The Davis In-Order (DINO) CPU

A Teaching-focused RISC-V CPU Design

Jason Lowe-Power
University of California, Davis
jlowepower@ucdavis.edu

Christopher Nitta
University of California, Davis
cjnitta@ucdavis.edu

ABSTRACT

The DINO CPU is an open source teaching-focused RISC-V CPU design available on GitHub (https://github.com/jlpteaching/dinocpu). We have used the DINO CPU in the computer architecture course at UC Davis for two quarters with two separate instructors. In this paper, we present details of the DINO CPU, the tools included with the DINO CPU, and our experiences using the DINO CPU.

ACM Reference Format:
Jason Lowe-Power and Christopher Nitta. 2019. The Davis In-Order (DINO) CPU: A Teaching-focused RISC-V CPU Design. In *Workshop on Computer Architecture Education (WCAE'19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3338698.3338892

1 INTRODUCTION

This paper introduces the UC Davis In-Order (DINO) CPU. We have designed the DINO CPU for use in our senior-level computer architecture course (ECS 154B) which covers performance modeling, pipeline design, and memory systems.

In our computer architecture course, we use a set of four to five assignments over a 10 week quarter that ask the students to implement an in order CPU pipeline. In the past, we asked the students to implement their hardware design in Logisim [4]. Some students have found Logisim frustrating and lacking in resources, with complaints such as "there isn't much documentation on Logisim online" and "I hate logisim with a passion." Additionally, Logisim limited the complexity of the design we could ask students to create (e.g., the maximum register size is 32 bits), and it was very difficult to test. Finally, it was nearly impossible to execute any real applications on students' design in Logisim which limited its use for comparing different architectural design decisions.

In order to move away from Logisim and enable more complex designs, we implemented a single cycle and a five stage pipelined CPU in a hardware description language (HDL). The instructors of the computer architecture courses at UC Davis have long desired to use a HDL instead of Logisim; however, tools that supported existing HDLs were significantly larger in size or supported many unneeded features. We chose to implement the DINO CPU in Chisel [3], a domain-specific language written in Scala. Using Chisel instead of Verilog or other low-level HDLs allowed the instructors to more quickly and easily implement tests, write simulators, and use autograders. Additionally, we designed the DINO CPU for a computer science class with the knowledge that the students are familiar with object oriented languages than HDLs.

The main features of DINO CPU are

- A single cycle processor design (Section 2.2)
- A five stage pipelined processor design with full forwarding and hazard detection (Section 2.3)
- Nearly full support for 32-bit RISC-V integer (`rv32i`) instruction set allowing for many C programs to be compiled and executed without modifications
- A suite of tests including unit tests for each component, instruction tests, and small benchmarks (Section 3.1)
- An RTL simulator interface for debugging (Section 3.2)

We have used the DINO CPU for two quarters at UC Davis in our Computer Architecture class (ECS 154B). So far, the students' feedback has been generally positive. From our feedback, we received comments such as "Very challenging but rewarding course. Please keep using Chisel in the future!" Additionally, after teaching this course for the first time, one student reported that they received an internship offer because they had Chisel experience.

So far, we have designed a set of four assignments based around the DINO CPU. In the first assignment, to expose the students to Chisel and hardware design languages, we ask the students to implement a simple ALU control unit and wire part of the CPU data path. In the second assignment, we have the students extend their code to a full single cycle RISC-V processor which successfully executes full RISC-V applications. The third, and most challenging, assignment has the students create a pipelined CPU with full forwarding and hazard detection. In the fourth assignment, the students add a branch predictor to the pipeline, implement two different kinds of branch predictors, and compare the performance of different processor designs.

For each of these assignments, we gave the students a *code template* to ensure there was a common framework for their implementations. By giving the students a template, we were also able to leverage autograder software since all student solutions used the same interfaces.

All assignments were tested on real RISC-V binaries created with an unmodified GCC toolchain. Most of these binaries were simple instructions tests with only one or a few instructions until the third assignment. In the third and fourth assignments, the students ran full workloads written in C on their CPU designs through our Scala-based simulator culminating in an assignment which compared and contrasted different hardware design decisions on cycle time, complexity, and performance.

The rest of this paper is organized as follows. First, in Section 2 we explain the goals of the DINO CPU, details of the DINO CPU's

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

ACM ISBN 978-1-4503-6842-1/19/06…$15.00
https://doi.org/10.1145/3338698.3338892

design, and our four assignments. Then, in Section 3 we describe the tools included with the DINO CPU to enable its use in the classroom. In Section 4 we detail our experience and the lessons learned while using the DINO CPU in the last two quarters at UC Davis. Finally, Section 5 concludes.

2 DINO CPU

2.1 Overall goals and non-goals

Our goal in creating the DINO CPU is to use the *code* to teach senior-level computer science undergraduates details of in-order processor design. We use the public version of DINO CPU as a basis for four to five CPU design assignments. Our main goals for DINO CPU are:

- Simple design and easy to teach
- Implement enough of the RISC-V ISA to compile and run real C programs
- Clean and understandable code

To make the design easy to teach, we decided to closely follow the design in the RISC-V edition of the Patterson and Hennessy Computer Organization and Design book [7]. This allowed us to refer to the book for the reasons behind design decisions. We did this with minimal modifications for both the single cycle and pipelined designs.

There are a number of common hardware design goals that we explicitly forego in the design of the DINO CPU. We did not try to design a high performance CPU in terms of low CPI or fast cycle time. Instead, we focused on readability and pedagogical design. We also have not investigated emulating the design on FPGA or implementing the design with EDA tools.

Finally, the DINO CPU does not implement a fully compatible RISC-V CPU design. Specifically, we do not support most of the privileged instructions, the CSR instructions, or `ecall` and `ebreak`. We would like to support these instructions and enable machine mode so we can run a more diverse set of workloads (e.g., workloads with system calls like `printf`). However, we will focus on adding this feature in a modular way which does not affect the overall complexity of the control or the data path.

Chisel [3] is an emerging hardware design language. It is a domain-specific language written in Scala. We considered using Verilog; however, we chose Chisel for three main reasons. First, it is a more familiar programming interface for our predominately computer science students allowing them to concentrate on the hardware design and not on syntax. Second, we found it straightforward to integrate Chisel with autograding software. Finally, Chisel is easier to parameterize for many different designs which allows us to use the same interfaces for multiple CPU designs and different design variations.

Chisel is becoming increasingly popular in industry. Google recently announced they designed the edge TPU in Chisel [6], the fabless semiconductor startup SiFive uses Chisel, and many other companies are investigating Chisel.

2.2 Single cycle CPU design

The single cycle CPU design in the DINO CPU closely follows the design in the RISC-V edition of the Patterson and Hennessy (P&H)

```
class SingleCycleCPU extends Module {
  val io = IO(new CoreIO())
  val pc         = RegInit(0.U)
  val control    = Module(new Control())
  val registers  = Module(new RegisterFile())
  val aluControl = Module(new ALUControl())
  val alu        = Module(new ALU())
  val immGen     = Module(new ImmediateGenerator())
  val branchCtrl = Module(new BranchControl())
  val pcPlusFour = Module(new Adder())
  val branchAdd  = Module(new Adder())

  io.imem.address := pc

  pcPlusFour.io.inputx := pc
  pcPlusFour.io.inputy := 4.U

  val instruction = io.imem.instruction
  immGen.io.instruction := instruction
  control.io.opcode := instruction(6,0)

  registers.io.readreg1 := instruction(19,15)
  registers.io.readreg2 := instruction(24,20)
  registers.io.writereg := instruction(11,7)
  registers.io.wen      := control.io.regwrite

  aluControl.io.add       := control.io.add
  aluControl.io.immediate := control.io.immediate
  aluControl.io.funct7    := instruction(31,25)
  aluControl.io.funct3    := instruction(14,12)

  when (control.io.alusrc1 === 0.U) {
    alu.io.inputx := registers.io.readdata1
  } .elsewhen (control.io.alusrc1 === 2.U) {
    alu.io.inputx := pc
  } .otherwise {
    alu.io.inputx := 0.U
  }
  alu.io.inputy := Mux(control.io.immediate,
    immGen.io.sextImm, registers.io.readdata2)
  alu.io.operation := aluControl.io.operation

  branchCtrl.io.branch := control.io.branch
  branchCtrl.io.funct3 := instruction(14,12)
  branchCtrl.io.inputx := registers.io.readdata1
  branchCtrl.io.inputy := registers.io.readdata2

  io.dmem.address   := alu.io.result
  io.dmem.writedata := registers.io.readdata2
  io.dmem.memread   := control.io.memread
  io.dmem.memwrite  := control.io.memwrite
  io.dmem.maskmode  := instruction(13,12)
  io.dmem.sext      := ~instruction(14)

  when (control.io.toreg === 1.U) {
    registers.io.writedata := io.dmem.readdata
  } .elsewhen (control.io.toreg === 2.U) {
    registers.io.writedata := pcPlusFour.io.result
  } .otherwise {
    registers.io.writedata := alu.io.result
  }

  branchAdd.io.inputx := pc
  branchAdd.io.inputy := immGen.io.sextImm
  when (branchCtrl.io.taken ||
        control.io.jump === 2.U) {
    pc := branchAdd.io.result
  } .elsewhen (control.io.jump === 3.U) {
    pc := alu.io.result & Cat(Fill(31, 1.U), 0.U)
  } .otherwise {
    pc := pcPlusFour.io.result
  }
}
```

Listing 1: Single cycle RISC-V CPU data path. Highlighted lines show restricted data path for R-type only instructions.

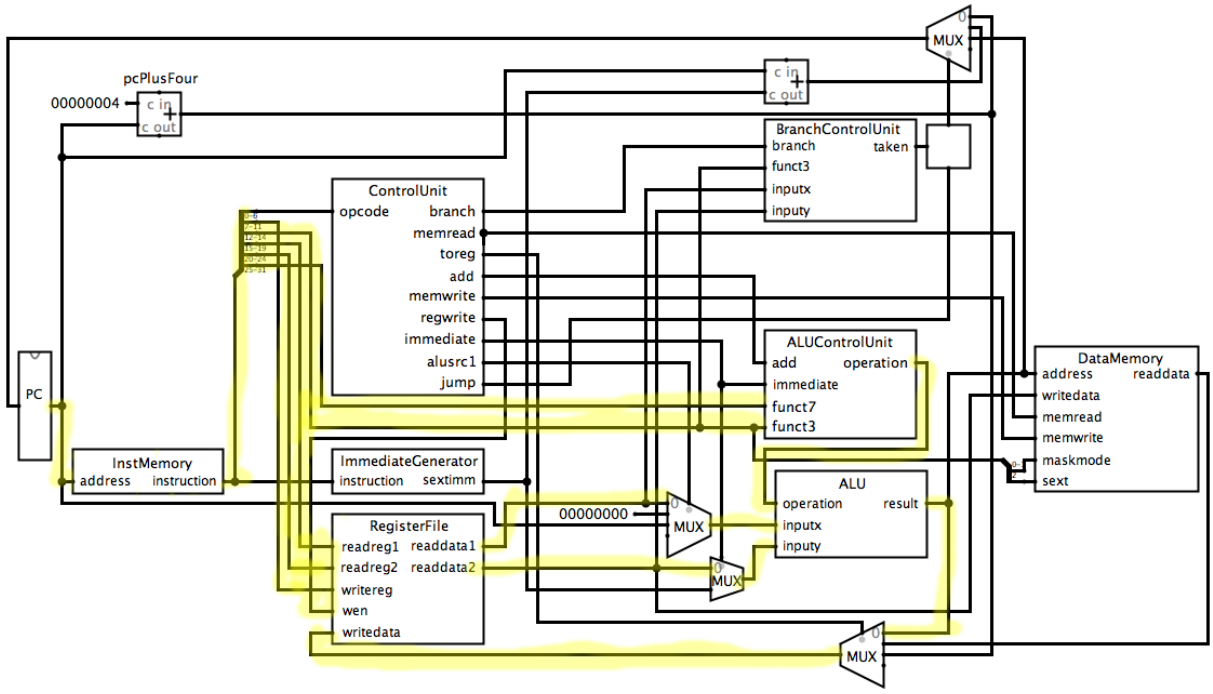

Figure 1: Single cycle RISC-V CPU diagram. Highlighted wires show restricted data path for R-type only instructions.

book [7]. There are two main differences between the DINO CPU design and the design detailed in the book. First, we rename some of the control signals. This is a small detail that does not affect the pedagogical design of the processor, but does decrease the prevalence and ease of cheating on future versions of this assignment. Second, since the DINO CPU implements all of the RISC-V instructions, some components require more hardware than presented in the original design. For example, the book only implements four of the ten R-type instructions.

In the largest deviation from the P&H design, we split the branch logic into its own unit (the branch control unit). However, this deviation is not a requirement. In our second offering of the class using the DINO CPU, we calculated the comparison flags in the ALU and passed them to the branch control unit instead of having the branch control unit do the calculations. This is an example of the flexibility of our design and an example of how minor modifications can enable high re-usability in the classroom of this design by discouraging copying code from previous offerings.

To simplify the code base, the DINO CPU makes heavy use of the modularity and encapsulation available in Chisel. Each of the modules in boxes in Figure 1 are their own Chisel class with a set interface (shown as the input/output in the figure). By using this modularity, the code for the data path shown in the figure is quite simple: only about 70 lines of code as shown in Listing 1. In the main CPU file, the students only need to instantiate all of the modules (we suggest giving them this in the template to ensure consistent naming), add any necessary multiplexers, and connect the wires.

Similarly, each of the modules can be tested on their own, and we provide the students with unit tests for each module. Thus, if the students pass the unit tests for each module and have wired the CPU correctly, then all of the applications will work correctly. In practice, we found that our initial set of unit tests did not have complete coverage, and we are currently working to improve this coverage. Improving the test coverage is discussed more in Section 4.2.

We used this single cycle design for the first and second assignments in our classes. The first assignment asks the students to create the `ALUControlUnit` logic to determine the ALU operation based on the only `funct7` and `funct3` bits of the instruction. Additionally, in the first assignment the students must wire the data path for R-type instructions (highlighted in Figure 1 and Listing 1). In this assignment, the students did not need to implement the control unit or add any multiplexers.

For the second assignment, the students were asked to implement the rest of the data path shown in Figure 1 and the logic in the control unit. We supplied the students with a simple outline which included lines 1–13 and line 18 in the code above. We also supplied the logic for all modules except the `ControlUnit`. The students' data path designs were tested with single instruction tests and with full applications as discussed in Section 3.1.

During the second time we used this set of assignments, we made minor changes to the data path so the solution would not be exactly the same between classes. For this class, we integrated the branch control logic into the ALU, similar to the original design in the Patterson and Hennessy book.

There are many other ways to make minor modifications to the data path or the signals which will produce a correct RISC-V CPU,

```
class IFIDBundle extends Bundle {
  val instruction = UInt(32.W)
  val pc          = UInt(32.W)
  val pcplusfour  = UInt(32.W)
}
class EXControl extends Bundle {
  val add       = Bool()
  val immediate = Bool()
  val alusrc1   = UInt(2.W)
  val branch    = Bool()
  val jump      = UInt(2.W)
}
class MControl extends Bundle {
}
class WBControl extends Bundle {
}
// Pipeline registers
class IFIDBundle extends Bundle {
}
class IDEXBundle extends Bundle {
  val excontrol = new EXControl
  val mcontrol  = new MControl
  val wbcontrol = new WBControl
}
class EXMEMBundle extends Bundle {
  val mcontrol  = new MControl
  val wbcontrol = new WBControl
}
class MEMWBBundle extends Bundle {
  val wbcontrol = new WBControl
}
```

Listing 2: Template Chisel code for control bundles and pipeline registers. Students are required to fill in missing control signals.

but make the Chisel code significantly different. As we continue using these assignments, we plan on making small changes to the CPU for each class.

2.3 Pipelined CPU design

Figure 2 shows the design of the baseline pipelined CPU. We also have a second version of this design with a branch predictor for the fourth assignment. This design supports full forwarding between writeback and memory to execute and hazard detection for load to use hazards and branch hazards.

The pipelined CPU uses all of the same modules from the single cycle design. The differences are the data path, which now includes pipeline registers and the forwarding and hazard detection units.

In the third assignment, we provided the students with outlines for the pipeline registers as shown in Listing 2, but did not specify the required control signals. The first step in the assignment was to fill in the pipeline registers with the required signals. This is a good example of the benefits of Chisel's modularity compared to using Verilog. The students were able to concentrate on one aspect of the design at a time (e.g., first deciding the required signals before writing the data path logic).

After specifying the signals in each pipeline register, the students implemented the pipeline without forwarding or hazard detection. We provided the students with a set of applications (about 50, most of which had a single instruction) which would correctly execute even without forwarding and hazard detection. Once they were able to correctly execute these tests, the students had confidence their data path logic was correct.

In this part, the students struggled the most with correctly updating the PC. The students tried to update the PC in the writeback stage instead of during the fetch stage even though this was specified on the diagram they were given (Figure 2). We believe this struggle was actually a good thing because instead of struggling with the tools or how to wire simple parts of the design, the students were struggling with the *concept* of pipelined designs. In many cases, after coming to office hours and asking questions about how pipelining works, the students were able to successfully complete the assignment. 95% of students in successfully completed this part of the assignment between the two times this assignment was used.

For the next part of the assignment, the students implemented full forwarding and hazard detection in the pipeline. The pipeline diagram the students were given contained all of the required wires, but it did not have the forwarding or hazard detection logic; the students had to consult the book or work it out on their own. The forwarding logic exactly matched the textbook, but the hazard detection logic was slightly different.

Similar to the first part, we supplied a set of applications which required forwarding (about 15) and another set of applications that required both forwarding and hazard detection (about 10). These were more complicated applications, usually containing a few instructions. 72% of the students successfully passed these tests.

Finally, we also provided the students with six full applications from the RISC-V test suite, which contained 100s of instructions and ran for 5,000–50,000 cycles. These were a multiply, median, quicksort, radix sort, towers, and vector-vector addition.

Many students failed to correctly execute these full applications on their pipeline design, which caused significant frustration. Only around half (57%) of the students executed all of the full applications successfully. We believe the main reason students were unsuccessful in this part of the assignment because the previous tests and applications did not cover all possible corner cases. When these large applications failed, it was very difficult to debug the exact problem. We are currently focusing on improving debugging support and adding more tests to cover the common mistakes found in the full application tests.

3 TOOLS INCLUDED WITH DINO CPU

We distribute a number of tools with the DINO CPU to enable students to use Chisel and develop the DINO CPU quickly. Chisel is a Scala-based language with a number of dependencies. Downloading these correctly is potentially error prone. We did not want to spend a significant amount of time in office hours or on online forums debugging the system configuration.

Therefore, we distribute a Singularity container [5] which contains all of the necessary software to develop, build, and test the DINO CPU. We chose to distribute a Singularity container instead of a Docker container to the students because our IT staff did not allow Docker for security reasons. This container contained the exact versions of Scala, Java, Chisel, and its dependencies the instructional staff used to develop the assignments so there would be no possibility of misconfigured software.

Using the container interface was relatively simple, although not painless. On Linux, working with the DINO CPU through Singularity was as simple as running a single command: `singularity run library://jlowepower/default/dinocpu`. This command will download the container image from Singularity Hub [8] and

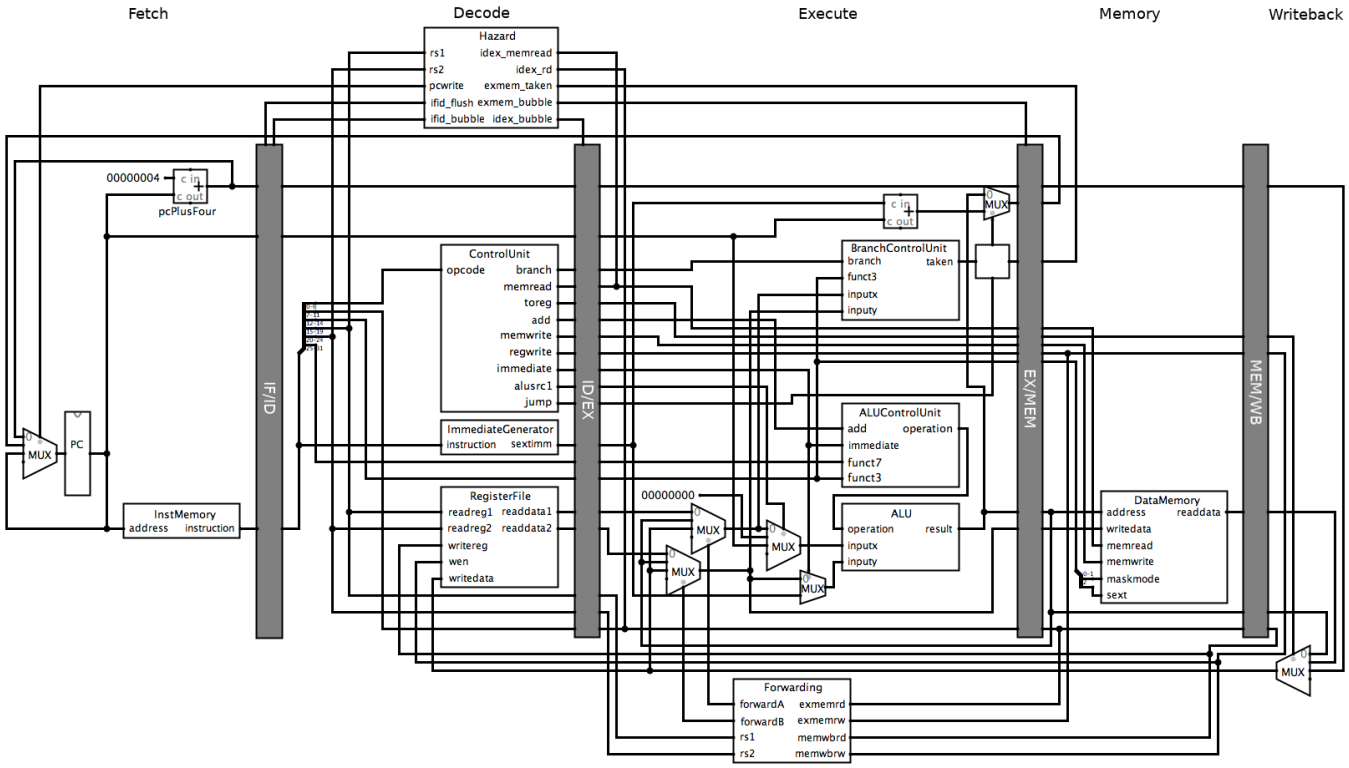

Figure 2: Pipelined RISC-V CPU diagram.

drops the user into the Scala Build Tool shell, which allows them to build, test, and run the DINO CPU simulator.

For students not using Linux (i.e., MacOS or Windows), they cannot directly use Singularity and must use a virtualized Linux environment. To make this simple, we distribute a Vagrant Box that contains all of the needed Linux tools for Singularity. Thus, on MacOS and Windows, the students had one extra step. Before running the container as above, they must start the Vagrant Box with `vagrant up && vagrant ssh`. Although there was some initial confusion on the first assignment with the tools, the students did not cite the tools as a point of frustration on any other assignment.

There were a few downsides to using this container/virtualization approach. The biggest downside to using Vagrant and Singularity is that on MacOS and Windows building the DINO CPU and simulating it was quite slow under default VirtualBox settings; increasing resources to 2 GB and 2 cores for the VM showed dramatic improvements. Also, the Singularity image was a 320MB file which took a significant fraction of our students' filesystem quota on our shared lab machines. Finally, requiring the students to use the command line interface on the Singularity container meant they could not use an IDE which would have made Chisel coding easier.

We decided not to include the entire RISC-V development toolchain (e.g., gcc, as, objdump, etc.) in our Singularity images. The main reason was that these tools take a significant amount of disk space (500 MB) and the images were already very large. Additionally, by not distributing the toolchain and requiring the students to use the provided binaries, we reduced the possibility of broken binaries causing tests to fail.

We also include a Dockerfile and code to autograde all of the assignments on Gradescope. This Dockerfile sets up an image with exactly the same versions of all dependencies as the Singularity container. We also have scripts to run the tests on Gradescope, and distribute a library which creates Gradescope compatible output. Finally, in our source repository, we have detailed documentation on using DINO CPU with autograders.

3.1 Testing support

We provide three different kinds of tests in DINO CPU to help the students and other developers test their designs: unit tests, simple instruction tests, and application tests.

Unit tests are modeled after Chisel unit tests and test a single component in isolation. We have unit tests for each of the units that we give to the students (e.g., ALU, memory, register file) and the units which the students design (e.g., the ALU control unit).

Unit tests are mostly helpful when designing the assignments and making updates to the DINO CPU infrastructure. We found it difficult for the students to effectively use these simple component-wise unit tests. In the future, as we require the students to fill in the logic for different units, these tests may be more useful.

We also constructed general CPU integration tests. These tests take a RISC-V binary, initial register values, and initial memory values as inputs as well as final register and memory values. Then,

the tester loads the binary into the simulator, simulates the CPU (either the single cycle or the pipelined design) for a set number of cycles. Finally, the tester compares the final register values and memory values to the correct values to generate the result.

We have two kinds of application tests: simple and full applications. Many of the simple application tests are just a single instruction (e.g., `add1` above is simply `add t1, zero, t0 # (reg[6] = 0 + reg[5])`). The full application tests range from simple assembly functions (e.g., swap using xors) to benchmarks (e.g., towers).

For simplicity, we distribute the RISC-V binaries built with the mainline of the RISC-V toolchain. We also distribute the RISC-V assembly source files so students can examine the assembly when the applications fail. In practice, we found the students rarely looked at the source code for failing applications and instead focused more on examining the output of the simulator.

Most of our tests are written in RISC-V assembly code; however, some of the benchmarks are written in C. As part of the DINO CPU distribution, we also include makefiles and loader files that will take RISC-V assembly and simple C code and create a RISC-V ELF binary that is compatible with our simulator, described next.

3.2 Debugging support

We also provide a Scala-based simulator and debugger with the DINO CPU. The simulator is based on Treadle [2] which is an circuit simulator that executes Firrtl IR written in Scala.

Our simulator loads RISC-V ELF files and can initialize registers and memory. It loads a subset of the symbols in the ELF file (`.text` and `.data`) into the simulated memory. It loads these symbols by parsing the ELF file and then writing a new text file which is loaded by the Chisel `loadMemoryFromFile` utility. The loader scripts distributed with DINO CPU ensure that all code and data are in these ELF sections.

Then, from a very simple read, execute, print loop (REPL), users can step through the application cycle by cycle. At each step, a subset of the CPU's state is dumped to the terminal. This state can be set by using `printf` in the Chisel code giving computer science students a familiar "printf debugging" interface. By default, we print the values of each of the pipeline registers, as shown in Listing 3.

Improving the debugging support by providing a more full featured REPL is high priority as debugging difficulty was one of the main pain points in the students' feedback " The output of Chisel is very difficult to read, so I cannot easily debug my program. "

```
Cycle=10 Cycles > ?
?       : print this help
q       : quit
number  : move forward this many cycles
Cycle=10 Cycles > 1
-----------------------------------------------
MEM/WB: MEMWBBundle(writereg -> 8, aluresult -> 0, readdata ->
↪  4294967187, pcplusfour -> 32, wbcontrol -> WBControl(toreg -> 0,
↪  regwrite -> 1))
EX/MEM: EXMEMBundle(writereg -> 9, readdata2 -> 0, aluresult -> 0,
↪  nextpc -> 36, pcplusfour -> 36, mcontrol -> MControl(memread ->
↪  0, memwrite -> 0, taken -> 0, maskmode -> 0, sext -> 1),
↪  wbcontrol -> WBControl(toreg -> 0, regwrite -> 1))
ID/EX: IDEXBundle(writereg -> 10, funct7 -> 0, funct3 -> 0, imm -> 0,
↪  readdata2 -> 0, readdata1 -> 0, pc -> 36, pcplusfour -> 40,
↪  excontrol -> EXControl(add -> 0, immediate -> 1, alusrc1 -> 0,
↪  branch -> 0, jump -> 0, prediction -> 0), mcontrol ->
↪  MControl(memread -> 0, memwrite -> 0, taken -> 0, maskmode -> 0,
↪  sext -> 1), wbcontrol -> WBControl(toreg -> 0, regwrite -> 1),
↪  rs1 -> 0, rs2 -> 0, branchpc -> 36)
DASM(593)
IF/ID: IFIDBundle(instruction -> 1427, pc -> 40, pcplusfour -> 44)
PC: 44
Cycle=11 Cycles >
```

Listing 3: Detailed debugging output from the single stepper included with the DINO CPU.

4 EXPERIENCE USING DINO CPU

Overall, we believe that DINO CPU infrastructure has succeeded at providing our students with an infrastructure to learn about tradeoffs in architectural design and details of pipelined processor control. Initially developing the DINO CPU infrastructure was a significant undertaking. This was our first experience with many of the tools including Chisel. However, after this initial investment, we have found that using DINO CPU in other classes is significantly less work.

4.1 Other instructor experience

As mentioned earlier, we have used the DINO CPU in two offerings of ECS 154B with different instructors. Despite being a researcher in the field of computer architecture for many years, this is the first time the second instructor has taught ECS 154B. The instructor had no prior experience with Chisel, nor did they have experience with RISC-V. Even without the prior experience the second instructor has found that they can take the provided assignments, modify them so as not to be identical to the publicly available DINO CPU, and solve them all on average of 4 hours.

There have been some advantages and disadvantages that the second instructor experienced during this quarter. The advantages the instructor has found thus far are:

- Ease of tool-chain setup
- Ability to easily modify existing assignments and to solve them
- Reduced overhead of managing assignments

The disadvantages the instructor has found thus far are:

- Questions that arise are difficult
- Many different languages/tools used

Ease of toolchain setup. The second instructor was able to setup the tool chain on OS-X with minimal effort. The requirements of VirtualBox and Vagrant were able to be met by simply installing them using the downloadable installers. Once Vagrant had been installed getting the rest of the tools setup were mainly cloning the DINO CPU and starting Vagrant. The biggest challenge the instructor found was not remembering to change directory into dinocpu before running singularity.

Ease of updating assignments. The assignments already had significant structure and well-written instructions, so making modifications for the second offering of ECS 154B was fairly straightforward. The instructor found that the most challenging (though not exceptionally challenging) part of updating the assignments were figuring out the parts of the testing infrastructure that needed to be modified to accomodate the modified assignments.

Reduced overhead of managing assignments. The instructor has found that there have been a relatively low number of questions

on the assignments as compared to past experience with Logisim assignments from prerequisite course (ECS 154A). The number of students that have attended office hours needing help on the assignments has been almost non-existent this quarter.

Questions that arise are difficult. There have been fairly few questions that have arisen this quarter; however, those that have arisen have been more complicated to answer. One common problem that arose was that the Chisel optimizer completely optimized out the register file and the error that was output was "`Cannot find cpu.registers.regs_5 in symbol table`". This problem occured because the write enable never was set to true so the register file would only ever output zeros. Fortunately, some of these difficult issues have been uncovered over the past months and have been added to the common issues.

Many different languages/tools used. There are many tools and languages being used in the DINO CPU projects. The use of Chisel, Scala, Java, Singularity, etc. makes it challenging to understand all of the parts and how they interact with one another. The file structure of projects makes it fairly easy to track issues down and to modiify the necessary parts.

4.2 Lessons Learned: What can be improved

Chisel is still a young language that is in flux with an average of 5–10 commits in the main Chisel repository weekly. Unfortunately, the simulator we used for debugging and testing applications required Chisel features that were not yet part of an official release. Since we could not use an official release supplied by the Scala package manager, we had to distribute our own compiled versions of Chisel and all of its dependencies to the students. This lead to bloated containers and long build times. We expect that Chisel will become more stable in the future, and we will be able to distribute DINO CPU with supported Chisel releases.

Another problem we ran into while using the relatively young Chisel language was that the documentation could be improved. The Chisel bootcamp [1] is a fantastic resource. However, most of the Chisel documentation targets graduate students and professionals with architecture and digital design background. Therefore, we supplemented this documentation with our own slimmed down documentation which only contained details required for completing the DINO CPU assignments.

Many of the student complaints were related to limited documentation. One of our main focuses while improving the DINO CPU will be to improve the documentation both for Chisel and for the DINO CPU more generally.

Finally, the biggest pain point in using DINO CPU in our classes has been students trying to debug their hardware. We believe this stems from two places: a misunderstanding of digital design logic and a lack of debugging support.

First, computer science students are often confused by the fact that Chisel is only *describing hardware* and not a programming language. We found many examples of students re-assigning the same wire twice as the example in Listing 4. The students expected that line 1 would execute first followed by line 3 like an imperative program. However, since this is describing hardware the above lines actually mean "Connect the output wire from `pcPlusFour` to the `pc` register. ... connect the output wire from the `branchAdd` to the `pc` register." The second statement *overwrites* the first. In the future, we believe that a Chisel or FIRRTL pass which detects overwritten wires may help to give warnings or errors when this misunderstanding occurs.

```
1 pc := pcPlusFour.io.output
2 ...
3 pc := branchAdd.io.result
```

Listing 4: Example common mistake. The students frequently left out multiplexers because they assumed the code was executed imperatively.

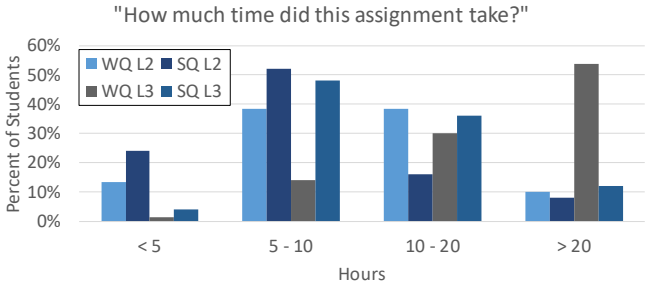

Figure 3: Time spent on second and third assignments.

Second, 37% of the class that provided feedback on one of the labs specifically mentioned that lack of debugging support made the assignment more difficult. The first time we used DINO CPU, we did not provide the students a way to "single step" through the execution of examples. There was significant frustration when running the applications on the simulator because it would fail after 100s or 1000s of cycles and there was no way to easily track the execution over time.

To partially alleviate this issues, we implemented a single step feature in our simulator as shown in Section 3.2 which allows students to step one cycle at a time and investigate the current state of the registers. We are currently working on extending this to be more feature rich. For instance, we would like the students to be able to print the value of any wire or register when the simulation is paused like gdb's print command. The use of the single step feature in this quarter appears to have reduced the amount of time students have spent on the assignments. Figure 3 shows the amount of time the students have reported they spend working on the assignments.

4.3 Using DINO CPU for future assignments

DINO CPU is an open source project, and we hope that our effort can be leveraged by other instructors. There are a number of ways to make minor changes to the CPU pipeline that will not affect functional correctness, but do require significantly different Chisel code. By making some of these minor modifications each time the assignments are used, we believe it will reduce the incidence of cheating. A few ideas are below.

- Change the way branches are resolved by changing the inputs on the branch control unit or merging the branch control unit with the ALU.
- Changing the meaning of the ALU control unit signals.

- Changing the way the auipc instruction is implemented (i.e., how the PC moves through the pipeline).
- Changing the way the zero register is implemented.
- Moving the branch resolution logic to the decode stage.
- Changing the I/O for the control unit.
- Change the I/O for the data memory (e.g. change maskmode and sext to just funct3)

Additionally, we have implemented a couple of extensions to the DINO CPU in our classes, but many other extensions are possible.

- Add a branch predictor. We added a branch predictor to the decode stage, but it could be moved to the fetch stage.
- Adding more stages to the pipeline or removing stages from the pipeline.
- Updating the instruction and data memory interfaces to work asynchrounously instead of sequentially and then adding caches will be possible.
- Adding multiple issue (e.g., 2-way superscalar) should be straightforward even without scoreboarding.
- Implementing more RISC-V instructions (e.g., floating point, compressed instructions, etc.)

Finally, we are working to extend the DINO CPU to implement a subset of the privileged instructions to be able to execute interrupts and exceptions. With this support, we believe students could even write simple operating systems that can execute on the DINO CPU.

5 CONCLUSIONS

After using the DINO CPU for two quarters, we believe that it has improved the learning outcomes in our computer architecture course. We are currently working to improve the DINO CPU, and we plan to continue improving it as we use it in future classes.

The code, tools, and documentation for the DINO CPU are available on GitHub at https://github.com/jlpteaching/dinocpu. The DINO CPU is an open source project, and we welcome any contrubitions from the community. We look forward to working with the architecture education community to improve the DINO CPU.

6 ACKNOWLEDGEMENTS

We wish to thank the students in UC Davis ECS 154B in Winter and Spring Quarters 2019, especially those in Winter 2019 who helped us institute a whole new set of assignments and provided feedback that helped us refine the DINO CPU design and the associated assignments. Additionally, Jared Barocsi, Filipe Eduardo Borges, Nima Ganjehloo, Daniel Grau, Markus Hankins, and Justin Perona have made contributions to the assignments, documentation, and DINO CPU code.

REFERENCES

[1] 2019. Chisel Bootcamp. https://github.com/freechipsproject/chisel-bootcamp
[2] 2019. *Treadle – A Chisel/Firrtl Execution Engine.* https://github.com/freechipsproject/treadle
[3] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. 2012. Chisel: constructing hardware in a Scala embedded language. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012.* 1216–1225. https://doi.org/10.1145/2228360.2228584
[4] Carl Burch. 2002. Logisim: a graphical system for logic circuit design and simulation. *ACM Journal of Educational Resources in Computing* 2, 1 (2002), 5–16. https://doi.org/10.1145/545197.545199
[5] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 2017. Singularity: Scientific containers for mobility of compute. *PLOS ONE* 12, 5 (05 2017), 1–20. https://doi.org/10.1371/journal.pone.0177459
[6] Derek Lockhart, Stephen Twigg, Doug Hogberg, George Huang, Ravi Narayanaswami andJeremy Coriell, Uday Dasari, Richard Ho, Doug Hogberg, George Huang, Anand Kane, Chintan Kaur, Tao Liu, Adriana Maggiore, Kevin Townsend, and Emre Tuncer. 2018. Experiences Building Edge TPU with Chisel. In *Chisel Community Conference.*
[7] David A. Patterson and John L. Hennessy. 2017. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[8] Vanessa V. Sochat, Cameron J. Prybol, and Gregory M. Kurtzer. 2017. Enhancing reproducibility in scientific computing: Metrics and registry for Singularity containers. *PLOS ONE* 12, 11 (11 2017), 1–24. https://doi.org/10.1371/journal.pone.0188511