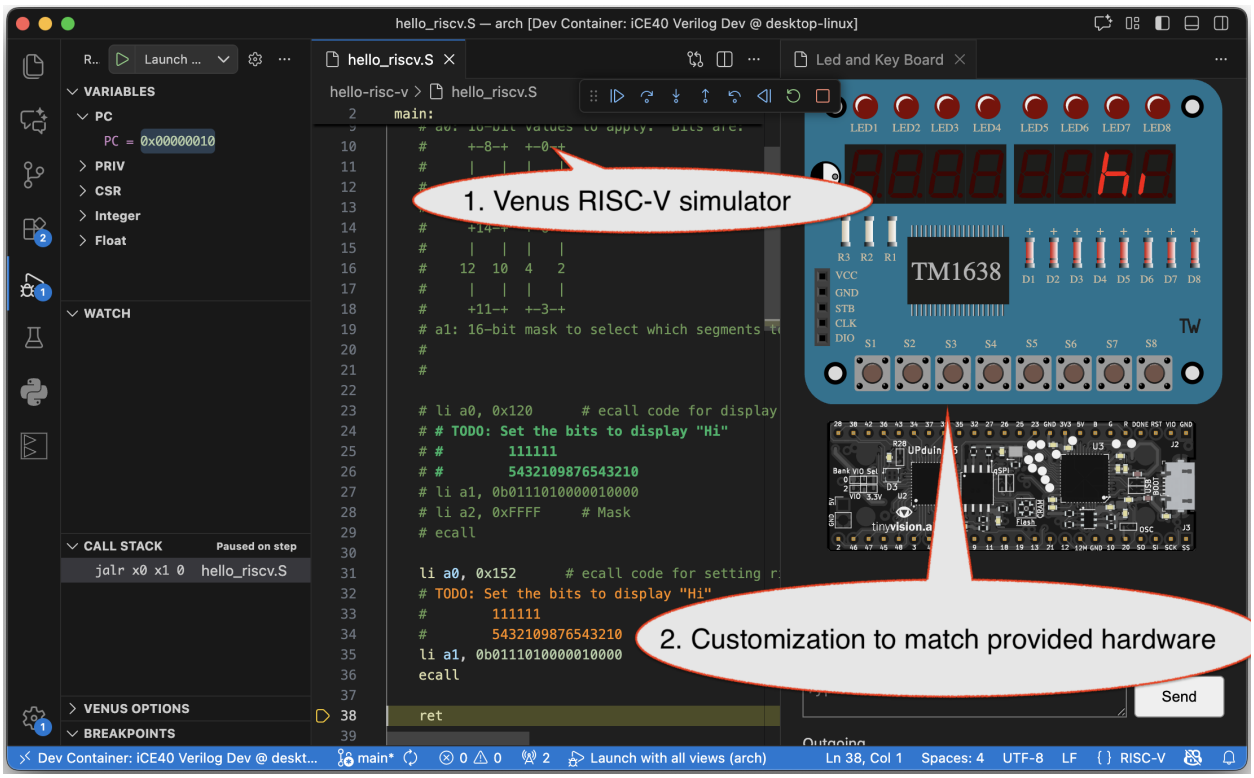

Figure 6: Architecture is explored via assembly language programming on the RISC-V ISA. The Venus simulator provides support to understand the semantics of assembly language instructions and their impact on machine state.

Examples of architecture-focused assignments given in the class include:

- Writing assembly language to control the I/O board.
- Writing code for for operations that aren't supported by hardware instructions, like unsigned integer multiplication.
- Designing `delay(int ms)` functions and tuning them for three environments: the simulator; the PicoRV32 soft core; and the final, student-augmented RISC-V model.
- Implementing a primitive state machine for a washer (previously implemented via digital logic).

Integration of Hardware and Software Concepts

The class culminates in an assignment that combines multiple aspects of prior assignments. Students update the pedagogically-focused RISC-V Verilog model that accompanies "Digital Design and Computer Architecture, RISC-V Edition" [38] to support additional instructions. As with prior assignments, the assignment ends in deployment: the prior assembly language implementation of a washing machine, which required use of the newly provided instructions, is deployed to the student's completed RISC-V model. The entire CPU is then synthesized and

deployed to the hardware, effectively creating a custom RISC-V computer using the student's work that can execute the student's program.

Overall Content Progress Supported

The course contains 11 active-learning sessions and 11 corresponding homework assignments. Assignments are a mix of written problems, description/analysis/justification, and reflection. There is a significant amount of repetition and review. The students implement the same state machine in three different ways: first, at the logic gate level of design using a schematic-based tool (e.g., [39] or [21]), second, using Verilog, and third, using RISC-V assembly language. Student-developed assembly language is run on both the PicoRV32 soft core and the student-augmented RISC-V model [38].

Iterative Refinement

Over the course of three semesters the environment has been refined and enhanced to streamline the student experience, to simplify development of new projects/assignments (which benefits instructors and students), and to further the learning objectives of the course:

- The bulk of error messages from underlying tools are now directed to a single, appropriate place, the "Problems" pane.
- Syntax highlighting tools have been selected based on student suggestions.
- Venus, the RISC-V simulator, has been enhanced to support the specific I/O board and I/O provided to students. RISC-V Environment Calls have also been provided to simplify interaction with the I/O.
- The user interface has has been streamlined to provide a simple, straight-forward experience in active learning sessions.
- The environment has been enhanced to simplify project configuration.

4 Conclusions and Future Work

The informal survey of teaching assistant candidates when using commercial tools indicated the homework and labs were a significant barrier for students, which led to many of the decisions to pursue an alternate environment. The fall 2026 standard course evaluation surveys included an optional, open-response question: "What did you like most about this course?". 86% of students participated in the evaluation. Approximately half of those (44% of those enrolled) provided an answer to this question. Approximately half of those (20% of those enrolled / 15 students) mentioned some aspect of the course strongly correlated with the active learning or homework facilitated by the environment described (e.g., "Homeworks ... reinforcing understanding", "work with real hardware", "applicable to industry"). The remaining comments were more generic. For example, other comments mentioned the overall content or described the integration of multiple concepts (the latter was also facilitated by the environment used). 20% of students voluntarily describing the homework and active learning as the best parts of the course seems promising.

(Other aspects of the course were also ranked favorably, so it wasn't merely that the homework was relatively good in contrast to an overall poor experience).

The approach described has been successfully used for four semesters with more than 250 students. Iterative updates have continued to improve the robustness of the environment and increase the quality of feedback provided on student activities. The use of development containers has provided low-barrier access to a niche that normally has significant overhead. It has also provided an opportunity to significantly reduce the complexity and distracting cognitive load that is required when working with vendor tools.

Future work includes: refinement of the environment, tools, and assignments; development of documentation and guides for others to adopt or adapt the approach; and assessment of student learning outcomes.

5 Acknowledgments

We would like to thank the colleagues who inspired much of this work, especially those who have shared resources and taken time to meet and discuss related efforts: Steven Bell ([10]), Hendrik Folmer ([11]), and Dustin Richmond ([40]).

Open clip art used in Figure 1 from [41] and [42].

References

[1] V. P. Heuring and H. F. Jordan, *Computer Systems Design and Architecture*. USA: Prentice-Hall, 2nd ed., 2003.

[2] M. M. Mano, C. Kime, and T. Martin, *Logic and Computer Design Fundamentals*. USA: Pearson, 5th ed., 2015.

[3] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*. USA: Pearson, 3rd ed., 2015.

[4] N. Nisan and S. Schocken, *The Elements of Computing Systems: Building a Modern Computer from First Principles*. The MIT Press, 2nd ed., 2021.

[5] B. Siever, M. Hall, J. Feher, and R. Chamberlain, "Teaching digital logic and computer architecture using open source tools," in *Proceedings of the 22nd ACM International Conference on Computing Frontiers: Workshops and Special Sessions*, CF '25 Companion, (New York, NY, USA), p. 38–41, Association for Computing Machinery, 2025.

[6] M. A. D. Mihaela Radu, Clint Cole and S. Sexton, "Extensive use of advanced FPGA technology in digital design education," in *2008 Annual Conference & Exposition*, no. 10.18260/1-2–4201, (Pittsburgh, Pennsylvania), ASEE Conferences, June 2008. https://peer.asee.org/4201.

[7] R. D. Chamberlain, J. Orr, D. Shook, and B. Siever, "Advancing your Arduino game: Early and engaging scaffolding for advanced CS," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2022, (New York, NY, USA), p. 1196, Association for Computing Machinery, 2022.

[8] R. D. Chamberlain, R. K. Cytron, D. Shook, and B. Siever, "Computers interacting with the physical world: A

first-year course," in *Workshop on Embedded and Cyber-Physical Systems Education* (R. Chamberlain, W. Taha, and M. Törngren, eds.), (Cham), pp. 197–205, Springer International Publishing, 2019.

[9] W. D. Richard, "Using Babbage's difference engine to introduce computer architecture," in *2017 IEEE International Conference on Microelectronic Systems Education (MSE)*, pp. 47–50, 2017.

[10] S. Bell, "Reimagining the digital lab with $30 FPGAs," in *2023 ASEE Annual Conference & Exposition*, no. 10.18260/1-2–44082, (Baltimore, Maryland), ASEE Conferences, June 2023. https://peer.asee.org/44082.

[11] U. of Twente, "EDAcation." Web: `https://edacation.github.io/`, 2022.

[12] "Docker." Web: `https://www.docker.com/`, 2026.

[13] "Visual Studio Code." Web: `https://code.visualstudio.com/`, 2025. Accessed: 2026-01-11.

[14] "Dev IDEs." Web: `https://survey.stackoverflow.co/2025/technology/#1-dev-id-es`, 2026.

[15] Microsoft, "Development containers: An open specification for enriching containers with development specific content and settings." Web: `https://containers.dev/`, 2026.

[16] "GitHub Codespaces." Web: `https://github.com/features/codespaces`, 2026.

[17] D. J. Malan, "Containerizing CS50: Standardizing students' programming environments," in *Proceedings of the Conference on Innovation and Technology in Computer Science Education*, vol. 1 of *ITiCSE 2024*, (New York, NY, USA), p. 534–540, Association for Computing Machinery, 2024.

[18] "GitHub Classroom." Web: `https://classroom.github.com/`, 2026.

[19] "Gradescope Autograders." Web: `https://gradescope-autograders.readthedocs.io/en/latest/`, 2026.

[20] "GitHub Classroom: Use Autograding." Web: `https://docs.github.com/en/education/manage-coursework-with-github-classroom/teach-with-github-classroom/use-autograding`, 2026.

[21] D. A. Poplawski, "A pedagogically targeted logic design and simulation tool," in *Proc. of Workshop on Computer Architecture Education*, pp. 1–7, 2007.

[22] M. Materzok, "DigitalJS." Web: `https://digitaljs.tilk.eu/`, 2026.

[23] Y. Yu, "DigitalJS - Visual Studio Marketplace." Web: `https://marketplace.visualstudio.com/items?itemName=yuyichao.digitaljs`.

[24] M. Materzok, "DigitalJS: A visual Verilog simulator for teaching," in *Proc. of 8th Computer Science Education Research Conference*, pp. 110–115, ACM, 2019.

[25] "Surfer." Web: `https://surfer-project.org/`, 2026.

[26] "Surfer - Visual Studio Marketplace." Web: `https://marketplace.visualstudio.com/items?itemName=surfer-project.surfer`.

[27] I. Radu, "VSCode Tasks in Sidebar - Visual Studio Marketplace." Web: `https://marketplace.visualstudio.com/items?itemName=iulian-radu-at.vscode-tasks-sidebar`.

[28] M. Hiramori, "Verilog-HDL/SystemVerilog/Bluespec SystemVerilog - Visual Studio Marketplace." Web: `https://marketplace.visualstudio.com/items?itemName=mshr-h.VerilogHDL`.

[29] A. Bond, "Better Comments - Visual Studio Marketplace." Web: `https://marketplace.visualstudio.com/items?itemName=aaron-bond.better-comments`.

[30] S. Icarus, "Icarus Verilog." Web: `https://steveicarus.github.io/iverilog/`, 2026.

[31] "Yosys Open SYnthesis Suite." Web: https://yosyshq.net/yosys/, 2026.

[32] tinyvision.ai, "UPduino v3.1 Codespaces." Web: https://tinyvision.ai/products/fpga-development-board-upduino-v3-1, 2026.

[33] S.-C. Sun, "Most Comprehensive RISC-V ASM Highlighting." https://marketplace.visualstudio.com/items?itemName=sunshaoce.RISC-V, Accessed Feb. 2025.

[34] H. Zebang, "Venus - Visual Studio Marketplace." Web: https://marketplace.visualstudio.com/items?itemName=ZAMBAR.riscv-venus-cs110, 2026.

[35] K. Vakil, "venus is a RISC-V instruction set simulator built for education." Web: https://github.com/kvakil/venus, 2026.

[36] "PicoRV32 - A Size-Optimized RISC-V CPU." Web: https://github.com/YosysHQ/picorv32.

[37] "RISC-V GNU Compiler Toolchain." GitHub Repository: https://github.com/riscv-collab/riscv-gnu-toolchain, 2017. Accessed: 2026-01-11.

[38] S. Harris and D. Harris, *Digital Design and Computer Architecture, RISC-V Edition*. Elsevier Science, 2021.

[39] "Logisim Evolution." Web: https://github.com/logisim-evolution/logisim-evolution, 2015.

[40] "Teaching with open source tools." Web: https://blog.yosyshq.com/p/community-spotlight-teaching-with-open-source-tools/, 2024.

[41] "Cloud." Open clip art https://openclipart.org/detail/193560/cloud, 2014.

[42] "Server." Open clip art https://openclipart.org/detail/216936/dell-2u-server, 2015.