Creating a RISC-V heterogeneous CPU architecture

Progress report

Matthew Knight

Department of Computer Science

University of Warwick

1 Introduction

The aim of this project is to design a heterogeneous SoC using the RISC-V open source core designs. Once designed, the SoC should be implementable on an FPGA and be able to execute bare-metal C code and assembly with SMP, demonstrating that all cores in the SoC can be used at the same time.

Designing a processor is a complex challenge. They need to be power efficient to reduce the cost of running the system, as well as providing adequate processing power when required. Often one is sacrificed to improve the other: for instance, energy efficiency is often reduced due in designs of large processors containing multiple cores. Due to the increased size and complexity, more power has to be supplied to achieve the same performance as a smaller processor.

A heterogeneous CPU attempts to solve this issue by combining dissimilar core designs; often a powerful core, or p-core, and an efficient core, or e-core. When only light processing is required, the p-core can be effectively shutdown and the e-core will do all processing, resulting in less power used. For heavy processing, the p-core is then used to increase peak performance. Depending on the exact implementation, the p-core can be used either individually or in tandem with the e-core, but both result in greater performance than just the e-core.

This has been used extensively in many mobile devices. ARM released the big.LITTLE in 2011[3], a mobile heterogeneous architecture. The architecture provides the low power usage needed in mobile devices the majority of the time when idle, only running tasks like checking for new messages. It also provides the high performance that can be demanded from phones when used to browse the internet, play mobile games, etc. Most big.LITTLE designs use HMP (heterogeneous multiprocessing) where all cores are available to have processes assigned to them at all times. The alternatives to this are: clustered switching, where either the big cores or the LITTLE cores are in use, and in-kernel switching, where big and LITTLE cores are paired to form a virtual core and only one performs the tasks assigned to the virtual core at any one time.

1.1 Related work

TODO: To be updated - mostly copied from specification

1.1.1 A RISC-V Heterogeneous SoC for Embedded Devices

A 64-bit RISC-V host core that offloads tasks to a PMCA (Programmable Many-Core Accelerator) made from 8 32-bit RISC-V cores. Implements two types of RISC-V core in a single SoC, but one is used as an accelerator for the other as opposed to them being of equal standing.

1.1.2 Muntjac multicore RV64 processor

Homogeneous RV64GC multicore. Linux-capable base to enable specialised designs. Implemented on Xilinx Kintex 7. Aims to be easily understood and extendable - a great starting point for development of specialised designs. Limited to only one type of core in base repository, but contains interconnect for other designs.

1.1.3 BlackParrot[2]

Homogeneous RV64G multicore. Linux-capable general purpose processor. Implemented on an IC at 12nm. A general purpose RISC-V multicore. Implicitly allows heterogeneous CPU by hiding the multicore components beneath an ISA layer and allows easy extensions by keeping the design tiny, modular and friendly. Heterogeneous designs are limited to those that share the exact same instruction set, so wouldn't allow RV32 and RV64 in a single processor.

1.2 Objectives

The overall aim is design a heterogeneous SoC containing two types of RISC-V core that can execute bare-metal C code and assembly with SMP. Objectives have been labelled according to the MoSCoW method to indicate their importance and expected completion.

1. Design a heterogeneous RISC-V SoC containing 2 dissimilar cores, each capable of executing at least the RV64 instruction set (Must).
2. Execute instructions on both cores when implemented in an SoC on an FPGA (Must).
3. Measure the performance of the SoC for comparison against single-core designs. (Should)
4. Run embedded Linux on the SoC and connect to it via serial or SSH (Could).
5. Allow processes to execute on both cores inside of embedded Linux (Could).
6. Intelligently select which core the process will run on, depending on factors such as process priority and resource usage (Won't).

2 Research

2.1 Learning Chisel

As the HDL that is used in RocketChip[], the underlying SoC generator that this project utilises, being able to write and understand Chisel to at least a basic extent is necessary. The Chisel Learning Journey[] is a step-by-step guide to quickly becoming effective at writing Chisel and producing effective code. This has been incredibly useful in understanding the vivado-risc-v project and RocketChip generator, though has required a large amount of time. I have not yet fully completed the guide, but have achieved a basic proficiency with Scala that has made creating new SoC designs possible, even if no complex topics have been covered.

2.2 Existing Project

The documentation for the existing project is minimal at best. There is adequate to generate existing designs and implement on an FPGA after a few

hours of reading and compilation, but little exists after this. This is somewhat expected - the project is open-source, but primarily for usage by the maintainers. A lot of the time spent has been in understanding the existing project, as well as creating a development environment that matches what is required.

2.2.1 RocketChip

Much time has also been spent in reading RocketChip documentation[]. As the SoC generator used by this project, understanding it's capabilities is key to effectively using it. The documentation is very sparse and mostly points to that of the sub-repositories which are used extensively in the generator. Code analysis of the RocketChip generator has been the most effective way to learn about the generator, but this has not been entirely successful. The code is often uncommented and assumes a great deal of knowledge about CPU design, as well as the RISC-V ISA, making reading it very difficult. Due to the niche

2.3 RISC-V ISA and Assembly

There is excellent documentation on the RISC-V ISA, hosted by the RISC-V foundation. There are 2 documents, the unprivileged specification and privileged specification. The unprivileged specification details instructions that can be executed in U-mode, or user mode - when user programs are executing. The privileged specification details instructions that are executed in M-mode, or machine mode and are intended to be called by the OS or bootrom/bootloader. The documents describe all instructions and registers on RISC-V cores, but do not contain examples or context for many of them. In addition to this, there is huge variance between cores - parts of the RISC-V ISA are optional and are implementation specific, describing what behaviour should be seen as opposed to how much of it is done.

There are also few sources for writing RISC-V Assembly code. TODO: write more about RISC-V assembly and why im going to be using it.

3 Progress

TODO: add a lot of references

3.1 Component Resourcing

It was quickly identified that a larger FPGA was required in order to achieve the full project aims. The current Artix A7-100T [] has 15,850 slices, each of which contain 4 LUTs for a total of 63,400. This is inadequate, as the SoC requires 10,800 LUTs + 27,500 LUTs per "big" RV64 rocketcore, preventing more than 1 "big" RV64 rocketcore being implemented on the FPGA.

The other option for a larger core is the Sonic BOOM RV64 cores. These have a much larger LUT requirement, with the "medium" BOOM RV64 core using 148,500 LUTs, which is clearly not possible to implement on the current FPGA.

Alternative boards are the Nexys Video and Genesys 2. The Nexys Video contains 134,600 LUTs and would allow for multiple "big" rocketcores to be implemented, but would be unlikely to fit any BOOM cores. The Genesys 2 contains 203,800 LUTs and would allow for a BOOM and "big" rocketcore to be implemented at once. Due to monetary reasons, the Nexys Video was chosen as the FPGA to be sourced. The board is currently out of stock from University sources, so no progress has been made towards purchasing it.

3.2 SoC Design

An initial SoC design has been created, consisting of a large and small core. This is using the rocketchip[1] RV64 cores, and are generated using a Chisel description of the processor, with the large core being the "big" rocketcore and the small core the "small" rocketcore. The difference between these are changes to the instruction set extensions implemented and the size of the instruction and data caches. This change reduces the core footprint and LUT utilisation on the FPGA significantly, from an estimated 27500 LUTs to only 7600 LUTs. This is a significant reduction, though comes with the cost of reduced functionality in the smaller core.

The "big" core implements the RV64IMAFDZicsrZifenciC instruction set extensions. The IMAFDZicsrZifenci are commonly shortened to G, and are instructions for integer multiplication and division, atomic instructions, single-precision floating-point arithmetic, double-precision floating-point arithmetic, control and status registers and instruction-fetch fence. The final extension, C, is for compressed instructions, allowing the width of an instruction to be just 16 bits instead of the standard 32 for certain instructions.

The "small" core reduces this to RV64IAZicsrZifenciC, removing multiplication and both types of floating-point arithmetic. This reduces the capability of the core significantly, and means programs compiled for the "big" core that utilise the extensions unique to the "big" core will not run on the "small" core. This does not follow the original project idea exactly, as the cores were to be equal/very similar in ability to execute instructions, but dissimilar in total processing power due to other factors like pipelining, cache, superscalar, etc. However, this has been unavoidable due to the inability to source a new FPGA in the time frame.

3.3 Software

3.3.1 Linux

The "big" RV64 rocketcore is the minimum required to run mainstream Linux, as smaller designs forgo the memory-management unit (MMU). When present, the MMU[] is responsible for the transfer for data between the registers and memory, as well as ensuring only valid/safe memory addresses are used. When an MMU is not present, there are no checks on the memory that a program is accessing. This is a massive security issue if allowed in the OS, especially in a mainstream OS that could be attacked by malware. As such, mainstream Linux is not compatible with a processor without an MMU[] and therefore not with any processor containing a core smaller than the RV64 "big" rocketcore. As the Nexys A7 FPGA supports a single "big" rocketcore, Debian Linux[4] was successfully run on an SoC design containing a single "big" rocketcore. This demonstrates that a processor containing cores equal or larger than the "big" rocketcore would be able to run mainstream linux.

The MMU requirement blocks the objective of running the RISC-V Debian Linux distribution on a custom heterogenous SoC, as the larger Nexys Video FPGA has not yet been sourced and these is no indication of when it will be. As such, the aims for software to run on the produced SoC have been reduced to bare-metal, with the stretch objective of embedded linux in the future. Embedded linux would not require an MMU, but would provide an extremely small amount of features when compared to Debian Linux.

3.3.2 Bare-metal software

Bare-metal code has successfully been written, compiled and run on the FPGA with various SoC designs implemented on it. A design containing a single "big" rocketcore was implemented on the FPGA, and an SD card inserted into the board containing an ELF executable that was then run by the SoC after boot. The code outputted text over a serial connection that was then read by the host computer, and identified the 'hart' (hardware thread) of the core.

This was followed by attempting to run on the heterogeneous design previously mentioned. The only serial output received only identified a hart of 0, indicating only a single core was running the ELF executable. Investigation of the default bootrom, assembly code that sets up the core before user code is run, revealed that only hart's with an id of 0 started running user code - else they were trapped in a loop of waiting for an inter-processor interrupt to indicate they should execute code in a region of memory stored in a local register. The solution for this is to generate an interrupt and load the register with the location of the ELF, or to adjust the bootrom so that harts always branch to user code.

The current bare-metal software has all been written in C, and then cross-compiled using GCC for RISC-V. Rust is also a candidate language for bare-metal RISC-V programming[], and comes with certain benefits that improve usability, such as good type checking and memory safety. Changing from using C to Rust is currently being considered, and will be tested.

4 Project management

TODO: chart of what has been achieved and a gantt chart of what will be achieved in the future, split into roughly 1/2 week chunks The progress made so far has been mixed - some objectives have been achieved much faster than was expected, while other objectives have taken a significantly longer amount of time than expected. This has been tracked effectively using the Trello[] board, and allowed for the project supervisor to be updated frequently as to how the project has been progressing.

References

[1] K. AsanoviÄ, 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. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[2] M. W. D. C. J. S. D. P. G. C. Z. Z. A. S. C. B. V. T. G. A. J. J. M. O. M. B. T. D. Petrisko, F. Gilani. Blackparrot: An agile open source risc-v multicore for accelerator socs. *IEEE Micro*, 2020.

[3] A. Ltd. big.little processing. `https://web.archive.org/web/20121022055646/http://www.arm.com/products/processors/technologies/bigLITTLEprocessing.php`, 2011.

[4] K. M. Manuel A. Fernandez Montecelo. Risc-v - debian. `https://wiki.debian.org/RISC-V`, 2021.