The BRISC-V Platform: A Practical Teaching Approach for Computer Architecture

Rashmi Agrawal Sahan Bandara Alan Ehret Mihailo Isakov Miguel Mark
Michel A. Kinsy*
{rashmi23,sahanb,ehretaj,mihailo,mmark9,mkinsy}@bu.edu
Adaptive and Secure Computing Systems (ASCS) Laboratory
Boston University
Boston, Massachusetts

ABSTRACT

Computer architecture lies at the intersection of electrical engineering, digital design, compiler design, programming language theory and high-performance computing. It is considered a foundational segment of an electrical and computer engineering education. RISC-V is a new and open ISA that is gaining significant traction in academia. Despite it being used extensively in research, more RISC-V-based tools need to be developed in order for RISC-V to gain greater adoption in computer organization and computer architecture classes. To that end, we present the BRISC-V Platform, a design space exploration tool which offers: (1) a web-based RISC-V simulator, which compiles C and executes assembly within the browser, and (2) a web-based generator of fully-synthesizable, highly-modular and parametrizable hardware systems with support for different types of cores, caches, and network-on-chip topologies. We illustrate how we use these tools in teaching computer organization and computer architecture classes, and describe the structure of these classes.

CCS CONCEPTS

• **Applied computing** → *Computer-assisted instruction*; • **Computer systems organization** → *Multicore architectures*; *Interconnection architectures*;

KEYWORDS

computer architecture, computer organization, risc-v, simulator, Verilog, generator

ACM Reference Format:
Rashmi Agrawal Sahan Bandara Alan Ehret Mihailo Isakov Miguel Mark and Michel A. Kinsy. 2019. The BRISC-V Platform: A Practical Teaching Approach for Computer Architecture. In *Workshop on Computer Architecture Education (WCAE'19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3338698.3338891

*All authors contributed equally.

WCAE'19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6842-1/19/06…$15.00
https://doi.org/10.1145/3338698.3338891

1 INTRODUCTION

Courses such as Computer Organization and Computer Architecture are fundamental for a computer engineering student. Knowledge in these subjects can help students gain a deeper understanding of the concepts in other courses including operating systems, algorithms, programming and many more. But the steep learning curve and large time investment associated with hardware design and development limits the scope and depth of hands-on laboratory exercises of these courses. We believe that the best way to maximize student engagement and educational value of Computer Organization and Computer Architecture classes is via a practical and hands-on approach. The BRISC-V Platform delivers this practical teaching approach for computer architecture by providing an open-source single and multi-core design space exploration platform that eliminates much of the overhead associated with developing a complete processing system.

The BRISC-V Design Space Exploration Platform [1] provides many opportunities for a hands on computer architecture education. The platform consists of (1) a RISC-V simulator to test software independently of any hardware system, (2) a RISC-V toolchain to compile a user's code for bare-metal execution, (3) a modular, parameterized, synthesizable multi-core RISC-V hardware system written in Verilog, and (4) a hardware system configuration Graphical User Interface (GUI) to visualize and generate single or multi-core hardware systems.

In this paper, we describe how the BRISC-V Design Space Exploration Platform can be used to teach an undergraduate level Computer Organization class, a graduate level Computer Architecture class, and a research focused graduate level Hardware Systems Security class. Programming and Assembly labs are supported with a browser based tool named the BRISC-V Simulator for writing, compiling, assembling and executing RISC-V code. Students can use the platform-independent simulator to get started with RISC-V software quickly and easily. The BRISC-V Simulator provides a valuable resource when teaching students about low level concepts, including calling conventions, memory allocation, and the compilation flow.

The resources provided by the BRISC-V Platform streamline RTL based laboratory exercises by providing functional and tested starting points for hardware systems. The modular and parameterized RISC-V hardware system included with the BRISC-V Platform provides a useful template for students building their first processor. The modular nature of the hardware system allows assignments

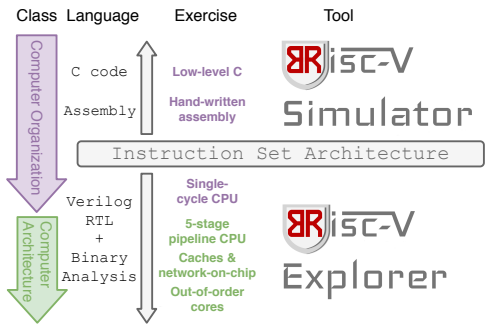

Figure 1: Overview of the Computer Organization and Computer Architecture courses.

to be crafted for each stage of the processor: fetch, decode, execute, memory and write-back. More experienced students in a Computer Architecture class can use the hardware system configuration GUI named BRISC-V Explorer to configure baseline systems. Then, students can add micro-architectural features to their highly modular configured hardware system. For example, students can configure a single cycle, single core processor to add pipeline registers to it. Students looking to experiment with more advanced micro-architectural features can configure a complex, multi-core processor. The cache architecture is a more complex feature and we have designed a few assignments around it. Selecting a single or multi-level cache in The BRISC-V Explorer enables students to experiment with different cache size, associativity, replacement policy or custom cache architectures.

The complete BRISC-V Design Space Exploration Platform (including Verilog source code) is open-source and can be downloaded at https://ascslab.org/research/briscv/index.html.

2 COURSE OVERVIEW

In this section, we illustrate the organization of the (1) Computer Organization and (2) Computer Architecture courses.

The Computer Organization class aims to familiarize students with low-level coding in C and assembly, and provide a high-level view of a processor. The students have no prerequisite programming skills, and are expected to learn the basics of the C programming language and RISC-V assembly. Students will build on their experience with Verilog from the prerequisite Digital Design course.

They are expected to complete several exercises in C, writing simple programs using functions, recursions, floating point and bitwise operations. These exercises also explore how each datatype they use is actually stored in memory. Next, they analyze how these C programs are compiled to assembly, and learn how to write their own functions and recursions. Finally, as a class project, students are tasked with writing a single-cycle CPU supporting the RV32I instruction set. For both the C and assembly labs, the students make heavy use of the BRISC-V Simulator, presented in Section 3. The simulator allows them to compile C to RISC-V assembly, hand-write assembly, test it, debug it, and view the state of the registers and memory at every instruction of the program. For the single-cycle CPU design project, the teaching assistants use the BRISC-V Explorer (presented in Section 4) to quickly generate a bare-bone single-cycle CPU, and remove all functionality from it while leaving the modules as a project skeleton.

By the end of the course, the students will have a demystified view of hardware, as they have both programmed a bare-metal CPU and created a simple but fully-synthesizable processor. Even if this is the last hardware course the student may take, they will have a solid footing when exploring topics such as writing high-performance code, using optimizing compilers, or diving deeper into computer architecture.

The second course we describe is Computer Architecture, with Computer Organization as a prerequisite. In this course, students will gain an understanding of a "modern" processor, as concepts such as pipelining, caching, inter-core communication, multiple-issue and out-of-order processors are introduced. The students are expected to be familiar with bare-metal C code, assembly, and Verilog. In order to save time, the students are given a fully-functional and tested single-cycle processor, along with testbenches and assembly programs. The labs are structured in such a way that students can run code on their processors in the very first lab, and the labs only explore modifications to this processor. They are expected to complete several labs focused on the hardware implementation of (1) a 7-stage pipelined processor based on the single-cycle processor, (2) a simple L1 cache and an optimized cache hierarchy, (3) a multi-core processor and (4) an advanced micro-architectural feature covered in the class lectures. In the fourth laboratory exercise, students explore the effectiveness of hardware modifications such as multiple-issue processors by analyzing software binaries using a binary analysis tool such as Intel PIN [4].

In this course, the students use the BRISC-V Simulator to write bare-metal code which can run on students' processors. In the lab exploring caches, students are asked to use the BRISC-V Explorer to find a cache configuration that provides the highest performance on a given task. By the end of the course, students should have a good grasp of major concepts in computer architecture. While this is an architecture class, the students should also walk away with actionable knowledge in writing software, and be able to answer questions such as "why can two algorithms with the same computational complexity have an order-of-magnitude difference in performance", or "how might the on-chip network topology affect the performance of multi-threaded algorithms".

3 THE BRISC-V SIMULATOR

The BRISC-V Simulator is a RISC-V simulator targeting the RV32I feature set. It is a single-page web application written in JavaScript, that allows the user to (1) compile C to RISC-V assembly, (2) run or step through assembly, and (3) analyze the state of the processor at every instruction. A web-based implementation provides users with flexibility in terms of running the simulator. There are three ways users may run this application: the user may run the application locally, which requires no installation, but does not come with a compiler, (2) a student or teaching assistant may host the website, which requires installing several python libraries, and (3) the user may access the public version of the simulator from our website which is located at https://ascslab.org/research/briscv/simulator/simulator.html. The simulator executes the code on the client machine. The only computation that happens on the server hosting the simulator is (1) distributing the static website, and (2) the optional compiler support, which allows users to compile their C code from the browser. In large classes, this is advantageous because student machines often may not have privileges to install and run the compiler.

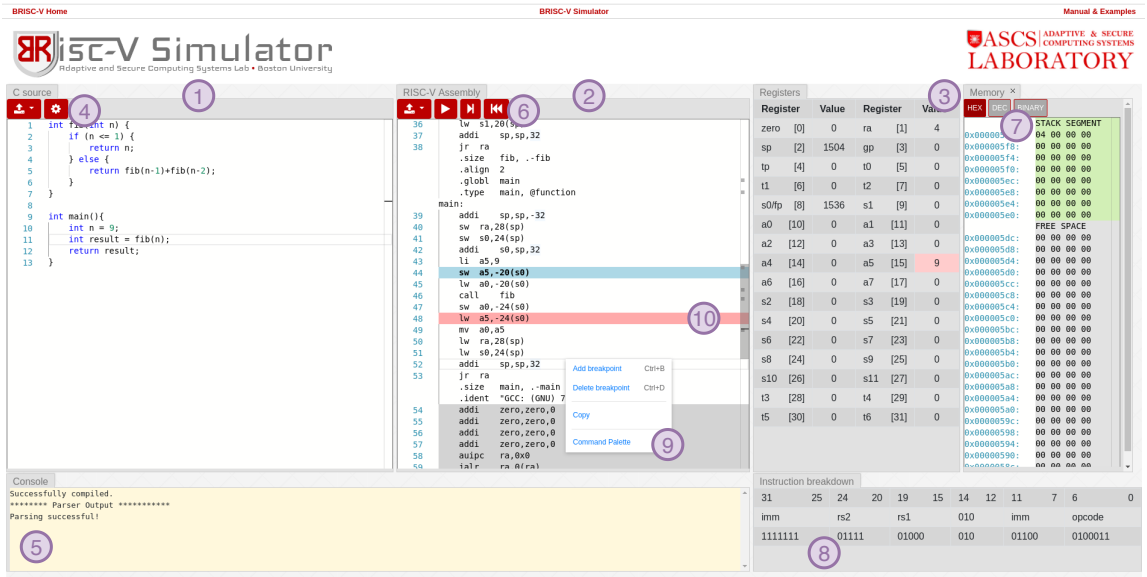

Figure 2: A screenshot of the BRISC-V simulator.

3.1 Using The BRISC-V Simulator

Figure 2 shows a screen-shot of the simulator GUI. The page is split into three main columns: the compiler pane (1) on the left, the assembly pane (2) in the center, and the register and memory pane (3) on the right.

Compilation: in the compiler pane, the user is free to write C code. The user can load C code from the filesystem and compile it using the compiler pane buttons (4). The compiled assembly will be written to the central assembly pane (2), and any standard output from the compiler will be written to the console (5).

Executing assembly: to add the RISC-V assembly to the assembly pane, the user has three options: compile C code, load assembly code from the file system, or load example code. Above the assembly pane are four buttons (6): *load assembly*, *run*, *single instruction step*, and *reset* simulator. The *load* button allows the user to either load their own assembly, or select one of the three example files provided. After loading the assembly program, the simulator wraps the assembly with our *kernel code*, which consists of both a program prologue and epilogue. The program prologue is tasked with initializing the registers, as well the stack pointer, and the program epilogue traps the simulator in an infinite loop upon program termination. The kernel code has a grey background in the assembly pane to distinguish it from the user code. If the user chooses to load their own assembly, any errors and the result of the parser will be shown in the console (5). A successful message is presented to the user if no errors are found in the assembly program. However, if at least one error is found, the console window will enumerate each error and its corresponding line number; the last line of the error output will display the total number of errors found. Simulation is also disabled and the simulation buttons will be greyed out as a visual cue. The *run* button will run the code until the code (1) hits an exit syscall, (2) hits the kernel while loop, (3) hits a breakpoint, or (4) gets trapped in an infinite loop for more than 100000 instructions. The *single instruction step* button runs only a single instruction, and ignores any breakpoints. The *reset* button moves the instruction pointer back to the start of the program, and resets the state of the registers and memory.

Observing the state: after every single instruction, the user can monitor the state of the registers and memory in the register and memory panes (3). If an instruction updates the value of a register, that register will be highlighted in red, as shown in Figure 2. To the right of the register pane is the memory pane. It is represented as a descending list with five foldable regions: the stack segment, free segment, heap segment, data segment, and the text segment. Each line of the memory pane represents one word (32 bits). The address is shown on the left in hexadecimal format (light blue), and the value is shown on the right in either hexadecimal, decimal, or binary. The format buttons (7) allow the user to show the memory in their preferred format. Additionally, the breakdown pane (8) lets the user to view the breakdown of the instruction at the current IP.

Additional Features: the user may right-click on an instruction, which will open a context menu (9). If the user selects the "Add breakpoint" option, a pink line will cover that instruction (10). If the user moves the mouse above any labels, they will see an option to "fold" that region of code. Also, all of the panes are resizable and movable - the user can hide e.g., the memory or the compilation pane, or may stack the console and the instruction breakdown.

3.2 System Call Support

The BRISC-V Simulator implements support for seven system calls, as shown in Table 1. To call a system call (syscall), the program

Syscall Type	`t0`	Description
Print Integer	1	Prints integer stored in `a0`
Print Character	2	Prints ASCII character stored in `a0`
Print String	3	Prints string stored at address in `a0` with length stored in `a1`
Read Integer	4	Reads integer and stores it in `a0`
Read Character	5	Reads an ASCII character and stores it in `a0`
Read String	6	Reads a null-terminated string and stores it at address in `a0` with length in `a1`
Stop Execution	7	Stops the program

Table 1: System calls supported by the BRISC-V Simulator.

needs to set the appropriate syscall ID value in the `t0` register. Next, any parameters to the syscall should be placed in registers `a0` and `a1`. Finally, the program runs the `SCALL` or `ECALL` pseudo-instruction. If the system call returns any value, it will be stored in registers `a0` and `a1`. One of the example assembly files (`syscalls.s`) provided by the simulator illustrates how the system calls are used.

3.3 BRISC-V Simulator Extensibility

Through the BRISC-V Simulator, we aim to provide students with a simple 'hackable' tool that they can use in computer organization classes to gain familiarity in assembly and to confirm that their compiled code behaves as expected. In computer architecture classes, the simulator can be used to explore existing ISA extensions (i.e. floating point or vector instructions). Likewise, in a hardware security class, new security specific ISA extensions can be quickly added and tested.

As an example, to implement the multiplication operations `MUL`, `MULH`, `MULHU`, `MULHSU` defined in the RISC-V specification [5], one needs to:

(1) Edit the `parser.js` file of the BRISC-V Simulator so that these instructions are parsed with the two source registers and a destination register. This requires minimal coding as the registers are already extracted by the parser.
(2) Add a new case condition in the `emulator.js` file so that `MUL*` instructions (1) update the IP by 4, and (2) store the correct multiplication result in the correct register. The emulator already has the registers as local variables, so the user just needs to refer to the appropriate ones.

4 THE BRISC-V EXPLORER

The BRISC-V Design Space Exploration Platform provides a suite of tools to quickly develop single and multi-core RISC-V processors. The BRISC-V Explorer GUI is used to configure the hardware components of a user's system. Users can select core, cache, main memory and NoC configuration parameters. Figures 3 and 8 show two views of the BRISC-V Explorer. The Explorer contains several panes with different configuration settings and other information. Figure 3 shows the core configuration settings pane ① with the downloads pane ②, console pane ③ and block diagram of the entire configured hardware system ④. In the core configuration settings pane, users can select between single-cycle, five stage pipelined and seven stage pipelined cores. Pipelined cores can be configured with or without data forwarding logic. The number of cores in the system is can also be selected by the user, and ranges from 1 to 8. The console pane outputs information about invalid configurations and the status of exporting a project. The block diagram pane shows the hardware system that will be exported by clicking download in the download pane. Figure 8 shows the core configuration settings pane, memory hierarchy configuration settings pane and block diagram of the memory hierarchy. Users can select various cache configuration settings in the memory hierarchy pane. Tunable settings include cache associativity, line size, number of lines and depth of the cache hierarchy. Panes in the BRISC-V Explorer GUI can be moved and resized to easily configure settings relevant to the current user.

After a configuration has been chosen, clicking the "Download Project" button in the downloads pane will output a highly modular Verilog implementation of the system. These advanced multi-core configuration features provide a rich design space for advanced computer architecture classes to explore. Students in these classes can configure a single or multi-core system and add features such as branch prediction, out-of-order execution or more experimental features tied to graduate student research. For students in their first computer organization class, the BRISC-V Explorer can generate a single cycle RV32I core without caches, utilizing a simple dual ported main memory. This simple processor serves as a template for exercises in which students are asked to build their first processor.

5 COMPUTER ORGANIZATION CLASS

In this section we give an overview of the Computer Organization class, how the BRISC-V Simulator (Section 3) tool we developed is used in it, and how the BRISC-V Explorer (Section 4) is used for generating a template for a single-cycle processor.

As seen in Figure 1, the Computer Organization class covers concepts both 'above' and 'below' the instruction set architecture (ISA). Each class has seven laboratory exercises with an eighth exercise serving as the final project. Two of the exercises are dedicated to C, two to assembly, and the rest are dedicated to implementing a single-cycle processor in Verilog. This class project is further explained in Section 5.3.

5.1 C/C++ Exercises

The goal of the C/C++ exercises is to provide a smooth transition into writing RISC-V assembly for students who have no prior experience with assembly language programming. Additionally, students can often be confused by concepts such as pointers or interfacing with the operating system. By providing a machine-level view of these concepts, the C/C++ exercises should clear up any misconceptions they may have.

Exercise 1: In the first exercise, the students should become familiar with common formats such as signed and unsigned integers, conversion between binary, decimal, and hexadecimal formats, using arrays and pointers, and writing recursive functions. For extra points, students are asked to convert a floating point value to its binary representation (without simply using C's union data type), and the reverse. To complete this exercise, the students can either write code in an editor, and compile and run it from the command line, or they can use the built-in compiler in the BRISC-V Simulator.

Exercise 2: Since the system the students will create can only run bare-metal code, and has no kernel running on top of it, dynamically allocating memory is not possible out-of-the-box. The goal of this exercise is to demystify the workings of heap memory and modern malloc implementations. In this exercise, the students are asked to write a simple library for dynamic memory allocation. The library statically allocates some amount of memory at the start of the program, and provides functions `malloc`, which takes a size in bytes and returns a pointer to the first contiguous piece of free memory, as well as `free`, which takes a pointer to the previously allocated block of memory and frees it in the library's internal data structures.

5.2 Assembly exercises

In the assembly exercises, the students gain an in-depth understanding of the RISC-V ISA and the inner workings of a processor. At the time of the creation of these exercises, no suitable tool was

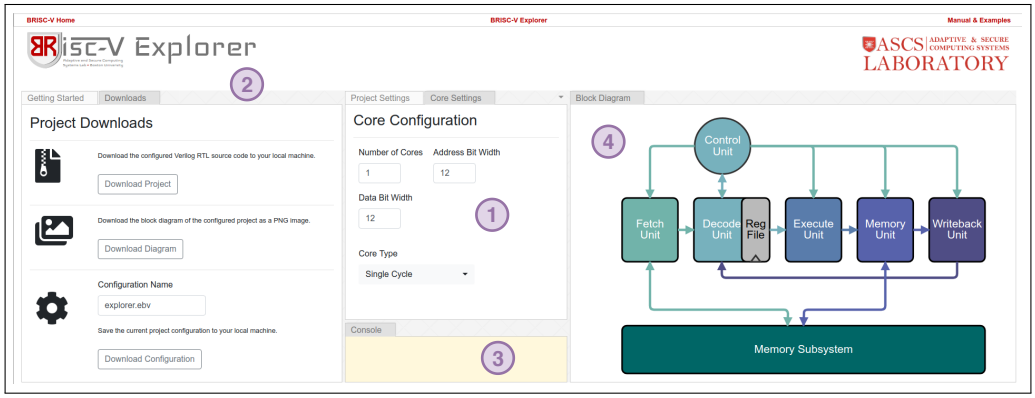

Figure 3: The BRISC-V Explorer GUI with the single cycle core configuration used as a baseline in the pipelining exercise.

available that would allow students to execute RISC-V assembly instruction-by-instruction and monitor the state of the registers and memory. Thus, we have created the BRISC-V Simulator (Section 3). For exercises 3 and 4, the students are supposed to load their assembly programs into the simulator and execute them to confirm their correctness or to find bugs. The simulator provides them with the needed tools: the registers and memory are visible right next to the assembly instructions, and as instructions are executed, the updated values are highlighted. Additionally, the simulator allows setting breakpoints, simplifying debugging.

Exercise 3: In this exercise, the students should explore how C code is compiled to assembly, how functions and the stack work, and become familiar with the RISC-V calling convention. To become familiar without having to write assembly right away, we first task students with analyzing how C code is converted to assembly. Here the students are provided with a C program, and are required to (1) compile it in the BRISC-V Simulator's compiler, and match each line of the C with a sequence of instructions in the assembler. For the second task, we explain the limitations of the RV32I ISA, namely the lack of a multiply instruction. The students are tasked with writing a multiply label in assembly and using it to perform simple calculations. Finally, we expose the students to the RISC-V calling convention, and task them with writing a non-recursive factorial function. This exposes students to basic stack concepts, and how parameters are passed and returned to and from functions.

Exercise 4: By this exercise, the students should be familiar with the majority of RISC-V instructions and concepts, and are ready to write more complex programs. First, the students are tasked with writing both non-recursive and recursive versions of a Fibonacci function. Next, they are asked to write a matrix multiplication function, so that they will have to write nested loops. Finally, we expose them to a set of simple system calls built into the BRISC-V Simulator. These system calls allow the students to read and write integers, characters and strings to and from the console, as well as end the program. A list of all system calls can be found in Table 1. This exercise concludes the assembly exercises of the class, and the students can then dig down into the processor microarchitecture.

5.3 Class Project: Building Your First Processor

The hands on experience of building a processor is an indispensable part of any computer organization class. After students have been introduced to the RISC-V ISA, assembly programming, and other fundamental computer organization concepts they can begin to implement their own CPU. A simple single cycle core generated with the BRISC-V Explorer serves as a template for the computer organization class exercises. The modular nature of the single cycle core allows it to be broken up into discrete exercises, guiding students through the process of building a CPU.

The single-cycle core contains separate modules for the fetch, decode, execute, memory and write-back stages. In each exercise, students will build a new stage of the processor. Students are given an interface specification for each module and an empty module template with a port list. Breaking each assignment into discrete modules helps reinforce modular design practices necessary for the complex RTL designs. Providing the template and port list also ensures the modules can be graded with a single test bench. Figure 4 shows how the single cycle core is broken up into modules and how individual exercises are created.

Exercise 5: In this exercise, the students implement an arithemetic logic unit (ALU) for the RISC-V RV32I instruction set. The ISA does not explicitly give an encoding for the ALU control signals, so these are given to students in the exercise description. To keep the first few exercises manageable, students are asked to implement only a subset of the RV32I instructions. Students start by implementing the simple arithmetic and bit-wise logic instructions. Incorporating the remaining instructions is left for future exercises when students have a better understanding of their processor design.

Exercise 6: This exercise covers the decode stage of the core. Students build the decode and control logic, as well as the register file.

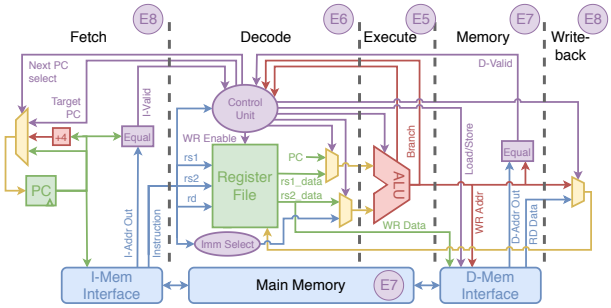

Figure 4: Block diagram of the single cycle core. Dotted lines show the boundaries between modules. Labels E5 (Exercise 5) to E8 (Exercise 8) highlight the exercise associated with the design of each module.

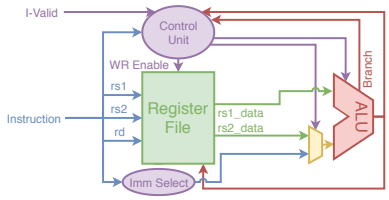

Figure 5: Block diagram of the decode logic, register file and ALU integration completed in exercise 6.

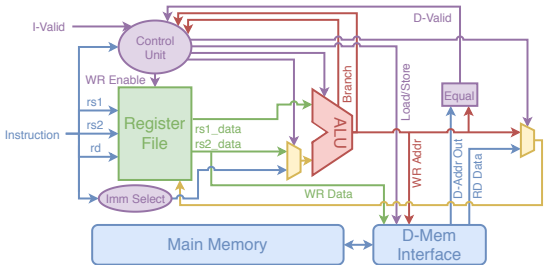

Figure 6: Exercise 7 block diagram with the memory and writeback stages integrated with the decode logic and ALU.

The students are only required to implement decode logic for the same subset of instructions used in the first exercise.

After the decode, register file and ALU modules are complete, students must demonstrate (in simulation) the execution of a simple instruction stream, feeding instructions into the decode module. The decoded instructions are sent to the ALU which computes a result. The ALU output is directly written back to the register file. Figure 5 shows the block diagram used in this exercise.

Exercise 7: In this exercise, students create the main memory module and the main memory stage of the core. Students must add load and store instructions to their decode module. To support the new load and store instructions, students must add the writeback stage to the processor as well. The writeback stage consists of a single multiplexer to select between the data memory output and the ALU output. The selected value is sent to the register file.

The new main memory, memory stage and writeback stage modules must be added to the previous demonstration, so students can demonstrate the execution of a more complex instruction stream. Figure 6 shows a block diagram with the additional modules used for this exercise.

Exercise 8: The fetch stage of the single cycle core is relatively simple and does not warrant an entire exercise on its own. Instead, the fetch stage development is combined with the final integration stage of the processor design in the eighth exercise. In addition to completing the fetch module, students add the remaining RV32I instructions not implemented in the previous assignments and verify that their processor can correctly execute a program. This exercise includes the addition of branch and jump instructions to the processor. This integration stage is given as a final project for the computer organization class instead of as a shorter laboratory exercise.

Students are expected to read and understand the relevant sections of the RISC-V Instruction Set Manual [5] to correctly implement instructions including AUIPC (add upper immediate to PC), BEQ (branch if equal) and JALR (jump and link register). These instructions take inputs from or output to the fetch stage, making

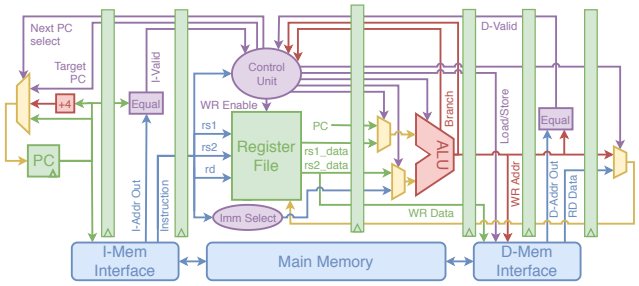

Figure 7: Block diagram of the seven stage pipeline core.

it easier to incorporate them after students have a semi-complete processor.

Students are encouraged to build the fetch and write-back stages first and integrate the modules for their subset of instructions. Then they can expand their existing modules to include the remaining instructions, testing their additions along the way. Although students performed some integration tests in the previous exercises, inevitably more issues will be discovered as instructions are added and more complete testing is performed. Students are given simple test programs to demonstrate on their processor and must develop additional test programs to provide fuller test coverage. This final integration exercise forces students to devote attention to the correctness of the processor as a whole. Any issues found during integration highlight the challenges of integrating modules in a large project and the need for thorough tests for any RTL design.

Students demonstrate the correct execution of their custom test programs and the provided ones in a simulation. After completing the final exercise, students have completed a functional RISC-V processor compatible with the RISC-V toolchain included with the BRISC-V Design Space Exploration Platform. Optionally, more ambitious students may want to synthesize their design. As the BRISC-V template has strongly encouraged the use of structural Verilog and has provided the students with asynchronous BRAM used for storing instructions and data, we have witnessed few challenges to running the final designs on an FPGA.

6 COMPUTER ARCHITECTURE CLASS

The Computer Architecture class builds on the Computer Organization class to introduce more advanced micro-architecture features, including pipelining, branch prediction, and out-of-order execution. This class is designed for undergraduate seniors and graduate students and consists of four in-depth exercises. In the exercises, students will (1) pipeline a processor, (2) build a cache and optimize a cache hierarchy, (3) develop a multi-core processor and (4) implement an advanced micro-architectural feature covered in the class lectures.

6.1 Pipelining a Processor

The first micro-architecture feature covered in this Computer Architecture course is pipelining. To gain first hand experience with processor pipeline logic, students implement a seven stage pipelined RISC-V processor. Students use the BRISC-V Explorer to configure a baseline single cycle processor with a simple synchronous memory. Figure 3 shows the single cycle processor configuration in the BRISC-V Explorer.

The synchronous memory used in the processor registers its read port and can be implemented with FPGA BRAM, making larger on-chip main memories practical for synthesized designs. Five of the processor's seven pipeline stages are placed between each of the fetch, decode, execute, memory, and write-back modules. The last two pipeline registers placed in between the operation issue and receive sides of the fetch and memory stages. Adding a register between the issue side of the memory or instruction fetch interface and the receive side of the interface prevents the need to insert a pipeline bubble while the synchronous memory spends a cycle completing a memory read. A diagram of the pipelined processor is shown in Figure 7. In addition to inserting pipeline registers, students must add the necessary control signals to detect hazards and insert bubbles as needed. Control signals are derived based on in-class examples of hazard resolution logic.

6.2 Memory Organization

Caches play a vital role in most modern processors. Cache hierarchies with multiple levels are used to overcome the "memory wall". Students in a Computer Architecture class can benefit from designing a processor with caches in the BRISC-V Explorer and analyzing the impact of cache configuration on the performance of a processor.

Part 1: Students are required to build a simple direct mapped cache. While cache size is specified, student are allowed to pick a line size and set count of their choice. Students are encouraged to vary these values and observe the changes in resource usage with a synthesis tool of their choice. The implemented cache should follow the memory interface of the seven stage pipelined RISC-V core implemented using the BRISC-V explorer.

Part 2: Students use the BRISC-V explorer to implement a complete cache hierarchy. The students are provided with binaries for two benchmark programs. They are required to optimize the cache structure to provide the best performance for the benchmark programs. THE First phase of the lab is implementing two different cache hierarchies that will be optimal for each of the programs.

The BRISC-V platform includes blocking caches that implement the write back with write allocate policy. The primary caches have one-cycle pipelined access. Secondary caches are based on a configurable cache module that can be used at level 2 or 3 of the hierarchy. The modular design allows most of these properties to be easily modified. The BRISC-V explorer allows the students to vary a multitude of cache hierarchy parameters in their optimization efforts. Some of these parameters are: (i) number of levels in the cache hierarchy, (ii) size of each cache, (iii) associativity, set count, and line width for each cache, and (iv) replacement policy for each cache. Students are allowed to configure any of the tunable parameters as long as the total resource usage for the cache hierarchy remains below a specified threshold.

Next, the students are required to implement a cache hierarchy that is optimized for both the benchmark programs. This lab is graded based on the level of performance achieved by the individual implementations.

6.3 Multi-core Architecture

The next step in advanced features labs is implementing a multi-core processor.

Part 1: Students use the BRISC-V explorer to implement a dual-core processor. A shared bus based cache hierarchy is used for the dual-core implementation. Students are then required to write a simple program with two threads to be executed on the two cores. Students are allowed to pick a program from a list of programs that include integer search, FFT, counting prime numbers in a given range, and matrix multiplication. Implementing a multi-core program in bare-metal code will give the students an opportunity to appreciate the complexities of implementing parallel programming libraries such as OpenMP and Open MPI.

Part 2: Students implement quad and octa-core processors using the BRISC-V explorer. For this step, a Network-on-Chip (NoC) based architecture is used 10. The NoC router is also fully parameterized. In other words, the number of ports and the number of virtual channels (VCs) per port can be modified, different arbitration schemes, VC allocations, and routing algorithms can be implemented. This gives the students an opportunity to familiarize themselves with different parameters and implementation details of an on-chip network, which is an essential part of current multi-core processors. Next, the students are required to modify the program from the previous step to utilize four or eight cores according to the processor configuration. The students are also required to observe and compare the performance variations with the number of cores.

6.4 Other Advanced Features

For the final project of the Computer Architecture class, students implement an advanced micro-architecture feature covered in class lectures. Students implement the advanced features using the single-core seven stage pipelined processor as a baseline system. Students are free to choose if they want to use a cache hierarchy or not. The advanced micro-architecture feature is selected from a list including branch prediction, vector instructions, very long instruction word

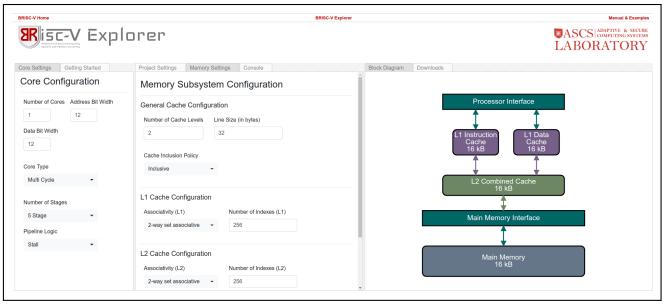

Figure 8: The BRISC-V Explorer GUI with a sample cache configuration.

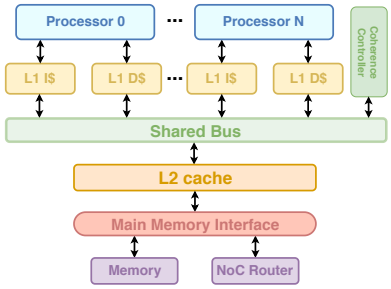

Figure 9: An example multi-core architecture.

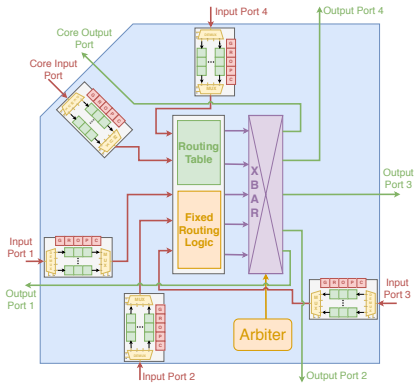

Figure 10: NoC router architecture.

(VLIW), hardware multi-threading, floating-point unit, out-of-order execution, and speculative execution, among others.

Part 1: Students must instrument an example program with Intel PIN [4] to predict performance improvements and justify their design choices. Instrumented code is executed on lab machines before RTL development of an architectural feature. To support design decisions, students develop software models of their feature and estimate the performance of the RISC-V architecture with and without their new feature.

Part 2: Students develop the RTL implementation of their micro-architectural feature and incorporate it into the baseline seven stage processor. Students analyze the impact their implemented feature has on the performance of an executed program of their choice. Program runtime is measured with the cycle count control status register included in the RV32I ISA. Students compare their predicted improvements with their actual improvements.

7 DISCUSSION

Computer Organization: Feedback from students in our computer organization class has been positive. Many students have expressed their excitement about completing their first processor design. After completing the computer organization class, several students have continued to work with the BRISC-V Design Space Exploration Platform, contributing to the base hardware system and adding features to the Explorer GUI. Although the single cycle processor students build in the computer organization class is simple, it is compatible with the RISC-V toolchian included in the BRISC-V Platform. With a complete software tool-chain, students can write custom C code for their processor and incorporate it in future projects as a soft core in a more complex design.

Computer Architecture: By the time students have completed both our Computer Organization and Computer Architecture classes, they have an in-depth knowledge of the BRISC-V hardware system. This detailed knowledge of the existing code-base makes further study focused on experimental architecture features easier. A firm grasp of the inner workings of the hardware system allows students to quickly add custom features.

Using the BRISC-V Explorer to implement the baseline system allows the students to focus on the advanced architecture features. The BRISC-V platform also includes the RTL implementations of several advanced architecture features such as Network-on-Chip and caches. Students have the opportunity to look at concrete implementations of these advanced features as opposed to learning about a feature from lecture notes or a textbook. This provides the students with deeper insight regarding certain tradeoffs involved in a real implementation.

BRISC-V platform in other courses: The use of the BRISC-V platform goes beyond the canonical computer architecture courses. It can be used or adopted for any course where the students require an RTL code base of a processor to be used as the starting point of laboratory exercises of class projects. Our graduate level Hardware and Systems Security class is one such course. It focuses on in-depth analysis of hardware security's role in cybersecurity, and the computer hardware related attacks and defenses in computing systems. Students have been able to use the BRISC-V Platform to further their research while working on the class project, which requires the students to implement a security feature on a baseline hardware system configured with the BRISC-V Explorer. The BRISC-V platform allows the students to quickly implement a working processor and focus on implementing their security extension.

One successful project implemented hardware multi-threading (HMT) on the seven stage core available in the BRISC-V Explorer. A cache hierarchy was connected to the HMT core and a cache side channel was demonstrated. This side-channel demonstration served as the baseline for future research in adaptive cache architectures to mitigate such side-channels. Another project implemented a multi-core processor with a hardware-isolated core, which provides secure execution capability. Other successful projects have developed micro-architectural support for control flow obfuscation [2] and hardware based Return-Oriented-Programming mitigation techniques [3]. These projects were presented at the 2019 Boston Area Architecture Workshop.

8 CONCLUSION

In this work, we introduce the RISC-V-based BRISC-V Simulator and BRISC-V Explorer. The web-based BRISC-V Simulator enables students and teachers to quickly write software for RISC-V systems. The BRISC-V Emulator allows users to rapidly design a single- or multi-core RISC-V RTL processor to go with their software.

Together, these tools and the rest of the BRISC-V platform provide a wealth of RISC-V based resources for computer architecture education, streamlining software and hardware based laboratory exercises. Our experience with the BRISC-V Platform has been positive and our students have appreciated the support provided by a complete platform with the hardware system, compiler toolchain, software simulator and hardware configuration GUI.

REFERENCES

[1] Sahan Bandara, Alan Ehret, Donato Kava, and Michel Kinsy. 2019. BRISC-V: An Open-Source Architecture Design Space Exploration Toolbox. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. ACM, New York, NY, USA, 306–306. https://doi.org/10.1145/3289602.3293991

[2] N. Boskov, M Isakov, and M. A. Kinsy. 2019. CodeTrolley: Hardware-Assisted Control Flow Obfuscation. *Boston Area Architecture 2019 Workshop (BARC19)* (Jan. 2019). arXiv:1903.00841

[3] Michael Graziano, Miguel Mark, Stefan Gvozdenovic, and Michel A. Kinsy. 2019. Hardware Assisted Transparent ROP Mitigation for RISC-V. *Boston Area Architecture 2019 Workshop (BARC19)* (Jan. 2019).

[4] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[5] A Waterman and K Asanovic. 2017. The RISC-V Instruction Set Manual-Volume I: User-Level ISA-Document Version 2.2. *RISC-V Foundation (May 2017)* (2017).