Design and Implementation of a RISC V Processor on FPGA

Ludovico Poli
Embedded and Intelligent Systems Laboratory
University of Essex
Essex,UK
lp17239@essex.ac.uk

Sangeet Saha
Embedded and Intelligent Systems Laboratory
University of Essex
Essex, UK
sangeet.saha@essex.ac.uk

Xiaojun Zhai
Embedded and Intelligent Systems Laboratory
University of Essex
Essex, UK
xzhai@essex.ac.uk

Klaus D. Mcdonald-Maier
Embedded and Intelligent Systems Laboratory
University of Essex
Essex, UK
kdm@essex.ac.uk

2021 17th International Conference on Mobility, Sensing and Networking (MSN) | 978-1-6654-0668-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/MSN53354.2021.00037

***Abstract*—**

The RISC-V ISA is becoming one of the leading instruction sets for the Internet-of-Things and System-on-Chip applications. Due to its strong security features and open-source nature, it is becoming a competitor to the popular ARM architecture. This paper describes the design of a light weight, open-source implementation of a RISC-V processor using modern hardware design techniques, the implementation of the design onto a Field Programmable Gate Array (FPGA), and its testing. We wanted to create a RISC-V processor that is easy for beginners to learn from and lightweight enough to be implemented on even small FPGAs. While there are existing opensource implementations of RISC-V processors, none are intuitive enough for a beginner to follow. For this reason, in this paper we have minimised the use of conventions and components in modern processors that are not strictly necessary for a barebones implementation. For example, the processor does not include pipelining and uses a simple Harvard architecture. The barebones nature of the design allows for a lot of potential for upgradability. The implementation of each component, and the corresponding test benches, are written in concise and conventional System Verilog. The project produced a RISC-V processor with files for targeting Basys 3 Artix-7 FPGA. Performance was tested using the Dhyrstone benchmark and achieved a strong 2.276 DMIPs/MHz, even outperforming the ARM Cortex-A9, while maintaining very low resource utilization on the FPGA.

Keywords—RISC-V, CPU, Verilog, FPGA, open-source

I. INTRODUCTION

Instruction set architectures (ISA), are standards for how the hardware of a processor should function and interact with the ISA's own assembly language. ISAs have all the information necessary to create a processor that will run machine code correctly and consistently according to the standard [1]. This includes information about instructions, registers, memory access, arithmetic, data buses and so on. While every function of the processor is defined in the standard, how hardware and circuits are implemented is left up to the designer. Most ISAs will typically have many extensions and variations to suit the requirements of different designs. This could include extensions with support for multiplication, floating point numbers, or different data, address and instruction widths, or ISAs targeted for embedded systems, personal computers, super computers, etc.

Commercially successful ISAs have typically been proprietary and required licenses. With the specific designs and implementations being hidden behind patents and non-disclosure agreements. The new RISC-V ISA [2] is promising for the hardware industry with it being the first open-source modern ISA. RISC-V is a combination of work by the University of California and companies such as AMD, Google, Microsoft, IBM and many more. Since the finalisation of the first variants of RISC-V, many open-source tools and designs have been created and published [3].

RISC-V is not only exciting because it is open source, but also an ISA that can compete with the other restricted commercial ISAs by Intel, AMD and ARM. While most academic designs are optimised for learning purposes, RISC-V is intended for modern practical use.

The open-source aspect of RISC-V made it particularly well suited for this project as we could use it to create an open-source CPU without having to use one of the "toy" or academic ISAs, and instead use something more realistic and practical. While there already exist opensource implementations of RISC-V processors, none is simple and intuitive enough for a beginner to follow.

978-1-6654-0668-0/21/$31.00 ©2021 IEEE
DOI 10.1109/MSN53354.2021.00037

In this paper, we have designed a light weight, open-source implementation of a RISC-V processor using modern hardware design techniques and implemented the design onto a Field Programmable Gate Array (FPGA), and to test it. We wanted to create a RISC-V processor that is simple enough to learn from and lightweight enough to be implemented on even small FPGAs that students could afford.

While there are already several open-source implementations of RISC-V processors, they are only intended for practical usage. The consequence of this is that they are often too complex for a student to learn from. The commonly used PicoRV32 by Claire Wolf [5] is considered to be a very compact CPU, but even this uses complex periphery such as XIP SIP Flash Controllers, UARTs, GPIO controllers, and the RV32IMC variable length instruction ISA. The Berkeley CPUs (Rocket, BOOM and Sodor [6, 7, 8]) have the difficult entry requirement of being designed using the unique and rarely used Chisel [9] hardware design language. SCR1 by Synthacore [10] is considered to be industry grade with the core including 2 to 4 stage pipelining, debug support, AXI4/AHB-Lite external interfaces, machine and many other powerful features. There are many instances of processors with this level of complexity, and so there has been a rising demand for RISC-V processors that can be used as a learning example.

To make the design accessible, in our CPU we neglected all those conventions and components in modern processors that are not strictly necessary for a barebones implementation. For instance, the processor does not include pipelining and uses a simple Harvard architecture which reduces complexity. Also, the implementation of each component, and the corresponding test benches, are written in concise and conventional System Verilog [4].

II. Methodology

A. The Instruction Set Architecture

The ISA used in this project is a reduced version of the RV64I extension. RV64I only supports 64-bit integers and the instructions included are based on the minimal instruction set from *"Computer Organization and Design RISC-V Edition: The Hardware Software Interface."* [11] The exact seven instructions are presented in

Table 1.

Table 1 Instruction set

Instruction	Description
ld	Load double integer
sd	Store double integer
add	Addition
sub	Subtraction
and	Bitwise AND operation
or	Bitwise OR operation
beq	Branch if equal

ld and **sd** are two memory related instructions. **ld** takes a double (64-bit value) from a given memory address and puts it into a register. While **sd** does the opposite and takes a double from a register and stores it at an address in memory.

add and **sub** are two arithmetic instructions. **add** takes the contents of two source registers, adds them together using signed addition, and stores the results in a destination register. **sub** subtracts the contents of the two source registers instead. If the result of an arithmetic operation overflows (i.e., the result cannot fit in a double integer), then the result will wrap around back through the negative numbers. As all arithmetic is signed, the negative numbers are represented in two's complement.

and and **or** are the two logical instructions in the instruction set. **and** takes the contents of two source registers, performs a bitwise AND operation, and stores the results in a destination register. **or** does the same but performs a bitwise OR operation on the source registers instead.

beq is the only control instruction (a conditional instruction). beq stands for *"branch if equal"*. It takes the contents of two source registers and verifies if they are equal by checking if their difference by subtraction is zero. If they are equal, then the program will jump to a new instruction, based on a signed offset provided by the **beq** instruction. Normally the program counter register will increment through each instruction one at a time with each new clock cycle. However, when **beq** is used, the program counter can be incremented or decremented by a specified amount, jumping the program forwards or backwards by some number of instructions.

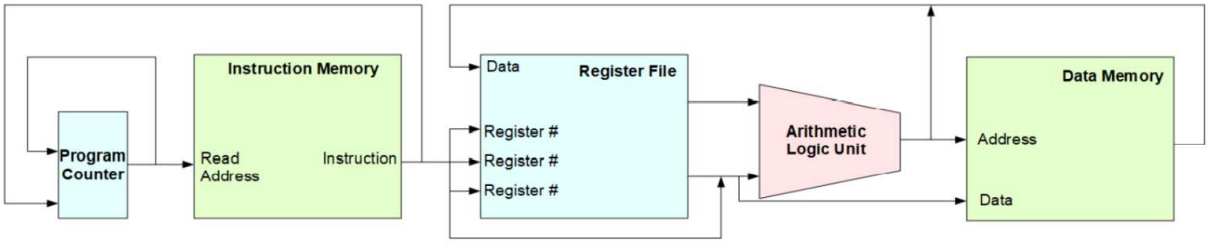

Figure 1 Processor Architecture

B. System design

The top level of the processor is based on the theory from *"Computer Organization and Design RISC-V Edition: The Hardware Software Interface."* [11]. Each component is implemented as a System Verilog module with a focus on the code being conventional and highly readable. Each component includes a simple test bench written in System Verilog so that the behavior of the module can be tested in simulation. The entire system is designed so that each instruction is executed in one clock cycle (no pipelining). The main components can be seen in Figure 1 (the control unit is not included for simplicity as it connects to every other main component with control signals).

The instruction memory is a read-only memory (ROM) that is flashed with the program instructions when the design is uploaded onto the FPGA. The size of the memory is parameterizable within the System Verilog code of the instruction memory. The width of each memory location is 32 bits so that they can each contain exactly one 32-bit RISC-V instruction. The instruction memory can be flashed by providing the compiler with a ".mem" file; a space separated value file.

The register file contains the 32 general-purpose registers, with each register being 64-bit wide. For the sake of simplicity, this deviates slightly from the RV64I ISA standard as usually certain registers are dedicated to holding specifics values (stack pointers, frame pointers, etc). It has two read address inputs and two corresponding asynchronous read data outputs. This is useful so that two registers can be read from in the same clock cycle, like in the case of adding two registers together. It has a single write address input and a synchronous write data input; this means that any register that is written to will only be updated with the new values on the next clock cycle. Writing is enabled with a control signal from the control unit. Register #31 is tied to the output pins so that it can be used as a debug register output for better observability.

The ALU performs four different operations on its inputs and output the result. It can perform addition, subtraction, logical AND and logical OR. Which operation is performed is determined by the control signal coming from the control unit, which in turn is determined by the instruction type. Naturally, the ALU is used by the **add**, **sub**, **and**, **or** instructions. However, also the **ld**, **sd** and **beq** instructions also require a calculation to be performed (such as the calculation of memory addresses). The ALU also has a "zero flag" output, which is a single-bit output that goes high when the result of an operation is zero. This is used during the **beq** instruction execution to decide whether to branch or not.

The data memory is an asynchronous read, synchronous write memory used for storing data generated by the program or data flashed during power up. It has a single address input shared during read and write operations. There is a write data input and a read data output. There are also control signals for enabling write and enabling read separately. The write data input is provided by the contents of one of the read register outputs of the register file. The read data output goes to the register file's write data input. The address input is provided by the ALU output since the address for reading and writing needs to be calculated by adding a register from an instruction's source register field to the offset field. Additionally, a small region of the data memory is read only and flashable on start-up which is useful for a programmer since the small instruction set doesn't provide any immediate instructions. This can be flashed with the same method as instruction memory by using a ".mem" file.

The control unit is the most important component as it sends control signals to all the other components so they can coordinate with each other. It takes the opcode field of an instruction (see Figure 2) as the input and uses a lookup table to send the corresponding signals to the components. We chose to use a purely combinational control unit, rather than a microprocessor with microcode, since the instruction set is small and does not require the additional layer of abstraction provided by microcode.

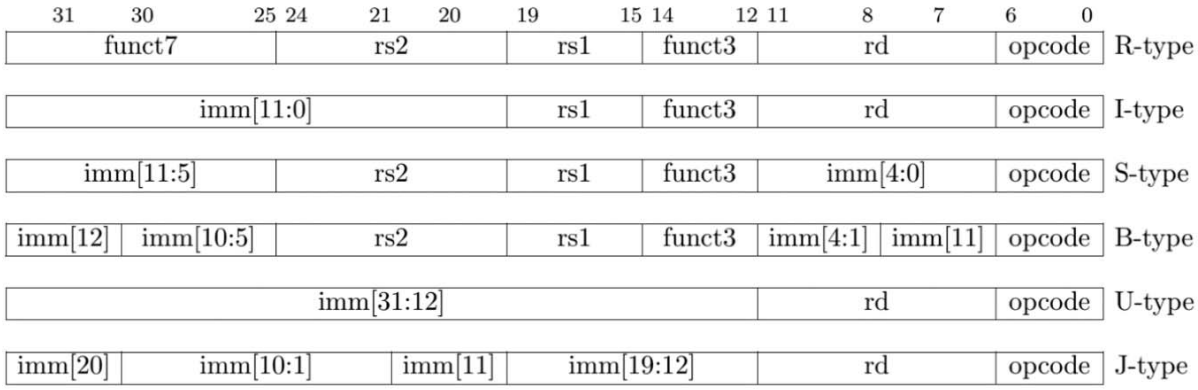

Figure 2 Instruction format

A fundamental design principle of the processor is that each instruction would be executed in one clock cycle. This is because it makes many aspects of the design simpler (e.g., no pipeline registers are required) and because getting high clock speeds is not a priority for the project. A design consequence of this is that any logical path in the design can only have one clocked process. In the design there are only 3 clocked processes: the program counter incrementing, the synchronous write of the register file and the synchronous write of the data memory. Even though the latter two processes are on the same logical path, they cannot happen at the same time due to the control unit never asserting both write enables high in the same clock cycle.

C. Turing Completeness

While the instruction set of our CPU is much reduced, it is possible to prove that it is Turing complete. An instruction set/language is Turing complete if it can implement a Turing Machine or if it can implement the instructions of a language that has already been proven to be Turing Complete.

We use the second method to show Turning Completeness. In particular, we want to show that our assembly code can implement the instructions of the 8 simple instructions of the Brainfuck language [14], which is Turing Complete. The instructions are shown in Table 2 Instructions in the Brainfuck language and their C equivalent. with their equivalent C programming language implementation.

Table 2 Instructions in the Brainfuck language and their C equivalent.

brainfuck command	C equivalent
(Program Start)	`char array[30000] = {0}; char *ptr = &array[0];`
`>`	`++ptr;`
`<`	`--ptr;`
`+`	`++*ptr;`
`-`	`--*ptr;`
`.`	`putchar(*ptr);`
`,`	`scanf(" %c",ptr);`
`[`	`while (*ptr) {`
`]`	`}`

For each of the C equivalent statements one can create an equivalent with our CPU assembly language:

- **(Program Start)** We make sure we have RAM (>30K) memory preallocated. We devote a register (let us call it **P**) to act as `ptr`. The CPU already initialises all registers to 0.
- **'<'** We can use **add** to increment **P**
- **'>'** As above but for **sub**
- **'+'** We use **ld** to load the data stored where **P** points, then **add** to increment and finally **sd** to store the result back
- **'-'** same as above but using **sub**
- **'.'** We reserve part of the RAM (e.g., above address 30000) to write output. We use another register, **O** (for output), to point to that area. We use **ld** to load the data stored where **P** points, then **sd** to store the result where **O** points, and finally we use **add** to increment **O** (so future output is not overwritten)
- **','** We reserve part of the ROM to get inputs from. We use another register, **I** (for input), to point to that area. We use **ld** to load the data stored where **I** points, then **sd** to store the result where **P** points, and finally we use **add** to increment **I** (so it points to the next input data, for future input operations)
- **'['** and **']'** We implement the while loop by simply devoting a register, **Z** (for zero), to containing the constant 0, then adding a few instructions before and after the body of the loop. At the beginning we use one **ld** to load the data stored where **P** points, then **beq** to compare such data with **Z** and jump forward to the end of the loop if the condition is met. At the end of the body, we do **beq Z Z**: because you are comparing two things that are identical, the instruction will always cause a jump. So, we can jump back to the conditional instructions just before the body of the loop.

D. Physical Implementation

Synthesizable designs can either be sent to a manufacturer to be printed onto an Application-specific Integrated Circuit (ASIC) or it can be programmed onto a Field Programmable Gate Array (FPGA), the latter being what was used in this project. FPGAs are programmable digital circuits that can have their logic reconfigured to match the HDL code [5]. The main appeal of FPGAs, compared to traditional circuit printing methods, is that digital circuits can be prototyped and changed quickly, easily and with no added cost. A processor put onto an FPGA, rather than an ASIC, is known as a soft processor.

The board we used is the Basys 3 Artix-7 FPGA Trainer Board [12]. This a board suited for educational purposes as it offers lots of I/O (switches, buttons, LEDs, displays, etc.) and examples to use online. The main drawback of this board is that high performance and high clock speeds are difficult to achieve, which is an acceptable compromise given the aims of this project. The design tool that was used in this project is the Vivado Design Suite [13], which handles design synthesis, implementation and programming the FPGA.

III. RESULTS

Once the CPU was fully designed in System Verilog and tested successfully in simulation, two tests were run on the FPGA implementation described in *II. Methodology D. Physical Implementation*. The first involved running a very

small program to test the FPGA utilization of the processor (how "light weight" it is). The program simply calculates the Fibonacci Sequence and stores the sequence in the debug register using six assembly instructions, with one instruction being executed per clock cycle:

```
ld r0, 33, r1
add r1, r2, r31
add r2, r0, r1
add r31, r0, r2
add r31, r0, r2
beq r0, r0, -2
```

The instruction memory ROM, data memory ROM and data memory RAM were parameterized to be 256 bits each in depth. Using these parameters, the implementation was able to be optimized to achieve very low utilization as show in table Table 3.

Table 3 FPGA Utilization Metrics

Resource	Utilization	Utilization %
Look-up tables	322	1.55
Flip-Flops	229	0.55
IO	18	16.98

The second test was to run a standard benchmark on the processor. The benchmark we used is the industry standard Dhyrstone benchmark by Reinhold P. Weicker [15] Dhyrstone is a synthetic benchmark that measures the non-floating-point performance of a CPU using a realistic set of operations. The program itself is written in simple C and the main program loop is small which makes it well suited for small processors. Dhyrstone performance is measured in DMIPs and is usually normalized for the clock speed using DMIPs/MHz. Using a clock speed of 100 MHz, our processor performs very well against other common CPU as seen in Figure 3.

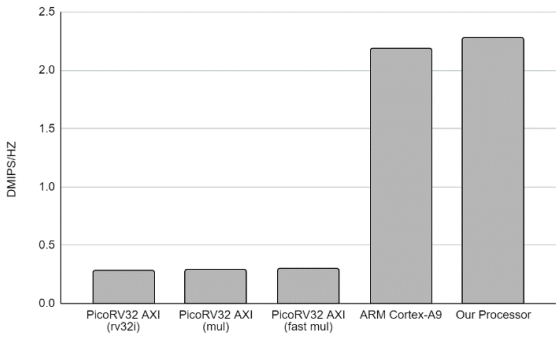

Figure 3 Dhyrstone Benchmark Performance

One of the reasons our processor performed so well might be due to the small instruction set, and hence the small amount of logic, means that it can be optimized more significantly by the compiler. It could also be the fact that the memory of our chip is completely on-chip and doesn't require slower memory interfaces to do memory accesses. The small amount of FPGA utilization as shown *Table 3* might also allow the compiler to optimize instruction memory ROM into combinational logic which could potentially increase the performance significantly.

IV. Conclusions

This paper presented the design methodology, lightweight, and open-source RISC-V processor. By sticking to only implementing what we considered to be the essential theory, we were able to create a system design we believe to be simple and intuitive for students to learn from. Due to the minimalistic instruction set, the logic required is small and so the utilization on hardware is very efficient, which achieves the objective of being lightweight. In addition to this, the Dhyrstone performance could allow for potential practical and industry uses even if the processor was not originally intended for this application.

Due to our RISC-V instruction set being so minimal, it is not currently supported by common RISC-V toolchains. So, a natural progression to this project would be providing support for assemblers and compilers. The design hardware also has a lot of intentional room for upgrades with the possibility of variants being created with pipelining and support for additional instruction sets such as RV64IM.

A. Source

The System Verilog design files and source can be found at *https://github.com/Ludini1/minimal-risc-v-cpu.*

B. Acknowledgements

This work is supported by the UK Engineering and Physical Sciences Research Council through grants EP/R02572X/1, EP/P017487/1 EP/V000462/1 and EP/V034111.

REFERENCES

[1] David A. Patterson and John L. Hennessy *"Computer Organization and Design RISC-V Edition: The Hardware Software Interface."* Page 22 Morgan Kaufmann Publishers Inc., 2017, San Francisco, CA, USA.

[2] Andrew Waterman, Krste Asanovi, and Five Inc. *"The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2."* CS Division, EECS Department, University of California, Berkeley 2017

[3] RISC-V *"RISC-V GNU Compiler Toolchain"* Github 2021 https://github.com/riscv/riscv-gnu-toolchain/

[4] IEEE *"IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language,"* IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) 2018

[5] Claire Wolf *"PicoRV32 - A Size-Optimized RISC-V CPU"* GitHub 2019 https://github.com/cliffordwolf/picorv32

[6] CHIPS Alliance *"Rocket Chip"* Github 2021, RISC-V International https://github.com/chipsalliance/rocket-chip

[7] Christopher Celio *"The Berkeley Out-of-Order RISC-V Processor"* Github 2021, University of California https://github.com/riscv-boom/riscv-boom

[8] Christopher Celio, *"The Sodor Processor Collection"* Github 2021, University of California https://github.com/ucb-bar/riscv-sodor

[9] Bachrach et al. *"Chisel: constructing hardware in a Scala embedded language"* Proceedings of the 49th Annual Design Automation Conference (DAC 2012). San Francisco, California, USA

[10] Synthacore *"SCR1 RISC-V Core"* GitHub 2021 https://github.com/syntacore/scr1

[11] David A. Patterson and John L. Hennessy *"Computer Organization and Design RISC-V Edition: The Hardware Software Interface."* Morgan Kaufmann Publishers Inc., 2017, San Francisco, CA, USA.

[12] Digilent *"Basys 3 Reference Manual"* reference.digilentinc.com 2021 https://reference.digilentinc.com/reference/programmable-logic/basys-3/reference-manual

[13] Xilinx *"Vivado Design Suite - HLx Editions"* Xilinx.com 2021 https://www.xilinx.com/products/design-tools/vivado.html

[14] Brian Raiter *"Brainfuck: An Eight-Instruction Turing-Complete Programming Language"* muppetlabs.com 2013, http://www.muppetlabs.com/~breadbox/bf

[15] Alan R. Weiss *"Dhrystone Benchmark History, Analysis, "Scores" and Recommendations"* ECL, LLC, 2011, El Dorado Hills, California, USA