12

Investigation of RISC-V ISA using Rocket-Chip

Ian Swepston, Ashton Johnson

***Abstract*—RISC-V is a new and open RISC instruction set architecture (ISA). There are a series of repositories provided by The University of California, Berkeley to support the ISA help to design hardware. In this paper we investigate these tools and use them to learn more about RISC-V and RISC-V processors.**

***Index Terms*—RISC-V, ISA, Zedboard, Zynq, Rocket-Chip**

I. Introduction

THE RISC-V open instruction set architecture commissioned in 2010 has been making waves in the in world of open hardware within both the academic community and industry. As students pursuant of masters degrees in computer engineering, we investigate the RISC-V processor through venues that we are already familiar with: Field Programmable Gate Arrays (FGPAs).

The Rocket-Chip generator allows for developing a system-on-chip (SOC) design rapidly. [1]. We utilize the fpga-zynq [2] repository to target a Zynq-based Zedboard FPGA development platform to instantiate and test out a RISC-V implementation.

II. Project Goals

The goals of our project are to investigate the source material provided by UC Berkeley's Rocket Chip Generator. Specifically, we will attempt to replicate the successes of those listed in the fpga-zynq [2] repository. Goals are listed in order or complexity, but also different layers of the supplied design.

- Load Prebuilt Image
- Modify FPGA Project
- Modify RISC-V ISA
- Document Results

III. RISC-V Overview

RISC-V is an open risc instruction set architecture (ISA) commissioned in 2010 by a research team at UC Berkeley [3]. The stated goal of the team was to create a free and open ISA for commercial, educational, and open source settings. RISC-V is governed by the RISC-V foundation and is under the BSD (Berkeley Software Distribution) Licence, which is similar to the LGPL. The foundation is intended to maintain the ISA in the long term and guide it toward being an industry standard.

Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, 94720

Prof. Alexsander Milenkovic with the Department of Electrical and Computer Engineering, University of Alabama in Huntsville, Huntsville, AL 35899

RISC-V is not the first attempt at an open ISA, but RISC-V appears to be positioned to succeed where others failed. The RISC-V foundation is important. They can act in the same way a company, like ARM Holdings, and will continue to improve the ISA, but will not monetizing it. Free hardware designs are also helpful. Other open ISAs have failed because they were impractical to implement. Potential adopters needed to either design their own cores or try to implement untested designs. Finally, educators are eager to adopt this ISA. The possibility of free-to-use ISA and hardware designs would allow us to use real assembly and real hardware to teach architecture. This would create demand and would flood the design world with scientists and engineers that know the ISA and will choose to use it due to familiarity.

A. Design Benefits

- BSD License:
  Allows proprietary derivative work
- Scaleability:
  The ISA is largely microarchitecture-agnostic and flexible enough for most application workloads. It can be scaled to cover everything from microcontrollers to warehouse-scale servers.
- Code Compression:
  The ISA supports compression of its binary programs down to 16 bits per command without losing core functionality. This allows for significant reductions in code size for small scale embedded devices.
- 128 Bit:
  Comes with fully-functional 128-bit design. This does not appear to be commercially supported and may be of use soon in large-scale cloud computing applications.
- Software Support:
  There is a free software suite for application development. This will be discussed later.

IV. Pre-Built Image

It is possible to load a pre-built RISC-V core design onto a Xilinx Zedboard. The image is provided in the fpga-zynq repository. This image contains a basic implementation that allows the toolchain to be tested and for the execution of user programs, such as benchmarks. This configuration was investigated and a simple program was compiled using the toolchain and was executed on the core. We were able to compile and run the basic matrix multiplication program we developed in the early part of the CPE 631 class. It should be noted that this program utilized double precision floating points numbers.

A. Image Makeup

The pre-built configuration is a simple design that can be instantiated on the Zedboard. It is a single core design for the purpose of rapidly testing and running RISC-V in hardware. The basic design is an RV64G core. This means that it is a RISC-V (RV) 64 bit core servicing the General Purpose (G) extension set. The general purpose set can be thought of as a "normal" processor. We will cover the details in the Rocket Chip Generator section later.
Surrounding the core in this design is a split L1 cache for instructions and data. These caches are designed to be simple and save FPGA space. Both are 4-way set associative with 64 sets per cache and 8 TLB entries. They are non-error correcting, use the random policy for replacement, and do not use scratchpad memory to store calculations.
This design does not contain an L2 cache due to size and complexity, so after L1, the processor must go directly to main memory. For main memory, the design splits the 512MB of on-board Zedboard DDR memory in half, utilizing 256MB for its needs and the other half allocated to the ARM cores in the Zynq.

B. Toolchain

To complete this goal, cross-compilation was necessary. For this, Berkeley has developed and released the sources for RISC-V variants of the LLVM and GNU toolchains. These are included in the RISC-V repository under fpga-zynq/rocket-chip/riscv-tools. They are available as a standalone or as a link within the fpga-zynq project for Zynq boards. The toolchains can be installed several ways, but for performance and simple installation, we recommend using the provided build scripts in the riscv-tools directory. Instructions for installing the toolchain can be found in risc-gnu-toolchain repository [4]. Once built on the native system, the tools are named like: riscv64-unknown-elf-g++ and can be installed anywhere on that system. These toolchains appear to be complete and operate identically to their mainstream versions. As an example, compilation can be completed using the normal formats: [path to tools]/riscv64-unknown-elf-g++ main.cpp -o executable.

While the toolchains are being built, some additional useful tools are constructed:

1) Spike and pk: The executable for spike is built here. Spike is the RISC-V open source cross-platform simulator designed by Berkeley. It allows for execution of RISC-V binaries on an arbitrary platform through emulation. This is paired with a program called pk (proxy kernel). pk is designed to format and provide RISC-V instructions to either a kernel-less core or the spike program for execution and to receive and handle output from the core.

2) Benchmarks: Embedded in this repository are a series of RISC-V benchmarks compiled and ready to be executed on a system with a RISC-V core.

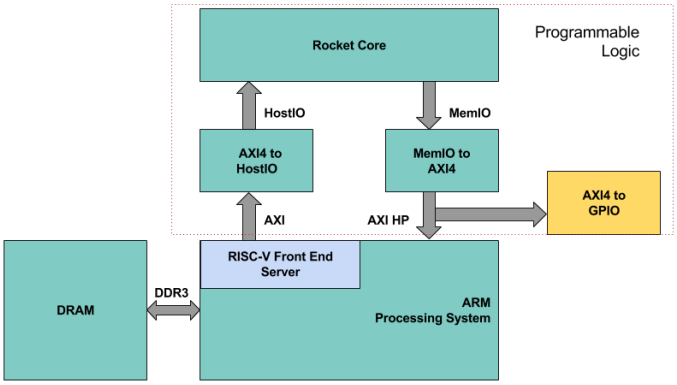

Fig. 1. Design with GPIO peripheral.

V. FPGA Project

Using the supplied makefiles, a Xilinx Vivado project was generated according to the the default Rocket Chip configuration. From this, we investigated the design that was targeting the Zedboard. On the Zedboard, the Rocket core interfaces with the ARM processing system via two AXI interfaces. One AXI interace is provides the ARM the ability to load user executibles into the rocket core. The other AXI interface provides the rocket core access to the DRAM that is on the zedboard. The memory side interface utilizes one of th AXI-HP ports on the Zynq Processing Module that is instantiated in a Block Diagram.

In Fig 1, we attached a Xilinx supplied AXI GPIO IP core. At the time, this method seemed to be the easiest for making a peripheral available to the RISC-V processor. The GPIO could then be accessed as a memory-mapped IO device using an address generated by the Xilinx block diagram. While we were successful in generating a new bitstream for the zedboard, and loading the design onto the board, we were not successful in actually manipulating the GPIO device. When running an executable using the Proxy Kernel [5], an error is reported of a 'user store segfault' at the address of the GPIO. This result was also confirmed using the Spike simulator. Upon researching further, there are similar efforts to attach peripherals to the processor as you might expect. We discovered that the lowRISC project provides an exampled on how peripherals should be added [6]. Using the diagrams they provided, we now understand that peripherals should be connected via non-cachable channels. This would require a unique AXI based bus from the processor that doesn't route through the memory interconnects from the L1 cache to the main memory. We were not able to pursue this further given the time

VI. Rocket Chip Generator

In order to rapidly produce prototypes of a design and to make RISC-V more accessible, the Berkeley team has created a system called the Rocket Chip generator. By using this tool, a hardware architect can have a RISC-V core up and running within minutes of downloading the appropriate repository and can begin to make changes as they are needed.

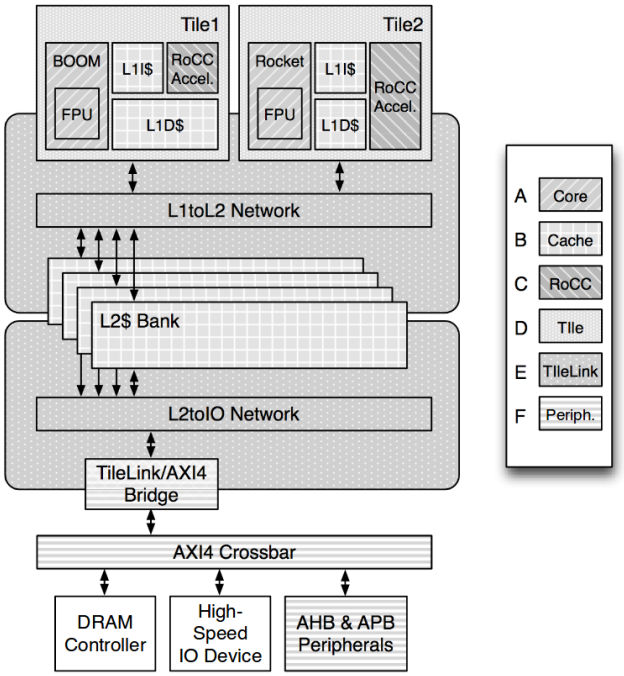

Fig. 2. Architecture that may be generated by the Rocket Chip Generator [1]

A. Chisel

The Rocket Chip generator relies on a Berkeley developed language extension of the Scala programming language, called Chisel [7]. Using Chisel, functional blocks of hardware are described for all the components required in the rocket core. Chisel is a new breed of hardware description language. Compared to traditional HDLs, such as Verilog or VDHL, Chisel can be considered a higher level language. It is syntactically similar to Java and employs many Object Oriented principles such as classes, inheritance and polymorphism. Within Chisel, hardware configurations are developed by instantiating base modules, which are treated like objects, and then including options by overriding the default parameters of each module. The hardware is described and connected hierarchically and upon evaluation the Chisel code generates Verilog which may be used in synthesis.

B. Generation Options

The Rocket Chip Generator is intended to be a do-it-all hardware design tool for hardware architects. As such, it is very flexible. Fig 2 is an example of an SoC created by the Rocket Chip generator and it demonstrates the power of the tool's sub-generators working together. For example, this instance could be generated and contains two separate, self-contained procesors connected by an interconnection network and L2 cache plus an AXI4 Crossbar with peripherals like a memory controller attached. The processor in Tile1 contains a small coprocessor and Berkeleys custom-made out of order RISC-V core (BOOM) and Tile2 has a general Rocket core for basic operations and a larger coprocessor.

C. Extensions

Extensions are a mechanism used by the chip generator to allow easy manipulation of the core and control the supported section of the instruction set. Rocket cores can be as simple or as complex as needed to complete tasks. Much of this complexity is due to the inclusion and exclusion of extensions. For convenience, many commonly used extensions are available to designers pre-written within the generator. The "G" extension is a combination of the extension for Integer (I), Multiply/Divide (M), Atomic (A), Floating Point (F), and Double Precision (D). This could be known as RV64IMAFD, but because these are so common, they are combined as G and referenced as RV64G [3].

VII. Rocket Chip Modification

The first step in understanding the generator is attempting to modify the design. This was our goal. Using chisel we successfully modified the number of cores instantiated in the design. This can easily be performed by modifying the following text in the file [path-to-repo]/rocket-chip/src/main/scala/rocketchip/Configs.scala to include the bolded text:

```
class DefaultFPGAConfig extends ++ new
WithNCores(2) Config(new FPGAConfig ++ new
BaseConfig)
```

While generating a dual core configuration was simple, it was not we could not implement this on the Zedboard because the resource utilization was too great. This was due to two reasons:

A. LUT Utilization

The processing coreplexes take up around 60 percent of the device LUT utilization. Therefore two core utilize 120 percent. Using a larger device could mitigate this problem.

B. Block RAM Utilization

When instantiating two or more cores, an L2 cache is automatically inferred. This is quite convenient; however, the default L2 instantiation allocates 2048KB of memory. This required around 1,200 percent of Block RAM utilization! To resolve this, we modified the previous snippet by adding the follow bold text:

```
class DefaultFPGAConfig extends ++ new
WithNCores(2) + new WithL2Capacity(256)
Config(new FPGAConfig ++ new BaseConfig)
```

To faciliate testing out the different configurations, we defined several classes beyond the `DefaultFPGAConfig` that can be instantiated. Their names are self-explanatory:

```
class DefaultFPGARoccConfig
class DefaultFPGA32KBL2Config
class DefaultFPGA256KBL2Config
class DefaultFPGADualCoreConfig
```

VIII. RoCC Accelerator

The Rocket-Chip generator provides a framework for develoeping custom co-processors that are housed in the processing coreplex, allowing for low latency access to the co-processor. These are the Rocket Custom Coprocessor (RoCC). These coprocessors utilize reserved opcodes in the instruction set. To get started with the accelerators, three example coprocessors are already defined in the repository. These are defined down in .../rocket/rocc.Scala. We were sucessful in instantiating these cores, and provided `class DefaultFPGARoccConfig` to make these easier to implmement. Resource utilization of the custom coprocessors is negligible.

We ran into difficulties when developing software for targeting the custom coprocessors. Documentation for how to target these is absent, so we were left to seaching user forums on the internet to see guidance. After much research, we were still not sucessful in using the custom opcodes. We discovered in previous iterations of the gnu-toolchain, the custome(0/1/2/3) opcodes were supported. Since then, the team has removed default support for the opcodes, and now require the use of pseudo opcodes [8].

IX. Notable Failures

Throughout this process, we experienced success; however, we experienced even more failures. We note these so as not to disguise, but to bring forward issues with the system, so that follow-on efforts might not be able to resolve them, or at least be aware of them.

1) Custom Opcodes: As mentioned previously, we were unsuccessful in utilizing custom opcodes, even those designed as pre-planned custom opcodes. While we were finally able to compile programs using the pseudo opcodes technique, the proxy kernel would not allow the execution of those instructions. They were designated as illegal opcodes. We suspect the proxy kernel would need to be recompiled to allow these instructions.

2) Deploying Linux: Through multiple paths we were not able to rebuild GNU/Linux and execute it on the processor. This included both on the development board and the Spike simulator. To give background to this, The Berkeley Bootloader (BBL), which is found in the riscv-pk repository [5], is an executable that is used to load in an ELF file into the processor. Most notably, this is used to boot GNU/Linux. When compiling the BBL, the Linux ELF file must be specified, otherwise and dummy ELF file is included. The output of the BBL build process is a single executable that contains the kernel.

We were successful in building the Linux kernel using the riscv-linux repository [9]. Additionally, we were successful in building a GNU/Linux distribution using the Yocto Pocky and the recipes provided in riscv-poky [10]. For Poky, we validated the build using the Spike simulator and BBL that is compiled at alongside the kernel. Unfortunately, moving that working configuration of the BLL and the kernel image and rootfs to the SD card did not yield success. Reflecting more on this configuration, it appears that version of the BBL does not require the kernel to be compiled into it. It seems to be an argument to it.

3) AXI Peripherals: We were not able to access a memory mapped I/O device. In this case a GPIO peripheral. Our test code was written to directly read and write to a pointer-address location (0x4000 0000). Yet again, the proxy kernel reported a segfault when reaching this step of the program. This restriction could be removed with recompilation of the proxy kernel, but this is only speculation.

X. Document Results

For documentation we are providing an Oracle VMWare disk image to reach each of the three goals listed above using the virtual machine provided. The disk image is an Ubuntu 16.04 distribution with all the required repositories and examples necessary to replicate our goals. We felt this was the best approach for allowing others to verify and replicate our results. Additionally, instructions for the virtual machine getting a new virtual machine up and running using the virtual disk provided. While this will not provided the fastest avenue for building the different items required, it provided a path with all dependencies removed.

XI. Conclusion

RISC-V is an open ISA designed to break open the world of computer architecture. It is making the push to be the first successful open ISA. The ISA is surrounded by many advantages including a software toolchain and a core design tool to generate inexpensive and reliable hardware. This paper documents our investigation into this ISA and its supporting tools. During the course of the paper we have discussed how we tested these tools utilizing the existing documentation from UC Berkeley for the purpose of improving upon it and understanding RISC-V.

We were able to load and test the pre-built designs in their repositories as well as complete their examples. We were also able to learn about the Rocket Chip generator tool and Chisel, its supporting language. Using these we were able to build a basic rocket core from source. Also using the generator, we made modifications to the core design and configuration and synthesized these cores. We also investigated adding a coprocessor (called a RoCC) and how to add it to a core design. We were able to synthesize this as well.

We had a few failures along the way. Many of these were due to the state of the repositories being in flux or due to venturing into underdocumented or undocumented pieces of the Rocket Chip generator and RISC-V. We believe that these failures can bring to light information that needs future investigation and documentation.

It is these authors opinion that RISC-V is poised for success. It is at a critical time in which it is being presented to the world, picked apart, and tested. As such we feel that researching and understanding the ISA and the tools that have been created to support it will help expand it use and usability.

Acknowledgment

The authors would like to thank the UC Berkeley Architecture Research team for the use of their materials and research. Thanks are also due to the RISC-V foundation and the open source community for their documentation of the RISC-V ISA.

REFERENCES

[1] K. Asanovi, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[2] "Fpga-zynq github repository." [Online]. Available: https://github.com/ucb-bar/fpga-zynq

[3] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The risc-v instruction set manual, volume i: User-level isa, version 2.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-118, May 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

[4] "Risc-v gnu toolchain repositry." [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain/

[5] "Risc-v proxy kernel repository." [Online]. Available: https://github.com/riscv/riscv-pk

[6] "lowrisc system on chip." [Online]. Available: http://www.lowrisc.org

[7] 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*. ACM, 2012, pp. 1216–1225.

[8] "Use of psuedo opcodes for custom instructions." [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain/issues/190

[9] "Risc-v linux kernel repository." [Online]. Available: https://github.com/riscv/riscv-linux

[10] "Risc-v yocto pocky repository." [Online]. Available: https://github.com/riscv/riscv-pocky

Ian Swepston Received his B.S. in Computer Engineering from Clemson University in 2015. He is currently pursuing a Master's Degree in Computer Engineering at the University of Alabama in Huntsville. He currently works as a Computer Engineer at Dynetics, Inc. in Huntsville.

Ashton Johnson received his B.E. degree in wireless engineering from Auburn University in 2011. He is currently seeking his Master's degree in computer engineering at the University of Alabama in Huntsville. He current works as an electrical engineer at Dynetics, Inc. in Huntsville.