IV. Implementation

We implemented a RISC-V soft core processor on the Zybo Z7 development board (Zynq XC7Z010) in Verilog. It is a simple CPU capable of running C programs compiled with gcc-riscv. It's capable of referencing a .mem file compiled into the hardware design, or booting off of an SD card with the program on it.

A. CPU

Our base CPU design was generated using the CoDAL architecture description language, and compiled using Codasip Studio. Codasip is a processor technology company with the goal of increasing the accessibility of processor design by allowing processor description in CoDAL, a high level language, and supporting the generation of RTL (VHDL, SystemVerilog, or Verilog).

The base CPU design is a 5-stage pipelined processor with data, structural, and control hazard prevention. It was adapted from University of Colorado Boulder (CU Boulder) undergraduate Computer Organization class. The pipeline stages are as follows:

1) Fetch (IF): In the fetch stage the CPU utilizes the current program counter (PC) to request information from that memory address from instruction memory. The PC can come from 4 possible sources:

- The prior PC+4, normal operation
- An unconditional branch instruction (jump)
- A conditional branch instruction, predicted by the branch predictor
- The PC of the interrupt handler

The first 3 sources were supported by prior work, and our work extends this to implement an interrupt handler.

2) Decode (ID): The decode stage reads the values sent back from instruction memory, decodes the source and destination registers, and reads the values out of the source registers. It also decodes the operation code (opcode) and sends this information on to the next stage in the pipeline. In the case of dependencies, previous instructions' values can be bypassed from the execute, memory, and write-back stages.

3) Execute (EX): The execute stage executes the ALU operations, makes a branch prediction, and sends signals out to the main memory unit. In the case of dependencies, data can be bypassed from the memory or write-back stages and used as an input of the execute stage.

4) Memory (ME): The return signals from the main memory unit are read back. In the case of dependencies, previous instructions' resulting data can be bypassed from the write-back stage.

5) Write-Back (WB): The write-back stage updates the register file with any and all results from the execute and memory stages.

B. Memory

The code from prior work communicates with independent instruction (instr) and main (ldst) memories. This communication is done over the AHB-lite interface. We leverage code from RoaLogic for both the instruction and main memory instances [1]. Instruction memory has 16 KB, and main memory has 32 MB. The connections between instruction memory and main memory are completely independent, and controlled entirely by the CPU during normal operation.

C. Initialization

Prior to operation the instruction memory must be populated with the machine code. Although RoaLogic's AHB memory allows for an initialization file to be built into the design, the Verilog code must be recompiled for every new program. We wanted to create a general purpose CPU, and elected to support compiled designs loaded onto an SD card or hardware-compiled code. If no SD card is found, the program built into the bitstream will be executed.

A top-level state machine has two states - initialization and runtime. In initialization the instruction memory is first cleared, then code is loaded from either source (SD card or verilog-defined) and copied into the instruction memory via the AHB3-Lite bus. After the instructions are loaded the state machine transitions to the run state, resets the CPU, and begins execution. If the **BUTTON 1 / RESET** is pushed on the FPGA while in operation the state machine re-enters the initialization state. This allows users to remove the SD card while the program is running, copy a new program, then restart the soft core processor.

1) SD Card: To load RISC-V instructions into the CPU memory, we implemented a simple SDIO controller that will read machine code from a binary file stored on a micoSD card. The SDIO protocol extends the standard SD card interface to allow for read and write communication with embedded devices. This is the protocol that comes standard on the Zybo Z7 development board and thus the most straight-forward way to implement the SD card functionality.

To create our SD card controller, we utilized Wang Xuan's SD Card reader [2]. Xuan's code aims to support several versions of SD cards including SDv1.1, SDv2 and SDHCv2 as well as both FAT16 and FAT32 file systems and various clock frequencies. The controller establishes communication with the SD card, navigates the specified file system, and retrieves data from the file. We will be using a [TBD - SDHC probably] microSD card with a FAT32 file system and [TBD] Hz clock.

D. Hardware Integration

We also memory map input/output (IO) devices on the development board to specific memory addresses, as shown in Table 1.

We implement a custom interrupt controller which handles input interrupts from any of the general purpose input peripherals, as well as a timer. The interrupt controller detects interrupt signals from any of these sources, and instructs the CPU to fetch the PC of the interrupt handler next. Our custom hardware design intercepts any stores/writes to the general purpose output memory locations and creates the corresponding control signals out to the peripherals.

TABLE I
MEMORY-MAPPED IO

Peripheral	Use	Memory Address	Bit Mask
LED 1-4	General Purpose Output	0x00001000	0x1 << LED#
Button 1	Reset Input	0x00001004	0x1 << BTN#
Button 2-4	General Purpose Input	0x00001004	0x1 << BTN#
Switches 1-4	General Purpose Input	0x00001008	0x1 << SW#
Timer	Timer Inputs	0x0000100C	None

The timer is configured by default to generate an interrupt every millisecond. It can be configured to create interrupts every 100 microseconds to once per second. The range of values it supports is [TBD] and acts as a multiplier where [TBD] is the base value. Any invalid configuration of this register will result in the default timer configuration.

E. Software

The soft-core CPU supports the RV32I instruction set, and supports any code compiled compiled against this ISA. We suggest one of the following options:

- Hand-written Assembly converted to machine code. We suggest using gcc to convert a handwritten .s file into machine code. instructions TBD
- C-code. We suggest compiling with gcc to get the machine code. instructions TBD

The output of any of the above methods will result in a binary file, suitable for compiling into the Verilog design (after converting to file extension .mem) or loading onto the SD card (directly as a binary file).

V. METHODOLOGY

VI. RESULTS

VII. CONCLUSION

ACKNOWLEDGEMENTS

We'd like to thank Tamara Lehman for the original CPU design in CoDAL.

REFERENCES

[1] RoaLogic, "Ahb-lite memory," https://roalogic.github.io/ahb3lite_memory/.
[2] W. Xuan, "Fpga sd card reader," https://github.com/WangXuan95/FPGA-SDcard-Reader/blob/main/.