Engineering Educators Bringing the World Together
2025 ASEE Annual Conference & Exposition
Palais des congrès de Montréal, Montréal, QC • June 22–25, 2025 ASEE
Paper ID #48548

BOARD # 96: WIP: Teaching Computer Architecture Using a Python Hardware Description Language

Dr. Alan Marchiori, Bucknell University

WIP: Teaching Computer Architecture Using a Python Hardware Description Language

Introduction

Undergraduate Computer Science students typically take at least one course that introduces computer architecture. This course might cover binary representation of data, assembly programming, memory hierarchy, and a RISC datapath. However, beyond writing and simulating assembly language programs, it is difficult to provide practical hands-on experience with these computer architecture concepts. Various software tools exist to allow students to simulate digital logic, such as Digital [1] and Logisim Evolution [2]. Although you can build a full CPU using these systems, they are not intended to study computer architecture more broadly and can be quite tedious. Full hardware design languages such as Verilog, SystemC, and VHDL are quite adept at simulating modern computer architectures. However, these tools are specialized, and undergraduate computer science students find a steep barrier to learning and correctly applying them.

Our goal is to guide students in the design and implementation of a RISC-V CPU, culminating in a functional implementation of a single cycle CPU capable of executing bare-metal binary files compiled by standard GCC tools. Students begin by simulating simple digital logic circuits and incrementally add functionality to implement the fully functional CPU in four structured projects: 1) decode, 2) fetch, 3) registers & ALU, and finally 4) branches, jumps, and memory. The final CPU functionality is verified by running the official `rv32ui` test suite from `riscv-tests` [3]. Building upon the single-cycle implementation, students can then explore more advanced topics such as pipelining (3- or 5-stage), cache performance, and branch prediction in the single-cycle implementation. In three iterations of this approach, students report a strong sense of accomplishment by completing the project, and at the same time it helps students develop a deeper understanding of the architecture, datapath, and control concepts.

Python as an HDL

Python is not a hardware description language, but as a general purpose computing language, it can be used as both an HDL and a simple HDL simulator. In fact, MyHDL [4] is a full-featured open-source Python HDL. However, this and other similar solutions focus on synthesis to produce ASIC and FPGA designs, which adds significant complexity and constraints. In this work, we define four basic rules to develop a minimal behavioral HDL in Python.

1. **Modules** are implemented as **classes**.
2. Inputs and outputs are **functions**.
3. Inputs are passed into the class **constructor**.
4. Outputs are **class methods**.

In Verilog, the basic building block is the `module` [5]. A module provides an interface (inputs and outputs) to an implementation of some operation. Since a module can contain internal state, the analogous building block in Python is a `class` with class methods providing output. Table 1

shows a side-by-side example of implementing and testing an AND gate in Verilog and our Python HDL. We will next walk through this example to demonstrate the key points.

Verilog AND Gate

```
1  module andgate (
2    output wire out,
3    input wire in1,
4    input wire in2  );
5    assign out = in1 & in2;
6  endmodule
7
8
9
10
11
12
13 module testbench;
14   reg a, b; wire out;
15   andgate device (out, a, b);
16   initial begin
17     $monitor("a=%b, b=%b, out=%b",
18       a, b, out);
19     // step through the inputs
20     #10 a = 0; b = 0;
21     #10 a = 0; b = 1;
22     #10 a = 1; b = 0;
23     #10 a = 1; b = 1;
24     #10 $finish;
25   end
26 endmodule
```

Python AND Gate

```
class andgate:
  def __init__(self, in1, in2):
    self.in1 = in1; self.in2 = in2
  def __call__(self):
    return self.in1() & self.in2()
class reg:
  def __init__(self, val = None):
    self.val = val
  def __call__(self):
    return self.val
  def set(self, val):
    self.val = val
def testbench():
  a = reg(); b = reg()
  device = andgate(a, b)
  def monitor():
    print("a=%s, b=%s, out=%s" %\
      (a(), b(), device()))
  # step through the inputs
  a.set(0); b.set(0); monitor()
  a.set(0); b.set(1); monitor()
  a.set(1); b.set(0); monitor()
  a.set(1); b.set(1); monitor()

testbench()
```

Table 1: Verilog and Python implementations for an AND gate.

In the above example implementations, we keep the functionality of each line as close as possible. Lines 1–5 are the definition of the AND gate. In Verilog, a continuous assignment statement wires the output to the logical and of the two inputs. In Python, the constructor stores the inputs, which are `reg` instances that are called in line 5 to provide their current values and generate the output. Lines 6-12 are unique to Python. Since Python's primitive data types are immutable, we must define a mutable object to support changing a value passed to a module (i.e., `andgate`). Defining a `reg` class allows us to bind inputs to class instances in their constructor and change their values throughout the simulation. To obtain the value of the reg, we call the object as seen on line 5. We then define a test bench to exercise the `andgate` in lines 13-24. Lines 20-23 step through the possible inputs of the AND gate. Python does not have a way to monitor variables, so we simulate this using a monitor function that prints the desired information when called. Verilog schedules execution of the monitor at the end of the simulation time step. Since we did not develop a full simulator, calling monitor is the implicit end of each simulation step in our Python simulation. The output of both implementations are show in Table 2. The Verilog monitor displays the values before the initial assignment, resulting in an additional line of output. We could achieve a similar result in Python by executing the monitor prior to setting the values. However, the andgate logic would have to be modified to handle undefined inputs, because `None & None` is undefined in Python.

Verilog AND Gate	Python AND Gate
a=x, b=x, out=x a=0, b=0, out=0 a=0, b=1, out=0 a=1, b=0, out=0 a=1, b=1, out=1	a=0, b=0, out=0 a=0, b=1, out=0 a=1, b=0, out=0 a=1, b=1, out=1

Table 2: Verilog and Python AND gate output.

Using these basic rules, we can describe and simulate more complex systems. Table 3 shows example logic gates. The `xorgate` demonstrates the composition of multiple classes. The `mux` class implements an N-to-1 multiplexer, where the first input selects which input to use for output. The mux is used as a hardware `if` statement. Finally, sequential logic is demonstrated with a D flip-flop implementation using the previously defined `reg` class. The `clock` method is called by the test bench when the system clock occurs. Once we understand these basic rules, we can move on to constructing the RISC-V processor.

```
class gate:
  "Base class for all gates"
  def __init__(self, *args):
    self.args = args
  def sample(self):
    return (x() for x in self.args)
  def __call__(self):
    return self.out()
class notgate(gate):
  def __init__(self, a):
    super().__init__(a)
  def out(self):
    return not self.args[0]()
class andgate(gate):
  def __init__(self, *args):
    super().__init__(*args)
  def out(self):
    return all(self.sample())
class xorgate(gate):
  "A ^ B = AB' + A'B"
  def __init__(self, a, b):
    super().__init__(a, b)
    self.out = orgate(
        andgate(self.args[0], notgate(self.args[1])),
        andgate(notgate(self.args[0]), self.args[1]))
```

```
class orgate(gate):
  def __init__(self, *args):
    super().__init__(*args)
  def out(self):
    return any(self.sample())
class mux(gate):
  "N input multiplexer"
  def __init__(self, *args):
    super().__init__(*args[1:])
    self.sel = args[0]
  def out(self):
    return self.args[self.sel()]()
class DFlipFlop(gate):
  def __init__(self, d, init='X'):
    super().__init__(d)
    self.q = reg(init)
  def out(self):
    return self.q()
  def clock(self):
    self.q.set(self.args[0]())
```

Table 3: Example Python hardware descriptions.

Single Stage RISC-V

We leverage the UC Berkeley Architecture Research Sodor 1-stage implementation [6] as a foundation and decompose the project into three distinct stages, illustrated in Figure 1.

Project 1: Instruction Fetch Logic: Students begin by implementing the instruction fetch unit,

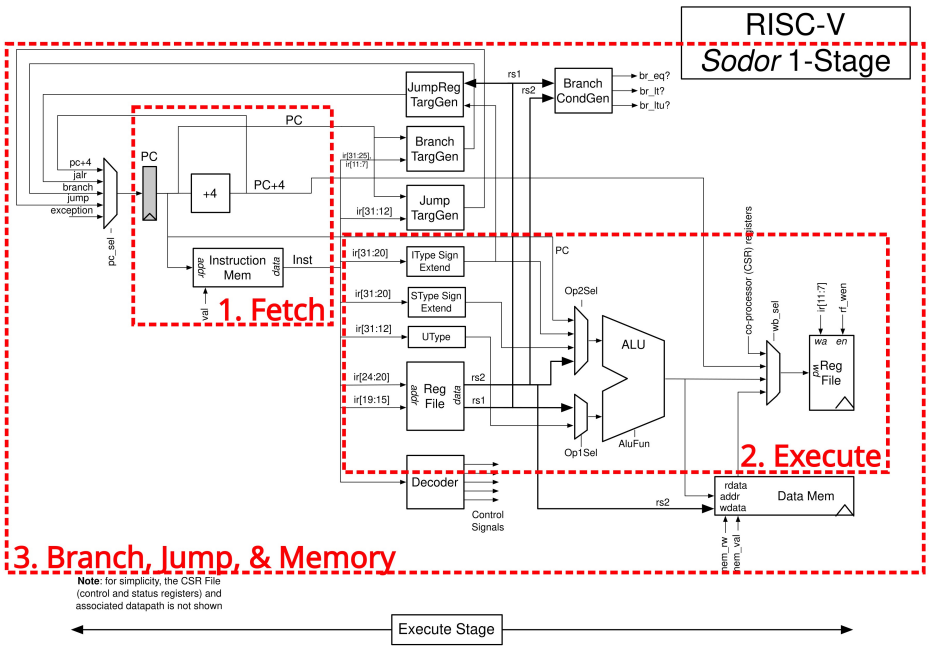

Figure 1: Sodor 1-stage RISC-V Processor split into three development phases.

which is responsible for retrieving instructions from memory and displaying them to the terminal.

Project 2: ALU and Register File: The second project focuses on the implementation of the Arithmetic Logic Unit (ALU) and the register file, enabling the execution of integer arithmetic operations.

Project 3: Branching, Jumping, and Data Memory: In the final project, students implement the logic for branch and jump instructions, along with data memory access. Upon completion, the CPU becomes capable of executing the full rv32ui instruction set and the provided unit tests [3].

This modular approach allows students to progressively build upon their understanding and facilitates a step-by-step exploration of the core components of a RISC-V processor. Following the single-cycle implementation, students can go into more advanced topics such as pipelining, cache optimization, and branch prediction.

For testing, we provide students with ELF-format binary files generated using the riscv GCC compiler built with the Spike simulator proxy kernel [7]. These ELF files can be parsed using the pyelftools library [8] to extract the symbol table and memory segments.

Comprehensive project descriptions and links to starter code are available in the following public repository: `https://burl.live/asee_riscv`.

Discussion and Conclusions

This project has been refined through multiple iterations of our Undergraduate Computer Architecture course, building upon prior experiences with Verilog assignments. In previous iterations, students lacking prior Verilog experience faced significant challenges, struggling to grasp both Verilog syntax and the course material concurrently. Some students still encounter difficulties in adhering to the fundamental principles of hardware description, often resorting to procedural coding techniques to simulate the RISC-V processor. For example, they might employ `if` statements and dynamically instantiate new objects after clock cycles, inadvertently deviating from the inherent immutability of the hardware during execution. To mitigate these challenges, we emphasize the principle of hardware immutability and provide students with well-structured sample code, clearly delineating sections for hardware description.

Student feedback consistently indicates a positive reception towards the project. They recognize its significant value in solidifying their understanding of key concepts in computer architecture, particularly the RISC-V Instruction Set Architecture (ISA) and CPU design. Students particularly appreciate the hands-on experience, finding it instrumental in clarifying many of the theoretical concepts explored throughout the course, such as:

- The project provided a clearer picture of how a CPU works, solidifying theoretical concepts through practical implementation.
- It was a valuable hands-on experience that deepened understanding beyond memorization.
- The assignment improved debugging and problem-solving skills.
- The project helped students understand the importance of control signals in CPU operation.
- Students expressed appreciation for seeing how all the course material came together in one project.

Althoguh student feedback consistently indicates a positive reception towards the project, several challenges were also identified:

- Navigating the starter code: Students found the starting code difficult to understand and navigate, which added to the complexity of the assignment. There was confusion about where to start with the code and how to build on previous assignments.
- Debugging Difficulties: Debugging was a significant challenge due to the complexity of the project, especially with interconnected parts of the CPU. Errors were sometimes confusing or did not directly point to the source of the problem.
- Two's Complement Confusion: Understanding and implementing two's complement calculations was a common source of difficulty and confusion for students.
- Time management challenges: Students struggled with time management and the workload for the project.

We believe that this approach can serve as a valuable foundation for helping undergraduate students effectively explore fundamental concepts in computer architecture. While this approach demonstrates one approach to using Python, we acknowledge that there is always room for improvement. We encourage collaboration and open dialogue within the educational community and welcome suggestions for further refinement and improvement to enhance the student learning experience.

References

[1] H. Neemann, “GitHub - hneemann/Digital: A digital logic designer and circuit simulator.” [Online]. Available: https://github.com/hneemann/Digital

[2] C. Burch, T. Kluter, T. Maehne, K. Walsh, D. H. Hutchens, M. Orlowski, T. Niget, M. Berman, and T. Cruz Franqueira, “Logisim-evolution,” Aug. 2024, original-date: 2014-09-19. [Online]. Available: https://github.com/logisim-evolution/logisim-evolution

[3] RISC-V Software, “riscv-tests,” Dec. 2024, original-date: 2013-04-24. [Online]. Available: https://github.com/riscv-software-src/riscv-tests

[4] J. Decaluwe, “MyHDL.” [Online]. Available: https://www.myhdl.org/

[5] S. Palnitkar, *Verilog HDL: A guide to digital design and synthesis*. Prentice Hall, 2003.

[6] UC Berkeley Architecture Research, “GitHub - ucb-bar/riscv-sodor: educational microarchitectures for risc-v isa.” [Online]. Available: https://github.com/ucb-bar/riscv-sodor

[7] R.-V. Software, “Spike, a risc-v isa simulator,” Jan. 2025, original-date: 2011-08-26. [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim

[8] E. Bendersky, “Parsing elf and dwarf in python,” Jan. 2025, original-date: 2013-06-08. [Online]. Available: https://github.com/eliben/pyelftools