

The RISC-V REPL is a Read Eval Print Loop for RISC-V - an open source Instruction Set Architecture (ISA)[4] - hosted at github.com/harrysarson/riscv-repl.

This project allows execution of individual RISC-V instructions via a terminal application. When an instruction is entered, it is compiled into machine code which is sent via UART to an FPGA where it is executed. Once the FPGA has executed the instruction it sends the register file back over UART where it is displayed in the console.

1 State of the Art

Assembly REPL's which allow execution of native instructions can be found online. `rappl` supports linux systems whilst Windows users can use `WinREPL`. Both use similar approaches, creating a separate process and editing the binary whilst it is running. The instructions entered are (after assembling) run directly by the processor.

Additionally, RISC-V emulators such as `tinyemu` or the `sunflower-simulator` allow RISC-V programs to be run on a desktop computer by simulating execution of the instructions in software[1][5].

RISC-V processors have been written in verilog with the aim of running on an FPGA, two examples are `PicoRV32` and `RV32I_iCE40`[7][6].

2 Approach

To create the RISC-V REPL I chose to use the `RV32I_iCE40` verilog modules as they are designed to run on the same FPGA (the iCE40 Ultra Plus) and because I knew I could seek help from Ryan Voo to get started.

`RV32I_iCE40` has support for transmitting bytes over UART from software using memory mapped registers. As UART would be used for loading instructions and writing back the register file, the ability to write to UART from software was removed. Additionally, modules to receive bytes over UART were added into the project. The modules were sourced from FPGA-peripherals - the same place as the transmitting modules already included[3].

When not executing an instruction, the processor clock is held high. Then, once an instruction is received, that instruction is sent to the processor and the processor clock is made to complete one cycle. The processor has a five stage pipeline and thus four more cycles are required to fully execute the instruction. The clock cycled four

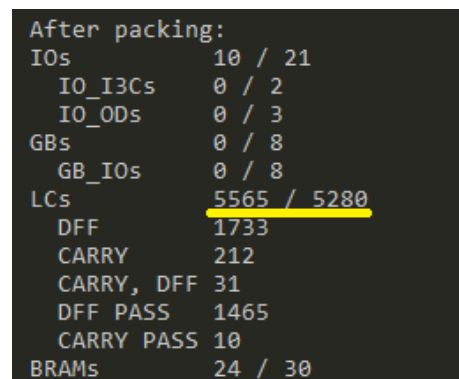
more times whilst no-op instructions are sent to the processor. Finally, the register file is read and transmitted. Figure 3 shows a timing diagram of this process.

Further technical details can be found in the documentation directory of the RISC-V REPL repository.

3 Running in Simulation

During this project verilator was used to test the design. Verilator is a verilog to c++ compiler and by running of the processor in a software testbed, debugging information can be printed to the terminal. Additionally, compiling the simulations is much faster than synthesizing a hardware layout and flashing to an FPGA. The simulation allows connecting the FPGA UART to a TCP port, this is used to connect the user interface to the processor when it is running in simulation.

Moreover, the iCE40 Ultra Plus FPGA used in this coursework contains a limited number of resources. Figure 1 shows that the current implementation of the RISC-V REPL processor requires more logic cells than the FPGA has available. It is likely that a better HDL implementation of the processor would be able to cut down the number of logic cells needed. The method of writing the register file is particularly clunky - a 1014 bit vector wire is assigned to the bits of the registers and this vector is transmitted one byte at a time over UART. Unfortunately, efforts to reduce the number of logic cells required were fruitless and, due to time constraints, were abandoned. With a better understanding of verilog and some more time, it is believed that a synthesizable version could be obtained.



After packing:	
IOs	10 / 21
IO_I3Cs	0 / 2
IO_ODs	0 / 3
GBs	0 / 8
GB_IOs	0 / 8
LCs	5565 / 5280
DFF	1733
CARRY	212
CARRY, DFF	31
DFF PASS	1465
CARRY PASS	10
BRAMs	24 / 30

Figure 1: The synthesis requires more logic cells than are available.[2]

4 REPL User Interface

The processor provides an UART interface, this project includes a terminal application written using Node.js. I chose to use node as I knew I could use the `serialport` package to interface with the FPGA.

The user interface must assemble RISC-V instructions into machine code. To use gcc as a RISC-V assembler it must be built as a cross compiler which was done by including the Sunflower-toolchain as part of this project[5]. Instructions entered into the REPL can be converted into a valid assembly file as shown in listing 1.

Listing 1: Create assembly from single instruction.

```
const createAssembly = instruction => `
.globl _start

_start:
    ${instruction}
`;
```

A small Makefile uses the cross compiler to generate both machine code and disassembly from an assembly file. My node script reads an instruction from the terminal, converts it into an assembly file, writes this file to disk, executes `make` and finally reads the disassembly

and machine code from the disk. The disassembly is displayed to the user and the machine code is sent to the processor. As figure 4 shows, if the user enters invalid RISC-V assembly, then the error message generated by gcc is shown to the user.

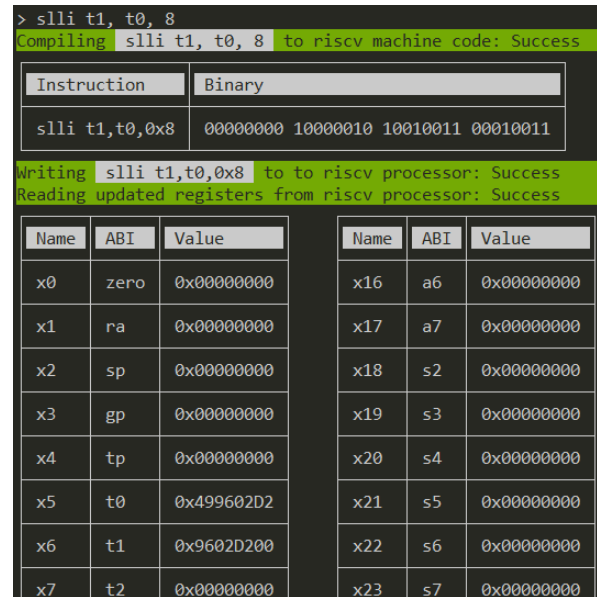


Figure 2: Screenshot of RISC-V REPL showing execution of the logical shift left immediate instruction.

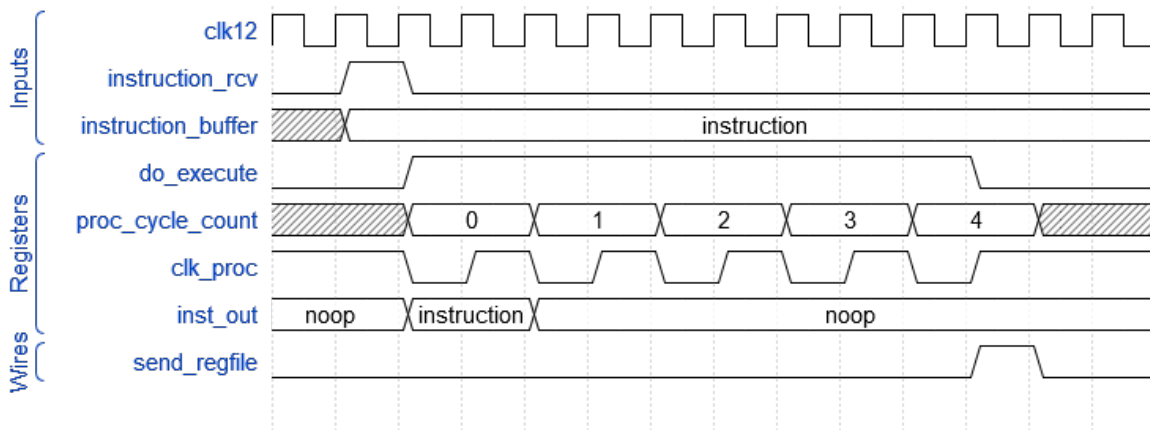


Figure 3: Timing diagram, the processor clock (clk_proc) cycles five times to pass the instruction through the pipeline. inst_out is the instruction fetched by the processor at each rising edge of the processor clock.[2]

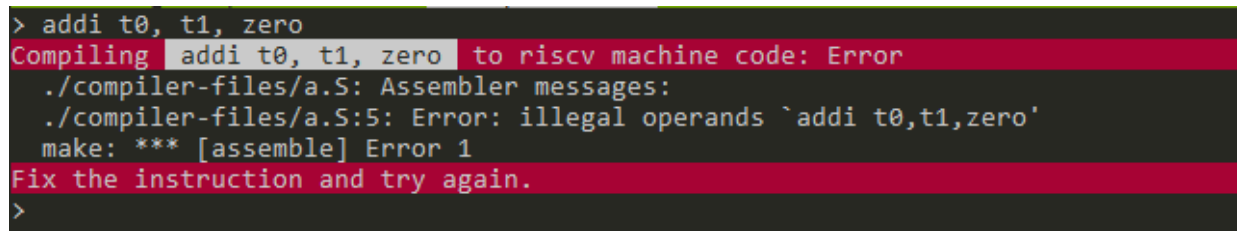


Figure 4: Error generated when an invalid instruction is entered.

References

- [1] Fabrice Bellard. 2018. URL: <https://bellard.org/tinyemu/L>.
- [2] Aliaksei Chapyzhenka. 2019. URL: <https://wavedrom.com/>.
- [3] FPGAwars. 2017. URL: <https://github.com/FPGAwars/FPGA-peripherals>.
- [4] RISCv Foundation. 2019. URL: <https://riscv.org/>.
- [5] Phillip Stanley-Marbell and Diana Marculescu. “Sunflower: Full-system, Embedded, Microarchitecture Evaluation”. In: 2007.
- [6] Ryan Voo. 2018. URL: https://github.com/physical-computation/RV32I_iCE40.
- [7] Clifford Wolf. 2018. URL: <https://github.com/cliffordwolf/picorv32>.