

# CPTR380 RISC-V RV32I FPGA Implementation

Jaron Brown & Ethan Jansen

**Abstract**—**TBD**

**Index Terms**—ISA, RISC, RISC-V, RV32I FPGA, VHDL.

## I. INTRODUCTION

**S**INCE designing a custom ALU-based CPU last quarter for the Intro to Digital Design class, both of the authors developed an interest in implementing a more commonly supported instruction set architecture (ISA). When presented with the opportunity to pursue a project related to computer architecture in their Computer Architecture class, both authors jumped at the opportunity to implement another CPU design. They initially considered implementing the MIPS ISA, which was being studied in class, but, after conducting more research, decided on implementing one of the RISC-V standard ISAs, RV32I. This decision was made based on the abundant documentation for the RISC-V ISAs, RV32I's relative simplicity, the high commercial interest in these ISA specifications, and the interest in supporting open-source projects.

One of the dominant reasons many companies are developing RISC-V implementations is its open-source nature. The dominant ISAs (x86, ARM, MIPS, etc.) all have steep licensing fees. The RISC-V ISAs, in comparison, are completely free. Additionally, the closed-source nature of the dominant ISAs limit the freedom those designing implementation of these ISAs. Having a completely open-source ISA just makes sense—it lowers the barrier to entry for designing hardware to implement standard ISAs and allows for more innovation in the realm of CPU design.

## II. A BRIEF INTRODUCTION TO RISC-V

The original design of RISC-V was begun in 2010 at the University of California, Berkeley. It was designed not only as an academic learning aid but also a set of ISAs that could have commercial viability. As its name suggests, RISC-V is a RISC (Reduced Instruction Set Computer) architecture. This means, in comparison to CISC (Complex Instruction Set Computer) ISAs, RISC ISAs have simple instructions. This means more individual instructions will be required to complete a given task compared to CISC ISAs but also allows the hardware implementation to be much simpler than that for CISC ISAs. In fact, implementations of x86-64, the most widely adopted CISC ISA, breaks its CISC instructions into "microinstructions" which are easier to implement in hardware, similar to RISC ISAs.

RISC-V is not actually *an* ISA but rather a collection of ISAs, ranging from 32-bit to 128-bit standards and supporting various levels of complexity by way of extensions to the base ISAs. Some official extensions include the M extension which

adds multiplication and division support, the A extension which adds support for atomic instructions, and the F and D extensions which add support for single- and double-precision floating numbers, respectively [1]. Developers are encouraged to design their own compatible extensions to customize the base RISC-V ISA to their specific application.

The RV32I ISA defines four main instruction types, with an additional two instruction types which are subsets of the main instruction types. They are as follows:

- R-type: Supports integer register-register arithmetic operations
- I-type: Supports integer register-immediate arithmetic operations
- S-type: Supports store instructions
- SB-type: Supports branch instructions
- U-type: Supports load upper immediate (LUI) and add upper immediate to pc (AUIPC) instructions
- UJ-type: Supports jump and link (JAL) instructions

In this project, revision 2.1 of the RV32I unprivileged ISA will be implemented in VHDL and tested on a Xilinx Artix 7 FPGA (Field-Programmable Gate Array). RV32I is the RISC-V 32-bit base integer ISA and implements the core functionality of the RISC-V architecture family [1]. Using this, the implemented microcontroller should easily support C code cross-compilation via clang. This will help test/debug the microcontroller during implementation. Even more notable, however, is this will greatly improve the accessibility of this project as a useful processor.

### A. Stretch Goals

The following goals would all improve the design, however are not necessary to meet a basic RV32I implementation. Because of time constraints these are items are not actively being pursued.

- I/O: I/O support is not part of RV32I. Because of this, an extension, or a co-processor, would be needed for proper implementation. For these reasons, proper I/O implementation is a stretch goal. However, because basic I/O support is required for testing/debugging, it will be unofficially implemented.
- Pipelining: Although RV32I supports pipelining, this project's implementation will not implement this feature due to projected time constraints. Pipelining is not necessary to meet ISA requirements, and the lack of pipelining will only increase the clock cycles per instruction (CPI) but will greatly simplify the control logic and data path unit (DPU).
- Multiplication & Division: These are technically part of the M extension [1] and thus are not high priorities. However, because `IEEE.numeric_std` (the VHDL

library) has built in support for multiplication and division this would not be difficult to add to the existing arithmetic logic unit (ALU). The ALU would simply require another bit of control.

- Standard C Code Compilation Support: Basic C support is planned for this project, however extensive support is currently not guaranteed.

The authors intend to continue supporting this project by refining the design and implemented the discussed stretch goals.

### III. IMPLEMENTATION

#### A. Overview

This project's implementation of non-pipelined RV32I consists of a few core components:

- instruction & data memory (separated)
- a register file
- an ALU
- various other DPU components related to sign-extension, incrementing the program counter, branching, and routing signals
- a control module

Unlike the authors' previous CPU designs which used an accumulator-based design, this implementation uses a register file with 32 32-bit wide registers. Besides the different ISA support (the original accumulator-based CPU used a custom ISA), the switch to using a bank of registers is the most significant hardware design change between the two implementations. This change, however, is in keeping with the RV32I standard and allows for more flexibility when programming the CPU.

#### B. Instruction & Data Memory

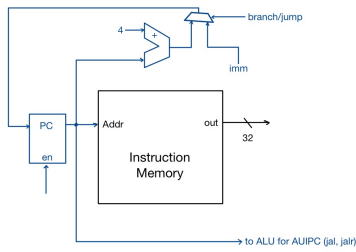


Fig. 1. Instruction Memory & Program Counter with Preliminary Step & Branch Support

Each memory (instruction and data) block was implemented in VHDL using AMD's write-first memory example using an array of `std_logic_vectors` **needs citation**. This allows for quick implementation; furthermore, it ensures that the memory blocks will be mapped to a physical FPGA memory block for improved performance –the clock will not have to be further delayed as it may have been otherwise.

One change from AMD's example that needed to be made was support writing bytes and half-words for data memory. This is to support the `sb` and `sh` instructions as they do not overwrite the upper bits. Without the additional support baked

into the data memory, an additional register would need to be added to temporarily hold the upper bits so that they do not get overwritten—this would require additional clock cycles to implement.

The instruction memory has been implemented and holds a maximum of 1024 32-bit instructions. Because the implementation does not support all of RV32I's 32-bit ( $2^{32}$ ) addresses, the lower ten bits are masked and are used as the instruction memory address. The current implementation of instruction memory is untested. Currently, data memory is not implemented due to time constraints. When implemented, data memory will hold 1024 32-bit wide values and will support read and write. As per RV32I specification, data memory will support storing/loading word, halfwords, and bytes. **Incomplete..**

#### C. Register File

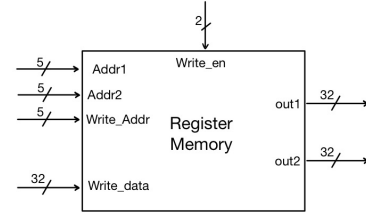


Fig. 2. 32-bit Register File

The implementation of the register block fully supports the RV32I specification. Register `x0` is writable but, when read, is always a vector of zeros and the other generic registers (`x1-x31`) function according to the RV32I specification. The control signals to the register block are write-enable and enable. These control signals are important as they lock the register block so nothing changes (enable) or, when enable is high, allow writes to the registers (write-enable). Locking the register block so nothing changes (using write-enable) is important in this non-pipelined implementation where the values of different components need to remain stable after being updated by an instruction for the remaining clock cycles required to complete that instruction. The register block is implemented using a write-first memory block which is an array of `std_logic_vectors`, similar to how the memory is implemented. Custom logic is implemented to allow for two registers to be read at once.

This implementation has been tested using the VHDL testbenches built into ISE. Tests include reading and writing to registers, reading and writing to register `x0`, and writing to and reading from the same register.

#### D. ALU

The arithmetic logic unit (ALU) is a fully combinational block which will perform the desired arithmetic operation based on the input control lines. Shown in Table III-D are the main operations the ALU is required to support. Because

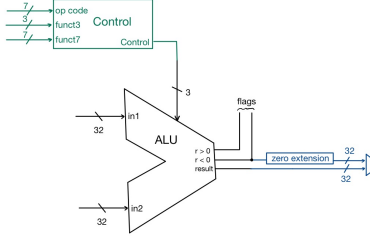


Fig. 3. ALU with ALU Control and Control Flag Output

there are various instructions associated with the same arithmetic operation (such as needing to determine the address used for load and store using ADD) the control line into the ALU is only 3-bits wide in order to support the eight different arithmetic operations, but the actual control signal is determined from a separate "ALU Control" block rather than directly from the instruction. In addition to arithmetic operations, the ALU performs comparisons between the two inputs, which are used for branch conditions and for SLTI and SLTIU. These comparisons are made visible through four flags: unsigned less-than and greater-than and signed less-than and greater-than. With these four flag outputs any of the signed and unsigned branch conditions can be supported:

- BEQ: equals
- BNE: not equals
- BLT & BLTU: less-than
- BGE & BGEU: greater-than

Furthermore, SLT, SLTU, SLTI & SLTIU (set less-than) instructions can be supported by selecting between the signed and unsigned less-than flags (using a mux), zero-extending the one-bit value, and selecting it as an output (via another mux) instead of the ALU's arithmetic result. The set less-than instructions set the least-significant bit (LSB) of an otherwise zero 32-bit vector to 1 or 0 depending on if the first input is less-than/greater-than the second input, respectively.

The ALU has been tested using the VHDL testbenches built into ISE. All the core arithmetic functions have been tested. Additionally, the four comparison flags have been tested in all their combinations. One might expect full coverage testing of the flags to require  $2^4$  tests but, since the flags are not independent, the total number of required tests is reduced to five.

This ALU Control block takes in `opcode`, `funct3`, and `funct7` from the encoded instruction and translates this into the necessary 3-bit ALU control signal. Separating this block also allows more easily decoding the necessary information from the six different [1] instruction types.

#### E. DPU Miscellaneous

TBD

#### F. Control

Incomplete...

The control is primarily separate from the DPU (apart from a few components such as the ALU control). This nicely

Operation	Instructions
ADD	ADD(I), Load & Store (various), JALR, AUIPC
SUB	SUB, SLT (various), Branch (various)
XOR	XOR(I)
OR	OR(I)
AND	AND(I)
Left Shift	SLL(I)
Right Shift	SRL(I)
Right Shift (Arith.)	SRA(I)

TABLE I  
RV32I BASE ARITHMETIC OPERATIONS MATCHED TO THE INSTRUCTIONS

separates the control from the rest of the CPU processing and makes a nice division in the design. We were able to do this by not considering pipelining, thus removing any forwarding registers and branch prediction modules from being interweaved into the DPU.

However, because there is no pipelining, the controller needs to be aware of how long each instruction takes to execute. This is mostly separated into the six different instruction types—where each instruction within an instruction type takes the same number of clock cycles to execute. For the current implementations, the instruction length is hardcoded into the controller. For instance, the controller will wait four clock cycles for an ADD to execute before continuing to the next instruction.

The controller also has support of pausing and single-stepping. This greatly improves the debugging of the device. This is useful not only during hardware design of the CPU but also when writing software to run on the CPU. Currently, however, there is no proper I/O support, so there are dedicated buttons on the FPGA board to control the single-stepping. Thus, single-stepping can not be controlled via a software debugger. This is planned for future work.

#### IV. TESTING & DESIGN REVIEW

To verify this project's RISC-V implementation is valid, a few methods can be used. First of all, the individual modules must be tested in isolation. This can be done one of two ways: implementing testbenches in either VHDL or Python using either simulation testbenches or by creating testing wrappers which provide inputs to the modules and display the outputs on either the LEDs, seven-segment display, or digital pins on the FPGA board. Two compelling options for implementing testbenches are utilizing the VHDL testbench tools built into ISE or using cocotb. Testbenches are definitely the preferred way to test since they cut out some of the opportunity for error that comes from setting switches and reading LEDs. Additionally, they allow for automated pass/fail tests and provide timing diagrams, which simplify debugging significantly compared to testing on hardware. The VHDL testbench tools built into Xilinx ISE are relatively simple to use and auto generate the component imports. Additionally, the ISE testbench tools provide timing diagrams for the tests, which was found useful for this project. cocotb is a testbench suite which allows users to define testbenches in Python.

It has many compelling features but the time required to setup and learn the tool was prohibitive. Since both authors are already familiar with Xilinx ISE for its synthesis and FPGA configuration tools, they opted to use the ISE testbench tools. The ISE VHDL testbenches were fairly simple to learn and using them for component testing saved the authors a significant amount of time.

## V. CONCLUSION

TBD

## COMPLETE CPU BLOCK DIAGRAMS

TBD

## VHDL IMPLEMENTATIONS

Sample: TBD due to current overfull issues...

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity program_counter is
  port
  (
    clk      : in std_logic; --! Clock
    reset    : in std_logic; --! Reset
    → to 0
    en       : in std_logic; --! Enable
    br       : in std_logic; --! Branch
    jump_value : in std_logic_vector(11
    → downto 0);
    count     : out std_logic_vector(10
    → downto 0)
  );
end program_counter;

architecture counter_signed of
  → program_counter is
    signal count_buf : signed(11 downto 0);
  begin
    process (clk)
    begin
      if rising_edge(clk) then
        if reset = '1' then
          count_buf <= to_signed(0, 12);
        elsif br = '1' then
          count_buf <= count_buf +
            → signed(jump_value) - 1;
        elsif en = '1' then
          count_buf <= count_buf + 1;
          if count_buf(11) = '1' then
            count_buf <= to_signed(0, 12);
          end if;
        end if;
      end if;
    end process;
  end process;
```

```
count <=
  → std_logic_vector(unsigned(count_buf(10
  → downto 0)));
end counter_signed;
```

## REFERENCES

- [1] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*, pp. i–30, Dec 2019. [Online]. Available: <https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf>