
AFRL-RY-WP-TP-2019-0127

# REVERSE ENGINEERING RISC-V GENERATOR-BASED DESIGNS FOR TRUST AND ASSURANCE

**Jeffrey Durrum, Adam Bryant, and Jeremy Porter**

**Trusted Electronics Branch**
**Aerospace Components & Subsystems Division**

**JULY 2019**
**Final Report**



**AIR FORCE RESEARCH LABORATORY**
**SENSORS DIRECTORATE**
**WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320**
**AIR FORCE MATERIEL COMMAND**
**UNITED STATES AIR FORCE**

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. REPORT DATE *(DD-MM-YY)* | 2. REPORT TYPE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| July 2019 | Technical Paper | 15 May 2019 – 16 May 2019 |

**4. TITLE AND SUBTITLE**
REVERSE ENGINEERING RISC-V GENERATOR-BASED DESIGNS FOR TRUST AND ASSURANCE

**5a. CONTRACT NUMBER**
FA8650-17-F-1044, FA8650-18-D-1614, FA8650-18-F-1615, and FA8075-14-D-0025 DO 0013

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**
N/A

**6. AUTHOR(S)**
Jeffrey Durrum, Adam Bryant, and Jeremy Porter

**5d. PROJECT NUMBER**
N/A

**5e. TASK NUMBER**
N/A

**5f. WORK UNIT NUMBER**
N/A

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Air Force Research Laboratory, Sensors Directorate (AFRL/RYDT)
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command,
United States Air Force

**8. PERFORMING ORGANIZATION REPORT NUMBER**
AFRL-RY-WP-TP-2019-0127

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

**10. SPONSORING/MONITORING AGENCY ACRONYM(S)**
AFRL/RYDT

**11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)**
AFRL-RY-WP-TP-2019-0127

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution is unlimited.

**13. SUPPLEMENTARY NOTES**
PAO case number 88ABW-2019-2242, Clearance Date 8 May 2019. To be presented at the GOMAC, Albuquerque, NM, May 15, 2019.This is a work of the U.S. Government and is not subject to copyright protection in the United States.. Report contains color.

**14. ABSTRACT**

We reverse engineered the open source Rocket-Chip system-on-a-chip (SoC) generator so we could understand the constructs and IP generation in sufficient detail to trust it for technology transition into our own SoC. We investigated its parameterization and use, configuration of the core, design of generated interfaces and peripherals, and how it ultimately generates Verilog register transfer language code that will be input to the synthesis work flow. Our goal was to understand and verify the assumptions involved in generation at each stage of the tool chain so we can have higher assurance in our ultimate design, and that we could verify the IP we will be using in our system. We discuss the reverse engineering methodology we used to gain precise information about details of the chip, and discuss areas for future work in creating tests and extracting information to verify generator-based designs.

**15. SUBJECT TERMS**
reverse engineering, RISC-V

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT: | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON (Monitor) |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Vipul Patel |
| Unclassified | Unclassified | Unclassified | SAR | 8 | 19b. TELEPHONE NUMBER *(Include Area Code)* N/A |

# Reverse Engineering RISC-V Generator-Based Designs for Trust and Assurance

Jeff Durrum
Booz Allen Hamilton
Dayton, Ohio 45433
Email: durrum_jeffrey@bah.com

Jeremy Porter
The Design Knowledge Company
Dayton, Ohio 45433
Email: jporter@tdkc.com

Adam Bryant
The Design Knowledge Company
Dayton, Ohio 45433
Email: abryant@tdkc.com

***Abstract*—We reverse engineered the open source Rocket-Chip system-on-a-chip (SoC) generator so we could understand the constructs and IP generation in sufficient detail to trust it for technology transition into our own SoC. We investigated its parameterization and use, configuration of the core, design of generated interfaces and peripherals, and how it ultimately generates Verilog register transfer language code that will be input to the synthesis work flow. Our goal was to understand and verify the assumptions involved in generation at each stage of the tool chain so we can have higher assurance in our ultimate design, and that we could verify the IP we will be using in our system. We discuss the reverse engineering methodology we used to gain precise information about details of the chip, and discuss areas for future work in creating tests and extracting information to verify generator-based designs.**

## I. Introduction

In our plans to create a RISC-V system-on-a-chip (SoC), we investigated the open source Rocket-Chip SoC generator [1] developed from work started at The University of California, Berkeley. The core idea behind the Rocket-Chip is that users specify their design using a high-level hardware generation language called Chisel [2], use the supplied tools to compile it to the FIRRTL intermediate representation language [3], and can quickly generate a customized RISC-V chip design. Designers then can use higher-level programming constructs to constrain and parameterize changes to their chip design. Design modifications propagate from the alteration point to the rest of the SoC structure, or alternately, they flag errors to be caught and handled at design time. The process enables an agile methodology that allows late stage design modifications and feature additions [4].

Our team is including a RISC-V core into our design, however, there is not enough information to trust the Rocket-Chip generator or the SoC designs it can produce. For this reason, we sought to:

- Understand IP generation,
- Informally evaluate its quality,
- Generate IP for our SoC, and
- Develop methods to trust generated IP.

We analyzed numerous artifacts including:

1) The Rocket-Chip source code (mostly Scala with some Chisel),
2) The build system (FIRRTL, sbt metadata, Make),
3) The generated RTL (Verilog),
4) Ancillary generated files (JSON, C++, RISC-V, ELF, GraphML, ASCII, Device Tree files, etc.),
5) Test files (RISC-V, C++, Scala), and
6) Data from simulation.

While we spent time to learn each of the technologies involved, we directed our main focus to understanding how the circuit works by studying the Verilog code that the generator emits.

## II. Configuration

We targeted our first milestone, "Build-0," at constructing a self-hosted default configuration of the Rocket-Chip. We planned no modifications, parameterizations, or configuration changes other than modifying the boot-up process.

Rocket-Chip requires installation of the RISC-V tools which are included as submodules in the Rocket-Chip git repository. These include several packages:

- riscv-fesvr – the front end server,
- riscv-isa-sim – the Spike instruction set simulator,
- riscv-gnu-toolchain – the compiler tool chains including GDB, Binutils, QEMU, GCC, glibc, and DejaGnu,
- riscv-gnu-linux – needed to build the Linux kernel,
- riscv-open-ocd – the OpenOCD debugger interface,
- riscv64-unknown-elf – programs to build Linux,
- riscv-pk – a proxy kernel for handling system calls,
- riscv-opcodes – RISC-V instruction opcodes,
- riscv-tests – a battery of processor tests [5].

We built the RISC-V tools and Rocket-Chip software on hosts connected to our local *Research & Development Test & Engineering Network* (RDT&E) and on the cloud-based *Trusted Silicon Stratus* platform. Like many secured networks, these environments restrict users from having root access or installing their own software. This required installing more than 30 packages, which required network approval. They could have also been built with user-level software installations such as LinuxBrew [6], but in our experience, these solutions are at worst prohibited by network policy, and at best seriously mistrusted by network administrators.

Installing Rocket-Chip in a "secured" environment can require weeks to work through configuration issues. Our team ran into numerous such problems such as:

- Bootstrapping GCC 5.0+ and GLibC,
- Installing the Scala Build Tool (SBT) and compiler,

- Build errors caused by an outdated Java compiler,
- Errors building systemd (a pre-requisite for libusb), and
- Difficulties installing the IntelliJ and the yEd viewer.

Each of these challenges is surmountable with time and patience, but combined, they factor heavily into the overall cost and schedule of an SoC project.

## III. Reverse Engineering

The Rocket-Chip generates a single RTL file with around 400 thousand lines of behavioral Verilog. It also generates several other ancillary Verilog and data files. Because of the size and ambiguity of the code, it did not provide an intuitive understanding of the design. It was difficult to discern the data path or determine the locations of important structures such as the register files, boot ROM, SRAM memories, busses, and core modules.

To better understand the data path and visualize the processor logic, we used three techniques:

1) Separate the file into individual Verilog files with one module per file (resulting in 273 files),
2) Follow the data flow on the Cadence Palladium Z1 Enterprise Emulation Platform to verify the functionality of individual logic units, and
3) Build simple visualizations to track connections between modules and between assignment statements.

Rocket-Chip builds a GraphML file containing information about the connections between important modules in the SoC. It shows the as-described connections between components and can be viewed with an application that can process GraphML files, such as the yEd viewer [7]. The names in the GraphML diagram do not directly correlate to the names of the modules in the generated Verilog and neither directly correlate to class or object names in the Chisel source code. This made it difficult to track down the location and use of each module with precision. Each line of the RTL file contains auto-generated annotations that point the user to locations in Chisel and FIRRTL files. These are usually helpful in locating a Chisel file, but are less helpful in locating specific variables and assignments. In some cases there will be up to 40 lines of Verilog all with the same annotation pointing to a location in a seemingly unrelated source code file.

### A. Finding the RISC-V Registers

A central feature of the RISC-V instruction set architecture is that it is designed around a set of 32 registers and uses them carry out instructions so finding where the Rocket-Chip implements its registers was very important. Finding the RISC-V application binary interface (ABI) registers was not straightforward because there was nothing specifically indicating their location in the RTL. We had to narrow down possible locations using the hierarchical structure of the SoC exposed with Cadence SimVision (Figure 1).

The single Rocket-Chip core is located in a structure called a tile. The tile communicates to other internal SoC modules through a system bus over the TileLink protocol. The system bus is connected to the main tile, a memory-mapped I/O unit

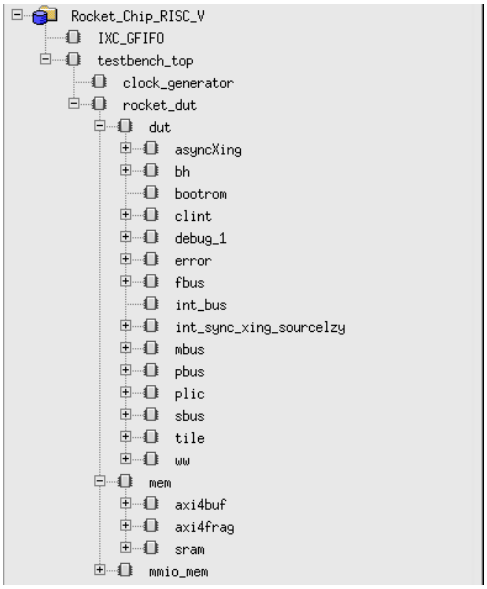


Fig. 1. The default RISC-V SoC structure as a tree

(MMIO), an error module, a memory bus, a front bus, and a periphery bus. The periphery bus is connected to two interrupt controllers (platform-level and core local interrupt), a debug module, and a module called "TLROM" that provides logic to handle the first stage boot up process. This information helped us isolate which modules were important to finding the register files and which were not.

Secondly, we searched for anything related to "register," "register_file," or "regfile" in the Verilog, but these searches only led us to numerous register red herrings in places such as the floating point unit or or the location of the command status registers (CSR). We used regular expressions to find a 2-dimensional array of 32 registers that are either 32 or 64 bits wide. In the "Rocket" module, we found the declaration:

```
reg [63:0] _T_1671 [0:30];
```

By excluding other candidates, we interpreted the Rocket module to contain the processor's data path and this to be the main register file. It only has 31 registers because the zeroth register is hard-wired to zero.

One thing that was surprising was that the register file had a temporary auto-generated variable name of "_T_1671." On subsequent re-compilations, the variable name changed. Ideally, important constructs would have permanent and intentional variable names, but often that is not the case when the Rocket-Chip generates RTL. This makes it difficult to locate important structures and to create test benches to verify the pre- or post-synthesis Verilog.

### B. Initial Experimentation

After generating the chip and identifying important components, the next step was to emulate it and observe its activity. We wanted to see how the chip communicates data to and from the registers. This required knowing what stimulus to provide,

how to send instructions to the processor, and what protocol was needed to send instructions. Unfortunately, documentation did not provide adequate insight and none of these things were immediately obvious.

We began an exploratory probe of the chip with Tcl commands using the Cadence Palladium XeDebug gui while observing signal activity in the Cadence Simvision Simulation Analysis Environment. In these initial emulation efforts, the Rocket Chip system showed very few signs of expected functionality. We tried writing bare metal RISC-V instructions directly to the FPU, data cache, periphery bus, and front bus modules (*fpu*, *dcache*, *pbus*, and *fbus*, respectively). We also tried toggling bits on the core local interrupt (*clint*), platform level interrupt (*plic*), and debug module control (*dmcontrol*) modules. None of these actions seemed to coax the system into an operational state.

After a lot of observations and bolstered by documents published by the Lo-RISC project [8], we began to investigate the concept of a *tethered* vs. *un-tethered* Rocket Chip. We eventually realized supplying the processor with instructions and other associated signals, as we had been attempting, was the role of the front end server (*fesvr*) connected to a tethered RISC-V chip.

Our goal was to run the processor in an un-tethered mode and tape it out on an ASIC, so it was necessary to make the chip come to life with only a clock and a reset signal. The processor still would be sent instructions, but it would be the logic of the un-tethered Rocket Chip System that would handle this task, rather than a software-driven routine from a co-processor.

### C. Monitoring Boot Sequence

The first observable activity should be the emergence of instructions from a Boot ROM, followed by a larger boot loader – in this case, the Berkeley Boot Loader (BBL). For this reason, we focused on how the signals propagate from reset through the boot sequence to see how the chip initializes. This focus drove our discovery of a large area of the chip's architecture and data flow.

Five clock cycles after reset, the initial signals eventually flow through an instance of a *TLROM* module instance called *bootrom* which issues a burst of eight 64-bit words. In addition to these eight values, there are about 300 other 64 bit values in the *bootrom* module instance. The eight 64-bit values are propagated through the logic of the Rocket Chip and 13 clock cycles after emerging from the *bootrom* module, they appear on the instruction buffer, parsed into sixteen 32-bit instructions. The first five instructions are valid RISC-V instructions ('hF140_2573, 'h0000_0597, 'h03C5_8593, 'h1050_0073, and 'h0000_BFF5), followed by eleven more that are populated with zeros.

We disassembled these instructions, which are hard coded as muxes in the *bootrom* module of the RTL. We compared that code to the "bootrom/bootrom.img" and "bootrom/bootrom.S" files in the rocket-chip source tree and found that they are almost identical.

At this point, the chip activity diverged from what we expected. Rather than jump to the BBL we had loaded into memory via a Tcl emulation load command, the processor went into an idle state. The instruction buffer froze on the last boot loader instruction, 'h0000_BFF5, and all observable signals went static.

It appeared that the chip was expecting an interrupt, or some further instructions, possibly from the front end server as if it were in tethered mode. We investigated numerous explanations for this deadlocked boot sequence but had no success until we mapped out crucial structures of the RISC-V and traced the path from the registers, through the logic and routing of the chip, and then correlated this with the boot ROM assembly code. Once we did this, we were able to understand the problem and get the chip to boot correctly.

### D. Global Reset Vector and moving on to BBL

The presence of over 500 RISC-V instructions in the disassembled boot ROM image file raised the question of how the processor chooses the eight values from the boot ROM after reset. The *bootrom* module instance is simply a look-up table with no control over what it does. Its indexing operations are performed elsewhere in the chip.

Working backwards from the address port of the *bootrom* module, we saw that the address originates from somewhere in the periphery bus. A module instance in the periphery bus, called *fragmenter* and nested within in an instance called *coupler_to_slave_named_bootrom*, generates the address signal. It does this by OR-ing a static vector (the hex value 'h1_0040) with a signal that counts upward from 'h8 to 'h38 in increments of eight. This addressing logic targets a burst of eight sequential values that begin at a specific address; the vector establishes the first address of the burst and the counter signal steps through the values.

The generation of the signal that increments the index into the boot ROM demonstrates how the Rocket Chip implements logic networks. It is driven by a long chain of temporary signals, automatically named with a number and the prefix "_T_" and linked together with Verilog assign statements. Each step in this cascade performs some sort of logical operation, comparison, negation, or arithmetic on the previous temporary-named signal. This is also how the Rocket Chip implements the look-up table functionality of the *bootrom* module.

We traced the source of the 'h1_0040 vector through the system bus and *RocketTile* module, all the way back to a magic numbered register called *global_reset_vector*. The *global_reset_vector* is generated in the BootROM.scala file and given the value of 0x1_0040. This value corresponds to a parameter called *hang* that serves the additional function of creating a *_hang* boot routine at that same address, which is the same as what is described in the bootrom.S file (Listing 1).

The code in the `_hang` routine initializes a hardware thread, points the `a1` register to a space allocated for a device tree blob, clears machine mode interrupts, and then spins waiting

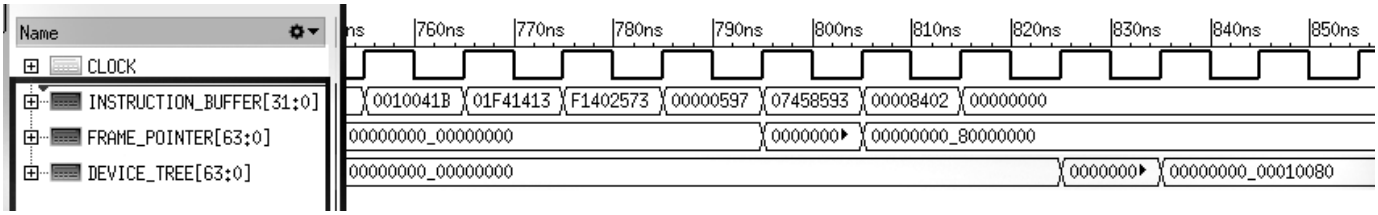


Fig. 2. Boot ROM instructions on the Instruction Buffer resultant writes.

for an interrupt. The bootrom.S file also defines a routine called `_start` which we experimentally determined to be at location 'h1_0000 in the TLROM. The `_start` routine loads a pointer to a memory value into the `s0` register, initializes a hardware thread, loads a simple device tree blob into the `a1` register, then jumps to the memory address loaded earlier (`DRAM_BASE`: 'h8000_0000). By changing *global_reset_vector* to 'h1_0000, we successfully booted the chip in `_start` mode. Booting from `_start` prompted the chip to perform an AXI-4 request for the BBL from the memory module at the *DRAM_BASE* address 'h8000_0000.

Listing 1. _hang routine in bootrom.S.

```
_hang:
  f1402573    csrrs       a0, mhartid
  00000597    auip        a1, 0x0
  03C58593    add         a1, a1, 0x3C
1:
  10500073    wfi
  0000bff5    j           -0x4
```

Listing 2. _start routine in bootrom.S.

```
_start:
  0010041b      addiw   s0, zero, 0x1
  01f41413      slli    s0, s0, 0x1f
  f1402573      csrrs   a0, mhartid, zero
  00000597      auipc   a1, 0
  07458593      addi    a1, a1, 0x74
  00008402      jr      s0
```

The Berkeley Boot Loader (or another suitable boot loader) should be loaded at *DRAM_BASE* to set up the processor to load and run code, and then to load and execute the operating system.

Figure 2 shows the boot instructions on the instruction buffer module named *ibuf* and the resultant writes on the ABI registers. The base address is written to register 8 (*so/fp*, signal `FRAME_POINTER[63:0]`) and the pointer to the device tree blob is written to register 11 (*fa1*, signal `DEVICE_TREE[63:0]`).

On reset, the default *tethered* mode initializes the device then waits for an interrupt from an external module to send it instructions to execute. After modifying the global reset vector, our Build-0 boots in emulation and directly initializes the first-stage boot loader (FSBL) from the Boot ROM.

### E. Boot Sequence Execution Flow

Using the RISC-V specification v2.2, we disassembled the six instructions corresponding to the `_start` routine coming out of the boot ROM after reset. These include several writes we could verify on the ABI registers (Figure 2).

The first two instructions increment the s0/fp (saved register/frame pointer) register and left shift the base '1f (31 bits) to the most significant bits of that register, setting the frame pointer to the value of `DRAM_BASE` ('8000_0000). The third instruction `cssrw` (atomic read/write to command status register) writes zero to register 10 (*fa0*) to indicate the hardware thread that is executing. The fourth and fifth instructions (`auipc` and `addi`), write the program counter (set to 'h0001_0044) and add 'h74 to register 11 (*fa1*), changing it to 'h0001_0080 and setting up the address of the device tree blob. The last instruction jumps to the memory mapped address written to register 8 (s0/fp) Figure 3 illustrates this boot sequence as well as the bursts of BBL data from the memory that result from the jump to `DRAM_BASE` ('8000_0000).

### F. RISC-V Instruction Disassembly / Assembly

To make sense of the inner activity of the RISC-V, we had to understand the instructions appearing on the instruction buffer. Disassembling the instructions by hand became tedious and the version of *objdump* included with the RISC-V tools was only suited for disassembling binary files. We adapted an open source RISC-V disassembler [9] to meet our needs by porting the C code to Python and modifying it to disassemble individual instructions and perform on-the-fly instruction bit width adjustment. This let us compare code in the bootrom.img file, bootrom.S, and the *bootrom* module in the Verilog RTL (Listing 3).

Listing 3. Boot ROM generation in Verilog.

```
. . .
assign index = auto_in_a_bits_address[11:3];
assign high = auto_in_a_bits_address[15:12];
assign _T_673 = high != 4'h0;
assign _GEN_1 = 9'h1 == index ? 64'
    h597f1402573 : 64'h1f414130010041b;
assign _GEN_2 = 9'h2 == index ? 64'
    h840207458593 : _GEN_1;
. . .
```

### G. Boot ROM Generation and Device Tree Extraction

As shown in Listing 3, the boot ROM is hardcoded into the chip. The device tree code follows the boot ROM. From the Verilog code we were able to extract the boot ROM and device tree and compare it with the bootrom.img and bootrom.S files. The boot ROM and device tree match what is in the boot ROM in the Verilog.

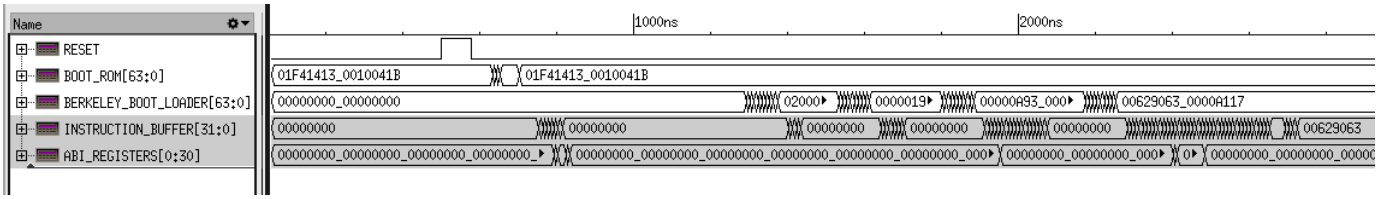


Fig. 3. Boot Sequence Response on Reset showing activity from the Boot ROM, BBL, and Registers

## IV. Connections and Writing out to Memory

Most of the modules contain ports that relate to the TileLink and AXI-4 connections in the chip. There is no documentation describing how the modules are connected or the specifics of parameter negotiation. The paper describing the diplomatic parameter negotiation [10] discusses the process in a general sense, but not how this theory is implemented in Rocket-Chip. In the RTL, we saw that each bus communicating outside the SoC goes through a sequence of modules that perform various tasks, such as:

- Indexing the message (TLIndexer),
- Fixing the width (TLWidthWidget),
- Converting between TileLink and AXI4 (TLtoAXI4),
- Fragmenting a burst into beats (AXI4Fragmenter),
- Disambiguating agents (AXI4UserYanker),
- Sequencing messages (AXI4IdIndexer),
- Buffering to wait for an enable (AXI4Buffer).

These modules and their use are not documented in the source code or in [10]. On the other side of each of these modules is a cross-bar that maps the sender and receiver modules. The MMIO connects from the Tile to a sequence of modules to convert from TileLink messages to AXI4 messages, to a cross bar, and then to a buffer before it goes out to the connected devices mapped on MMIO. The memory bus has communication from a filter and buffer, through a memory bus cross bar, and then prepares the messages for AXI4 to go out to the memory controller.

Each of the modules can have slightly different configurations of the same TileLink and AXI-4 ports. The AXI-4 relevant modules are given by the terminology used in the names of the module's ports, and how that relates to the AXI-4 protocol. There are no documentation or included test benches that determine the extent to which the TileLink and AXI-4 protocols are implemented. These are areas for future work in designing test benches and verification for the Rocket-Chip.

## V. Trust and Assurance

Our purpose in studying this chip is to learn how we can understand generated IP in general. While this is a start, there are many things we still do not know how to do. We know there are many potential vulnerabilities that can be hidden in the transformation steps [11]. Vulnerabilities include memory access patterns, race conditions, improper combinations of signals, improper control flow, leaking information, and control flow or data flow integrity issues. Protecting against these vulnerabilities requires the capability to perform security analysis on the chip with methods such as data flow analysis, address alias analysis, deadlock analysis, and so on. We plan to experiment with these techniques and others, but must first enumerate what is in the circuit, connect what is there, connect the design with the generated IP, and then understand how data flows between these components.

### A. Transformations

Our trust and assurance goals with the Rocket-Chip are to focus on the assurances cases. Transformation assurance is one such case. This involves verifying what is in the Chisel is represented faithfully in the Verilog. This includes nothing is inserted or removed during IP generation, and that everything in the RTL and physical design is traceable back to an intentional construct in the Chisel code. There is no longer a 1-to-1 correspondence between classes in Chisel and modules in the RTL. This makes it difficult to know what kinds of test benches need to be written. Test benches targeting the Verilog RTL miss the point of developing in the agile style described in [4]. However, test benches written in Chisel target a high-level model of the circuit, leaving the quality of the transformations unverified. Other verification work includes testing single module behaviors, testing module combinations, verifying sequences of signals in memory-holding structures, and tracing signal flow. To support this, one must understand the input space, behavior, and output space at each of these levels of testing.

### B. Barriers to Understanding Rocket-Chip

Besides the challenges of understanding the RTL, the Rocket-Chip source code is also complex and difficult to understand. The Rocket-Chip has several hundred thousand lines of complex, undocumented Scala code that relies heavily on concepts from the Chisel domain specific language [2] and FIRRTL [3] intermediate representation language. It relies heavily on deeply-nested object oriented relationships, advanced functional techniques like partial function application, delocalized behavior, and dependency inversion patterns. The code is only sparsely documented and most documentation is from space-restricted archival or conference publications. The learning curve factors heavily into the time and cost of any SoC project.

It is easy to make major changes to the Chisel source code that change the generated circuit in big ways, but it is difficult to predict the effects those changes will have on the RTL. It is not clear that the features that make Rocket-Chip's code complex are necessary to generating high-quality circuits. Any

complexity that is not required for the functionality of the generator, but which makes the RTL harder to understand, is a liability for anyone taping out an SoC using these tools.

The Rocket-Chip generator contains many more files besides just the Chisel source. A newly-cloned Rocket-Chip repository contains 372 files in 63 directories. The repository grows to 1,115 files in 282 directories after bringing in the required sub-modules (Chisel3, FIRRTL, HardFloat, and Torture); grows to 151,613 files in 8,625 directories after bringing in the required sub-modules from the RISC-V tools repository; and grows to 185,049 files in 10,780 directories after running the build scripts to create a first working project. The complexity of the build system also makes it difficult to understand everything that goes into Rocket-Chip, which also impacts the cost and schedule of projects that use it.

## VI. Related Work

Our work builds on and intends to further clarify the Berkeley team's own work such as the Rocket-Chip IP generator [1], Chisel [2], FIRRTL [3], diplomatic parameter negotiation for IP generation [10], and the paper on Chris Celio's Berkeley Out-of-order Machine [12], and the overall agile process these tools support [4]. We have found all of these sources to be incredibly useful.

Lo-RISC performed a similar reverse engineering effort to ours to develop their platform and provided a lot of documentation that was invaluable to us in our own work [8]. Researchers from the University of Washington also presented some of their work in understanding how to use and implement the Rocket-Chip [13]. With the amount of activity in Rocket-Chip and Chisel, we expect many more groups to use this IP to tape out their own SOCs and we expect each team will have to go through a process similar to what we have done to understand in depth how the various parts of the architecture work together.

## VII. Conclusions

We presented our preliminary work in understanding the Rocket-Chip RISC-V implementation for the purposes of improving our methodology in trust and assurance. Our goal was to understand how the various parts of the Verilog processor implementation fit together and connect when building up a system-on-a-chip, and to reverse engineer existing implementations for helping in design and answering questions. Our goal is to transition this platform into one for which we can build and test security and assurance features into the hardware we generate. As part of that, we must see how these work together and understand the impacts of these changes. For future work, there is a lot of need for improved visualization and understanding of hardware at various levels of abstraction. There seems to be a lot of promise in manipulating the FIRRTL description of the circuit, but there is a steep learning curve to work with that representation. Additionally, there is work that is needed in visualizations that help us easily see situations that create vulnerabilities or susceptibilities to attack.

## References


[1] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep., 2016.

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *Proceedings of the 49th annual design automation conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1216–1225. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228584

[3] P. S. Li, A. M. Izraelevitz, and J. Bachrach, "Specification for the FIRRTL language," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9, Feb 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html

[4] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic *et al.*, "An agile approach to building risc-v microprocessors," *IEEE Micro*, vol. 36, no. 2, pp. 8–20, 2016.

[5] "RISC-V Tools (GNU Toolchain, ISA Simulator, Tests)." [Online]. Available: https://github.com/riscv/riscv-tools

[6] "Linuxbrew: The Homebrew package manager for Linux." [Online]. Available: http://linuxbrew.sh/

[7] "yEd - Graph Editor." [Online]. Available: https://www.yworks.com/products/yed

[8] "lowRISC." [Online]. Available: https://www.lowrisc.org/

[9] M. J. Clark, "riscv-disassembler," 2018. [Online]. Available: https://github.com/michaeljclark/riscv-disassembler

[10] H. Cook, W. Terpstra, and Y. Lee, "Diplomatic design patterns: A TileLink case study," in *First workshop on computer architecture research with RISC-V*, Boston, 2017. [Online]. Available: https://carrv.github.io/2017/papers/cook-diplomacy-carrv2017.pdf

[11] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984. [Online]. Available: http://doi.acm.org/10.1145/358198.358210

[12] K. Asanovic, D. A. Patterson, and C. Celio, "The Berkeley out-of-order machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor," University of California at Berkeley Berkeley United States, Tech. Rep., 2015.

[13] T. Ajayi, K. Al-Hawaj, S. Dai, S. Davidson, P. Gao, G. Liu, A. Rao, A. Rovinski, N. Sun, C. Torng, L. Vega, B. Veluri, S. Xie, C. Zhao, R. Zhao, C. Batten, R. G. Dreslinski, R. K. Gupta, M. B. Taylor, and Z. Zhang, "Experiences using the RISC-V ecosystem to design an accelerator-centric SoC in TSMC 16nm," in *First Workshop on Computer Architecture Research with RISC-V*, 2017. [Online]. Available: https://carrv.github.io/2017/papers/ajayi-celerity-carrv2017.pdf