# EECS 151: FPGA Lab
# Final Project Report

## RISC-V CPU and Audio Synthesizer

**By Sukrit Arora and Rafael Calleja**
**[Team 30]**

## Project Functional Description and Design Requirements

The objective for this project is to create a RISC-V CPU with a 3-stage pipeline (Instruction Fetch and Decode, Execute, and Memory Access / Write Back). This processor is able to handle a subset of the instructions detailed in the RISC-V Green Card that accounts for data and control hazards. The CPU has access to four distinct pieces of memory: BIOS, for storing the base program, IMEM, for storing flashed programs, DMEM, for storing CPU calculated data, and MMIO, for peripheral interaction,. The CPU is also connected to a user I/O interface and synthesizer using memory mapped I/O. The user interface involves reading both button and switch inputs on the FPGA board and writing to the leds on the FPGA board. The synthesizer is a monophonic subtractive synthesizer that is constructed using a numerically controlled oscillator and is capable of producing four types of tones (sinusoidal, sawtooth, square, and triangular). The audio output of the FPGA is achieved using a PWM DAC that operates at a frequency (125 MHz) much higher than the CPU clock (50+ MHz).

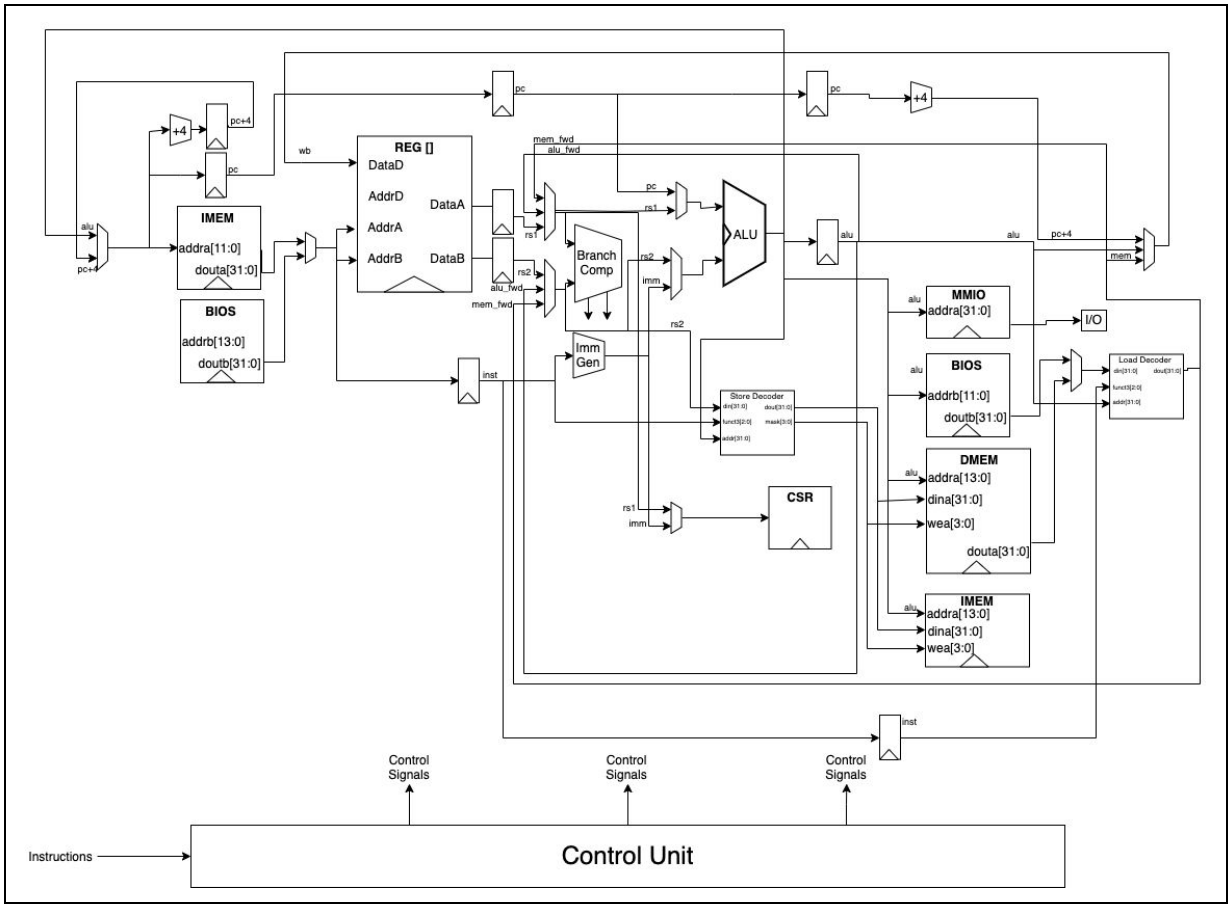


Figure 1: The high-level datapath of the 3-stage RISC-V pipeline.

# CPU Modules

Every large, nameable piece of the CPU (i.e. ALU, Control Unit, etc.) is broken down into submodules. This practice of grouping each large, independent piece and smaller, reusable pieces of the CPU into modules allowed for very readable and reusable code. In the end, our `Riscv151.v` file contains almost nothing but wires and modules.

## RISCV151-core:

- **Control Unit**
  The Control Unit is the brain of the CPU. It takes in every instruction across the pipelined stages as inputs and produces the control signals that dictate the flow of data through the datapath. It computes logic for forwarding (data/control hazards), mux selection (ASel, BSel, ALUSel, etc), branching and jumping and its corresponding NOP injections, and memory and regfile write enabling.

- **ALU**
  The ALU is simple and implements specific mathematical operations useful for different instructions. Our ALU takes in an ALUSEL and the current rs1 and rs2 values and outputs the calculation based on these inputs. In order to map the opcode to the specific instructions there is an `alucode.vh` file that defines constant we could refer to and use in multiple files.

- **Branch Comp**
  The Branch Comp is probably the simplest piece of the CPU. It takes in a control signal BrUn (whether the comparison is signed or not) and the current rs1 and rs2 values, and outputs BrEq and BrLt, the results of the comparison, to the control unit.

- **Regfile**
  The Regfile is a 2D register that holds 32 bits of data for each of the 32 registers. It is synchronous write and asynchronous read so that we can write and read on the same clock cycle.

- **Immediate Generator**

The immediate generator does exactly what it sounds like it would do: generates the immediate from the instruction, and outputs the one corresponding to the control signal from the control unit.

- **Register**
  The Register module is a simple, clocked register. We created this submodule in order to make the resulting Verilog code more readable and avoid having several `always @(posedge clk)` littering our code. This was useful not only for the pipeline registers throughout the RISC151V-core code, but also when registers were necessary for the CDC handshake and various other pipelining problems.

## Memories

- **BIOS/IMEM/DMEM/MMIO**
  The memory architecture of the RISC-V CPU is depicted in the figure below. The BIOS is the foundation of the CPU and provides the initial instructions. The IMEM provides a space to flash new programs onto the CPU. The DMEM provides a space to store and load CPU calculated data. And, the MMIO is how the CPU interacts with most peripherals. The BIOS/IMEM/DMEM files were provided in the skeleton code, however MMIO is a new module created to be similar to the three other memory modules except instead of a 2D-register memory space, the MMIO interacts with off-CPU peripherals.

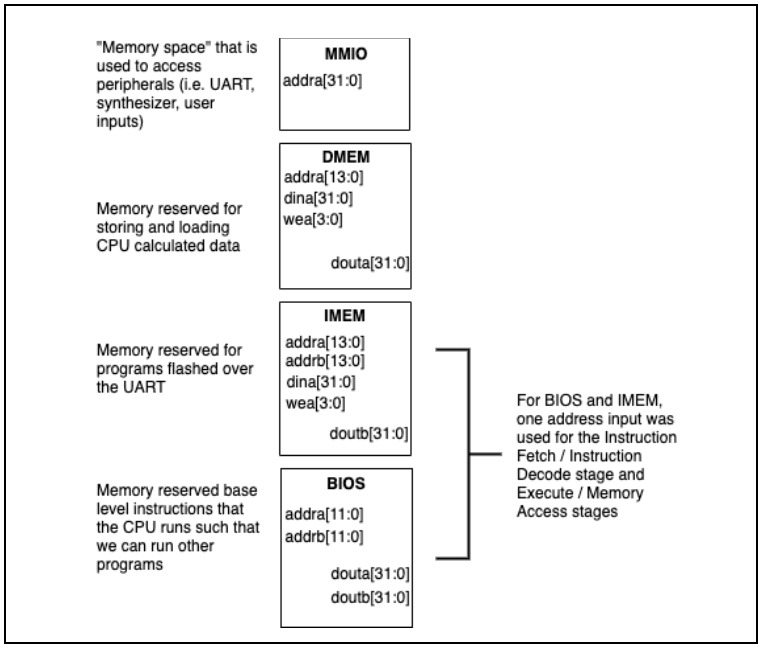


Figure 2: The memory architecture of the CPU.

- **Load Decoder and Store Decoder**
  We created a Load and Store decoder to handle the reading and writing respectively, of differently sized data (word, half-word, and byte) to and from memory. Specifically, this made organizing the spec's write enable mask for store instructions more readable.

# Non-CPU Modules

These modules were a bit harder to define as there was a lot of freedom in the interpretation of the specification. Without skeleton code for many of these files, we made a rough outline of which pieces (and sub-pieces) warranted making their own module and which ones didn't. In general, each block in the block diagrams of each circuit were made into modules which ended up working nicely in cases such as the `FIFO` which were used in both the `UART` pipeline as well as the `Button Parser` pipeline.

## IO Circuits

- **UART**
  The UART is how the off-board computer communicates with the PYNQ-Z1. It implements asynchronous communication through a known baud rate and a predetermined start and stop sequence. It primarily functions as a way for the off-board computer to load programs to the CPU. This is achieved via a `hex_to_serial` script that tells BIOS where to place the instructions, placing each instruction at its appropriate location.

    - **FIFO**
      Buffers inputs (generally key presses) such that rapid inputs can all be registered. This is necessary to prevent dropped inputs from sources such as `hex_to_serial` and keyboard.

- **Button Parser**
  In order to consistently and accurately detect on-chip button presses the signal that the buttons produce must be processed. We have to design input conditioning circuits to handle metastability and button bounce.

    - **Synchronizer**
      Synchronizes the button press signal to the PYNQ-Z1's clock frequency using two shift registers.

    - **Debouncer**
      Filters the synchronous "glitchy" raw button presses that appear to a digital circuit as multiple button presses. In order to deal with this problem we

create the debouncer only registers “saturated” button presses (presses that last a certain number of clock cycles).

- **Edge Detector**
  Filters the debounced button press to output only the posedge and negedge transitions as single clock cycle pulses.

- **FIFO**
  Much like the UART, buffers incoming signal to prevent dropped inputs.

## Audio

- **Synth**
  The synthesizer acts as the parent class to play tones from two sources: (1) a software synth that writes MMIO-defined frequency values directly to the Digital to Analog Converter (DAC) and (2) a hardware synth that uses a Numerically Controlled Oscillator (NCO) that samples values from oscillatory functions through LUTs.

  - **Pulse Width Modulation Digital to Analog Converter (PWM DAC)**
    This circuit maps a 12-bit input to a PWM signal with a duty cycle equal to `input[11:0]/(2^12 - 1)`. This controls the volume of our audio through the PWM signal but also acts as a medium through which the UART can play specific frequencies.

  - **Numerically Controlled Oscillator (NCO)**
    The NCO gives us the option of sampling values directly from common oscillatory functions such as `sin`, `square`, `sawtooth`, and `triangle`. This allows us to input well-defined oscillatory functions into our DAC whereas without it, the DAC would only be able to receive a square wave function.

  - **Clock Domain Crossing (CDC) Handshake**
    The DAC runs at the PWM clock frequency (125MHz) in order to produce a higher resolution output, however our PYNQ-Z1 runs at a much lower clock frequency (50-60MHz). In order to resolve this problem, we use a CDC handshake to ensure the chip signal successfully synchronizes to the PWM clock frequency.

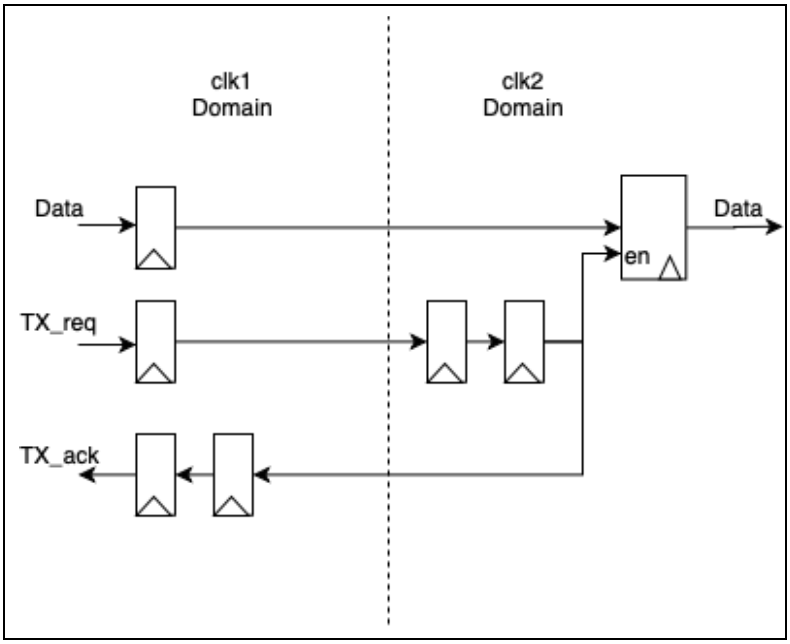


Figure 3: The Clock Domain Crossing circuit.

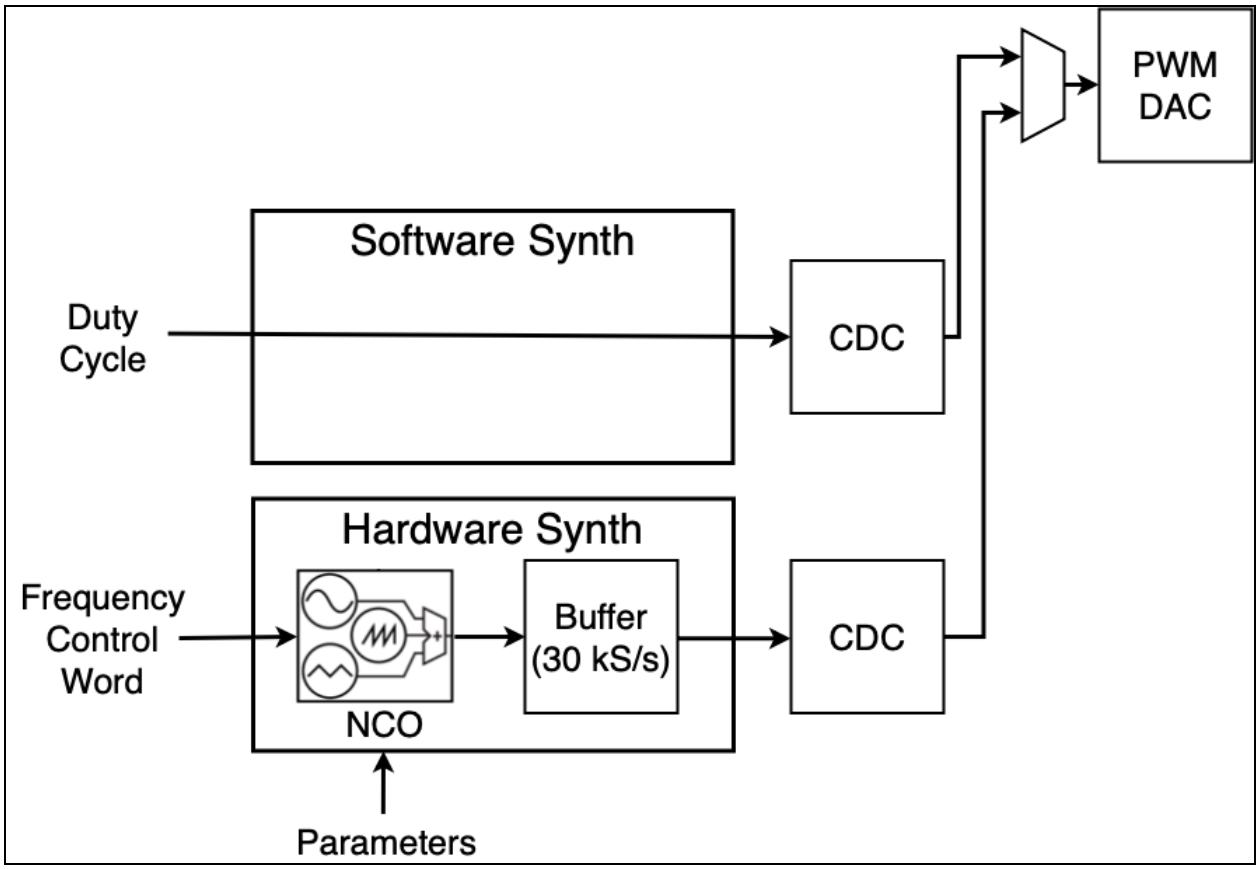


Figure 4: The Synthesizer circuit which incorporates both the Software (off-chip) and Hardware (NCO) synthesizers.

## Status and Results

| | |
|---|---|
| **Total LUTs Used:** | 3668 (6.89%) |
| **Total SLICE registers used:** | 1716 (1.61%) |

The operations described in the spec (assembly testbench, ISA-tests, mmult, user I/O tests, square piano testbench, and NCO testbench) are all fully functional. The mmult command resulted in a CPI of approximately 1.18.

We are able to run at 60 MHz in both simulation and synthesis (on-board). However, pushing the clock frequency any further resulted in setup time violations. Our critical path was a memory to memory path. We realized we could fix it by routing our memory forwarding not directly from the MEM to EX stages, but rather from the MEM to IF/ID stages (and NOPing appropriately), however we were not able to do so within the time constraints of the project.

For the optimization component, we focused our efforts on mitigating unnecessary resource utilization. We did this by combing through our `make synth` logs and refactoring our modules to optimize space and resources according to the suggestions provided by the routing tools. This included reducing wire width to exclude unused bits, deleting unused wires and registers, and ensuring no latches were inferred throughout our code.

While we were able to get everything required in the specifications to work, we were not able to finish working on some of the additional components such as further optimization and synthesizer features. Specifically, we started to look into building the State Variable Filter (SVF) for the synthesizer. Additionally, we started to take a look at a 2 bit dynamic branch predictor, that would keep track of the number of consecutive times we took a branch by keeping track with a counter. This would intuitively decrease our CPI as most programs that take a branch several consecutive times would continue to do so (i.e. for/while loops).

## Conclusions

Working on this project has been an extremely gruelling yet rewarding experience. The massive amount of successes (i.e. modularizing code, commenting code) and failures (i.e. blocking vs. non-blocking, synchronous vs. asynchronous) we encountered through writing Verilog code tested our skills both as hardware designers and software engineers. Furthermore, reading and understanding the base specifications of the project posed many challenges which became opportunities to learn more about communication and interpretation.

While we learned many skills and concepts during this experience, the most valuable lesson learned was familiarizing ourselves with the testing and debugging workflow in Verilog. This included writing assembly and/or C code to send test instructions to our CPU, creating a testbench for every module to test for expected inputs and outputs, and looking at waveforms to check for timing and data inconsistencies in our datapath. These lessons extend beyond the scope of this project and has been an insightful experience of the day-to-day ventures of a systems engineer.

If we could go back and do it all again, the one thing we would change would be to do more extensive unit testing. We assumed that our implementation of the basic blocks (i.e. `imm gen`, `mmio`) would work without needing to be rigorously tested. However, while running integration tests, we realized we had overseen some edge cases that, had we done more unit testing, could have been resolved earlier. Also, if possible, we would go back in time with our current implementation, so that we could spend more time on the equally interesting extra credit portions (i.e. optimization, SVF, ADSR, and polyphonic synthesis) that we were unable to complete due to our timeline constraints.